Universidade de Aveiro Departamento de Eletrónica, Telecomunicações e Informática
2018

Eduardo Miguel Oliveira Duarte

Análise colaborativa de grandes conjuntos de séries temporais

Collaborative analysis over massive time series data sets

Universidade de Aveiro
2018

Departamento de Eletrónica,
Telecomunicações e Informática

Eduardo Miguel Oliveira Duarte

Análise colaborativa de grandes conjuntos de séries temporais

Collaborative analysis over massive time series data sets

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Informática, realizada sob a orientação científica do Doutor Diogo Nuno Pereira Gomes, Professor auxiliar do Departamento de Eletrónica, Telecomunicações e Informática da Universidade de Aveiro.

Para o meu pai, que me inspirou a ser sempre ambicioso e dedicado à minha paixão. Apesar de ele já não estar entre nós, a sua alegria e influência estará sempre presente em mim e na sua família.

Para a minha mãe, que me ajudou nos meus momentos altos e baixos e me concedeu a sua paciência e devoção infinita.

To my father, who inspired me to always be ambitious and dedicated to my passion. Even though he is no longer among us, his joy and influence in me and his family are ever-present.

To my mother, who aided me in my highest and lowest moments and bestowed me with her infinite patience and devotion.

agradecimentos / acknowledgements

Firstly, I would like to thank my supervisor, Professor Diogo Gomes, for his encouragement and assistance throughout the development of this project and the writing of this dissertation. His recommendations and advise were essential in conducting the required research, engineering the solution, and contributing to the overall body of work hereby presented. His insights, sincerity and goodwill undeniably had a huge and positive impact in my personal and professional life.

I would also like to thank Bosch Termotecnologia S.A. and Pedro Ribeiro for giving me a fantastic work opportunity, where I was able to develop my engineering skills and get a small taste for the fascinating innovations in the software world. I thank all the great co-workers I had the pleasure of working with: Alexandre Chambel, Pedro Correia, Tânia Correia, Paulo Costa, António Marques, Rafael Peixinho, José Santos and Pedro Saraiva. I will be forever thankful for everything I have learned from them, and will dearly miss all the lunch breaks playing cards and having fun discussions.

I would like to express a special thanks to David Campos, to whom I am greatly indebted for having an instrumental role in my life throughout both my Bachelor's and my Master's. His guidance was influential in exposing me to the thrilling world of computer science, developing my skills in designing powerful and intuitive applications as well as writing concise and elegant code. His leadership, tenacity and concern for his peers is inspiring, and his care for my work and well-being was paramount in getting me to the finish line. A sincere thank you for taking a chance on me, for your patience and for your understanding.

I send my profound gratitude to the group of colleagues and close friends from Portugal, England and Denmark, who have been present in this path from the very beginning and on a daily basis, indulging me with interesting and hilarious conversations and supporting me through all ups and downs. I also want to thank my brother for his genuine concern and affection for me, as well as for all the fun times we had and will continue to have.

Finally, and most importantly, I wish to thank my mother, Isabel Nascimento, for having a crucial role in educating me and encouraging me to follow my interests and to fulfill my goals. Her unmatched and unconditional support gave me the foundations for me to be where I am today.

The present study was developed in the scope of the Smart Green Homes Project [POCI-01-0247 FEDER-007678], a co-promotion between Bosch Termotecnologia S.A. and the University of Aveiro. It is financed by Portugal 2020, under the Competitiveness and Internationalization Operational Program, and by the European Regional Development Fund.

Palavras Chave

séries temporais, anotações, sistemas de anotação, software colaborativo, análise de dados, ciência da informação, modelagem de dados, gestão do conhecimento, sistemas de gestão de bases de dados, sistemas distribuídos, visualização de informação, AVAC

Resumo

A recente expansão de metrificação diária levou à produção de quantidades massivas de dados, e em muitos casos, estas métricas são úteis para a construção de conhecimento apenas quando vistas como uma sequência de dados ordenada por tempo, o que constitui uma série temporal. Para se encontrar padrões comportamentais significativos em séries temporais, uma grande variedade de software de análise foi desenvolvida. Muitas das soluções existentes utilizam anotações para permitir a curadoria de uma base de conhecimento que é compartilhada entre investigadores em rede. No entanto, estas ferramentas carecem de mecanismos apropriados para lidar com um elevado número de pedidos concorrentes e para armazenar conjuntos massivos de dados e ontologias, assim como também representações apropriadas para dados anotados que são visualmente interpretáveis por seres humanos e exploráveis por sistemas automatizados. O objetivo do trabalho apresentado nesta dissertação é iterar sobre o software de análise de séries temporais existente e construir uma plataforma para a análise colaborativa de grandes conjuntos de séries temporais, utilizando tecnologias estado-de-arte para pesquisar, armazenar e exibir séries temporais e anotações. Um modelo teórico e agnóstico quanto ao domínio foi proposto para permitir a implementação de uma arquitetura distribuída, extensível, segura e de alto desempenho que lida com várias propostas de anotação em simultâneo e evita quaisquer perdas de dados provenientes de contribuições sobrepostas ou alterações não-sancionadas. Os analistas podem compartilhar projetos de anotação com colegas, restringindo um conjunto de colaboradores a uma janela de análise mais pequena e a um catálogo limitado de semântica de anotação. As anotações podem exprimir significado não apenas sobre um intervalo de tempo, mas também sobre um subconjunto das séries que coexistem no mesmo intervalo. Uma nova codificação visual para anotações é proposta, onde as anotações são desenhadas como arcos traçados apenas sobre as curvas de séries afetadas de modo a reduzir o ruído visual. Para além disso, a implementação de um protótipo full-stack com uma interface reativa web foi descrita, seguindo diretamente o modelo de arquitetura e visualização proposto enquanto aplicado ao domínio AVAC. O desempenho do protótipo com diferentes decisões arquiteturais foi avaliado, e a interface foi testada quanto à sua usabilidade. Em geral, o trabalho descrito nesta dissertação contribui com uma abordagem mais versátil, intuitiva e escalável para uma plataforma de anotação sobre séries temporais que simplifica o fluxo de trabalho para a descoberta de conhecimento.

Keywords

time series, annotations, annotation systems, collaborative software, data analysis, information science, data modeling, knowledge management, database management systems, distributed systems, information visualization, HVAC

Abstract

The recent expansion of metrification on a daily basis has led to the production of massive quantities of data, and in many cases, these collected metrics are only useful for knowledge building when seen as a full sequence of data ordered by time, which constitutes a time series. To find and interpret meaningful behavioral patterns in time series, a multitude of analysis software tools have been developed. Many of the existing solutions use annotations to enable the curation of a knowledge base that is shared between a group of researchers over a network. However, these tools also lack appropriate mechanisms to handle a high number of concurrent requests and to properly store massive data sets and ontologies, as well as suitable representations for annotated data that are visually interpretable by humans and explorable by automated systems. The goal of the work presented in this dissertation is to iterate on existing time series analysis software and build a platform for the collaborative analysis of massive time series data sets, leveraging state-of-the-art technologies for querying, storing and displaying time series and annotations. A theoretical and domain-agnostic model was proposed to enable the implementation of a distributed, extensible, secure and high-performant architecture that handles various annotation proposals in simultaneous and avoids any data loss from overlapping contributions or unsanctioned changes. Analysts can share annotation projects with peers, restricting a set of collaborators to a smaller scope of analysis and to a limited catalog of annotation semantics. Annotations can express meaning not only over a segment of time, but also over a subset of the series that coexist in the same segment. A novel visual encoding for annotations is proposed, where annotations are rendered as arcs traced only over the affected series' curves in order to reduce visual clutter. Moreover, the implementation of a full-stack prototype with a reactive web interface was described, directly following the proposed architectural and visualization model while applied to the HVAC domain. The performance of the prototype under different architectural approaches was benchmarked, and the interface was tested in its usability. Overall, the work described in this dissertation contributes with a more versatile, intuitive and scalable time series annotation platform that streamlines the knowledge-discovery workflow.

Contents

List of Figures

List of Tables

Acronyms

ACID	Atomicity, Consistency, Isolation and Durability
API	Application Programming Interface
CAP	Consistency, Availability and Partition tolerance
C+A	Consistency + Availability
CDBMS	Column-oriented Database Management System
CDN	Content Delivery Network
CRUD	Create, Read, Update and Delete
CSS	Cascading Style Sheets
DDL	Data Definition Language
DOM	Document Object Model
DSS	Decision Support System
ECG	Electrocardiogram
E+C	Consistency + Availability over Low-Latency
EEG	Electroencephalogram
E+L	Availability + Low-Latency over Consistency
FIFO	First-In-First-Out
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation and Air-Conditioning
IoT	Internet of Things
JDBC	Java Database Connectivity
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSON	JavaScript Object Notation
JWT	JSON Web Token
LDAP	Lightweight Directory Access Protocol
LESS	Leaner Style Sheets
LIFO	Last-In-First-Out
LRU	Least-Recently-Used
LSM	Log-structured Merge
MVCC	Multiversion concurrency control
ORM	Object-relational mapping
OWL	Web Ontology Language
RAM	Random-access memory
RDBMS	Relational Database Management System
RDF	Resource Description Framework
REST	Representational State Transfer
SMTP	Simple Mail Transfer Protocol
SoC	Separation of Concerns
TCP	Transmission Control Protocol
TSDBMS	Time Series Database Management System
UI	User Interface
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
UUID	Universally Unique Identifier
UX	User Experience
XML	Extensible Markup Language

CHAPTER 1

Introduction

As we progress further into the modern age of knowledge-oriented digital innovation, the requirements for digital data processing and storage keep increasing at an exponential rate. The quantity of digital data being generated and stored in systems is estimated to be approximately 4.4 zettabytes [1]. While in 2016 the annual rate of data traffic was at 1.2 zettabytes per year, it is projected that this rate will increase to 3.3 zettabytes per year by 2021 [2].

One of the major growth spurts that has led to this large increase in data consumption is the increased metrification and remote control of internet-connected devices like smartphones, wearable gadgets, real-time sensors, Heating, Ventilation and Air-Conditioning (HVAC) boilers, Smart Home devices and other equipment composing the Internet of Things (IoT). These devices can generate events and metrics from a variety of spaces in the physical world, from home assistant applications for small households to large distributed platforms administering entire cities [1]. These systems, capable of collecting data and providing means for remote management at a massive scale, have been deployed in a wide range of domains, like scientific research, medical diagnosis, smart cities, among others. More recently, IoT technologies have entered the industrial and enterprise domains so as to enable intensive metering of manufacturing processes and other business operations [2].

While data has grown at an alarming rate, increasing in the three *Vs*, Volume, Variety and Velocity [3], most business solutions collect more data than they can process to produce the fourth and fifth *Vs*, Value and Veracity. To generate meaning out of large collections of data, systems for analysis, automation, monitoring and anomaly detection have been developed and deployed. Analysts can easily explore and refine these data sets under a myriad of criteria, leading to the discovery of new segments of knowledge and relationships. Some analytical tasks have also been allocated to automated processes, expediting the knowledge discovery. For example, in the medical domain the analysis performed over streams of real-time data has enabled the further understanding of human ailments [4].

[1] `https://www.emc.com/leadership/digital-universe/2012iview/index.htm`

[2] `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.html`

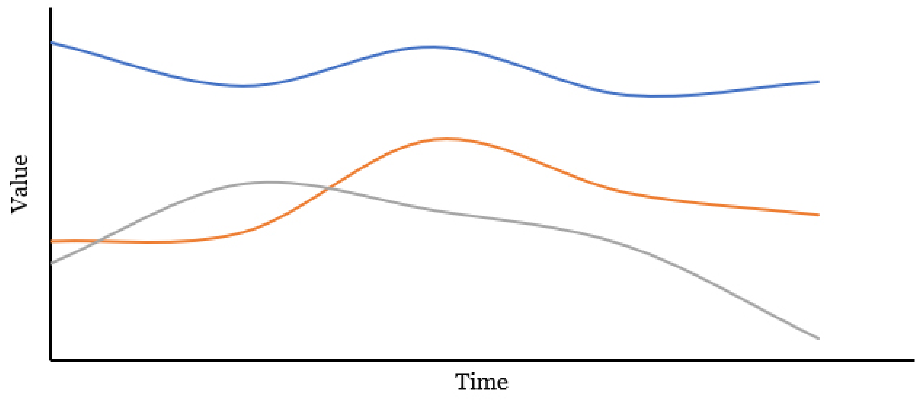


Figure 1.1: Three time series represented in a line graph visualization.

For most of the mentioned use cases, the collected data is only useful if the various metric events are logged and presented in an orderly manner, as some of the most important observations are made not to a single data point but to sets of data points that constitute a pattern over time. The overall set of sorted points essentially constitutes a time series.

1.1 Time series analysis

Time series are commonly described as a sequence of ordered numeric values that correspond to observations of an entity over discrete, non-uniform time intervals [5]. A common notation used to formally specify a single time series data point that represents a measure of state is to structure it as a tuple of two elements (i.e. a 2-tuple) or a key-value pair, and expressed as (t, v), where t corresponds to a valid time indicator of when a value v was measured. As a key-value pair, every t is unique throughout an entire series, whereas values can be equal. In other words, an entire time series can be expressed as in (1.1), where every t is a unique time indicator and n is the maximum count of data points in TS.

$$TS = (t_1, v_1), (t_2, v_2), ..., (t_n, v_n) \tag{1.1}$$

These data types are a natural part of data sets that contain massive amounts of events over time, as the continuous nature of time series make it particularly suitable to represent these events for monitoring and analysis purposes [6]. In fact, time series can be found in almost every aspect of human life [7], like medical diagnosis based on Electroencephalograms (EEGs) [4], [5], [8]–[12] and Electrocardiograms (ECGs) [13], financial technical analysis [14]–[16], measurement of meteorological or natural occurrences like earthquakes [17], monitoring of athlete's performance and total daily energy expenditure [18], and even telemetry for satellites and aerospace ships [19].

The act of analysis can be defined by the process of deriving and inferring meaning, concepts and ontologies from raw data, breaking large sequences of data into smaller, more understandable patterns. As the pertinent data in the platform grows, collaborating users have a higher chance of growing a mutual data warehouse of useful knowledge. Furthermore, expert knowledge in itself should be used recurrently to infer new segments of knowledge, making the act of analysis more efficient [5].

A common use case for analysis of time series is to enable the development of a Decision Support System (DSS). These systems allow analysts and other observing users to deduce a set of actions that have to be done outside of the analysis platform for a given purpose, like technical maintenance of devices or medical diagnosis and treatment of patients. In [20] and [21], DSSs were deployed in intensive care units, enabling the medical staff to analyze neurophysiological data, perform diagnosis and prognosis based on historical patterns, and carry out subsequent treatment, which is a time-critical process with a limited window of opportunity.

However, the previously mentioned domains typically produce high amounts of data due to sensors that rapidly send state changes over short intervals of time. This can often lead to data entropy and visual pollution that increase the complexity and difficulty of analytical tasks, as well as demand higher development and financial costs. One of the main challenges with critical DSSs, as well as general-purpose time series analysis platforms, is to handle massive data sets at both an architectural level and at a visual level. Techniques must be developed to read massive amounts of data and extrapolate patterns and trends at a reasonable speed, as well as to visualize these in a intuitive manner [19].

1.2 Visualization of time series

Within the domain of Information Visualization, time series are among the most common type of data explored in quantitative graphical perception studies. A core aspect of time series is the fact that the most useful information is frequently observed in a sequence of data points instead of individual ones. Hence, an integral part of the analysis process is to leverage the human ability to visually identify trends, anomalies, correlations and patterns [22].

For this, line graphs, as shown in Figure 1.1, have been the visualization of choice for temporal data ever since its initial proposal by William Playfair in 1786 [23]. However, in highly heterogeneous use cases such as the analysis over massive quantities of sensor data, there are various time series that tendentially need to be visualized simultaneously, so as to observe how different metrics affect and depend on others [24]. This requirement quickly exposes the issues of the common line graph, as data becomes harder to parse visually when a high number of time series fill up too much of the available visual space or compete with each other by tracing the same trajectories.

In a shared-space model, where multiple time series co-exist in the same chart, as the number of simultaneous time series on display increases, the chart is only readable up to a perceptual threshold. This issue is accentuated when all series follows a very limited color-scheme, compounded by the limited color acuity of the human visual system, which can lead to various time series being colored similarly and becoming easily misinterpreted. In a split-space model, where multiple time series are displayed in separate charts, as the number of charts increases, it becomes exponentially harder to visually compare how series behave within the same segment of time. Plus, the amount of visual space required to display every individual chart can exceeded the maximum available space within a limited window of perception. In sum, both space distribution models have perceptual thresholds directly correlated with the number of time series on display [25], and the relationship between the two corresponds to a trade-off between visual clutter and usage of visual span [26].

Time series visualization can be a very challenging task, as temporal data from realistic scenarios is often complex with high cardinality [27]. To build visual overviews of this type of data, visualization tools have to be developed and designed to integrate appropriate graphical schemes and features such as filtering and summarization [26]. This summarization, or clustering, technique can be a very valuable tool that helps analysts understand massive time series data sets by observing representative clusters and their relationships [28].

1.3 Annotations over time series

While time series in themselves are highly capable of representing a measurement over time, exposing fluctuations, spikes and drops, their meaning and semantic is not directly conveyed. A human observer can infer meaning from metric changes in relation to the universe the data lives in, and a machine can be guided by a curator to interpret metrics and make those same semantic associations, but the data itself does not communicate that semantic explicitly. Because of this, human or machine observers are incapable of conveying these interpreted semantics through the data alone, and require the assistance of appended metadata to create this unit of knowledge. This concept of data annotation has been applied in a wide scope of domains, both traditional and digitally-oriented.

Annotations are traditionally used in the process of critiquing and updating documents or segments of data [29]. Catherine Marshall [30] expressed that “an annotation of data, be it paper-based written material or digital data, represents an engagement with the material”. In this case, this user-specified engagement can be represented as a note for memory aid, an highlight, a description of a taxonomy, or a warning notice for other readers. In all these cases, the annotation acts as a layer over the original document or data set [29], identifying a single point or an interval of data as relevant to a specified context.

Digital annotations, operating over data presented in any given software tool, have been initially designed to correspond directly to their paper-based counterparts [31]–[36], adapting to existing activities and use cases in medical and technical fields. Kawase et al. [37] evaluated the performance of digital annotation for these specific activities in a web-based prototype, and concluded that digital annotations provide more advantages than paper-based schemes by leveraging capabilities inherent to computer solutions. In general, studies on the subject of digital annotations outline the following features of digital annotations over the traditional, physical approach:

- **editing of contents**: annotations are persisted in information systems and can be changed over time to correct factual or grammatical errors made at the point of conception. Changes can be historically recorded as a *version* of that annotation, so users can progressively iterate on new versions and at any point roll-back to a previous point in time without losing data [12];
- **search over contents**: annotations, when stored as free-text, can be indexed and searched at a later time. Users can take advantage of this feature to quickly find data associated with a particular meaning;
- **search over segments**: users can highlight a segment of data to identify a pattern and quickly find it again by describing the required pattern;
- **relationship discovery**: when a similar annotation acts over separate segments of data, the digital system can quickly find this relation and expose these to the user in a more expedited manner. Similarity search techniques can also be used to automatically detect similar patterns across the entire document or data set.
- **communication layer with other users**: by expressing a notice or a highlight, other users that participate in the observation of the data set are aided by the annotations. Users can cooperate knowledge building by editing annotations of others with critiques or by appending complementary information as annotations over the same segment of data;
- **communication layer with machines**: by attaching metadata to segments of data in a structured manner, an automated task can interpret this and be trained further in how to interpret the remainder of the data [38]. This is useful for supervised learning tasks where a human curator guides the annotation task so that a machine-learning model can become an active participant in the construction of knowledge;
- **enforcing of read-write access policies**: annotations and discussions can have selective read-write permissions based on the user's roles and authorities in the hierarchical tree of collaborators [12], preventing users from making changes to annotations outside of their scope of work;
- **circumvention of rigid records**: since annotations act as attachments that do not affect the original data nor its structure requirements (e.g. medical databases and financial data warehouses), then these can be used to express additional data that was not planned by the original architects of the data schema [12]

In this manner, an annotation can be expressed as a materialization of knowledge on the original data that is shared between partners. Each user is given the possibility to observe and propose meaning, and other users can then discuss that meaning, classify it in terms of degree of usefulness or confidence, and evolve it until an agreement between collaborators is reached. Based on this, there is sufficient evidence available that shows that the usage of annotations in collaborative analysis platforms enables a more adequate knowledge-building and decision-making process [39], and that this is made more flexible and simplified through digital means. Therefore, the process of effective collaborative exploration and learning of time series data should involve annotation tasks [38].

1.4 Distributed systems

Analysis and annotation software solutions can be made available to a set of users in many ways. The system could be deployed locally in a single machine, for example as a desktop application, where multiple analysts agree to store and share knowledge within this unique instance [40]. However, this approach is very limited: 1) only one analyst would be able to interact with the analysis application at a time, so contributions and refinement would happen in a sequential, one-by-one manner; 2) analysts could not monitor the data remotely and would have to be present in the same geographical location as the machine, which is not always feasible since requesting the physical presence of analysts has associated transportation and financial costs [10], [12]; and 3) mechanisms for storing, archiving and backing-up data would have to be manually assured by the users of this system, as a failure of this single machine could imply the loss of all recorded time series data and annotations. A way to overcome this would be to establish a protocol for recording the application's data store and replicating it to other machines, but without a direct message-passing bridge or network between machines, there is no flexible way to synchronize this data store when changes are made by different users in separate machines. Even if multiple machines were connected through a network, a decentralized data store means that simultaneous changes can lead to update conflicts and collisions, so mechanisms for non-destructive synchronization of data changes to a single converged state would have to be implemented manually.

With this, a centralized system, remotely accessible inside a network, presents itself as the most efficient candidate for handling simultaneous analysis and annotation tasks. In theory, the software solution could now be composed of multiple internal units of data processing and storage deployed in a single network-accessible machine. These units would then transfer data between each other over memory, which corresponds to a direct data-channel that is absent of network latency. At a small scale, simple hardware upgrades for single nodes is more cost-effective and performant than distributing the system to multiple nodes. However, with the steady increase in volume and acquisition rate of events relevant to the system's needs, as well as the increase of concurrent users, the hardware resource usage applied to this node is drastically increased. After a certain threshold of hardware requirements, the most cost-effective solution is to propagate over to multiple inter-connected machines [41]. Therefore, distributing and load balancing an analysis system across multiple machines becomes a fundamental requirement when dealing with storage and querying of time series and annotation data at a massive scale [2]. Distributed computing concepts and theorems are at the heart of every solution that require a scalable centralized agreement and communication protocol between multiple persons connected over a network.

An important property to measure efficiency of a distributed system is *Consistency* (or *Linearizability*), which corresponds to the average amount of time a system is maintaining a synchronized view of a continuously-growing data set for multiple users at an acceptable response time and for the majority of its life span [42]. While the system settles on a synchronized state, other simultaneous queries can return stale data for a variable period of time, known as the *inconsistency window.* In other words, a high *consistency* system should be capable of handling a multitude of requests while keeping the *inconsistency window* as small as possible.

Another important property is *Availability*, which describes the average amount of time a system is up and accessible by clients. The amount of downtime is reduced by implementing fault tolerance mechanisms, where the system compensates for possible failures. These mechanisms can range from automatically retrying failed requests or data ingestion operations without losses, to imposing redundancy and replicating data into multiple nodes so that a failure of a node does not make the data inaccessible, to partitioning data and increasing the number of nodes required to fail until *availability* is lost. Both replication and partitioning are useful techniques for avoiding bottlenecks and slow computation, but have associated financial costs. Moreover, replicated data has to be kept in sync across multiple machines, so increased replication can make *consistency* harder to maintain [43]. A lack of *consistency* does not affect *availability*, since even highly *available* systems that partition data can succeed to deliver any results, even though these results might not be the expected records if a specific partition fails.

The concept of latency that is most relevant to distributed system architects is one of system latency, which can be expressed as the total amount of time t_1 that a system takes to receive, process and return reliable results to a client minus the unavoidable amount of time t_2 that a network of connected machines can take to send requests between the system and the client. The latter delay, known as network latency, is inherent to hardware limitations and to transport protocols, and hence, nondeterministic. The system architect can attempt to minimize network latency by scaling the solution to multiple geographic locations or by taking advantage of a Content Delivery Network (CDN) [44].

The Consistency, Availability and Partition tolerance (CAP) theorem [45] is an essential theorem in distributed systems that outlines how every system can only guarantee two out of three properties, *Consistency*, *Availability* and *Partition Tolerance.* An extension to the CAP theorem, the PACELC theorem [46], suggests that when in the absence of network partitioning there is another trade-off between *low-latency* and *consistency.* Distributed systems tend to be designed to expedite results at a fast rate using data partitioning and replication approaches, which means that the software architect must choose between a Consistency + Availability over Low-Latency (E+C) architecture, which ensures that writes are propagated appropriately and consistently throughout the entire system, or a Availability + Low-Latency over Consistency (E+L) architecture, which quickly delivers the currently available snapshot even if a simultaneous write has not yet been fully propagated.

For a distributed platform that targets the timely analysis and monitoring of critical data, and depending on the nature of this data, observers might either require the most recent metrics of data, so as to take swift action external to the platform, or a more synchronized delivery of historical metrics. In the clinical domain, the focus is more often than not on the monitoring of the most recent data. For example, in neurophysiological monitoring, a delay in providing the recent measures can have critical consequences in the diagnosis and treatment of patients [5]. Two separate use cases that might be a part of the desired workflow in any given analysis system can be outlined, which correspond directly to the *consistency* and *low-latency* trade-off suggested in the PACELC theorem:

- focus on E+C: the analyst role is to discover alarming events as close as possible to the moment they are measured, prioritizing strong writes into the platform and synchronizing reads to catch the latest updates more consistently;
- focus on E+L: the analyst role is to go over historical data instead of the latest updates of any metric, prioritizing fast reads.

1.5 Contributions

The work presented in this dissertation is intended to contribute to the communities of time series analysis, information visualization of annotations, and distributed systems. The model (and consequent software prototype) proposed in this document iterates on the aspects developed in existing analysis platforms by integrating various features and techniques found in other domains such as Information Visualization and Distributed Systems. The end goal for the proposed model is to outline a feature-complete time series analysis web platform that handles common use cases for knowledge discovery, reviewing and sharing in an optimized manner. In order to build such a collaborative tool, the following requirements are outlined:

- **User management**: multiple users must be authenticated and be allowed to use the platform concurrently. Every user has an associated authority over a set of annotations, acting as the direct author of these or a curator that reviews and discusses their practicality. Every action made within the platform should always be validated and secured;
- **Annotations**: users in the platform introduce and share knowledge over the observed data in a controlled manner, as each annotation requires the review and approval of partners. Users working on the same domain of data can discuss the degree of suitability for each new annotation and refine it. Each step of progress for every annotation should be recorded historically so users can track how it evolves over time and rollback undesirable changes;
- **User tasks**: the platform should be autonomous in determining whenever certain actions are required from users, especially when related to responsibilities over a set of annotated data the users are affiliated with. When new annotations are added to a specific domain of data and knowledge and require reviewing, a curator of that domain should be notified and prompted to do so. If at any point during the life span of the platform a set of time series is inaccurate in relation to what the original measurement should be, or lost in any way due to inadvertent data corruption, the annotations that were attached to these segments should still be persisted, and the users responsible for these annotations should be notified to adjust the annotations to the changes in the data universe by manually relocating these or deleting these in case their meaning is no longer applicable;

- **Flexible pattern detection**: while the usage of similarity measures for time series pattern detection and prediction is outside of the scope of this dissertation, this particular feature can be empowered by the existence of annotations. If the user annotates a specific pattern using a fixed catalog that is universal to all annotation tasks, then the platform can be autonomous in predicting that other similar patterns of data should also be annotated with the same group, category or semantic. Users can be notified to verify the integrity and appropriateness of a set of automated annotations, interacting in a human–computer collaborative approach with the platform [38].

Along with the above collaborative features, another essential contribution of this dissertation is to leverage what the author considers to be, at the time of writing, the most suitable open-source tools and techniques for querying, storing and displaying time series, annotations, user sessions and versioned data that are standard practice in the commercial software industry, all of this under a distributed architecture with polyglot persistence. This includes the usage of modern frameworks for backend and frontend development and deployment, as well as user-facing visualizations that take modern User Experience (UX) and User Interface (UI) design fundamentals into consideration. Many academic documents already present benchmarks for existing storage technologies with time series data [2], [47]–[51], and this dissertation will operate under the assumptions and conclusions taken from these works in order to build optimal structures and architectures for storing massive data sets.

In state-of-the-art time series annotation applications, the chosen visual encoding for annotations matches their data model, as these can only relate to a segment of time, but it is a limited visual encoding when annotations are linked with a subset of the series within that segment. A final important contribution this dissertation proposes to the time series visualization academic community is the usage of an alternative encoding for annotations over time series that takes the affected series into consideration, requiring a considerably smaller amount of visual space and decreasing the complexity of perceptual tasks.

Throughout this dissertation, a massive data set is quantified as at least a dozen terabytes or more of time series when stored digitally as uncompressed TSV files, commonly reaching one million data points. However, this classification should be considered a moving target, so that as more and more series are collected from multiple data sources over a long span of time, the platform should scale accordingly. While the time series data sets that are ingested by the implemented prototype were specific to the HVAC domain, the frontend interface, backend and architecture modules should be capable of handling time series, users, annotations and other metadata in a generic manner, supporting any domain other than HVAC.

1.6 Dissertation outline

The outline of the rest of this dissertation is as follows:

- Chapter 2 presents an in-depth overview of state of the art methodologies and technologies currently applied to analysis, storage and visualization of time series and annotations;
- Chapter 3 proposes a blueprint for the development of a time series analysis and annotation platform at the theoretical level, describing an ideal architecture and visualization tool for handling both time series and ontology data;
- Chapter 4 iterates on the blueprint proposed in Chapter 3 by implementing and describing a working prototype for a collaborative time series analysis architecture and web application, listing the specific tools that were employed for it and the techniques that allowed further optimized usage of state of the art technologies to handle the mentioned requirements;
- Chapter 5 takes the implemented platform and benchmarks its features, in order to evaluate how its architecture handles realistic scenarios and how it compares with other potential architectures, as well as how its interface adheres to interaction design standards;
- Chapter 6 gives an account of the observed behaviors and caveats in the prototype during development and evaluation phases, relates the ways in which the proposed model for time series annotation improves on existing tools, and leaves a few clues to how the proposed platform can be iterated on in order to extend its capabilities and improve its overall performance and quality.

CHAPTER 2

State of the art

2.1 Time series analysis systems

There are multiple research papers and academic documents that outline a system for collaborative exploration of multivariate time series data. All of the studied analysis systems are designated for a specific universe of data, the majority being for psychological data streams derived from EEG terminals [4], [5], [8]–[12], as intensive care units are environments that typically produce a high number of rich and heterogeneous streams of temporal metrics [8]. A few others focus on biomedical and health-care time series data streams, processing metrics like heart-rate, temperature, oxygen saturation, blood pressure and glucose from ECGs [13]. There is also some literature dedicated to the analysis, pattern-detection and prediction of financial time series data [14], which presents its own set of unique challenges due to its high-frequency yet volatile nature that embodies the characteristics of the financial market.

The Artemis platform [4] is a fully-fledged server and web application that presents patient data collected from physiological data streams. The stream data is stored long-term for retrospective analysis and data mining purposes. Other solutions implemented a similar web platform with the goal of solving similar issues like analytics, data mining and decision support for diagnosis based on physiological patient data waiting for care in the emergency department [9], [11], [52]. The main goal of these is to reduce delays and financial costs that can arise when offsite expert analysis is required. However, none of these papers mention the usage of annotations for knowledge building and discovery, and the ones that do represent a smaller subset of the state of the art for analysis systems, even though there is sufficient evidence available in academia that shows that the usage of annotations in collaborative platforms for data analysis bring a wide number of benefits associated with a more adequate interpretation of data and a more responsive decision-making [39].

In [12], [53] and [5], the respective authors implement systems that, similarly to the previously mentioned solutions, store data from real-time physiological streams and present it in a web-based analytics application. In addition, these systems support annotations made by analysts at remote locations and by EEGs devices that automatically annotate when an impedance test is performed or a loose lead is detected. Guyet et al. [38] proposed in 2007 a human-computer collaborative approach for exploring biomedical time series, where clinicians cooperate with an automated system in the process of knowledge exploration. In the proposed system a human curator would indicate segments of interest in the time series data set by annotating it, prompting the system to automatically extract similarly significant patterns. The human curator would then manually observe these patterns and determine their adequacy. Finally, Lee et al. [18] proposed a framework that gathers, stores and interprets large and inhomogeneous time series data sets from wearable sport trackers, which include accelerometers for running and inertial sensors for swimming. The data arriving from sensors was modified at a stage prior to storage to ensure that each temporal event contains additional searchable metadata, similar to how annotations are used in previously mentioned solutions.

2.2 Data model

This section develops on how time series and annotations are commonly represented at theoretical and practical levels, and is split into two subsections: the first subsection describes how time series are represented in software systems; and the second subsection specifies how annotations, users, and other entities are modeled in existing time series annotation platforms.

2.2.1 Time series

The data model of time series in the state of the art is faithful to its formal notation (1.1), where each data point from the series contains a unique time indicator and a numerical value. The time indicator is traditionally represented as a *date*, which expresses days as the minimum unit of time, a *time*, which expresses hours as the maximum unit of time and can contain time up to a nanosecond precision, or a *timestamp*, which combines both *date* and *time* data types.

Timestamps use the universal representation defined either by the ISO-8601[1] or the UNIX Epoch standards. The ISO-8601 is a human-readable and machine-interpretable representation of the string type. Each string explicitly contains date and time up to a microsecond precision. To prevent inconsistencies related to managing times from multiple regions across the globe, where there are time differences derived from the various timezones, a common practice in the industry is to convert every timestamp to a single consistent time zone, usually the Coordinated Universal Time (UTC). Then, when sending a timestamp to multiple clients over a global region, this is converted to the each of the client's time zone. The UNIX Epoch standard corresponds to the counting of seconds since January 1st 1970, represented as a simple integer. While this standard is not human-readable and not appropriate for logging or temporal queries, its compact nature and numeric signature translates into smaller storage requirements and transfer payloads (e.g. data streams between sensors and a central server). An epoch integer is consistent across the globe, as the counting of seconds does not vary by time zone. An aspect that requires special attention when using this standard is that this value should always be stored and interpreted as a 64-bit integer. Storing or reading

[1]`https://www.iso.org/iso-8601-date-and-time-format.html`

it as a 32-bit integer at any point in the system means that this value will overflow after January 19th 2038, resulting in serious consequences to the stability of the platform. Due to the necessity for human exploration of time series data to extract meaningful information, most solutions proposed in existing academic literature use the ISO-8601 standard as a more suitable representation for querying.

2.2.2 Annotations

There are architectural challenges associated with "how" the annotations should be stored, and how should the relation between annotations and time series data be represented internally. Storing annotations in an unstructured plain-text format over an inverted index allows these to be highly readable by humans and searchable by any keyword in a free-form manner, using text-mining techniques for processing and for recognizing concepts from text. However, this approach lacks the support for filtering and aggregation over a consistent structure. It is not possible to establish a guideline on how to parse a consistent set of properties in free-text fields, as these fields are not guaranteed to follow a predictable structure. In order to allow automated systems to easily explore the stored data, the internal representation of annotations should be modeled in a way that is both readable by human observers and interpretable by machines for indexing, searching and data mining purposes.

Within the domain of time series data analysis platforms there have been multiple approaches to standardize the annotation structure, but these tendentially end up following proprietary custom schemas that are advantageous for the corpus and use cases that each platform deals with, and not assuredly flexible for corpora in other contexts. This issue is made even more complex when annotation curators are involved, as the logic of authors and their roles, which is impractical to generalize and standardize, would have to be taken into account by the schema.

For the clinical domain applied to psychological data, there has been similar attempts to group annotations into distinct types [5], [12]:

- **waveform classifications** - a categorization over a single point or a segment of time series data, interpreted directly from the original data;
- **events** - an occurrence or observation originated from a context outside of the data

Other papers proposed domain-agnostic ways of representing annotations by using semantic web notations and ontologies [54]–[56]. The usage of Resource Description Framework (RDF) standards for data exchange and inference processing, as well as ontology languages like Web Ontology Language (OWL), allow annotations to provide contextual cues for the study of correlations between different time series and for the inference of additional knowledge based on ontologies [57]. Using a RDF schema notation means that annotations are both human-readable and machine-interpretable, cross-domain and cross-language. A major issue with RDF in its current state is the lack of flexible and consistent syntax and data models that follow the RDF notation, as well as mature frameworks and tools that interpret and query these at a massive scale and under a distributed architecture.

Annotations have, by nature, large heterogeneity of content that varies widely per domain, making it challenging to conceive a conceptual model of annotations where searching, sharing and versioning is adaptable and domain-agnostic. Thus, it is difficult to answer "how" annotations should be modeled in a detailed manner that can be adaptable to any domain, but it is possible to streamline the data structure of annotations to its most reduced form, containing only the minimum number of common fields that are usable in any domain, which then the architects of each platform can extend and adjust to the corpora of the platform in question. The core benefits of enforcing a set of common fields over a free-text format are:

- **searchability**: if all annotations contain the same fields, for example, category, priority-level, expiration and non-validity conditions, then users can execute conditional queries over these fields with the absolute certainty that every stored annotation has these fields populated with relevant content;
- **user relation**: every annotation can include which users of the analysis platform are related to the annotation, either as original creators or collaborators, enabling a user access-control policy that fine-grained at the annotation level [58];
- **flexible versioning**: the expansion of annotation content into multiple fields ease the process of tracking changes over a set of historical snapshots. Understanding how annotations evolve over time, and determining the differences made between two or more snapshots, becomes a simple process of comparing contents of the same field. If annotations were stored as free-text, determining data changes may require text mining techniques for concept recognition and extraction, along with similarity search algorithms that calculate the distance between the two texts. If an annotation that only contains a category or classification were to be compared with another annotation that only contains unstructured notes could lead to low precision and low recall, showing the user a high number of false-positives and false-negatives.

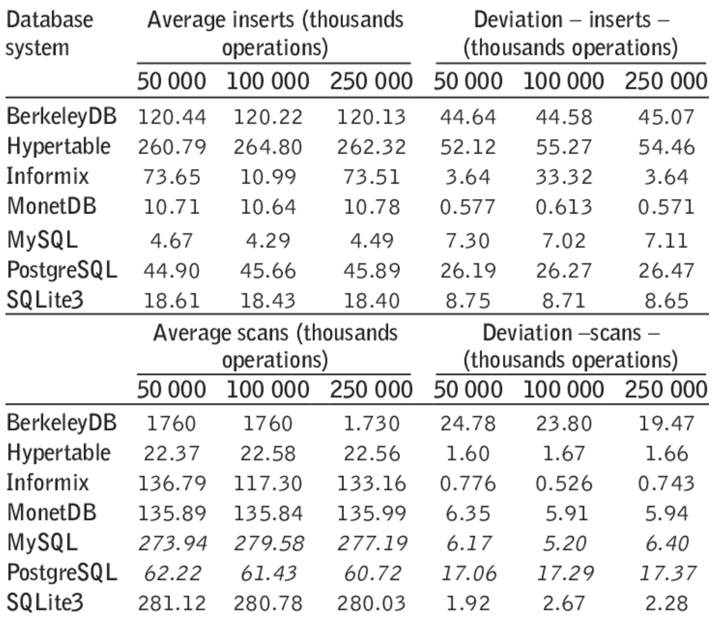
2.3 Data management

This section serves as a survey over specific technologies for efficient acquisition, storage and querying of time series data, annotations and collaborators, employed by existing time series analysis tools found in both academic and technical documentation. It is split into three subsections: the first subsection performs a survey over technologies used for storing time series, in order to determine which technologies are the most appropriate for the established requirements; the second subsection performs a similar technological survey, but over tools and databases that are the most suitable for storing annotations and users; and the third subsection describes various techniques and models for storing versions of entities, evaluating how these models perform when implemented in Relational Database Management Systems (RDBMSs).

2.3.1 Time series

Time series in RDBMSs

The time series analysis systems mentioned in previous sections either leave storage logic unexplained or implement it with a single relational database node. The usage of a RDBMS can enable the optimal storage and indexing of a wide variety of data types (e.g geospatial locations), as well as the modeling of a strong relational schema. However, the question of



Database system	Average inserts (thousands operations)			Deviation – inserts – (thousands operations)		
	50 000	100 000	250 000	50 000	100 000	250 000
BerkeleyDB	120.44	120.22	120.13	44.64	44.58	45.07
Hypertable	260.79	264.80	262.32	52.12	55.27	54.46
Informix	73.65	10.99	73.51	3.64	33.32	3.64
MonetDB	10.71	10.64	10.78	0.577	0.613	0.571
MySQL	4.67	4.29	4.49	7.30	7.02	7.11
PostgreSQL	44.90	45.66	45.89	26.19	26.27	26.47
SQLite3	18.61	18.43	18.40	8.75	8.71	8.65
	Average scans (thousands operations)			Deviation –scans – (thousands operations)		
	50 000	100 000	250 000	50 000	100 000	250 000
BerkeleyDB	1760	1760	1.730	24.78	23.80	19.47
Hypertable	22.37	22.58	22.56	1.60	1.67	1.66
Informix	136.79	117.30	133.16	0.776	0.526	0.743
MonetDB	135.89	135.84	135.99	6.35	5.91	5.94
MySQL	*273.94*	*279.58*	*277.19*	*6.17*	*5.20*	*6.40*
PostgreSQL	*62.22*	*61.43*	*60.72*	*17.06*	*17.29*	*17.37*
SQLite3	281.12	280.78	280.03	1.92	2.67	2.28

Figure 2.1: Benchmark results in [48]: TSDBMSs performance with large-scale data sets, insertion above and reads below.

wherever these RDBMSs were the most effective choice for storing time series when compared with other recent technologies is left unanswered.

The benchmark in [48] was one of the first attempts at answering this, comparing the performance of various databases with sensor data at a massive scale of approximately 10 million time series data points. The compared relational data stores were Oracle BerkeleyDB[2], IBM Informix[3], MySQL[4], PostgreSQL[5] and SQLite[6]. The authors concluded that when queries are made solely to a timestamp, Informix, which offer better support for time-indexed data natively, had the highest performance. However, as more independent data columns other than timestamp keys needed to be fetched, the more its performance decreased, making the other tested solutions more ideal.

Chourasia et al. [17] implemented an analysis system that catalogs millions of earthquake events, patches and actions in a SQLite database. This database performed well for the use case, where data is rarely modified but read repeatedly for visualization and analysis purposes, and where the main user-base corresponds to scientists in the geological domain that will have to set up and handle SQLite files in their remote work locations.

