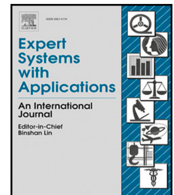
Expert Systems With Applications 226 (2023) 120149

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa



# Chrontext: Portable SPARQL queries over contextualised time series data in industrial settings☆

Magnus Bakken [a,b,*], Ahmet Soylu [a,c]

[a] Norwegian University of Science and Technology – NTNU, Teknologivegen 22, Gjøvik, 2815, Norway
[b] Prediktor AS, Habornveien 48b, Fredrikstad, 1630, Norway
[c] OsloMet – Oslo Metropolitan University, Pilestredet 35, Oslo, 0176, Norway

ARTICLE INFO

*Keywords:*
Time-series database
Static data
Query rewriting
Semantic Web
SPARQL

ABSTRACT

Industrial information models are standardised ways of representing industrial devices, equipment, and processes together with the data collected from associated sensors and control systems. Companies invest in such models to enable digitalisation and modular, reusable solutions. They also invest heavily in analytics (e.g. machine learning) based on time series data sets to improve operations. Queries that use such context to retrieve time series data can make industrial data sets more accessible to practitioners performing analytics and application development. Moreover, they can enable scalable deployment of resulting analytical models. Industrial availability constraints require that queries over context and time series should be portable in general, as they should be able retrieve data for training in a cloud setting and production data for deployment in an on-premise setting. Solving this problem is challenging with existing approaches as context and time series data tend to exist in separate, specialised databases. We address the issue by proposing a hybrid query engine, namely Chrontext, in the setting of a SPARQL database hosting the static model, and an arbitrary time series database. We show how with a set of annotations in the knowledge graph, SPARQL queries can be evaluated over such a hybrid architecture. We provide a proof showing that our approach correctly answers SPARQL 1.1. queries. We implement our approach in Rust under the Apache 2.0 license, and use the Apache Arrow-based Polars library together with configurable pushdowns to achieve high performance. We compare the performance of Chrontext against Ontop, one of the most prominent virtual knowledge graph systems, on a synthetic data set based on industrial standards. Data are stored in a S3 data lake and PostgreSQL with the Dremio data lakehouse as the SQL integrator. We find that our approach performs 10× to 85× faster and consumes much less memory than Ontop.

## 1. Introduction

Information models are ways of representing domains of interest in a standardised way (Lee, 1999), and industrial information models represent industrial devices, equipment, and processes. Uniformity and consistency in information modelling is important in order to make the most of industrial digitalisation efforts, as it enables greater scalability for applications that consume the information model, and lessens the burden on engineers. Firstly, in an industrial setting where there are a multitude of ways of representing an instrument, e.g., valve and data from valve instrumentation, it becomes much harder to write and integrate reusable software used to monitor the instrument. For this reason, companies, equipment vendors, and industry consortia have been investing in creating standards for information models (e.g. Bartusiak et al., 2022; Großmann & Diedrich, 2022; International Electrotechnical Commission, 2013, 2017, 2020; International Organization for Standardization, 2009; MTConnect Institute, 2022; Tantik & Anderl, 2017). Secondly, companies are facing pressure to invest in analytics to monitor, understand, and optimise the performance of their processes and assets and decision making (Dehghani, 2022). Modern analytics is highly dependent on large data sets. In industry, these data sets are often collections of values at particular points in time (time series data) produced by regular samples from sensors and events generated by actuators. These data sets are stored in special purpose databases called time series databases (Bader, Kopp, & Falkenthal, 2017), that are optimised for the patterns found in generation of- and access to such data. Data will typically be appended to a time series database and be of

☆ This work is jointly funded by The Research Council of Norway and Prediktor AS, Norway, where the first author is employed, under the industrial Ph.D. scheme (316656).

* Corresponding author at: Norwegian University of Science and Technology – NTNU, Teknologivegen 22, Gjøvik, 2815, Norway.
*E-mail addresses:* magba@stud.ntnu.no (M. Bakken), ahmet.soylu@oslomet.no (A. Soylu).

https://doi.org/10.1016/j.eswa.2023.120149
Received 20 December 2022; Received in revised form 21 March 2023; Accepted 11 April 2023
Available online 24 April 2023
0957-4174/© 2023 The Author(s). Published by Elsevier Ltd. This is an open access article under the CC BY license (http://creativecommons.org/licenses/by/4.0/).

a uniform type per sensor. Users are often interested in the values from one or more sensors in an interval, subject to an aggregation operation such as *min*, *max* or *mean*. In this respect, information models can help analytics application development produce scalable applications with relatively uniform and standardised integrations with these time series data sets. Yet, uniform and consistent representation and access to industrial information models including static and time series data is non-trivial and still an active research domain.

Semantic Web and linked data technologies have received much attention for their role in representing, capturing, and integrating industrial data (e.g., Giese et al., 2015; Kharlamov et al., 2017; Zheng et al., 2022a). Ontologies are used for modelling domains and integrating data, using knowledge representation languages such as OWL and RDF, while SPARQL is the key language in querying such integrated data sets. Ontologies provide higher level abstractions closer to end users' understanding (Soylu et al., 2018), apart from their advantages in terms of data integration and reasoning. Integrating and accessing data through ontologies and related technologies is often referred to as ontology based data access (OBDA) (Corcho, Priyatna, & Chaves-Fraga, 2020; Poggi et al., 2008). SPARQL databases are a good fit for models of industrial assets and their processes, but are not optimised for the time series data produced by the sensors and actuators in industrial assets as storage is not optimised for sequential reads of timestamped values (e.g. Bast & Buchhold, 2017) and serialisation/transport involves text-based representations (World Wide Web Consortium, 2013b). Similarly, time series databases lack features important for linked data. Queries over linked data often involves random access (e.g. a piece of equipment by name), which needs indices to perform well and heavily rely on joins. Indices are not present in many time series databases, and joins are not supported by time series databases such as InfluxDB (Bader et al., 2017; InfluxData, 2022a). Neither general SPARQL databases nor time series databases are well suited to support OBDA to contextualised time series data. One particular approach in OBDA which has received considerable attention is the notion of the Virtual Knowledge Graphs (VKG) (Xiao et al., 2019). With VKGs, data are kept in an underlying database, typically SQL, and queries written using SPARQL based on an ontology are translated into equivalent SQL queries on demand in order to improve the ease of data access. Using modern, cloud based infrastructure, VKGs appear better suited to query contextualised time series data.

Modern enterprises often use what are called data lakes using cloud infrastructure to store their time series data in the column oriented Parquet-format. Using distributed SQL query engines called data lakehouses, users can use on-demand cloud infrastructure to process queries over large data sets (Armbrust, Ghodsi, Xin, & Zaharia, 2021; Melnik et al., 2010). This means that compute (query processing) is separated from storage of data, which yields better utilisation of shared resources. We will use the term time series databases somewhat loosely in this article, and include modern data lake/data lakehouse architectures with columnar storage, as these are well suited and commonly adopted for time series data, even though they are more general. Like with specialised time series databases, most data lakehouses lack support for indices (Armbrust et al., 2021), but support joins and other operations required to implement SPARQL. Hence, contextualised time series querying using OBDA can be implemented on top of a data lakehouse infrastructure. As we will shortly discuss, such infrastructure is not always available in industrial settings. When integrating analytics applications with industrial data, it is not always desirable nor possible to do so in the cloud. This is due to the fact that for business critical applications, an internet or cloud outage can present too high a risk for operations (Lee, 2008). Besides, running an on premise data lake may be too costly, time consuming for many companies. Even in these situations however, the cloud has an important role to play in performing resource intensive training. Ideally, we want users to be able to use OBDA to access data from the cloud in a training setting, and to use the same queries to access data in a deployment setting using on-premise infrastructure. We will use the term *query portability* to mean support for executing the same queries across heterogeneous infrastructures. To enable SPARQL-based access to industrial time series data in way that supports query portability and leverages the capabilities of graph- and time series databases, we propose a hybrid query engine, namely Chrontext.

Chrontext relies on a set of annotations to a knowledge graph representing the static aspects of an industrial asset. We will refer to this graph as the *context graph*. These annotations allow nodes in the context graph to be associated with time series in the time series database. SPARQL enables queries over this architecture by a process of rewriting the original query to a static rewrite appropriate for the context graph, i.e., the static rewrite excludes time series information, but includes the mentioned annotations necessary to retrieve it. Next, the appropriate time series are retrieved from the time series database and combined with the results from the context graph to produce the query results. Chrontext partially follows virtual knowledge graph approach, as time series data is kept in time series databases, but static context data is provided by a SPARQL endpoint. Although hybrid approaches (e.g., Bakken, 2021; Hu et al., 2016; Steindl, Frühwirth, & Kastner, 2019) have been discussed in the literature previously, a detailed description of the mechanics of a hybrid query approach does not appear to exist. We provide such a detailed description of our approach in this article. To show the validity of our approach, we provide a detailed proof showing that our approach of rewriting queries and combining their results leads to correct SPARQL 1.1 results.

The solution is implemented in Rust, and is based Polars (Vink, 2022b), a fast library for manipulating in-memory tables stored using Apache Arrow. The solution does not implement a SPARQL endpoint, but instead returns a Polars DataFrame. DataFrames are in-memory tables, and are extensively used in data engineering and data science. Chrontext currently supports SQL and OPC UA Historical Access (HA) time series database APIs. When time series data is stored in a data lakehouse, it is often accessible through SQL. OPC UA HA is an API that is often used to access time series data in on premise, industrial settings. When the time series database supports Apache Arrow, serialisation costs can be reduced drastically compared to ODBC connections (Ahmad, 2022). Our solution supports time series APIs that are not SQL based, and adjusts the type of processing, which is offloaded onto the time series database according to the specified capabilities of the time series database. Supporting heterogeneous time series databases allows us to support industrially relevant use cases in on-premise environments without an SQL integrator. This means that contextualisation and storage of time series data becomes decoupled, allowing organisations to potentially develop analytical applications in the cloud and deploy them on existing infrastructure on-premise, using the same queries to access time series data, and realising the value of standardised information modelling in a practical way.

In order to evaluate the performance of our approach, we compared it against Ontop (Xiao et al., 2020), which is one of the most prominent open source virtual knowledge graph systems. The data lakehouse Dremio allows data lakes and databases to be made available using a common SQL interface, and since it is supported by Ontop, allows the VKG approach to work in hybrid architectures. We constructed a synthetic data set based on a set of wind farms represented using the Reference Designation System (RDS) for Power Systems (International Organization for Standardization, 2022) based on an example wind power case (RDS 81346 Technique ApS, 2022). The RDS can be applied to a wide variety of industries and to building management. The queries we use in the benchmark are fairly straightforward, and focus on the core task of using an RDS information model to find and extract time series. Comparing the performance of our solution to Ontop on twelve different queries, we found that the memory consumption of the SQL queries generated by Ontop causes Dremio to run out of memory in one out of three queries. Chrontext completes all queries and uses very little memory in Dremio. For queries completed by both solutions, our

solution completes the queries between 10x and 85x faster than Ontop. For the case of using a widely applicable and adopted information modelling approach for industry, and extracting time series data in a straightforward way using SPARQL, our approach outperforms the leading open source VKG by a wide margin. The evaluations lead us to suggest routes to improve performance for Dremio, Ontop and Chrontext respectively.

The rest of the article is structured as follows. We begin by introducing background relevant to our work in Section 2. The background section covers OPC UA, information modelling with ISO/IEC 81346 (RDS), column based data analysis with DataFrames and modern data warehouse concepts of the data lake and data lakehouse. We discuss related work on hybrid query systems and virtual knowledge graphs in Section 3. We motivate and describe the problem of querying hybrid architectures with SPARQL, describing our requirements and how existing hybrid querying and VKG solutions fail to meet them in Section 4. We then describe the solution approach in Section 5, by elaborating how the approach rewrites and processes one of the queries from the benchmark. We describe the solution approach in mathematical detail and prove the correctness of the approach in Section 6. Implementation details can be found in Section 7, describing how we leverage existing libraries, support for time series APIs as well as important optimisations when the time series database is a SQL data lakehouse. We present the set-up and discuss the results of our evaluations in Section 8, looking in detail at why Chrontext outperforms Ontop. We conclude that our solution approach and implementation meets our requirements in Section 9. We discuss the novelty of our approach, the possible benefits for application development. Finally, we discuss current limitations of the approach and implementation, how they can be overcome and outline future work.

## 2. Background

In this section, we introduce technologies, approaches and standards underlying our work or used to put our work into relevant industrial context. Semantic Web technologies and the SPARQL query language are core technologies adapted in this work, while OBDA and VKGs allow us to use ontologies and SPARQL as high level abstractions to integrate and access data in existing databases. Given our industrial focus, we extend our solution to support OPC UA, which is a well known set of specifications for interoperability in industry. Additionally, our industrial approach targets organisations that have created standardised representations of their assets and processes. The ISO/IEC 81346 standard is an important and broadly applicable standard in this respect, and it is used in our running example and evaluation. We incorporate Apache Arrow into our solution in order to cater for data engineers and data scientists and leverage the performance improvements this technology can yield in analytic settings. Analytical data is typically stored in data lakehouses, and understanding how data lakehouses work is important to compare the performance of our solution with that of a VKG.

### 2.1. Semantic Web technologies

Semantic Web technologies are a set of technologies for representing and exchanging knowledge on the Web in standardised formats and they allow data integration and reasoning (Hitzler, 2021). This vision has for the most part not come to pass yet, although the technologies developed are useful in developing data intensive applications (Kleppmann, 2017), including industrial and public data integration scenarios (e.g., Soylu et al., 2022; Zheng et al., 2022b). A core Semantic Web technology is the Resource Description Framework (RDF); it is a graph-based data model and associates Uniform Resource Identifiers (URI) with entities in a domain of interest, and describe these resources and their relationships using triples. The first element of a triple is a URI, called the *subject*. The second element is also a URI which denotes a relationship or property that the URI has, and is called the *predicate*. The third element of the triple is either a piece of data, or another URI, and is called the *object* of the triple. Collections of such triples are called knowledge bases. Ontologies describing a domain of interest in terms of concepts, properties, and relationships are another important part of the Semantic Web, which employs languages such as RDFS and OWL for incorporating more meaning and reasoning power into RDF graphs; there are multiple reasoning engines available (Singh, Bhatia, & Mutharaju, 2020). Finally, SPARQL is a powerful query language used to interrogate knowledge bases. SPARQL databases are also called triple stores. There are multiple highly scalable commercial and open source SPARQL engines (see Ali et al., 2021).

### 2.2. OBDA and VKGs

OBDA refers to the use of Semantic Web technologies to improve the accessibility of data to users (Soylu et al., 2018). This is done by abstracting away technical details connected to how data are stored in databases. Instead, the domain is modelled using Semantic Web technologies, i.e., ontologies, with domain-oriented predicates, which has a non-trivial relationship to the underlying data (typically in a relational database). The specification of relationships between the underlying data and the ontology elements is known as a mapping (Xiao et al., 2018). Using OBDA, users can formulate queries and explore results in a language that is closer to the domain using ontology terms and SPARQL, and then queries translated into the query language(s) of the underlying (relational) database(s). An important distinction in OBDA is whether triples are materialised into a SPARQL database or not. The class of systems where triples are only virtual and only exist implicitly or as query results to users is called VKG (Xiao et al., 2019). The open source software Ontop (Calvanese et al., 2017b) is among the most popular open source VKGs (Xiao et al., 2018).

### 2.3. OPC UA historical access

OPC UA is a set of standards facilitating secure and interoperable communication and semantics in industry (OPC Foundation, 2022). In industrial automation, the life of protocols, software, and devices is longer than in many information technology settings, and OPC UA allows a myriad of legacy technologies to be abstracted away. It is a large standard covering many use cases (Schleipen et al., 2016) ranging from the field level in the automation pyramid to cloud deployments. Crucially, OPC UA supports information modelling allowing the meaning and context of data from sensors and actuators to be preserved. OPC UA information models contain nodes that may be associated with historical data. OPC UA Historical Access allows clients to ask for historical data from such nodes which is either raw or aggregated (OPC Foundation, 2018a), but is agnostic with respect to how these data are stored. That is, it is up to OPC UA service implementers to decide where and how to store data. Servers implementing OPC UA HA are often called historians, and may be used to support applications that rely on historical data, such as dashboards and intermittent analyses and reports.

### 2.4. ISO/IEC 81346

The Reference Designation System (RDS) is an ISO and IEC standard (81346) for uniquely referring to components of an asset such as a building, factory or power plant across the lifecycle of an asset (International Organization for Standardization, 2009). By describing how assets are organised along different aspects, they allow models of such assets to be relatively static, even if equipment realising important functions are replaced (Pfaffel, Faulstich, & Rohrig, 2017). Using models based on RDS, one can navigate to equipment using such as the role it plays in achieving some function (e.g. producing power or ensuring safe operations) or by where it is located physically, or by

some combination of such aspects. There are several companies offering products to structure static asset data in accordance with the RDS. The Engineering Base product is a commercial collaborative engineering tool sold by AUCOTEC (2022). It supports representing industrial assets in RDS and is in use across multiple industries. Similarly, Keel Solution focuses on using RDS to manage information in the energy sector, including wind farms (Keel Solution, 2022). The existence of these commercial offerings corroborate the industrial relevance of our RDS wind power example presented later.

### 2.5. Column oriented DataFrames and Apache Arrow

Columnar data layouts have been found to perform better than row based layouts for databases that perform analytical processing (Abadi, 2008; Kleppmann, 2017). When performing analyses locally, data scientists use DataFrames, which are typically in-memory columnar data structures. We assume that data scientists using our solution to extract time series data will prefer to work with DataFrames.

Apache Arrow is a specification for laying out data in memory in a column oriented way (The Apache Software Foundation, 2022a), to facilitate high performance analytical processing and efficient, zero copy data exchange between processes on the same computer. For instance, a database backed by Apache Arrow could execute procedures defined in R or Python on its data without copying data. Apache Arrow serves as the backend for multiple DataFrame-like interfaces (Neal Richardson et al., 2022; The Apache Software Foundation, 2022d; Vink, 2022b). Several analytically oriented databases have adopted Apache Arrow as their native format, including Dremio and InfluxDB IOx (Dremio, 2022; InfluxData, 2022b).

Data Scientists must often retrieve data using SQL from analytical databases, which is made available through row based APIs. For instance, the Open Database Connectivity (ODBC) API requires that data is made available in a row oriented way. The target representation for such queries is often a column oriented DataFrame. This process is wasteful if the source database is column oriented, as the intermediary row-based representation is superfluous. Apache Arrow Flight is a specification for transferring Apache Arrow data over a network in a column oriented way, avoiding the above problem. Apache Arrow Flight has been found to drastically improve the throughput of database connections in such cases (Ahmad, 2022).

### 2.6. Data lakes, Dremel and data lakehouses

Dremel is a massively scalable Google-internal distributed query engine, which changed the way large data sets are queried (Kleppmann, 2017; Melnik et al., 2010). Before Dremio, querying large data sets was done using MapReduce-based solutions that required coupling of computation and data, as data must be loaded into the cluster before queries can be run on the data set (Kleppmann, 2017). Such coupling leads to inefficient resource usage. In contrast, Dremel allowed what is termed "in situ" query processing, meaning that data can be stored in a dedicated storage mechanism and rapidly accessed at query time (Melnik et al., 2010). Such a passive data storage mechanism is called a data lake.

Dremel has inspired a number of products for scalable SQL data integration decoupling storage and compute layers, which are sometimes called data lakehouses (Armbrust et al., 2021). Data lakehouses often rely on data stored in the columnar Apache Parquet (The Apache Software Foundation, 2022b) file format, which are effectively converted to Apache Arrow. Data lakehouses use data partitioning schemes to limit the amount of data that must be processed instead of using indices. One popular data lakehouse is Dremio (2022) (a play on "Dremel" Preimesberger, 2017), which is supported by Ontop. Relying on the scalability of data lakehouses is one possible way to scale ODBA generated SQL queries over time series and context models.

## 3. Related work

In this section, we discuss the related work on OBDA involving time series data; support for queries that span context and time series data in OPC UA; and, hybrid query architectures. We then summarise the related work across various categories.

### 3.1. Applications of OBDA to time series

OBDA has been applied to contextualised access to time series data at Siemens by Kharlamov et al. (2014). The data in the Siemens case mainly consist of time series data from instrumented power production equipment such as turbines but also of important static data which contextualises the time series data. The authors employ a query language for windowed streams of linked data called STARQL (Özçep, Möller, & Neuenstadt, 2014) to support the unique requirements of their use case, but use Ontop to transform such queries into SQL queries for the underlying databases. The execution of these SQL queries are in turn orchestrated across a distributed infrastructure. The use of OBDA for accessing time series data from manufacturing to support analytics use cases was proposed by Mörzinger et al. (2018). The feasibility of using SPARQL to access such data is demonstrated, but an OBDA approach is proposed instead of triple stores to handle large volumes of time series data.

Mörzinger (2019) implements ontology based access to time series data arising in manufacturing in order to perform analyses. Data is stored in an SQL-database, and Ontop is used to access data using SPARQL. Mörzinger (2019) argues on the basis of literature review and interviews with subject matter experts that querying contextualised time series is crucial in the manufacturing domain. Furthermore, Mörzinger finds that aggregation support in OBDA is insufficient to cover the analyses of manufacturing data, and suggests a that a domain specific language is required to meet these needs. Brandt et al. (2019) also represents such an approach, and allows aggregations such as those required by Mörzinger (2019) to be formulated. The language is transformed into SQL queries and executed in a distributed way on an Apache Spark cluster (The Apache Software Foundation, 2022c) which supports user defined aggregate functions. Similarly, Güzel Kalayci et al. (2018) extends OBDA to cover derived temporal relationships, such as the validity of a statement in an interval, and map these relationships to SQL queries. Güzel Kalayci et al. (2018) provide benchmark results on an open data set of hospital admissions which is loaded into a PostgreSQL database. Calvanese et al. (2017a) focus on the case of encoding higher order properties found in the field of process mining (Van Der Aalst, 2012) as higher order abstractions that are queryable using SPARQL. The field of process mining has produced techniques for analysing log data, a type of time series data. As such, it enables analysts engaged in process mining to pose queries on an appropriate, non-technical level of analysis.

Brandt et al. (2019), Güzel Kalayci et al. (2018), and Özçep et al. (2014) are the only examples we know of that benchmark OBDA for time series data. Only Güzel Kalayci et al. (2018) provides a data set that is openly available, and benchmarks provided by Brandt et al. (2019) and Güzel Kalayci et al. (2018) concern features and aggregation functions that are not found in most OBDA solutions. Botoeva et al. (2018) considers the extension of OBDA to NoSQL data sources, with an implementation and benchmark for the case of MongoDB (2022), while hybrid architectures are not discussed.

### 3.2. OPC UA query support

OPC UA inherently separates static data used to model static assets from dynamic data generated by those assets. Instantiated information models are static artefacts that are represented using formats such as XML and JSON. These instantiated models describe which nodes contain historical and real time values, but do not themselves contain

them. The OPC UA Query service specifies a query API allowing the use of information models as context to retrieve time series data. The API however has been criticised for being highly verbose and for only being accessible through an API (Schiekofer & Weyrich, 2019). There are no known commercial vendors that support OPC UA queries.

An effort exists to transform OPC UA information model instances to RDF (e.g. Schiekofer et al. (2019) and Perzylo et al. (2019)), so that they can be queried using the SPARQL query language. This approach is successful in querying the resulting information models. However, time series data are not suitable for storage in most SPARQL engines. Steindl et al. (2019) describe an approach for solving this problem based on the custom property functions found in Apache Jena, which allow users to implement functions that may be called as part of queries, and exposed to query processing as data properties. Bakken (2021) instead relies on SPARQL query rewriting to access time series data indexed by OPC UA information models. The present work is an extension and generalisation of this rewriting approach.

In a similar vein to OPC UA queries, Alvanou, Lytra, and Petersen (2018) investigate SPARQL queries for contextualised access to data from manufacturing equipment exposed by the MTConnect standard. The MTConnect standard "... offers semantic vocabulary for manufacturing equipment to provide structured, contextualized data with no proprietary format" (MTConnect Institute, 2022). There exists official mappings from MTConnect to OPC UA available on the MTConnect website. Alvanou et al. (2018) only conduct a feasibility study of queries that span static context and time series data, and store small example data sets (each less than 10 kB) in a triple store.

### 3.3. Querying hybrid architectures

Graube, Urbas, and Hladik (2016) describe a hybrid approach based on combining a static SPARQL database with "rapidly changing transient data", for instance from OPC UA. The authors propose that the SPARQL 1.1 Federated Query Extension (World Wide Web Consortium, 2013a) be used to integrate dynamic data with static data, and that OPC UA servers are equipped with linked data adaptors allowing them to form part of a federated query. The authors acknowledge that their proposed architecture incurs a performance penalty, but argue that scaling to large data sets is not critical for their use case which retrieves a small snapshot of transient data in context.

The domain of building information modelling (Borrmann, König, Koch, & Beetz, 2018) also contains work on querying hybrid architectures (Tang et al., 2019). Notably, Hu et al. (2016) describe an approach for extracting building performance data, which relies on first querying a SPARQL database to determine which data to extract from one or more SQL databases. The data from SQL is then queried in a second step. Hu et al. (2016) find that their two-step process outperforms an approach which stores all data in a SPARQL engine. Petrova et al. (2019) similarly combine SPARQL queries that determine appropriate time series data, but instead of an time series database rely on a HTTP endpoint per time series that accepts parameters such as the time interval and a refresh rate. The approach is used to discover associations between sensor data.

Donkers et al. (2021) use SPARQL to query static building context, which is used to determine what data to extract from the time series database InfluxDB (InfluxData, 2022a). InfluxDB does not provide SQL support. The extracted time series data sets are used to compute proxies for building performance and comfort. The approach by van Gool, Yang, and Pauwels (2021) has similar goals but instead integrate a building topology model in SPARQL and time series data in NoSQL database MongoDB in order to compute the above proxies. Esnaola-Gonzalez and Diez (2019) describe an architecture for querying data pertaining to energy use in residential buildings. These data are to be used for a mobile application for economical energy use. Esnaola-Gonzalez and Diez (2019) rely on queries for InfluxDB that are stored in a SPARQL database. First, appropriate queries are retrieved by running SPARQL queries, and then those queries are run to retrieve appropriate data sets from InfluxDB. Performance is an important motivation for using a hybrid architecture in most of the aforementioned work, as SPARQL engines are not seen as appropriate for time series data. Although several studies discussed contain benchmarks, the associated data sets are not openly available.

### 3.4. Summary of related work

In summary, we see that there are heterogeneous time series data sources such as SQL, non-standard HTTP endpoints, NoSQL time series databases (InfluxDB and MongoDB) and OPC UA Historical Access. There are also a variety of approaches to integrating time series data with data stored in SPARQL, involving query rewriting (Bakken, 2021), property value functions (Steindl et al., 2019), relying on SQL-based data integration (Kharlamov et al., 2014), SQL based data lakehouses such as Dremio (2022) and client side integration of data (e.g. Donkers et al. (2021) and Hu et al. (2016)). In the field of OBDA, researchers have focused on using queries to analyse time series data of various types, extending OBDA or creating new languages to express these properties and encoding them as SQL queries. The problem of time series data extraction using SPARQL has received less attention.

Contextualisation is seen as important for extending the capabilities of applications and in doing so in an automated way with less need for engineering. For instance, contextualisation of time series data using a specialised ontology allows (Donkers et al., 2021) to decide automatically on which streams of sensor data are relevant to models of building performance, and how they are relevant. Although proponents of hybrid architectures have conducted benchmarks, these have been done in comparison with storing all data in triple stores, and data sets are not openly available. Benchmarks comparing the performance of OBDA and hybrid architectures for query processing appear not to exist in the literature.

## 4. Problem analysis

The aim of this work is to provide analysts and data engineers with effective ontology based data access to data from industrial assets. These are contextualised in standardised ways, leveraging and being compatible with modern analytical technologies. In particular, we argue that it is time series data which represents both the main volume and velocity of data in many industrial situations, while contextual information is often of far smaller volume and less time sensitive. We discuss a particular feature which we call *query portability*, which is the notion that the same queries can be reused in cloud and on premise infrastructure. We argue that query portability is of particular importance in industrial settings. We then conclude with a set of requirements as a result. Next, we discuss the limitations of existing solutions to meet these requirements.

### 4.1. Motivation

In building management, manufacturing, and energy domains, there have been persistent trends of increased digitisation in last decades (e.g., Bucchiarone et al., 2019; Kanabar, McDonald, & Parikh, 2022; Zheng et al., 2022c). The cost of sensor technology has decreased drastically, and there has been considerable effort towards creating standardised models of physical assets (e.g., digital twins Intizar Ali et al., 2021). In addition, these physical assets are equipped with sensors and control systems. Therefore, it is essential to capture and represent enormous amount of data being generated and enable convenient and timely access.

Considerable amount of work has been done on scaling triple stores to handle enormous sets of linked data (e.g., Abdelaziz et al., 2017; Huang, Abadi, & Ren, 2020). Although models of physical assets can be

very large and detailed in industry, the time axis can contain many orders of magnitude more data. This means that industrial assets can only grow so big due to commercial and physical limitations. For instance, the Gansu wind farm in China was the biggest in the world in 2017 and has 7000 wind turbines (Hernández, 2017), but most wind farms are much smaller due to limitations in the available area. The data generated by sensors and control systems associated with the turbines change very rapidly and accumulate over time. It is appropriate to think of models of physical assets as static, as re-configuring an industrial asset is a time consuming process, and even in demonstrators that display extremes in flexibility, such changes takes several minutes (Kim et al., 2020). We suspect that it is more common for changes to take days, weeks or months. That is for instance the case in commissioning of a new wind turbine. Improving the performance of SPARQL querying for industrial applications should take into consideration the fact that considerable amount of the data consists of time series.

For analysts and software systems that manage these physical assets, it is often important to process historical data accumulated from sensors and control systems. Examples include performance monitoring and analytics involving prediction and optimisation. Such analyses and applications often involve the use of data from multiple, related sensors. Analysts must extract sequences of such values, while keeping track of the context. Advances in machine learning such as deep learning mean that it is possible to improve prediction performance with ever increasing sets of data. The last ten years have seen massive industrial adoption of machine learning using these data-hungry approaches, shifting demands more towards time series data. As has been discussed extensively in the literature, OBDA can provide an integrated view of asset and process models and historical sensor data, which is more accessible to end users than querying technically oriented SQL schemas, or worse, manually integrating data across systems containing static and dynamic data. OBDA has the potential to speed up the development of analytical applications, and if models are standardised, to improve application reuse and ease of deployment. Moreover, OBDA can reduce the need for costly data (re-)modelling (Kharlamov et al., 2014).

The last decade has seen important improvements in how analytical data is stored and transferred. In particular, analytical data tends to be stored in the binary, compressed, columnar formats (Bian & Ailamaki, 2022; Jin, Bian, Chen, & Du, 2022). With the advent of Apache Arrow and Apache Arrow Flight, analytical data sets can also be transferred in this way between processes and over networks (McKinney, 2019). In-memory data processing with column oriented tables of data called DataFrames is very popular among data scientists and data engineers. The Pandas library has pioneered and popularised this approach (McKinney, 2010; The pandas development team, 2020). At the time of writing, it had 98 million downloads in the last month from python package repository PyPI. The requests-library, an essential library for HTTP-requests in Python had 227 million downloads over the same period. In order to be relevant in the world of analytics, linked data approaches should support and incorporate recent technologies having considerable use.

There are approaches that go in the direction of allowing users to specify the analysis of time series data in the query language as discussed in the related work section (e.g. Brandt et al., 2019). The goal in this contribution is not for users to specify analyses in the query language, but to make contextualised time series data available to established tools that analysts already know how to use. We have argued that there is a need for scalable SPARQL querying over data sets containing both static context data as well as dynamical time series. OBDA is an approach to SPARQL query processing that allows us to utilise existing infrastructure. In particular, OBDA can allow SPARQL users to utilise modern technologies for high performance extraction of time series data sets.

### 4.2. Query portability

A trend has been for companies to build cloud-based data lakes, which may be queried using an SQL based data lake house. Given that Ontop supports one such data lake house (Dremio) (Ontop VKG, 2022), one may choose to realise OBDA for industrial time series in this way. However, a fully cloud based OBDA deployment may not be compatible with the availability constraints found in industry, and the full benefits of OBDA may not be realised:

- First, if analytics applications have a critical role to play in operations, plant managers may be reluctant make themselves dependant on cloud based infrastructures and an operational internet connection. Losses from lost production are typically much larger than the compensation cloud vendors are prepared to provide under standard service level agreements. These agreements typically provide a refund for cloud costs in periods with degraded service.
- Second, security considerations often mean that data tends to move only *to* the cloud, but not necessarily back to the industrial asset. Exposing the ability to influence physical processes to the cloud is seen as a major security risk, particularly in an age where security breaches have become highly profitable and common.
- Third, making all sensor data available to the cloud places enormous demands on network infrastructure for upload, and may not be economically feasible. In such situations, a subset of the data may be uploaded to the cloud in order to rapidly develop analytics applications that are deployed then deployed to on-premise infrastructure.
- Fourth, data lakehouses can also be deployed on premise. However, requiring such infrastructure artificially limits the set of organisations which can use the solution both due to the cost involved and limited support for industrial standards such as OPC UA HA in data lakehouses.

Companies may still want to leverage the power of cloud computing to rapidly train and prototype machine learning models. Once models are trained however, the above considerations may well mean they should be deployed on premise (cf. Pedone & Mezgár, 2018). To accomplish this goal, data access should be possible with the same queries across infrastructures. We will refer to this capability as *query portability*. The notion of query portability requires that the query engine is able to support a wide variety of interfaces to time series data which do not necessarily support SQL. As discussed in the related work section, many such varieties have been discussed in the literature. For some time series interfaces, it is very important to limit the set of data being queried. OPC UA allows federation (OPC Foundation, 2018b), and in some cases this means that accessing data historical time series data on a OPC UA server results in federated requests for data stored OPC UA servers co-located with the equipment in question. If these queries are too broad, federated queries may result in an infeasible amount of data given network throughput constraints.

Following this discussion, we formulate a set of requirements; the proposed solution must:

- R1 support context based extraction of time series data using SPARQL 1.1,
- R2 achieve high throughput low latency time series data extraction,
- R3 fit well into established data science and data engineering tools,
- R4 support heterogeneous APIs for time series data, among them OPC UA HA and SQL in order to provide query portability,
- R5 and offload computation to the time series database whenever possible in order to improve performance.

The first two requirements simply constrain the solution to existing technologies with extensive tooling. This makes our work feasible in terms of implementation and in terms of comparability with existing

work. The third requirement constrains us to a highly interoperable and performant data engineering tools, such as DataFrame. The final two requirements allow us to support query portability that makes the most of the computing resources at hand. In the next sections, we discuss disadvantages of existing solutions with respect to these requirements.

### 4.3. Disadvantages of existing solutions

In this section, we first discuss how solutions that store all data in triple stores lose out on the performance improvements gained by exploiting the structured nature of time series data in industrial settings. Secondly, we discuss how fully virtual knowledge graphs lose out on the performance improvements gained by representing static context with indices and do not yet fully leverage columnar formats. While these approaches offload processing to a secondary database, neither of these approaches support query portability. Finally, we discuss the property function approach, which supports portability, but represents a procedural extension of SPARQL 1.1, and does not utilise modern columnar formats.

#### 4.3.1. General triple stores are not optimised for time series data

Graph databases typically build indexes to accelerate complex graph queries. Such indices are costly to build and maintain, and decreases the throughput performance of data inserts (Kleppmann, 2017). Moreover, data insertion in SPARQL is text based, which creates a large serialisation and transport overhead cost. Time series data arising from sensors can however arrive at very high rates. Although columnar storage has been beneficial in SPARQL databases (Abadi, 2008; Albahli & Melton, 2016; Erling, 2012), this does mean that data are laid out sequentially in memory according to timestamps.

Time series data can benefit greatly from sequential columnar storage. This is because we tend to access- and perform computations over intervals of time series data, and since storing data sequentially makes it more amenable to compression (Kleppmann, 2017), greatly increasing the amount of data that can be stored (see e.g. Villalobos et al., 2020). In an industrial use case, we may poll the discrete state of some sensor (e.g. valve open or closed) at regular intervals using the Modbus (The Modbus Organization, 2012) protocol. Likely, there are long periods where the discrete state (e.g. valve open) is identical, and a database system laying out such values in sequence may be able to compress them very efficiently.

Triple stores that can be configured to be especially suited for time series data are of course possible, and the present work takes some steps on how to construct such a database from a general triple store and a time series database.

#### 4.3.2. Disadvantages of existing VKGs

Virtual knowledge graphs are typically constructed by mapping data from SQL databases via queries to triples based an ontology at query time. It is possible to use virtual knowledge graphs such as Ontop to query contextualised time series data either by storing both time series and context in the same SQL database, or by using a SQL based data integrator/data lakehouse such as Dremio. In order for SPARQL queries to be performant, the database must maintain indices. Data lakehouses often do not use indices but rely on partitioning and materialised aggregations to speed up queries (Armbrust et al., 2021; Weintraub, Gudes, & Dolev, 2021). These strategies do not support the flexible access patterns possible with SPARQL well. VKGs implementing the SPARQL 1.1 protocol (World Wide Web Consortium, 2013b) rely on text-based transport of data, for instance using JSON or XML. Transporting data as text entails a lot of overhead, on the order of 10 times depending on the underlying data type. Data is harder to compress, and incurs serialisation costs if the target structure is an analytics friendly columnar representation such as a DataFrame.

#### 4.3.3. Disadvantages of property functions

The property function approach for querying hybrid architectures described by Steindl et al. (2019) allows SPARQL databases to define property functions with user defined implementations. Steindl et al. (2019) implement property functions for accessing time series data from the OPC UA historical data access service. The main issue with property functions is that they mix imperative non-standard constructs with a declarative query language, and makes it impossible to reach R1. The longevity of declarative languages such as SQL and CSS is explained in part by the fact that they are declarative (Kleppmann, 2017). Declarative SQL queries are powerful because they allow implementations to be drastically rewritten and optimised without having to make a single change to the query. Although performance improvements certainly are possible for imperative languages, they are harder to accomplish, since execution flow is fixed to a much greater degree.

## 5. Solution approach

This section describes the proposed solution approach with a running example, including the static rewriting process and how it applies to the running example; the process of combining static query results with time series query results; the process of preparing time series queries and pushing down computation into the time series database; and, pushdowns and time series execution for OPC UA and for SQL databases.

### 5.1. Overview of the solution approach

Our solution approach assumes that static context data and time series data are stored separately. Static context data is assumed to be available through a SPARQL 1.1 endpoint. We assume that time series data are collections of timestamps and values. Time series data is assumed to be made available through an endpoint where time series can be accessed by their identifiers and by bounds on the timestamp. Crucially, the SPARQL database is annotated with metadata such as the identifiers of these time series, so that a compound, inferred graph may be constructed. Our approach to processing hybrid SPARQL queries is to rewrite the original SPARQL query into one or more SPARQL queries that only concerns static context data, and one or more queries for a time series database. Users should not have to know about the annotation scheme in the SPARQL database, and should be able to make hybrid queries transparently. The static queries are not merely fragments of the original query, but are extended to retrieve the relevant metadata about the time series. We denote the phase where we create static and time series queries the *rewriting* phase.

The queries for the time series database are better thought of as proto-queries that are translated into the appropriate API call for the underlying time series database such as SQL or OPC UA HA. For performance, it can be crucial to push operations such as filters or aggregations into a time series database, since these typically have greater computational resources than a client, doing so limits the amount of data that has to be transmitted to the client. Therefore, we have developed a method for pushing down parts of the query into the time series database. However, not all database backends support all pushdowns. For instance, OPC UA HA supports a limited set of aggregation operations, but does not permit filtering on values prior to aggregations. The method must therefore be configurable depending on the characteristics of the time series database.

When there are multiple queries involved, the results from one query should constrain the remaining queries where possible. One obvious case is that the static queries extract time series identifiers that are crucial for extracting the correct time series. However, other constraining relations are also possible. We denote the phase where the rewritten static queries and time series queries are executed, further constrained and combined the *combination* phase. Fig. 1 shows the dependencies between steps in query processing. In Fig. 2 we illustrate

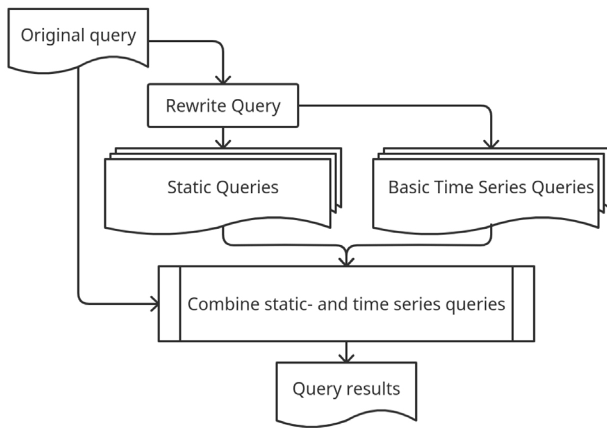



**Fig. 1.** Overview of query processing.

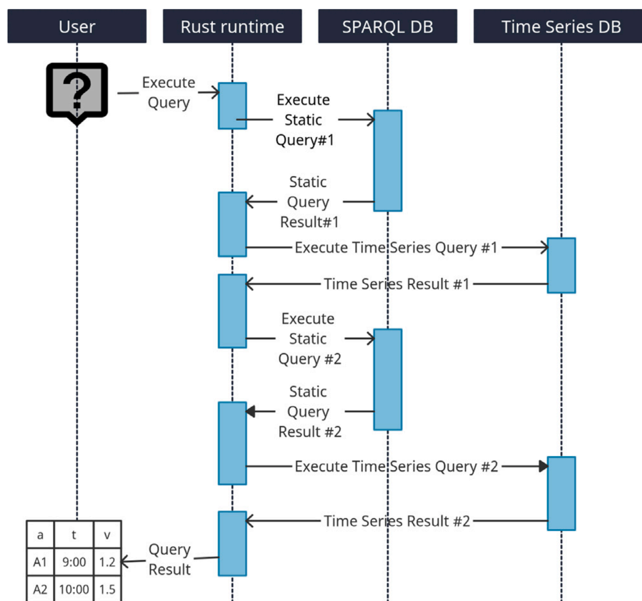



**Fig. 2.** Overview sequence diagram of the solution approach.

a possible sequence of executions of our solution approach. Note that the rewriting phase is completed immediately after a query is received, and that all subsequent processing is part of the *combination* phase.

It is necessary to define the scheme for metadata annotations to the static SPARQL graph, and the implied RDF representation of the composite graph, before we can discuss query processing. We discuss this scheme, introduce the running example, and discuss query processing using the running example in the following sub-sections.

### 5.2. RDF representation of time series

RDF representation of time series in queries is fundamental to our approach. An example use of the RDF representation is given in Fig. 3. We associate a resource with a time series using the `ct:hasTimeseries` predicate. The property and the time series resource is represented in a SPARQL database. The time series resource is always associated with a unique data type resource using the `ct:hasDatatype` predicate, typically in the xsd-namespace. The time series resource is also associated with a unique external identifier using the `ct:hasExternalId` predicate. We have assumed for simplicity that the external identifier is a string. This external identifier is associated with a collection of timestamped values in an external

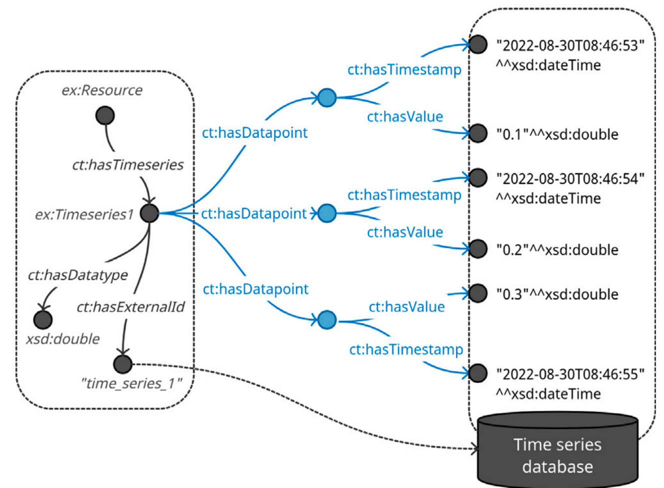



**Fig. 3.** Example use of the RDF representation for time series data.

database. Both the data type and the external identifier are stored in the SPARQL database, but users should not query them explicitly.