While many RDBMSs such as PostgreSQL already support temporal segments, these do not integrate temporal data rollup procedures that sort and summarize time series under data views with decreased detail. These batch processing procedures would allow queries to be made to the data set without knowing the range ahead of time and while avoiding the need to scan a high amount of records. There has been a recent surge in new time

[2]`https://www.oracle.com/database/berkeley-db/index.html`
[3]`https://www.ibm.com/analytics/informix`
[4]`https://www.mysql.com/`
[5]`https://www.postgresql.org/`
[6]`https://www.sqlite.org/index.html`

series databases built with RDBMSs as a backend, implementing these rollup mechanisms while inheriting their flexible data model, their battle-tested reliability and performance, and their ecosystem of open-source extensions. One such database is TimescaleDB[7], which uses PostgreSQL as backend. Due to the relational model, TimescaleDB has the ability to easily fetch the series relationships with other relevant entities (i.e. annotations) stored in the same system through joins. There are a few studies that suggest TimescaleDB as having favorable performance for both insertions and queries and across many different conditions [59], even when directly compared with other database systems as previously mentioned like InfluxDB[8] [60], Cassandra[9] [61] and PostgreSQL [62], [63]. However, a wide majority of the available benchmarks are written by the authors of TimescaleDB and have not been independently reviewed, which can lead to a biased comparison. Moreover, at the time of writing, TimescaleDB is at version 0.12.1, so it is still at a pre-release development stage where major changes that break backwards-compatibility can occur, making its use in a production environment very delicate for the time being.

Time series in TSDBMSs

While a strong relational schema for time series could be a positive feature for aggregate queries that require time series, annotations and users all-in-one, Jensen et al. [1] and Mathe et al. [64] suggest that using a traditional RDBMS for time series management can result in less than optimal performance at scale. Alternative storage and querying database management systems that index timestamps as primary identifiers and that perform temporal rollups to improve query speed, designated as TSDBMSs, should be used instead [65], [66].

Since the early 2000s there have been multiple attempts at implementing TSDBMSs, but a majority of these has limited support for parallel query processing. Jensen et al. [1] performed a comprehensive benchmarking over multiple proof-of-concept TSDBMSs found in academia and industrial research, and concluded that within the smaller subset of the studied data stores that presented acceptable read performance, none of these were sufficiently scalable or versatile for application with multivariate data with high cardinality. Only the most recent generation of open-source TSDBMSs have been developed with a deliberate focus on handling massive amounts of time series, responding to modern requirements of data processing. These data stores are predominantly implemented using the Log-structured Merge (LSM) tree data structure [67] as a backend. The LSM tree specializes in storing key-value mappings, fitting the formal notation of time series (1.1). Bader et al. [2] evaluated the performance of various TSDBMSs such as InfluxDB, Druid[10], ElasticSearch[11], MonetDB[12] and Prometheus[13], as well as RDBMSs like MySQL and PostgreSQL. The authors concluded that all of the tested TSDBMSs were comparable to RDBMSs in terms of feature-completeness, but that both

[7]`https://www.timescale.com/`

[8]`https://www.influxdata.com/`

[9]`http://cassandra.apache.org/`

[10]`http://druid.io/`

[11]`https://www.elastic.co/products/elasticsearch`

[12]`https://www.monetdb.org/`

[13]`https://prometheus.io/`

Druid and InfluxDB offered the best long-term storage functionality. However, TSDBMSs also have a more limited data model when compared with RDBMSs. InfluxDB specifically is only capable of indexing Strings and can only store Floats, Booleans, Integers and Strings fields.

Finally, Mathe et al. [64] presented a distributed infrastructure based on the DIRAC Grid framework[14], containing multiple computing tasks whose performance is measured as time series stored in ElasticSearch. Although most of the above mentioned TSDBMSs were designed from the ground up to handle time series data, ElasticSearch was designed primarily for text documents, but its internal structure based on the LSM tree and its optimized indexing for timestamps makes it more than capable for storing time series as well. According to the authors, ElasticSearch effectively fulfilled the requirements of handling huge amounts of time series data in both real-time and batch-time. However, ElasticSearch stores time series under a file-based architecture, which tendentially leads to a higher usage of disk space than other solutions.

While TSDBMSs implement their own proprietary NoSQL language for complex querying, time series data stores built on top of RDBMSs engines leverage the powerful SQL language. Most open-source RDBMSs also contain features such as constraints, data validation, fault-tolerance mechanisms and deep-rooted backup/restore, all of which the TSDBMSs evaluated above cannot do. RDBMSs like PostgreSQL contain an Multiversion concurrency control (MVCC) model that is compliant with Atomicity, Consistency, Isolation and Durability (ACID) properties, which allows high amounts of simultaneous read/write activity while ensuring atomicity [68]. Existing TSDBMSs had to implement similar functionality from scratch, and are naturally less prepared to ensure highly *consistent* activity at scale.

Time series in CDBMSs

The most critical issue with all of the previously mentioned TSDBMSs is the lack of clustering techniques to allow them to scale as the data set grows. Only Enterprise InfluxDB, a commercial, non-open-source variant of InfluxDB, provides clustering natively. This can be mitigated by deploying instances of these TSDBMSs across multiple machines and implementing ad-hoc load balancing mechanisms that manually sync the data set. These mechanisms have to be implemented in a way that maintains data integrity and avoids data corruption at all costs, which typically take many years to develop and mature. Some of these mechanisms already exist in the open-source community, but these are typically not actively supported nor sufficiently mature. One example is InfluxDB Relay[15], from the developers of InfluxDB, whose official open-source repository shows that, at the time of writing, the last contribution was made in November 2016.

An alternative solution is to leverage data stores geared for scaling like Cassandra, an open-source, distributed NoSQL Column-oriented Database Management System (CDBMS) that also uses an LSM tree as backend. Cassandra supports multiple active clients connected in

[14]`http://diracgrid.org/`
[15]`https://github.com/influxdata/influxdb-relay`

thread-safe sessions and implements all the *consistency* and *availability* mechanisms necessary when building a distributed system. Under this approach, one can directly leverage the same CDBMS data store for storing time series, users and annotations simultaneously, avoiding the need to deploy additional databases specific for each task and facilitating the handling of *consistency*. Moreover, CDBMSs like Cassandra allows users to specify their preference in the *consistency/latency* trade-off on individual requests, allowing different queries to either collect the currently available record or to wait for pending writes to finish in order to collect the most up-to-date snapshot instead. In [51], Kalakanti et al. proposed a benchmarking toolkit that evaluates concurrency, load management, scalability, *consistency*, BI query performance and fault-tolerance for HBase[16] and Cassandra. The authors observed that while both are linearly scalable for native writes, both showcase lower performance for aggregate temporal queries of large volumes of data.

To address the shortcomings of CDBMSs with massive time series data sets, a number of TSDBMSs with the necessary rollup operations and optimized aggregation of time series were implemented on top scalable CDBMSs like Cassandra and HBase, such as Blueflood[17], KairosDB[18], OpenTSDB[19], Chukwa[20] and Databus[21]. Tomasz Wiktorski [49] evaluated the performance of OpenTSDB and Chukwa, both being built on top of distributed processing frameworks such as Hadoop[22] and with support for HBase, concluding that OpenTSDB was the most flexible solution for most use cases. In [50] it is suggested that KairosDB is able to fulfill near-linear scalability, load balancing, clustering and reliability requirements better than Databus and OpenTSDB. Bader et al. [2] compared the performance of Blueflood, KairosDB and OpenTSDB with other non-scalable TSDBMSs and RDBMSs. The authors concluded that while the TSDBMSs built on top of CDBMSs were feature-complete in terms of scalability, they severely lacked in time series processing functions, continuous calculations and long-term storage mechanisms.

Conclusion

Based on the studies mentioned in this section, modern LSM tree TSDBMSs databases are still the best candidates for fully-featured and efficient querying over massive amounts of historical data, despite the fact that these databases lack high *availability* mechanisms built-in. InfluxDB shows over multiple studies to have both the smallest impact in disk storage and the lowest latency when querying long-term historical time series, which fits the intended use case. However, InfluxDB also comes with a few drawbacks when compared to other database systems, such as the limited data model and the limited ability to store relations, making it unwieldy to use for storing non-series data like annotations. This essentially prompts the

[16]`https://hbase.apache.org/`
[17]`http://blueflood.io/`
[18]`https://kairosdb.github.io/`
[19]`http://opentsdb.net/`
[20]`http://chukwa.apache.org/`
[21]`https://github.com/deanhiller/databus`
[22]`https://hadoop.apache.org/`

need for a polyglot data management model where the full data set is split between two independent storage tools, each with their own data models and *consistency* mechanisms.

2.3.2 Annotations

The final step for the technological survey of data management technologies was to study various approaches for storing annotations and other associated entities such as users. The existing research on how time series annotation tools manage this type of data is slightly restrictive, as many of these tools are not built with massive data sets and heavy workloads in mind. Nevertheless, a common trend in all of these tools is that RDBMSs were predominantly used to store annotations due to the strong ties that annotation data had with other analogous data, such as categories or collaborator profiles.

In [5] all active entities, including annotation services, are modeled as agents [69]. These agents are persisted in a HyperSQL relational database, but this choice is described as "fortuitous" only due to the inclusion of Java, Java Database Connectivity (JDBC) and SQL definitions. The authors do not benchmark its performance in comparison with other, more mature and battle-tested relational databases like MySQL and PostgreSQL, and with other Object-relational mapping (ORM) frameworks, such as ones that extend on the Java Persistence API (JPA) like Hibernate[23] or EclipseLink[24].

Bhagwat et al. [70] describe an implementation for a management system where annotations are stored embedded in records that represent the target series data, instead of placing these annotations in their own separate table. While this brings performance improvements for series queries, since only one table has to be queried and no aggregations/joins are required, the management of stored annotations is impractical for massive data sets, since a single annotation covering a segment of data has to be replicated across all data points belonging to that segment, and any changes to the annotation have to be propagated accordingly. For example, if an annotation were to be assigned to tens of thousands of metrics, the transaction required to create it or update it would require a cursor over every single one of the affected metrics, which is unwieldy and not appropriate for scalable systems with massive data sets.

Lastly, Eltabakh et al. [71] and Felipe da Silva [72] iterate on the previous approach and propose a more granular schema, where annotations are instead linked to the annotated data points by containing identifiers of that data. Changes made to annotated time series or to the annotations directly do not reflect on the other data type nor require propagation over multiple records. Under this polyglot model, annotation and user logic is stored in a separate data store, and associated with time series data by attaching to the annotated data points an extra field with the annotation's unique identifier.

With a polyglot framework in place, there is now a plethora of storage formats and databases that can be used. The choice of data store, relational or otherwise, should enforce the aspects of searchability, user access-control, and versioning that the suggested model in 2.2.2 supports. However, an important thing to take into consideration is that, as annotation

[23]`http://hibernate.org/`
[24]`http://www.eclipse.org/eclipselink/`

and user logic is separated from time series between independent data stores, the increased granularity can potentially translate into a higher number of bottlenecks and points of failure. The estimated time for queries is firmly increased as the client requests more simultaneous data types in a single operation. A query for time series within a certain interval, as well as annotations that comprehend those time series, could require a set of sequential and dependent calls to the time series database, the annotation store and the user store. For example, the query would lock and wait for results from the annotation store, and then lock again for results from the user store to collect data about the authors of each annotation. This is highly impractical for a system intended to support multiple concurrent queries at scale. Instead, queries have to be taken into consideration during backend architecture planning, and the system architect has to ensure that, in the case of queries that require data from more than one storage unit, these can be fetched asynchronously.

Nonetheless, the granular model implies that the chosen data store for time series must support the inclusion of additional metadata for each data point, in order to store the annotations unique identifiers. All of the LSM tree-based data storage solutions mentioned above do support this, but while most LSM tree data stores like InfluxDB, Cassandra, OpenTSDB and KairosDB append key-value metadata on the same LSM tree used as backend, Prometheus leverages a separate key-value store, LevelDB, to store simple text labels indexed by a unique key. The latter method is more efficient, as LevelDB offers high *consistency* with strong writes and compression under-the-hood.

The research focus was now to evaluate which data management system between RDBMSs and CDBMSs was a better fit for the storage of annotation and user state. It is important to point out that, within this use case, these entities have a strong relationship with each other, so they benefit from being well expressed in the data management layer. While CDBMSs provide more distributed solutions, they do so by weakening relations between two separate entities/tables. To ensure *consistency* in a single update that affects multiple entities simultaneously, as well as their relationships, requires a very ad-hoc process in CDBMSs. In an asynchronous environment where multiple users propose changes in parallel, an update to the state of multiple objects (and their corresponding version records) must be handled within the same transaction or by using concurrency mechanisms such as locks and blocking queues, so as to prohibit users from reading outdated versions and overriding changes or performing invalid actions, such as editing an annotation after it has been deleted. To further check if the changes are being applied by a user that has read the latest snapshot of the target annotation, a contract can be established where users are required to send the last-modified timestamp of the object they are altering, and if this timestamp matches with the actual last-modified timestamp of the latest object, then the update is applied (3.3.8).

With this, RDBMSs present themselves as the strongest candidates for structuring strong relations between annotations and users. Based on the studied benchmarks alone, PostgreSQL and MySQL are the most preferred open-source RDBMSs based on their performance and feature-completeness [2], [48], [62], [63], [73]. Between these two, MySQL tendentially reports faster reads and writes. However, MySQL achieves this by prioritizing *availability* over *consistency* in its design[25], and is only ACID-compliant when using specific storage engines that implement a MVCC model, such as NDB Cluster[26] and InnoDB[27]. When compared with MySQL with the InnoDB storage engine, PostgreSQL shows a mature MVCC model where Data Definition Language (DDL) statements are always transactional [68]. As the platform scales and structural changes to the tables are required, such as changing indexes or updating large tables of records, PostgreSQL aggressive *consistency* and strict policies will avoid instances of data corruption or loss at all costs [28]. Whereas databases like Oracle[29] and MySQL attempt to reconstruct previously committed versions to capture uncommitted changes and avoid data loss, PostgreSQL stores all row versions in its internal data structure until one user performs a commit, merging changes and making them readable to other users [74]. This means both users can simultaneously insert, update or delete records without being locked in place, and are only required to commit all changes once, which will make the changes atomically available for subsequent readers.

2.3.3 Versioning

Another aspect to consider when choosing a database engine is the ability to handle versioning of contents under a distributed platform. Conventional databases represent the state of multiple entities from a single moment in time, and most of the existing commercial and non-commercial databases deal with update operations over entities by modifying their state and forgetting the old contents, losing all historical data. Although traditional databases were not built from the start to support versioning directly, mechanisms for versioning data in these databases have been studied in depth [75].

Versions within records

Snodgrass [77] proposed extensions to the SQL language in order to allow entities to have a preserved historical state within their tables. As collaborators introduce sequential changes, the revisions of the changed entity are stored as records in the same table as the original entity, unified by the same identifier and distinguished by a version id or a timestamp. This approach minimizes the need to create additional tables, but also leads to many more disadvantages, since the appending of multiple versions into a single table will cause queries to scan over an exponentially higher number of records that scale both vertically (as the number of separate entities increases) and horizontally (as the number of versions for each entity increases) [78].

Fujita et al. [79] propose an approach where versions are stored within an array that is embedded in their respective records. This reduces storage overhead, leading to more optimal reads, but any update to an entity will require both an update to the affected columns in the record and an insertion of the previous version to the embedded array, leading to more complex writes.

[25] `https://wiki.postgresql.org/wiki/Why_PostgreSQL_Instead_of_MySQL:_Comparing_Reliability_and_Speed_in_2007`

[26] `https://dev.mysql.com/doc/refman/8.0/en/mysql-cluster.html`

[27] `https://dev.mysql.com/doc/refman/8.0/en/innodb-storage-engine.html`

[28] `https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/`

[29] `https://www.oracle.com/database/`

Versions in tables

In order to tackle the drawbacks of the previous approaches, the historical data can instead be stored in full on separate versioning tables. This is one of the core aspects of event-sourcing design [80], where all application events are stored in a append-only log and each event includes all of the data that the updated object had during that event, regardless if it was changed in that event or not. This can be optionally coupled with boolean flags that indicate whenever a certain field was part of the changes made in that specific version. This approach translates to higher-performant queries for both the current state of the data and for any specific version, while simultaneously enabling a highly *consistent* system with low-latency writes. However, without any restriction, compression mechanisms or expiration policies, this approach produces the highest amount of redundancy of contents out of all versioning approaches, leading to the largest disk overhead, as each unchanged field is repeated as many times as the number of versions that go by without affecting that field.

An alternative model is one based on deltas [78], where each record in a versioning table contains only the contents of fields that were changed in that version. The contents of unchanged fields can be reconstructed by performing a theta join over all prior versions where those fields were last changed. This approach translates into a considerably lower disk usage, but it can be cumbersome to rebuild the contents of a version when dealing with massive amounts of changes. A query for the full data record of a specific version v will not only require the changed fields in v but also the unchanged ones, which would be empty in v. Each unchanged field must therefore be collected from previous versions v_p where those fields were last changed prior to v. Given a situation where a certain field is set when created and never or rarely changed, and where the entity in question has a constantly-growing quantity of versions being produced, then the process of rebuilding the complete contents for any given version requires an expensive cursor over all these records.

Versions in non-linear branches

While all previous models enable the exploration of snapshots over a linear chain of versions, they do not allow users to create their own independent environments of changes that can then be synchronized with others at a later point in time. A multi-branch model would enable a highly *consistent* architecture, since each version branch is an independent state that does not have to be synchronized with other branches at all times. Each client has essentially a local copy of the centralized contents, so multiple edits and deletes can be performed freely. Integrity checking and conflict-resolution are only required when merging a revision branch to a single unifying "master" branch of common knowledge. Each client is forced to poll the "master" contents and be up-to-date with them before they can commit their own branches, handling merges as they see fit.

This pattern is similar to how distributed version control systems function, such as Git[30] and Mercurial[31]. Halilaj et al. [81] proposed Git4Voc, a system in the context of collaborative

[30] `https://git-scm.com/`

[31] `https://www.mercurial-scm.org/`

vocabulary development that leverages Git directly as a versioning protocol for text documents. However, every version control system that is built primarily for plain text and source code does not support the advanced querying capabilities that are available in RDBMSs and CDBMSs, such as filtering, joining or aggregations of entities. Git is also known to have several shortcomings for large data sets [82], which is a direct impediment to the system's ability to scale. As a way to provide branching and merging over the existing RDBMSs within a centralized system, Bhardwaj et al. [73] propose Datahub, a management tool that uses PostgreSQL as backend and manages separate databases to represent each client branch. However, as a result of the substantial redundancy of databases, this approach leads to a high disk usage that does not scale judiciously for massive data sets [83]. This makes distributed versioning protocols incompatible with the proposed core philosophies, but its features for sequential contribution, where collaborators cannot freely commit changes before polling the most recent snapshot of data and merging potential collisions, would be immensely advantageous to enable a safer collaborative framework.

Conclusion

Overall, all of the approaches listed correspond to a storage vs efficiency trade-off. Any attempts at compression over the version data set, either by using version deltas or by compressing the data representation itself, will negatively affect query performance, while any attempts at improving query performance will negatively affect disk usage. Another major issue with managing data set versioning comes from allowing many users to read and modify entities in parallel. The versioning approaches that rely on appending version data inside the modified records, such as the embedded array approach, or on rebuilding the full state of objects on every query, such as the delta-based table approach, will inevitably lead to *consistency* issues when multiple updates are triggered in parallel.

2.4 Architecture

The next step of research is to survey how existing platforms set up and connect their data storage and processing components to handle intense workloads. Although some of the previously mentioned time series analysis systems use strong data management models that can persist large data sets, these systems fail to implement distributed techniques like replication and partitioning to improve *availability* of the service.

Sow et al. [8] presented a middleware for health-care analytics, processing real-time EEGs data streams. The results can be exported and presented to analysts using standard protocols such as Hypertext Transfer Protocol (HTTP) and Simple Mail Transfer Protocol (SMTP). The Artemis platform [4] allows the output of data streams to a clinical information management system through Transmission Control Protocol (TCP). Both of these solutions build a modular backend architecture that manages multiple independent processing adapters, using IBM InfoSphere[32]. This architecture is said to scale well to the increasing number of patients connected to the system, but no distributed computing techniques are mentioned.

[32]`https://www.ibm.com/analytics/information-server`

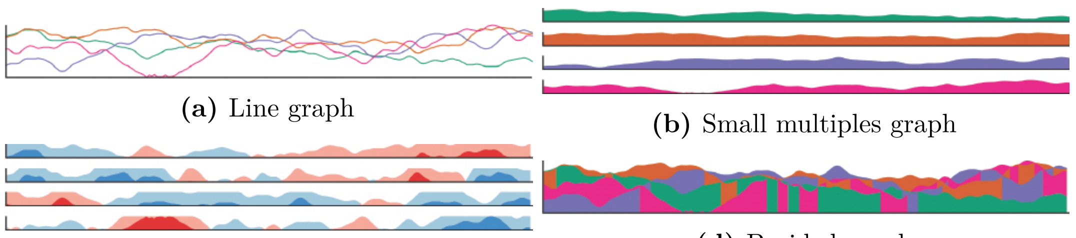
In [5], O'Reilly implements an analysis platform that focuses on core design concepts like modularity, portability, Separation of Concerns (SoC) and operability. It leverages an event-driven model where the system records all operations that other components can subscribe to, following a publish-subscribe pattern. When the system receives new data or there are changes to a component's state, these events are notified to every subscribed module at a system-wide level. For example, when an analyst creates a new annotation in a time series segment, this event is sent to the data server and is consequently propagated to the frontend of other clients, updating the browser application of analysts viewing that same segment in real-time and without requiring a refresh. However, the author indicated that the persistence logic is performed by a single backend node and there is no load-balancing of components, so the platform could run into *availability* issues when concurrently used by multiple users.

O'Reilly also takes advantage of agent theory [69] as a design paradigm to structure data flow and publish-subscribe relations throughout the platform. These agents, as plain Java objects, are serialized and deserialized into files for the purposes of long-term persistence, using the Java Serialization Application Programming Interface (API). Transmission of data also uses a compact binary serialization format, albeit its specifics are not clear. However, the Java serialization process is generally considered as a weak format for long-term usage [84], as the rules of serialization can change when the Java class definition or the Java compiler itself changes. Moreover, the usage of a non-standard binary data serialization format means that this data will only be readable for as long as its parsing tools are maintained. Other highly contributed, machine interpretable serialization standards should be used, like the human-readable JavaScript Object Notation (JSON) [85] or Extensible Markup Language (XML) [86] formats or the platform-neutral Protocol Buffers [87] or FlatBuffers[33] binary formats. All of these formats have been matured for fast serialization and deserialization, as well as for the passing of compact messages between servers and network-limited mobile devices [88]. The mass support of these standards across various software frameworks and applications, and the open-source nature of the many parsing libraries that support these, means that these formats have better chances of surviving in backwards-compatible ways.

The platform mentioned in [53] implements a HTTP backend server that provides a Create, Read, Update and Delete (CRUD) interface using the Representational State Transfer (REST) protocol [89]. Time series are stored in a relational database, and all communication between this database and a web-based application is made through REST *POST* and *GET* calls using an XML payload. Although the proposed platform leverages modern web technologies, the architecture is still not properly optimized to handle massive amounts of time series, as it has a considerable delay of approximately fifteen seconds between acquiring new data and presenting it in the frontend. In addition, this frontend does not poll real-time data from a continuous streaming pipeline, but by using a scheduled task instead.

Finally, in [18] the authors iterated on this by building a secure system that connects multiple sensors from wearable devices, using the HTTP protocol for data flow between a web frontend and a data management module while authenticated with an API token.

[33]`https://google.github.io/flatbuffers/`



(a) Line graph

(b) Small multiples graph

(c) Horizon graph

(d) Braided graph

Figure 2.2: Comparison between various Cartesian time series visualizations, as shown in [26].

2.5 Visualization

The aspects of time series visualization outlined in the Introduction chapter (1.2) compose the main challenges in observing time series in line charts. Many alternative visualization techniques for time series encoding were developed in an attempt to solve the listed challenges, but in doing so they introduce their own set of issues. The main contribution from these, however, comes from the display of time series in a manner that improves readability of a specific pattern-type or occurrence.

Additionally, all of the researched time series analysis solutions implement unique interfaces for visualization and querying of time series. Some of the explored works also propose novel interface components for displaying and interacting with time series and their respective annotations, so it was essential to study these for the purposes of this dissertation.

This section is separated into three subsections: the first subsection studies various alternative techniques for displaying time series, comparing their performance over different use cases; the second subsection lists visualization tools and their interface design choices, without going into detail on what software technologies were used; and the third subsection performs a survey over technologies used for rendering interfaces and time series data.

2.5.1 Techniques

Usage of alternative visualization schemes might not always be appropriate depending on the context of the data, so it is important to thoroughly benchmark these in comparison with the always-versatile line charts. The main goal of researching these visualization techniques should be to find schemes that can be appended to line charts in order to improve the overall visual perception over the time series data set, and not to replace them.

Javed et al. [26] evaluated the performance of various visualization techniques such as simple line graphs (Figure 2.2a), small multiples [94] (Figure 2.2b) and horizon graphs [95] (Figure 2.2c). The authors observed that, for a low number of simultaneous time series on display, multiples and horizon graphs can generate more visual clutter than line graphs, making line graphs a better option. Additionally, a new graph named braided graph (Figure 2.2d) is introduced, which acts similarly to a line graph but attempts to improve color perception by filling the area under each curve with the line's color encoding. The performed evaluation showed that task efficiency for braided graphs was only equivalent and never better than for line graphs. However, this study was performed in a static environment where users could

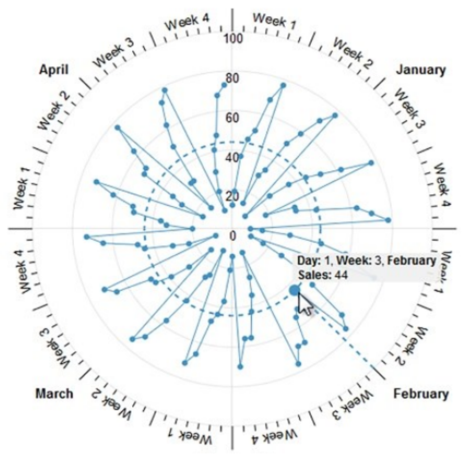
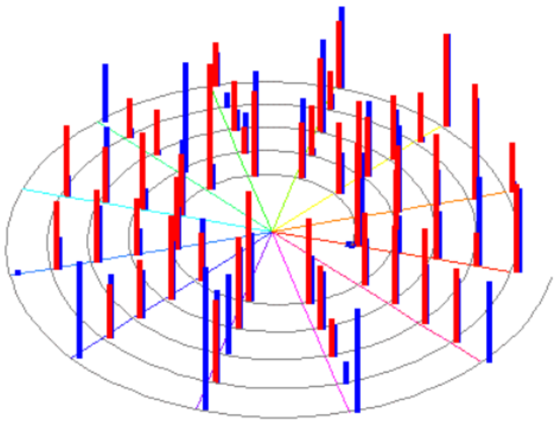


Figure 2.3: Radar chart, as shown in [90].

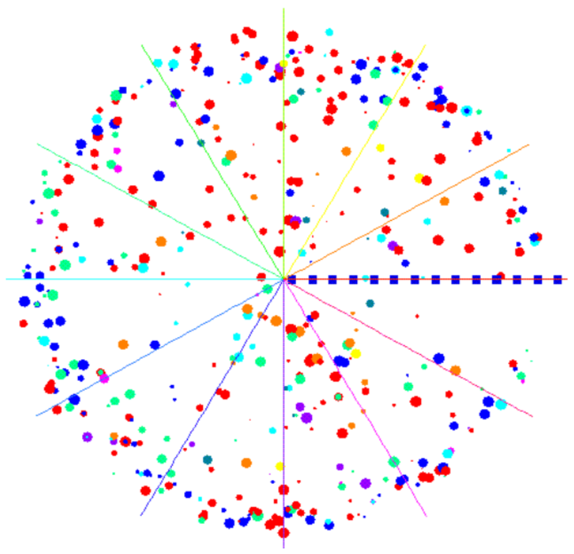
not manipulate with the data through filtering and zooming, which could imply that the evaluation results are not comprehensive nor adaptable to a real-life context.

In [91], Cleveland et al. propose the usage of the Cartesian coordinate system and line graphs for time series representation, suggesting that the strength of these graphs is in the ability to perceive time and values for each data point and to understand how they relate to each other. However, in [90] Adnan et. al suggest that there is limited empirical evidence available that establishes Cartesian and Polar coordinate systems as stronger candidates for time series visualization. Within the Polar coordinate system, Tague [92] suggests the representation of values over a narrow time-frame under a radar chart (Figure 2.3). Fuchs et al. [93] compared the effectiveness of line graphs in the Cartesian system against various radial encodings in the Polar system, concluding that while Cartesian and Polar systems had generally comparable performance, line graphs performed better for detection of peaks, anomalies and trends. It further suggests that Polar coordinate systems and radar encodings were better suited for reading exact values at a single-point precision, although the most common analysis tasks performed over time series benefit more from a pattern-wide precision.

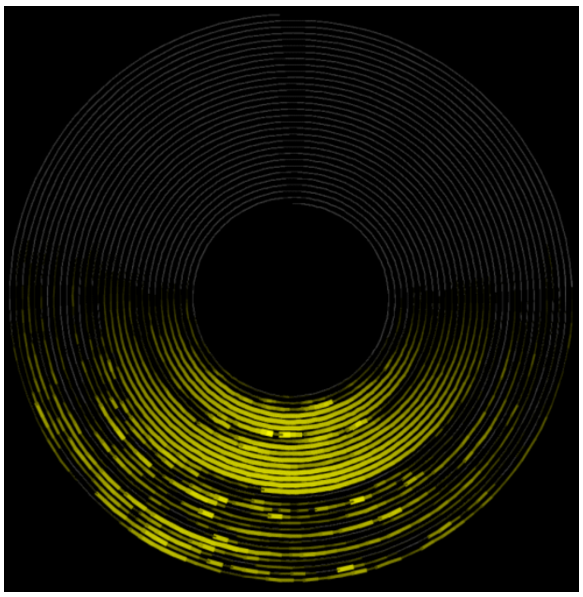
Various studies suggest an approach for visualization of time series based on rings or spirals [101]–[103] (Figure 2.4), where each periodic segment of time series is structured in a ring, mapping values to their radius, color and outline width. This visualization excels over line graphs when displaying data sets with periodic patterns, since every event will naturally be mapped in proximity to other events in similar timeframes and be more visually evident. While a novel approach, this visualization will more often than not occupy a large amount of the available visual space. As the number of data points increases, while keeping the ring diameter constant so that it fits in its assigned chart space, the less clear the color-coding and the width of the rings will be. Due to this, this approach is primarily useful for a relatively small segment of time with a low number of data points, in the order of seventy-five thousand (75 000) points or lower [104].



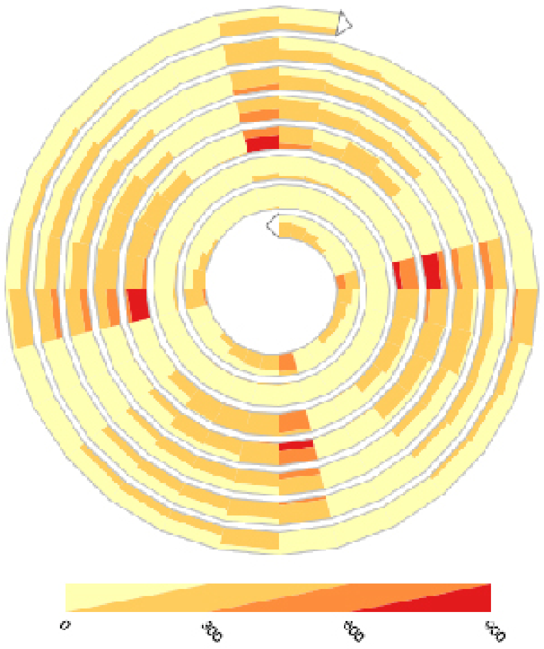
(a) Maximum sizes for two chimpanzees in [101].



(b) Movies by release date in [101], sized by popularity and colored by genre.

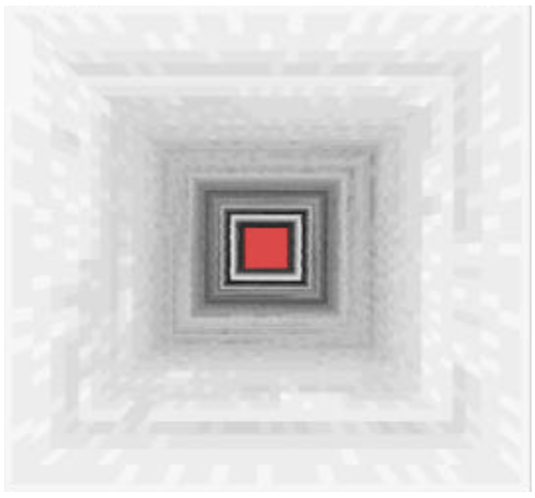


(c) Sun-light throughout multiple days in [102], where wider ribbon widths correspond to moments of higher sunshine intensity

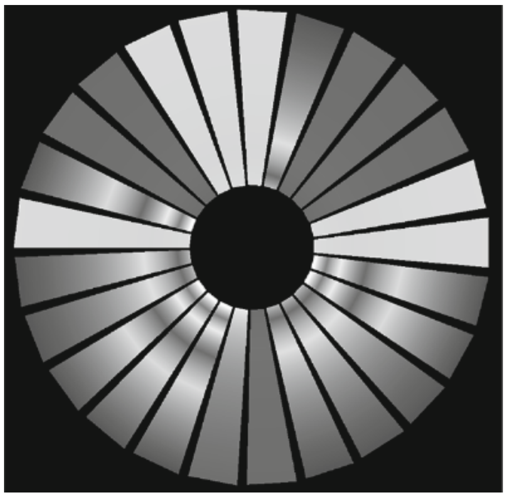


(d) Human health data set in [103], where more intense colors correspond to a higher number of new infections detected.

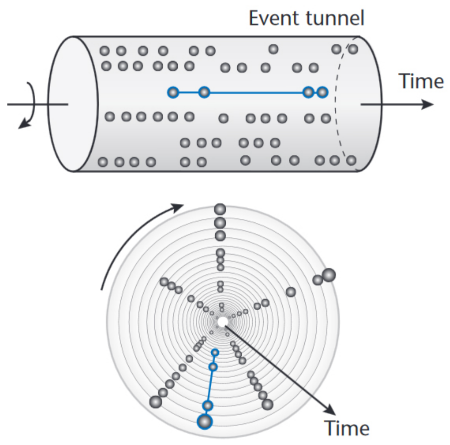
Figure 2.4: Various spiral visualizations, where cyclic patterns are visually identifiable by observing crowded areas.



(a) DataTube [105] perspective tunnel displaying daily aggregate share prices on the Wall Street stock exchange. Older data is further away, newer data is near.



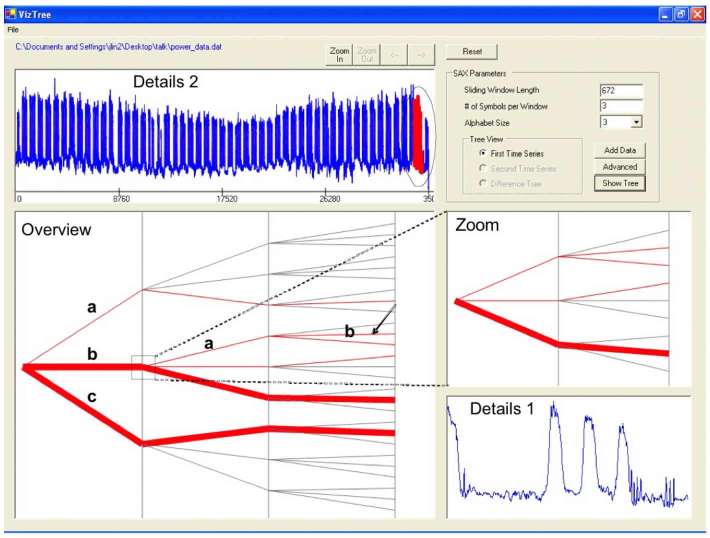
(b) DataTube2 [104] perspective tunnel, which iterates on DataTube by adding clustering techniques and interactive features.



(c) Event Tube [106] visualization: above shows a 2D display of the tube from the outside, below shows the tube from the inside.

Figure 2.5: Various tubular visualizations, where data that diverges from the norm is visually distinguishable from the full context by observing intensely colored areas.

To deal with the issues present in two-dimensional ring visualizations, various authors proposed the usage of a three-dimensional tubular or helix shape instead [104]–[107] (Figure 2.5). This encoding consists of using the depth of a tube or helix to represent the progress of time, starting from the center and expanding in the direction of the viewer, while its surface, diameter and color represent values. Bouali et al. [104] (Figure 2.5b) iterate on the first DataTube visualization [105] by introducing various interactive features such as 3D navigation, zooming and annotation over facets of the tube. This technique is an interesting take on time series representation that can ease data mining tasks that rely on clustering analysis. However, the implementation and usage of 3D graphical tools is more taxing on computing resources client-side than 2D visualizations, translating into lower compatibility.



C:\Documents and Settings\jlin2\Desktop\talk\power_data.dat

SAX Parameters

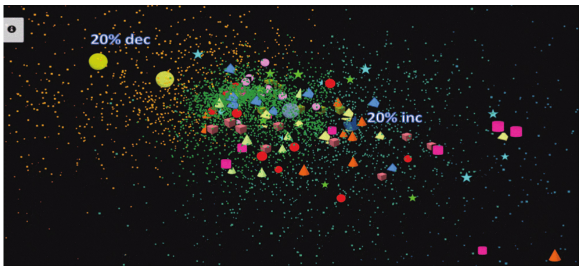
Sliding Window Length 672

# of Symbols per Window 3

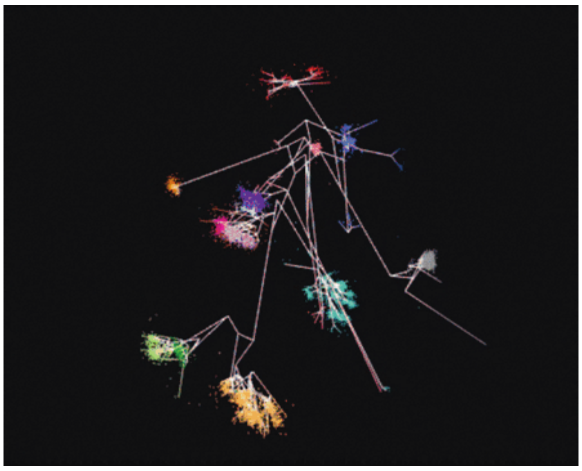
Alphabet Size 3

(●) First Time Series ( ) Second Time Series ( ) Difference Tree

Figure 2.6: VizTree interface overview.



(a) Cluster view of a data set with stock velocities between 2013 and 2014.



(b) Tree view of a data set with 100 000 points. Lines connect clusters of data based on similarity or user-defined conditions.

Figure 2.7: PlotViz/WebPlotViz interface where data can be mapped with different shapes and colors in order to be identifiable.

Some propose the visual exploration of time series within graph or tree structures [19], [96], [112], [113], using concepts like arcs, vertex, edges, leaves and nodes. Lin et al. [19] propose *VizTree*[34] (Figure 2.6), a suffix tree [97] visual encoding for large-scale time series data. Each time series segment is converted into a symbol string and arranged into leaves. In addition, similarity between two time series can be calculated by directly comparing their trees with any tree-diff algorithm [98]. In [112], [113] the respective authors propose *PlotViz*[35] (Figure 2.7), a tri-dimensional point cloud and tree visualization. Different data points can have custom sizes, colors and special shapes in order to be visually distinguishable. Analysts can arrange custom clusters of data points and tag them with user-defined keywords.

[34]`https://cs.gmu.edu/~jessica/viztree.htm`
[35]`http://salsahpc.indiana.edu/plotviz/`

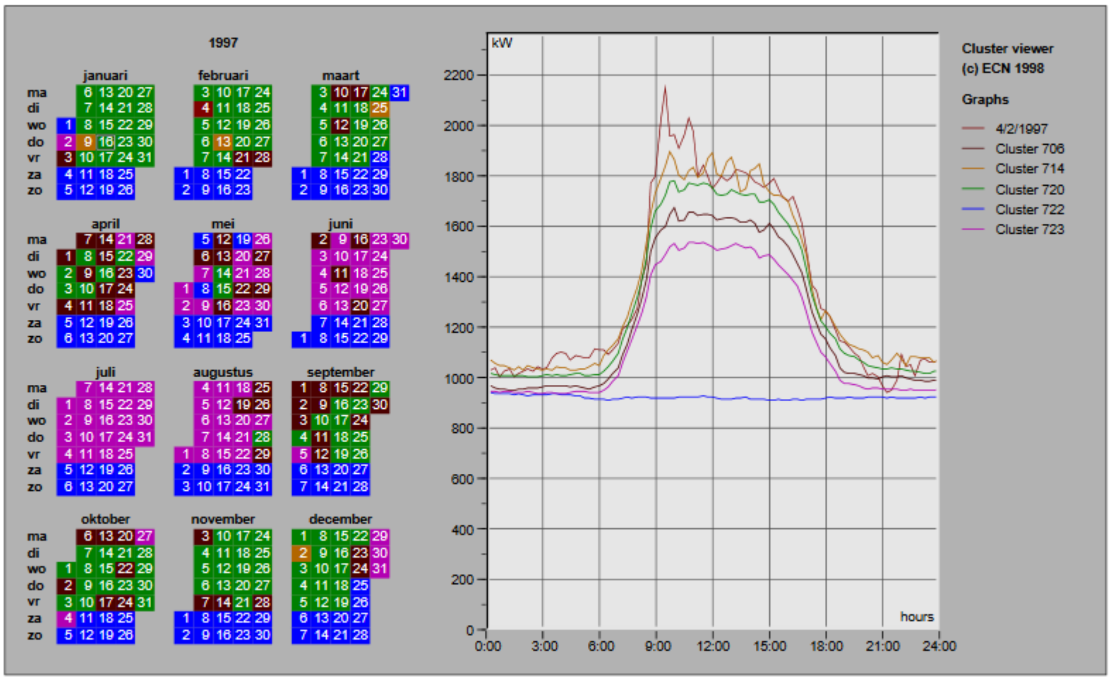


Figure 2.8: Calendar visualization with a cluster analysis of power demands of a research facility.