Instead, the timeseries will virtually be associated with the timestamped values using the `ct:hasDatapoint` predicate. Each datapoint has a timestamp (`ct:hasTimestamp`) and a value (`ct:hasValue`). The datapoints and the edges entering and leaving it exist only virtually, and are not stored in the SPARQL database. The timestamps and values are virtualized, and exist in the time series database. Data points however, are merely a syntactic construction in order to refer to pairs of timestamps and values.

### 5.3. Running example

At this point, we introduce the context graph used in our evaluation. Introducing this graph allows us to follow the processing of an example query in the subsequent sections. Our example is based on the Wind Power Example (RDS 81346 Technique ApS, 2022). It contains a number of sites, each with a number of wind turbines. Each wind turbine has a generator system, and the generator system has a time series representing the power production measured in Watts. The corresponding RDF graph is given in Fig. 4. In addition to representing the power production facility, we have attached metadata pertaining to the time series as described in Section 5.2. Note also that the technical particulars of the RDS standard require us to create auxiliary functional aspect nodes. The labels assigned to these nodes are strictly speaking associated with the edges, and we have accommodated this fact in our representation.

In the sections that follow, we will consider query processing for the query given in Fig. 5. This query identifies all wind turbines (`?wtur`) with functional aspect labels "A1"–"A10" belonging to the site labelled "Wind Mountain". We further identify the generators (`?generator`) belonging to each wind turbine. From the generators, we identify the "Production" time series, and extract the mean of the production each 10 min interval between two points in time (`?avg_val`).

### 5.4. Static query rewriting

The rewriting approach operates on graph patterns introduced in the SPARQL 1.1 algebra defined in the SPARQL 1.1 W3C recommendation (World Wide Web Consortium, 2013c). This algebra allows the semantics of SPARQL 1.1 queries to be defined. Many SPARQL 1.1 parsers, such as RDFLib in Python (Swartz et al., 2020) and Spargebra in Rust (Tanon, 2022b) produce the SPARQL Algebra structures corresponding to a query.

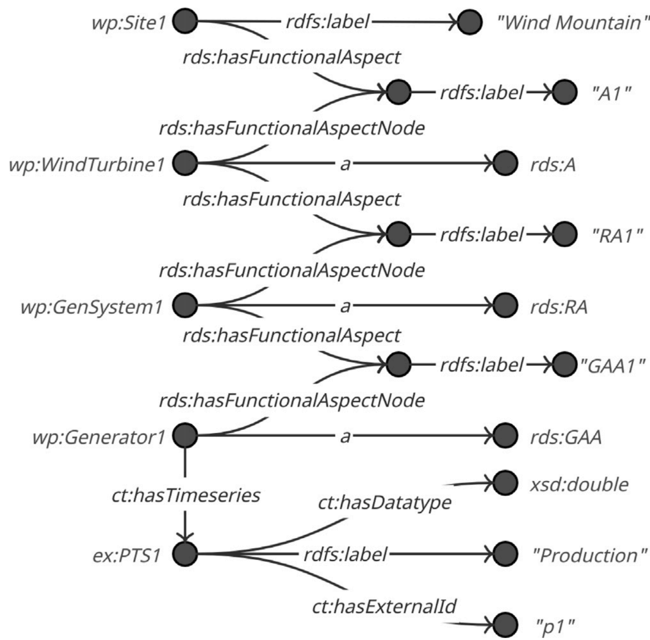



**Fig. 4.** Part of wind power graph used in the evaluation.

A query is first rewritten into one or more queries intended for the SPARQL database, when it is received. As the SPARQL database does not store timestamped values, nor the datapoints that link these to time series resource nodes, any references to data points, values and timestamps or expressions and bindings where they take part must be removed. The static queries must instead resolve the data types and external identifiers necessary to retrieve the appropriate values and timestamps from the time series database. These external identifiers and data types must be projected (selected) in the static rewrites of the query. Additionally, we must project any term stored in the SPARQL engine which is necessary to fully evaluate the query once the time-series data is available. To retrieve time series data, rewriting produces one or more proto-queries intended for the time series database.

The rules governing when we need to split our query into multiple queries to the static SPARQL database are covered in detail in Section 6. Generally, this happens when not doing so will result in the potential loss of a correct solution, or in potentially losing track of the correct number of solution duplicates.

### 5.4.1. *Rewriting basic graph patterns*

Basic graph patterns (BGP) are the fundamental building blocks of SPARQL queries, and correspond to collections of triple patterns. The basic graph patterns introduce variables. Property paths and other graph patterns can also introduce variables, but these are not discussed here for simplicity. When the static rewriting procedure encounters a basic graph pattern, the distinct time series variables, their associated data point variables, values and timestamp variables are collected in what we call a Basic Time Series Query (BTSQ). These are in fact proto-queries, and will be completed with additional information once the corresponding static query is evaluated and translated into the query API of the time series database. Triples involving data points, values and timestamps are removed from the basic graph pattern. Instead we introduce triples with variables to extract the external id and data type of the time series value. These variables are added to the BTSQ, and we ensure they are projected out through the outermost select-statement.

The names of variables are in general not sufficient to identify an occurrence of a variable in a query. We also record the sequence of graph patterns and expressions that contain the basic graph pattern when we store the BTSQ, in order to uniquely determine what BGP it corresponds

```
PREFIX xsd: <http://.../XMLSchema#>
PREFIX ct: <https://.../chrontext#>
PREFIX wp: <https://.../windpower_example#>
PREFIX rdfs: <http://.../rdf-schema#>
PREFIX rdf: <http://.../22-rdf-syntax-ns#>
PREFIX rds: <https://.../rds_power#>
SELECT ?site_label ?wtur_label ?year ?month ?day
       ?hour ?minute_10 (AVG(?val) as ?avg_val)
WHERE {
?site a rds:Site .
?site rdfs:label ?site_label .
?site rds:hasFunctionalAspect ?wtur_asp .
?wtur_asp rdfs:label ?wtur_label .
?wtur rds:hasFunctionalAspectNode ?wtur_asp .
?wtur rds:hasFunctionalAspect ?gensys_asp .
?wtur a rds:A .
?gensys rds:hasFunctionalAspectNode
    ?gensys_asp .
?gensys a rds:RA .
?gensys rds:hasFunctionalAspect ?generator_asp .
?generator rds:hasFunctionalAspectNode
    ?generator_asp .
?generator a rds:GAA .
?generator ct:hasTimeseries ?ts .
?ts rdfs:label "Production" .
?ts ct:hasDataPoint ?dp .
?dp ct:hasValue ?val .
?dp ct:hasTimestamp ?t .
BIND(10 * FLOOR(minutes(?t) / 10.0) as ?minute_10)
BIND(hours(?t) AS ?hour)
BIND(day(?t) AS ?day)
BIND(month(?t) AS ?month)
BIND(year(?t) AS ?year)
FILTER(?site_label = "Wind Mountain"
    && ?wtur_label in ("A1", "A2", "A3", "A4",
        "A5", "A6", "A7", "A8", "A9", "A10")
    && ?t >= "2022-08-30T08:40:00"^^xsd:dateTime
    && ?t <= "2022-08-30T21:40:00"^^xsd:dateTime) .
}
GROUP BY ?site_label ?wtur_label
    ?year ?month ?day ?hour ?minute_10
```

**Fig. 5.** Grouped production query with ten wind turbines from the evaluation. The query has been slightly rewritten for presentation purposes. It serves as our running example.

to. We call this sequence the path. Fig. 6 shows the basic graph pattern belonging to our running example and the rewritten example graph pattern is given in Fig. 7. We note that logically speaking, this basic graph pattern has become weaker. Our metadata requirements dictate that any time series should have both an associated data type and external id. A time series may however be empty. The basic time series query which is created when rewriting the basic graph pattern is shown in Fig. 8.

### 5.4.2. *Extend graph pattern rewriting*

The extend graph pattern contains an expression bound to a variable, together with an inner graph pattern. Extend graph patterns result from the BIND-keyword, but can also result from non-aggregating expressions that are applied to a variable after the SELECT keyword, e.g. `((?a + ?b) as ?absum)`. We keep the extend graph pattern during rewriting only if all variables involved in the expression are

```
?site a rds:Site .
...
?generator ct:hasTimeseries ?ts .
?ts rdfs:label "Production" .
?ts ct:hasDataPoint ?dp .
?dp ct:hasValue ?val .
?dp ct:hasTimestamp ?t .
```

**Fig. 6.** Basic graph pattern belonging to the running example query.

```
?site a rds:Site .
...
?generator ct:hasTimeseries ?ts .
?ts rdfs:label "Production" .
?ts ct:hasExternalId ?ts_external_id_0 .
?ts ct:hasDatatype ?ts_datatype_0 .
```

**Fig. 7.** Rewritten basic graph pattern belonging to the running example query.

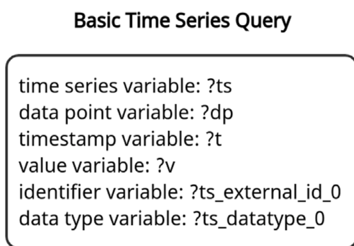



**Fig. 8.** Basic time series query resulting from the basic graph pattern in our running example.

```
BIND(10 * FLOOR(minutes(?t) / 10.0) as ?minute_10)
BIND(hours(?t) AS ?hour)
BIND(day(?t) AS ?day)
BIND(month(?t) AS ?month)
BIND(year(?t) AS ?year)
```

**Fig. 9.** Extend patterns from the running example query.

present in the inner graph pattern after it has been rewritten. In the case of the running example, there are five layers of extend graph patterns, nested on top of the basic graph pattern discussed previously. These extend patterns are shown in Fig. 9. All these extend patterns are dropped in the static rewrite as the data are not available in the static context graph.

### 5.4.3. Filter graph pattern rewriting

In the SPARQL algebra, a filter graph pattern is composed of an inner graph pattern and a boolean-valued expression. They result from the FILTER-keyword in SPARQL queries, which serves to limit the set of solutions to the inner graph pattern according to some condition that is specified immediately after the FILTER keyword. These conditions can be highly complex, and even allow nested subqueries. When our rewriting approach rewrites filter graph patterns, the expression must be rewritten to remove any variable located in or derived from the time series database. However, care must be taken to not create a rewritten condition that is stricter than the original condition, because this can cause us to lose solutions. For instance, a condition of the form:

```
?s > 2 || ?v > 2
```

where ?v is a time series value and ?s is contained in the SPARQL database, cannot be rewritten to ?s > 2, since we may lose solutions from the SPARQL database. When rewriting conditions, we initially

```
...
FILTER(?site_label = "Wind Mountain"
    && ?wtur_label in ("A1", "A2", "A3", "A4",
        "A5", "A6", "A7", "A8", "A9", "A10")
    && ?t >= "2022-08-30T08:40:00"^^xsd:dateTime
    && ?t <= "2022-08-30T21:40:00"^^xsd:dateTime) .
...
```

**Fig. 10.** Filter expression from the running example query.

```
...
FILTER(?site_label = "Wind Mountain"
    && ?wtur_label in ("A1", "A2", "A3", "A4",
        "A5", "A6", "A7", "A8", "A9", "A10")) .
...
```

**Fig. 11.** Rewritten filter expression from the running example query.

```
...
WHERE {
...
}
GROUP BY ?site_label ?wtur_label
    ?year ?month ?day ?hour ?minute_10
```

**Fig. 12.** Group by graph pattern in the running example query.

require that the expression is unchanged or in case it is a logical expression, weaker. Processing the negation operator flips this requirement, and we instead require that the negated expression is unchanged or stronger. For instance, we are permitted to rewrite the expression:

```
NOT(?s > 2 || ?v > 2)
```

into

```
NOT(?s > 2)
```

Composite expressions that are not logical operators generally must have unchanged arguments. If any of the parts of such composite expressions are changed, we cannot ask the SPARQL database to evaluate it, and must do it in a later stage. In the running example, we have a filter graph pattern where the condition is given in Fig. 10. The filter graph pattern is rewritten in Fig. 11. The rewrite in Fig. 11 is permitted since the condition now has become weaker.

### 5.4.4. Group by graph pattern rewriting

Group by graph patterns are introduced to the SPARQL algebra by the GROUP BY keywords. The group by is followed by the variables one should group by, and requires that aggregation expressions such as sum are applied other variables occur after the SELECT keyword, much like in SQL. The group by graph pattern consists of an inner graph pattern, variables to group by and aggregation expressions bound to variables. During a rewrite, the group by graph pattern can potentially lose information contained in its inner graph pattern, as it groups by certain variables and performs aggregation operations which are in general lossy. Any non-trivial rewrite of the contained graph pattern of a group by operation will cause the group by graph pattern to be dropped. In the case of the running example, we have a group by graph pattern shown in Fig. 12.

We note that, in previous sections, we detailed several non-trivial rewrites of the inner graph patterns contained in the WHERE-clause. We must therefore drop the group by graph pattern and instead only preserve the inner graph pattern when rewriting. In general, we must create a sub query, when we encounter a group by pattern where the inner graph pattern is non-trivially rewritten.

```
...
SELECT ?site_label ?wtur_label
       ?ts_external_id_0 ?ts_datatype_0
...
```

**Fig. 13.** Projection graph pattern from the running example query.

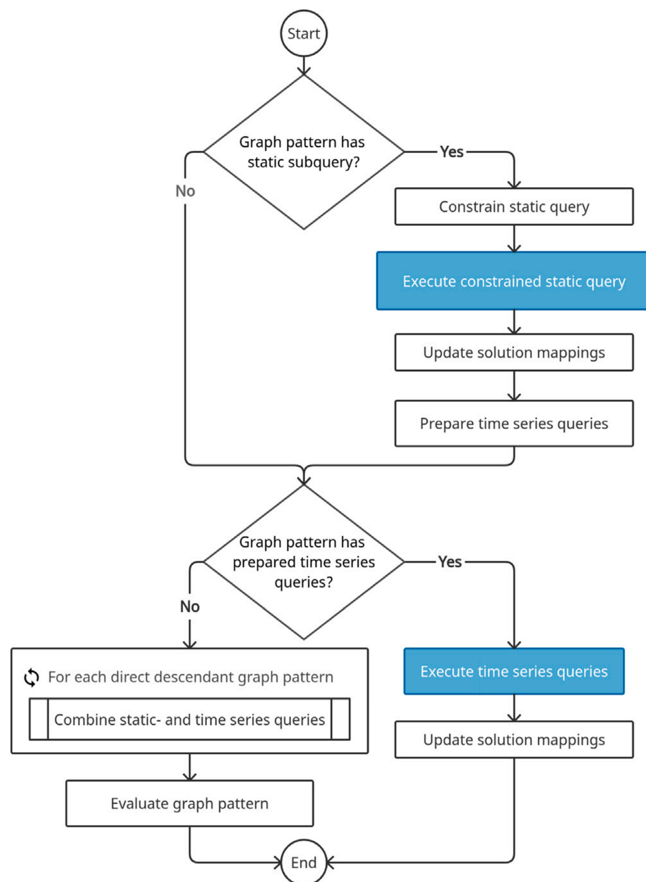



**Fig. 14.** Detailed view of combining static queries and time series queries.

#### 5.4.5. Project graph pattern rewriting

The project graph pattern is introduced by the SELECT keyword, which is followed by a set of variables. Following the WHERE-keyword is the inner graph pattern associated with the project graph pattern. Any expression-defined variables are encoded in the SPARQL algebra as aggregation expressions of an inner group by graph pattern, or as Extend graph patterns nested inside the inner graph pattern.

When a subquery is created, we must create a projection pattern for it to be executable in the static SPARQL endpoint. We should project all variables that are necessary to process the entire query, e.g. variables that occur in expressions or in the top level projection. Additionally, if the subquery has been rewritten by our procedure, e.g. a filter containing an expression not fully evaluated by the static SPARQL endpoint, we may have to project additional variables that will be used in the combination phase to complete the evaluation of the subquery. It is also important to project any variables that we introduced in the rewrite, as these contain important metadata for querying and combining the results from the time series database with the results from the static SPARQL endpoint.

In the case of the running example, following the principles described above, a sub query with the projection given in Fig. 13 is created.

**Table 1**
Excerpt of solution mappings associated with the static query in the running example.

| site_label | wtur_label | ts_external_id_0 | ts_datatype_0 |
|---|---|---|---|
| Wind M. | A1 | ep1 | xsd:double |
| Wind M. | A2 | ep2 | xsd:double |
| ... | ... | ... | ... |
| Wind M. | A10 | ep10 | xsd:double |

**Table 2**
Excerpt of solution mappings after grouping pushdown modification.

| site_label | wtur_label | Group |
|---|---|---|
| Wind M. | A1 | 1 |
| Wind M. | A2 | 2 |
| ... | ... | ... |
| Wind M. | A10 | 10 |

### 5.5. Combining static and time series queries

Recall that after rewriting the static query, we have one or more static subqueries, indexed by the path where they occur in the original query. Additionally, we have zero or more BTSQs. These BTSQs are always associated with a given data point-variable inside a basic graph pattern with a particular path. Fig. 14 zooms in on the process of combining static queries and time series queries.

Initially, the process starts with an empty multiset of solution mappings, the map of static subqueries and the basic time series queries. The combination process visits the nodes in the algebraic structure of the SPARQL query in a depth first way. The procedure checks when arriving at a graph pattern (as part of another graph pattern or nested within an exists expression), if there is an associated static (sub)query. If there is no static subquery associated with our graph pattern, we recursively evaluate our procedure on the constituent graph patterns of the present graph pattern. If there is a static sub query, it is amended using the multiset of solution mappings to bind the overlapping variables that are in scope appropriately. We use the values-graph pattern to accomplish this. Next, the amended static query is executed, and we join the results with the existing solution mappings. At this point there should be no more static queries associated with any deeper path in the tree. In the running example, we recursively process the outermost projection pattern and reach the group by pattern. We note that there is a static query associated with this path. There is no existing solution mapping, so we do not need to constrain the static query. For the running example, the results from the static query are given in Table 1.

Having received the static query solution mappings, we are ready to prepare the time series queries for the present sub-tree of the query. The preparation step is discussed in detail in Section 5.6. The preparation step also traverses the present sub-tree until it reaches a basic graph pattern. We complete the time series query with the associated external identifier information found in the solution mappings. It constructs time series queries, where we push down as much of the nested graph patterns as possible. From the preparation step, we receive a map of time series queries, where the keys are paths to the highest graph pattern pushed down into the time series query. In the running example, time series query preparation produces a map which has one entry, a time series query associated with the path of the group by graph pattern. This time series query is shown in Fig. 15. Preparing the time series query and discovering that we are able to push down the group by graph pattern to the time series database, we also must update the solution mapping with a grouping variable, since the grouping may not be 1:1 with the external identifiers. The updated solution mappings are given in Table 2.

Next, we check if the present path has an associated time series query. If there is no static subquery associated with our graph pattern, we recursively evaluate our procedure on the constituent graph patterns of the present graph pattern. If there is an associated time series

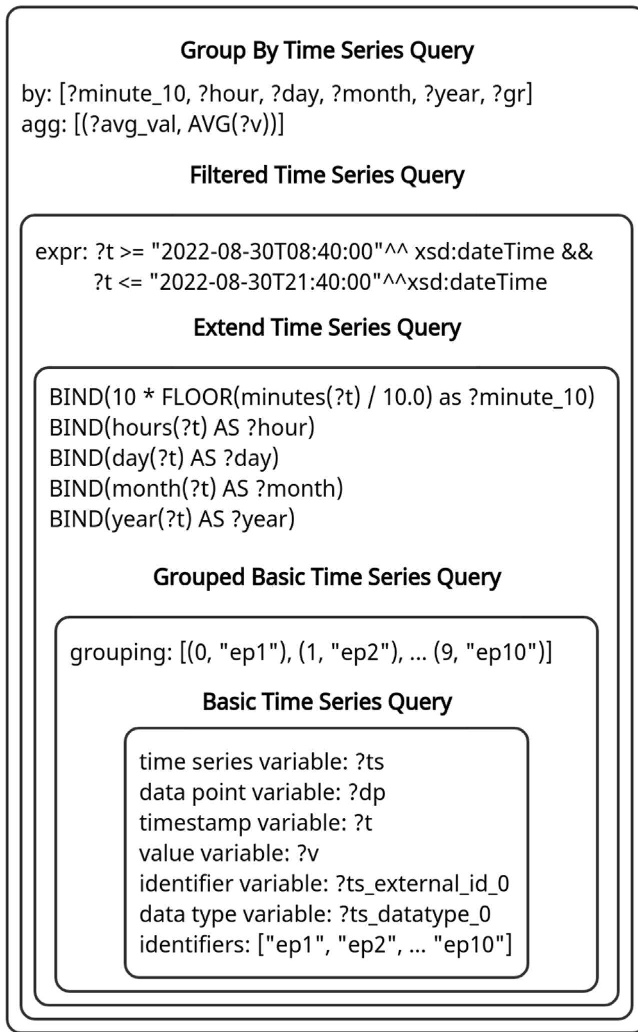



**Fig. 15.** Grouped Time Series Query corresponding to the running example query.

```
? t>= "2022-08-30T08:40:00"^^xsd:dateTime
&& ?t <= "2022-08-30T21:40:00"^^xsd:dateTime
```

**Fig. 16.** The part of the filter which is pushed down into the time series query.

query, the time series query is executed and we join the results with the solution mappings using the external identifier or the grouping column (see Section 5.6) if we have pushed down a group by graph pattern. At this point, there should be no time series query associated with a deeper path in the tree, and we do not traverse further down. In the running example, there is now a time series query associated with the group by graph pattern. In the benchmarks, we use a SQL-backend, and so the time series query will be translated to SQL before it is executed. The results of executing the time series query in our example is given in Table 3. We join these results to the existing solution mapping from the static query and grouping column. If on the other hand the time series queries are associated with deeper nodes in the tree, we recursively evaluate our procedure on the constituent graph patterns. If there is no static subquery associated with our graph pattern, we recursively evaluate our procedure on the constituent graph patterns of the present graph pattern.

Having reached the bottom of the tree or termination through evaluating every time series query associated with a graph pattern, we make our way up the tree, evaluating every graph pattern on the solution mappings returned from processing the constituent graph patterns. The details of this evaluation depend on the specific graph patterns. For the running example, there are no more time series queries after having executed and joined the time series query at the group by pattern. We exit the recursive call associated with the group by graph pattern, returning the solution mappings. We can now evaluate the projection pattern, which produces no change, as we already have exactly the variables we need. The result is identical to that of Table 4.

**Table 3**
Excerpt of time series query results.

| Group | Year | ... | sum_v |
|---|---|---|---|
| 1 | 2022 | ... | 9.15E6 |
| 2 | 2022 | ... | 9.16E6 |
| ... | ... | ... | ... |
| 10 | 2022 | ... | 9.14E6 |

**Table 4**
Solution mappings after having joined time series solution results to the static solution mappings.

| site_label | wtur_label | Year | ... | sum_v |
|---|---|---|---|---|
| Wind M. | A1 | 2022 | ... | 9.15E6 |
| Wind M. | A2 | 2022 | ... | 9.16E6 |
| ... | ... | ... | ... | ... |
| Wind M. | A10 | 2022 | ... | 9.14E6 |

### 5.6. Time series query preparation

We further prepare the time series queries, once the static query results are ready. The job of the TSQ preparation stage is to determine which external identifiers the TSQ should query, the data type of the time series value and to push down as much processing as possible into the time series database. This functionality is important in order to support the fourth requirement (R4) on query portability and the fifth requirement (R5) on offloading computation to the time series database. Since we projected both the external identifiers and the data type of the time series value in the static query, we can gather these from the static query results. The external identifiers of interest and data type are added to the basic time series queries.

In order to push down processing into the time series database, we again traverse the SPARQL algebraic structure corresponding to the original query, starting from the graph pattern where we found an associated static query.

There are four main pushdown types relevant to the benchmark cases in the article.

- Filter pushdowns
- Extend pushdowns
- Synchronisation pushdowns, that push down multiple time series queries with related timestamp variables
- Group by pushdowns

We discuss each kind of pushdown in turn below.

Filter conditions may be pushed down into the time series query, such as those constraining the timestamp value by some literal or constraining the value variable to be greater than some literal value. Recall that in Section 5.4.3, we described how expressions in filters are weakened, when these are rewritten. An analogous process happens to expressions in filters that are pushed into the time series query. In the running example, we will only push down the constraints on the timestamp variable `?t` into the time series database, as seen in Fig. 16.

We push down extend constructions into the time series query if the database supports expression pushdowns. This broad criteria has the weakness that it lets through expression types and variable types that are not actually supported by the underlying database. For OPC UA HA, it is only possible to push down those extend pushdowns that are involved in constructing the periods on which to aggregate the

```
...
SELECT ?w ?datetime_seconds (SUM(?v) as ?sum_v)
WHERE {
...
?dp ct:hasTimestamp ?t .
?dp ct:hasValue ?v .
FILTER(?v > 100 &&
?t > "2022-06-01T08:46:53"^^xsd:dateTime)
} GROUP BY ?w ?datetime_seconds
```

**Fig. 17.** Group by pushdown into OPC UA is impossible since there is a condition on the value variable.

time series. In the future, we plan to add support for specifying extend pushdown support with better granularity, perhaps by a pushdown validation function associated with the time series database interface implementation. In the running example, we have no problems pushing down the extend patterns, which we already listed in Fig. 9. We are also able to translate these expressions into SQL.

A use case we are targeting with this solution is extracting related time series values that are synchronised in time. One way in which this situation occurs is if multiple basic time series queries share the same timestamp variable within a basic graph pattern. If the time series database supports joins, we detect this situation and create a synchronised time series query. In practice, this only happens when the time series database is SQL-based. The time series variables in question may be identical, but situations also exist where we are interested in two related time series values where the timestamps of one series is shifted by some fixed duration. This case is currently not supported, and we will get back to it in our discussion of limitations in Section 9.4.

In order for a group by pushdown to be correct, it is important that the inner graph pattern is not in any way relaxed. In general, if the time series database supports group by pushdowns, we attempt to push down any group by pattern. The attempt fails if any part of the inner graph pattern cannot be pushed down. For instance, if the inner graph pattern contains a filter on a time series value, this part of the filter cannot be pushed down into a OPC UA HA API. The result is that the group by itself cannot be pushed down. Such a situation where a group by pushdown is impossible is shown in Fig. 17. Additionally, we must also be able to push down any aggregation function into the time series database.

In the running example, we are able to push down all filters and all extend patterns. When pushing down the group by graph pattern however, we must be careful to perform the grouping correctly. In particular, the grouping is not necessarily done by each external id. Instead, we associate each unique row of variables to be grouped by with an integer. Next, we extend each involved basic time series query with pairs of external identifiers and grouping identifiers. In the time series database, grouping must be done based on this grouping identifier.

## 6. Proof of correctness

In this section, we show that our solution approach preserves SPARQL 1.1 correctness for a large subset of SPARQL 1.1. Showing this, we are able to ensure that we meet the first requirement (R1) to a large degree. We will consider only RDF graphs without blank nodes. We omit blank nodes since this simplifies the proof, and since the RDS based knowledge bases discussed here contain no blank nodes. Additionally, we will only consider the case where a basic graph pattern contains one time series variable. This is done since assuming otherwise makes the presentation of the combination procedure considerably more complicated and verbose. A further simplification is that we do not consider the concept of active graphs, which arises when the GRAPH-keyword is used, and instead assume that a single graph is active at all times. While the solution does support multiple time series variables per Basic Graph Pattern (BGP), we do not currently support blank nodes nor a concept of active graphs. Furthermore, we assume that the graphs on which our queries are evaluated contain no duplicate triples. We begin by defining some basic terms.

**Definition 6.1** (*Graph*)**.** A graph $G$ is a finite set of triples of the form $(\mathtt{s}\ \mathtt{p}\ \mathtt{o})$ where $\mathtt{s}$ (the subject) and $\mathtt{p}$ (the predicate) are IRIs and $\mathtt{o}$ (the object) is either an IRI or a literal. We let $G_N$ be the set of all IRIs and literals occuring in the triples of the graph.

Since a graph is a set, a subgraph $G'$ of $G$ is simply a subset of $G$, i.e. $G' \subseteq G$.

**Definition 6.2** (*Context Graph*)**.** A context graph $G_{ctx}$ is a graph where for all triples $(\mathtt{s}\ \mathtt{p}\ \mathtt{o})$, $\mathtt{p}$ is *not* one of:

$\mathtt{ct:hasTimeseries}$

$\mathtt{ct:hasDatatype}$

$\mathtt{ct:hasExternalId}$

$\mathtt{ct:hasDatapoint}$

$\mathtt{ct:value}$

$\mathtt{ct:timestamp}$

where the prefix ct refers to the namespace of chrontext, which we set to the GitHub repository URL.

**Definition 6.3** (*Annotated Context Graph*)**.** Given a static context graph $G_{ctx}$, an annotated context graph $G_{anno}$ is an extension of $G_{ctx}$ with N > 0 sets of triples of the form:

$(\mathtt{n}\ \mathtt{ct:hasTimeseries}\ \mathtt{s})$

$(\mathtt{s}\ \mathtt{rdfs:label}\ \mathtt{l})$

$(\mathtt{s}\ \mathtt{ct:hasExternalId}\ \mathtt{e})$

$(\mathtt{s}\ \mathtt{ct:hasDatatype}\ \mathtt{d})$

$G_{anno}$ is subject to the constraint that every timeseries has exactly one label, one external id and one data type.

$$\begin{aligned}
&\forall n, s \in G_{anno\,N} : (n\ \mathtt{ct:hasTimeseries}\ s) \in G_{anno} \Rightarrow \\
&\quad (\exists! l : (\mathtt{s}\ \mathtt{rdfs:label}\ l) \in G_{anno} \\
&\quad\ \wedge \exists! e : (s\ \mathtt{ct:hasExternalId}\ e) \in G_{anno} \\
&\quad\ \wedge \exists! d : (s\ \mathtt{ct:hasDatatype}\ d) \in G_{anno})
\end{aligned}$$

**Definition 6.4** (*Time Series Database*)**.** Given a set of data types $D$, a datatype-indexed family of identifier sets $I_{d\in D}$, a datatype-indexed family of value sets $V_{d\in D}$, a time series database is a family of functions $f_{d\in D} : I_d \rightarrow (\mathbb{N}, V_d)$. The set of natural numbers here represents a set of timestamps.

**Definition 6.5** (*Implied Time Series Graph*)**.** Given an annotated context graph $G_{anno}$, and a time series database $f_{d\in D}$ the implied time series graph $G_{impl}$ is an extension of $G_{ctx}$ with the minimal set of triples that make the statements below hold:

$$\begin{aligned}
&\forall n, s, d, e \in G_{anno} : \\
&\quad ((n\ \mathtt{ct:hasTimeseries}\ s) \in G_{anno} \\
&\quad \wedge (s\ \mathtt{ct:hasDatatype}\ d) \in G_{anno} \\
&\quad \wedge (s\ \mathtt{ct:hasExternalId}\ e) \in G_{anno}) \\
&\quad \Rightarrow \\
&\quad ((n\ \mathtt{ct:hasTimeseries}\ s) \in G_{impl} \\
&\quad \wedge (f_d(e) = (t, v) \Rightarrow \\
&\qquad ((s\ \mathtt{ct:hasDatapoint}\ p_{s,t,v}) \in G_{impl}
\end{aligned}$$

$\wedge (p_{s,t,v}$ `ct : value` $v) \in G_{impl}$

$\wedge (p_{s,t,v}$ `ct : timestamp` $t) \in G_{impl}))$

where $p_{s,t,v}$ is a unique URI associated with the tuple $(t, v)$ and time series $s$. We define utility functions $value(p_{s,t,v}) = v$ and $timestamp(p_{s,t,v}) = t$.

A SPARQL select query can be translated automatically into a SPARQL algebra graph pattern (World Wide Web Consortium, 2013c) We denote the set of variables occurring in the graph pattern $P$ by $P_V$. The definition of solution mappings, their compatibility and the merge operation are adapted from the SPARQL 1.1 W3C recommendation (World Wide Web Consortium, 2013c).

**Definition 6.6** (*Solution Mappings*)**.** Let $G$ be a graph and let $P$ be a graph pattern. A solution mapping is a partial function:

$$\mu : P_V \to G_N$$

Subject to criteria that are defined on the algebraic structure of a graph pattern. The solution mappings associated with a graph pattern $P$ and a graph $G$ are denoted by $\Omega(P, G)$, and is a multiset of solution mappings. Multisets are just like sets, except each member can occur multiple times. The number of times each member occurs is the cardinality of the member in the multiset. The cardinality is denoted by $|m|$, where $m$ is a member of a multiset.

For basic graph patterns it is required that the graph induced by the solution mapping is a subgraph of $G$. We will discuss the criteria for other graph patterns in the sections that follow. Later, we will use the function *dom* to refer to the domain of a solution mapping $\mu$, e.g. $dom(\mu) = P_V$ in the definition above.

**Definition 6.7** (*Compatible Solution Mappings*)**.** Given graphs $G, H$ and graph patterns $P, Q$ with solution mappings:

$$\mu_P : P_V \to G_N$$

$$\mu_Q : Q_V \to H_N$$

$\mu_P$ and $\mu_Q$ are compatible if:

$$\forall x \in P_V \cap Q_V :$$
$$\mu_P(x), \mu_Q(x) \text{ defined } \Rightarrow \mu_P(x) = \mu_Q(x)$$

**Definition 6.8** (*Merge*)**.** Given graphs $G, H$ and a graph patterns $P, Q$ with compatible solution mappings $\mu_P$ and $\mu_Q$, their merge $merge(\mu_P, \mu_Q) : P_V \cup Q_V \to G_N \cup H_N$ is given by:

$$merge(\mu_P, \mu_Q)(x) = \begin{cases} \mu_P(x) & \text{if } x \in P_V \wedge \mu_P(x) \text{ defined} \\ \mu_Q(x) & \text{if } x \in Q_V \wedge \mu_Q(x) \text{ defined} \\ undefined & otherwise \end{cases}$$

### 6.1. Intuitive motivation

Intuitively, given a correct implementation of SPARQL semantics, we can evaluate any query by evaluating its basic graph patterns. That is, we can for each basic graph pattern (BGP) in a query create a subquery that projects all of the involved variables. Producing correct results in this case is simply a matter of evaluating SPARQL 1.1 starting with correct solution mappings of any involved BGP. In our setting, even this is not possible, since the triples in a given BGP live in different databases. Going even further then, we can split every BGP into at most two BGPS depending on the database where the predicate belongs, projecting all variables. Next, we can evaluate the BGPs on the appropriate databases and join them to obtain the original BGP. Such an approach would produce correct results in our situation, but would be highly impractical. The core idea of our solution then, is that we attempt to evaluate larger parts of involved queries starting from these split BGPs in each database, and combine the results to obtain the result of the original query.

To show that our approach produces correct results, we need to prove that evaluating the original query on the implied graph gives the same result as our approach. However, following the argument in the above paragraph, it is enough to show identical solution mappings for the subqueries, and assume that we have a perfect SPARQL 1.1 implementation which combines these multisets of solution mappings into exactly those of the original query.

### 6.2. Structure of the proof

Let $P$ be any graph pattern which is associated with a static subquery in our approach. Let $r(P)$ be the associated static rewrite. We must prove that the solution mappings $\Omega(P, G_{impl})$ are identical to the solution mappings obtained by evaluating $r(P)$ and any associated time series queries according to our solution approach. In order to describe the structure of the proof, we need to define the crucial compatibility relation $C$.

**Definition 6.9** (*Relation C*)**.** Let $\Omega_1, \Omega_2$ be multisets of solution mappings. The relation $C(\Omega_1, \Omega_2)$ holds if and only if:

$$\forall \mu_1 \in \Omega_1 : \exists \mu_2 \in \Omega_2 :$$
$$\mu_1 \text{ is compatible with } \mu_2 \wedge$$
$$|\mu_1| = |\mu_2|$$

We note that $C$ is reflexive by the reflexivity of = and compatibility. In the proof, we will let the graphs $G_{anno}$, $G_{impl}$ be fixed, but arbitrary. We will use the shorthand $G_{comb}$ to denote the union of these graphs, i.e. $G_{comb} = G_{anno} \cup G_{impl}$. We are now ready to describe the steps in the proof. We let $P$ be the graph pattern corresponding to a subquery.

1. We show that for every solution mapping $\mu_{impl} \in \Omega(P, G_{impl})$ obtained when evaluating a graph pattern $P$ on the implied graph $G_{impl}$, there exists a compatible solution mapping $\mu_{anno} \in \Omega(r(P), G_{anno})$ associated with the static rewrite $r(P)$. We will thus have $C(\Omega(P, G_{impl}), \Omega(r(P), G_{anno}))$.
2. We show that when the combination procedure finishes processing $P$ and produces a new multiset of solution mappings, denoted by $\Gamma(P, G_{comb})$, it has the following properties:
    - We have not lost any solutions, and so the property $C(\Omega(P, G_{impl}), \Gamma(P, G_{comb}))$ holds.
    - We have not lost any variables in the solution mappings of $\Gamma(P, G_{comb})$.
    - Every solution mapping is a real solution, i.e., $C(\Gamma(P, G_{comb}), \Omega(P, G_{impl}))$.

Taken together, we obtain exactly the same multiset of solution mappings as required. The first step is completed in Section 6.3. The second step is completed in Section 6.4.

### 6.3. Query rewriting

This section proves the first step in the proof. Before we begin this step of the proof, we discuss an excerpt of the code under discussion in Section 6.3.1. This is done both to elucidate the structure of the programme, and in order explain why the proof of the preservation of the relation $C$ only needs to consider certain outcomes of rewriting. In Sections 6.3.2 to 6.3.11, we prove the relation $C$ for each type of graph pattern.

```
def rewrite(gp:GP, path:Path,
  btsqs:Dict[Path, BTSQ],
  subqs:Dict[Path, Query]) -> Option[GP]:
  match gp:
    case BGP(ts):
      # Creates the triples extracting metadata
      new_ts = create_add_btsq(ts, path, btsqs)
      # Filters out triples from timeseries db
      filtered_ts = filter_ts(ts)
      return BGP(filtered_ts+new_ts)

    case Filter(i, e):
      i_path = path.extend(FilterInner)
      i_rw = rewrite(i_rw, i_path, btsqs, subqs)
      e_path = path.extend(FilterExpr)
      e_rw = rewrite_expr(e, i_rw, e_path, subqs)
      if i_rw is None: # inner rewrite is subquery
        return None
      if e_rw is None: # expr rewrite is subquery
        subqs[i_path] = create_subquery(i_rw)
        return None
      return Filter(i_rw, e_rw)
    ...

def create_add_btsq(ts:List[TP], path:Path,
    bstqs: Dict[Path,BTSQ]) -> List[TP]:
  # For each triple
  for TP(s,p,o) in ts:
    if p == HAS_DATATYPE:
      (btsq, new_ts) = create_btsq(o, ts)
      btsqs[path] = btsq
      return new_ts
  return []
```

**Fig. 18.** Simplified excerpt of rewriting procedure in Python-syntax.

### 6.3.1. The rewriting procedure and proof

An excerpt of the rewriting procedure (presented in Python for simplicity), is given in Fig. 18. The rewriting procedure is called using the query (a project graph pattern), with an empty path and with empty dictionaries of basic time series queries and subqueries which will be altered during processing. Paths are built as we traverse the tree in a post-order sequence, and are used to index created subqueries and basic time series queries. If rewriting produces a rewrite for the project graph pattern which is not `None`, we add the rewritten project graph pattern on the empty-path in the subqueries-dict (`subqs`). When the rewriting procedure returns a graph pattern we will say that the rewrite of the graph pattern exists. Otherwise (returns `None`) the procedure has created one or more subqueries when processing the graph pattern. Our proof will consider each matching rule in the procedure, and assume the induction hypothesis that $C$ holds between the original constituent and rewritten constituent graph pattern for each constituent graph pattern. Next, we prove that the rewriting rule is such that $C$ now holds between the original matching graph pattern and the rewritten graph pattern. In our proof, we will be able to assume that no rewrite of the constituent graph patterns or expressions produced a subquery.

The processing of the non-terminal filter graph pattern is representative for other non-terminals. The filter graph pattern processes the inner graph pattern first, and then processes the condition expression in light of the rewritten inner graph pattern. The `rewrite_expr` function returns the rewritten expression `e_rw` – which is `None` case if there are exists-expressions in the filter. If the inner rewrite is a subquery, we return `None`. If the expression contained one or more exists-expressions, the rewrite will have created a subquery, and so the inner graph pattern too must be made a subquery if it exists. This is a general pattern in our rewriting procedure. If rewriting a constituent graph pattern or expression produces a subquery, then all constituent graph patterns must be made into subqueries. Consequently we only need to prove the relation $C$ for graph patterns $P$ where for every constituent graph pattern $P'$ there exists a rewritten graph pattern $r(P')$.

It is also the case that if the rewrite of a graph pattern is trivial, producing no change, the relation $C$ holds by reflexivity. When rewriting any graph pattern $P$, if all constituent graph patterns are trivially rewritten, the rewrite of $P$ is also just $P$, so $C$ holds. We consider only non-trivial rewrites in the discussion below, as these lead us to have to prove that $C$ is preserved.

### 6.3.2. Basic graph pattern rewriting

The Basic Graph Pattern (BGP) does not consist of other graph patterns, and forms a basis condition for the proof. As mentioned, the solution mapping $\mu$ of a BGP $B$ forms a subgraph of $G$. We can express this as $\mu(B) \subseteq G$, where $\mu(B)$ is the set of triples resulting from replacing each variable $v$ occurring in a triple of $B$ with $\mu(v)$.

We let $B$ be a Basic Graph Pattern. We let $r(B)$ be the rewritten BGP for the annotated context graph $G_{anno}$. If no triples are removed, the relation $C$ holds by reflexivity. Assuming instead that a unique `(?node ct:hasTimeseries ?s)` exists in the BGP, such a set of related triple patterns $R$ to be removed will take the following form:

$$(?\mathtt{s}\ \mathtt{ct:hasDatapoint}\ ?\mathtt{p}) \tag{1}$$

$$(?\mathtt{p}\ \mathtt{ct:value}\ ?\mathtt{v}) \tag{2}$$

$$(?\mathtt{p}\ \mathtt{ct:timestamp}\ ?\mathtt{t}) \tag{3}$$

Triple 1 will always exists in the set, and at least one of 2 and 3 exist, otherwise we reject the query, as it is not supported. Let the deletion of these triple patterns from $B$ be known as $B_{anno}$, since this graph pattern now only involves the annotated context graph $G_{anno}$. If there are any other triple patterns in $B$ involving the terms `ct:hasDatapoint`, `ct:value` or `ct:timestamp` we declare the query void and abort. We now add the triples below:

$$(?\mathtt{s}\ \mathtt{ct:hasDataType}\ ?\mathtt{d})$$

$$(?\mathtt{s}\ \mathtt{ct:hasExternalId}\ ?\mathtt{e})$$

By construction, we know that in any case where there is a time series `?s`, there exist metadata making the above triples hold. After adding these triples to $B_{anno}$ we obtain $r(B)$. We make sure to name the variables `?d` and `?e` so that no other variable in the query has the same name, but use these single letter names here for simplicity.

Let $\mu : B_V \rightarrow G_{impl}$ be a solution mapping. We observe that there exists a restriction of $\mu$ to $\mu_{anno} : B_{anno\,V} \rightarrow G_{anno\,N}$. By construction of $G_{impl}$, this mapping will be a solution mapping for $B_{anno}$. I.e. if $\mu(B) \subseteq G_{impl}$ then since we drop every triple pattern $T \in B$ such that $\mu(T) \in G_{impl} \setminus G_{anno}$, then $\mu_{anno}(B_{anno}) \subseteq G_{anno}$.

There will in general be $N$ solution mappings $\mu_1, \mu_2, ..\mu_N$ having the same restriction $\mu_{anno}$, which we will recover in the combination phase. There exists an assignment $\mu_{anno}(?\mathtt{t}) = \mathtt{t}$. By construction there exists exactly one $\mathtt{t_dt} \in G_{anno}$ and exactly one $\mathtt{t_eid} \in G_{anno}$ such that:

$$(\mathtt{t}\ \mathtt{ct:hasDatatype}\ \mathtt{t_dt}) \in G_{anno} \wedge$$

$$(\mathtt{t}\ \mathtt{ct:hasExternalId}\ \mathtt{t_eid}) \in G_{anno}$$

We extend $\mu_{anno}$ to $r(B)$ by the assignments:

$$?\mathtt{d} \mapsto \mathtt{t_dt}$$

$$?\mathtt{e} \mapsto \mathtt{t_eid}$$

The new mapping $\mu_{r(B)} : r(B)_V \rightarrow G_{anno\,N}$ is a solution mapping of $r(B)$ over $G_{anno}$ whenever $\mu_{anno}$ is a solution mapping over $G_{anno}$

and by extension whenever there exists a solution mapping $\mu$ that restricts to $\mu_{anno}$. Moreover $\mu$ is compatible with $\mu_{anno}$. Since we made no assumption about $\mu$ when constructing $\mu_{anno}$, we conclude that the relation $C(\Omega(B, G_{impl}), \Omega(r(B), G_{anno}))$ holds. We do not have to consider cardinalities as they are always 1 for BGPs.