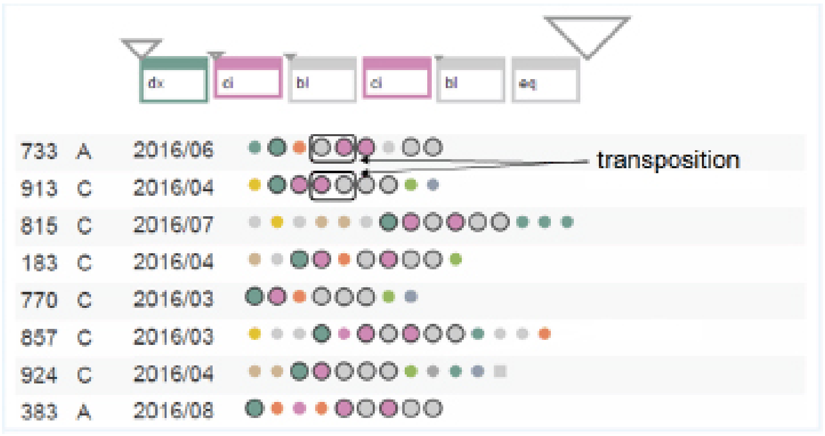


Figure 2.9: Sequence synopsis visualization where a cluster of sequences contains events with deviations from the pattern.

Another alternative visualization scheme is the calendar-based visualization [99], [100] (Figure 2.8), where time series are split into sequences of days, weeks or months, and organized into a calendar view. This visualization is advantageous when needing to provide a general overview of the data set over an entire year, but it is limited in usage for non-uniform data sets where this overview is irrelevant, such as financial data from the stock market.

In [27] (Figure 2.9) a novel technique for sequence synopsis is proposed. This visualization reduces visual clutter and increases clarity by using the Minimum Description Length principle [108]. This principle corresponds to a compromise between ease of human interpretation and the amount of detail on display. While an useful technique that can reduce the amount of time that analysts might require to extrapolate meaning from data, in its current state it has

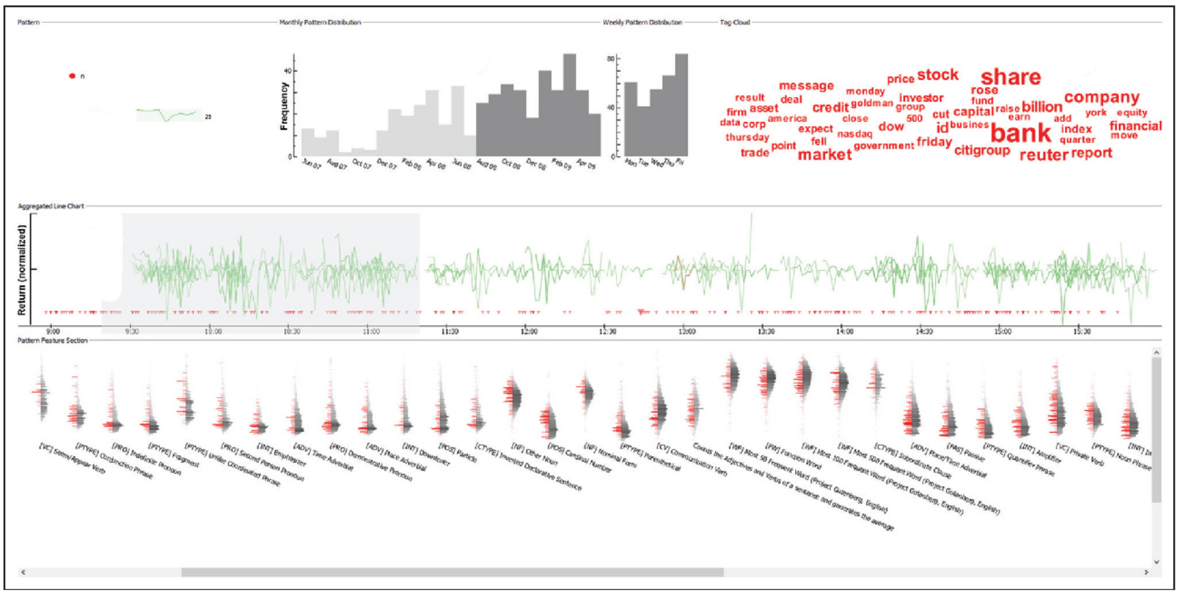


Figure 2.10: Financial data visualization with information extraction algorithms that match real-world developments with metric variations.

scalability issues. One such issue is that data can be summarized to an extreme level, so the larger the data set, the more will distinct patterns and important events be filtered out. Plus, the implemented algorithm has a high requirement on computational resources.

Finally, in [109] (Figure 2.10), Wanner et al. propose a novel visualization tool that automatically extracts user-highlighted patterns in time series data sets, displayed in a line graph, and searches these patterns in financial reports and news articles to discover textual information that gives context to the extracted patterns. The temporal data is joined with the textual data that explain the phenomena in the data, so that analysts are given as much feedback as possible. Users and automated processes can also annotate series with metadata that semantically explains the pattern, similarly to the concept of annotations outlined in this dissertation.

Throughout this survey of visualization techniques, a recurring aspect is that line graphs are ubiquitous in usage due to their capacity to scale and to adapt to a domain-agnostic data set. Even though both VizTree [19] and 3D Tube [104] showcased inventive techniques for displaying temporal data that scale well with growing data sets, most visualizations were not as flexible and scalable for analysis purposes.

2.5.2 Software tools and interfaces

Time series

Part of the research effort of this dissertation is to study existing visualization platforms for massive time series data sets, and evaluate their features and interface paradigms.

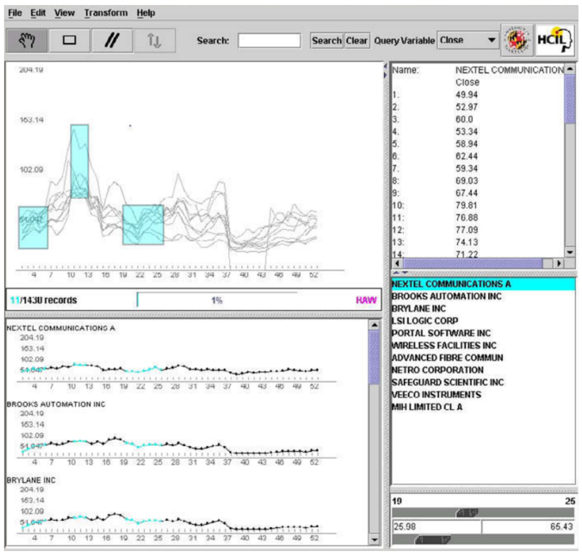
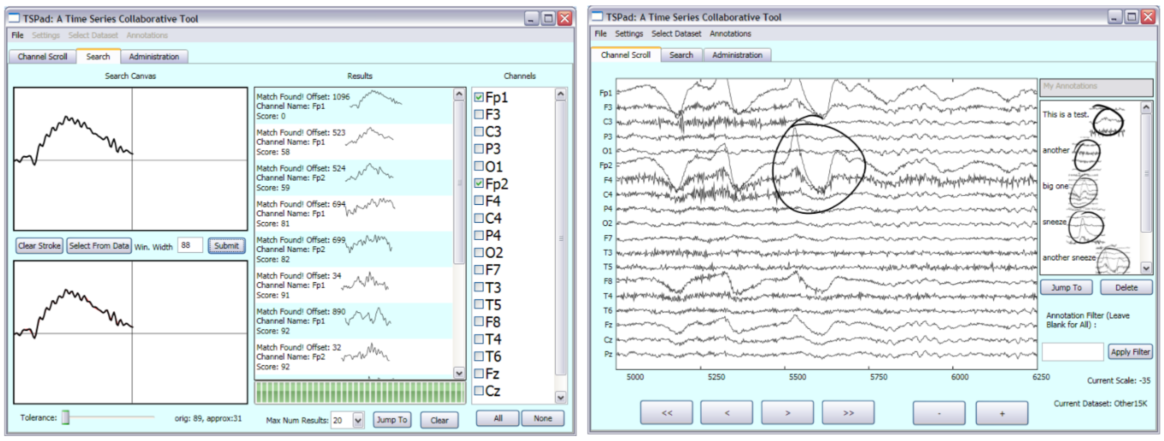


Figure 2.11: TimeSearcher interface showing the full time series data set as line graphs and user-defined query areas as blue rectangles. The remaining panels display the queried data in detail and the selected items.



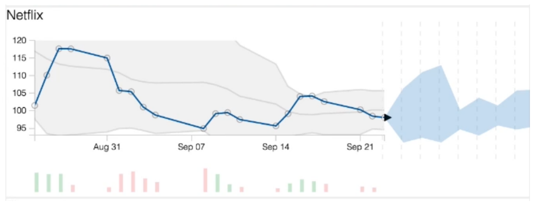
(a) The user manually draws the intended waveform and a normalized waveform approximation of it is displayed below, the latter of which is used to match with the data set. The normalized waveform can be adjusted by using the approximation tolerance slider.

(b) The user manually draws any shape that represents the intended annotation on top of displayed data set, annotating a smaller subset of series in the same sequence of time. Annotation shapes can be named and shown on a list in the right panel.

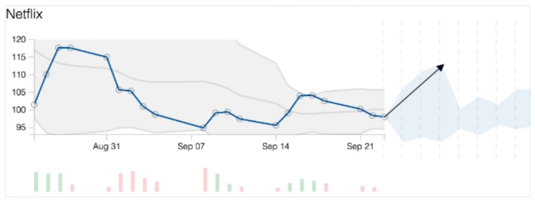
Figure 2.12: TSPad interface for search and annotation features, leveraging tablet-PCs capabilities.

In [24], [110], [111] the authors describe a visualization tool for line graphs called *TimeSearcher* (Figure 2.11), that uses rectangular locators called *TimeBoxes* to query regions of interest directly in the data. The user can make a drag-and-drop selection over the graphs to intuitively and interactively trigger queries for similar patterns instead of setting the filter criterias manually.

TSPad [40] (Figure 2.12), a standalone tablet-PCs Windows application for collaborative time series analysis that allows users to browse series data and annotate it. Annotations are used by the platform to aid the training of automated diagnosis algorithms and machine



(a) The blue area on the right side indicates the potential area that future stocks can be based on historical pattern.



(b) The user can drag and drop the black arrow, setting their prediction of stock values and influencing the future behavior of the automated prediction module.

Figure 2.13: TimeFork interface that allows users to observe stock market data, view automated predictions and indicate their own predictions.

learning models. Analysts create annotations under a restrictive access policy, where some analysts have limited read and write permissions over annotations of others. A view window displays a smaller subset of the overall time series data, and users can navigate through past or future events by clicking on left and right arrow buttons respectively. Users can increase or decrease the amplitude of the series waveforms to increase the clarity of data fluctuations. Moreover, *TSPad* supports waveform search through free-form shape sketching or through the selection of similar waveforms in the visualization (Figure 2.12a). When performing queries, the user can set a similarity threshold that searches for series with an adjustable degree of deviation from the one drawn or selected. Sketchable search is a novel idea, but it is impractical for non-tablet devices, and can generate a considerable amount of False Positives and False Negatives.

TimeFork [15] (Figure 2.13) is a web application that, like *TSPad*, establishes a human-machine interaction through the usage of tablet features. Users can observe the machine-generated window of predictions and attempt to predict future stock patterns by sketching where the future patterns, continuously training a machine-learning model.

WebPlotViz [16] implements the *PlotViz* visualization within a standalone web application. Users can interact with both the cluster view (Figure 2.7a) and the tree view (Figure 2.7b) by zooming, rotating and panning. This application also implements similarity search through selection of time series vectors or clusters, so users can rapidly find similar clusters. Finally, it provides versioning support for plot points, so users can manage multiple perspectives of the same data set and save changes as separate versions that do not overwrite others.

In [12] (Figure 2.14), Healy et al implemented a platform that displays real-time physiological streams and enables the usage of annotations for knowledge sharing and discussion between collaborators. This application also performs authentication and management of users for the purposes of establishing a learning platform, where junior analysts can lookup annotations from senior analysts and, in turn, senior analysts can monitor and rate the work of junior staff. Users can query, annotate and comment any annotation in a public or private manner, as well as rate annotations using the Likert scale as a confidence measure. Users also have a profile page, where all their annotations are listed.

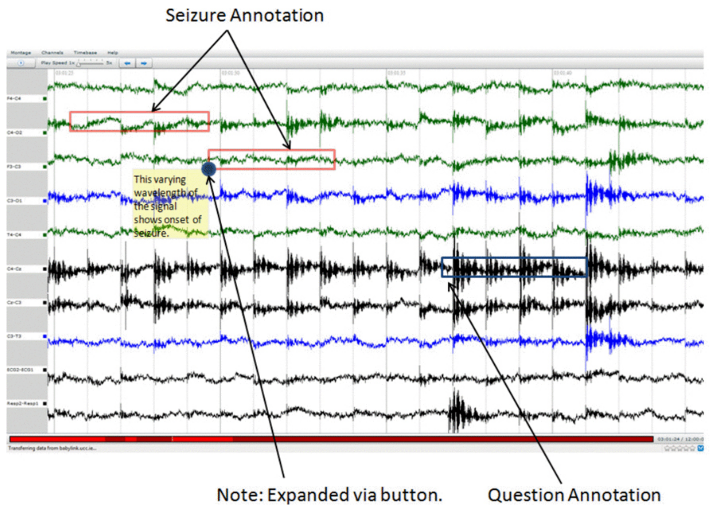

Seizure Annotation
This varying wavelength of the signal shows onset of seizure.
Note: Expanded via button.
Question Annotation

Figure 2.14: Web-based viewer of EEG data set with waveform classification annotations, associating the data set directly with a category or semantic, and text annotations, relating events outside of the data.

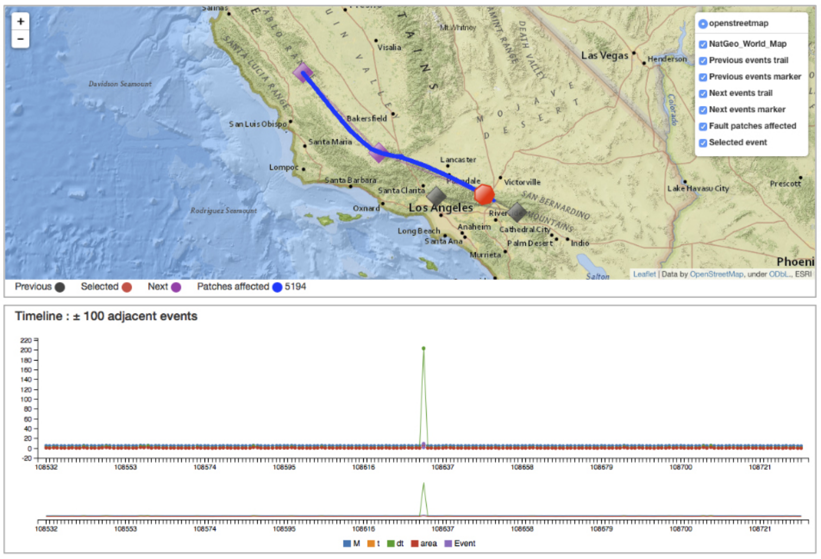
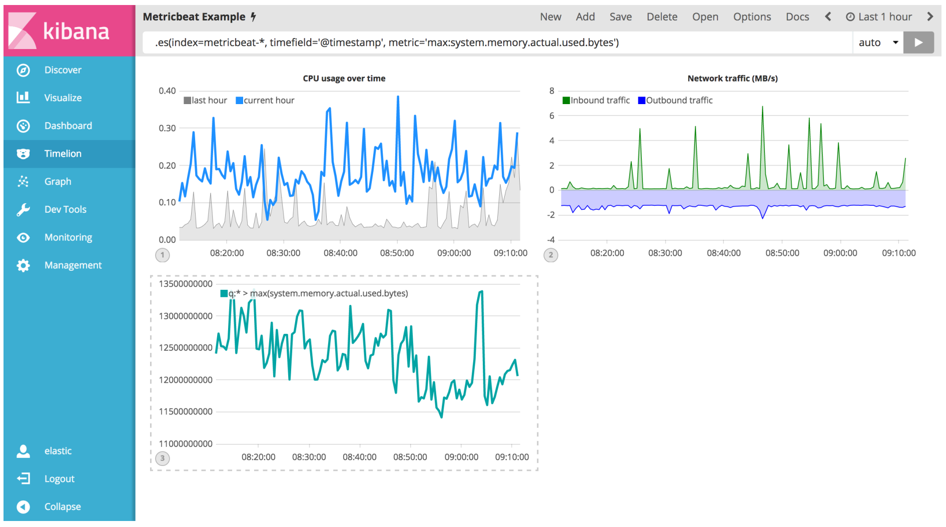


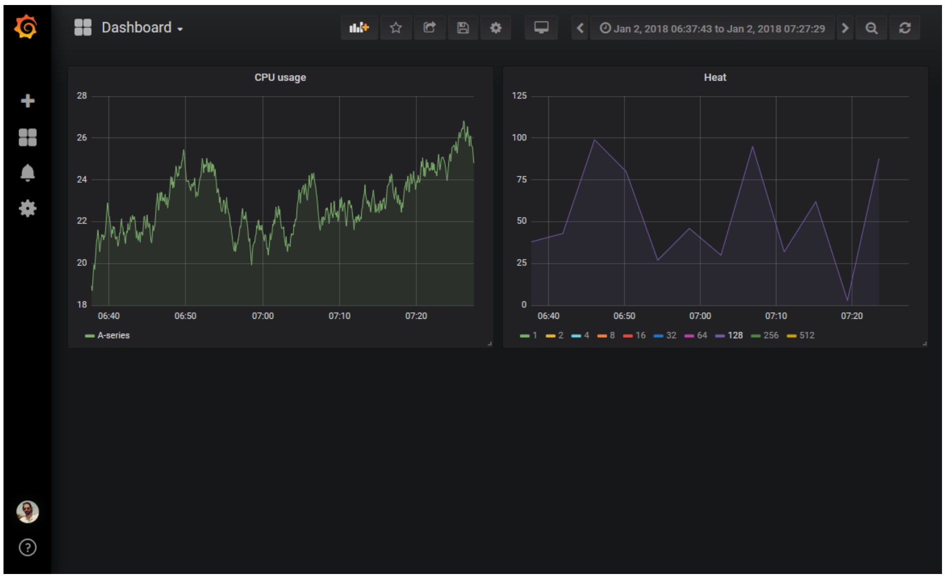
Figure 2.15: Web interface of earthquake historical events in California. The map view shows a selected earthquake as a red hexagon, previous earthquakes as gray diamonds, and events after the selected one as purple diamonds, all of these connected by a line that follows their chronological order. The line graph shows the magnitude and area of previous and later earthquake events in relation to the selected one.

Lastly, in [17] (Figure 2.15), Chourasia et al. presented a web application for earthquake event analysis where domain researchers and students can remotely explore the multi-variate time series present in this system. The implemented interface allows querying by timestamp for an exact year/day/hour/second in three ways: 1) as an input field where users can type manually into; 2) via a slider, for easy scrubbing; or 3) via increment and decrement buttons. The slider can be configured to different time scales (1 second, 1 hour, 1 day, 1 year, 10 years, etc...), so the user can seek the required timestamp at larger scales and then progressively reduce this scale to pinpoint the necessary timestamp with a higher degree of precision.



.es(index=metricbeat-*, timefield='@timestamp', metric='max:system.memory.actual.used.bytes')

(a) Timelion / Kibana



(b) Grafana

Figure 2.16: Open-source time series analysis dashboards displaying multiple time series charts in simultaneous. Time series that are part of the same query are shown in the same panel (shared-space), while time series from different queries are shown in separate panels (split-space). Panels are adjustable in size and position.

In the commercial and open-source space, there are multiple dashboard tools for visualization of time series metrics, like Timelion[36] (part of the Kibana[37] toolkit), Grafana[38] and Freeboard[39]. These tools are fully-fledged and adaptable to all kinds of input data, as well as to custom authentication schemes and user directories from Lightweight Directory Access Protocols (LDAPs), and provide extensive support for time series visualization using line graphs. These tools are advantageous for research teams when needing to quickly deploy a

[36]`https://www.elastic.co/blog/timelion-timeline`
[37]`https://www.elastic.co/products/kibana`
[38]`https://grafana.com/`
[39]`https://freeboard.io/`

web visualization for a proprietary data set, but these tools are incapable of creating flexible plots that integrate application-specific implementations [64]. Aspects like custom annotation logic with relationship validation and versioning stages, which are a part of the previously mentioned requirements (1.5), are not supported natively by these dashboards utilities.

Annotations

One of the aspects that was thoroughly explored during research is how annotations are commonly implemented in terms of interaction and representation in the visual space.

In [90], the authors benchmarked the effectiveness of tooltips and colored highlights over time series. Tooltips are typically used to append textual metadata to a point or region of interest, while highlights are used as a tool to attract the attention of collaborators to a region of interest, suggesting that the region has a connotation that does not need to be explicitly described [114]. Both techniques were implemented using mouseover and drag-and-drop interactions, and the authors concluded that both considerably enhance UX without compromising accuracy or clarity of data. The benchmark results show that tooltips have better task performance than highlights, which the authors attribute to the fact that users have an inherent preference towards textual instructions rather than implicit indicators of semantic through color encodings.

In *TSPad* [40], as a tablet-PC application, annotations can be created by sketching a shape around the intended area, as shown in Figure 2.12b. While this free-form shape annotation is an interesting approach in the visual angle, it increases complexity of storage and presentation without providing much benefit over tooltips or highlights. In addition, such shapes result in unstructured or binary data that automated systems can only interpret through the use of graphical parsing algorithms. *TSPad* also displays a list of annotations contained in the displayed data, so users can quickly jump to an annotation in the data by clicking it on the list. However, annotations only contain a free-text field, and no structured fields nor a parent category that matches a common taxonomy. The usage of text-only annotations comes with a few disadvantages, which were already discussed in this dissertation (2.2.2).

In the platform proposed in [12], annotations can match either the tooltip or highlight paradigms, and contain both a free-text field and fields such as category or type, public/private visibility, and affected data points. The visibility field enforces a read-write access permission policy that cannot be defined granularly on a user-by-user basis. Annotations that have not been seen by other users are rendered in a different color to draw attention. However, hovering the mouse over any annotation will automatically expand it, which can provide a bad UX when observing a time series window with multiple annotations.

All of the above-mentioned open-source dashboard tools for time series visualization support annotations following the highlight paradigm (Figure 2.17). Point annotations are drawn as vertical lines and region annotations are drawn as rectangles that occupy the full vertical area of the chart. Some of these tools only interpret clicks and drags to create annotations when a keyboard key is held, so users do not create annotations accidentally while navigating the time series data.

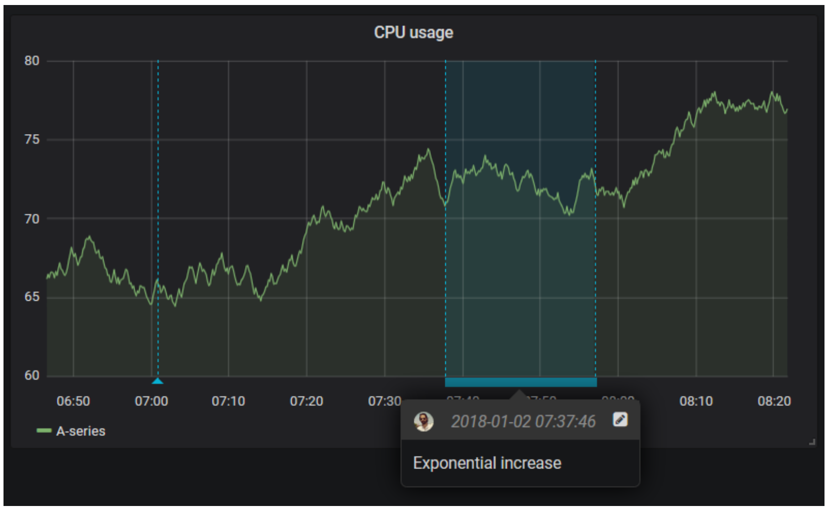


Figure 2.17: Annotation encoding in Grafana. To create an annotation, the user can click or drag the mouse while holding the *Ctrl* button, drawing a vertical line or a rectangle over the intended area.

2.5.3 Technological survey

In [115], the author presents a visualization of energy statistics using web-based frameworks like Google Charts[40] and the currently-abandoned IBM Many Eyes. *WebPlotViz* [16] is a web-based tri-dimensional visualization application built with WebGL[41] and three.js[42].However, WebGL frequently has high memory requirements when displaying high amounts of rendered data points, taxing on the machine resources of the end-user.

In [17], the presented web application for earthquake event analysis uses open-source JavaScript libraries to display charts and the Google Maps API[43] to render earthquake events in a map view. The observed interval of time series can be changed with a slider, which is implemented with noUiSlider[44].

Although the studied visualization tools use a variety of JavaScript toolkits for plotting and rendering charts, it is not immediately clear which of the available toolkits is more flexible and performant. In [117], the authors tested multiple visualization toolkits commonly used in both academic and commercial environments, such as Google Charts, JIT[45] and Protovis[46]. The authors concluded that Protovis was the most versatile tool out of the tested ones. However, by the time this research was published in 2012, Protovis had already been discontinued by their maintainers one year prior. The performed tests did not include other visualization libraries that were widely-used at the time such as D3[47] (from one of the authors of Protovis).

[40]`https://developers.google.com/chart/`
[41]`https://www.khronos.org/webgl/`
[42]`https://threejs.org/`
[43]`https://cloud.google.com/maps-platform/`
[44]`https://refreshless.com/nouislider/`
[45]`https://philogb.github.io/jit/`
[46]`http://mbostock.github.io/protovis/`
[47]`https://d3js.org/`

Both [53] and [5] implement a web application using Adobe Flash[48] plugin technology to display real-time physiological streams from EEGs. The performance of Flash was only compared with other third-party plugins like Java applets and Microsoft Silverlight[49], and ultimately Flash was preferred due to its popular use. Modern frontend technologies that leverage HTML5[50] and open-source JavaScript visualization libraries enable the application to be cross-browser compatible in both desktop and mobile devices without requiring pre-installed third-party plugins, but these were not discussed. HTML5 was mentioned, but ultimately discarded as an option due to it not being a well-established standard by the W3C[51] at the time the authors were developing their applications. Nevertheless, building applications using proprietary technology like Flash is widely unadvised due to vendor lock-in, where the owners of the technology cannot feasibly ensure future-proofness and continued maintenance. In fact, Adobe has announced that Flash will be discontinued in 2020 [116].

[48]`https://www.adobe.com/products/flashplayer.html`
[49]`https://www.microsoft.com/silverlight/`
[50]`https://www.w3.org/html/`
[51]`https://www.w3.org/`

CHAPTER 3

Proposed Model

This chapter presents an in-depth blueprint for the implementation of a distributed platform for time series analysis and annotation. The model presented here describes how data should be represented throughout the application and which techniques should be used in order to optimally store and query the data, without being restricted to a domain or to a specific software tool. In this manner, this model can be used as a specification for implementing a collaborative analysis solution for massive time series data sets while providing enough margin for other potential prototypes to be developed while using different tools and frameworks from the ones picked for the prototype described in the next chapter (4).

As part of the previously outlined work requirements (1.5), the intended platform should iterate on existing software models for time series annotation by integrating standard practices in the software industry and bleeding edge methods from the Information Visualization, Information Security and Distributed Systems domains. Based on the research findings in the previous chapter, there already is a good starting point for choosing the most appropriate visualization and storage tools to fulfill these requirements. In this chapter, the first section leverages this groundwork to describe how annotations, time series and other entities should be modeled throughout the entire application stack. The second section lists what protocols or systems should be used for storage of each entity. The third section describes the various components that should compose the architecture of the platform, and how these components synergize and communicate with each other. Finally, the fourth section illustrates the interface and design paradigm of the platform using technology-agnostic wireframes.

3.1 Data model

To fully express and organize a collaborative knowledge base of series and annotations, a set of entities is commonly used to describe the data set and all the different stateful objects that the system requires. In order to better understand the proposed architecture and workflow, these entities will be described first in this section.

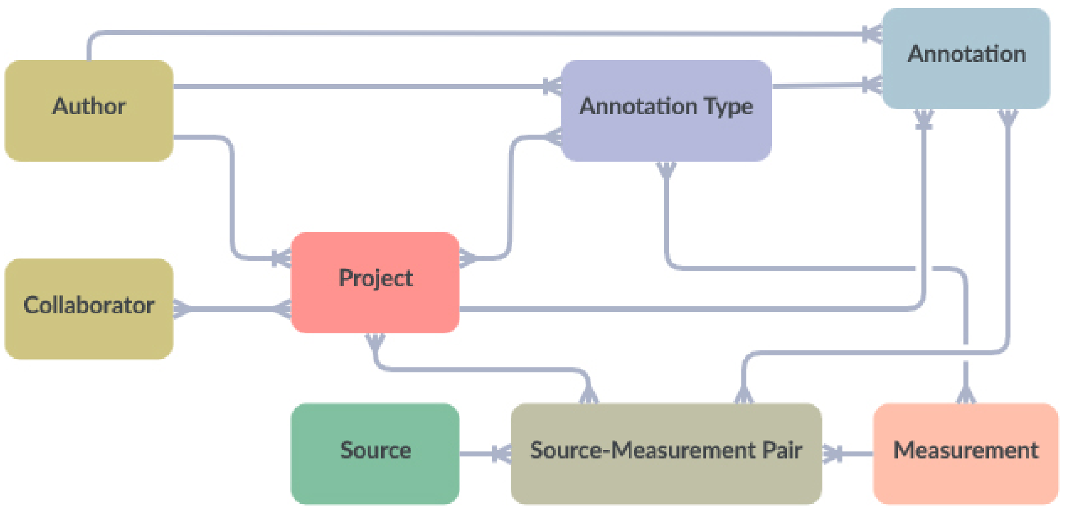


Figure 3.1: Relational diagram of entities

Throughout this dissertation, the term *ontology* is used to describe the data that composes the knowledge base or data-bank in the system, which corresponds to all of the data that is generated from user input (i.e. annotations, annotation types, user profiles, authentication sessions and revisions) combined with all of the metadata that is derived from a series data set (i.e. measurements and source devices), excluding the series points and values themselves. These entities should be homogeneously represented across every component of the application. For example, the data structure of annotations in the client's dashboard, including all of its fields, should be consistent with how annotations are stored internally. In this way, collaborators, system maintainers and automated services (such as machine learning models) can interact with the system in a uniform manner while selecting the most optimal interface for their use case, be it a web interface, a CRUD API through HTTP clients, a WebSocket API[1] through TCP sockets, or a database client interface that is directly connected to the existing data stores.

Figure 3.1 shows how ontology entities relate to one another. This same anatomy can be directly transposed into a relational database of choice by adapting the entities to tables. While projects, annotations and annotation types have a set of users related to them, in the form of authors and collaborators, data sources and measurements do not. This is because sources and measurements are generated at the system level and based on the input time series data. Additionally, while this diagram separates *Author* from *Collaborator*, when stored in a RDBMS these two entities can be unified under the same *User* table.

3.1.1 Time series

Much like how time series are commonly modeled in the studied State of the Art solutions (2.2.1), time series in this model should be represented as a XY coordinate point that maps a timestamp in the X axis, using the ISO-8601 standard in the UTC time zone, to a value in the Y axis. This value can be: a *number* (or a *continuous* type) for series that contain a

[1]`https://developer.mozilla.org/en-US/docs/Web/API/WebSockets_API`

numeric value, usually as a double-precision floating point; a *state* (or a *boolean* type) for series that represent how the measurement varies between active and inactive conditions over time; and a *string*, for series that contain an unstructured free-text value.

The input series data set is assumed to contain points from different series and from separate sources. The term *measurement* is used to describe a specific series and all of the metadata that distinguish it from other series, such as a name, a color, and an indication of its data type (i.e. number, state or string). Furthermore, the term *data source*, or *aggregator*, is used to describe a series or measurement origin device. Each source can measure multiple measurements, and each measurement can exist in multiple data sources. As a result, every series data point is better identified by the source-measurement pair it was collected from. If the input data set does not contain any information regarding point of supply or does not distinguish separate measures, then the platform can be configured to accommodate the entire data set within a single data source or a single measurement respectively. Measurement and source names are used throughout the entire platform as a public identification of these, but not as a primary key for system-level identification. Users should have the ability to customize these public names, as well as measurement colors, in order to make measurements and sources easily discoverable by all users of the application regardless of the names provided by the input data set.

3.1.2 Ontology

As previously stated in subsection 2.2.2, an optimal annotation model should be designed in order to be readable by both human curators and machines, where analysts can visually interpret and edit annotations while automated tasks can parse them for the purposes of search, pattern detection and data mining. The annotation data structure should have the minimum amount of domain-agnostic, indexable fields that makes it applicable in any platform and with any data set.

Therefore, the proposed annotation model can be expressed as in (3.1), where t is a timestamp in the ISO-8601 format and in the UTC standard that is not lower than the minimum timestamp nor higher than the maximum timestamp available in the series data set; t_1 is a starting point where this annotation becomes functional in its association with the series data set; t_2 is an ending point where this annotation is no longer in effect; p is the parent project or scope of this annotation; c is a parent category or type of this annotation; u is the author of the annotation; txt is a free-text field that contains all of the data that cannot be structured, like a set of additional notes to share with collaborators; s is the series, identified by their unique source-measurement pair, that this annotation is associated with; and n is the maximum count of annotated series. All of these fields, with the exception of txt and the s series-set, are mandatory so as to ensure a ubiquitous and universal representation of annotations.

$$A = (t_1, t_2, p, c, u, txt, s_1, s_2, ..., s_n) \tag{3.1}$$

The starting and ending points can be set to the same timestamp, fundamentally representing a point annotation instead of a region. Consequently, the annotation notation can then be shortened to be expressed as in (3.2), where t represents the single point in time where this annotation is contextually active.

$$A = (t, p, c, u, txt, [s_1, s_2, ..., s_n]) \tag{3.2}$$

The *category* or *type* field should correspond to an annotation type entity that is taken from a global repository of types. Because any annotation has to be associated with a unifying catalog of semantic, analysts can easily find similar knowledge in different series and segments. Collaborators can quickly create annotations and associate types without being forced to write an unstructured commentary, so usability is considerably improved. Moreover, annotation types can have additional metadata beyond a public identifier or name. A color code can be included in order to make annotations more visually distinguishable when shown along with time series. Furthermore, a set of constraints can be defined in order to restrict which annotations can be set with the type in question. These constraints can limit the annotated measurements and segment of time (i.e. either a point, a region, or both).

One of the key differentiators of the proposed solution is the ability to specify a set of annotated series, limiting the scope of the annotation within the specified segment of time to only a subset of the series that are available in that segment. This series-set can also be left empty, denoting a global annotation that represents a universal event in a specific segment that is not associated with any series, becoming identical to the annotation model of existing solutions. In this manner, annotations are versatile in the way they connect with series and more adaptable to realistic situations of note-taking and commentary.

It is still necessary to introduce a separation between different annotations at the contextual level in order to avoid information entropy and visual clutter, especially when these exist in the same segment and affect the same series. For this, the proposed model enforces that all annotation activities are contained within projects. A project establishes a scope of analysis and a specific annotation goal over a subset of the available series, limits the type of annotations that can be used, and lists the specific users that can contribute. In other words, any user can create an annotation project by setting up a query over series data, a repository of allowed annotation types, and a set of collaborating users. A project can optionally include a free-text field that instructs collaborators on a set of guidelines.

Finally, a user entity contains personal data that identifies a person, such as a unique username, a first and last name, and a profile picture. Ideally, this entity should match a collaborator that is already identified in other information systems within the organization, so all of these fields should be automatically imported to the platform. If these systems do not exist, the platform should directly request this data from the user during the sign-up process, along with a mandatory password field (3.3.3). In addition to this, each user can only update their corresponding profile data.

3.2 Data Management

Based on the listed research (2.3.2) and on the proposed data model for annotations (3.1.2), the choice of a relational data store would be beneficial in this platform, since annotations contain strong relationships with user data, parent projects, parent annotation types, and annotated series. Other database systems like CDBMSs would facilitate the deployment of a partitioned and replicated database, where the workload for queries and writes could be distributed and processed in parallel, but this would constitute a trade-off in strong *consistency*-handling and relationship fetching, which is a common scenario in the proposed application. With this, a RDBMS should be used to model the complete ontology. As for the time series data set, the researched benchmarks suggest that modern TSDBMS that use an LSM tree as backend deliver the highest performance while also providing built-in mechanisms that lower the impact of the data set on disk, promoting higher scalability (2.3.1). Thus, a TSDBMS should be used to contain all series data points.

There are multiple approaches in how series and annotations can be stored and linked with each other, but the most performant approach will likely rely on a polyglot persistence model that leverages the most optimal data storage tool for each data type (2.3.1). At the architectural level, a granular framework for data storage [71], [72] means that the overall traffic load and persistence requirements are distributed over multiple independent units of data storage. Therefore, granular models are naturally more scalable and provide higher *availability* of data. Data granularity can be employed by separating data based on their type or on any arbitrary rule set, or by partitioning a database using mechanisms provided by the database technology.

3.2.1 Persistence model

For the purposes of finding the most performant architecture for querying and storing massive amounts of series data and associated annotations, a few studies were made on how different persistence models fit with the data model described in the previous section (3.1). First, the minimum required queries were outlined. These queries represent a set of common searches that users will realistically perform in the system for the purposes of analysis, and that the system should be able to perform with as minimum workload as possible and without bottlenecks. They are as follows:

- Q_1 - get series S and annotations A over a range of time R;
- Q_2 - get series S that are annotated with annotations A;
- Q_3 - get series S that are annotated with annotation type T;
- Q_4 - get annotations A that belong to the project P;
- Q_5 - get series S that are annotated in the project P.

With this, the first approach was to follow a granular model where annotations and series are stored separately from each other. Series are stored in a TSDBMS while the ontology (i.e. annotations, annotation types and projects) is stored in a separate RDBMS. This approach is represented graphically in Figure 3.2.

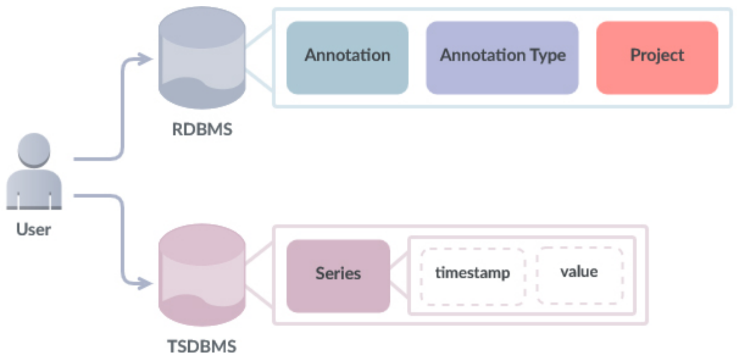


Figure 3.2: Data management approach A.

For Q_1, the user must request the TSDBMS for S within R, and then request the RDBMS for A within the same R. Because both queries are independent and do not require the results of one another, these can be performed asynchronously, distributing the overall workload between two databases without affecting each other's performance in any way.

For Q_2, the user would have to know the source-measurement pairs that were annotated by A beforehand, either by having the exact list of annotations and their annotated pairs or by fetching these from the RDBMS and waiting for results. Once the user has the exact measurements and the range of time of A, there is sufficient information to request for S in the TSDBMS. This can lead to a bottleneck where the queries are performed in a consecutive manner, and a failure or timeout in the ontology database would mean that there is not enough data to use as conditions for a query to the time series database.

For Q_3, the user must request the RDBMS for A whose parent type is T, either by having the exact list of primary keys for each type or by querying for types with a set of predicates. If the user has no prior knowledge of T, when querying these in the RDBMS the user can perform a join-fetch for the annotations table, reducing the need to perform two individual and consecutive sub-queries to the same database. After that, the user then proceeds as described for Q_2, inheriting the same bottleneck.

For Q_4, much like in Q_3, the user can perform a query to RDBMS for A while joining the projects table in order to find annotations of the project P without leading to any bottlenecks.

Finally, Q_5 is essentially a combination of Q_4 with Q_1, where the user queries the RDBMS for annotations joined with the projects table in order to find the measurements annotated by A, and use those along with the timestamps of A to fetch the matching series S in the TSDBMS, leading to the same bottleneck as in Q_2 and Q_3.

This first approach has the advantage of keeping the data separated logically, so that versioning these is a more streamlined process. However, as it can be observed from these results, the majority of the queries will require multiple data-types in simultaneous, and this approach struggles from not having direct associations between time series and annotations in its architecture. Additionally, the various locks for results will eventually translate into a longer delay for the end user, which will only increase as the overall data set grows.

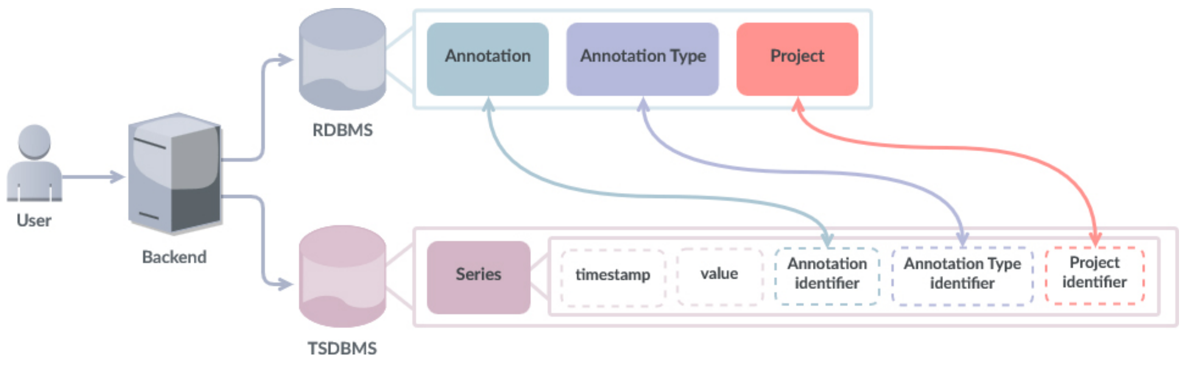


Figure 3.3: Data management approach B.

The second approach was to iterate on the previous approach in order to solve the listed caveats by inserting annotation identifiers, as well as their respective annotation type and project identifiers, as part of the metadata for each of the annotated time series points. Each metric is labeled with a list of tuples *{annotation ID, annotation type ID, project ID}*, as shown in Figure 3.3. Because the data linking is custom-made and not natively provided by the database tools, there must also exist a central unit that acts as a message broker between the requesting user and the available databases. This central backend or broker unit enforces the data access logic between all storage units by dictating the queries that can be sent and imposing where data is written, abstracting the real location of the data away from the user.

Under this new approach, the query Q_1 is identical to the one in the previous approach, which is already optimal. For Q_2, provided that the user knows the exact list of annotations A and, more specifically, their respective IDs, then the user can now query these IDs directly in the TSDBMS. Similarly, for Q_3 and Q_5, provided that the user has a list of IDs for annotation types T or projects P respectively, the user can also query these directly in the the TSDBMS. Q_4 however remains unchanged in this approach. In other words, this second approach completely mitigates the need to wait for results from a RDBMS in order to query the TSDBMS, avoiding the previously mentioned bottlenecks. By taking advantage of the TSDBMS querying by both timestamp and labels, queries Q_2, Q_3 and Q_5 can change from needing up to three sub-queries to just one.