#### 6.3.3. Filter rewriting

Syntactically, a filter $\text{Filter}(I, expr)$ is composed of an inner graph pattern $I$ and an expression $expr$. We recall the discussion at the start of Section 6.3. I.e. if the rewritten inner graph pattern $I$ results in one or more subqueries, we do not include the filter in these subqueries, and leave filter evaluation to the combination stage. Additionally, $expr$ may contain one or more exists-expressions, which themselves contain graph patterns. Exists expressions always lead us to create one or more static subqueries, and so we must also create a sub-query of the inner graph pattern $I$ and postpone evaluation of the filter in this case.

We consider the case where neither rewriting $I$ nor rewriting $expr$ leads to the creation of a static subquery, but rewriting $I$ leads to a non-trivial rewrite $r(I)$. The multiset of solution mappings resulting from the evaluation of the filter has the following definition:

$$\Omega(\text{Filter}(I, expr), G) = \{\mu | \mu \in \Omega(I, G) \wedge expr(mu, G) = \text{true}\}$$

We assume that the induction hypothesis that:

$$C(\Omega(I, G_{impl}), \Omega(r(I), G_{anno}))$$

For filters, our proof depends on the notion that rewritten expressions only depend on the actual contents of the graph $G_{anno}$, which is guaranteed in the case where the expression contains no exists-expression.

If the rewrite of $expr$ does not exist, we simply let $\Omega(r(\text{Filter}(I, expr)), G_{anno}) = \Omega(\text{Filter}(r(I), \texttt{true}), G_{anno})$, which is a special case of the argument below. If the rewrite of $expr$ exists, then $r(expr)$ may only be weaker than- or equivalent to $expr$. Given

$$\mu \in \Omega(I, G_{impl})$$

and

$$r(expr)(\mu, G_{impl}) = \texttt{true}$$

we know that

$$r(expr)(\mu_{anno}, G_{anno}) = \texttt{true}$$

for any $\mu_{anno} \in \Omega(r(I), G_{anno})$ where $\mu_{anno}$ and $\mu$ are compatible, since these agree on all variables in $r(expr)$. Filters do not modify cardinalities. It follows that as required:

$$C(\Omega(Filter(I, expr), G_{impl}), \Omega(\text{Filter}(r(I), r(expr)), G_{anno}))$$

Note that the loss of- or weakening of the expression $expr$ can cause there to be $\mu_{anno}$ for $I_{anno}$ that has no compatible counterpart for $I$ due to the weakening of $expr$. We will correct this situation in the combination phase.

#### 6.3.4. Join rewriting

Let $L, R$ be the left hand side and right hand side graph patterns of a join graph pattern $\text{Join}(L, R)$ where rewrites of $L$ and $R$ exist and are not subqueries. We assume the induction hypothesis that:

$$C(\Omega(L, G_{impl}), \Omega(r(L), G_{anno})) \wedge C(\Omega(R, G_{impl}), \Omega(r(R), G_{anno}))$$

The semantic interpretation of the join is reproduced below:

$$\Omega(\text{Join}(L, R), G) = \{\text{merge}(\mu_L, \mu_R) | \mu_L \in \Omega(L, G) \wedge \mu_R \in \Omega(R, G) : \mu_L \text{ compatible with } \mu_R\}$$

Cardinalities are given by the product of the respective cardinalities of $\mu_L$ and $\mu_R$.

$\mu_{r(L)} \in \Omega(r(L), G_{anno})$ and $\mu_{r(R)} \in \Omega(r(R), G_{anno})$ are compatible whenever $\mu_L \in \Omega(L, G_{impl})$ and $\mu_R \in \Omega(L, G_{impl})$ are, since any variables introduced in a BGP in $r(L)$ or $r(R)$ are disjoint. This means that:

$$\text{merge}(\mu_L, \mu_R) \in \Omega(\text{Join}(L, R), G_{impl}) \Rightarrow \text{merge}(\mu_{r(L)}, \mu_{r(R)}) \in \Omega(\text{Join}(r(L), r(R)), G_{anno})$$

It follows from the induction hypothesis that:

$$|\mu_L| = |\mu_{r(L)}| \wedge |\mu_R| = |\mu_{r(R)}|$$

Hence:

$$|\mu_L| * |\mu_R| = |\mu_{r(L)}| * |\mu_{r(R)}|$$

It follows that the cardinality of $\text{merge}(\mu_{r(L)}, \mu_{r(R)})$ equals the cardinality of $\text{merge}(\mu_L, \mu_R)$. Now, we must show that

$$\text{merge}(\mu_L, \mu_R) \text{ compatible with } \text{merge}(\mu_{r(L)}, \mu_{r(R)})$$

In the case where $x \in L_V \cap r(L)_V$, then by compatibility of $\mu_L$ with $\mu_r(L)$, it follows that $\mu_L(x) = \mu_{r(L)}(x)$, and consequently that:

$$\text{merge}(\mu_L, \mu_R)(x) = \text{merge}(\mu_{r(L)}, \mu_{r(R)})(x)$$

If $x \in L_V$, $x \notin r(L)_V$ and $x \notin r(R)_V$ we have that compatibility holds trivially since $x$ is not in the domain of $\text{merge}(\mu_{r(L)}, \mu_{r(R)})$. We consider instead $x$ such that $x \in L_V$ but for which $x \notin r(L)_V$ and $x \in r(R)_V$. But for such a variable it must hold that $x \in R_V$, since we never introduce variables that overlap with existing variable names in the rewrite. It follows that $\mu_L$ and $\mu_R$ agree on $x$, and by compatibility $\mu_R(x) = \mu_{r(R)}(x)$. By extension:

$$\text{merge}(\mu_L, \mu_R)(x) = \text{merge}(\mu_{r(L)}, \mu_{r(R)})(x)$$

Symmetric arguments can be made for $x \in R_V$. Hence, we have proved that as required:

$$C(\Omega(\text{Join}(L, R), G_{impl}), \Omega(\text{Join}(r(L), r(R)), G_{anno}))$$

#### 6.3.5. Left join rewriting

The semantics of the left join is reproduced in a compact and simplified way below. "Diff" is an auxiliary construction used in the SPARQL 1.1 language recommendation to define the semantics of left join (World Wide Web Consortium, 2013c).

$$\Omega(Diff(L, R), G) = \{\mu | \forall \mu' : \text{merge}(\mu, \mu') \notin \Omega(\text{Join}(L, R), G) \vee expr(\text{merge}(\mu, \mu')) = \texttt{false}\}$$

The semantics of LeftJoin are given by:

$$\Omega(LeftJoin(L, R), G) = \Omega(\text{Filter}(\text{Join}(L, R), expr), G) \cup \Omega(Diff(L, R), G)$$

Assume that $\mu_{impl} \in \Omega(\text{Diff}(L, R), G_{impl})$ with $\mu_{impl}$ compatible with some $\mu_{r(L)} \in \Omega(r(L), G_{anno})$. There can now exist $\mu_{r(R)} \in \Omega(r(L), G_{anno})$ compatible with $\mu_{r(L)}$ such that the merge exists and:

$$r(expr)(merge(\mu_{r(L)}, \mu_{r(R)})) = \texttt{true}$$

Since the expression can become weaker. In this case we have that:

$$merge(\mu_{r(L)}, \mu_{r(R)}) \in \Omega(\text{Filter}(\text{Join}(r(L), r(R)), r(expr)), G_{anno})$$

This situation poses a problem for $C$, since the cardinality of $merge(\mu_{r(L)}, \mu_{r(R)})$ is no longer equal to that of $\mu_{impl}$ if $|\mu_{r(R)}| > 1$. To ameliorate the situation we would have to try to recover the original cardinality in the combination phase. Trying to do so complicates matters excessively. For instance, to be guaranteed to be able to recover the cardinality, we could not permit the LeftJoin graph pattern to be included in another LeftJoin, as this would potentially erase the information required to recover the correct cardinality. For this reason, we only allow trivial rewrites of the LeftJoin.

*6.3.6. Union rewriting*

Let $\text{Union}(L, R)$ be a union graph pattern, and assume the induction hypothesis as above. The semantics of union are given by:

$$\Omega(\text{Union}(L, R), G) \\ = \Omega(L, G) \cup \Omega(R, G)$$

Cardinalities are preserved for solution mappings that are not present in both $\Omega(L, G)$ and $\Omega(R, G)$, and summed for elements present in both. The rewriting procedure could potentially produce for distinct $\mu_L \in \Omega(L, G_{impl})$ and $\mu_R \in \Omega(R, G_{impl})$ but identical $\mu_{r(L)} \in \Omega(r(L), G_{anno})$ and $\mu_{r(R)} \in \Omega(r(R), G_{anno})$ that are the only compatible solution mappings for $\mu_L$ and $\mu_R$ respectively. We would be unable to retain the correct cardinality. For this reason, we only permit trivial rewrites of Union. If $r(L) \neq L$ or $r(R) \neq R$ we will produce subqueries of and produce subqueries of $r(L)$ and $r(R)$ if they exist.

*6.3.7. Ordering and distinct rewriting*

Orderings are rewritten so as to exclude variables that are dropped in the rewrite, but reordering must generally speaking be redone during the combination phase, which we will rely on for correctness of this type of graph pattern. We only keep the distinct graph pattern $\text{Distinct}(I)$ if the rewritten contained graph pattern $r(I)$ has no changes (trivial rewrite) in which case the relation $C$ holds by reflexivity. Otherwise, we create a sub-query of the rewritten inner graph pattern $r(I)$ (if it exists).

*6.3.8. Minus rewriting*

Let $\text{Minus}(L, R)$ be a minus graph pattern and assume the induction hypothesis holds as before. A simplified variant of the semantic interpretation of the minus-pattern is given below.

$$\Omega(\text{Minus}(L, R), G) = \{\mu_L | \mu_L \in \Omega(L, G) : \\ \forall \mu_R \in \Omega(R, G) : \mu_L \text{ not compatible with } \mu_R\}$$

If the rewrite of the right hand side has changed $R$ in any way, we may lose solutions if we proceed with constructing a minus graph pattern from $r(L)$ and $r(R)$. Similarly, since $\mu_{r(L)} \in \Omega(r(L), G_{anno})$ may not be defined for every variable that $\mu_{r(L)} \in \Omega(L, G_{impl})$ is defined for, it may inadvertently be compatible with all $\mu_{r(R)} \in \Omega(r(R), G_{impl})$. Hence, for cases where $L$ or $R$ have non-trivial rewrites, we must create a subquery of $r(L)$ if it exists, and a subquery of $r(R)$ if it exists.

*6.3.9. Extend rewriting*

The extend graph pattern $\text{Extend}(I, V, expr)$ has semantics reproduced below:

$$\Omega(\text{Extend}(I, v, expr), G) = \\ \{\mu \cup \{v \mapsto expr(\mu)\} | \mu \in \Omega(I, G)\}$$

If $expr$ can be rewritten with no change, then we are certain that all relevant variables still exist in the domain of $\mu_{r(I)} \in \Omega(r(I), G_{anno})$. Moreover, if $\mu_I \in \Omega(I, G_{impl})$ is compatible with some $\mu_{r(I)}$ then $expr(\mu_{r(I)}) = expr(\mu_I)$ (provided the expression is deterministic) and so $\mu_{r(I)}(v) = \mu_I(v)$ which implies that the relation $C$ is preserved. If $r(expr)$ does not exist, or $r(expr) \neq expr$ then we create a subquery of $r(I)$.

*6.3.10. Group by rewriting*

Let $\text{GroupBy}(I, A, V)$ be a group by graph pattern, where $I$ is the inner graph patterns, $A$ is a set of aggregation expressions each bound to a variable and $V$ is a set of variables to group by. We may only keep the group by graph pattern if $r(I) = I$, $r(A) = A$, i.e. if the rewrite is trivial, in which case the relation $C$ holds by reflexivity. In any other case we create a subquery based on $r(I)$, but associate it with $\text{GroupBy}(I, A, V)$.

*6.3.11. Projection rewriting*

Let $\text{Project}(I, V)$ be a projection pattern with $I$ an inner graph pattern and $V$ a list of variables to project. If the rewrite of $I$ is non-trivial, we will create a subquery of $r(I)$. We surround all subqueries $P$ with trivial projections of $r(P)_V$, in order to execute them and extract all the necessary information from the triplestore. Such a projection does not change any cardinalities and leaves all solution mappings intact.

*6.4. Combination*

A simplification of the combination function is presented in Fig. 19. The function `combine` is called for each subquery $P$ with the solution mapping variable (`sm`) set to `None`. `subqs` and `btsqs` are the subqueries and the basic time series queries from the rewriting-function. `tsqs` will be initialised with prepared time series queries during processing by the `prepare`-function, and is initially called with `None`. When combination processing is done for the subqueries, we may use ordinary SPARQL 1.1 processing to algebraically construct the query results. The `submap(map, path)` utility-function restricts the `map` to only those keys occurring on an extension of the `path`, and is necessary to ensure that we reach a condition where both `subqs` and `tsqs` are empty and we may return `sm`. Time series preparation (`prepare`) attempts to push down as much of the graph pattern as possible into the time series database, conditional on the level of support the database provides, it is discussed in Section 6.4.1. Applying `combine` to the graph pattern $P$ and produces a result we denote $\Gamma(P, G_{comb})$. When starting processing of a subquery, $\texttt{sm} = \texttt{None}$ and we immediately reach the case where:

$$\texttt{sm} = \texttt{exec_subquery}(path)$$

We may denote this resulting `sm` by $\Omega(r(P), G_{anno})$. Two cases are possible in this case. If there is no time series query-part to the subquery $P$, in which case $r(P) = P$, and it follows immediately that:

$$\Omega(P, G_{impl}) = \Gamma(P, G_{comb})$$

and by the reflexivity of $C$ that

$$C(\Omega(P, G_{impl}), \Gamma(P, G_{comb})) \wedge \\ C(\Gamma(P, G_{comb}), \Omega(P, G_{impl}))$$

This forms the first basis condition for our inductive proof. Otherwise, there are one or more time-series queries associated with the subquery $P$. There may be a single time series query associated with a group by graph pattern (identical to $P$), or one or more time series queries associated with basic graph patterns contained in (or equal to) $P$. We will in any case arrive at `exec_tsq`, either directly in the present function call in case the group by has been pushed down, or through zero ($P$ may be a BGP) or more recursive calls to `combine` in case the time series query/queries are associated with one or more BGPs. We let $R$ be the value of `gp` passed to `combine` when we call `exec_tsq` for the first time. We show in Section 6.4.2 that the result of this call produces $\Gamma(R, G_{comb})$ with the properties:

$$C(\Omega(R, G_{impl}), \Gamma(P, G_{comb})) \wedge \\ C^*(\Gamma(P, G_{comb}), \Omega(R, G_{impl}))$$

This forms the second basis condition for our inductive proof. We must annotate the second $C$ with $^*$ to denote the fact that Join-patterns enclosing $R$ in $P$ can cause cardinalities to be too large by a factor. This problem is corrected after we have finished the combination procedure for the topmost Join in $R$, and so we write $C^*$ to mean that $C$ will hold eventually. We discuss this problem in detail in Section 6.4.2.

Additionally, we show that `exec_tsq` preserves these relations when `exec_tsq` is called for a basic graph pattern $B$ at a later time when executing `combine` for the subquery $P$. This forms the first part of the inductive step in our proof. We also show that the combination function

```
def combine(gp:GP, sm:Option[SM], path:Path,
  btsqs:Dict[Path, BTSQ],
  subqs:Dict[Path, Query],
  tsqs:Option[Dict[Path, TSQ]]) -> SM:

  if path in subqs: # Associated subquery
    # Execute static query
    sm = exec_subquery(subqs.pop(path))
    tsqs = prepare(gp, btsqs, sm, path, False)

  if sm is not None and path in tqs:
    # Execute tsq with constraints from sm
    # and join result with sm
    sm = exec_tsq(sm, tsqs.pop(path))

  if subqs.empty() and tsqs.empty():
    return sm

  match gp:
    case Filter(i, e):
      i_path = path.extend(FilterInner)
      sm = combine(i, i_path, sm,
                   btsqs, subqs, tsqs)
      sm = filter(sm, e)
      return sm

    case Extend(i, v, e):
      i_path = path.extend(ExtendInner)
      sm = combine(i, i_path, sm,
                   btsqs, subqs, tsqs)
      sm = extend(sm, v, e)
      return sm

    case Join(l,r):
      l_path = path.extend(JoinLeft)
      l_subqs = submap(subqs, l_path)
      l_tsqs = submap(tsqs, l_path)
      sm = combine(l, sm, l_path,
                   btsqs, l_subqs, l_tsqs)
      r_path = path.extend(JoinRight)
      r_subqs = submap(subqs, r_path)
      r_tsqs = submap(tsqs, r_path)
      sm = combine(r, sm, r_path,
                   btsqs, r_subqs, r_tsqs)
      return sm
```

**Fig. 19.** Simplification of combining function in Python-syntax.

preserves these properties for Filter, Extend and Join in Sections 6.4.3–6.4.5. Additionally, we note that the processing of Join will eventually lead to correct cardinalities, and the removal of $^*$.

The real combination procedure is slightly more advanced. Instead of always using `sm = None` when processing a subquery, we allow `sm` to be a multiset of solution mappings that will constrain and be joined to the solution mappings. In particular, we pass solution mappings `sm` to `exec_subquery` and rewrite the static subquery to be compatible with these solution mappings using Join and Values-graph patterns where we encode the permissible solution mappings, and join the results of `exec_subquery` to `sm`. To keep the proof relatively simple, we do not cover these advanced optimisations.

We move now to discussing time series query preparation, which is important for proving that the result of combing the solution mappings with the results of the time series query is correct.

### 6.4.1. Preparing time series queries

Time series preparation starts at the level of the static subquery. Fig. 20 contains a simplified excerpt of time series query preparation. We only consider the basic graph pattern and the group by graph pattern in this figure. We consider only the case where a single (not multiple) basic time series query (BSTQ) is associated with a given BGP. A BSTQ is originally created when rewriting a BGP (c.f. Fig. 18). We illustrate here only how time series queries pushing down filter and group by work, and assume the time series database has support for group by pushdowns but not for all expression types.

When Time Series Query (TSQ) preparation considers a group by-pattern, it first tries to prepare the inner graph pattern in a special mode where preparation terminates if information is lost. For filters, preparation fails in this grouping mode if the conjunction of the rewritten expression and the prepared expression is weaker than (i.e. not equivalent to) the original expression. This could for instance happen if some parts of the filtering expression are not supported in the time series database. We also must be able to push down bindings (extend-graph patterns) contained in the inner graph pattern of a group by to complete the group by pushdown. Currently, joins always return None in grouping mode, as we have not yet implemented pushdowns for joins. Finally, we must be able to keep every aggregation operation in identical form (`prepare_aggregations`), which is conditional on support in the time series database. If we successfully complete these steps, a grouped time series query is associated with the path of the group by pattern. The function `filter_by_variables` will filter out the grouping variables that do not originate in the time series database from the group by variables (`b`). Instead, we will associate the time series identifiers with a grouping variable (cf. Section 5.5) with the function `add_grouping_map`. This happens both to the solution mappings `sm` and to the grouped time series query `group_tsq`. Otherwise, we prepare the inner graph pattern without trying to include it in a group by pushdown.

Note that in the non-grouping mode, we allow filter pushdowns that are not jointly (with the rewritten expression) equivalent to the original expression. The filter pushdown is in this case associated with the original path (the path of a basic graph pattern $B$).

### 6.4.2. Combining time series queries

There are two cases where we combine the solution mapping with the results from the time series query. One is if there are some sequence of nested extend and filter graph patterns containing a basic graph pattern, where some of these graph patterns are pushed down into a time series query. We noted in Section 6.4.1 that these results will be attached at the contained BGP, and we discuss this case of time series combination here.

Let $B$ be a basic graph pattern embedded in the graph pattern $P$ which corresponds to a static subquery, and let $B$ be the attachment point of a time series query. We let $\Lambda(B, G_{comb})$ be the input solution mapping in the combination after entering `combine` for $B$ and if applicable executing an associated static subquery (`exec_subquery`).