However, and as mentioned before, this approach requires for a central backend to manually implement all writing logic that affects both series and ontological data, such as inserting or updating an annotation in the RDBMS and propagating these changes to the TSDBMS. Between the moment where the ontology is updated and the series are updated, the entire system sits in an *inconsistency window* and is *eventually-consistent*, which means that at some point in the future the system will converge on a synchronized state if no more updates are sent. For example, the user could perform Q_1 while an update to A was in progress, and even though the annotations sub-query to the RDBMS could return an up-to-date A, the series sub-query to the TSDBMS could ignore series that were recently annotated by A but whose IDs were not yet propagated as a label, resulting in false-negatives. Even if this duration is infinitesimally small, it is not feasible to ensure a perfectly atomic, *consistent* behavior when

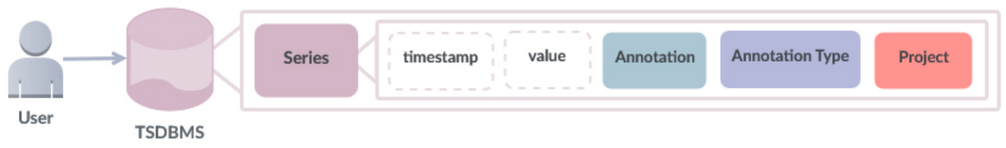


Figure 3.4: Data management approach C.

dealing with two independent database management systems. Moreover, in an environment where a large amount of write requests should be handled simultaneously, the central backend has to ensure that these requests will not overwrite each other nor cause conflicting writes that affect the *consistency* of the two databases. The proposed solution for this is to rely on constraint validation, ACID transactions, and an optimistic locking mechanism (3.3.8).

To avoid this added complexity and the requirement of manually managing the *consistency* of the system between two different data structures, all annotation data could be moved directly to the TSDBMS, avoiding the need for a separate RDBMS as shown in Figure 3.4. All necessary data is stored in the time series database, so that each series contains the full data set of annotations, types and projects, instead of just including their IDs. Because every series point can have multiple annotations, each point is labeled with an array of tuples *{annotation, annotation type, project}*. Since global annotations are not associated with any metric, these could be stored as part of an independent time series with a unique key *none* that distinguishes it from authentic series. All of the advantages of the RDBMS for modeling relations between annotations and other active entities in the system are lost, but the system is *consistent* for as long as the TSDBMS alone is *consistent*.

For Q_1, it is sufficient to simply query for S within the range R in the TSDBMS, since S already contain A on each series point. Contrastingly, for Q_4 the query has to essentially filter every series point that contains the project P and return the corresponding annotations in each point. This makes queries very unwieldy and harder to manage, since the query has to scan through all series points and all of the tuples in each, leading to a considerably lower read performance. All other queries remain essentially the same as in the previous approach, with the exception that the user no longer has to have prior knowledge of the IDs of A and can instead query any annotation data directly in the TSDBMS using any predicates other than ID equality.

The latter approach leads to a high *consistency* and high *availability* model that is optimal in four out of the five queries listed. However, every other query that requires annotations, types or projects directly, irrespective of their association with the series data, would lead to a considerably lower performance than previous approaches. Because the ontological data is not separated logically from the actual series, and because the actual metadata of annotations, types and projects is fully present in every single annotated metric, then any update would have to be propagated through all of these points, potentially having to cover hundreds of thousands of points per update. As the platform scales and the knowledge corpus grows, this operation becomes exponentially more resource-heavy and time-consuming. In addition, attempting to apply versioning of data that is spread across several points would be a very

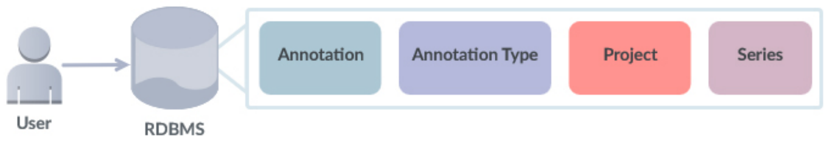


Figure 3.5: Data management approach D.

cumbersome, ad-hoc process. Due to these major caveats, the second approach remains the preferable model.

As a final approach, instead of moving the ontology to the TSDBMS, the time series data could be moved to the RDBMS as shown in Figure 3.5, emulating the architecture of existing time series annotation software. Like the previous approach, this removes granularity and avoids the need to manage *consistency* between two data stores. The advantages of TSDBMS, such as better capabilities for querying and aggregating time series, are lost in the process, but in exchange the strong relational model can be used to link annotations, annotation types and projects directly with series. Every time series is stored in a unique *series* table, set up with a bi-directional many-to-many relationship with associated annotations. Queries that fetch annotations, such as Q_1 and Q_4, and that fetch series, such as Q_2, Q_3 and Q_5, can poll from this relationship the necessary attributes that are available in second-level or even third-level relationships, such as parent projects or annotated measurements. Moreover, system *consistency* is easier to handle in ACID-compliant RDBMSs than in most of the existing TSDBMSs, as mentioned in the State of the Art chapter (2.3.1).

At a theoretical level, the latter approach could be the preferred schema for storing data while ensuring that all of the requirements for high *consistency* and fault-tolerance are fulfilled. However, it is not possible to theoretically compare this approach with the second approach and to immediately identify a clear winner in terms of performance, since many practical factors in the implementation of the selected TSDBMS and RDBMS, as well as varying data set sizes and requests per second, can influence the performance of each approach. In this dissertation the second approach, using PostgreSQL for annotation data and InfluxDB for time series, is tested in comparison with this fourth approach, using solely PostgreSQL for the entire data set. The results of this benchmark (5.1) showed that, for the selected technologies, the second approach is the most performant for storing and querying massive time series data sets by a wide margin, compromising a *purely-consistent* system in order to have higher scalability. However, there is some optimism and receptiveness toward the fact that RDBMS technologies will be further developed and, in time, may be able to compete with TSDBMSs in both read and write performance for massive time series data sets. If RDBMSs reach this point, one could consider that an optimal, *consistent* platform for time series analysis and annotation should be modeled after the fourth approach instead. In fact, even if the overall performance of RDBMSs is only slightly lower than the one found in TSDBMSs, this can be interpreted as a negligible difference in query speed and an acceptable drawback to gain atomic transactions and relational fetching over the entire data set.

Table 3.1: Summary of data management approaches, each row indicating if the respective approach is optimal for each query without leading to bottlenecks. Some queries are only optimal when forked, fetching the required data asynchronously and joining it afterwards.

	Consistency	Writes	Q_1	Q_2	Q_3	Q_4	Q_5
A	High	✓	✓(forked)	x	x	✓	x
B	Eventual	✓	✓(forked)	✓	✓	✓	✓(forked)
C	Low	x	✓	✓	✓	✓	✓
D	High	✓	✓	✓	✓	✓	✓

3.2.2 Versioning

In order to allow collaborators to revert unsanctioned changes, all entities related to the annotation workflow, such as annotation, annotation types and projects, should be versioned. Versioning of entities corresponds to the act of storing a reference to previous contents so they can be fetched and used to overwrite other changes that came after it. These references should be automatically created atomically within database transactions every time an object is created or updated, in order to maintain system *consistency*. A rollback of a revision can be implemented as a query over the target revision, collecting all of the contents that the entity had during that revision, and an update over the same entity using the retrieved contents. This will effectively rollback the entity to previous contents while generating a new revision. Measurements and data sources should also be versioned in a similar manner, since their public names can be changed by anyone in the platform. User data is not versioned due to the personal nature of this data and due to it not being part of the collaborative process of building a knowledge-base, as each user can only update their own profile information (3.1.2).

Versioning in the proposed platform is based on the full-table versioning model as described in 2.3.3. In this approach, each entity revision is a read-only immutable record that contains all of the data that the entity had during an update event, regardless if the corresponding field was changed in that specific revision or not. These records are stored in a revision table that is separate from the main table of current state, emulating an append-only log of versions. The revision table also contains boolean flags for each of the entity fields, indicating if the corresponding field was indeed changed in that specific version. While this translates to a high disk usage that will grow exponentially as more and more objects are created or updated, it will also translate into a higher performance in both revision querying and insertion, both of which happen during any entity updates and rollbacks.

Deletions are treated differently from the other CRUD operations: whereas creating or updating an entity results in the insertion or update of a record with the current state, as well as the insertion of a revision with the previous state, deleting an entity will not effectively delete the corresponding record nor create a revision representing the deleted entity, but rather flag the specific entity as *inactive*. In this way, entity relations are not immediately made obsolete and lost, but rather kept and associated with an inactive object. Every query made to the ontology database includes a predicate that filters out entities that are inactive, so even though the inactive objects and their relations are preserved, these are never sent to the client until re-activated. Recovering a deleted entity can be done by simply changing the *inactive* flag to false.

3.2.3 Caching

Along with the long-term persistence, it is also important to persist commonly used entities in short-term memory storage for faster access. This is especially critical for reducing latency between requests that perform multiple sub-queries of related entities, usually for validation purposes (3.3.6). For this, an in-memory key-value store such as Redis[2], Memcached[3] or Apache Ignite[4] should be employed. The benefit of leveraging these tools for short-term storage of data over a common RDBMS or any database system comes from the fact that these tools implement mechanisms to keep data based on a duration or expiration date and the amount of available memory in the machine. Caches also implement eviction policies based on First-In-First-Out (FIFO), Last-In-First-Out (LIFO) and Least-Recently-Used (LRU) principles, and others. These policies are followed strictly over time, but cached data may be evicted due to other conditions based on the available resources of the host machine or on the individual cost that one piece of data may have over others stored in the same cache.

Caching should be carefully exercised so as to not break the *consistency* of the system, and mainly applied to entities that have a low rate of updates. The cache should be setup with a LRU eviction policy, which means that as the memory space becomes more limited, the least frequently requested entities will be automatically evicted. This automated task contributes to a healthier level of self-preservation and orchestration in the global architecture. However, when a cached entity is updated in the database, the system should not update it in the cache but rather evict it, since this allows the cache data store to have a more realistic perspective of which entities are the most required instead of just the most changed.

A potential behavior that can break data *consistency*, however, is if the caching mechanism is called at a stage where an entity is about to be changed, erroneously caching an outdated snapshot. This can easily happen if a snapshot is evicted before or while changes are being propagated to the persistence units, but then a query called in parallel returns this same snapshot before the changes are fully committed, which will then prompt the system to cache this outdated result without knowing that an eviction is required. To ensure high *consistency*, evictions should only be called within write transactions or query-hooks, so as to be forcibly called during or immediately after the changes are effective and never before.

Under the proposed data model, there are a few entities that have a low chance of being updated. One of these entities is the source-measurement pair, which corresponds directly to a join-table of sources to measurements and serves as an identifier of a series in the time series database. Another such entity is the user profile, which can only be changed by the person corresponding to that profile. These entities, and other entities that follow a similar pattern in the platform, can be cached with minimal concern that the system will become *inconsistent*. In addition, these entities should have more memory space allocated for caching,

[2]`https://redis.io/`
[3]`https://memcached.org/`
[4]`https://ignite.apache.org/`

as other cached objects that have a higher chance of being modified and requiring eviction will have lower usefulness.

Along with specific entities, the queries that are performed over global catalogs, such as annotation types, source devices and measurements, do not change based on the user that is calling them, and do not have a high cardinality of query predicates. Therefore, it is acceptable to cache query results for these entities, corresponding them to a key that represents the sum of all query parameters. As per the caching philosophies expressed above, infrequent queries with regularly updated results should avoid being cached as much as possible.

Time series in analysis systems are usually immutable, receiving new data points over time but never changing these once inserted. Because of this, it would be reasonable to believe that caching the results of series queries is safe and will not break the *consistency* of the system. A time series point can be properly reduced to a unique key by combining its timestamp with its source-measurement combination. This key can work well in series queries for a single point in time and for a single source-measurement pair, but these are not at all frequent in the common workflow. Since the key-value data store that serves as backend to the cache can only perform lookups using a single key and not comparisons or approximations of keys, it is not possible to use these keys in a complex time series query. Furthermore, caching series points individually can lead to a high level of cache pollution without much added benefit. Thus, series points are not cached.

An alternative strategy could be to cache series query results directly while using the query parameters as the identifying key. However, when annotations are inserted or updated, the only way to assess which query results were affected is by iterating a list of keys in cache, obtaining the query parameters from each key, and determining manually if the query parameters match with the annotation's segment and annotated pairs, evicting them if so. This leads to some glaring drawbacks: i) a complete list of keys in the cache has to be managed and stored in memory separately from the query cache; ii) the wide variety of series queries that can be made to the platform is extensive, hence this caching strategy will lead to a high number of key-value members occupying the existing memory constraints. As such, series queries are also not acceptable for caching.

3.2.4 User tasks

When time series points are deleted, associated annotations should still be persisted so that the user can associate them to new metrics or permanently delete them at a later point in time. If the metrics associated with an annotation are deleted, or if the parent annotation type has been deleted, then this annotation should be flagged as *invalidated*, similarly to how deleted entities are expressed by flagging them as *inactive* (3.2.2). Annotations are kept invalidated until a collaborator can review it and update it in order to fit the new context. This responsibility is notified in the form of a user task to the target user, which corresponds to a collaborator in the project that is parent to the invalidated annotation in question.

A user task object is generated by a separate process of the processing layer, designated as *user task scheduler*. This scheduler is not triggered by a user request, but rather by a scheduler thread that checks for invalidated annotations on an hourly basis. When an invalidated annotation is found, a task that requests a set of target users to review it is stored in an in-memory cache database. With this, users must update the annotation so that the invalidated status is turned off, and the user task will be cleared.

3.3 Architecture

As it was previously suggested, an ideal platform for collaboration between peers is one where the knowledge base is located in a centralized system that is remotely accessible inside a network (1.4). This centralized system should employ a distributed architecture in order to handle high amounts of simultaneous access and updates, along with a visualization that displays the up-to-date state of all entities and is locally deployed at the client level.

The end goal of the proposed architecture is to implement both a frontend dashboard and a backend system that support the intuitive and interactive visualization of series data and annotations while maintaining high-performance over multiple operations in parallel. Under the CAP theorem, this architecture should follow the Consistency + Availability (C+A) model, prioritizing *Consistency* and *Availability* over resilience to network failures. Most if not all of the architectural decisions are agnostic to the data analyst's workflow, save for a few that prioritize analysis tasks instead of monitoring tasks. Within this analytical scope, the analyst's main use case is to go over historical data and gain insight over visible patterns by proposing and refining annotations, so under the PACELC theorem, an E+L architecture is chosen for time series, preferring low latency of queries over read-write *consistency*, while a E+C architecture is chosen for annotations, so that annotation queries and update proposals from collaborators should always iterate and overwrite the most recent data changes.

For this, every aspect of this solution's architecture is grouped within four different layers of abstraction: a visualization layer, a communication layer, a processing layer, and a persistence layer. Figure 3.6 shows an overview of the platform, where all of the separate components that implement the platform's operations are placed in a layer of abstraction based on their overall responsibility and relationship with the other components. From the perspective of a request performed by a user in the frontend application, the stack should be read in top-to-bottom order, where the user request will translate into a query, an insertion, an update or a deletion that is sent all the way down to the responsible databases in the persistence layer at the bottom. From the perspective of a response that must be sent back to the user, the stack should be read in bottom-to-top order, where the database processes a result for the requested read or write operation and sends it all the way up to the user in the visualization layer.

In the top-to-bottom order, the visualization layer is composed of a reactive web application that is designed with dynamic data in mind (3.4), and contains a set of units that allow collaborating users to login, manage their associated projects, query and annotate over series, manage the repository of annotation types, and change the public names and colors of measurements and data sources. Every request made in the dashboard or management UI is sent through HTTP to the communication layer below it. The most common end user will interact with the system primarily through the visualization layer, but users may also

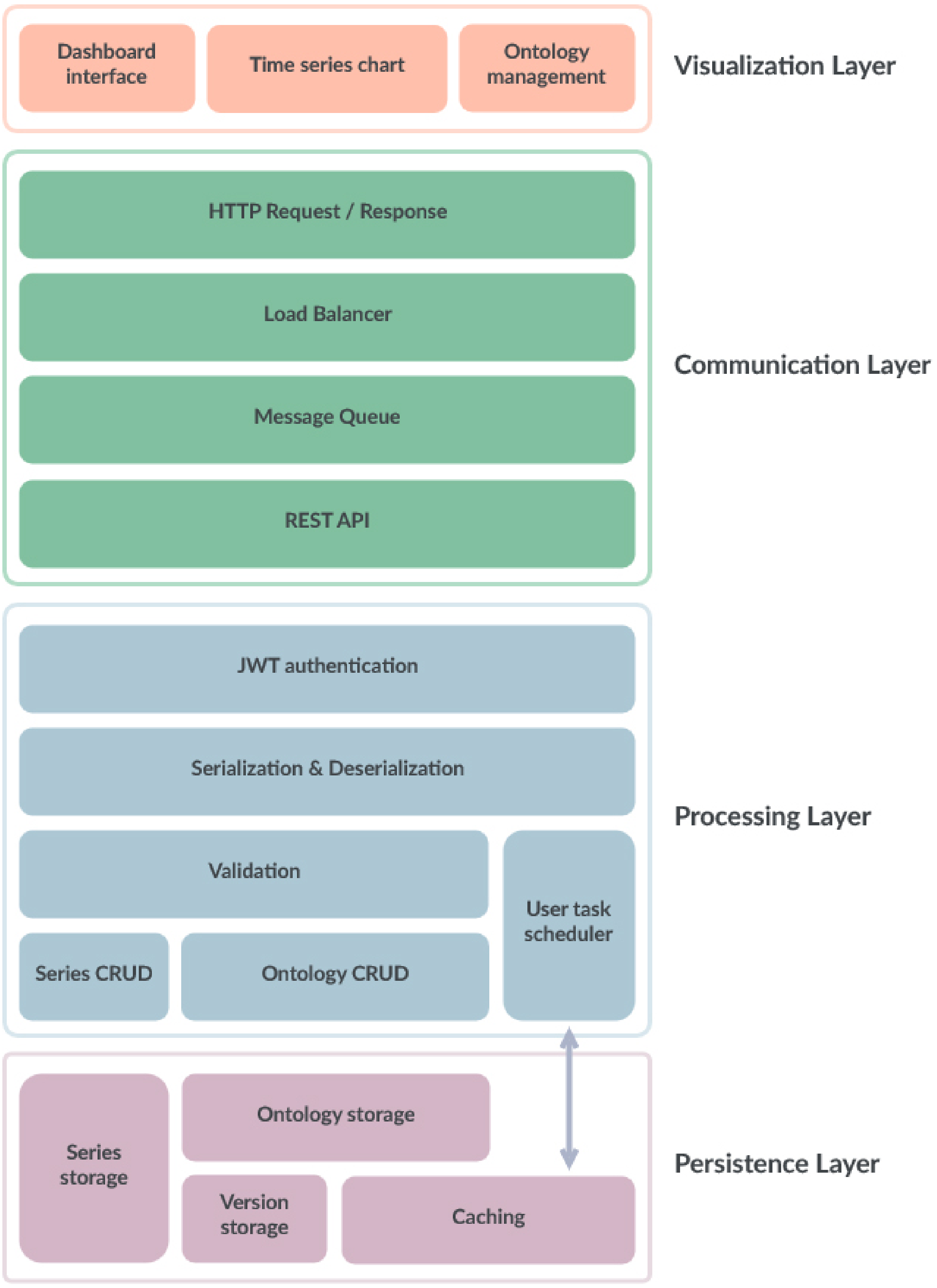


Figure 3.6: Architecture abstraction layers

send requests directly to the communication layer by using its REST API (3.3.2). Regardless of how the messages are encoded and how the API is implemented, this API is a vital key in enabling additional layers to be applied to the architecture stack, as the platform scales and evolves over time to accommodate further features. For example, a potential input layer could be used to parse new annotation types from an external dictionary, which would then be inserted in the platform by sending these through the communication layer, by-passing the visualization layer. Moreover, a machine learning model could export the annotations and series stored in this platform by querying these through the communication layer. While the deployment of automated import/export systems are not part of the scope for this dissertation, it is still important to build the architecture in a way that enables it. The communication layer is essentially a link between the end user and the processing layer, as the first component in the communication layer is the single point of entry for every request made to the platform. Without a careful architecture, a failure of this entry point and succeeding layers will render the system as unavailable (3.3.1). The processing layer is best viewed as a set of processing pipelines (3.3.4), where data at the input must be authenticated, read, validated, resolved as a valid query statement, and then converted into a response that is sent back to the end user. Finally, the persistence layer, which was already described in detail in the previous section (3.2), is responsible for storing, versioning and caching the overall ontology, as well as storing the entire series data set.

3.3.1 Load balancing

When running systems with high amounts of requests, there are essentially two main challenges to deal with in the operational perspective: orchestration to deal with traffic increase over time, and failure handling to guarantee as high of an uptime as possible. A distributed systems technique that is applied to the proposed architecture in order to ensure high *availability* and uptime is replication of the processing logic. Essentially, the processing layer, below the communication layer and above the persistence layer, is deployed over multiple servers/nodes.

To distribute the load, one could follow an approach that uses a single load balancer. In this approach, every request is intercepted by a load balancer, which will then redirect the request to one of the processing servers. The balancing policy is set with a Least Connections balancing policy, where requests are sent to the machine with the least amount of strain at the time. However, this approach means that if at any moment all of the deployed servers are struggling to handle requests and under strain, the load balancer will still continue to redirect more requests as it does not have the ability of keeping requests queued in place until the backend servers are freed up.

Instead, the proposed approach should leverage both a load balancer and a distributed message queue to keep the subsequent requests in a FIFO queue, as shown in Figure 3.7. Essentially, the load balancer is set with the same balancing policy, but will instead redirect all requests to a distributed event queue with the lowest number of queued requests. Event queues are replicated as well, not only so they can scale along with the increase of simultaneous requests, but also to provide a failover measure. All processing nodes subscribe to one or

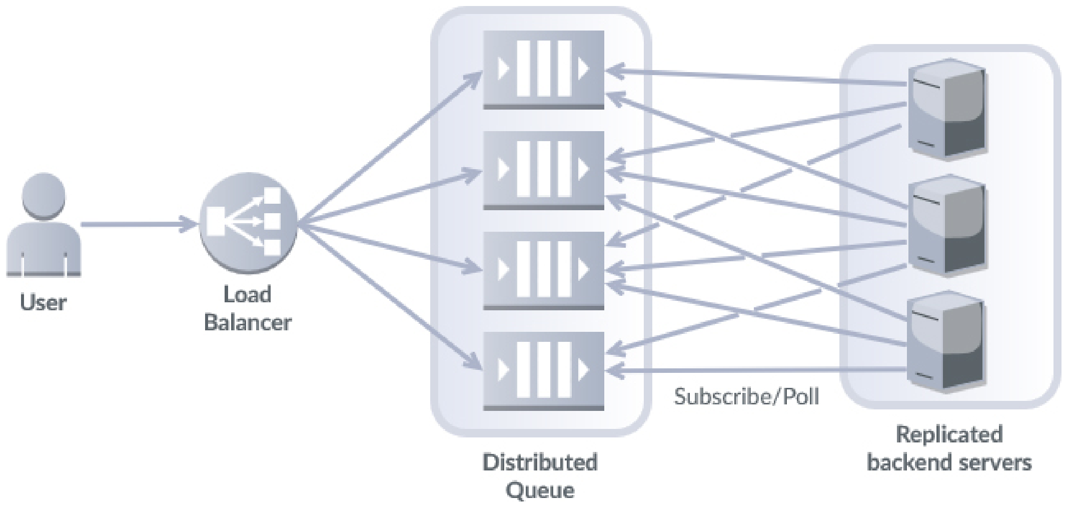


Figure 3.7: Load balancing architecture

more of the event queues, and when any queue is populated with requests and a node is free to process it, this node will poll the first event from that queue. This way, when all servers are under strain, the event queue will block the requests until a server is available to open a processing pipeline for it. This approach comes the added drawback of introducing an additional component and point of failure in the communication layer. If a failure occurs on all event queues, the system will not be able to recognize it and receive requests.

There are many open-source implementations of message queues that are mature and highly contributed to, such as RabbitMQ[5] and Apache Kafka[6], both of which contain automatic scaling mechanisms. These queues already handle failover quite well: if a request is sent at a moment in time where a server fails, the server will naturally not be able to poll that request from the event queue, so the request will continue to be in the queue until another server polls it. Moreover, if one of the servers crashes or fails while handling the request, the event queue can detect connection losses and put the polled request back in the queue, so that other servers may poll it. In order to not increase the overall delay in response handling when a server fails, this request is placed back at the head of the queue (instead of at the tail), making it the immediate next request to be polled by a failover server.

3.3.2 REST interface

The supported operations directly correspond to the basic functions of CRUD, and these are all made available through a REST API. Every request that arrives at the communication layer follows the HTTP protocol, and the corresponding responses are modeled as an HTTP response. The REST API is deliberately designed in order to ensure uniform usage and interoperability for all requests while following well-established conventions and recommendations for good API design. Every response is delivered with the appropriate status code, headers and media-types, and every request is modeled using the most appropriate method: *GET* for read-only queries, *POST* for insertions, *PUT* for revision rollbacks, *PATCH* for updates, and *DELETE* for

[5]`http://www.rabbitmq.com`
[6]`https://kafka.apache.org/`

deactivations. The data required for insertions and updates is sent in the request body, and is written in a structured markup format, like XML or JSON.

All *GET* queries receive their predicates as Uniform Resource Locator (URL) query parameters, allowing any user to easily specify a search criteria using the URL alone. However, because series queries require a more complex query object that cannot be easily modeled into an URL (3.3.5), the request body is used instead. Since the REST standard does not recommend the inclusion of meaningful request bodies for *GET* requests, which has led to a wide variety of software, like proxies, caches and other tools being built with that assumption, series queries are structured as a *POST* request.

GET queries are always idempotent and non-state-modifying. All non-annotation queries are paginated, returning smaller windows of data on every request that are faster to output. This is because the common use case does not typically require a listing of the entire catalog of entities in the ontology at the same time. Annotations queries, however, should still be returned non-paginated, as chunking the results can affect how annotations are represented in a chart. Instead, the window of queried annotation should only be narrowed by filtering these by their parent project and by their segment of time.

Users can specify how their results should be paginated through the URL query parameters *size* and *page*, where *size* limits the amount of results that will be queried for that request and *page* sets the result page number. The page number ranges from 0 to the total number of pages that a paginated query will output based on the specified *size* and on the total count of resulting entities. In other words, the total number of pages that will be generated matches the total count of entities divided by the specified *size*.

Relational fetching clauses can also be set for some *GET* requests, depending on the amount of second-hand data in the relationship graph of the queried entity that the user will also require. For example, if the user requires a list of annotation types and the allowed measurements for each, a boolean flag *fetchMeasurements* can be turned on for the annotation type query, returning the intended result without needing to perform extra measurement queries for each annotation type. Fetching related entities can be a costly process, and other searches should not have to unnecessarily load additional entities that they do not require, so this is made optional and turned off by default.

An essential design decision in this API is that the *PATCH* HTTP method is used for update requests. *PATCH* has a clear advantage over *PUT*, as it allows the user to specify only the attributes that require changes in the request body. There is no need to supply a complete snapshot of an entity, which the user would only have by fetching the entity beforehand.

Along with requests, the responses should also be interoperable and homogeneous, regardless of the entities that are being affected. All responses contain a payload, also serialized in a structured markup format, with two baseline fields: a *result* and a *result type*. These fields should provide enough information at the root of the response body for the user to be able to algorithmically parse it, even without knowing the initial context or request that prompted this response. The result field corresponds to the serialized output from a database query or statement, and the *result type* corresponds to a string that can either be “object” or “list”. If the *result type* is “object”, then the user can expect that the *result* field will contain a single entity, and if it is “list”, then the user can expect that the *result* field will contain multiple entities. A list response body will also include a set of numeric and boolean fields:

- *count* - the number of results that the current pagination window returns, being only equal or inferior to the specified *size* parameter on the request;
- *page* - the current page number;
- *total count* - the total number of entities that match the query criteria, or conversely, the sum of all *count* fields in all available pages;
- *total pages*, the total number of pages that the current pagination constraints will output;
- *has previous* - if there are pages before the current one;
- *has next* - if there are pages after the current one;
- *first* - if the current page is the first one in a stream of pages, or in other words, if $page = 0$;
- *last* - if the current page is the last one in a stream of pages, or in other words, if $page + 1 \geq total\ pages$.

Additionally, every serialized entity object should contain a baseline properties that indicates the name of the entity itself, which is called *object name*. This way, an automated system that has access to the response body, but not the surrounding context or request that triggered it, can use the *result type* field to easily interpret that the result is a single entity or a list of entities, and use the *object name* field to identify the entity or entities contained in the result field. The latter will allow the automated parsing task to know what remaining fields are available based on the entities inherent properties, as specified in the data model section (3.1). For example, if the *object name* corresponds to “project”, the system can immediately assume, without risk of read failures, that this entity will also have a *name* field, a *guidelines* field, and so on.

If a request leads to an erroneous response, then the HTTP response will be generated with the most appropriate status code for the error. For example, a request with invalid query parameters will return 400, which corresponds to the error *BAD REQUEST*, and a request with an invalid credential will return 401, which corresponds to the error *UNAUTHORIZED*. However, this schema for reporting errors is limited, as it does not inform the user what exactly about the request prompted the error itself. For example, under the *BAD REQUEST* status code, the error can correspond to an URL parameter being unexpectedly null or empty, to an invalid value in the request payload, or to invalid pagination parameters.

For this, the response body also includes two additional baseline fields: an *error* and an *error type*. The *error* field contains a string message that specifies what was the aspect of the request that led to an error. The *error type* field contains a unique key that identifies the different types of errors that can occur in the platform. Users can use the *error type* field to distinguish errors with more precision than what would be possible with the HTTP standard for status codes alone. Plus, the frontend application can also recognize the error and write appropriate custom error messages that are localized to the end user’s preferred language.

The available *error type* values are as follows:

- **null param** - a specified URL parameter cannot be null nor empty;
- **invalid param** - a specified URL parameter is not valid, and a set of allowed values is suggested in the *error* field;
- **at least one of param** - one of a group of URL parameters must be set, and these parameters are listed in the *error* field;
- **invalid field** - a specified field in the request payload is not valid;
- **invalid pagination** - the specified *size* and *page* URL parameters are not valid;
- **invalid date range** - a date and range of time, composed of a starting timestamp and an ending timestamp, is not valid;
- **unique field already exists** - a specified field in the request payload corresponds to a unique attribute of the entity and it already exists;
- **object not found** - the requested entity with the specified ID was not found. If the requested object exists but is personal to a specific user (i.e. projects and annotations they collaborate in), then any attempt by another user to access it will cause this error to be returned, instead of an "unauthorized" variant. This is a security measure that serves to conceal the existence of said confidential data to unauthorized users;
- **conflict** - an object was changed while performing the operation. This error results from a mechanism of optimistic locking, described in section 3.3.4;
- **wrong credentials** - wrong username or password. This error occurs only in login requests;
- **expired jwt** - the authentication session is expired, and the user has to re-login (3.3.3);
- **blocked request** - too many failed authentication attempts, which resulted in the IP address being blocked for one hour (3.3.3);
- **server error** - catch-all *error type* for unexpected errors that occurred internally. The maintainer of the platform should be able to assess the system through logs or health checks in order to fix this error.

3.3.3 Authentication

All operations in the platform, other than login requests, are protected and require user authentication. The authentication scheme proposed in this module varies based on whenever the user is imported from a directory information service that already contains an authentication process, such as an LDAP directory, or if it is created from scratch in this platform. If the platform uses the former, then the user record in the platform should not store a password field but rather use the authentication mechanisms available, sending a login request with a username-password pair directly into this service. If instead the platform uses the latter, the user should be prompted to set a password, which is then hashed with a function such as SHA [118] or BCrypt [119] and stored in the respective user record. This hash algorithm has to be strong in generating highly diverse hashes and deterministic so that the same hash is generated when the same input is provided. This protocol provides a security layer for passwords when stored in the database, as any maintainer or malicious user that gains access to the database records will not be able to determine the original password nor use the hashed password directly in an authentication request, since this hashed password would be interpreted as a raw password by the system and re-hashed, resulting in a new hashed string that would not match with the one stored in the database.

If a malicious user were to constantly send authentication requests with commonly used passwords or common sequences of letters and digits, it is possible that this user could eventually find a valid username-password combination. This attack would be slow and unwieldy to perform manually, but an automated task could be designed in order to send these requests in rapid succession, or even asynchronously. To avoid these dictionary attacks, every request is identified with an IP address and, after ten requests that attempt to use an invalid token or that provide wrong credentials, this IP address is stored as a blocked address in a cache data store configured with a one-hour expiration policy.

Once an initial authentication request has been successful, the platform should be able to provide the user with a token that grants access to all operations without needing to provide their credentials in subsequent requests. This is essentially the concept of an authentication session, where users are flagged in the system as logged in after a successful authentication, and are provided with a unique token that will identify them in subsequent requests.

This protocol of authentication sessions can be implemented in various ways. One approach would be to store the valid and non-expired session tokens in a database or cache from the backend side while sending a copy of it to the user on a successful login. When a request is made to the backend system, a query to the database or cache storage is made in order to compare the existing session token with the one provided on the request. However, this approach would lead to an extra impact on the database or cache storage, since each and every request mandatorily requires an additional query to these. In a distributed environment, where replicated backend systems receive multiple, simultaneous requests and treat them asynchronously, this can quickly lead to a severe workload on the databases, affecting the performance of other queries and statements in parallel and potentially timing them out due to exhaustion of the connection pool.

A better approach that avoids hitting the database or cache storage to validate the authentication session, and the one adopted by this model, is to generate tokens using a message authentication code derived from a cryptographic hash function and a secret cryptographic key. Much like the hash function that is used for passwords of non-imported users, the hash function that is used here should be deterministic, so that using the same input message and the same secret key will result in the exact same hash. However, unlike the hash function used for passwords, this message code algorithm should generate a hash that can be easily reversed back to the original input when matching it with the same secret key that was used to encrypt it. Moreover, performing direct changes over the hash directly without having access to the secret key should result in an invalid authentication code. There are many keyed-hash message authentication code algorithms that can be used for this approach, such as HMAC [120], RSASSA-PKCS [121], RSASSA-PSS [122], and ECDSA [123]. All of these protocols rely on a strong hash function, such as SHA-256, SHA-384 or SHA-512 [118], so the strength of the final hash depends solely on the size of the secret key.

A message authentication code algorithm can be used to generate a session token from a structured object that contains a set of unique and personal data, such as the user primary key, the login request date, and other metadata that identifies users and their authorities. This is essentially the concept of JSON Web Tokens (JWTs), where the set of data that is hashed corresponds to a key-value map serialized in JSON. The resulting hash is sent to the user, with the expectation that the user cannot reverse the hash since it does not have access to the secret key. Users will then append this hash to the Authorization header of requests as an identification of their session, which the backend can properly verify by attempting to decrypt it and read its metadata. If the computed result is not a valid object, then the token is considered invalid and the request is canceled.

Finally, because a user request to the platform can be tampered by a man-in-the-middle attack, providing an attacker access to the platform for as long as the session token is valid in the system, then without an expiration policy or any kind of orchestration mechanism at the service level, the session would have to be invalidated manually by the maintainer of the platform. This also fundamentally requires the maintainer to know that the session is compromised, which might not be assured. To solve this, the session token should be set up with an expiration policy, so that both authorized users and malicious users are forced to request another session after a set amount of time, which only authorized users that know their credentials can do. The selected approach of using message authentication code algorithms to encrypt structured objects enables an expiration policy to be easily enforced, by simply including an expiration date as part of the message input that gets hashed into a session token. When a token is received in subsequent requests and decrypted in the backend, the expiration date is obtained and compared with the current date, so that if this expiration date is before or equal to the current date, then the token is considered expired. Users may hold on to a token for a long time during a continuous work session, so it is important that this token's expiration date is constantly refreshed so that the user is not repeatedly asked for their credentials. For this, every response provides a new session token with an updated expiration date, which the user then saves locally (replacing the previously used token) and sends with the next request.

3.3.4 Processing pipeline

Between the communication layer and the persistence layer sits a backend server (3.2.1), responsible for implementing the available CRUD operations for all active entities. When an HTTP request arrives in this backend, a worker thread is opened to validate the request, convert it to a set of instructions that is sent towards the persistence layer, receive its results, and convert these results to ones that are readable to the user while concealing extraneous data. Thus, this worker thread corresponds to a stateless message broker or processing pipeline that transforms an input request into a relevant output.

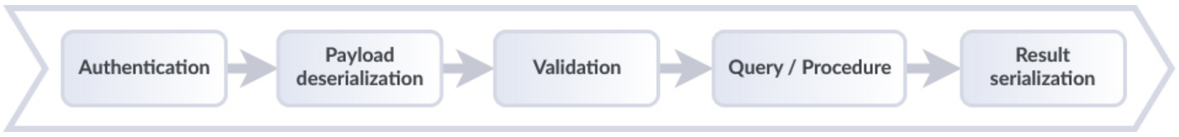


Figure 3.8: Processing pipeline for queries and insertions

Figure 3.8 shows how a common processing pipeline is implemented for any of the CRUD operations. First, the request is authenticated, as described in the previous section (3.3.3). Then, in case of read-only queries, the URL parameters are parsed, and in the case of other operations, the request body is deserialized. Since the request body sent through the REST API is encoded in a markup format like XML or JSON, as mentioned in section 3.3.2, this data can to be deserialized to binary in order to be usable in the backend system. The deserialization process itself ensures that the request follows the specification, effectively validating the payload in terms of data types. When deserialized, the URL parameters and request body fields are further validated to ensure that the request follows the data model specification and integrity (3.1). Once all validation checks pass, the query or write request is sent to the database, locking the thread for results. Once the database delivers a successful response, the resulting entities are serialized in a markup format and the HTTP response is sent to the user, putting these serialized results in the response body.

3.3.5 Read validation

Users should have the ability to freely query over the complete data set using as many filters as necessary. However, a query statement is dependent on the way the data is organized and named, so the user would be required to have prior knowledge of this data model. Additionally, queries should be properly validated so the user can create queries within a consistent structure that uniquely apply to this platform's ontology logic, and without the risk of generating erroneous results or of allowing unbounded parameters that contain update or delete statements, constituting an injection attack. Finally, it is important for the query to be technology-agnostic and structured instead of an unstructured free-text query statement written in the language of the selected database. With a structured object or a set of parameters, both the backend and the frontend application can traverse a valid query in an expected way, so parsers and visual representations (i.e. a query criteria builder) can be implemented. If the full data set is changed or migrated to different databases, the system itself can read and translate these queries as necessary.

The proposed model introduces a central backend that acts as a mediator between what the user sends and what the database returns (3.2.1), so the processing pipelines in this backend can be leveraged to properly validate and enforce a set of structured URL query parameters or a structured query object. These act as an abstraction layer for queries, and should be built in a way that conceals extraneous data but do not limit the query capabilities of RDBMSs and TSDBMSs. For ontology queries that are sent to a RDBMS, the already described URL query parameters in the REST API (3.3.2), as well as the deserialization stage of processing pipelines, are enough to uphold database-agnostic, structured searches that fit the data model. However, time series queries sent to a TSDBMSs should have the ability to nest and combine filters in a flexible manner. Multiple filters can be aggregated through *AND*, *OR* or *NOT* operators, and can be applied to all of the fields that exist in series points, such as timestamps, numeric, state or string values, measurements, data sources, and annotations (3.1.1). For each field, the available operands and operators can vary: while numeric values can be compared with user-specified values using a comparison operator that supports < (less than) or > (greater than), other fields can only support equality operators. The selected field also restricts the viable user-specified values: numeric-value fields can only be compared with numeric values, state-value fields with boolean values, and string-value fields with string values. Finally, time series can be labeled with annotation IDs and their respective type IDs and project IDs (3.2.1), so queries should naturally support filtering by these fields as well.

For this, the proposed platform uses a structured, serializable time series query object that mirrors the mentioned nesting capabilities while introducing type safety and guiding users in how time series are modeled. The user can send a *POST* request with the intended query contained in the request body, serialized in JSON or XML. Then, as part of a processing pipeline's validation stage, the query object is parsed, its data-types are checked, and its fields are validated according to the rule-set above. By having a human-readable and machine-interpretable structure, the backend can directly translate a valid query object into an optimized query statement in the language of the selected TSDBMS.

Ideally, this series query object should allow users to directly specify the pairs of source-measurement combinations, regardless of how measurements and sources are actually modeled in the TSDBMS. The query object should also contain a set of filters that are applied globally to all source-measurement pairs, as well as a set of filters for each individual pair. Moreover, to allow users to specify sub-queries with any number of nested filters, the filters themselves should contain a pointer to child filters, which can be recursively traversed. Moreover, the available field names should be: *numeric value*, applicable to continuous series; *state-value*, applicable to boolean series; *string-value*, applicable to string series; *annotation*, which should indicate an annotation ID, and will limit the query for series that are annotated with the corresponding annotation; *annotation type*, which should indicate an annotation type ID; and *project*, which should indicate a project ID. Along with this, the operators should also be limited to the chosen field name, so that only expressions with the *numeric value* field name should be allowed to use a numerical comparison operator.

3.3.6 Write validation

In the case of insertions or updates, the validation step usually involves the verification of a set of field constraints (i.e. character limits, data types, string formats) and the querying of relationships to guarantee that the user-specified data is valid based on the pre-established relationship constraints, a few of which were already described in section 3.1. For example, an annotation type can be updated to prohibit child annotations from annotating certain measurements, at which point the validation stage will have to query the child annotations in order to confirm that none of them are annotating the newly forbidden measurements.

For an annotation A, a parent annotation type T, a parent project P, a measurement M, and a source-measurement pair SM that combines any source with M, the complete list of relationship constraints that must be validated are as follows:

- P allows T, both being parents of A;
- A is annotating SM, which P allows;
- A is annotating SM, hence is annotating M, which T allows;
- A is annotating a segment of time (point or region) that T allows.

When attempting to remove certain relationships, all of the constraints above already regulate the removal attempts in order to ensure that these constraints are always true. In the same order as the constraints listed above, their respective corollaries are:

- P cannot revoke SM if at least one of A is still annotating SM;
- T cannot revoke M if at least one of A is still annotating SM, hence annotating M;
- T cannot revoke a segment type (point or region) if at least one of A is set with it;
- P cannot revoke T if at least one of A is still of type T.

3.3.7 Asynchronous updates

The validation stage allows the pipeline to know if the changes fulfill all of the constraints, and hence its persistence will be almost guaranteed. It is, however, impossible to assume a 100% uptime of the database due to unexpected errors that may occur, so conversely, it is impossible to guarantee that a valid write request will be persisted. Nevertheless, because a write transaction to a database can take a long time to complete, depending on the amount of data and relationships that are being committed, the user is suspended from further action until the write request has been completed. While the frontend can be developed so that a write request is sent in the background, so that the user can perform other operations in parallel, the user will still have to wait for a successful response if their next task is dependent on the results of this request. As a solution, the proposed model should allow all state updates to be committed to the database in an asynchronous manner.