If we are processing a leftmost BGP, we the equality that: 6.4:

$$\Lambda(B, G_{comb}) = \Omega(r(P), G_{comb})$$

this equality allows us to conclude that:

$$C(\Omega(P, G_{impl}), \Lambda(B, G_{comb}))$$

We must also show the basic condition that:

$$C(\Lambda(B, G_{comb}), \Omega(B, G_{impl}))$$

```
def prepare(gp:GP, btsqs:Dict[Path, BTSQ],
            sm:SM, path:Path, try_group:bool)
    -> Option[Dict[Path, TSQ]]:

  match gp:
    case BGP(ts):
      for btsq in btsqs:
        if btsq.datapoint.path == path:
          return {path:btsq}
      return None

    case Group(i,a,b):
      i_path = path.extend(GroupInner)
      i_g_prep = prepare(i, btsq, i_path, sm, true)
        # We check that preparation succeeded, i.e.
        # all of i can be pushed down without loss
        if i_g_prep is not None:
          # Extract unique key, val
          {k:i_prep} = i_g_prep
          a_prep = prepare_aggregations(a, i_prep)
          # Aggregations can be pushed down?
          if a_prep is not None:
            b_prep = filter_by_variables(b, i_prep)
            i_prep = add_grouping_map(i_prep, b, sm)
            group_tsq = GroupTSQ(
              i_prep, a_prep, b_prep)
            return {path:group_tsq}
      return prepare(i, i_path, sm, false)

    case Filter(i, e):
      i_path = path.extend(FilterInner)
      i_prep = prepare(
              i, btsq, i_path, sm, try_group)
      if i_prep is None:
        return None
      e_rw = get_expr_rewrite(e)
      out_map = dict()
      for (k,v) in i_prep:
        e_prep = prepare_expr(e, v)
        out_filter = FilterTSQ(v, e_prep)
        if try_group:
          if And(e_rw,e_prep).equivalent(e):
            out_map[k] = out_filter
          else:
            return None
        out_map[k] = out_filter
      return out_map
    ...
```

**Fig. 20.** Simplified excerpt of the time series query preparation procedure in Python-syntax.

We may substitute using the equality above, and can show instead the equivalent statement that:

$C(\Omega(r(P), G_{comb}), \Omega(B, G_{impl}))$

$P$ and $r(P)$ consists of a combination of a Join, Filter, Extend, BGP together with any trivially rewritten graph pattern. Since Extend is not allowed to shadow any variable, a variable $v$ that exists in both $B_V$ and $r(P)_V$ originates in one of the solution mappings of $B$ over $G_{impl}$, and must be bound. Let:

$\mu_B \in \Omega(B, G_{impl})$

In particular, for the set of variables in $r(P)$ that occur in $B$ there is a solution mapping:

$\mu_{r(P)} \in \Omega(r(P), G_{comb})$

such that:

$\forall v \in B_V \cap r(P)_V : \mu_{r(P)}(v) = \mu_B(v)$

In other words, $\mu_{r(P)}$ is compatible with some $\mu_B$. The variables in $r(B)$ introduced by the rewriting process are guaranteed not to occur in $B$, and so impose no constraints on the compatibility. However, each Join in which $r(B)$ is included in $r(P)$ potentially multiplied the cardinality of $\mu_{r(P)}$ by some factor $\geq 1$. When the combination procedure is finished processing this join, we will have the correct cardinality, but at the level of $B$ we can only show that the relation $C$ holds up to the factors introduced by the Joins in which $r(B)$ is embedded in the subquery $P$. We denote this weaker variant of $C$ by $C^*$:

$C^*(\Lambda(B, G_{comb}), \Omega(B, G_{impl}))$

If we are not in the leftmost BGP, we may simply assume as our induction hypotheses:

$C(\Omega(P, G_{impl}), \Lambda(B, G_{comb})) \wedge$
$C^*(\Lambda(B, G_{comb}), \Omega(B, G_{impl}))$

For a given $\mu_{comb} \in \Lambda(B, G_{comb})$, we introduce a set of solution mappings $\text{combined}(\mu_{comb}, f_{d\in D})$, where $f_{d\in D}$ is a time series database. To simplify the proof, we only consider the case where there is one time series variable in $B$, but can be generalised by iteratively applying the construction below for each time series variable. Let ?s be the time series variable in $B_V \cap r(B)_V$. The datapoint variable corresponding to ?s will be denoted by ?p, the datatype variable by ?d and the external identifier variable by ?e. We assume here that $?d$ has a unique assignment common to all $\mu_{comb}$. As discussed above, there is no way in which $\mu_{comb}(?t)$ is unbound.

We construct $\text{combined}(\mu_{comb}, f_{d\in D})$ using the corresponding datapoints in the time series database. Given $\mu_{comb} \in \Lambda(B, G_{comb})$, and $p \in f_{\mu_{comb}(?d)}(\mu_{comb}(?e))$ we define a solution mapping $\mu_{comb,p}$:

$$\mu_{comb,p}(x) = \begin{cases} \text{value}(p) & x = ?\text{v} \\ \text{timestamp}(p) & x = ?\text{t} \\ p & x = ?\text{p} \\ \mu_{comb}(x) & \text{otherwise} \wedge \\ & x \notin \{?\text{e}, ?\text{d}\} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Now we can define the new set of solution mappings.

$$\text{combined}(\mu_{comb}, f_{d\in D}) = \{\mu_{comb,p} | p \in f_{\mu_{comb}(?d)}(\mu_{comb}(?e))\}$$

This construction respects compatibility and thus $C$, since we are simply mirroring the construction of $G_{impl}$ when we extend each $\mu_{comb}$. $\Gamma(B, G_{comb})$ is given by the union of all such sets:

$$\Gamma(B, G_{comb}) = \bigcup_{\mu_{comb} \in \Lambda(B, G_{comb})} (\text{combined}(\mu_{comb}, f))$$

The cardinality of the members of $\Gamma(B, G_{comb})$ is inherited by the original cardinalities of the $\mu_{comb}$ from $\Lambda(B, G_{comb})$ used to generate them. This preservation of cardinality respects $C$. We conclude that:

$C^*(\Gamma(B, G_{comb}), \Omega(B, G_{impl}))$

Since we have only enforced conditions that must be met by $\mu_{impl} \in \Omega(P, G_{impl})$, we preserve the corresponding relation:

$C(\Omega(P, G_{impl}), \Gamma(B, G_{comb}))$

In the above proof, we have not considered pushdowns directly. We sketch how these pushdowns can be handled in what follows. Pushing down an Extend pattern or a Filter, works well with the proof above. The variable introduced by Extend will be determined by other values in the solution mapping, and can be treated in the very same way as combining Extend graph patterns in the discussion in Section 6.4.4. Similarly, the variable introduced by Extend does not overlap with $B_V$. In sum, both $C$-relations are preserved. For Filters, none of the solutions eliminated by eagerly evaluating the expression and filtering will have made it into $\Omega(P, G_{impl})$, so pushing down filters also preserves $C$. Related time series queries attaching to the same BGP can be pushed down into a single time series query. A proof would have to show that the iterative combination described above is equivalent to jointly evaluating them in the time series database. A grouped time series query is associated with a group graph pattern. We have seen that these time series queries contain all the information necessary to evaluate the group by graph pattern. Assuming that the time series database does so faithfully, we only have to join the results from the time series database with the solution mapping on the grouping column.

#### 6.4.3. Combining filter graph patterns

Let $\text{Filter}(I, expr)$ be a filter graph pattern contained in a graph pattern $P$ which is rewritten as a subquery. We call `combine` using $I$ and `sm`:

$$\texttt{sm} = \texttt{combine}(\texttt{gp} = I, \texttt{sm} = \texttt{sm}, ..)$$

We denote `sm` by $\Gamma(I, G_{comb})$. We can now assume the induction hypotheses that:

$$C(\Omega(P, G_{impl}), \Gamma(I, G_{comb})) \wedge$$
$$C^*(\Gamma(I, G_{comb}), \Omega(I, G_{impl}))$$

The combining procedure will simply enforce the filter expression per SPARQL 1.1 semantics, i.e.:

$$\texttt{sm} = \texttt{filter}(\texttt{sm}, expr)$$

We let `sm` be denoted by $\Gamma(\text{Filter}(I, expr), G_{comb})$, since this is what we return after running `combine` for Filter. Since solution mappings lost when processing inner graph patterns of $P$ never reappear in $\Omega(P, G_{comb})$, the solution mappings lost from $\Gamma(I, G_{comb})$ after filtering could never have had a compatible mapping in $\Omega(P, G_{impl})$ to begin with. Second, if any $\mu_{impl} \in \Gamma(I, G_{impl})$ has $expr(\mu_{impl}) = \texttt{false}$ then this surely applies to any compatible $\mu \in \Omega(I, G_{comb})$ also. Hence we have as required that:

$$C(\Omega(P, G_{impl}), \Gamma(\text{Filter}(I, expr), G_{comb})) \wedge$$
$$C^*(\Gamma(\text{Filter}(I, expr), G_{comb}),$$
$$\Omega(\text{Filter}(I, expr), G_{impl}))$$

#### 6.4.4. Combining extend graph patterns

Let $\text{Extend}(I, v, expr)$ be an extend graph pattern contained in a graph pattern $P$ which is rewritten as a subquery. We compute an updated set of solution mappings:

$$\texttt{sm} = \texttt{combine}(\texttt{gp} = I, \texttt{sm} = \texttt{sm}, ..)$$

We assume the induction hypotheses that:

$$C(\Omega(P, G_{impl}), \Gamma(I, G_{comb})) \wedge$$
$$C^*(\Gamma(I, G_{comb}), \Omega(I, G_{impl}))$$

We now modify the solution mappings according to the SPARQL 1.1 specification for Extend, computing the value of $v$ using $expr$ applied to each solution mapping. Strictly speaking, we may skip this step if $r(expr) = expr$, as this will produce no change.

$$\texttt{sm} = \texttt{extend}(\texttt{sm}, v, expr)$$

We will call this updated multiset of solution mappings `sm` $\Gamma(Extend(I), G_{comb})$, as this is the one we return after processing Extend. Since we are ignoring the topmost projection in $P$, and assume deterministic $expr$, any solution mapping in $\Omega(P, G_{impl})$ will continue to be compatible with an extended solution mapping in $\Gamma(\text{Extend}(I, v, expr), G_{comb})$. The second relation $C$ is obviously preserved, since extensions are deterministic. Hence we have as required that:

$$C(\Omega(P, G_{impl}), \Gamma(\text{Extend}(I, v, expr), G_{comb})) \wedge$$
$$C^*(\Gamma(\text{Extend}(I, v, expr), G_{comb}),$$
$$\Omega(\text{Extend}(I, v, expr), G_{impl}))$$

#### 6.4.5. Combining join graph patterns

Let $\text{Join}(L, R)$ be a filter graph pattern contained in a graph pattern $P$ which is rewritten as a subquery. Join graph patterns starts with computing first an updated multiset of solution mappings `sm`.

$$\texttt{sm} = \texttt{combine}(\texttt{gp} = L, \texttt{sm} = \texttt{sm}, ..)$$

We call this updated value $\Gamma(L, G_{comb})$. We may now assume the induction hypothesis:

$$C(\Omega(P, G_{impl}), \Gamma(L, G_{comb})) \wedge$$
$$C^*(\Gamma(L, G_{comb}), \Omega(L, G_{impl}))$$

Next we call `combine` for $R$ with the updated solution mappings `sm`.

$$\texttt{sm} = \texttt{combine}(\texttt{gp} = R, \texttt{sm} = \texttt{sm}, ..)$$

We call this updated value $\Gamma(R, G_{comb})$. Assuming `combine` has the desired properties, it follows that:

$$C(\Omega(P, G_{impl}), \Gamma(R, G_{comb})) \wedge$$
$$C^*(\Gamma(R, G_{comb}), \Omega(R, G_{impl}))$$

From Section 6.4.2, we see that the process of combining time series queries is in fact a join (or iterated joins if there is more than one time series variable). There is nothing more to do in join processing, and so we have that:

$$\Gamma(R, G_{comb}) = \Gamma(\text{Join}(L, R), G_{comb})$$

We can use this identity to rewrite our induction hypothesis to the required conclusions.

$$C(\Omega(P, G_{impl}), \Gamma(\text{Join}(L, R), G_{comb})) \wedge$$
$$C^*(\Gamma(\text{Join}(L, R), G_{comb}), \Omega(\text{Join}(L, R), G_{impl}))$$

If we are finishing the combination procedure for the topmost Join in our subquery, we can write as the second conclusion without $^*$:

$$C(\Gamma(\text{Join}(L, R), G_{comb}), \Omega(\text{Join}(L, R), G_{impl}))$$

## 7. Implementation

In this section, we describe our implementation of the solution approach, called Chrontext. The implementation is open source software,[1] and has a permissive Apache 2.0 license. At the time of writing, it is a prototype which has support for a large part of SPARQL 1.1.

Chrontext is a system for portable, high performance ontology based data access to contextualised time series data using SPARQL. Chrontext requires that the static graph (context) is augmented with metadata linking nodes to time series in the time series database as described in Section 5.2. It currently supports both OPC UA Historical Access (HA) and SQL APIs for accessing time series data, and is extensible to other APIs by interpreting an intermediary format for queries over time series data. As OPC UA Historical Access tends to be found in on-premise deployments, and SQL tends to be found in cloud deployments,

[1] https://github.com/magbak/chrontext.

the solution offers a portable query interface across infrastructures. Depending on time series API capabilities, Chrontext pushes down data processing such as filters and aggregations in order to optimise query performance. It is especially developed for situations where the main use of context is to select appropriate time series data from very large data sets. As such, it only supports select-queries. Chrontext exploits this situation by always querying static context data before time series data in order to retrieve a minimal part of the time series data set.

The implementation is based on the Rust programming language, and uses the Polars (Vink, 2022b) library for in-memory columnar data processing. To maximise performance, Chrontext supports SQL over Arrow Flight. We use a Python wrapper, and transfer data from the Rust binary to Python by reference using Arrow Flight IPC. This is important in order to meet the second and third requirements (R2 and R3). Moreover, the Polars library features support for lazy evaluation of common data set manipulation operations such as filters and joins. When evaluation is required, Polars optimises the sequence of operations. We utilise this feature whenever it is possible. Chrontext uses the Spargebra library (Tanon, 2022b) to parse SPARQL and manipulate SPARQL graph patterns and expressions. It makes heavy use of the oxrdf-library (Tanon, 2022a) to represent concepts in RDF. Admittedly, Pandas Dataframes are more popular, but are not optimised to reduce serialisation costs. Polars Dataframes can however easily be converted to the more popular Pandas Dataframe, meeting R3.

In order to execute the original query, Chrontext creates one or more static SPARQL queries, which are run on the SPARQL engine. Chrontext then creates one or more time series queries (TSQ), an intermediate structure. The TSQs are transformed into API calls of the type appropriate for the underlying time series database, e.g. SQL. Additional time series APIs such as OSIsoft PI (OSIsoft, 2022) and InfluxDB (InfluxData, 2022a) can be supported by translating the intermediary TSQs to API-calls and providing expected results as Polars dataframes. Chrontext does not implement a SPARQL 1.1 HTTP endpoint, but is a client library where SPARQL queries result in an in-memory Apache Arrow backed result set called a Polars DataFrame in either Rust or Python. In the result set, variables are represented natively using Apache Arrow datatypes. It does not currently support result sets where a variable has multiple data types. Supporting such result sets is in principle possible by casting to the string-representation of the RDF term. Such functionality would however increase the complexity of Chrontext in many parts of the solution. We rely on fast, column based operations from Polars to perform almost all SPARQL expression-computations using native data types that are uniform in a single column, but this does not work for columns with heterogeneous data types represented as strings. Adding support for heterogeneous types likely requires a separate implementation of all SPARQL expressions. As benchmarking the core functionality of Chrontext was possible without this functionality, we have not prioritised it.

There are two supported time series APIs, SQL and OPC UA HA. In case the time series database is an OPC UA HA service, we must limit what is pushed down into the time series query. The historical access service supports raw access to time series data within an interval or aggregated access to time series data with a limited set of aggregation functions. We only create grouped pushdown queries if filters on time series data only involve to- and from constraints on the timestamp. In other cases we fall back to raw access, and perform aggregations client side.

It is possible to enable all pushdowns discussed in Section 5.6 with the SQL backend, but users can configure a reduced set of pushdowns according to the level of SQL support available and the performance characteristics of the database. With Chrontext, if the SQL backend is configured with partitions by year, month and day, we automatically rewrite the builtin SPARQL functions to select these columns instead, and to rewrite any filters on timestamps to exploit this partitioning in a way similar to what is done in Fig. 24. Additionally, in the presence of year, month and day partitioning, Chrontext rewrites join conditions

```
FROM (...) as "q1"
INNER JOIN (...) as "q2" ON
"q1"."t" = "q2"."t"
```

**Fig. 21.** Join on timestamp column before partition optimisation.

```
FROM (...) as "q1"
INNER JOIN (...) as "q2" ON
"q1"."t" = "q2"."t" AND
"q1"."year_col" = "q2"."year_col" AND
"q1"."month_col" = "q2"."month_col" AND
"q1"."day_col" = "q2"."day_col"
```

**Fig. 22.** Join on timestamp column with partition optimisation.

on timestamps to also join on these partitioning columns. For instance, assume that we are joining subqueries `q1` and `q2` on a common timestamp column `t`. Such a query is given in Fig. 21. We ensure that the partitioning columns `year_col`, `month_col` and `day_col` are found in `q1` and `q2` respectively. This allows us to rewrite the join condition; the result is given in Fig. 22.

## 8. Evaluation

In this section, we evaluate whether our solution meets the second requirement (R2) of providing high throughput low latency access to contextualised time series data by comparing our solution to the popular open source virtual knowledge graph Ontop. We first introduce the scenario, the data set, and data lake with lakehouse infrastructure. Both the static context and time series data are accessible through a common SQL interface. We describe how we map this data to Ontop and present the Chrontext configuration along with a set of four queries used in the evaluation.

### 8.1. Experimental setting

In this section, we describe scenario, infrastructure, data set, and queries used in the experiments. To construct the evaluation scenario, we use the stOttr terse syntax for the Ottr templating language (Skjæveland, 2022; Skjæveland et al., 2021) to define templates for a wind farm with very simple instrumentation. The wind farm is modelled according to the Reference Designation System — Power Systems as described part 10 of the in ISO/IEC 81346-10:2022. The scripts used to create the scenario are available in an associated repository.[2] Fig. 4 in Section 5.3 displays part of this graph.

In the evaluation benchmark, we use an AWS Elastic Kubernetes Cluster consisting of EC2 m6i.2xlarge instances using third generation Intel Xeon processors. These instances have eight cores and 32 GB of RAM each. An S3 bucket is used for the time series data. The time series data consists of generated data for 400 wind turbines. Each wind turbine is associated with double-valued wind speed (m/s), wind direction (degrees) and production (Watts) and with a boolean valued operational status. Generated data are available for a 72 h period and is sampled every ten seconds. The information model in our experiment is straightforward and contextualises a limited set of time series data that nonetheless represents a potential real world scenario. It can thus be seen as a baseline example for evaluating the core task of using information models to access time series data. This data is stored in an Apache Hive format, with the following folder structures generated by the PyArrow library (The Apache Software Foundation, 2022e):

[2] https://github.com/magbak/chrontext_benchmarks.

**Table 5**
Query types in the benchmark.

| Query | Tme series | Aggregation |
|---|---|---|
| Production | Production | Raw |
| Grouped production | Production | Mean 10 min. |
| Grouped multiple value | Production, Wind Direction, Wind Speed | Mean 10 min. |

```
/timeseries_boolean/year/month/day/id/part-i.parquet
/timeseries_double/year/month/day/id/part-i.parquet
```

In Dremio, these directory names (except for the top one which names the table) are exposed as columns `dir0`, `dir1` and so on in increasing depth. Filtering on these directory names exploits the inherent partitioning, and reduces the amount of data Dremio must fetch. The S3 bucket is in the same region as the Kubernetes cluster. Our Dremio deployment uses one node for the coordinator, three executor nodes and three nodes for ZooKeeper.[3] We allocate one node to a PostgreSQL Database containing tables with the static model. Two other nodes are allocated for the Ontop endpoint and running the benchmark queries respectively. Dremio is connected to both the S3 bucket and the PostgreSQL database. The folders containing time series data are promoted to queriable data sets in Dremio. All AWS resources are located in the same region.

There are three query types, which are run with changes to their filtering to extract time series data for 1, 10 100 and 400 wind turbines respectively. The *production queries* extract the raw time series data for production for the given turbines. The *grouped production queries* extract the mean production for each 10 minute interval for the given turbines. The grouped production query for ten wind turbines was already shown in Fig. 5. The *grouped multiple value queries* extract averaged production values together with averaged weather values (wind direction and speed) for each 10 minute interval for the given turbines. Originally, the grouped multiple value queries also included a filter on a boolean value indicating whether or not a turbine was operating, but these queries produced an unknown error in Ontop which we were unable to diagnose. The queries in our benchmark are straightforward and simply navigate to the time series data to extract it, possibly in aggregated form. They can be seen as a core, baseline case for the kind of queries that should be performed by query systems supporting contextualised access to time series data in industry. The three query types are summarised in Table 5.

### 8.2. Ontop configuration

Ontop is configured to connect to Dremio, which makes available both the time series data in S3 and the data in Postgres in a single SQL interface. Additionally, Ontop must be configured to interpret the results of predefined SQL queries as RDF triples in an OBDA mapping file. Mapping the PostgreSQL tables is trivial, as there is exactly one table for each property type in the static part of the model. Mapping time series data is more elaborate. Ontop does not support blank nodes in mappings. This means that we either must construct an identity (IRI) for data points, or that we must create one and include it along with the value and timestamp in the parquet files. We have opted for the second alternative, due to concerns that query-time data point identity construction could impact Ontop query performance negatively.

In order to exploit partitioning, we associate each data point with its year, month and day found in the `dir0`, `dir1` and `dir2` columns respectively. Fig. 23 contains the relevant part of the Ontop mapping. The Ontop variants of the benchmark queries are written to use these predicates to derive the year month and day of a timestamp instead

[3] https://zookeeper.apache.org/.

```
[PrefixDeclaration]
wp:     https://.../windpower_example#
bm:     https://.../chrontext_benchmarks#
ct:   https://.../chrontext#

[MappingDeclaration] @collection [[
mappingId       timeseriesdouble

target
wp:{dir3} ct:hasDataPoint wp:{datapoint_id}.
wp:{datapoint_id} bm:hasYear {dir0}^^xsd:int ;
    bm:hasMonth {dir1}^^xsd:int ;
    bm:hasDay {dir2}^^xsd:int ;
    ct:hasValue {value}^^xsd:double;
    ct:hasTimestamp {timestamp}^^xsd:dateTime .

source
SELECT "dir0", "dir1", "dir2", "dir3",
       "datapoint_id", "timestamp", "value"
FROM "s3"."chrontext-benchmark"."timeseries_double"
...
```

**Fig. 23.** Ontop time series data mapping. The mapping has been rewritten for readability in a narrow format, at the cost of breaking Ontop syntax rules.

```
PREFIX
bm:<https://.../chrontext_benchmarks#>
...
?dp bm:hasYear ?year .
?dp bm:hasMonth ?month .
?dp bm:hasDay ?day .
BIND(10 * FLOOR(minutes(?t) / 10.0) as ?minute_10) .
BIND(hours(?t) AS ?hour) .
FILTER (?site_label = "Wind Mountain"
&& ?wtur_label in
("A1", "A2", ..., "A10")
&& (
(?year = 2022 && ?month = 8 && ?day = 30
&& ?t >= "2022-08-30T08:46:53"^^xsd:dateTime) ||
(?year > 2022) ||
(?year = 2022 && ?month > 8) ||
(?year = 2022 && ?month = 8 && ?day > 30))
&& (
(?year = 2022 && ?month = 8 && ?day = 30
&& ?t <= "2022-08-30T21:46:53"^^xsd:dateTime) ||
(?year < 2022) ||
(?year = 2022 && ?month < 8) ||
(?year = 2022 && ?month = 8 && ?day < 30))) .
```

**Fig. 24.** Ontop partitioning workarounds.

of the built-in SPARQL functions. In order to apply the partitioning scheme to filters, we need to expand filters on timestamps to allow the partitioning to be exploited. Fig. 24 illustrates how the filter in the grouped production query extracting data from ten wind turbines is rewritten.