Figure 3.9 shows a variant of the pipeline in Figure 3.8, where any write action that corresponds to an update, a deletion or a version rollback, is committed to the database in an asynchronous manner. After a transaction has been started asynchronously, the pipeline creates a simulated entity that corresponds to a snapshot of how the entity will be represented in subsequent queries after the changes have been committed, including all of the changed fields. Then, this simulated result is sent to the user in the response body. This technique is not applied to insertions because the backend cannot properly simulate a new entity with the appropriately-generated primary key and relationship graph before it has been created, thus, every insertion is *consistently* synchronized with the end user. All other write operations are *eventually-consistent* at the local level, so for the requesting user alone, this approach will simulate an E+L architecture under the PACELC theorem, rather than the proposed E+C architecture. In other words, between the user receiving the simulated result and the asynchronous thread completing its writing task, the user is temporarily sitting in a local *inconsistency window*. Although the user observes the snapshot of how the object will look like in the short-term future, assuming the writing task is successful and expeditious, this

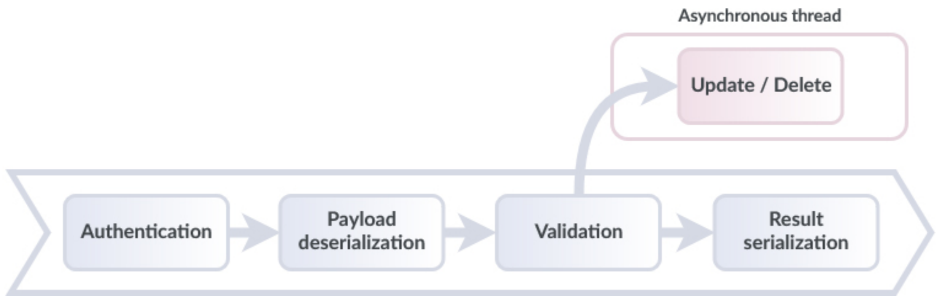


Figure 3.9: Update, delete or rollback pipeline

does not break overall system *consistency* for other users, as these are not supplied with the simulated snapshot and the actual changes are committed to the RDBMS atomically.

Furthermore, all write transactions of entities that are labeled in the series data set are also propagated asynchronously to the TSDBMS. A hook is appended to the update asynchronous thread, so that when and only if this background operation is successful, a new asynchronous thread commits these results to the TSDBMS. The changes are not committed in parallel with the ontology database but rather in succession, since, as previously mentioned, the system cannot guarantee a 100% probability of success for the ontology database to commit changes. In other words, if the ontology database were to fail to commit the changes due to an unexpected error, then there would be no way to cancel the parallel propagation of these changes to the TSDBMS. It is important that the ontology database succeeds before the changes are labeled in the TSDBMS, so as to ensure that both data stores are converged on the same data without breaking data integrity.

As a caveat, once a simulated snapshot is sent to the requesting user, no further messages like errors or logs can be sent because the HTTP connection with the user is already closed. In other words, an asynchronous attempt to persist a set of proposed changes should only happen when these changes synchronously pass all of the field and relationship constraints. The previously mentioned validation stage will ensure that the requested changes contain valid data, and as such will very likely be committed to the database successfully, but this cannot be guaranteed in the face of concurrent updates, database crashes or network failures.

3.3.8 Optimistic Locking

A user that interacts with the platform in a continuous session, regardless of the interface used, will naturally observe the existing entities of the knowledge base and perform changes over these based on their contents at the time. Conversely, when two or more users are simultaneously working on the platform and over the same entities, one user will inevitably run into a situation where the last queried snapshot of an entity has already been internally updated by another user, and hence corresponds to an outdated snapshot. Without any form of detecting stale data and a locking mechanism, the database can allow the user to submit changes based on an outdated snapshot, leading to data loss. Ideally, when attempting to propose changes, users should be notified that the ontology has been updated and that their notes and observations may no longer apply. With this, the user then can locally save their changes, reload the results, and merge their local changes with the recently published changes the way they see fit. This is similar to a behavior found in distributed version control systems, where each user has to synchronize their view of the data with a "master" branch before they can commit changes to it 2.3.3.

To enforce this sequential order of updates, a mechanism of optimistic locking is implemented, blocking any updates made to an entity that may have been sent based on obsolete data and forcing the user to manually re-fetch the entity. The user is required to always provide the last-modified date of the currently viewed entity for every update request, and when this request reaches the update pipeline, a checksum is generated between this date and the last-modified date stored in the database. If the checksum reveals a difference between the two dates, then the update attempt is blocked and the user is properly notified. This mechanism is enforced not only at the processing pipeline level, prior to the validation stage, but also at the persistence level, so that the last-modified date comparison is performed in a atomic transaction. By forcing this integrity checking in transactions, in any given case where two or more updates to the same object are triggered in parallel, only the first update to be received will survive while the second will be canceled and appropriately notified to its author.

While this model enables a strict policy for concurrent updates, it can lead to a few issues when combined with the asynchronous update feature previously described. While the first optimistic locking check is executed in a synchronous manner, if the second check at the transactional, asynchronous level were to fail, then the resulting error message cannot be provided to the user that already received a simulated response and closed their HTTP channel. When users refresh the page expecting their updates to be applied, instead they might find changes proposed by a simultaneous user without having been properly notified of this, losing their own changes haphazardly.

3.4 Visualization

3.4.1 Dashboard

Most of the studied state of the art software solutions for time series visualization, with the exception of *TSPad* [40] and *TimeFork* [15], were implemented as web applications (2.5.3). The most recent developments made to the web browser and JavaScript technologies, the near-universal availability of web browsers in the majority of computers and mobile devices, and the ubiquity in the way users are accustomed to interact with these, make the web platform a more attractive choice for the development of data science software [16]. Therefore, the proposed frontend interface should be implemented within a web application, using bleeding edge web technologies and frameworks. Many application interfaces also provide a workspace dashboard, where collaborators can quickly change the query criteria while keeping the series and annotations always in view. Dashboard interfaces like the one proposed by Healy et al. [12] or like Timelion and Grafana, allow the user to visually structure, separate and display the time series data, as well as functionality for user management and authentication. Additionally, when multi-series or multi-source visualization is supported, the researched

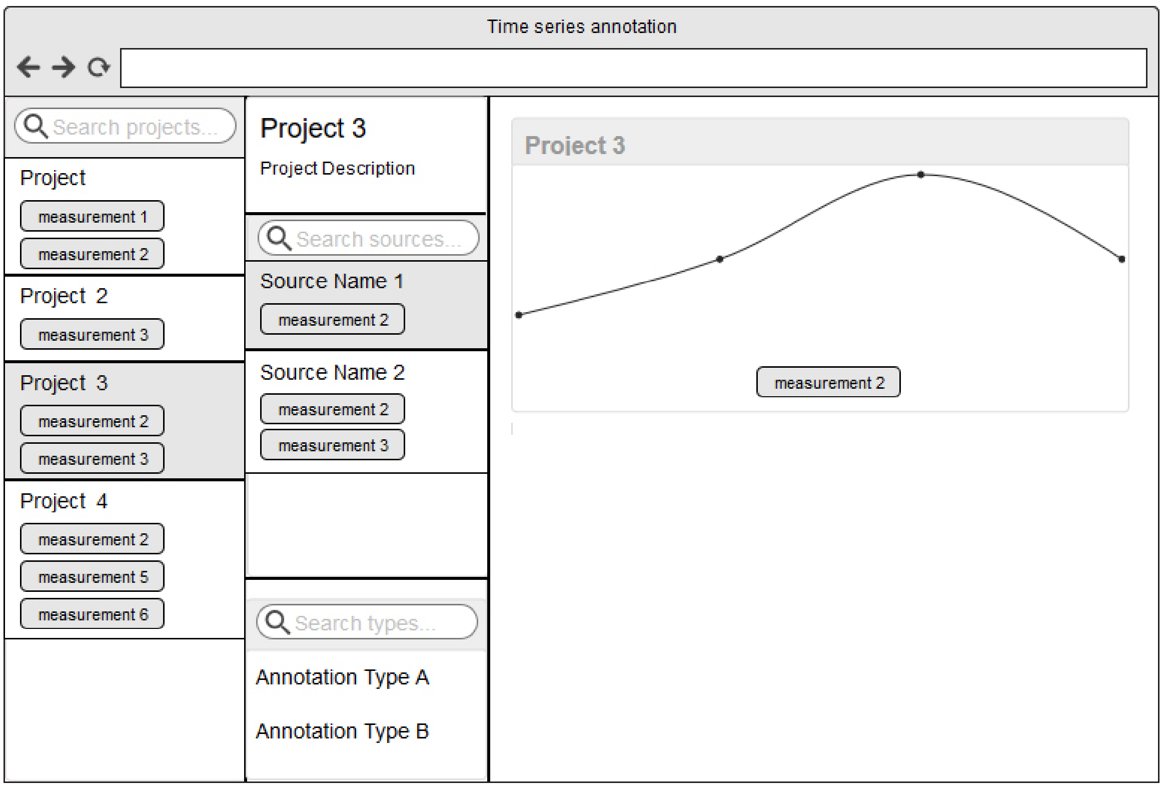


Figure 3.10: Wireframes for the dashboard interface displaying a list of projects and details for a project that queries multiple sources. Only one of the sources is selected.

interfaces display these in either a shared-space model or a split-space model, usually not enabling a combination of the two. Both the split-space model and the shared-space separately have UX trade-offs (1.2) that can make the complete picture harder to parse visually.

Overall, the desired frontend application in this platform should present an interface with a flexible and customizable working area for each user, listing the projects that each user is authorized to collaborate in, the measurements, sources and annotation types supported by the project, and the respective project chart that displays the time series corresponding to the project's scope. For this, the interface leverages the UI paradigm of a dashboard with panels and windows, as seen in Figure 3.10. On the left, the dashboard displays a panel with a list of projects that the authenticated user is an author of or a collaborator in. This list is searchable by its name, its supported annotation types, and the queried segment of time, sources and measurements. After the user selects a project by clicking it, a second panel will show detailed information that corresponds to the selected project. This includes its name, guidelines, the queried source devices, the queried measurements grouped by source, and the visible annotation types. The latter corresponds to a list of annotation types that are being actively used by annotations that are displayed in the series chart. Both the source list and the annotation type list are searchable by name and their supported measurements. When filtering the list of queried sources by measurement, the application will first look up for source-measurement pairs that are queried by the selected project and then find the pairs that

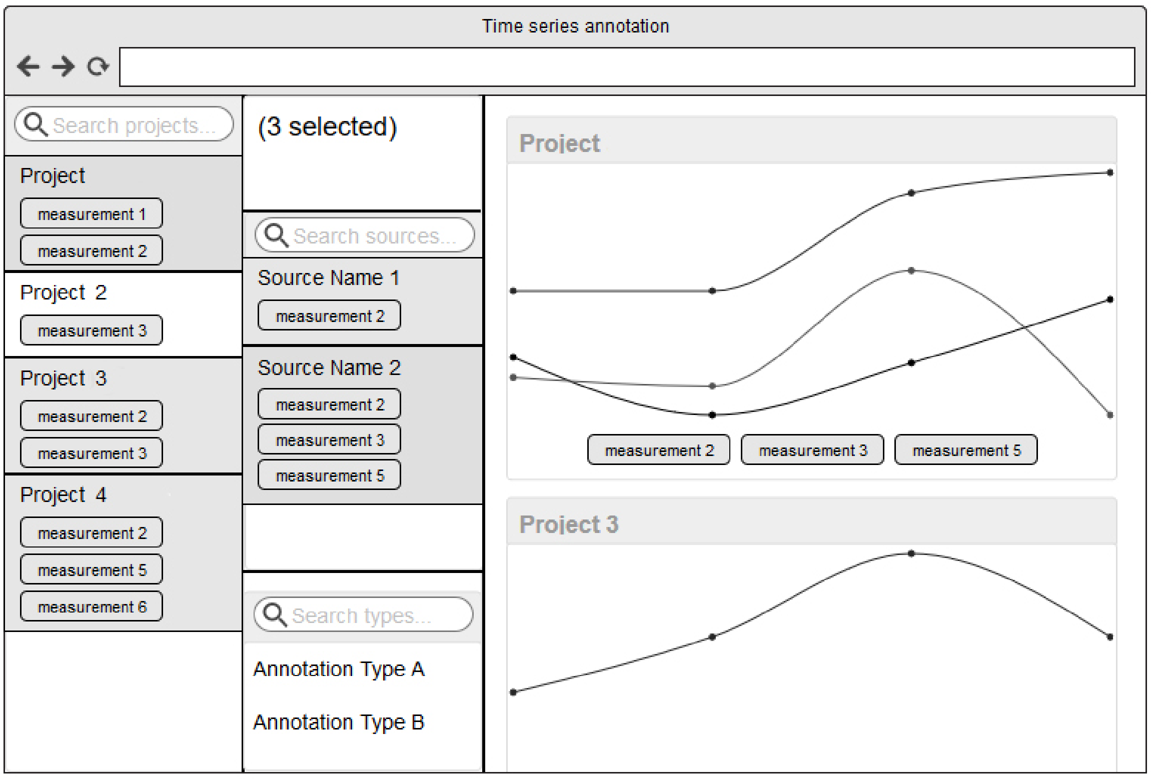


Figure 3.11: Wireframes for the dashboard interface displaying a list of projects and details for three project that query multiple sources. Two out of the three total sources are selected.

contain that measurement. When filtering the list of viewed annotation type by measurement, the application will look for annotation types that are being actively used in the project and that support the specified measurement.

Every visualization chart is grouped by the queried input sources because the most common use cases of the application will have the collaborator quickly jumping between different data sources within the same project. A project is essentially an aggregator of metrics contained within the same interval of time, and users tendentially setup projects so that they can analyze how the exact same set of measurements behaves in different devices. A data scientist that analyzes the project will commonly look at each of the source's chart, clicking each of the sources in the list, and find visual distinctions between the data in order to annotate it. Moreover, if a project contains only a single source, this is automatically selected.

After a data source is selected, a window with a chart is spawned on the side. This chart displays all of the series points that are contained in the project's interval of time and in the selected source, and are color-coded based on their respective measurement's color. Alongside it, the chart also renders all of the annotations in the project that are covering the displayed series. This also includes all of the global annotations, which are always shown in that project regardless of the selected source, since these are global to the entire project. The user can click-and-drag in order to zoom in on the chart, refining the queried interval of time and horizontally expanding the visible series.

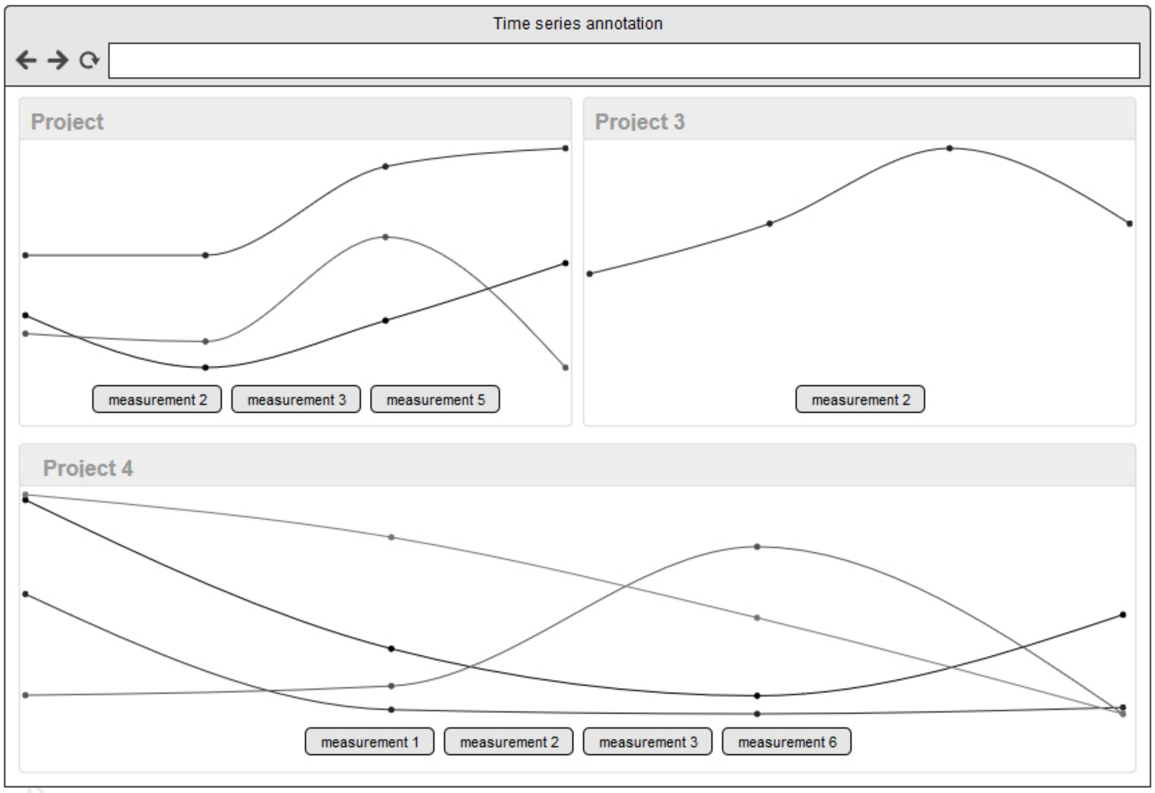


Figure 3.12: Wireframes for the dashboard interface displaying the same selection as Figure 3.11, but the side panels are hidden and two of the charts are resized.

If the user selects multiple projects through *Ctrl*-clicks or *Shift*-clicks, the details panel will show the combined information of all projects, merging the lists of sources and annotation types (Figure 3.11). It will also spawn multiple chart windows, each one representing one of the selected projects. If the user selects multiple sources, the charts for projects that query said sources will display the respective series. With this, both the shared-space model and the split-space model (1.2) are used in simultaneous. The shared-space view is applied for multi-series queries within a single project, so that the entire scope of a project, including all of the selected sources, is shown in a single chart. The split-space view is applied for multi-project views, where each project will be shown in its own individual chart window. This is because separate projects can query different ranges of time, but a single project will query the same range of time over all queried sources.

Like any UI component that mirrors the design pattern of windows, a chart window can be re-arranged with the others, resized as the user sees fit, or closed, the latter of which will de-select the project. Additionally, both the project list panel and the project details panel can be hidden in order to extend the horizontal workspace available for charts (Figure 3.12).

Finally, the dashboard allows the user to perform all of the CRUD operations over projects, such as creating, updating and deleting. When a project is selected, the user can find buttons for editing projects in both the details panel and on the corresponding chart window. On the chart window there are also buttons that will open a project details revisions page and a deleted annotations page. The former will list a table with all of the historical updates that were made to the project's name, guidelines, queried dates, queried source-measurements pairs, and allowed annotation types, identified with a revision ID and a date of when this revision was made. Any collaborator can rollback a project to one of the previous versions, provided that the existing annotations are not covering a segment of time, measurement or source that would be discarded with the rollback (3.3.6).

3.4.2 Annotations

In the state of the art section, it was observed that the highlight design pattern was commonly used to represent annotations (2.5.2). For example, in the dashboard open-source platforms Timelion and Grafana, users can draw annotations as vertical lines (in the case of point annotations) or rectangles (in the case of region annotations) that occupy the full vertical area of the chart. This visual encoding is sufficient for the annotations in these platforms, which are designed to express a global semantic over a segment of time. While this encoding is more than ideal to express global annotations in the proposed platform, the data model for annotations also indicates that an annotation can be attached to one or more of the time series available in the same segment of time, expressing knowledge to only a subset of the data on display (3.1.2). If the previous encoding were to be used, then an annotation would overlay all visible series in its segment, creating a dissonance between the annotation's internal association with series and their visual shape.

Instead, the proposed frontend application should draw annotations as arcs that follow the series curve when inside the specified segment of time, as shown in Figure 3.13. These arcs over curves are designated throughout this dissertation as *snakes*. *Snakes* will only paint over the curves of series that are associated with the annotation, while leaving other series in the same segment uncovered. Similarly to the solution proposed in [90], the color-scheme is based on the designated color of the annotation's parent type, so that an analyst may visually interpret the connotation or severity level of the annotation. If the annotation covers more than one series in the same segment, then a *snake* will be painted over the curves of the affected series and an overlay with a lighter shade of the type's color will be painted in a way that vertically connects these annotations. This overlay does not cover the entire vertical space, allowing series whose curve is traced above or below the covered series to not be painted over. If non-annotated series have curves that pass in the middle of two connected *snakes*, these series will be painted over by the overlay but will still be visually distinguishable from the annotated series, as these will not have a *snake* painted over them.

When the mouse pointer hovers a *snake* or overlay, the respective shapes that constitute the hovered annotation are highlighted with a lighter shade of their color. All overlay shapes are painted under all *snake* shapes, so when hovering a *snake* inside an overlay, the *snake* always takes precedence of interaction so it can be selected.

To create a new annotation in a specific segment of time, the user should be able to click-and-drag on the chart with the *Shift* key held, and the application should automatically discover which series exist in this segment of time and suggest them as series to annotate.

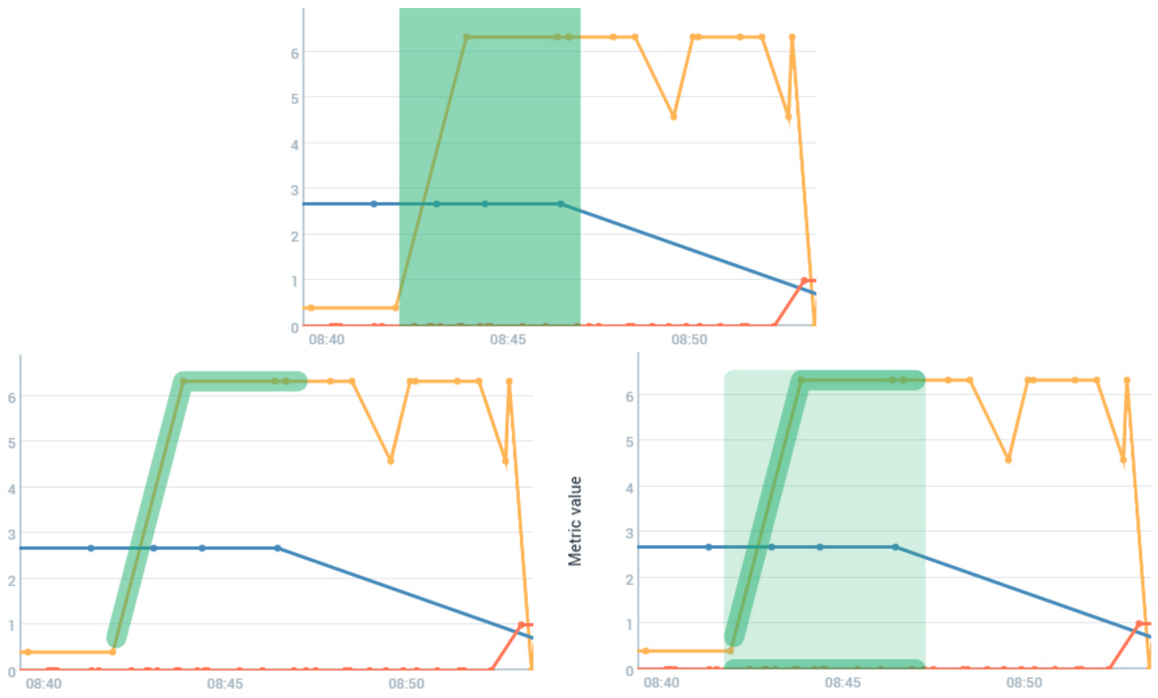


Figure 3.13: Visual encoding of annotations. From top to bottom and left to right: i) annotation encoding in state-of-the-art platforms; ii) proposed annotation encoding, covering only the affected series; ii) proposed annotation encoding, covering two affected series and linked by an overlay, indicating that the blue series is not annotated.

When *Shift*-dragging, or when a click event is intercepted inside an existing annotation, a popover is displayed at the annotation's location containing all of its metadata in editable components. Popovers are similar in interaction design to tooltips, except that these do not automatically close when the mouse is no longer hovering the respective annotation. Additionally, and unlike the solution proposed in [90], this popover is opened with clicks instead of mouseovers, avoiding the existence of a constantly opening obstruction that would constitute an undesirable UX when navigating the chart. With this, users can freely create and edit annotations without leaving the dashboard.

When two annotations intersect over one another, by overlapping on the same segment of time, two output encodings may occur, as illustrated in Figure 3.14. If the annotations are covering different series, the encoding as previously described will be sufficient to visually differentiate the two annotations, since one of the annotations will only have a linking overlay between the painted *snakes* and the other annotation will paint a different *snake* over its respective series. In this scenario, if the overlay between two *snakes* is hovered, the overlay and corresponding *snakes* will be highlighted while any *snake* that does not correspond to the hovered annotation will not. However, if the annotations are covering the same series, one of the *snakes* will assume a wider radius in order to nest the other, keeping both in view and clickable. The nesting rules are based on the annotation's segment of time and on the number of annotated series. Nesting is disregarded for global annotations, which cover a wider vertical area and are painted below all other annotations.

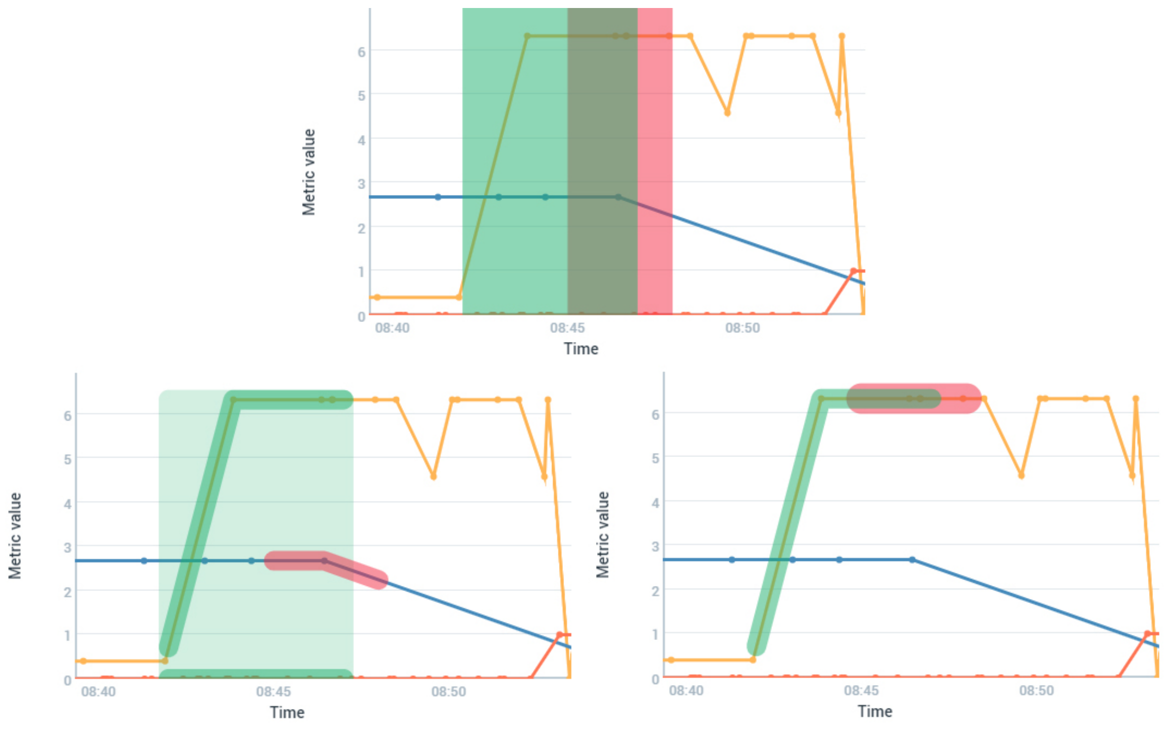
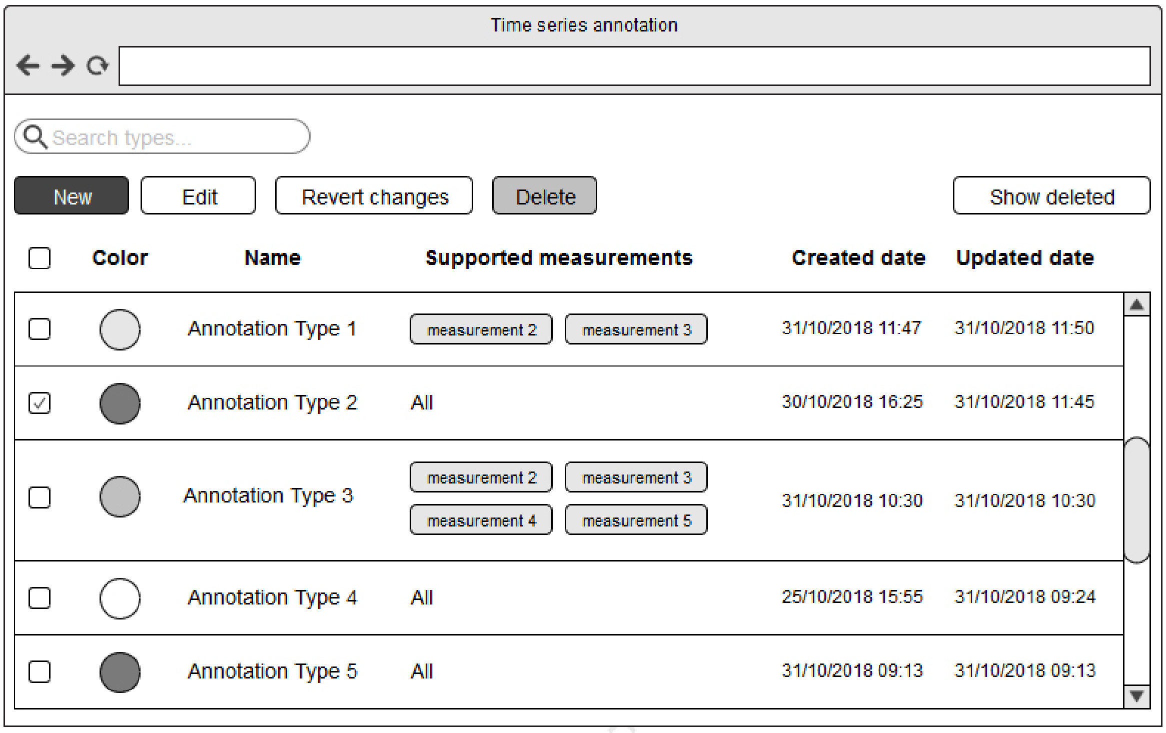


Figure 3.14: Visual encoding of annotations that intersect in the same segment of time. From top to bottom and left to right: i) intersection encoding in state-of-the-art platforms; ii) proposed intersection encoding, where one annotation is covering the yellow and red series while another one is covering the blue series; iii) proposed intersection encoding, where both annotations cover the same time series.

Having any two annotations A_1 and A_2:

- if A_1 is a point and A_2 is a region, point annotations always take precedence over any other shape, so A_1 is painted on top and A_2 is enlarged to accommodate A_1 inside it;
- if both A_1 and A_2 are a point, the one that has less annotated measurements is enlarged and painted under the other, as their connecting overlay will naturally provide a wider area of visualization and selection. If both annotations contain the same series, the nesting precedence is picked arbitrarily;
- if both A_1 and A_2 are regions, and if A_1 starts and/or ends inside A_2, then A_2 is enlarged and painted under A_1. Looking at all annotations sorted by time, the inner annotation is the one that started its region first and ended on the other. If both annotations start their region at the exact same time, the inner annotation is the one that ends first. If both annotations have the exact same region, the nesting precedence is picked arbitrarily.

By using this encoding, annotations are granularly represented and correctly associated with the affected series, constituting a major improvement in graphical perception over the simpler rectangular overlays.



☐	Color	Name	Supported measurements	Created date	Updated date
☐	○	Annotation Type 1	measurement 2, measurement 3	31/10/2018 11:47	31/10/2018 11:50
✓	●	Annotation Type 2	All	30/10/2018 16:25	31/10/2018 11:45
☐	●	Annotation Type 3	measurement 2, measurement 3, measurement 4, measurement 5	31/10/2018 10:30	31/10/2018 10:30
☐	○	Annotation Type 4	All	25/10/2018 15:55	31/10/2018 09:24
☐	●	Annotation Type 5	All	31/10/2018 09:13	31/10/2018 09:13

Figure 3.15: Wireframes for the annotation type management page. All other ontology management pages should look identical. Some features, such as creating and deleting, are not available for measurements and sources, as these are handled by the system and dependent on the input time series data set.

3.4.3 Ontology management

While users can perform CRUD operations for projects and annotations in the dashboard itself, the visualization layer should also allow users to manage other active entities that compose the ontology: measurements, sources and annotation types. All management pages are implemented as a multi-selection, paginated table, displaying all of the available metadata in columns. The user can select entities from the table, by toggling their checkbox on the left, and click the buttons above in order to edit, rollback, or delete them. Multiple entities can also be selected in order to perform batch edits or batch deletions. The annotation edit popup allows collaborators to change metadata like names, colors and constraints. In the case of annotation types, users can add new allowed measurements or remove existing ones. If the user attempts to remove a measurement while a child annotation is still annotating it, then the validation stage of the backend will properly validate this and report this action as invalid to the user (3.3.6).

Annotation types can only be deleted if they are unused, and the versioning feature allows any user to quickly recover these in case they were deleted erroneously (3.2.2). Any collaborator should be able to rollback changes made to annotation types, measurements and sources by using the “Revert changes” feature, or to recover an annotation type that was deleted, by using the “Show deleted” feature.

CHAPTER 4

Implementation

This chapter outlines the implementation of a full-stack prototype that follows the proposed architecture, data model, visualization techniques, and UI specified in the previous chapter. All of the previously described architecture abstraction layers and their respective components (3.3) are implemented using open-source tools and state of the art methodologies for building high-performant, scalable backends and reactive web applications. While the previous chapter described an architectural model in a domain and data-agnostic manner, this chapter describes how this model was implemented and tested while using an HVAC data set, collected from 1000 boilers over the course of 1.3 years. This data set is ingested into the platform only once, during the first deployment, and was ideal for testing how the prototype performed when handling reads and updates over massive quantities of time series. Along with this data set, in order to properly benchmark the prototype throughout the development phase, automated tasks were implemented to generate thousands of annotation types and over a million annotations within mock projects.

Based on the state of the art on data management techniques and existing database systems (2.3), and in order to optimize querying of both ontological data and time series, these are split into their own specialized databases (3.2.1). An essential requirement to uphold a custom granularity rule set, which delineates how data is linked together and how it can be accessed by authenticated users, is to implement a central backend system that enforces that rule set, validating user requests and converting these into appropriate statements towards the relevant databases. For this, a monolithic backend materializes all of the components that are part of the processing layer, such as the authentication process (3.3.3) and the processing pipeline (3.3.4), as well as the REST endpoint that is part of the communication layer (3.3.2). This backend is implemented in Java 8[1] and using the Spring Framework[2]. More specifically, it uses various modules from the Spring Boot 2.0[3] stack.

[1]`https://www.oracle.com/technetwork/java/javase/overview/java8-2100321.html`
[2]`https://spring.io/`
[3]`https://spring.io/projects/spring-boot`

When performing a survey over existing benchmarks on time series storage technologies (2.3.1), InfluxDB presented itself as the most appropriate TSDBMS option for storing the time series data set. The combination of InfluxDB's reported high read performance and on-disk compression capabilities allows an ever-increasing massive time series data set to be stored without being concern over disproportionate decreases in query speed or over exponential scaling of disk space. InfluxDB leverages a SQL-like syntax for queries, named InfluxQL, which allows for complex querying capabilities similar to SQL. InfluxQL's primary differences with common SQL are: the ability to set time expressions that are relative to the current time or to a specific date, enabling queries over ranges of time such as "the last hour"; the ability to perform aggregated calculations of numeric values for continuous series; the ability to test regular expressions over string fields; and the lack of table aggregations through the *join* operator, requiring queries to contain sub-queries that select the required tables instead. All of these features and limitations are taken into consideration when designing the prototype queries (4.1.1).

When surveying how the many open-source RDBMS options performed in existing solutions (2.3.2), PostgreSQL consistently showed excellent results, which motivated its usage as the ontology database system. The high level of maturity and feature-completeness it provides, the fault-tolerance mechanisms, and the high query speeds only further encouraged this choice. Furthermore, additional functionality like indexing of text vectors, which enables a performant textual search without requiring a full-text search inverted-index database like Elasticsearch or Solr[4] (4.2.2), has further pushed PostgreSQL to the top of potential choices, since multiple aspects of the platform require name searches. Finally, because the established priority in the architecture design for the ontology data set in on *consistent* reads and writes, so that collaborators are guaranteed to always iterate over the most recent set of data and not overwrite data in an unsanctioned manner, then PostgreSQL remains the preferred choice due to its strong MVCC model.

4.1 Data model

4.1.1 Time series

Source and Measurement linking

The common workflow for time series querying always starts at the ontology level: when a user has to determine which measurements or data sources are needed to work with, the first step is always to discover these in the ontology (by name or through a list), and the second step is to use the fetched information to query the respective series. Likewise, the proposed strategy for storing and managing the series data set follows a similar pattern: i) sources and measurements are extracted from the original data set and added to PostgreSQL, generating primary keys for each (4.2.1); ii) every series point is then inserted into InfluxDB, identifying measurements and sources with the generated primary keys.

[4] `http://lucene.apache.org/solr/`

```
SELECT * FROM single_table_with_all_series
WHERE (source = :sourceId1 AND measurement = :measurementId1)
OR (source = :sourceId2 AND measurement = :measurementId2)
```

Code 1: Query for single table model in InfluxDB

```
SELECT * FROM (
  (select * from :sourceId1_:measurementId1) AND
  (select * from :sourceId2_:measurementId2)
)
```

Code 2: Query for table-per-combination model in InfluxDB

```
SELECT * FROM (
  (select * from :sourceId1 WHERE measurement = :measurementId1) AND
  (select * from :sourceId2 WHERE measurement = :measurementId2)
)
```

Code 3: Query for table-per-source model in InfluxDB

One of four data model approaches could be employed:

- **i)** write all series data into a single table, and identify all series by their source and measurement (using tags for both);
- **ii)** create tables for each of the available source-measurement combinations, and separate series as such;
- **iii)** create tables by source, using a tag to distinguish each series point's measurement;
- or **iv)** create tables by measurement, using a tag to distinguish each series point's source.

For all of the given approaches, query statements must be modeled accordingly. In approach *i* (code snippet 1), queries would have to be written so that only one single table is selected, but a *where* condition would have to be added for each of the requested source-measurement combinations. In approach *ii* (code snippet 2), because each individual table is an unique combination, then the query requires multiple sub-queries and no *where* conditions. In approach *iii* (code snippet 3), each table is a source, so the *where* conditions are reserved for measurement equality. However, these *where* conditions must be placed at the sub-query level, and not the global query that joins the two sub-queries. In approach *iv* (code snippet 4), the query is similar to the one in approach *iii*, but sources and measurements are swapped.

An important factor in deciding the best approach is the fact that, by design, lower data cardinality leads to better performant queries in InfluxDB. Cardinality, in InfluxDB terms, is the total number of unique tables, points, tag sets and field keys returned for any given query, so that the higher the number of unique tables and indexed tags, the more tables it will have to join together and the more rows of each table it will have to scan. Moreover, InfluxDB

```
SELECT * FROM (
  (select * from :measurementId1 WHERE source = :sourceId1) AND
  (select * from :measurementId2 WHERE source = :sourceId2)
)
```

Code 4: Query for table-per-measurement model in InfluxDB developers also recommend that each table belongs to a single measure, and to not encode more information in one table name or tag [5]. This is suggested so that all of the rows in a given table contain the same data-types and common behavior that can easily be indexed and aggregated for rollups.

At first there was some apprehension about the measurements being so spread-out, so that most of the common queries that filter multiple measurements would need multiple sub-queries. After benchmarking the various data model approaches, the reduced cardinality actually leads to a major improvement in querying speed, which is due to how InfluxDB internally stores measurements in an aggregated manner that makes sub-queries for points with the same data-type faster than a more singular schema of the single-table approach or the spread-out schema of the table-per-combination approach. With this, the approach that grants the highest performance is approach *iv*. Because series are indexed by timestamp and tags, there is no fear of the series points being indexed in a way that would overlap and discard each other.

Annotation linking

Annotation linking should follow a similar pattern to the above, where annotation IDs generated in PostgreSQL are stored as a part of each series point. The difference in this model is that annotations are not part of a set of attributes that uniquely identify a series, but rather of a set of non-indexed attributes. In InfluxDB terms, annotations are stored as a string field rather than a tag. However, multiple annotations should exist in each series point, and the limited data model of InfluxDB does not support the storage of lists or the modeling of relations with data stored separately in another table. To work around this limitation, multiple annotation IDs A_1, A_2 and A_3 are joined in a single *annotation* field separated by semicolons ";", being stored as "A_1;A_2;A_3". When querying for A_2 through its ID, InfluxQL's regex-like pattern-matching over strings is used by setting a *where* condition that searches for A_2 with the expression "/.*A_2*/", as shown in code snippet 6.

Similarly, additional metadata that was listed in the proposed persistence model (3.2.1), such as the parent annotation type and project of each of the linked annotations, should also be linked in the series points, in order to reduce the dependency on PostgreSQL when performing common series queries. An initial approach would be to contain a set of type IDs and project IDs in new fields *annotation type* or *project* respectively, similarly to how it is done with annotations. When needing to remove annotation links, their parent annotation type and project links must also be removed. However, if two annotations contain a common annotation type or project, there is no way of knowing if these links must be kept or removed

[5] `https://docs.influxdata.com/influxdb/v1.6/concepts/schema_and_data_layout/`

```
SELECT * FROM :measurementId
WHERE source = :sourceId
AND annotation =~ /.*:annotationId.*/
```

Code 5: Query for annotated series in InfluxDB without checking in PostgreSQL the parental relationships of each individual annotation that is also linked, which can be a very heavy task that does not scale well for massive data sets.

Instead, all of the linking fields *annotation*, *annotation type* and *project* must be persisted in synchronized lists with the same number of total elements, where *annotation* only contains unique IDs while the latter two will contain repeated IDs. The three entities are matched together by their position in the three lists, so that the *Nth* annotation that is in the *annotation* list has the *Nth* annotation type in the *annotation type* list and the *Nth* parent project in the *project* list. This way, when removing an annotation link, the annotation type and project links in the same position will be removed as well. While this solves the issue above and enables the correct modeling of annotation type and project linking, it presents a caveat where new linked annotation will increase the total disk usage exponentially, as they occupy three times the size they normally would.