To run the queries in Ontop, we use the SPARQLWrapper-library (Herman, Fernández, Alonso, & Zakhlestin, 2022), which produces a Python dict (a dynamically typed map) with the results. SPARQLWrapper is a thin wrapper around the requests-library used for

HTTP-requests. Ideally, the results should be converted to DataFrames. However, we chose to keep the Python-native format in order not to inadvertently introduce poor performance in a DataFrame-conversion step. Our hypothesis is that Chrontext will outperform Ontop, and eliminating the conversion step only makes this task more difficult for Chrontext.

### 8.3. Chrontext configuration

We have created a Python wrapper for Chrontext. Query execution happens in Rust, but the resulting Polars DataFrame is passed by reference to Python using Apache Arrow IPC. We configure Chrontext to enable all pushdowns and partitioning on dates. Chrontext is configured to use Ontop as a SPARQL endpoint and Dremio as the SQL backend. The required metadata is added to PostgreSQL and mapped in Ontop. When running the static rewrite of the SPARQL query then, Chrontext sends the query to Ontop, which generates an SQL query to Dremio, which then forwards the query to PostgreSQL. Performance would likely improve with a dedicated triple store and by removing Dremio from static SPARQL processing, but using Ontop with Dremio allows us to compare performance on identical infrastructure.

After having run the benchmark we investigated strongly nonlinear performance found in the first set of queries and found that it was due to a costly join. In particular, after executing the time series query, the results must be joined with static context on the client side. After the benchmark was run, we became aware of an important feature in the Polars library, allowing for sorted merge joins (Vink, 2022a), which have been used to parallelise and speed up SPARQL query processing in distributed settings (e.g. Groppe & Groppe, 2011; Przyjaciel-Zablocki et al., 2013). The main cost of a sorted merge join is the phase where the tables to be joined are sorted on the join columns (Albutiu, Kemper, & Neumann, 2012). By adding an order by-clause to the SQL to the time series database, we were able to offload part of the cost of the join to the Dremio database and achieve a large performance improvement. The optimisation is now part of Chrontext. We keep both results so as to not cherry pick data. We also verified that query processing times in this second run for the unaffected queries were highly similar for Chrontext, and that the result sets were identical (up to ordering). This verification strongly indicates that the improvement is not due to a misconfiguration in the second run.

### 8.4. Results

We present the results of the evaluation benchmarks below. Recall that there are three query types (c.f. Table 5) and that each query type has variants extracting data from 1,10,100 and 400 wind turbines, making 12 query instances. Raw data is presented in scatter plot. We jitter data horizontally, to display the points individually. In reality, all points $(x, y)$ have $x \in \{1, 10, 100, 400\}$, where $x$ is the number of wind turbines we extract data for. Note that the $x$ axis has a $log_{10}$ scale.

#### 8.4.1. Production queries

In Production Query 1, one of the 10 runs for Ontop crashed due to an error during query processing in Dremio. We have not attempted to diagnose the cause of this error. This data point was dropped. In Production Query 4, the SPARQL Python client (SPARQLWrapper) used for Ontop ran out of memory before the query could complete. Looking at the query log in Dremio, we were able to see that the 400 wind turbine query took 240 seconds in Dremio for Ontop. Since this number is not directly comparable, we have not included it in our plots. Chrontext completed every query. The raw results are plotted in Fig. 25. Summary statistics can be found in Table 6.

Investigating production queries with 100 and 400 wind turbines for Chrontext revealed that Dremio query processing and data transport completed quickly, and that most of the time was spent processing the results from Dremio. In contrast, Ontop processing time was mainly spent waiting for query processing in Dremio. As discussed in Section 8.3, Chrontext processing time was drastically improved after applying optimisations.

**Table 6**
Summary statistics for processing times (seconds) for the production queries by the number of wind turbines and solution used.

| Turbines | Solution | N | Mean | St. Dev. |
|---|---|---|---|---|
| 1 | Ontop | 9 | 32.87 | 6.21 |
| 1 | Chrontext | 10 | 0.41 | 0.02 |
| 1 | Chrontext Opt. | 10 | 0.43 | 0.03 |
| 10 | Ontop | 10 | 37.23 | 4.69 |
| 10 | Chrontext | 10 | 2.43 | 0.06 |
| 10 | Chrontext Opt. | 10 | 2.29 | 0.06 |
| 100 | Ontop | 10 | 91.23 | 6.43 |
| 100 | Chrontext | 10 | 39.52 | 1.02 |
| 100 | Chrontext Opt. | 10 | 6.53 | 0.04 |
| 400 | Ontop | 0 | – | – |
| 400 | Chrontext | 10 | 557.95 | 0.53 |
| 400 | Chrontext Opt. | 10 | 17.49 | 0.13 |

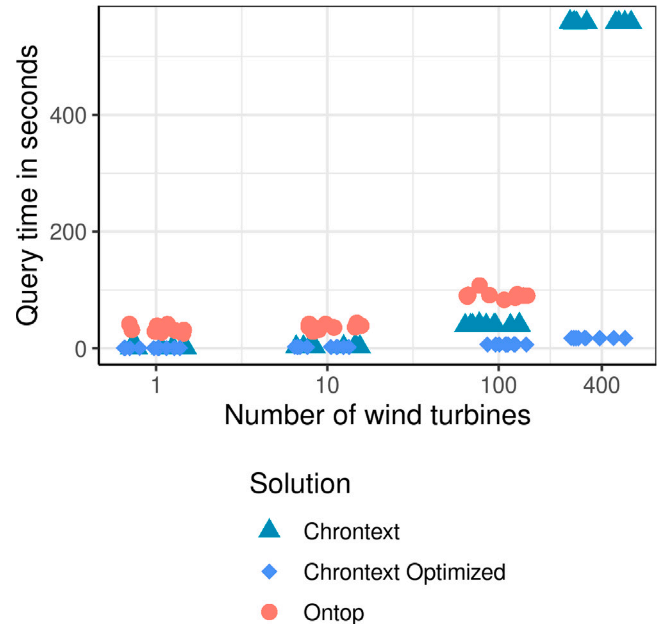



**Fig. 25.** Production queries.

In order to better understand the reasons for the difference in performance, we extracted number of rows scanned, total memory usage and query execution time in Dremio manually. We provide one sample for solution and each number of wind turbines, and these runs were separate from the runs in the experimental procedure. The results are presented in Figs. 26–28. We were able to include results for the 400-wind turbine case for Ontop in this case, as the out of memory-error happened client-side.

#### 8.4.2. Grouped production queries

Ontop and Chrontext both completed all grouped production queries. The raw results are shown in Table 7, and plotted in Fig. 29. Chrontext and Ontop processing times are dominated by query processing in Dremio for the most demanding queries with 100 and 400 wind turbines. We carried out identical checks for rows scanned, memory consumption and execution time in Dremio as for the production queries. For the grouped production queries, there is a similar relationship between the number of rows scanned and memory use for Chrontext and Ontop in Dremio. Chrontext scans 25 thousand rows when there is one wind turbine, and 10 million rows when there are 400 being queried. Ontop scans 207 millon rows in all cases. Chrontext

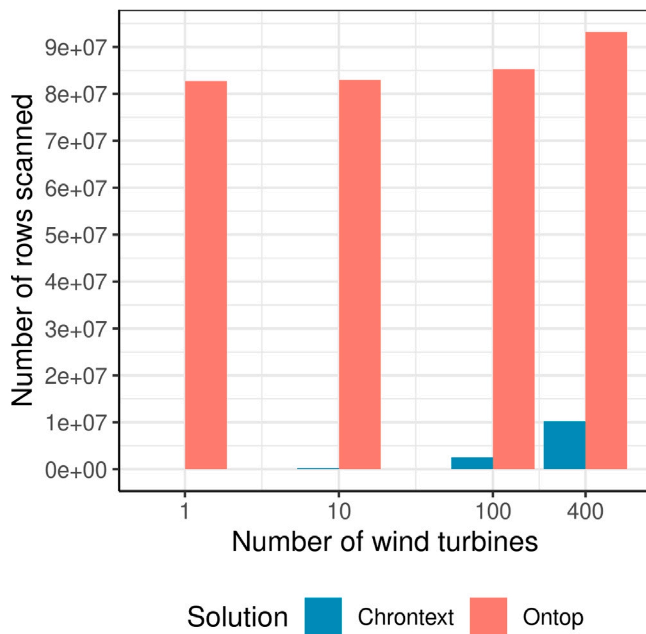



**Fig. 26.** Production queries: number of rows scanned in Dremio. For the case with one wind turbine, Chrontext scans 25920 rows, which are effectively not rendered due to the comparatively large scale of the $y$-axis.

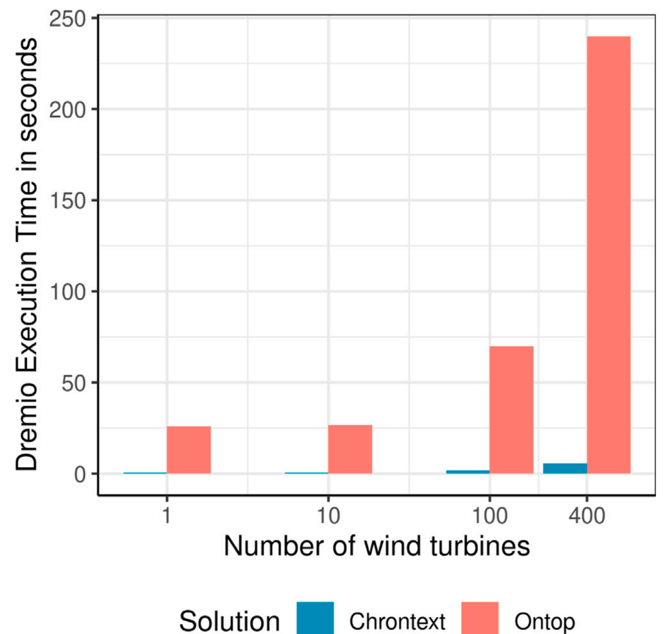



**Fig. 28.** Production queries: Total execution time in Dremio. Query execution times were below the threshold of one second to be reported in the Dremio GUI for chrontext in the one and ten-turbine cases, and have been entered as taking 0.5 seconds each.

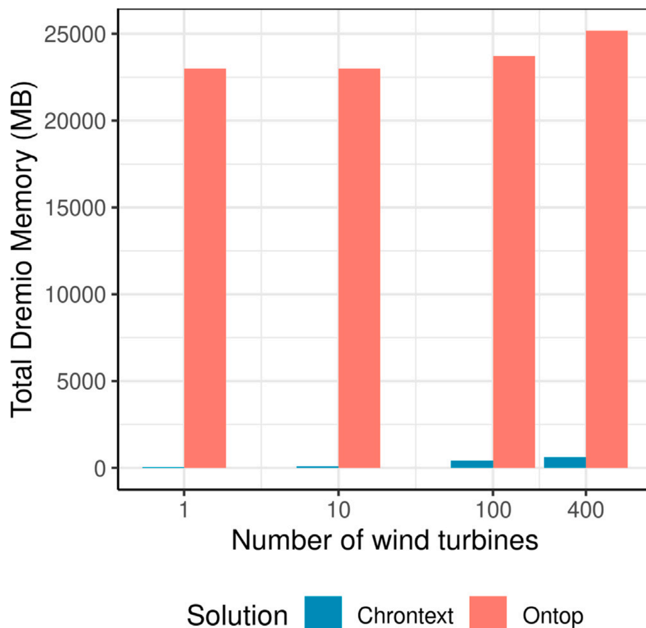



**Fig. 27.** Production queries: Total memory use in Dremio.

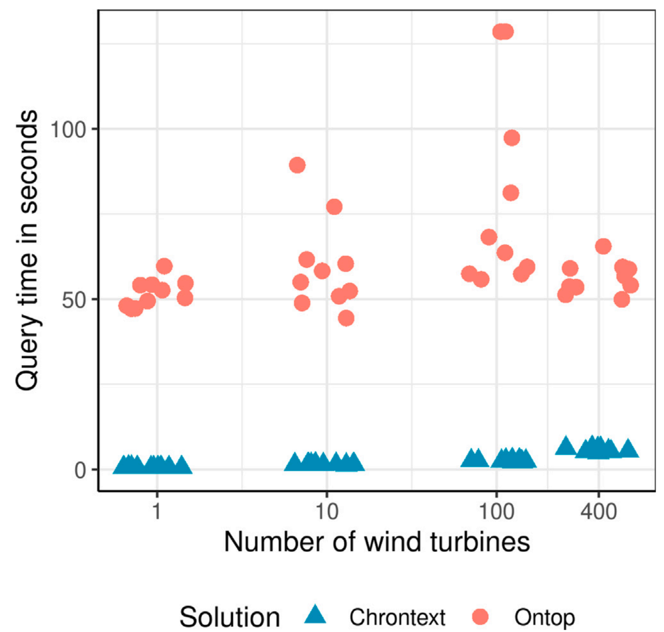



**Fig. 29.** Grouped production queries.

uses 87 MB of memory for one wind turbine, 1 GB for 400 wind turbines. Ontop uses 58 GB of memory in all cases. Dremio time use ranges from below 1 s (GUI does not show precise number) in the one turbine case to 7.39 s in the 400 turbine case for Chrontext. For Ontop, Dremio query time ranges from 53.13 s in the one turbine case, 62 s in the 100 turbine case and 57.6 seconds in the 400 turbine case.

#### *8.4.3. Grouped multiple value queries*

Ontop did not complete any of these queries due to out of memory-errors in Dremio. The raw results are plotted in Fig. 30 and summary statistics are given in Table 8. Both Chrontext and Ontop processing times are dominated by query processing in Dremio for the most demanding queries with 100 and 400 wind turbines.

### *8.5. Discussion*

In the production queries, Chrontext outperforms Ontop in all queries that complete on both solutions. These queries download the largest data sets of all the queries from Dremio. We expected the difference in query processing times for the production queries to be especially high as it reflects the expected increased deserialisation cost of Ontop. However, the difference between Chrontext and Ontop shrank as the size of the data set to be returned by the query grew. We

**Table 7**
Summary statistics for processing times (seconds) for the grouped production queries by the number of wind turbines and solution used.

| Turbines | Solution | N | Mean | St. Dev. |
|---|---|---|---|---|
| 1 | Ontop | 10 | 51.77 | 4.02 |
| 1 | Chrontext | 10 | 0.61 | 0.07 |
| 10 | Ontop | 10 | 59.83 | 13.71 |
| 10 | Chrontext | 10 | 1.46 | 0.09 |
| 100 | Ontop | 10 | 79.76 | 28.78 |
| 100 | Chrontext | 10 | 2.54 | 0.22 |
| 400 | Ontop | 10 | 56.20 | 4.65 |
| 400 | Chrontext | 10 | 5.69 | 0.56 |

**Table 8**
Summary statistics for processing times (seconds) for the grouped multiple value queries by the number of wind turbines and solution used.

| Turbines | Solution | N | Mean | St. Dev. |
|---|---|---|---|---|
| 1 | Chrontext | 10 | 1.80 | 0.10 |
| 10 | Chrontext | 10 | 1.74 | 0.04 |
| 100 | Chrontext | 10 | 4.98 | 0.17 |
| 400 | Chrontext | 10 | 28.16 | 0.51 |

found the cost of the local join operation to be the most likely cause. The local join was indeed the main cause as the performance after turning this join into a sorted merge join with offloading on Dremio improved performance by a factor of 32 for 400 wind turbines. After correcting the Chrontext join-problem, the bulk of the difference in query execution times was due to the speed with which Dremio could execute the respective queries.

We note that it is possible to push the join entirely into Dremio, at the cost of transferring more data. In a cloud-based Kubernetes setting, there is a great deal of bandwidth available, and a strategy of pushing the entire join into the SQL database may lead to better performance. In settings with less bandwidth however, it may perform worse. An evaluation under different bandwidth conditions is necessary to identify the best strategy. The downloaded data set for 400 wind turbines was only about 700 MB when exported as comma separated values (CSV), but still SPARQLWrapper ran out of memory. SPARQLWrapper relies on the performant requests-library to query a SPARQL-engine using HTTP, but converts the results to classes found in rdflib. This conversion is likely inefficient, and may confound findings comparing the deserialisation costs of Chrontext and Ontop. Further work on comparing the deserialisation costs of HTTP and Arrow for SPARQL results should better control for such issues.

The performance of Ontop is quite similar across the number of turbines, and we can see from the Dremio-extracted data that Dremio is processing equal amounts of data independently of whether we are querying one or 400 wind turbines. This finding strongly contrasts with the time series SQL query of Chrontext, which better constrains the Parquet-files that need to be scanned. The SQL query created by Chrontext requires that the identifier of the time series be a member of a collection of literals, and can straightforwardly be used to identify correct partitions. The SQL query created by Ontop however is querying the static tables in PostgreSQL for this information. Dremio appears not to use static information found in PostgreSQL to constrain which partitions of time series data to scan. In fact, Dremio even scans the boolean values, which are not required by any query in the benchmark. We observed both during the benchmark setup and during the benchmark that Dremio would occasionally produce an error when processing queries generated by Ontop. This happened when executing the Ontop-generated SQL query for one wind turbine. The SQL queries generated by Ontop are very large, and may be challenging for Dremio to execute reliably.

Regarding grouped production queries, Chrontext performs one to two orders of magnitude better than Ontop for the grouped production queries. Dremio is again unable to use static context from PostgreSQL

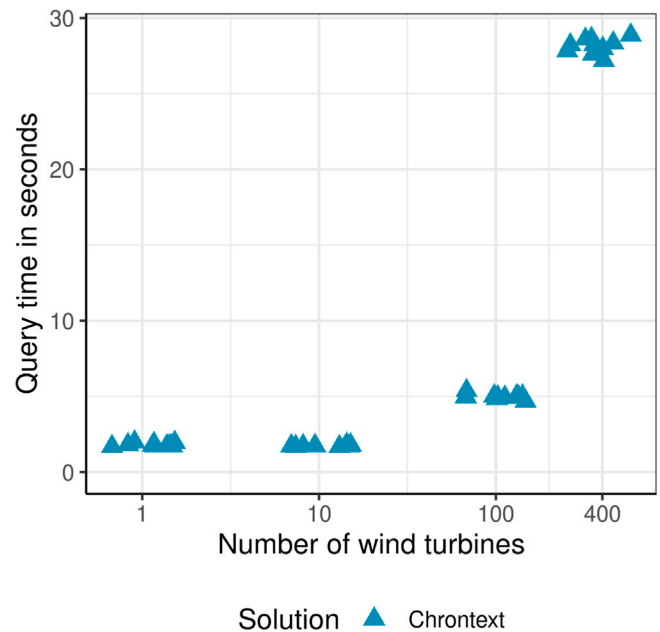



**Fig. 30.** Grouped multiple values queries.

to constrain which partitions of time series data are scanned. The Chrontext-queries scan exactly as much data as in the production queries, but uses somewhat more memory. However, the amount of data scanned and memory consumption in the grouped production queries when processing the Ontop queries has doubled. The grouped multiple values queries lead to out of memory errors in Dremio for Ontop. Note that the database only contains 905 MB of time series data, and that there are three executor nodes with 32 GB each in the Dremio cluster. For Chrontext, Dremio uses a total of 2.71 GB of memory to complete the time series SQL query in the 400 turbine case, indicating that we are nowhere near an intractable query memory-wise.

*8.5.1. Generalisability*

The performance of Dremio on the queries generated by Ontop may be due to the particular infrastructure combination of PostgreSQL and Dremio. Dremio may underestimate the cost of querying PostgreSQL, and make query plans that query PostgreSQL in parallel with S3. To investigate this possibility, we added Parquet-files corresponding to tables in PostgreSQL to AWS S3 instead, and rewrote the Ontop-mapping accordingly. We found very similar performance both for Ontop and Chrontext, and very similar numbers of rows scanned and memory use by Dremio when processing the queries. These results are available in the benchmark repository. Another possibility is that Dremio must be configured in a special way unknown to us to perform well in such situations.

Even if Dremio does not handle the Ontop-generated queries well, other data lakehouses may perform much better. However, Dremio is currently the only data lakehouse supported by Ontop. Our findings on using virtual knowledge graphs to query contextualised time series data is limited to this combination of software. For Chrontext, we have reason to believe that our findings on performance are more general. The queries generated by Chrontext involve literals constraining partitions with respect to time and the identifier, and any data lakehouse should be able to use such literal information to directly limit which partitions are scanned. Not being able to limit which partitions are scanned by using the literals in the query would lead any data lakehouse with a sufficiently large data lake to be practically useless.

#### *8.5.2. Summary*

The main difference between Ontop and Chrontext was the fact that Dremio was unable to limit the number of rows of the time series data being scanned when executing SQL queries generated by Ontop. This led to very similar execution durations across the number of wind turbines for Ontop. Additionally, the Ontop SQL queries led to much higher memory consumption, and were intractable for the grouped multiple values queries given our resource constraints. We predict that these problems would get worse when the size of the time series data set grows. Further research should identify if other data lakes are similarly affected.

For Chrontext, bounds on partitions were encoded as literals in SQL queries, and Dremio was able to limit the amount of time series data being scanned. We have good reason to believe that other data lakehouses will perform well with the SQL queries generated by Chrontext. Future research should attempt to corroborate these claims, and attempt to identify if there are queries generated by Chrontext where data lakehouses perform worse than expected. The effect of using Apache Arrow in the transport layer was not effectively investigated by this study. It should be studied further in a setting where deserialization of HTTP-based transport of SPARQL-results are better controlled and where bandwidth is more constrained.

## 9. Conclusion and future work

In this section, we summarise our findings and conclude that Chrontext meets the requirements we have outlined. Our work implies that Ontop, Dremio and Chrontext can be improved in order to perform better when processing time series queries. Finally, we suggest directions where Chrontext could be extended in order to better meet industry needs and discuss limitations of our work along with potential solutions.

### *9.1. Summary of findings*

We have presented Chrontext, a hybrid OBDA approach for querying time series data contextualised by data behind a SPARQL endpoint. The approach enables practitioners to query time series data stored in heterogeneous databases using the same query, which we call query portability, meeting the fourth requirement (R4). The approach is further able to exploit the computational power of heterogeneous time series APIs to push down parts of query execution appropriate for the capabilities of that database, meeting the fifth requirement (R5). Query portability in turn, can make it easier to train or estimate analytical models and deploy them in a new way, while respecting the availability constraints in industry. This type of analytical query portability using a declarative approach is not possible with existing tools. Query portability is enabled by an approach to query rewriting which has not been described in detail in the literature before. We further contribute a proof of the correctness of the query rewriting approach, meeting the first requirement (R1).