4.1.2 Ontology

The backend system fulfills the proposed data model specification (3.1) by modeling the entire ontology as Hibernate entities, using both the Hibernate and JPA APIs to describe tables, indexes, columns and relations (one-to-one, one-to-many, many-to-one and many-to-many). When the backend is deployed, these Hibernate entities will be properly created in the PostgreSQL instance using equivalent column types. The usage of ORM to model the database schema entirely within the backend codebase, using Java types and Java annotations, introduces type safety, improves readability, and brings a set of other features that grant easier maintenance over long development cycles, such as IDE support and debugging through the Java compiler. By directly using Hibernate objects when serializing a JSON response, validation and uniformity is easier to handle:

- objects that are used to model the entities are also the objects used to represent the entities in the REST API responses, so every request will have responses with deterministic field names and types;
- when refactoring an Hibernate entity model directly, the changes in this refactor will be reflected both in PostgreSQL and in the responses sent to the client.

One potential issue that can derive from the usage of an ORMs is the N+1 query problem[6], where relationships that are read by the backend will trigger a fetching of these relationships on-demand, splitting initial object fetching and relationship fetching over multiple queries. This issue is solved by forcing all relationships to be *lazy*, throwing exceptions if a non-queried relationship is accessed, and by introducing "join fetch" clauses in queries for objects whose relationships will be required by the backend or client.

In order to establish a homogeneous data model over all entities, every ontological entity in the system contains a set of baseline fields: a string ID, which corresponds to the entity's unique primary key; an *inactive* flag that is disabled by default, so that the entity is considered active; an *updated at* timestamp, which is automatically updated by the system on every update transaction; a *created at* timestamp, which is unmodifiable and automatically set by

[6]`https://stackoverflow.com/questions/97197/what-is-the-n1-select-query-issue`

the system when the entity is first stored in the database; and an *author*, which corresponds to a many-to-one relation with a user entity. The latter three fields are stored internally for every model, but only exposed to the end user for models that are created and maintained by the users rather than by the system or by only one user. In other words, the *updated at*, *created at* and *author* fields are always hidden for measurement, source and user entities.

As previously stated, sources and measurements are directly derived from the time series data set that is read on the first deployment of the system. Both follow the proposed data model directly (3.1.1), where they are composed of a public name, that can be changed by the end user without restrictions, and in the case of measurements, a used-configurable color and a data type of the measurement's series. Beyond that, both models are also composed of a private external name, which is a unique name that this source or measurement has on the input data set. This allows stored sources or measurements to be matched with their counterparts on the input data set, so that a potential new data set of series with the same sources and measurements can be appended to these already existing entities. However, these external keys are not used as primary keys, because it is not guaranteed that this identifier is always consistent in future input data sets. The external keys for sources and measurements can be changed over time in the input data set, and in such a case, this external name, while unmodifiable by the end user, can be modified through direct-access to the database by maintainers. Primary keys are instead generated by the platform itself (4.2.1).

4.2 Data management

4.2.1 ID generation strategy

In a distributed environment where the backend system is deployed and replicated across multiple machines, it would be onerous to use an additive identity such as an incrementing integer as a primary key, as this would require all instances of the backend to query the database or a central authority that holds the current increment on every insertion, in order to find the next incremented integer to be used on the new entity. Other than lower write speeds due to the added overhead of finding the next increment, when multiple instances receive simultaneous insertion requests, all instances could attempt to access the database or central authority simultaneously and potentially have collisions.

With this, it was necessary to find an ID strategy that would allow each backend instance to generate a unique ID independently, even if simultaneously. Unique objects at a single instance level, like a timestamp or a MAC address, are not enough for a multi-instance environment. Timestamps, even at nanosecond precision, have a non-negligible chance to collide if generated at the exact same time. MAC addresses, while normally being unique identifiers for machines in the same network, are spoofable, as they can be changed manually, and can collide with other threads in the same machine. However, a combination of all of these identifiers, along with an identifier of the current machine and thread namespace as a message digest, hashed by a hash-function like SHA[118] could reduce the probability of collisions to a negligible level. This is essentially the concept of an Universally Unique Identifier (UUID), which is the chosen ID generation strategy.

One issue that was detected while using UUIDs however, was the fact that its representation has a total of 36 characters, which include 32 alphanumeric characters and four hyphens. Multiple REST requests use multiple sets of UUIDs as part of the request URL or as part of their payload, extending their length and negatively affecting human readability. Multiple options were researched to reduce an UUID to a minimum amount of characters as possible without losing uniqueness, and the wide majority of options available used a transformative function like hashing or Base64 encoding. Hash functions were inappropriate for this use case, as they generated even lengthier representations, but by encoding the most significant and least significant bits of an UUIDs into a Base64 representation, it was possible to reduce the an UUID to a 22 character URL-safe encoding while retaining a low level of collisions. For example, the UUID "44bd2503-2180-4a54-82aa-13365a96c754" would become "RL0lAyGASlSCqhM2WpbHVA", which is significantly smaller and more readable.

4.2.2 Name search

One of the main requirements of the proposed model is that the ontological entities that are identified by a name, such as annotation types, sources or measurements, should be text-searchable. This is used throughout the implemented frontend application in various stages (e.g. searching for entities in their respective ontology management pages, setting a parent type for an annotation, filtering projects, filtering sources of the selected project, etc...). An initial approach to support this would be to use regular expression pattern-matching, which in PostgreSQL is done by using the *LIKE* operator. A *LIKE* predicate can be expressed with a user-specified keyword prefixed and suffixed with a % symbol, as shown in code snippet 6, instructing the database system to look for names that start and end with the queried keyword.

However, this approach will quickly result in major performance drops for name searches, since name patterns cannot be indexed in an universal format that allows further pattern queries to find matches in a more optimized manner. Every *LIKE* predicate requires a separate character-based lookup for pattern matches over all rows of data. It also lacks adaptability, where if the user inputs a lexical variant of a word that exists in the data, the query will still return a false-negative. With this, two separate approaches can be outlined: deploy a separate data store for text indexing and searching based on an inverted-index structure, like Elasticsearch or Solr, or take advantage of PostgreSQL-specific capabilities for full-text, using built-in full-text functions like *ts_vector* and *ts_query*.

When deploying a separate data unit for text searches, the data granularity increases further, and a more granular persistence model will enable higher *availability* of data. Query speed for text searches is significantly improved, as the system gains the advantages of a data

```sql
SELECT * FROM project AS p WHERE p.name LIKE %:name%
```

Code 6: Query for regular expression name search in PostgreSQL

```sql
SELECT * FROM project AS p WHERE to_tsvector(p.name) @@ to_tsquery(:name)
```

Code 7: Query for full-text name search in PostgreSQL

store that indexes ontology identifiers and names ahead of time. However, the same concerns related to the system's *consistency* listed during the discussion of persistence models in 3.2.1 are also applicable here. All names and searchable contents are required to be replicated into an additional unit of storage, and additional changes made to the ontology data store will have to be manually propagated to this data store. Because a propagated update over the ontology storage and the text-search storage cannot be built in a linearizable manner, ensuring that the proposed changes are reflected in the most updated index of the text-search unit, then every update will open an *inconsistency window* where the system is *consistent* for queries to PostgreSQL but *eventually-consistent* for name search queries. This search index essentially enables a better user experience, but is out-of-sync and *inconsistent* for a nondeterministic amount of time.

Additionally, most name searches should be capable of returning results that match not only the text-search predicate but also other predicates related to the entities' metadata and their relations. When querying the system, the user expects the resulting entities to contain the same fields that other non-name-search queries would, including *created at* and *updated at* dates, project descriptions, annotation timestamps, and others. This requires one of two methods to be implemented: i) the inverted-index data store is modeled in a similar way to PostgreSQL, containing not just names but also all other fields; ii) a query is forked between PostgreSQL and the inverted-index data store. With the first method, because the search index already contains all ontological data, every query can be directed towards the search index. However, the benefit of querying over a relational model is lost, losing with it the ability to fetch entities based on their associations with other entities. With the second method, a bottleneck is introduced for every query, since these will have to lock and wait for both data stores to deliver their results before they can merge the matches of both and return these to the client. This fork-join protocol for queries will also make paginated queries impossible to implement, since the merging of both matches will filter some results out and pages will not always contain a complete page of results.

In sum, the use of an additional data structure for name searches would be potentially viable if name queries were not required to have filters for additional, non-text fields. The need for deploying and additional data storage unit in the architecture would lead to a decrease in *consistency*, not to mention the added maintenance, migration, schema changes and failover concerns. Due to these requirements and the preference towards an E+C architecture for the ontology, this approach was excluded as an option.

The second approach was to use PostgreSQL built-in functions for full-text search, which allows the user to specify full-text searching as one of many other potential criterias over the same data storage unit. The built-in function *ts_vector* generates a list of vectors (or tokens) that are normalized to the smallest lexical unit, composing the information retrieval tasks of tokenization and stemming. These vectors are built on demand during a query, and then tested with the specified keyword through the use of another built-in function *ts_query.* The latter, similarly to *ts_vector*, tokenizes and stems the specified keyword, and then matches it with the vectors of stored names. The query is then structured as in the code snippet 7.

In this query, a lookup for projects with the keyword “heating boil” will not just look for exact matches of the word “heating” or “boil”, but also for other words that can be reduced to the lexical units “heat” and “boil”, matching project names such as “performance of heat recovery” or “boiler values”. The current implementation bases tokenization and stemming tasks on the provided English dictionary, building on the assumption that all stored textual contents use English words. However, PostgreSQL already supports multiple dictionaries, requiring only for the language to be specified in the *ts_query* function. A potential iteration over this approach that would support a multi-lingual community would be for the intended language to be provided directly by the client.

However, this built-in normalization process is not enough to handle corner cases with text such as casing differences, repeated spaces, special characters and diacritics. For this, an additional column is created for each of the full-text fields that are made searchable in the system, storing in these a normalized variant of the original contents. When an entity is created or its name is updated, the new name is put through a text cleaner pipeline that processes the new contents char by char in a single-pass. The original name is stored in the *name* column while the normalized variant is stored in the *name_indexed* column. All *ts_query* are then applied to the *name_indexed* column, so that the *name* column is ignored for querying purposes but still returned to the user as-is.

The above-mentioned cleaner pipeline reviews each character of the input text and decides if it should either replace it or remove it. A cleaner module, identified by its separate rule-sets, can be coupled together with other modules based on the deployment configuration file. Regardless of how many cleaning modules are included in a single pipeline, the textual content is always reviewed on a single pass, running at O(n). A set of text cleaning modules was implemented, although more can be implemented and added to the system through the use of the plugin pattern[7] and Java Reflection API[8]. The implemented cleaners deal with cases such as conversion to a single case (all lower), removal of repeated spaces, conversion of words with diacritics to their normalized variant, and removal of non-informative special characters. For the latter module, as a rule it will only clear paragraphs, tabs, and characters that do not provide much information on their own, such as quotation-marks and bullet-points, but keep characters that provide mathematical information, such as π, Σ, α, β and δ.

Because name vectors are built on a query-basis, this approach solves the *consistency* issues addressed in the previous approach while enabling relational and paginated fetching, and the system is always synchronized and *consistent* with updates made to the ontology data store. Moreover, it results in higher query performance over the regular expression approach, as well as improved adaptability to user-specified text, since the searches are matched with

[7]`https://martinfowler.com/eaaCatalog/plugin.html`
[8]`https://docs.oracle.com/javase/tutorial/reflect/`

data in regards to their lexical variants and individual words as opposed to exact matches.

4.2.3 Versioning

As specified in 3.2.2, the versioning approach should be one where full records of historical data are recorded in separate tables, along with boolean flags that identify which records have been changed in each version. For this, versioning is implemented by using Hibernate Envers[9], which allows the versioning schema to be modeled within the same Hibernate entities in 4.1.2, using the *@Audited* annotation. Once the backend is deployed, committing the complete data model into PostgreSQL, a set of tables with the "_log" suffix are automatically created, containing all of the audited fields using a consistent naming convention. As a rule, while text columns that correspond to searchable names are audited, the indexed text columns described in 4.2.2 are not, since the contents of the indexed column can be rebuilt from the original text by re-running the same normalization and text cleaning processes. Columns like *id*, *created at* and *author* are also not versioned because they correspond to immutable properties.

4.2.4 Caching

Following the specification in 3.2.3, the prototype requires a cache unit to be deployed in its architecture for short-term in-memory persistence of commonly used but infrequently changed data. For this, it leverages a single node instance of Redis to deal with all caching needs. Redis already implements all of the proposed requirements, such as a LRU eviction policy, a wide variety of supported data-types, and other features that allow the solution to scale over time such as clustering. The connection with Redis follows a similar pattern to the Spring JPA connection through repository components, using the Spring caching API [10] and Spring Data Redis[11] library. The latter library already provides functionality to connect to a Redis instance and to serialize ORM entities from and to cached objects.

With this, the backend codebase attaches caching mechanisms directly to Spring repositories by using the *@Cacheable* annotation on query methods that should be cached when called. A cacheable query is identified by a cache name, as each set of separate key-value bindings that use different key and value types should be grouped as separate caches, all stored in and managed by the same Redis instance. The *@Cacheable* annotation will effectively store the results of the associated query while using the input parameters, listed as function arguments, as keys that identify the cached object. This means that *@Cacheable* should only be placed on queries with a small number of arguments, since the higher the amount of arguments are used as keys, the higher the combination of arguments with different values will be. As the number of keys grows exponentially, the higher the number of key-value mappings will be occupying the available memory, causing cache pollution and degrading overall performance.

Because pair objects are widely used during validation and update operations, and are never changed over the course of the system's life cycle, these are cached with a high amount

[9]`https://hibernate.org/orm/envers/`

[10]`https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-caching.html`

[11]`https://spring.io/projects/spring-data-redis`

of allocated memory. Lookups by ID, whose identifying key is a single “id” argument, are also cached, as these correspond to high-reuse queries. All six active entities of the system have their own cache for lookups by ID. Finally, and as described in the proposed model (3.2.4), user tasks, which instruct users to re-add annotations that were invalidated, are also stored in cache. The expiry policy is disabled specifically and only for user tasks, as these correspond to short-term notifications that, were they constantly evicted, the user task scheduler would just repeatedly re-add them back to the cache, nullifying the usefulness of an expiration policy in the first place.

4.3 Architecture

The backend is deployed in an embedded Jetty[12] servlet with a set of always-on services, and is fully archived into a *jar* file. Running this *jar* file will deploy an embedded Jetty server in the local system where the file is placed, launching all of the required Spring components that connect to the RDBMS and cache data store according to a configuration file.

Each processing server is deployed on a servlet container with its own thread pool, so it can already handle a massive number of simultaneous requests independently, only limited by the resources of the machine where the backend is deployed to. In this way, each backend is essentially a dedicated cluster of processing pipelines. However, the replication of this server introduces additional parallelism and scalability that is only limited by the number of replicated servers and their individual resources. As the platform scales and servers become strained with simultaneous workloads, more backend servers can be deployed on demand.

The entire architecture, as illustrated in Figure 4.1, is deployed entirely using Docker Swarm[13]. Docker Swarm is a container manager within the Docker[14] platform where each unit of the architecture can be distributed between nodes or containers. All units, including message queues, databases and backend servers, are deployed as individual containers, configured through a set of *Dockerfiles* that are launched as part of the same swarm. Docker Swarm enables the infrastructure to scale, growing or decreasing the number of containers based on workloads and providing redundancy or failover in the case of unexpected errors. This is particularly helpful in facilitating backend replication, as both the number of distributed queues in RabbitMQ and the number of backend servers can automatically grow as the platform scales and more activity is detected. Moreover, the required units for load balancing are already provided by Docker Swarm built-in, only needing to be setup to redirect load between the existing distributed queues.

[12]`https://www.eclipse.org/jetty/`
[13]`https://docs.docker.com/engine/swarm/`
[14]`https://www.docker.com/`

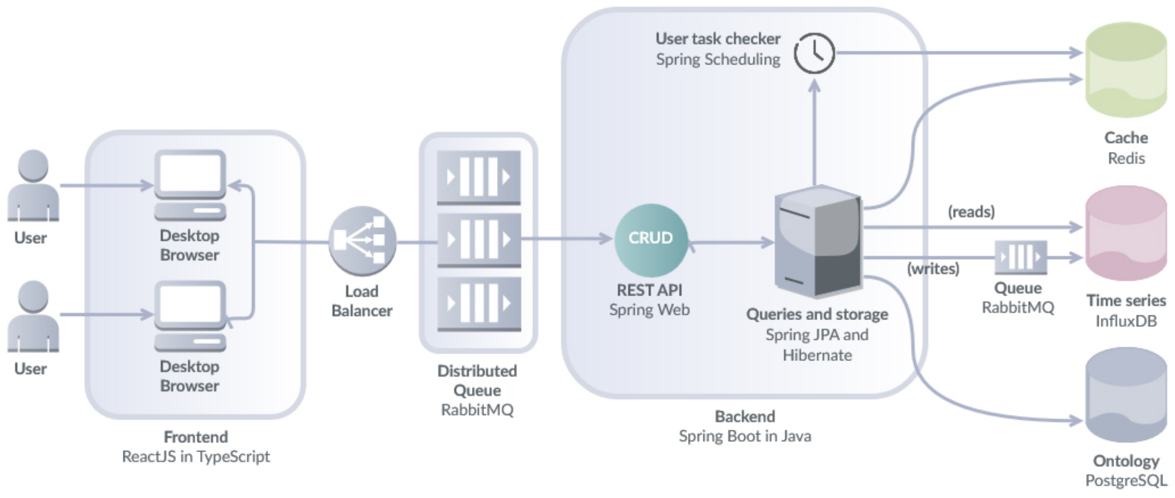


Figure 4.1: Platform architecture

4.3.1 REST interface

To implement the REST interface as blueprinted in 3.3.2, a collection of Spring components called *controllers* is used. Serialization and deserialization of request and response payloads is handled by the Jackson[15] library of utilities. Each implemented method/function in a controller corresponds to a separate REST entry point of requests. *GET* methods are mapped with a *@GetMapping* annotation, *POST* methods with *@PostMapping*, *PUT* methods with *@PutMapping*, *PATCH* methods with *@PatchMapping* and *DELETE* methods with *@DeleteMapping*. Multiple controllers are implemented, where each controller corresponds to one of the active entities of the application.

All REST calls are uniform between all entities, so that, for example, requests for annotations and annotation types should have the same *GET*, *POST*, *PUT*, *PATCH* and *DELETE* methods, varying only in query parameters. These REST calls are separated by the starting relative URL that matches the name of each entity, with the exception of annotation calls which are prefixed by a parent project ID. In other words, the REST API implements all the listed methods in the following base relative URLs: "/annotation types", "/sources", "/measurements", "/users", "/projects" and "/projects/{projectId}/annotations". Both *POST* for insertions and *GET* for queries use these base URLs with no additional changes. The *GET* method for list-queries always returns paginated results, of which the query parameters *size* and *page* can be used to change the pagination window. *GET*, *PUT*, *PATCH* and *DELETE* methods for single objects use the previously mentioned relative URLs with an ID as a suffix (e.g. "/annotation types/zWfsakd2Dc12Da"). The *PUT* method to revert deleted objects follows the same URL pattern, flagging the specified ID to active (as described in 3.2.2). Moreover, to enable bulk editing and bulk deletion, there are bulk variants for the *PATCH* and *DELETE* methods whose URL is the same as the base relative URLs, but requires the query parameter "id" to be set mandatorily. Multiple "id" parameters can be set, reading all of these as a list of IDs that must be edited or deleted. Finally, revision lists are implemented

[15]`https://github.com/FasterXML/jackson`

with a *GET* call using the previous pattern with the "/revisions" suffix (e.g. "/annotation types/zWfsakd2Dc12Da/revisions"), and rollbacks are implemented with a *PUT* call where this pattern is appended with the specific revision ID that the object needs to be rolled-back to (e.g. "/annotation types/zWfsakd2Dc12Da/revisions/5").

Only source and measurement controllers do not implement methods for insertion and deletion operations, as these are handled at the system level (3.1.1). Along with the above calls, there is a *POST* call that is used to query time series from InfluxDB, receiving in its request payload a structured query object (3.3.5).

4.3.2 Authentication

For all of the REST calls mentioned, these are intercepted by a once-per-request filter before being allowed to pass to their respective functions. This filter, as a component that is part of the Spring security API[16], implements the JWT authentication as specified in 3.3.3 using the JJWT library[17] to create, parse, and manage JWTs. Tokens must be included in every request as part of the Authorization header, and every valid response from an authenticated user will come with a refreshed token in the response's Authorization header, which the user and/or frontend application can read and keep as the newest token with a new expiration date. If a token is considered invalid, either because it cannot be resolved to a valid key-value map of claims or because the token itself has expired, then the filter will send an immediate error response as defined in 3.3.2, effectively canceling the request. There is only one call that is not intercepted by this security filter, and it is a *POST* method in the relative URL "/auth". This call is explicitly used for logging in with a username and password, so that users can authenticate themselves and receive a valid token to use in other requests.

4.3.3 Processing workflow

The codebase of the backend itself does not detail how it should connect to PostgreSQL nor how the queries should be written using SQL keywords. Instead, through the use of Spring JPA library[18] and Hibernate, the backend connects to PostgreSQL in an agnostic manner. Essentially, Spring provides a set of database-agnostic Spring components called *repositories* that allow the specification of queries using Java functions. Each function represents a query, and the function arguments are parameters passed on to the query in a secure manner, bounding the user-specified arguments and avoiding SQL-injection attacks. As the backend is deployed, Spring will read these classes and translate the required queries into PostgreSQL-specific queries. While these repositories allow queries to be specified using only the function name, a *@Query* annotation can be used to manually write the necessary query using Java Persistence Query Language (JPQL), a JPA SQL-like language. This abstracted connection corresponds to a hotswap-like architecture that adapts to any configured relational database, and can be swapped at any point. Data can be migrated to a different data store on-demand, and no changes to the backend codebase are required to accommodate this migration.

[16]`https://spring.io/projects/spring-security`
[17]`https://github.com/jwtk/jjwt`
[18]`http://spring.io/projects/spring-data-jpa`

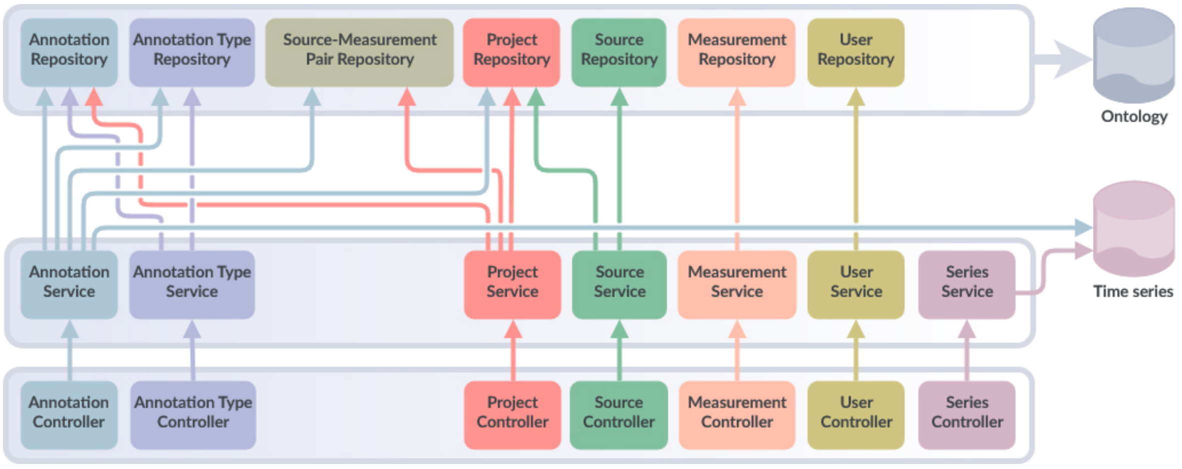


Figure 4.2: Controller to Service to Repository association

Beyond repositories, the custom logic that converts a request by the client, intercepted by a REST controller, into a well-formed request to a repository method, which in turn represents a query to the database, is implemented in a third and final Spring component called *service*. Whereas the JWT filter and REST controllers match the Authentication, Payload deserialization and Result serialization phases of the processing pipeline (3.3.4), the remaining aspects of the pipeline are implemented in services. Even though the backend could be entirely arranged as a single service that deals with all available request calls and repositories directly, this would translate into a monolithic component that is harder to maintain. By following best practices in Spring backend development and applying the design principle of SoC, the backend is structured with one service and one repository for each of the active entities, as shown in Figure 4.2. Note, however, that each service may need to access multiple repositories beyond the one that is assigned to its entity. In general, each service will make one or more queries to the database in other to fetch relevant data and validate the custom set of constraints (3.3.6), and will only return a single object or a list of objects with the respective entities that the service handles.

With this, the common roadmap for any query, insertion, update or deletion operation is as follows:

- **i)** the user sends a request with a set of URL query parameters and/or with a request payload, which is intercepted by one of the implemented REST controllers;
- **ii)** the request URL parameters and payload (if available) is deserialized into Java-equivalent data types, such as Java primitives for numbers and boolean flags, the Java String class, or the data structures available in the Java Collection Framework[19];
- **iii)** the set of Java objects derived from a request are sent into the respective service component;
- **iv)** the service validates the request by querying relationship validation checks in auxiliary repositories, and if these pass, it redirects the request to its respective repository;
- **v)** the result is returned to the controller and sent back to the user, serialized in JSON.

[19]`https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`

In the case of asynchronous updates (3.3.7), the *CompletableFuture*[20] API is used to create an asynchronous thread. Essentially, the repository method of *save*, for inserting data into the database, is called inside a *CompletableFuture* thread. All instantiated *CompletableFuture* use the same executor, which already handles concurrent instances and queues them by design when the server is under heavy load.

Whenever a cached object requires eviction after an update, the Spring API for managing caches can be used to evict the necessary object by specifying its key, triggering an eviction in Redis. These API calls are added as a hook to the above mentioned *CompleteableFuture.* The *CompleteableFuture* API contains a *whenComplete* method that allows other functions to be executed asynchronously after the original asynchronous process. With this, all *Completeable-Future* generated to update entities also contain a *whenComplete* function that will verify if the entity was successfully updated or rolled-back, and if so, will evict the respective lookup by ID caches. This way it is assured that the cache will evict obsolete objects after a commit to PostgreSQL, and not before, guaranteeing that any new queries will fetch these new results from PostgreSQL directly. An eviction is preferred to a replacement of the value in the cache, as the user will often update an object but not require it immediately after. Bulk-updates or bulk-rollbacks will proceed similarly, with the addition that all updated entities are iterated and evicted one by one, as Spring caching API does not provide a way to evict multiple objects by specifying a list of keys.

When an annotation is updated in order to cover a different set of series data points, an additional asynchronous function is hooked to the *CompleteableFuture* that was used to commit this update to PostgreSQL, using the *whenComplete* method. This function propagates an update to affected series points in InfluxDB if the annotation update succeeds, querying the series that were previously annotated and removing the respective annotation ID label, as well as the corresponding annotation type ID label and project ID label. Afterwards, the newly annotated series are queried, and the annotation is labeled appropriately in these. Finally, both old series points and new series points are committed to InfluxDB simultaneously.

4.3.4 Time series query

While Spring already provides database client implementations for various databases, such as MySQL, PostgreSQL, Redis, Elasticsearch and Cassandra, among many others, there are no database clients for InfluxDB. This means that there is no built-in way to connect the InfluxDB database to a Spring repository for series, so instead, the connection and query logic must be implemented in the prototype using the InfluxDB Java driver[21]. In order to maintain uniformity with the way ontology processing is modeled, this custom logic is implemented in a *SeriesService* class (as shown in Figure 4.2).

Once the series controller receives a series query object, this object is sent to a series service component where it will be properly validated. The structured series query object is a simple JSON object that follows the blueprint in 3.3.5, as shown in code snippet 8. Once validated, the

[20]`https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/CompletableFuture.html`
[21]`https://github.com/influxdata/influxdb-java`

```json
{
  "start": "2018-03-11T10:00:00Z",
  "end": "2018-03-13T14:00:00Z",
  "params": [
    {
      "source": "AUW41vuHRTWRzM16Cr2CTQ",
      "measurement": "pQ57aanNQAq5POHukLikiA",
      "filter": [
        {
          "rule": {
            "field": "numericValue",
            "op": "GREATER_THAN",
            "value": "4"
          }
        },
        {
          "cond": "AND",
          "rule": {
            "field": "numericValue",
            "op": "LESS_THAN_OR_EQUAL",
            "value": "10"
          }
        }
      ]
    },
    {
      "source": "2ZpjC7OgS3uSj12BFF6Fmw",
      "measurement": "yHSBj2OLQiyv-xs8AeJo4Q",
      "filter": [
        {
          "rule": {
            "field": "stateValue",
            "op": "EQUAL",
            "value": "true"
          }
        }
      ]
    }
  ],
  "filter": []
}
```

Code 8: Structured time series query object, which lists the exact segment and source-measurement pairs that will be queried, along with a few optional filters that apply only to their respective series.

query object is parsed and transformed into a query syntax similar to the one seen in the code snippet 4, converting a structured object into a valid InfluxQL query. This structured object mandatorily requires a start timestamp, an end timestamp, and a list of source-measurement pairs that will be queried. There needs to be at least one source-measurement pair, and any filter (global or pair-specific) is completely optional. Most of the user-specified parameters are sent to InfluxDB using the bound parameter API[22] in order to avoid injection attacks. However, the Java implementation of the bound parameter API is limited, as it only accepts field names and string values, but not measurement names or other serialized operators. This means that the remaining user-specified parameters (measurementId, rule operators and start and end dates) are required to be validated manually.

The InfluxDB Java driver already maps the results derived from InfluxDB into typed Java objects, but the response object structure does not fit the prototype needs for expedited time series query and display. The charting library used in the frontend, Dygraphs[23], uses an light-weight array format to read and parse time series, where all series points are structured as arrays of arrays in a similar way to a matrix representation. For a sorted list of series with source-measurement pairs G1-P1, G2-P2 and G3-P3, this format would correspond to a list of arrays where each array contains a timestamp in the Epoch format as the first item and the values displayed for each timestamp in the remainder of the array. In other words, each array corresponds to a timestamp that has values for one or more of the series, where the second position of the array always contains values of G1-P1, the third position always contains values of G2-P2, and the fourth position always contains values of G3-P3. If any of the series do not have values for a specific timestamp, their respective position in the array will contain a null value. This format supports values with either numeric and boolean values. Because this format is one that grants Dygraphs a high speed of parsing and displaying the time series, the series service transforms the custom-made serialized objects of InfluxDB Java driver into this matrix-like format. In this manner, the frontend immediately receives the complete matrix of points and feeds it into its chart directly.

4.3.5 Time series update

An important aspect of the architecture is that while a database like PostgreSQL has a strong MVCC model that allows concurrent reads and writes, InfluxDB does not have those mechanisms built-in. Assuming that two concurrent updates are sent, each one adding links for different annotations but in the same exact series points, InfluxDB has no way of checking if the two updates are applying over the same records, potentially overwriting each other, nor does it have a mechanism to automatically merge both changes together so that both updates survive. Instead, when multiple simultaneous changes on the same exact series are propagated to InfluxDB in parallel, the only changes that would be effectively committed to the database are the ones being written last.

As described in a prior subsection (4.3.3), updates to InfluxDB are always hooked to

[22] `https://docs.influxdata.com/influxdb/v1.7/tools/api/#bind-parameters`
[23] `http://dygraphs.com/`

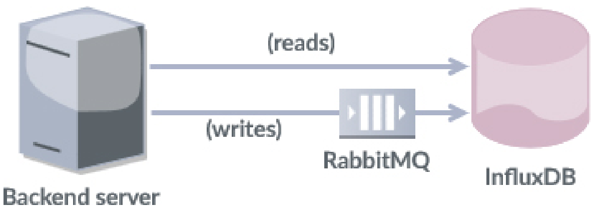


Figure 4.3: Backend connection with InfluxDB in the system architecture

successful updates applied to the ontology in PostgreSQL, which already contains optimistic locking and atomicity. If the optimistic locking of PostgreSQL fails, then the series changes are not even proposed to InfluxDB. Nevertheless, this alone does not prevent two simultaneous updates that affect the same series to be propagated to InfluxDB in parallel.

To fix this, the architecture was devised so that all InfluxDB write requests are placed inside a single RabbitMQ queue. This FIFO queue provides a sequential channel of write proposals, where each write proposal in the queue will make a new query over the series that require an update and make changes to these. For overlapping writes, each proposal will query the data that was written in the previously polled proposal. This way, all writers are executed sequentially and will never overwrite over each other, but the overall amount of time required for the entire data set to converge is longer. This is an acceptable drawback to ensure that the system is *eventually-consistent* without any data loss, especially since the architectural focus for time series data is to follow an E+L model.

4.4 Visualization

The frontend application is implemented as a dynamic website in ReactJS[24], using TypeScript[25] as the primary programming language and compiler. The application is highly modular, separating logic between multiple interface modules that can be re-rendered over time depending on the data contained in them. Some modules were written using JavaScript with the ECMAScript 6 (ES6 for short) specification, for cases where a less strict compiler was required. The development stack is configured in Webpack[26], which compiles all of the code and packs it into a set of compressed Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript files that are then interpreted by the browser of the end-user. The entire frontend codebase is validated using a Babel[27] compiler, ESLint[28] linting and Stylelint[29], ensuring good code quality that is highly maintainable and that follows standard practices and style guides in the software industry. Along with these, Prettier[30] automatically formats the code when committed to a distributed version control system like Git, maintaining

[24]https://reactjs.org/
[25]https://www.typescriptlang.org/
[26]https://webpack.js.org/
[27]https://babeljs.io/
[28]https://eslint.org/
[29]https://stylelint.io/
[30]https://prettier.io/

a consistent code style between developers of the application. Finally, Moment.js[31] is used for all chronology operations and modeling requirements.

To implement UI elements, the Ant Design framework[32] is used, as it already contains a wide variety of ReactJS and TypeScript compliant components for layouts, navigation, data display, forms, popups and popovers. For styling the UI elements and defining layouts, the Leaner Style Sheets (LESS)[33] stylesheet pre-processor is used. LESS stylesheets are written in a language that iterates on CSS by providing support for mixins and utility functions, nested rules that contribute to a cleaner code, and variables, among other features. Additionally, PostCSS[34] is used to leverage CSS modules[35] throughout the implemented code, allowing the ReactJS codebase to reference specific LESS style classes.

4.4.1 Data mapping

The usage of TypeScript was vital to ensure a type safe coding experience when dealing with the tightly modeled entities and requests. All of the entities that were modeled in Hibernate while in the backend (4.1.2) are modeled in TypeScript while in the frontend, and every response payload from the backend is deserialized into these entities, which means that communication attempts that deviate from the established REST API schema (3.3.2) will not be valid, and therefore discarded. Every field in each of the Hibernate entities, which has a Java data type, is converted into a TypeScript equivalent: a Java *String* becomes a *string*; a Java *Boolean* becomes a *boolean*; a Java *Long*, *Double* or *Integer* becomes a *number*; and every Java *ZonedDateTime* or *Instant* becomes a *Moment* from Moment.js. Moreover, because the response payload structure is deterministic and parsable, these are also modeled in TypeScript with their equivalent types. For example, because a response to a listed query of annotations will always return a list body of annotations, the developer can validate that the response is in fact a list by checking if the *resultType* field is "list", and if so, the fields *totalCount* and *totalPages* should be available. If the response contains a non-null *errorType*, then the *result* is assuredly null and the developer should treat this response as an error, displaying the *error* field or any other custom message that fits the *error type* to the user.

This forced structuring and compatibility of response payloads is especially useful in a reactive environment such as ReactJS, where these responses can be stored in a single data structure and updated over time by subsequent requests, and other modules and pages can then subscribe to this object if they require it, receiving re-rendering dispatches when this object is available and/or changed.

[31]`https://momentjs.com/`
[32]`https://ant.design`
[33]`http://lesscss.org/`
[34]`https://postcss.org/`
[35]`https://github.com/css-modules/css-modules`

4.4.2 Application state

Redux[36] is used to handle application state and reactive propagation of data throughout all application modules, while axios[37] is used as an HTTP client that communicates with the backend's REST API. Both of these enable a reactive environment where requests are triggered asynchronously without requiring the UI thread to be blocked from further actions by the user, and the ReactJS interface itself then "reacts" by re-rendering only the specific components that display the fetched data. Any further re-querying of said data will automatically trigger a re-rendering of these components alone, and any local changes made to objects in one section will immediately be dispatched into other sections that re-use it.

When a query is sent to the backend, this will dispatch a Promise[38] that waits for results asynchronously. As results are expedited from the backend to the frontend, a Redux reducer will intercept this response and reference it in a data structure. For example, the response could be an object body from a lookup-by-id query, which is then referenced in a *byId* field, and throughout the application, any module that is visible to the user and that subscribes to this *byId* field will be re-rendered, displaying the new data in the interface.

The design pattern of SoC that was applied to model the ontology in the backend 4.3.3 is used in the frontend as well, not only for modeling entities but also for modeling Redux reducers. There are six reducers, each corresponding to one of the six entities of the ontology. A base reducer contains a set of baseline data structures that store the results from all of the implemented REST requests, such as list-bodies or object-bodies. In this manner, all reducers have predictable, type safe behavior that matches the homogeneous REST interface (4.3.1). A seventh reducer is implemented specifically for the query of series, which matches the available series query *POST* call in the implemented series controller (4.3.4).

Because the application will sometimes have to display the same entity across different locations (e.g. the types of each annotation in the chart are also shown in the project details side-panel and in the annotation types management table), unnecessary requests for objects that were already fetched in the platform can be prevented. After fetching an entity, subsequent needs for that same entity will simply use the already fetched one until a page reload, instead of fetching a new one. After a long work session where a user queried a set of entities at the start and never refreshed them, the mechanism of optimistic locking (3.3.8) in the backend will ensure that the user cannot commit changes based on stale data.

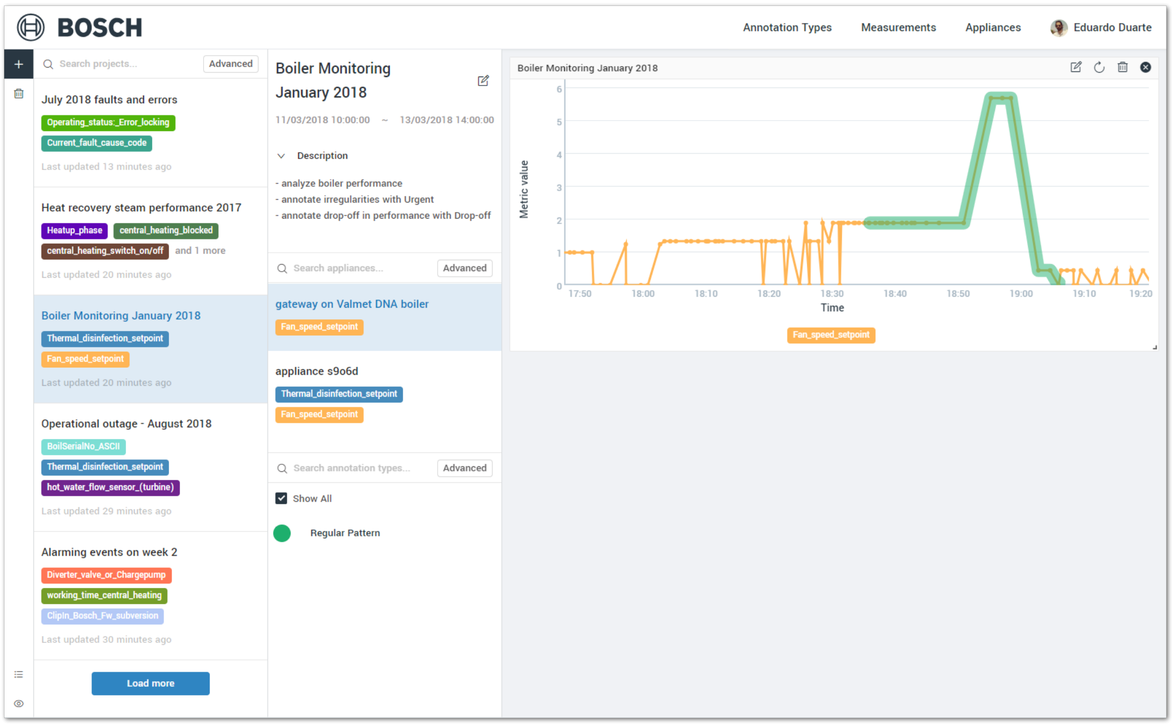
The implemented reducers obey the authentication protocol described in 3.3.3, as every HTTP call made to the backend contains the user's authentication token in the Authorization header. When a response is received from the backend, the refreshed token in the response's Authorization header is stored in the application, replacing the previously stored token. The token is stored in a LocalStorage[39], which is supported by all modern browsers, in order to survive page reloads.

[36] `https://redux.js.org/`

[37] `https://github.com/axios/axios`

[38] `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`

[39] `https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage`



July 2018 faults and errors
Operating_status:_Error_locking
Current_fault_cause_code
Last updated 13 minutes ago

Heat recovery steam performance 2017
Heatup_phase
central_heating_blocked
central_heating_switch_on/off and 1 more
Last updated 20 minutes ago

Boiler Monitoring January 2018
Thermal_disinfection_setpoint
Fan_speed_setpoint
Last updated 20 minutes ago

Operational outage - August 2018
BoilSerialNo_ASCII
Thermal_disinfection_setpoint
hot_water_flow_sensor_(turbine)
Last updated 29 minutes ago

Alarming events on week 2
Diverter_valve_or_Chargepump
working_time_central_heating
Clipin_Bosch_Fw_subversion
Last updated 30 minutes ago

Boiler Monitoring January 2018
11/03/2018 10:00:00 ~ 13/03/2018 14:00:00
Description
- analyze boiler performance
- annotate irregularities with Urgent
- annotate drop-off in performance with Drop-off

gateway on Valmet DNA boiler
Fan_speed_setpoint

appliance s9o6d
Thermal_disinfection_setpoint
Fan_speed_setpoint

Show All
● Regular Pattern

Boiler Monitoring January 2018

Figure 4.4: Dashboard displaying a list of projects and details for a project that queries multiple sources. Only one of the sources is selected.

4.4.3 Dashboard and management

Following the blueprint proposed in section 3.4.1 while using Ant Design components, the implemented dashboard contains multiple modules with various entities in simultaneous display, as shown in Figure 4.4. The reactive environment enabled by ReactJS and Redux is advantageous in this section, as any changes made to an entity anywhere in the nested panel structure of the dashboard will be propagated and reflected in other parts of the dashboard. The state of selected projects and selected sources is recorded in the browser's address bar through the URL, surviving page reloads. The URL is constantly changed as the user selects different projects or sources, and can be shared with other collaborators of the same projects in order to open the exact same view of the dashboard in their own browsers.

As proposed in the visualization model, the prototype also supports multi-selection of projects and sources (Figure 4.5), displaying projects in a split-space view within independent chart windows and sources of the same project in a shared-space view within the same chart. Additionally, it allows chart windows to be resized and panels to be hidden individually, expanding the horizontal space available for chart windows.

The ontology management pages follow their wireframes closely (Figure 3.15). As with the dashboard, performing a search using the given form components will change the URL in the address bar, allowing users to share a view of the management page with a pre-established search criteria. When pressing the "Rollback changes" button, the revision list is displayed in a popup dialog. Figure 4.8 shows the revision list for an annotation type, where the user can rollback all changes up to any version by pressing the respective "Rollback" button.

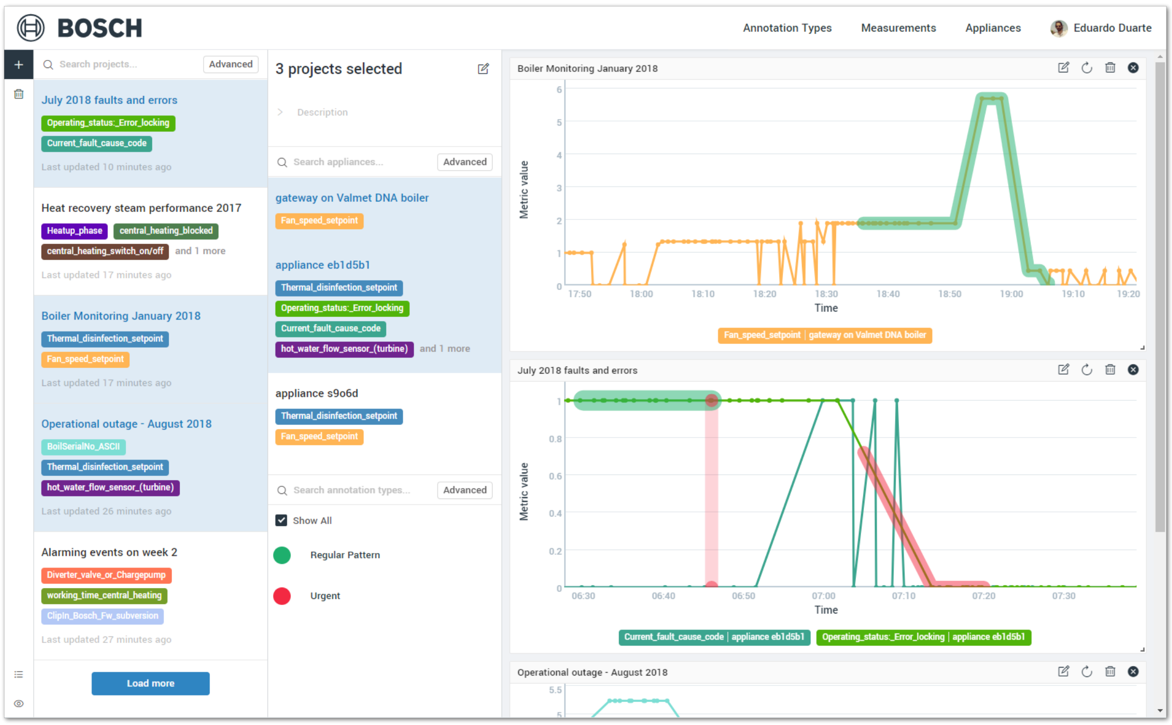

BOSCH
Annotation Types
Measurements
Appliances
Eduardo Duarte
Search projects...
Advanced
July 2018 faults and errors
Operating_status:_Error_locking
Current_fault_cause_code
Last updated 10 minutes ago
Heat recovery steam performance 2017
Heatup_phase
central_heating_blocked
central_heating_switch_on/off
and 1 more
Last updated 17 minutes ago
Boiler Monitoring January 2018
Thermal_disinfection_setpoint
Fan_speed_setpoint
Last updated 17 minutes ago
Operational outage - August 2018
BoilSerialNo_ASCII
Thermal_disinfection_setpoint
hot_water_flow_sensor_(turbine)
Last updated 26 minutes ago
Alarming events on week 2
Diverter_valve_or_Chargepump
working_time_central_heating
Clipln_Bosch_Fw_subversion
Last updated 27 minutes ago
Load more
3 projects selected
Description
Search appliances...
Advanced
gateway on Valmet DNA boiler
Fan_speed_setpoint
appliance eb1d5b1
Thermal_disinfection_setpoint
Operating_status:_Error_locking
Current_fault_cause_code
hot_water_flow_sensor_(turbine)
and 1 more
appliance s9o6d
Thermal_disinfection_setpoint
Fan_speed_setpoint
Search annotation types...
Advanced
Show All
Regular Pattern
Urgent
Boiler Monitoring January 2018
Metric value
Time
Fan_speed_setpoint | gateway on Valmet DNA boiler
July 2018 faults and errors
Metric value
Time
Current_fault_cause_code | appliance eb1d5b1
Operating_status:_Error_locking | appliance eb1d5b1
Operational outage - August 2018

Figure 4.5: Dashboard displaying a list of projects and details for three projects that query multiple sources. Two out of the three total sources are selected. The source identified as "source eb1d5b1" contains the measurements from two separate projects, and this information is merged and displayed on the source list.

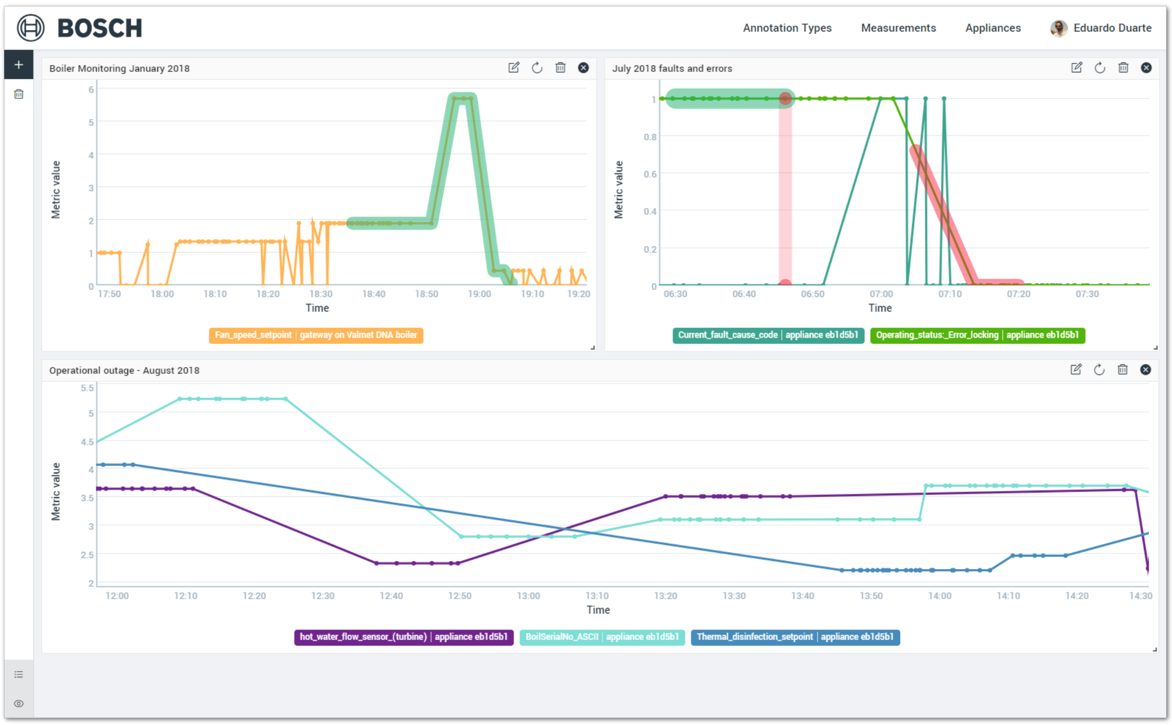
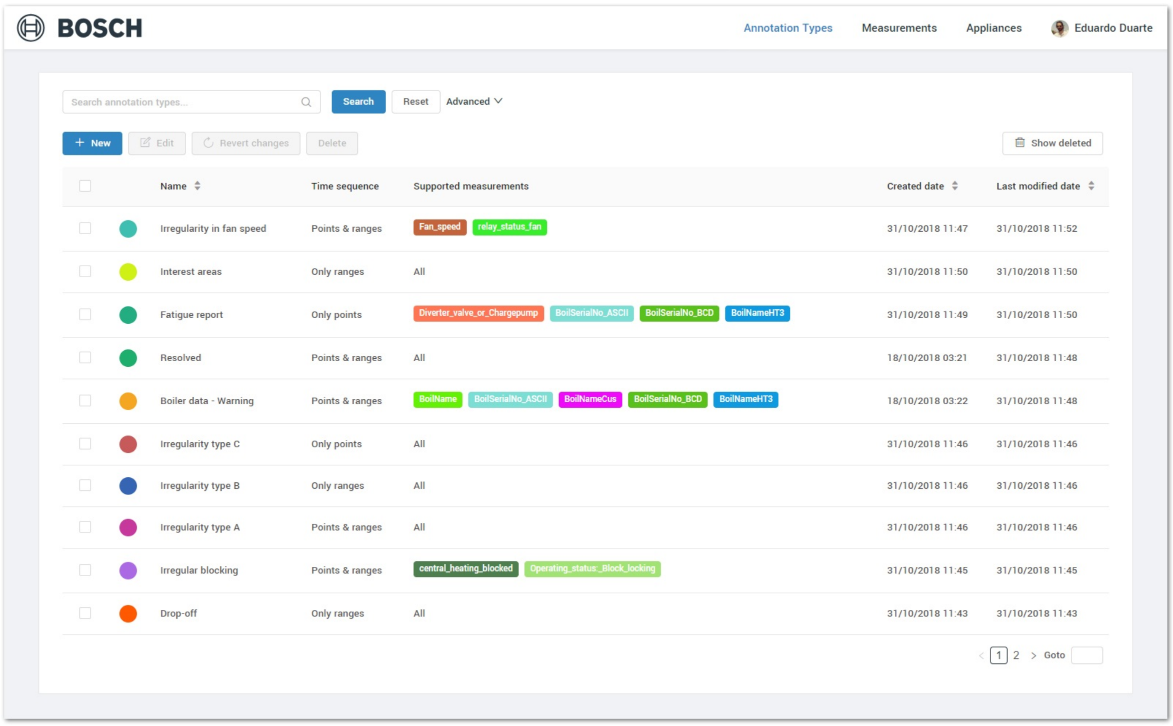
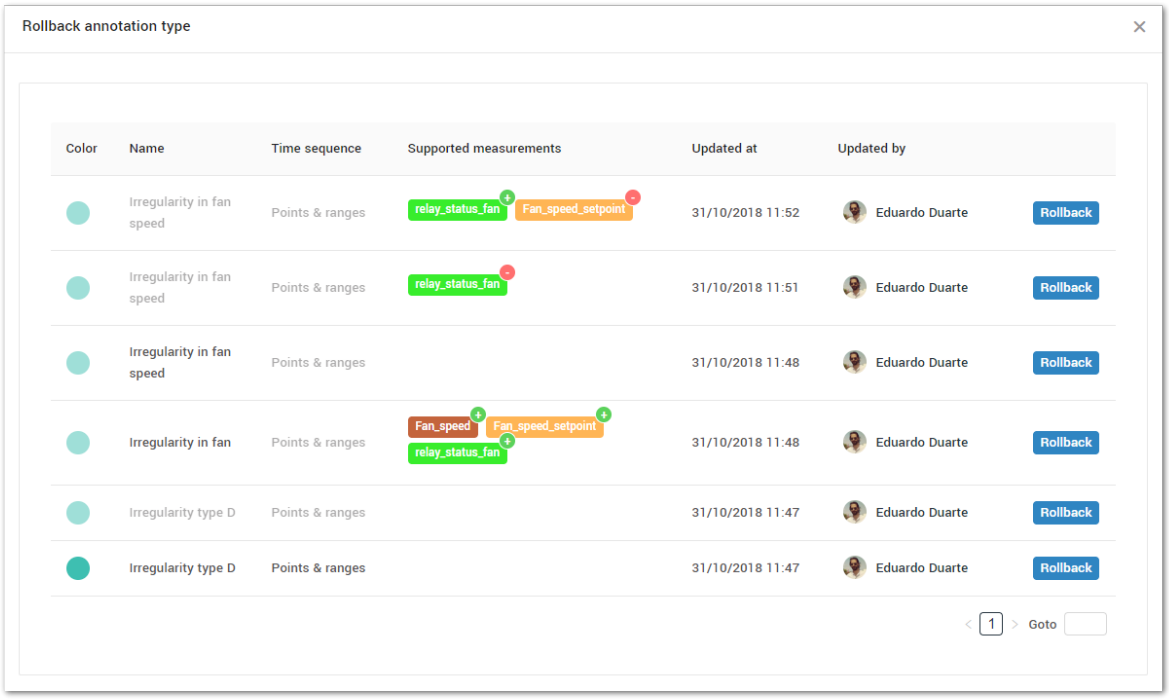

BOSCH
Annotation Types
Measurements
Appliances
Eduardo Duarte
Boiler Monitoring January 2018
Metric value
Time
Fan_speed_setpoint | gateway on Valmet DNA boiler
July 2018 faults and errors
Metric value
Time
Current_fault_cause_code | appliance eb1d5b1
Operating_status:_Error_locking | appliance eb1d5b1
Operational outage - August 2018
Metric value
Time
hot_water_flow_sensor_(turbine) | appliance eb1d5b1
BoilSerialNo_ASCII | appliance eb1d5b1
Thermal_disinfection_setpoint | appliance eb1d5b1

Figure 4.6: Dashboard displaying the same selection as Figure 4.5, but the panels are hidden and two of the charts are resized.



		Name	Time sequence	Supported measurements	Created date	Last modified date
☐	●	Irregularity in fan speed	Points & ranges	Fan_speed relay_status_fan	31/10/2018 11:47	31/10/2018 11:52
☐	●	Interest areas	Only ranges	All	31/10/2018 11:50	31/10/2018 11:50
☐	●	Fatigue report	Only points	Diverter_valve_or_Chargepump BoilSerialNo_ASCII BoilSerialNo_BCD BoilNameHT3	31/10/2018 11:49	31/10/2018 11:50
☐	●	Resolved	Points & ranges	All	18/10/2018 03:21	31/10/2018 11:48
☐	●	Boiler data - Warning	Points & ranges	BoilName BoilSerialNo_ASCII BoilNameCus BoilSerialNo_BCD BoilNameHT3	18/10/2018 03:22	31/10/2018 11:48
☐	●	Irregularity type C	Only points	All	31/10/2018 11:46	31/10/2018 11:46
☐	●	Irregularity type B	Only ranges	All	31/10/2018 11:46	31/10/2018 11:46
☐	●	Irregularity type A	Points & ranges	All	31/10/2018 11:46	31/10/2018 11:46
☐	●	Irregular blocking	Points & ranges	central_heating_blocked Operating_status:_Block_locking	31/10/2018 11:45	31/10/2018 11:45
☐	●	Drop-off	Only ranges	All	31/10/2018 11:43	31/10/2018 11:43

Figure 4.7: Annotation type management page



Color	Name	Time sequence	Supported measurements	Updated at	Updated by	
●	Irregularity in fan speed	Points & ranges	relay_status_fan + Fan_speed_setpoint -	31/10/2018 11:52	Eduardo Duarte	Rollback
●	Irregularity in fan speed	Points & ranges	relay_status_fan -	31/10/2018 11:51	Eduardo Duarte	Rollback
●	Irregularity in fan speed	Points & ranges		31/10/2018 11:48	Eduardo Duarte	Rollback
●	Irregularity in fan	Points & ranges	Fan_speed + Fan_speed_setpoint + relay_status_fan +	31/10/2018 11:48	Eduardo Duarte	Rollback
●	Irregularity type D	Points & ranges		31/10/2018 11:47	Eduardo Duarte	Rollback
●	Irregularity type D	Points & ranges		31/10/2018 11:47	Eduardo Duarte	Rollback

Figure 4.8: Revision list for an annotation type, sorted by newest-to-oldest, identified by the timestamp when the changes were made and by the user that triggered the changes. Grayed out fields, or empty in the case of relationship fields, correspond to fields that were not changed in that specific version, but rather on a previous one.

4.4.4 Annotations

The Dygraphs library is used to create the time series charts, performing well when rendering line graphs of massive quantities of series points and natively handling interactions from the user, such as zooming and panning over the graph. Dygraphs renders all series, X and Y axis, and labels on top of an HTML 2D Canvas[40]. The Canvas API, as well as the CanvasRenderingContext2D API[41] provide enough functionality in order to paint any shape as desired. Dygraphs does not provide direct access to the underlying canvas in its API, but it supports the attaching of plotters, which are light-weight functions that are rendered on every single interaction with the chart, including mouseover, clicks, zooms and pans. Plotters provide open access to a second canvas object that is positioned on top of the series graph canvas. The frontend prototype uses this plotter and Canvas APIs extensively in order process the required *snakes* of annotations that follow the series curves, as proposed in 3.4.2.

First, two variables are defined at an initial stage of rendering. These variables are the annotation width (in pixels), which corresponds to the initial radius that *snakes* will have, and the width increment (in pixels), which corresponds to the number of pixels that are incremented on top of the initial annotation radius in order to nest other overlapping *snakes.* Then, for each of the required annotations, the exact points in the Document Object Model (DOM)[42] that match the start and end of the annotation's time range must be discovered. This is done by iterating over every single point in the series data and matching it with the annotation timestamps. In the case of a point annotation, it is only necessary to find a single data point in the series that matches with the annotation timestamp.

However, when starting, ending or point timestamps are placed at a moment in time where there are no authentic data points, but rather a line that connects a past data point with a future one, then this point must be discovered through interpolation. The previous and next data points are found through timestamp comparison, by finding the nearest two points in which the annotation's start, end or point timestamp is between, and their respective values are then obtained. All of the series authentic data points are provided in the plotter API with their respective values. By using the previous real point 0 and the next real point 1, with timestamps t_0 and t_1 and values v_0 and v_1, the function in (4.1) can be used to obtain an interpolated Y value of an annotation that starts, ends or is a point in X.

$$Y = v_0 + (X - t_0) \ / \ (t_1 - t_0) \cdot (v_1 - v_0) \tag{4.1}$$

With this, the exact DOM location where a point annotation sits within the series, or where a region annotation has its starting or ending position, can be used to trace the corresponding *snake.* The remaining anchor points for a region *snake*, which are used to render a *snake* that follows the series curve between its starting and ending locations, corresponds directly to all of the real series points in-between. A line shape can be drawn between all these points with the initial annotation width, effectively drawing a *snake* over a series curve.

[40]https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API
[41]https://developer.mozilla.org/en-US/docs/Web/API/CanvasRenderingContext2D
[42]https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

There are a few optimizations applied over the described algorithm. For example, if the currently-viewed interval of time contains only a segment of the full annotation, because its starting and/or ending points are outside of the zoomed range, then a line shape can be drawn over all visible points without needing for further calculations. Additionally, if the current zoom hides just the starting point or just the ending point of the annotation, the first or the last visible series point can respectively act as stand-ins for these. Moreover, drawing global annotations does not require this algorithm, as a simple rectangle that occupies the entire vertical space can be drawn from the starting timestamp to the ending one.

Once all annotation *snakes* have known locations in the DOM, their rendering priority is sorted using the rules established in the proposed visualization model (3.4.2). All annotations are stored in a temporary array data structure, and the rendering priority is implemented in the form of a comparator pattern that sorts this array. When an annotation is added to the array, the comparator is executed in order to find the sorted position for this annotation. This comparator also fulfills the nesting rules, incrementing *snakes* sizes: if two point annotations intersect one another, the point annotation that is rendered below the other is incremented, enlarging it to accommodate the annotation above it; and if two region annotations intersect one another, then the one that starts and/or ends inside the other will be painted above, so that the annotation below is enlarged. This comparator ensures that the width is incremented multiplicatively, as the width increment property is added with the annotation's current radius multiple times based on the number of nested annotations.

Afterwards, it is necessary to render shapes that connect the same annotation displayed on separate series/measurements. In the case of point annotations, a line shape is drawn between all of the point *snakes*, resulting in a vertical line. In the case of region annotations, the top-left, top-right, bottom-left and bottom-right corners for all annotated points across all visible series are identified, composing a baseline for a rectangular area that covers all the region *snakes*.

The final step is to implement a rendering plotter that creates a lighter tint over mouse-hovered *snakes*. The mouseover interaction in the chart is captured and directly sent to this plotter, which will in turn detect if the mouse coordinates would be affecting a rendered *snake* or not. This is done by going over all of the painted points (both authentic and interpolated ones) and determining if the mouse coordinates intersect the area determined by the *snake* coordinates and their established radius, based on the rendering comparator that incremented them. If so, then the entire corresponding *snakes* and overlays are re-rendered in the plotter with a lighter shade of color. However, this would mean that the highlight could be rendered on top of non-selected nested shapes as well. To prevent this, every shape from other annotations that are painted in the same area as the highlighted annotation are repainted with a global composite operation of "destination-out". In other words, every shape that comes before the last highlighted shape in the temporary array of annotations, which is already sorted by rendering priority, is re-painted with "destination-out" composite, which effectively clears the highlight from those areas in the canvas.

CHAPTER 5

Evaluation

5.1 Architecture benchmark

In the State of the Art chapter, multiple approaches for storing massive time series data sets were studied (2.3.1), which ranged from using RDBMSs, TSDBMSs and CDBMSs, and existing benchmarks were leveraged to evaluate how each storage tool performed in this task. These studies suggested that InfluxDB was the ideal long-term storage tool for time series, which motivated its usage in the presented platform. However, as was previously suggested (3.2.1), in a distributed system that focuses on *consistency* the software architect should take into consideration that, as the granularity of data increases, the harder it is to attain high *consistency.* Moreover, InfluxDB has no built-in mechanisms to prevent overlapping writes and data loss, prompting the need to build them manually in the architecture stack (4.3.5).

Throughout the development of the prototype, the necessary queries proved to be quite performant for the planned use cases while using a massive HVAC data set, and the limited data model of InfluxDB was sufficient for storing series and their numeric, state or string values, as well as to index series that are identified by their data source and their measurement. However, this limited data model handicapped the ability to efficiently structure links for annotations, annotation types and projects. While this did not correspond to a major drawback with the tested data set, it could still lead a notable shortcoming of the platform when needing to scale this data set or to evolve the data model, in order to include more data types and relations with the ontology.

One major caveat that was found while using InfluxDB was that the database would timeout write requests when attempting to store batches with more than approximately three hundred thousand (300 000) time series points. This problem is not new, and has been documented for as long as 2015[1]. Both the amount of points and the number of fields in each will affect the ingestion rate, which can be calculated by multiplying both. For example, when attempting to write a batch of three hundred thousand points with five fields each, it would correspond to an ingestion rate of 1.5 million writes-per-second, and when attempting

[1]`https://github.com/influxdata/influxdb/issues/3349#issuecomment-122071094`

to write a batch of five hundred thousand (500 000) points with the same number of fields, it would lead to an ingestion rate of 2.5 million writes-per-second. For the machine that was used during testing and quality assurance (2.50GHz Quad-Core, 16GB Random-access memory (RAM)), the maximum ingestion rate that was recorded without timed-out requests was 1.8 million writes-per-second. Essentially, InfluxDB does not have a partitioned save feature built-in, so it cannot control the amount of series that are ingested per second in order to not exceed the CPU and memory constraints of the host machine, leading the engine to crash and the connection to timeout. In order to solve this issue, the prototype was forced to have a partitioned save algorithm that writes all metrics in smaller batches of three hundred thousand metrics or less at a time.

The existing benchmarks for time series storage in RDBMSs (2.3.1) is slightly inconclusive due to the lack of mature and battle-tested databases built around RDBMSs with time series in mind. During the planning phase prior to development, the choice of using a RDBMS for the ontology and a separate TSDBMS for time series, as illustrated in Figure 3.3, over using only a RDBMS for the entire data set, as illustrated in Figure 3.5, was still an uncertain one. With this, it is important to examine how would the implemented system behave and perform if both the ontology of annotations and the time series data set were stored in the same RDBMS. A set of benchmarking tests were implemented to evaluate how much of a performance improvement or drop off would be measured from storing the entire time series data set in PostgreSQL when compared to InfluxDB. The goal of these tests is to potentially recognize a threshold in which a performance drop could still be an acceptable trade-off for the potential improvements in reads, by polling series and annotations simultaneously through joins, and in writes, where one single atomic transaction would propagate changes to annotations and series at the same time.

For this, a series entity was modeled in JPA within the codebase, following the same pattern as other entities (4.1.2). Unlike InfluxDB points, which use labels to express related annotations, the new series entities model their association with annotations as a many-to-many relationship, and their association with measurements and data sources in a many-to-one relationship. There is no need to include a direct association between series and annotation types or projects because the series entity directly references a set of annotation entities, thus, series queries can be written to fetch these annotations and their hierarchy. Furthermore, the series query object (4.3.4) was extended in order enable its conversion into a query in the JPQL syntax. Similarly, the implementation of the series service component (4.3.3) was extended to process and send these queries to PostgreSQL through the use of the *Entity Manager* API[2]. In other words, a set of components was developed to extend on existing behavior rather than replace it, essentially providing a drop-in plugin. The two required architecture models illustrated in figures 3.3 and 3.5 are implemented in the same codebase, but through the use of Spring profiles, the RDBMS-only architecture model and code is only activated if a “ts-psql” profile flag is enabled on deployment. The result is a single backend *jar* file that can be launched in the same machine while toggling this flag on or off in order to

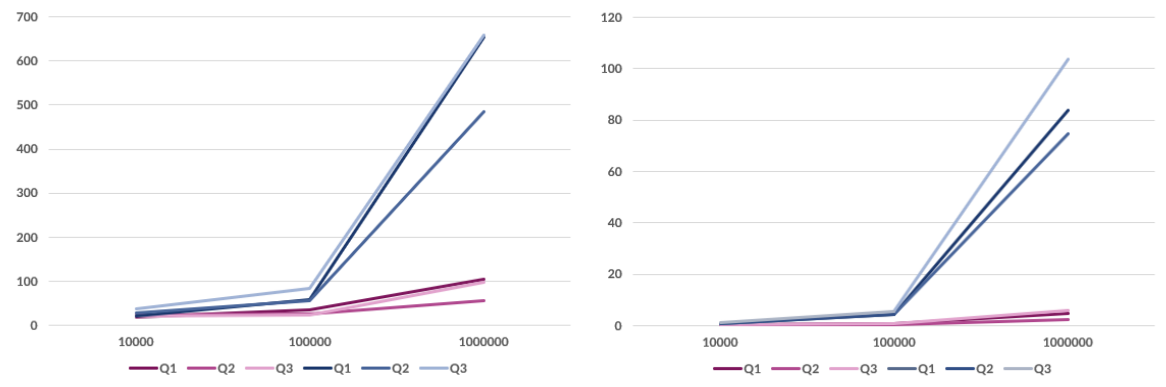
[2] `https://docs.oracle.com/javaee/7/api/javax/persistence/EntityManager.html`

test both architectures and data management models.

All tests were executed with the same set of rules, such as using the same machine and deployment conditions to run the backend server and required databases, and using the exact same queries in order to obtain a deterministic response. Three different queries with varying levels of complexity were tested to observe how the system behaves for data sets with ten thousand (10 000), a hundred thousand (100 000), and a million points (1 000 000). The queried segment of time was kept the same for all three queries, which were as follows:

- $Q1$ - get three source-measurement pairs, where the first and second pairs are from the same measurement and the second and third pairs are from the same source;
- $Q2$ - get the same three pairs as $Q1$, but two of the pairs contains numeric values, one being filtered for values greater than 4 while the other being filtered for values less than or equal to 2, and the third pair contains state values and is filtered for only values set to *true*;
- $Q3$ - get the same three pairs as $Q1$, but all series are filtered by annotation and annotation type.

Each of the queries was executed between ten to twenty times for each architecture while using a single backend node with a 2.50GHz Quad-Core processor and 16GB RAM memory. For read performance, the observed metrics were query speed (the elapsed time measured in milliseconds that a query takes to respond from the moment the request is sent) and CPU usage. Write performance was also evaluated by observing insertion speed (the elapsed time measured in milliseconds that a batch of points takes to write in the database), disk overhead and RAM memory consumption. The recorded RAM memory values correspond to an average measure that was constant throughout the entire writing operation. It is important to note that, for insertion tests, because all data is new and no existing data is being updated, the previously mentioned writing-queue (4.3.5) is disabled. Load balancing and request queuing mechanisms were also bypassed during both read and write tests.



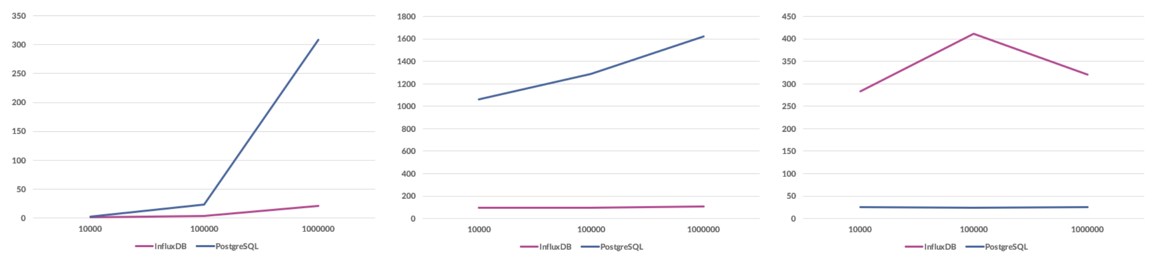
(a) Estimated time (in milliseconds) **(b)** CPU (in %)

Figure 5.1: Read performance as the data set size increases. Purple lines are InfluxDB, blue lines are PostgreSQL.

Table 5.1: Read benchmark: average query time and CPU usage observed for the 3 tested queries in the 3 data sets

10 000 points		**Q1**	**Q2**	**Q3**
InfluxDB	Speed	18.8 ms	22.4 ms	21.8 ms
	CPU	0.352%	0.354%	0.284%
PostgreSQL	Speed	21.8 ms	28.2 ms	37.6 ms
	CPU	0.656%	0.778%	1.046%
100 000 points				
InfluxDB	Speed	34.8 ms	24.8 ms	22.4 ms
	CPU	0.824%	0.5%	0.724%
PostgreSQL	Speed	59.2 ms	56 ms	84.6 ms
	CPU	4.17%	4.362%	5.714%
1 000 000 points				
InfluxDB	Speed	103.4 ms	56.2 ms	96.6 ms
	CPU	4.892%	2.53%	5.88%
PostgreSQL	Speed	653 ms	484.6 ms	659 ms
	CPU	83.934%	74.622%	103.884%

The benchmark results in Table 5.1 show that, for the smallest data set of ten thousand points, CPU usage and elapsed query speed do not differ severely between the two databases. For a relatively small data set with less than a hundred thousand data points, PostgreSQL reveals as the most attractive choice as it can keep up with InfluxDB in terms of performance while providing a strong relational schema and atomic transactions that support a *consistent* system instead of an *eventually-consistent* one. However, for any data set with more than a hundred thousand data points, the performance differences are noticeable, and InfluxDB outperforms PostgreSQL with higher query speed and lower CPU usage. For data sets that have one million series points or more, these differences are even more evident. The disproportionate



(a) Estimated time (in seconds) **(b)** Disk usage (in MB) **(c)** RAM usage (in MB)

Figure 5.2: Write performance as the data set size increases. Purple lines are InfluxDB, blue lines are PostgreSQL.

Table 5.2: Write benchmark: average insertion time, disk usage and RAM memory usage observed for the 3 data sets

10 000 points	**Speed**	**Disk usage**	**RAM usage**
InfluxDB	875.9 ms	93.36 MB	283.5 MB
PostgreSQL	2.945 s	1.064 GB	25.33 MB

100 000 points	**Speed**	**Disk usage**	**RAM usage**
InfluxDB	3.480 s	97.46 MB	412 MB
PostgreSQL	23.26 s	1.288 GB	24.61 MB

1 000 000 points	**Speed**	**Disk usage**	**RAM usage**
InfluxDB	21.40 s	108.3 MB	320.9 MB
PostgreSQL	5.139 min	1.624 GB	25.93 MB

increase of both request time and CPU requirement observed in the PostgreSQL database leads to the conclusion that PostgreSQL is not as scalable as InfluxDB for massive time series data sets.

The benchmark results in Table 5.2 show that, despite the reported issue where InfluxDB may crash and timeout when attempting to ingest a massive batch of series, it still showcased a higher insertion throughput than PostgreSQL. PostgreSQL took considerably longer to insert the full data set, but it inserted it in an atomic manner with the MVCC model, where the data set was only readable in the database once all of the series points were fully inserted. InfluxDB also offers significantly better on-disk compression than PostgreSQL, which is ideal for long-term storage of high amounts of historical data. As the data set grows exponentially, InfluxDB applies mechanisms that keep the full data set continually compressed. It is suspected that a big part of what makes InfluxDB so performant for reads even when strong compression mechanisms are applied, is that it leverages RAM memory to store a high amount of cached series and rollup windows. This is especially evident in the benchmark results, where the RAM usage of InfluxDB is always considerably higher than the one recorded for PostgreSQL.

In sum, PostgreSQL was shown to be cost-effective for small data sets, but not a viable option for massive ones. Additionally, the detected drop in performance for massive data sets is too high, to the point the added value of the relational and highly *consistent* model is not enough to balance it out. In an environment where the platform should be capable of storing and handling massive amounts of time series, InfluxDB stands out as the better solution. However, InfluxDB is not without its list of issues, many of which have already been mentioned. The InfluxDB data model is limited, and if the data cardinality were to also increase over time, InfluxDB performance would drop dramatically due to its reliance on the time-structured LSM merge tree, whereas PostgreSQL would potentially only see a moderate drop off [60]. By having implemented a careful architecture that takes this into consideration, and that keeps the cardinality of measurements to a minimum, it is possible to continue using InfluxDB for time series data in a scalable system.

5.2 Clicks to interaction

On the visualization side, it was important to test the minimum necessary amount of clicks that a user needs to perform in order to fulfill any specific task. A realistic workflow will typically correspond to a multi-task scenario, so each work session will require the summation of all of the minimum required clicks that were observed for each task. First, the most basic and unique tasks that correspond to this workflow were outlined, separated by starting point in the interface. Ideally, the interface should be capable of streamlining most interactions by following standard UI and UX design practices such as the three-click rule, and require the least amount of clicks as possible. More than three clicks are naturally permitted for as long as most of the clicks correspond to intuitive and passive actions.

From the dashboard and without any selected projects, the user can:

- A - go to any management page;
- B - display the chart of a project with a single source;
- C - display the chart of a project with multiple sources;
- D - search for a project by name;
- E - search for a project by annotation type, source or measurement;
- F - select a project and search for a source by name;
- G - select a project and search for a source by measurement;
- H - select a project and search for an annotation type by name;
- I - select a project and search for an annotation type by measurement;
- J - select a project and hide an annotation type from view;
- K - multi-select two projects;
- L - select a project with multiple sources and multi-select two sources;
- M - select a project and go to the Edit Project page.

Table 5.3: Minimum amount of clicks for each task in the dashboard

task	A	B	C	D	E	F	G	H	I	J	K	L	M
clicks	1	1	2	2	4	3	5	3	5	1	2	3	1

Table 5.4: Minimum amount of clicks for each task in a project chart

task	A	B	C	D	E	F	G	H	I
clicks	1	4	3	2	3	1	1	1	1

Table 5.3 shows the minimum amount of clicks that were estimated in order to perform a set of tasks in the dashboard. Due to the nested nature of the UI in this dashboard, the deeper is the end goal in terms of UI panels, the higher the number of clicks is. All tasks in the dashboard require an average of 2.5 clicks. However, because in a realistic environment the user will have one or more projects already selected, other tasks that involve using deeper panels in the nested UI, such as C, F, G, H, I, J and L, will lead a shorter number of clicks to reach their goal. In addition, because the frontend already detects if the project has only one source and automatically select it, the user will only need one click to select the project and open the chart in task A.

From any project chart:

- A - select an annotation;
- B - add an annotation;
- C - edit an annotation;
- D - review annotation changes;
- E - delete an annotation;
- F - go to the Edit Project page;
- G - review project changes;
- H - view deleted annotations;
- I - close the chart.

Table 5.4 shows the minimum amount of clicks that were estimated in order to perform a set of tasks in the chart component. Because annotations are clearly visible throughout the entire chart, and are only hidden when the user zooms in, all tasks that require selecting any annotation first, such as tasks A, C, D and E, will have a very low number of clicks. In average, tasks in the chart require 1.8 clicks. In a realistic scenario where a chart is populated with hundreds of annotations, any annotation with a very small range of time will require the user to zoom in so as to make these annotations easier to select, but each zoom-in requires only an additional click-and-drag. In task B, creating an annotation for any measurement and any segment of time requires a simple *Shift*-drag, two clicks to select an annotation type (the remaining mandatory field that is not yet filled), and then a final click on the Save button. In task E, deleting not only requires to select an annotation and click the Delete button, but also to accept the deletion during a confirmation dialog, contributing with an additional click while providing a safety measure.

Table 5.5: Minimum amount of clicks for each interaction in the project edit page

task	A	B	C	D	E	F
clicks	2	2	3	3	3	2

Table 5.6: Minimum amount of clicks for each interaction in the ontology management pages and in the edit popups

task	A	B	C	D	E	F	G	H	I	J	K	L	M
clicks	1	2	2	2	4	1	2	2	2	4	2	2	2

From the Edit Project page:

- A - change name and save;
- B - change guidelines and save;
- C - change series query and save;
- D - change annotation types and save;
- E - change collaborators and save;
- F - delete the project.

Table 5.5 shows how many clicks are required to change any attribute of a project, granted that all of these contain one extra click to press the Save button. Tasks in the Edit Project page have an average of 2.5 clicks. Because the extra click on the Save button is already included, and because multiple changes can be combined in the same edit process, then the number of clicks for two combined tasks will remain at a minimum of 2.5 clicks.

Finally, the clicks for all remaining tasks in the ontology management pages were analyzed, including their respective edit popups.

From any ontology management page (Annotation Types, Measurements or Sources):

- A - go to the project dashboard or to any management page;
- B - select an entity and open the edit popup;
- C - select an entity and open the revisions popup.

In the Annotation Types management page:

- D - add an annotation type;
- E - add an annotation type with at least one allowed measurement;
- F - open the deleted annotation types popup;
- G - select an annotation type and delete it.

In the Annotation Types edit popup:

- H - change an annotation type's name;
- I - change an annotation type's color;
- J - change an annotation type's allowed measurement.

In the Measurements edit popup:

- K - change a measurement's name;
- L - change a measurement's color.

Finally, in the Sources edit popup:

- M - change a source device's name.

As shown in Table 5.6, the overall number of clicks is quite low. The tasks that require most clicks, E and J, both involve opening the annotation type edit popup, swap to the Measurements tab, and add or change the allowed measurements there.

Looking at the overall clicks per task, the maximum amount of clicks that one single task might require is five, or three if selections have already been made in the dashboard. This means that most tasks are already well optimized for a very low number of interactions at their basic level. One of the major advantages of the dashboard interface design is the ability to nest and restrict scope, allowing different series in different segments, contained in separate projects and data sources, to be compared side-by-side. Because project and data source panels are always in view, the number of clicks to quickly swap between projects or source devices is considerably reduced from what would occur in a list-to-page design model.

There are a few drawbacks to the way the ontology updates are detached from the dashboard, existing instead in their own management pages. Users cannot create annotation types as they see fit within the chart, but instead have to go to the annotation type management page (1 click), add a new type there (minimum 3), and go back to the dashboard by using the back button in the browser (1), disrupting their focus and workflow. Moreover, when a project chart is visible in the dashboard, the user can quickly open the measurement edit popup by clicking it on their respective badge or label below the chart, but this does not occur to other ontological entities that are rendered in this page, such as the annotation types and the data sources. To change an annotation type's color for example, the user would have to go to the annotation type management page (1 click), find and select the specific type they need to change (nondeterministic number of clicks, since the user can attempt to find it page by page or by searching for it, but the minimum is 1 click), press the Edit button (1), make the necessary changes and Save (minimum 2), and go back to the dashboard (1).

CHAPTER 6

Conclusion and future work

6.1 Conclusion

The solution presented in this dissertation contains multiple architecture and design decisions built around the same focal points:

- versatility of usage for both frontend and backend, adaptable to any data irrespective of the domain;
- improved analysis tools that leverage both the shared-space model and the split-space model, in order to reduce the number of trade-offs that come from picking only one of the two;
- strong graphic representation for annotations where analysts can attach meaning to isolated segments of data where many unaffected metrics can co-exist;
- safe and reliable collaboration where all changes are highly *consistent*, validated, and properly documented;
- scalability for both massive data sets and heavy traffic.

Despite the prototype being focused on HVAC data, the mapping of data is domain-agnostic. The prototype can still be used with any other data set, regardless if the metrics are logically distinguished in the same way the HVAC data set is. All aspects of the platform adapt to the data set without complications. The presented data model and visualization allows researchers to reduce the full scope of the observed data in many different ways: by using projects that enforce a common analysis and annotation task between collaborators; by using the automatic grouping and quick switching of data by their input source device; by limiting annotation semantics to a set of measurements where this semantic is compatible; or by narrowing annotations to a subset of the visible series in the same segment of time. Moreover, the frontend application allows multiple projects and windows of data to be selected and displayed in parallel, so that analysts can compare different groups of series and potentially find commonalities that can be annotated and notified to collaborators. All of these tools emphasize the simplification of analysis and knowledge building tasks by human collaborators, improving productivity and saving them time.

It is the belief of the author that the proposed annotation encoding using *snakes* is a major improvement in readability. Annotations can be attached to smaller areas of the vertical space, outlining specific series and behaviors such as spikes or curves, and keeping the remaining vertical space clear, so that other series in the same shared-space chart can co-exist without visual clutter. While other tools have provided similar functionality through the manual and free-form drawing of shapes [16], [112], [113], these only provide added accessibility when annotating in tablet devices while generating unstructured annotation images that cannot be easily parsed by automated systems. With the proposed encoding, annotations have similar accessibility, but follow a structured schema that enables additional parsing and processing modules to properly interpret them.

Furthermore, the proposed model rises above existing time series analysis tools by leveraging distributed systems techniques and standard practices in the software industry to optimize the querying and storage of massive amounts of time series and annotations without compromising *consistency*, flexibility, or safety. The current prototype is quite capable of ensuring a high level of *linearizability* for high amounts of simultaneous visualization and annotation requests, providing *eventual-consistency* only for time series queries that are filtered by associations with annotations, annotation types or projects.

6.2 Future Work

User permission granularity

The proposed solution was designed primarily for application in a corporate environment where all authenticated peers are part of the organization and benefit from an open platform where every aspect of the ontology is customizable. The only read-write access policy implemented in the solution, thanks to the processing pipeline validation stage, is one where users can only perform actions over projects (and respective annotations) that they are a part of. Only a few other constraints to user actions are applied, such as authors of a project not being able to be removed as collaborators of it by anyone. Since authors are considered the single owner of the project, projects are only removable by the author. Otherwise, all collaborators in a project are free to change any aspect of the project, which include adding or removing other collaborators that are not the author, removing annotations from peers, and even changing the series query of the project. Any user in the application can also change any annotation type, measurement or source metadata in a “free-for-all” manner.

This also means that if an attacker is authenticated into the platform, they can perform malicious and unsanctioned changes over every aspect of the ontology that they are given access to. These unsanctioned changes can be recovered through the versioning feature, as any collaborator can revert any changes made over time for as long as the current annotations still adhere to their older constraints. However, this is not an optimal solution, as it can be a cumbersome process to rollback multiple changes that were made by a malicious user. Rather, the appropriate solution would be to apply more restriction to users based on their roles in the organization. In regards to project access, a user could be granted the roles of *normal*, *manager* and *owner*, which correspondingly would grant them more possible actions. For example, all roles would be able to create and edit annotations, but only owners and managers of a project would be allowed to delete annotations or change the project details. As for the ontology management pages, these could be visible only to an administrator of the system.

Despite the lack of privacy settings and user roles, this tool can still be deployed in a private network where all authenticated users are guaranteed to be a part of the same organization, as it was originally intended by the work proposal. By importing user identities and authentication protocols from the organization's LDAP, the platform is only open to users that exist in that directory. If a collaborator leaves the organization and the LDAP directory is updated to disable the respective user profile, this change will be propagated to the proposed platform, as the user will no longer be able to login.

Database sharding

In distributed systems that handle high amounts of requests, even if the backend server is replicated to reduce computational strain of each server, a single database node constitutes a point of failure that, if affected by downtime, will cause the entire system to be down. The developed solution replicates the components available in the processing layer, namely the processing pipelines, but does not propose any replication of the persistence layer. Both the *availability* of time series data and ontological data can be affected if their respective databases are unavailable, albeit separately. However, database sharding is a complicated process that is hard to get right in implementation, and many of the databases that provide it required many years of development time to solve it without data integrity issues. Some RDBMSs such as PostgreSQL, and most CDBMSs such as Cassandra and HBase, contain the necessary partitioning and replication techniques to scale and protect against system crashes or errors, as well as the orchestration necessary to detect faulty database shards and apply the necessary steps to preserve the data and maintain *availability*.

These conscious design choices come at a cost by prioritizing *availability* of data over *consistent* writes, since any writing request has to be propagated between shards. Furthermore, InfluxDB does not implement any replication or sharding in its open-source variant, requiring these to be implemented manually, which is not ideal due to the complexity of the task. Due to both of these reasons, the databases were deployed as a single node, and the system relies solely on the replication of the backend server to increase the *availability* of the data.

Light-weight transmission of massive query results

One aspect that the platform lacks when handling massive quantities of time series or annotations is when serializing and transmitting these massive quantities to a user frontend. Based on a few tests where all units of the architecture are deployed in a machine with a quad-core processor and 16GB of RAM memory, mitigating network delays, when attempting to send a massive set of time series points in the order of tens of millions, the platform was able to query InfluxDB and deliver the serialized response with over one mega-byte of series in under an average of 768.6 milliseconds. When attempting to render these series in the frontend chart, an average of 3057 milliseconds were required, leading up to a total of 3825.6

milliseconds. In other words, these local tests, which are based on machine that is not under strain from heavy traffic, showed an almost 4 second delay between triggering a query and displaying a high amount of series on the chart. The amount of time required for each user to visualize their series in parallel will only increase exponentially, providing a poor user experience. In an attempt to solve this, multiple techniques to lower the size of the payload could be implemented. Direct payload compression would reduce the full size of the HTTP packet that has to be sent to the user, but it would also increase the amount of time required to display each chart, as an additional workload would be required for the server to compress the query results and for the frontend to decompress them.

An appealing approach would be to leverage a WebSocket in order to transmit the entire data set partitioned into smaller chunks. When a user queries a server, a WebSocket channel is opened specifically for the results of that query. Then, the channel details can be sent to the user directly in the HTTP response, which the user's frontend application uses to establish a real-time streamed communication with the server. Then, the queried results could be manually partitioned and sent in smaller chunks to this WebSocket channel, being intercepted by the frontend application and applied directly to the chart. In this way, the chart can progressively render more and more of the full queried results as more and more chunks are received. The user experience would be massively improved, since all series would immediately start to be rendered on the chart from the first moment the WebSocket connection is established.

One pitfall of the approach above is that the workload to partition the full query results would increase the amount of wait time before the server can start sending chunked data sets through the WebSocket channel. Luckily, InfluxDB already provides support for chunked responses directly from the database[1], which can be sent asynchronously to the server. Therefore, the server can simply establish a connection between the asynchronous chunks that are received from InfluxDB and the WebSocket channel directed at the target user, feeding the resulting chunks directly. The server could attempt to parse and serialize each chunk in order to return it in the required matrix format for Dygraphs, which would require an additional translation task. However, an important design decision here is how granular the chunks should be. If these are too coarse, the translation and serialization tasks for each of the chunks would compose a high impact on performance. If these are too fine, then each of the chunks does not have enough information to be useful to the querying user.

The above approach can be applied similarly to annotations located in the PostgreSQL, leveraging the streamed queries functionality that the PostgreSQL extension PipelineDB[2] already implements.

[1]`https://docs.influxdata.com/influxdb/v1.6/guides/querying_data/#chunking`

[2]`https://www.pipelinedb.com/`

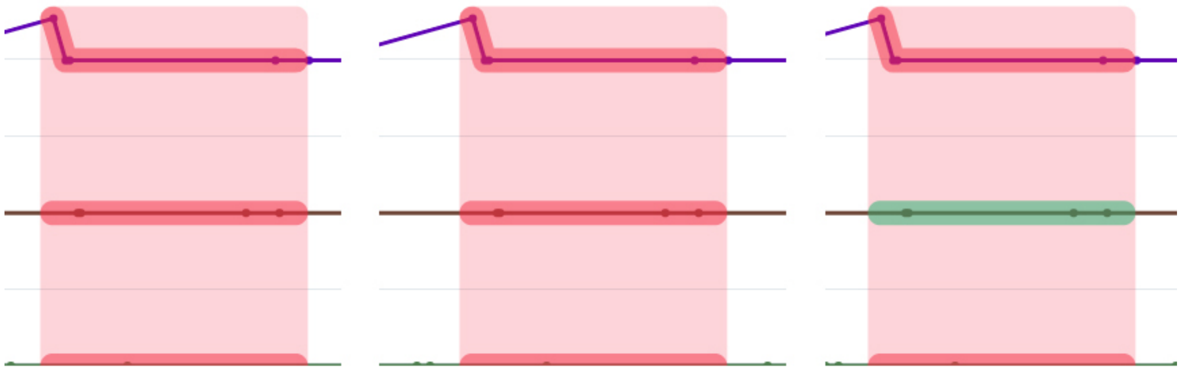


Figure 6.1: From left to right: i) one annotation is covering the three series; ii) one region annotation is covering the purple (top) and green (bottom) series, and another annotation with the exact same region and type is covering the brown (middle) series; iii) same scenario as in ii), but the type of the second annotation is changed in order for this scenario to be more evident.

Better visual encoding for snake connecting overlays

In the current implementation of the annotation encoding, two point or region annotations can be set so that both occupy the same segment of time and contain the same parent annotation type, where one is affecting two series in the top and bottom extremes of the chart while another is affecting a series in the middle of the chart. This will cause their visual representation to look exactly like it would if there was only one annotation covering the three series, as evidenced in Figure 6.1.

In a realistic scenario, an analyst would normally not benefit from making these two annotations separately and instead merge them into a single annotation. Additionally, when mouse-hovering the outer annotation, the highlight will correctly paint the *snakes* and overlay, and not the *snakes* that correspond to other annotations.

6.2.1 Research directions

The proposed model as-is establishes a open, modular architecture that enables complex task workflows to be set up while using the platform's knowledge base and annotation capabilities. Additional modules could be introduced in order to extract the existing annotations and deliver these into automated lexical and sentiment analysis tools such as information extraction systems, machine-learning models and deep neural networks. At its most basic level, the annotations and their relation with the affected series can be used to build a separate ontology or knowledge base. A machine learning model can be trained with annotation data and the corresponding series curves in order to recognize those patterns in other locations of the data set. Over time, this supervised learning process would allow a machine learning model to automatically feed new annotations based on the existing ones, allowing human collaborators to identify novel areas of the data set that are potentially useful. The data ingestion itself can already be done through the existing CRUD operations available in the REST API.

As with these output modules, real-time data input modules could be coupled with the platform, using data streaming technologies such as Esper[3], WSO2 CEP[4], Apache Spark Streaming[5], Apache Flink[6] or Apache Ignite Streaming[7]. This introduces the ability to stream real time critical data that requires a timely analysis, for example, in the domains of neurophysiology monitoring. While the current prototype fetches and indexes the full time series data on first deployment, nothing about its architecture prevents further time series from being ingested over time. These input modules can be designed to append newer sets of time series for either existing data sources and measurements, or even creating new ones. Additionally, input modules can be created in order to parse and feed a repository of annotation types that the analysts already own and are accustomed to, or to feed a repository of annotations that analysts already created in other systems.

6.3 Final thoughts

Overall, there is still much more that can be done to enable a fully-fledged universal platform for collaborative analysis of data sets. The proposed solution introduces a few additions and features over the existing state of the art for time series analysis software, and leaves many more guidelines and directions to be able to extend the model into a more mature software tool with increased usefulness in other domains, such as finance technical analysis, EEGs and ECGs reporting and medical diagnosis, or IoT services that support massive smart cities. The know-how presented in this dissertation serves as a foundation to build stronger collaborative frameworks and to ease the process of knowledge discovery. The world of analysis and collaborative software still has many advancements to go through, and with the development of techniques that allows us to more quickly and accurately derive meaning from raw data, ultimately we can discover new things that will further expand the already extensive monad that is human knowledge.

[3]`http://www.espertech.com/products/esper.php`

[4]`http://wso2.com/products/complex-event-processor/`

[5]`http://spark.apache.org/streaming/`

[6]`https://flink.apache.org`

[7]`https://ignite.apache.org/features/streaming.html`

References

[1] S. K. Jensen, T. B. Pedersen, and C. Thomsen, “Time series management systems: A survey”, *IEEE Transactions on Knowledge and Data Engineering*, vol. 29, no. 11, pp. 2581–2600, 2017, ISSN: 1041-4347. DOI: 10.1109/TKDE.2017.2740932.

[2] A. Bader, O. Kopp, and M. Falkenthal, “Survey and comparison of open source time series databases”, in *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband*, B. Mitschang, D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp, and M. Wieland, Eds., Bonn: Gesellschaft für Informatik e.V., 2017, pp. 249–268.

[3] D. Laney, *3d data management: Controlling data volume, variety and velocity*, 2001.

[4] M. Blount, M. Ebling, J Eklund, A. James, C. Mcgregor, N. Percival, K. Smith, and D. Sow, “Real-time analysis for intensive care: Development and deployment of the artemis analytic system”, *IEEE Engineering in Medicine and Biology Magazine*, vol. 29, no. 2, pp. 110–118, 2010, ISSN: 0739-5175. DOI: 10.1109/MEMB.2010.936454.

[5] R. D. O’Reilly, “A distributed architecture for the monitoring and analysis of time series data”, 2015.

[6] T. chung Fu, “A review on time series data mining”, *Engineering Applications of Artificial Intelligence*, vol. 24, no. 1, pp. 164 –181, 2011, ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2010.09.007. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197610001727.

[7] J. Lin, E. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom, “Visually mining and monitoring massive time series”, in *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’04, Seattle, WA, USA: ACM, 2004, pp. 460–469, ISBN: 1-58113-888-1. DOI: 10.1145/1014052.1014104. [Online]. Available: http://doi.acm.org/10.1145/1014052.1014104.

[8] D. Sow, A. Biem, M. Blount, M. Ebling, and O. Verscheure, “Body sensor data processing using stream computing”, in *Proceedings of the International Conference on Multimedia Information Retrieval*, ser. MIR ’10, Philadelphia, Pennsylvania, USA: ACM, 2010, pp. 449–458, ISBN: 978-1-60558-815-5. DOI: 10.1145/1743384.1743465. [Online]. Available: http://doi.acm.org/10.1145/1743384.1743465.

[9] H. Han, H. C. Ryoo, and H. Patrick, “An infrastructure of stream data mining, fusion and management for monitored patients”, in *19th IEEE Symposium on Computer-Based Medical Systems (CBMS’06)*, 2006, pp. 461–468. DOI: 10.1109/CBMS.2006.39.

[10] V. Nenov and J. Klopp, “Remote analysis of physiological data from neurosurgical icu patients”, *Journal of the American Medical Informatics Association*, vol. 3, no. 5, pp. 318–327, 1996. DOI: 10.1136/jamia.1996.97035023. eprint: /oup/backfile/content_public/journal/jamia/3/5/10.1136/jamia.1996.97035023/2/3-5-318.pdf. [Online]. Available: +http://dx.doi.org/10.1136/jamia.1996.97035023.

[11] C. Mcgregor, D. Sow, A. James, B., M. Ebling, E., J., and K. Smith, “Collaborative research on an intensive care decision support system utilizing physiological data streams”, Jan. 2009.

[12] P. D. Healy, R. D. O’Reilly, G. B. Boylan, and J. P. Morrison, “Interactive annotations to support collaborative analysis of streaming physiological data”, in *2011 24th International Symposium on Computer-Based Medical Systems (CBMS)*, 2011, pp. 1–5. DOI: 10.1109/CBMS.2011.5999131.

[13] A. Bar-Or, J. Healey, L. Kontothanassis, and J. M. V. Thong, “Biostream: A system architecture for real-time processing of physiological signals”, in *The 26th Annual International Conference of the*

IEEE Engineering in Medicine and Biology Society, vol. 2, 2004, pp. 3101–3104. DOI: 10.1109/IEMBS.2004.1403876.

[14] E. Hadavandi, H. Shavandi, and A. Ghanbari, "Integration of genetic fuzzy systems and artificial neural networks for stock price forecasting", *Knowledge-Based Systems*, vol. 23, no. 8, pp. 800 –808, 2010, ISSN: 0950-7051. DOI: https://doi.org/10.1016/j.knosys.2010.05.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950705110000857.

[15] S. K. Badam, J. Zhao, S. Sen, N. Elmqvist, and D. Ebert, "Timefork: Interactive prediction of time series", in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16, San Jose, California, USA: ACM, 2016, pp. 5409–5420, ISBN: 978-1-4503-3362-7. DOI: 10.1145/2858036.2858150. [Online]. Available: http://doi.acm.org/10.1145/2858036.2858150.

[16] S. Kamburugamuve, P. Wickramasinghe, S. Ekanayake, C. Wimalasena, M. Pathirage, and G. C. Fox, "Tsmap3d: Browser visualization of high dimensional time series data", *2016 IEEE International Conference on Big Data (Big Data)*, pp. 3583–3592, 2016.

[17] A. Chourasia, K. B. Richards-Dinger, J. H. Dieterich, and Y. Cui, "Visual exploration and analysis of time series earthquake data", in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, ser. PEARC17, New Orleans, LA, USA: ACM, 2017, 48:1–48:6, ISBN: 978-1-4503-5272-7. DOI: 10.1145/3093338.3093366. [Online]. Available: http://doi.acm.org/10.1145/3093338.3093366.

[18] J. Lee, D. Rowlands, N. Jackson, R. Leadbetter, T. Wada, and D. James, "An architectural based framework for the distributed collection, analysis and query from inhomogeneous time series data sets and wearables for biofeedback applications", *Algorithms*, vol. 10, no. 4, p. 23, 2017, ISSN: 1999-4893. DOI: 10.3390/a10010023. [Online]. Available: http://dx.doi.org/10.3390/a10010023.

[19] J. Lin, E. Keogh, S. Lonardi, J. P. Lankford, and D. M. Nystrom, "Viztree: A tool for visually mining and monitoring massive time series databases", in *In Proceedings of International Conference on Very Large Data Bases*, 2004, pp. 1269–1272.

[20] F. D. Turck, J. Decruyenaere, P. Thysebaert, S. V. Hoecke, B. Volckaert, C. Danneels, K. Colpaert, and G. D. Moor, "Design of a flexible platform for execution of medical decision support agents in the intensive care unit", *Computers in Biology and Medicine*, vol. 37, no. 1, pp. 97 –112, 2007, ISSN: 0010-4825. DOI: https://doi.org/10.1016/j.compbiomed.2005.10.004. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010482505001216.

[21] H. González-Vélez, M. Mier, M. Julià-Sapé, T. N. Arvanitis, J. M. García-Gómez, M. Robles, P. H. Lewis, S. Dasmahapatra, D. Dupplaw, A. Peet, C. Arús, B. Celda, S. Van Huffel, and M. Lluch-Ariet, "Healthagents: Distributed multi-agent brain tumor diagnosis and prognosis", *Applied Intelligence*, vol. 30, no. 3, pp. 191–202, 2009, ISSN: 1573-7497. DOI: 10.1007/s10489-007-0085-8. [Online]. Available: https://doi.org/10.1007/s10489-007-0085-8.

[22] D. A. Keim, F. Mansmann, J. Schneidewind, and H. Ziegler, "Challenges in visual data analysis", in *Tenth International Conference on Information Visualisation (IV'06)*, 2006, pp. 9–16. DOI: 10.1109/IV.2006.31.

[23] W. Playfair, *The commercial and political atlas: Representing by means of stained copper-plate charts the progress of the commerce revenues expenditure and debts of england during the whole of the eighteenth century*, 1786.

[24] H. Hochheiser and B. Shneiderman, "Dynamic query tools for time series data sets: Timebox widgets for interactive exploration", *Information Visualization*, vol. 3, no. 1, pp. 1–18, 2004, ISSN: 1473-8716. DOI: 10.1145/993176.993177. [Online]. Available: http://dx.doi.org/10.1145/993176.993177.

[25] G. Ellis and A. Dix, "A taxonomy of clutter reduction for information visualisation", *IEEE Transactions on Visualization and Computer Graphics*, vol. 13, no. 6, pp. 1216–1223, 2007, ISSN: 1077-2626. DOI: 10.1109/TVCG.2007.70535.

[26] W. Javed, B. McDonnel, and N. Elmqvist, "Graphical perception of multiple time series", *IEEE Transactions on Visualization and Computer Graphics*, vol. 16, no. 6, pp. 927–934, 2010, ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.162.

[27] Y. Chen, P. Xu, and L. Ren, "Sequence synopsis: Optimize visual summary of temporal event data", *IEEE Transactions on Visualization and Computer Graphics*, vol. 24, no. 1, pp. 45–55, 2018, ISSN: 1077-2626. DOI: 10.1109/TVCG.2017.2745083.

[28] J. Bernard, T. Ruppert, M. Scherer, T. Schreck, and J. Kohlhammer, "Guided discovery of interesting relationships between time series clusters and metadata properties", in *Proceedings of the 12th International Conference on Knowledge Management and Knowledge Technologies*, ser. i-KNOW '12, Graz, Austria: ACM, 2012, 22:1–22:8, ISBN: 978-1-4503-1242-4. DOI: 10.1145/2362456.2362485. [Online]. Available: http://doi.acm.org/10.1145/2362456.2362485.

[29] Y. Keraron, A. Bernard, and B. Bachimont, "Annotations to improve the using and the updating of digital technical publications", vol. 20, pp. 157–170, Sep. 2009.

[30] C. Marshall, "Annotation: From paper books to the digital library", in *Proceedings of the Second ACM International Conference on Digital Libraries*, ser. DL '97, Philadelphia, Pennsylvania, USA: ACM, 1997, pp. 131–140, ISBN: 0-89791-868-1. DOI: 10.1145/263690.263806. [Online]. Available: http://doi.acm.org/10.1145/263690.263806.

[31] ——, "Toward an ecology of hypertext annotation", in *Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia: Links, Objects, Time and Space—structure in Hypermedia Systems: Links, Objects, Time and Space—structure in Hypermedia Systems*, ser. HYPERTEXT '98, Pittsburgh, Pennsylvania, USA: ACM, 1998, pp. 40–49, ISBN: 0-89791-972-6. DOI: 10.1145/276627.276632. [Online]. Available: http://doi.acm.org/10.1145/276627.276632.

[32] I. A. OVSIANNIKOV, M. A. ARBIB, and T. H. MCNEILL, "Annotation technology", *International Journal of Human-Computer Studies*, vol. 50, no. 4, pp. 329 –362, 1999, ISSN: 1071-5819. DOI: https://doi.org/10.1006/ijhc.1999.0247. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1071581999902471.

[33] M. Zacklad, "Documentarisation processes in documents for action (dofa): The status of annotations and associated cooperation technologies", *Computer Supported Cooperative Work (CSCW)*, vol. 15, no. 2, pp. 205–228, 2006, ISSN: 1573-7551. DOI: 10.1007/s10606-006-9019-y. [Online]. Available: https://doi.org/10.1007/s10606-006-9019-y.

[34] S. Bringay, C. Barry, and J. Charlet, "Annotations for the collaboration of the health professionals", in *AMIA Annual Symposium Proceedings*, American Medical Informatics Association, vol. 2006, 2006, p. 91.

[35] Y. Keraron, A. Bernard, and B. Bachimont, "Annotations to improve the using and the updating of digital technical publications", vol. 20, pp. 157–170, Sep. 2009.

[36] N. Bricon-Souf, S. Bringay, S. Hamek, F. Anceaux, C. Barry, and J. Charlet, "Informal notes to support the asynchronous collaborative activities", *International Journal of Medical Informatics*, vol. 76, S342 –S348, 2007, Ubiquity: Technologies for Better Health in Aging Societies - MIE 2006, ISSN: 1386-5056. DOI: https://doi.org/10.1016/j.ijmedinf.2007.02.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1386505607000469.

[37] R. Kawase, E. Herder, and W. Nejdl, "A comparison of paper-based and online annotations in the workplace", in *Learning in the Synergy of Multiple Disciplines*, U. Cress, V. Dimitrova, and M. Specht, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 240–253, ISBN: 978-3-642-04636-0.

[38] T. Guyet, C. Garbay, and M. Dojat, "Knowledge construction from time series data using a collaborative exploration system", *Journal of Biomedical Informatics*, vol. 40, no. 6, pp. 672 –687, 2007, Intelligent Data Analysis in Biomedicine, ISSN: 1532-0464. DOI: https://doi.org/10.1016/j.jbi.2007.09.006. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1532046407001050.

[39] D. A. Kalogeropoulos, E. R. Carson, and P. O. Collinson, "Towards knowledge-based systems in clinical practice: Development of an integrated clinical information and knowledge management support system", *Computer Methods and Programs in Biomedicine*, vol. 72, no. 1, pp. 65 –80, 2003, ISSN: 0169-2607. DOI: https://doi.org/10.1016/S0169-2607(02)00118-9. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0169260702001189.

[40] W. B. S. Pressly Jr., "Tspad: A tablet-pc based application for annotation and collaboration on time series data", in *Proceedings of the 46th Annual Southeast Regional Conference on XX*, ser. ACM-SE 46,

Auburn, Alabama: ACM, 2008, pp. 527–528, ISBN: 978-1-60558-105-7. DOI: 10.1145/1593105.1593249. [Online]. Available: http://doi.acm.org/10.1145/1593105.1593249.

[41] J. Y. Halpern and Y. Moses, “Knowledge and common knowledge in a distributed environment”, *J. ACM*, vol. 37, no. 3, pp. 549–587, Jul. 1990, ISSN: 0004-5411. DOI: 10.1145/79147.79161. [Online]. Available: http://doi.acm.org/10.1145/79147.79161.

[42] J. Waldo, J. Waldo, G. Wyant, G. Wyant, A. Wollrath, A. Wollrath, S. Kendall, and S. Kendall, “A note on distributed computing”, IEEE Micro, Tech. Rep., 1994.

[43] R. van Renesse and F. B. Schneider, “Chain replication for supporting high throughput and availability”, in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI’04, San Francisco, CA: USENIX Association, 2004, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251261.

[44] F. T. Leighton and D. M. Lewin, *Content delivery network using edge-of-network servers for providing content delivery to a set of participating content providers*, US Patent 6,553,413, 2003.

[45] S. Gilbert and N. Lynch, “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services”, *SIGACT News*, vol. 33, no. 2, pp. 51–59, Jun. 2002, ISSN: 0163-5700. DOI: 10.1145/564585.564601. [Online]. Available: http://doi.acm.org/10.1145/564585.564601.

[46] D. Abadi, “Consistency tradeoffs in modern distributed database system design: Cap is only part of the story”, *Computer*, vol. 45, no. 2, pp. 37–42, 2012.

[47] I. Shafer, R. R. Sambasivan, A. Rowe, and G. R. Ganger, “Specialized storage for big numeric time series”, in *Presented as part of the 5th USENIX Workshop on Hot Topics in Storage and File Systems*, San Jose, CA: USENIX, 2013. [Online]. Available: https://www.usenix.org/conference/hotstorage13/workshop-program/presentation/Shafer.

[48] C. Pungilă, T.-F. Fortiş, and O. Aritoni, “Benchmarking database systems for the requirements of sensor readings”, *IETE Technical Review*, vol. 26, no. 5, pp. 342–349, 2009. DOI: 10.4103/0256-4602.55279. eprint: http://www.tandfonline.com/doi/pdf/10.4103/0256-4602.55279. [Online]. Available: http://www.tandfonline.com/doi/abs/10.4103/0256-4602.55279.

[49] T. W. Wlodarczyk, “Overview of time series storage and processing in a cloud environment”, in *4th IEEE International Conference on Cloud Computing Technology and Science Proceedings*, 2012, pp. 625–628. DOI: 10.1109/CloudCom.2012.6427510.

[50] T. Goldschmidt, A. Jansen, H. Koziolek, J. Doppelhamer, and H. P. Breivold, “Scalability and robustness of time-series databases for cloud-native monitoring of industrial processes”, in *2014 IEEE 7th International Conference on Cloud Computing*, 2014, pp. 602–609. DOI: 10.1109/CLOUD.2014.86.

[51] A. K. Kalakanti, V. Sudhakaran, V. Raveendran, and N. Menon, “A comprehensive evaluation of nosql datastores in the context of historians and sensor data analysis”, in *2015 IEEE International Conference on Big Data (Big Data)*, 2015, pp. 1797–1806. DOI: 10.1109/BigData.2015.7363952.

[52] D. W. Curtis, E. J. Pino, J. M. Bailey, E. I. Shih, J. Waterman, S. A. Vinterbo, T. O. Stair, J. V. Guttag, R. A. Greenes, and L. Ohno-Machado, “Smart—an integrated wireless system for monitoring unattended patients”, *Journal of the American Medical Informatics Association*, vol. 15, no. 1, pp. 44–53, 2008. DOI: 10.1197/jamia.M2016. eprint: /oup/backfile/content_public/journal/jamia/15/1/10.1197/jamia.m2016/2/15-1-44.pdf. [Online]. Available: +http://dx.doi.org/10.1197/jamia.M2016.

[53] P. D. Healy, R. D. O’Reilly, G. B. Boylan, and J. P. Morrison, “Web-based remote monitoring of live eeg”, in *The 12th IEEE International Conference on e-Health Networking, Applications and Services*, 2010, pp. 169–174. DOI: 10.1109/HEALTH.2010.5556574.

[54] A.-E. Rizzoli, G. Schimak, M. Donatelli, and J. Hřebíček, “Tatoo: Tagging environmental resources on the web by semantic annotations”, 2010.

[55] T. Pariente, J. M. Fuentes, M. A. Sanguino, S. Yurtsever, G. Avellino, A. E. Rizzoli, and S. Nešić, “A model for semantic annotation of environmental resources: The tatoo semantic framework”, in *International Symposium on Environmental Software Systems*, Springer, 2011, pp. 419–427.

[56] L. O. Batista and C. B. Medeiros, “Supporting the study of correlations between time series via semantic annotations”, 2014.

[57] ——, “Searching time series via semantic annotations”, 2013.

[58] J. Park, D. Nguyen, and R. Sandhu, “A provenance-based access control model”, in *2012 Tenth Annual International Conference on Privacy, Security and Trust*, 2012, pp. 137–144. DOI: `10.1109/PST.2012.6297930`.

[59] T. Pifferi. (2018). How to efficiently store and query time-series data, [Online]. Available: `https://medium.com/\@neslinesli93/efficiently-store-time-series-data-90313ff0ec20`.

[60] M. Freedman. (2018). Timescaledb vs. influxdb: Purpose built differently for time-series data, [Online]. Available: `https://blog.timescale.com/timescaledb-vs-influxdb-36489299877`.

[61] L. Hampton. (2018). Eye or the tiger: Benchmarking cassandra vs. timescaledb for time-series data, [Online]. Available: `https://blog.timescale.com/timescaledb-vs-cassandra-7c2cc50a89ce`.

[62] R. Kiefer. (2017). Timescaledb vs. postgres for time-series: 20x higher inserts, 2000x faster deletes, 1.2x-14,000x faster queries, [Online]. Available: `https://blog.timescale.com/timescaledb-vs-postgresql-6a696248104e`.

[63] E. Nordström. (2017). Problems with postgresql 10 for time-series data, [Online]. Available: `https://blog.timescale.com/time-series-data-postgresql-10-vs-timescaledb-816ee808bac5`.

[64] Z Mathe, C Haen, and F Stagni, “Monitoring performance of a highly distributed and complex computing infrastructure in lhcb”, in *Journal of Physics: Conference Series*, IOP Publishing, vol. 898, 2017, p. 092 028.

[65] P. Seshadri, M. Livny, and R. Ramakrishnan, “The design and implementation of a sequence database system”, in *Proceedings of the 22th International Conference on Very Large Data Bases*, ser. VLDB ’96, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1996, pp. 99–110, ISBN: 1-55860-382-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=645922.673634`.

[66] A. Lerner and D. Shasha, “Aquery: Query language for ordered data, optimization techniques, and experiments”, in *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, ser. VLDB ’03, Berlin, Germany: VLDB Endowment, 2003, pp. 345–356, ISBN: 0-12-722442-4. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1315451.1315482`.

[67] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (lsm-tree)”, *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[68] B. Momjian. (2018). Mvcc unmasked, [Online]. Available: `https://momjian.us/main/writings/pgsql/mvcc.pdf`.

[69] P. Keil, “Principal agent theory and its application to analyze outsourcing of software development”, in *Proceedings of the Seventh International Workshop on Economics-driven Software Engineering Research*, ser. EDSER ’05, St. Louis, Missouri: ACM, 2005, pp. 1–5, ISBN: 1-59593-118-X. DOI: `10.1145/1082983.1083094`. [Online]. Available: `http://doi.acm.org/10.1145/1082983.1083094`.

[70] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya, “An annotation management system for relational databases”, in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, ser. VLDB ’04, Toronto, Canada: VLDB Endowment, 2004, pp. 900–911, ISBN: 0-12-088469-0. [Online]. Available: `http://dl.acm.org/citation.cfm?id=1316689.1316767`.

[71] M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva, “Supporting annotations on relations”, in *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, ser. EDBT ’09, Saint Petersburg, Russia: ACM, 2009, pp. 379–390, ISBN: 978-1-60558-422-5. DOI: `10.1145/1516360.1516405`. [Online]. Available: `http://doi.acm.org/10.1145/1516360.1516405`.

[72] F. H. da Silva, “Serial annotator: Managing annotations of time series.”, 2014.

[73] A. Bhardwaj, S. Bhattacherjee, A. Chavan, A. Deshpande, A. J. Elmore, S. Madden, and A. G. Parameswaran, “Datahub: Collaborative data science & dataset version management at scale”, *arXiv preprint arXiv:1409.0798*, 2014.

[74] V. Mihalcea. (2017). How does mvcc (multi-version concurrency control) work, [Online]. Available: `https://vladmihalcea.com/how-does-mvcc-multi-version-concurrency-control-work/`.

[75] G. Ozsoyoglu and R. T. Snodgrass, "Temporal and real-time databases: A survey", *IEEE Transactions on Knowledge and Data Engineering*, vol. 7, no. 4, pp. 513–532, 1995, ISSN: 1041-4347. DOI: `10.1109/69.404027`.

[76] E. Sciore, "Using annotations to support multiple kinds of versioning in an object-oriented database system", *ACM Trans. Database Syst.*, vol. 16, no. 3, pp. 417–438, Sep. 1991, ISSN: 0362-5915. DOI: `10.1145/111197.111205`. [Online]. Available: `http://doi.acm.org/10.1145/111197.111205`.

[77] R. T. Snodgrass, "Temporal databases", in *Theories and methods of spatio-temporal reasoning in geographic space*, Springer, 1992, pp. 22–64.

[78] S. Bhattacherjee, A. Chavan, S. Huang, A. Deshpande, and A. G. Parameswaran, "Principles of dataset versioning: Exploring the recreation/storage tradeoff", *CoRR*, vol. abs/1505.05211, 2015. arXiv: `1505.05211`. [Online]. Available: `http://arxiv.org/abs/1505.05211`.

[79] H. Fujita, K. Iskra, P. Balaji, and A. A. Chien, "Versioning architectures for local and global memory", in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, 2015, pp. 515–524. DOI: `10.1109/ICPADS.2015.71`.

[80] M. Fowler, "Event sourcing", *Online, Dec*, p. 18, 2005.

[81] L. Halilaj, I. Grangel-González, G. Coskun, and S. Auer, "Git4voc: Git-based versioning for collaborative vocabulary development", in *2016 IEEE Tenth International Conference on Semantic Computing (ICSC)*, 2016, pp. 285–292. DOI: `10.1109/ICSC.2016.44`.

[82] P. Lundgren. (2013). On git's shortcomings, [Online]. Available: `http://www.peterlundgren.com/blog/on-gits-shortcomings/`.

[83] S. Huang, L. Xu, J. Liu, A. J. Elmore, and A. G. Parameswaran, "Orpheusdb: Bolt-on versioning for relational databases", *PVLDB*, vol. 10, no. 10, pp. 1130–1141, 2017. [Online]. Available: `http://www.vldb.org/pvldb/vol10/p1130-huang.pdf`.

[84] R. Diana. (2011). Is object serialization evil?, [Online]. Available: `http://regulargeek.com/2011/07/06/is-object-serialization-evil/`.

[85] D. Crockford, "The application/json media type for javascript object notation (json)", 2006.

[86] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, "Extensible markup language (xml).", *World Wide Web Journal*, vol. 2, no. 4, pp. 27–66, 1997.

[87] K. Varda, "Protocol buffers: Google's data interchange format", *Google Open Source Blog, Available at least as early as Jul*, vol. 72, 2008.

[88] A. Sumaray and S. K. Makki, "A comparison of data serialization formats for optimal efficiency on a mobile platform", in *Proceedings of the 6th international conference on ubiquitous information management and communication*, ACM, 2012, p. 48.

[89] R. Fielding, "Representational state transfer", *Architectural Styles and the Design of Netowork-based Software Architecture*, pp. 76–85, 2000.

[90] M. Adnan, M. Just, and L. Baillie, "Investigating time series visualisations to improve the user experience", in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16, San Jose, California, USA: ACM, 2016, pp. 5444–5455, ISBN: 978-1-4503-3362-7. DOI: `10.1145/2858036.2858300`. [Online]. Available: `http://doi.acm.org/10.1145/2858036.2858300`.

[91] W. S. Cleveland and R. McGill, "Graphical perception: Theory, experimentation, and application to the development of graphical methods", *Journal of the American Statistical Association*, vol. 79, no. 387, pp. 531–554, 1984. DOI: `10.1080/01621459.1984.10478080`. eprint: `http://www.tandfonline.com/doi/pdf/10.1080/01621459.1984.10478080`. [Online]. Available: `http://www.tandfonline.com/doi/abs/10.1080/01621459.1984.10478080`.

[92] N. R. Tague, *The quality toolbox.* Asq Press, 2005.

[93] J. Fuchs, F. Fischer, F. Mansmann, E. Bertini, and P. Isenberg, "Evaluation of alternative glyph designs for time series data in a small multiple setting", in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13, Paris, France: ACM, 2013, pp. 3237–3246, ISBN: 978-1-4503-1899-0. DOI: 10.1145/2470654.2466443. [Online]. Available: http://doi.acm.org/10.1145/2470654.2466443.

[94] E. Tufte, *The visual display of quantitative information*, 1983.

[95] T. Saito, H. N. Miyamura, M. Yamamoto, H. Saito, Y. Hoshiya, and T. Kaseda, "Two-tone pseudo coloring: Compact visualization for one-dimensional data", in *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.*, 2005, pp. 173–180. DOI: 10.1109/INFVIS.2005.1532144.

[96] M. Wattenberg, "Arc diagrams: Visualizing structure in strings", in *IEEE Symposium on Information Visualization, 2002. INFOVIS 2002.*, 2002, pp. 110–116. DOI: 10.1109/INFVIS.2002.1173155.

[97] E. M. McCreight, "A space-economical suffix tree construction algorithm", *Journal of the ACM (JACM)*, vol. 23, no. 2, pp. 262–272, 1976.

[98] P. Bille, "A survey on tree edit distance and related problems", *Theoretical Computer Science*, vol. 337, no. 1, pp. 217 –239, 2005, ISSN: 0304-3975. DOI: https://doi.org/10.1016/j.tcs.2004.12.030. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304397505000174.

[99] J. J. V. Wijk and E. R. V. Selow, "Cluster and calendar based visualization of time series data", in *Information Visualization, 1999. (Info Vis '99) Proceedings. 1999 IEEE Symposium on*, 1999, pp. 4–9, 140. DOI: 10.1109/INFVIS.1999.801851.

[100] C. Daassi, M. Dumas, M.-C. Fauvet, L. Nigay, and P.-C. Scholl, *Visual exploration of temporal object databases*, 2000.

[101] J. V. Carlis and J. A. Konstan, "Interactive visualization of serial periodic data", in *Proceedings of the 11th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '98, San Francisco, California, USA: ACM, 1998, pp. 29–38, ISBN: 1-58113-034-1. DOI: 10.1145/288392.288399. [Online]. Available: http://doi.acm.org/10.1145/288392.288399.

[102] M. Weber, M. Alexa, and W. Müller, "Visualizing time-series on spirals", in *Proceedings of the IEEE Symposium on Information Visualization 2001 (INFOVIS'01)*, ser. INFOVIS '01, Washington, DC, USA: IEEE Computer Society, 2001, pp. 7–, ISBN: 0-7695-1342-5. [Online]. Available: http://dl.acm.org/citation.cfm?id=580582.857719.

[103] C. Tominski and H. Schumann, "Enhanced interactive spiral display", 2008.

[104] F. Bouali, S. Devaux, and G. Venturini, "Visual mining of time series using a tubular visualization", *The Visual Computer*, vol. 32, no. 1, pp. 15–30, 2016, ISSN: 1432-2315. DOI: 10.1007/s00371-014-1052-0. [Online]. Available: https://doi.org/10.1007/s00371-014-1052-0.

[105] K. Mitchell and J. Kennedy, "The perspective tunnel: An inside view on smoothly integrating detail and context", in *Visualization in scientific computing '97: proceedings of the Eurographics Workshop*, ser. Springer Computing Science, Springer, 1997.

[106] M. Suntinger, H. Obweger, J. Schiefer, and M. E. Gröller, "Event tunnel: Exploring event-driven business processes", *IEEE Comput. Graph. Appl.*, vol. 28, no. 5, pp. 46–55, 2008, ISSN: 0272-1716. DOI: 10.1109/MCG.2008.97. [Online]. Available: http://dx.doi.org/10.1109/MCG.2008.97.

[107] M. Ankerst, "Visual data mining with pixel-oriented visualization techniques", in *Proceedings of the ACM SIGKDD Workshop on Visual Data Mining*, 2001.

[108] P. Grunwald, "A tutorial introduction to the minimum description length principle", *arXiv preprint math/0406077*, 2004.

[109] F. Wanner, W. Jentner, T. Schreck, A. Stoffel, L. Sharalieva, and D. A. Keim, "Integrated visual analysis of patterns in time series and text data - workflow and application to financial data analysis", *Information Visualization*, vol. 15, no. 1, pp. 75–90, 2016. DOI: 10.1177/1473871615576925. eprint: https://doi.org/10.1177/1473871615576925. [Online]. Available: https://doi.org/10.1177/1473871615576925.

[110] E. Keogh, H. Hochheiser, and B. Shneiderman, "An augmented visual query mechanism for finding patterns in time series data", Springer-Verlag, 2002, pp. 240–250.

[111] H. Hochheiser and B. Shneiderman, "Interactive exploration of time series data", in *Discovery Science*, K. P. Jantke and A. Shinohara, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 441–446, ISBN: 978-3-540-45650-6.

[112] S.-H. Bae, J. Y. Choi, J. Qiu, and G. C. Fox, "Dimension reduction and visualization of large high-dimensional data via interpolation", in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10, Chicago, Illinois: ACM, 2010, pp. 203–214, ISBN: 978-1-60558-942-8. DOI: `10.1145/1851476.1851501`. [Online]. Available: `http://doi.acm.org/10.1145/1851476.1851501`.

[113] J. Y. Choi, S. H. Bae, X. Qiu, and G. Fox, "High performance dimension reduction and visualization for large high-dimensional data analysis", in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, 2010, pp. 331–340. DOI: `10.1109/CCGRID.2010.104`.

[114] J. Liang and M. L. Huang, "Highlighting in information visualization: A survey", in *2010 14th International Conference Information Visualisation*, 2010, pp. 79–85. DOI: `10.1109/IV.2010.21`.

[115] J P. Vermylen, "Visualizing energy data using web-based applications", Dec. 2008.

[116] D. Winokur. (2011). Flash to focus on pc browsing and mobile apps; adobe to more aggressively contribute to html5, [Online]. Available: `https://blogs.adobe.com/conversations/2011/11/flash-focus.html`.

[117] J. R. Harger and P. J. Crossno, "Comparison of open-source visual analytics toolkits", vol. 8294, 2012, pp. 8294 –8294 –10. DOI: `10.1117/12.911901`. [Online]. Available: `http://dx.doi.org/10.1117/12.911901`.

[118] U. NIST, *Descriptions of sha-256, sha-384 and sha-512*, 2001.

[119] N. Provos and D. Mazieres, "A future-adaptable password scheme.", 1999.

[120] D Eastlake and T Hansen, "Rfc 6234: Us secure hash algorithms (sha and sha-based hmac and hkdf)", *IETF Std*, 2011.

[121] J Jonsson and B Kaliski, "Public-key cryptography standards (pkcs)# 1: Rsa cryptography, specifications version 2.1., 2003", *RFC3447*, vol. 5, p. 14, 2004.

[122] J. Schaad, "Use of the rsassa-pss signature algorithm in cryptographic message syntax (cms)", Tech. Rep., 2005.

[123] P. Hoffman, "Elliptic curve digital signature algorithm (dsa) for dnssec", 2012.