We have implemented the approach using state of the art technology as the open source software library Chrontext, providing strong evidence that the approach is practically feasible. Chrontext has support for large parts of the SPARQL 1.1 language. By using the Polars-library, it is indirectly based on Apache Arrow, and supports Apache Arrow Flight SQL backends for time series data. This lays the groundwork for meeting requirement two (R2). It exposes data to the user using Polars data frames, which can easily be converted into Pandas data frames, meeting the third requirement (R3). By supporting OPC UA HA and SQL time series databases, and with an extensible framework to support other time series databases, we meet the fourth requirement (R4) on supporting query portability also in an implementation. We are able to configure the degree to which we offload computations to the time series databases, and are able to push down group by graph patterns with aggregations into both OPC UA HA and SQL time series databases — meeting the fifth requirement (R5).

We have compared our solution with the OBDA approach of Ontop in an openly available industrially relevant scenario. We found that Chrontext outperforms Ontop by a factor that varies from 10-85x, and that the memory requirements of the Ontop generated queries mean it cannot complete one third of the benchmark. We attribute the difference in performance to the query execution strategy employed by Dremio, since Chrontext uses exactly the same infrastructure, but in a two step process. These results support the claim that our solution meets the second requirement (R2) on high throughput low latency time series data extraction in a data lakehouse setting. As we exploit core features of any data lakehouse, there is reason to believe that the high performance of Chrontext also extends to other data lakehouses.

The novelty of our contribution, compared to Bakken (2021) is that the approach is generalised to any industrial information model and any time series database satisfying elementary requirements, not just OPC UA information models and OPC UA HA, and that it is proven to be correct for a large part of SPARQL 1.1. Additionally, we describe and implement a novel approach to pushing down parts of the SPARQL query into the time series database, which can offload resource intensive query processing to a distributed query engine and reduce the amount of data that has to be transferred. Compared to the VKG approach of Ontop (Xiao et al., 2018), we are able to support a wider array of infrastructures that are relevant to industrial applications that consume time series data, and query time series data with much higher performance. Chrontext can thus help industry realise more of the benefits of the use of standardised information models. Chrontext makes it possible to use the cloud and on premise infrastructure to scale applications consuming time series across a portfolio of industrial assets, by leveraging existing models of these assets.

### *9.2. Improvements to ontop, dremio and chrontext*

Our research suggests ways of improving Ontop performance when the SQL engine is a data lakehouse through improved partition support. The data lakehouse Dremio can likely improve its performance when processing the queries produced by Ontop in this scenario by first resolving low latency contextual data sources before low latency data lakes with large amounts of data. In future work, we plan on generalising the metadata representing external data sources to overcome the limitation that data of a given type are always stored in the same SQL table in the same database of the same type. Allowing time series queries to span multiple databases of heterogeneous types introduces new challenges and opportunities in query sequencing and parallelisation.

We have expended little effort optimising the combination phase of query processing, and have simply relied on best practice guidelines on using Polars without investigating performance empirically. It is therefore likely that there are multiple opportunities to improve performance in the combination phase. With a hybrid architecture, there is also an opportunity to cache static query results which is not so easily exploited in a purely virtual OBDA approach. This opportunity is twofold. Static query results may be cached, particularly if the static context graph is only updated through a batch job. Second, if time series data tends to be monotonically increasing, we may also cache aggregates spanning intervals that are no longer subject to change. As exploratory data analyses become static artefacts and are deployed on recurring intervals, such a caching mechanism will likely be able to improve performance significantly.

### *9.3. Extensions to chrontext*

OBDA approaches generally do not implement access control to virtual knowledge graphs (Cima et al., 2020). Since our approach imposes no virtualisation constraints on the static query graph, it is

possible to employ a materialisation and query rewriting approach to access control (Abel et al., 2007; Padia, Finin, Joshi, et al., 2015). In data lakehouse solutions, such features are typically coarse grained (e.g. Google Cloud Platform (2022)) or part of an enterprise feature set (e.g. Dremio (2022)). Extending our OBDA-approach, we can granularity to such solutions. Role based access control to time series data can be accomplished by extending the metadata annotations and rewriting rule for basic graph patterns involving time series data, including a check that the user has the appropriate role to access the given time series.

Streaming is another area to which the hybrid architecture may be applied. There is however far less consolidation of standards in the streaming domain, and multiple competing approaches arising from different communities exist. When accessing contextualised time series data, SPARQL can significantly simplify the data access procedure. However, as Mörzinger (2019) points out, formulating these queries still requires considerable expertise. We believe it is possible to capture an industrially relevant and comprehensive set of queries for time series data extraction using a domain specific language, and plan on studying such an approach to further improve time series data accessibility for industrial practitioners.

### 9.4. Limitations

In this section, we describe the current limitations of the approach and implementation, together with ways of overcoming them if possible.

#### 9.4.1. Timeseries variables must be introduced in the same BGP

Our rewriting procedure currently assumes that a timeseries variable and related variables occur in the same basic graph pattern. I.e., we require that the timeseries variable (`?ts`) in Fig. 5, the data-point variable (`?dp`), value variable (`?val`) and timestamp variable (`?t`) all occur in the same BGP. Placing one or more of `?dp`, `?val` and `?t` in their own Optional-clause is not currently supported. This limitation will be removed in a future release.

#### 9.4.2. Limitations on pushdowns

Our rewriting approach is currently able to bundle time series queries that extract time series with identical timestamps (cf. synchronised pushdowns in Section 5.6). In the "grouped multiple value" query in the benchmark, our solution creates a single SQL query extracting multiple time series with identical timestamps. We may however be interested in extracting multiple time series where:

- one time series lags after the other by some time duration
- where measurements are not available for identical timestamps, but we would like to extract mean values from both time series for intervals where samples are available
- one time series occasionally has missing data, but we would nonetheless like to extract those time series that have data (i.e. using SPARQL Optional)

These use cases involve joins which it would be preferable to push down into a Data Lakehouse, but we are not able to do so yet. It is highly likely that the current approach can be extended to cover these cases, and we plan on doing so in the future.

#### 9.4.3. No support for custom functions for time series data

Our implementation lacks support for custom functions for time series data, such as those discussed by Mörzinger (2019). Missing support for such functions is a limitation of the implementation. We have assumed that analysts already are familiar with tools that can perform such analyses such as Python and R, and have not prioritised them. Users wanting to query time series data with custom functions or aggregations applied to time series data may do so in the current implementation by using Chrontext to extract relevant data sets, computing results with tools of choice and then uploading the updated time series data and updating the metadata linking static context to those computed time series, possibly with provenance data.

#### 9.4.4. One datatype per variable

The current implementation assumes that each variable is associated with exactly one datatype, but the solution approach has no such limitation. In many cases, the single datatype assumption is correct for time series data. If the time series database is SQL, this assumption holds for those values. Industrial standards for information modelling (e.g. International Electrotechnical Commission (2020)) also tend to contain such constraints.

However, users may still construct queries that combine solution mappings that bind the same variable to values with two or more data types. This can for instance be done with the union-construction. In this case our implementation returns an error. In the future, we may choose to cast such data to their RDF representations as strings, and return a warning that such queries may reduce performance by slowing down the evaluation of expressions and increase memory use by foregoing native Apache Arrow data types.

#### 9.4.5. One time series database and one SQL table per datatype

The metadata annotating our static query graph currently assumes that there is only one time series database, and in the SQL case, that each datatype is associated with only one table. We have seen in Section 3 that hybrid approaches have been proposed with many different time series backends (e.g. HTTP: Petrova et al., 2019 and InfluxDB: Esnaola-Gonzalez & Diez, 2019). We plan to lift this limitation and introduce a way of specifying the time series database back-end and associated metadata (e.g. a particular table) in future releases.

Supporting multiple time series databases, while supporting access control in a uniform way as suggested in Section 9.3 could be a way of creating what are called "domain analytical data interfaces" in the Data Mesh Architecture, for which there are no mature implementations (Dehghani, 2022, p. 149, 168). The Data Mesh architecture is a proposal for an organisational and technical architecture to enable agile analytics at scale. A key feature of domain analytical interfaces in Data Mesh is to be free of vendor lock-in and provide a layer of semantic interoperability to enable other teams to understand and reuse the analytical data set. It is possible that the extensions proposed here can meet important requirements of analytical data products. We leave the exploration of this possibility to further work.

#### 9.4.6. No support for optimal sequencing of time series queries

Certain types of queries spanning a static context and time series database require carefully sequenced multiple roundtrips to be performant. In the process industry, analysts are sometimes interested in high resolution data that coincide with rare but important events. For instance, high resolution data on valve pressure (millisecond) around the time of valve adjustment is highly important to identify problems. Such a query fundamentally requires us to identify valves of interest, identify time periods associated with valve movement, and to retrieve high resolution pressure data from these time periods. Ideally then, we should first extract data on valve movement and use this data to constrain the amount of high resolution pressure data retrieved. A similar problem occurs for SPARQL engines implementing the SERVICE construction (cf. Buil-Aranda, Arenas, and Corcho (2011)), as evaluating a federated query with a loosely bound or free variable may be infeasible. We currently do not guarantee a particular execution sequence, which could lead to unpredictable behaviour. The feature can however be added in the combination phase by correspondingly choosing to process first the constituent graph pattern or associated time series query (in case we are processing a BGP with multiple time series) which contains the prioritised database or table.

## CRediT authorship contribution statement

**Magnus Bakken:** Conceptualization, Methodology, Software, Formal analysis, Investigation, Data curation, Writing – original draft, Visualization. **Ahmet Soylu:** Conceptualization, Writing – review & editing, Supervision.

## Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Magnus Bakken reports financial support was provided by Prediktor AS. Corresponding author member of Norwegian mirror committee of ISO/TC 10/SC 10 "Process plant documentation". https://www.standard.no/standardisering/komiteer/sn/snk-381/ The committee works on ISO/IEC 81346 for energy production. This standard is used in the running example of the paper.

## Data availability

Code and data available in repositories linked in paper.

## References


Abadi, D. J. (2008). *Query execution in column-oriented database systems* (Ph.D. thesis), Massachusetts Institute of Technology.

Abdelaziz, I., et al. (2017). A survey and experimental comparison of distributed SPARQL engines for very large RDF data. *Proceedings of the VLDB Endowment*, *10*(13), 2049–2060. http://dx.doi.org/10.14778/3151106.3151109.

Abel, F., et al. (2007). Enabling advanced and context-dependent access control in RDF stores. In *The semantic web* (pp. 1–14). Berlin, Heidelberg: Springer Berlin Heidelberg.

Ahmad, T. (2022). Benchmarking apache arrow flight-a wire-speed protocol for data transfer, querying and microservices. In *Benchmarking in the data center: expanding to the cloud* (pp. 1–10).

Albahli, S., & Melton, A. (2016). Rdf data management: A survey of rdbms-based approaches. In *Proceedings of the 6th international conference on web intelligence, mining and semantics* (pp. 1–4).

Albutiu, M.-C., Kemper, A., & Neumann, T. (2012). Massively parallel sort-merge joins in main memory multi-core database systems. arXiv preprint arXiv:1207.0145.

Ali, W., et al. (2021). A survey of RDF stores & SPARQL engines for querying knowledge graphs. *The VLDB Journal*, *31*(3), 1–26. http://dx.doi.org/10.1007/s00778-021-00711-3.

Alvanou, G., Lytra, I., & Petersen, N. (2018). An mtconnect ontology for semantic industrial machine sensor analytics. In *CEUR workshop proceedings*: *Vol. 2112*, *Joint proceedings of MEPDaW, SeWeBMeDA and SWeTI 2018* (pp. 56–62). CEUR-WS.org.

Armbrust, M., Ghodsi, A., Xin, R., & Zaharia, M. (2021). Lakehouse: a new generation of open platforms that unify data warehousing and advanced analytics. In *Proceedings of the conference on innovative data systems research (CIDR 2021)* (pp. 17:1–17:8).

AUCOTEC (2022). Cooperative platform engineering base. URL: https://www.aucotec.com/en/products/engineering-base/. [Online; accessed 3-October-2022].

Bader, A., Kopp, O., & Falkenthal, M. (2017). Survey and comparison of open source time series databases. In B. Mitschang, & et al. (Eds.), *Lecture notes in informatics (LNI)*: *P-266*, *Datenbanksysteme für business, technologie und web (BTW2017) – workshopband* (pp. 249–268). Gesellschaft für Informatik e.V. (GI).

Bakken, M. (2021). Quarry: An open source tool for OPC UA SPARQL queries over hybrid architectures using query rewriting. In *Proceedings of the 26th IEEE international conference on emerging technologies and factory automation (ETFA 2021)* (pp. 1–7). IEEE, http://dx.doi.org/10.1109/ETFA45728.2021.9613487.

Bartusiak, R. D., et al. (2022). Open process automation: A standards-based, open, secure, interoperable process control architecture. *Control Engineering Practice*, *121*, Article 105034. http://dx.doi.org/10.1016/j.conengprac.2021.105034.

Bast, H., & Buchhold, B. (2017). QLever: A query engine for efficient SPARQL+Text search. In *Proceedings of the 2017 ACM on conference on information and knowledge management* (pp. 647–656). New York, NY, USA: Association for Computing Machinery, http://dx.doi.org/10.1145/3132847.3132921.

Bian, H., & Ailamaki, A. (2022). Pixels: An efficient column store for cloud data lakes. In *2022 IEEE 38th international conference on data engineering* (pp. 3078–3090). IEEE.

Borrmann, A., König, M., Koch, C., & Beetz, J. (2018). Building information modeling: Why? what? how? In *Building information modeling* (pp. 1–24). Springer.

Botoeva, E., et al. (2018). A generalized framework for ontology-based data access. In *LNAI*: *Vol. 11298*, *Proceedings of the XVIIth international conference of the italian association for artificial intelligence (AI*IA 2018)* (pp. 166–180). Springer, http://dx.doi.org/10.1007/978-3-030-03840-3_13.

Brandt, S., et al. (2019). Two-dimensional rule language for querying sensor log data: a framework and use cases. In *Leibniz international proceedings in informatics (LIPIcs)*: *Vol. 147*, *Proceedings of the 26th international symposium on temporal representation and reasoning (TIME 2019)* (pp. 7:1–7:15). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, http://dx.doi.org/10.4230/LIPIcs.TIME.2019.7.

Bucchiarone, A., et al. (2019). Smart construction: Remote and adaptable management of construction sites through IoT. *IEEE Internet of Things Magazine*, *2*(3), 38–45. http://dx.doi.org/10.1109/IOTM.0001.1900044.

Buil-Aranda, C., Arenas, M., & Corcho, O. (2011). Semantics and optimization of the SPARQL 1.1 federation extension. In *LNCS*: *Vol. 6644*, *Proceedings of the 8th extended semantic web conference (ESWC 2011)* (pp. 1–15). Springer, http://dx.doi.org/10.1007/978-3-642-21064-8_1.

Calvanese, D., et al. (2017a). OBDA for log extraction in process mining. In *LNCS*: *Vol. 10370*, *Proceedings of the 13th international summer school (RW 2017)* (pp. 292–345). Springer, http://dx.doi.org/10.1007/978-3-319-61033-7_9.

Calvanese, D., et al. (2017b). Ontop: Answering SPARQL queries over relational databases. *Semantic Web*, *8*(3), 471–487. http://dx.doi.org/10.3233/SW-160217.

Cima, G., et al. (2020). Controlled query evaluation in ontology-based data access. In *LNCS*: *Vol. 12506*, *Proceedings of the 19th international semantic web conference (ISWC 2020)* (pp. 128–146). Springer, http://dx.doi.org/10.1007/978-3-030-62419-4_8.

Corcho, Ó., Priyatna, F., & Chaves-Fraga, D. (2020). Towards a new generation of ontology based data access. *Semantic Web*, *11*(1), 153–160. http://dx.doi.org/10.3233/SW-190384.

Dehghani, Z. (2022). *Data mesh: delivering data-driven value at scale*. O'Reilly Media.

Donkers, A., et al. (2021). Real-time building performance monitoring using semantic digital twins. In *Procedings of the 9th linked data in architecture and construction workshop (LDAC 2021)*.

Dremio (2022). Dremio | the easy and open data lakehouse platform. URL: https://www.dremio.com/. [Online; accessed 11-October-2022].

Erling, O. (2012). Virtuoso, a hybrid RDBMS/Graph column store. *IEEE Data Engineering Bulletin*, *35*(1), 3–8.

Esnaola-Gonzalez, I., & Diez, F. J. (2019). Integrating building and iot data in demand response solutions. In *CEUR workshop proceedings*: *Vol. 2389*, *Proceedings of the 7th linked data in architecture and construction workshop (LDAC 2019)* (pp. 92–105). CEUR-WS.org.

Giese, M., et al. (2015). Optique: Zooming in on big data. *Computer*, *48*(3), 60–67. http://dx.doi.org/10.1109/MC.2015.82.

Google Cloud Platform (2022). Access control with IAM. URL: https://cloud.google.com/bigquery/docs/access-control. [Online; accessed 24-October-2022].

Graube, M., Urbas, L., & Hladik, J. (2016). Integrating industrial middleware in linked data collaboration networks. In *Proceedings of the 21st international conference on emerging technologies and factory automation (ETFA 2016)* (pp. 1–8). IEEE.

Groppe, J., & Groppe, S. (2011). Parallelizing join computations of SPARQL queries for large semantic web databases. In *Proceedings of the 2011 ACM symposium on applied computing* (pp. 1681–1686).

Großmann, D., & Diedrich, C. (2022). Das NOA-informationsmodell: Vorstellung der NE 176. *Atp Magazin*, *64*(1–2).

Güzel Kalayci, E., et al. (2018). Ontop-temporal: A tool for ontology-based query answering over temporal data. In *Proceedings of the 27th ACM international conference on information and knowledge management (CIKM 2018)* (pp. 1927–1930). ACM, http://dx.doi.org/10.1145/3269206.3269230.

Herman, I., Fernández, S., Alonso, C. T., & Zakhlestin, A. (2022). RDFLib/sparqlwrapper: A wrapper for a remote SPARQL endpoint. URL: https://github.com/RDFLib/sparqlwrapper. [Online; accessed 24-October-2022].

Hernández, J. C. (2017). It can power a small nation. But this wind farm in China is mostly idle. *The New York Times*, URL: https://www.nytimes.com/2017/01/15/world/asia/china-gansu-wind-farm.html. [Online; accessed 3-October-2022].

Hitzler, P. (2021). A review of the semantic web field. *Communications of the ACM*, *64*(2), 76–83. http://dx.doi.org/10.1145/3397512.

Hu, S., et al. (2016). Building performance optimisation: A hybrid architecture for the integration of contextual information and time-series data. *Automation in Construction*, *70*, 51–61. http://dx.doi.org/10.1016/j.autcon.2016.05.018.

Huang, J., Abadi, D. J., & Ren, K. (2020). Scalable SPARQL querying of large RDF graphs. *Proceedings of the VLDB Endowment*, *4*(11), 1123–1134. http://dx.doi.org/10.14778/3402707.3402747.

InfluxData (2022a). InfluxDB: Open source time series database. URL: https://www.influxdata.com/products/influxdb-overview/. [Online; accessed 11-October-2022].

InfluxData (2022b). Welcome to InfluxDB IOx: InfluxData's New Storage Engine. URL: https://www.influxdata.com/blog/influxdb-engine/. [Online; accessed 13-December-2022].

International Electrotechnical Commission (2013). *Application integration at electric utilities - System interfaces for distribution management - Part 9: Interfaces for meter reading and control*: *Standard IEC 61968-9:2013*, Geneva, CH: International Electrotechnical Commission.

International Electrotechnical Commission (2017). *System control diagram*: *Publicly available specification IEC 63131:2017(E)*, Geneva, CH: International Electrotechnical Commission.

International Electrotechnical Commission (2020). *Communication networks and systems for power utility automation - Part 7-3: Basic communication structure - Common data classes*: *Standard IEC 61850-7-3:2010+A1:2020*, Geneva, CH: International Electrotechnical Commission.

International Organization for Standardization (2009). *Industrial systems, installations and equipment and industrial products–structuring principles and reference designations - Part 1: Basic rules: Standard IEC 81346-1:2009*, Geneva, CH: International Organization for Standardization.

International Organization for Standardization (2022). *Industrial systems, installations and equipment and industrial products – Structuring principles and reference designations – Part 10: Power supply systems: Standard IEC 81346-10:2022*, Geneva, CH: International Organization for Standardization.

Intizar Ali, M., et al. (2021). Cognitive digital twins for smart manufacturing. *IEEE Intelligent Systems*, *36*(2), 96–100. http://dx.doi.org/10.1109/MIS.2021.3062437.

Jin, G., Bian, H., Chen, Y., & Du, X. (2022). Columnar storage optimization and caching for data lakes. In *EDBT* (pp. 2–419).

Kanabar, M., McDonald, J., & Parikh, P. (2022). Grid innovations and digital transformation: Grid innovations and digital transformation of power substations are accelerating the energy transition for global utilities. *IEEE Power and Energy Magazine*, *20*(2), 83–95. http://dx.doi.org/10.1109/MPE.2022.3153784.

Keel Solution (2022). Keel solution - asset data management for the energy sector. URL: https://keelsolution.com/. [Online; accessed 3-October-2022].

Kharlamov, E., et al. (2014). How semantic technologies can enhance data access at siemens energy. In *LNCS*: *Vol. 8796*, *Proceedings of 13th international semantic web conference (ISWC 2014)* (pp. 601–619). Springer, http://dx.doi.org/10.1007/978-3-319-11964-9_38.

Kharlamov, E., et al. (2017). Semantic access to streaming and static data at siemens. *Journal of Web Semantics*, *44*, 54–74. http://dx.doi.org/10.1016/j.websem.2017.02.001.

Kim, D.-Y., et al. (2020). A modular factory testbed for the rapid reconfiguration of manufacturing systems. *Journal of Intelligent Manufacturing*, *31*(3), 661–680. http://dx.doi.org/10.1007/s10845-019-01471-2.

Kleppmann, M. (2017). *Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems*. O'Reilly Media.

Lee, Y. T. (1999). Information modeling: From design to implementation. In *Proceedings of the second world manufacturing congress* (pp. 315–321). ICSC Academic Press.

Lee, E. A. (2008). Cyber physical systems: Design challenges. In *2008 11th IEEE international symposium on object and component-oriented real-time distributed computing* (pp. 363–369). IEEE.

McKinney, W. (2010). Data structures for statistical computing in python. In *Proceedings of the 9th python in science conference* (pp. 56–61). SciPy, http://dx.doi.org/10.25080/Majora-92bf1922-00a.

McKinney, W. (2019). Introducing apache arrow flight: A framework for fast data transport. https://arrow.apache.org/blog/2019/10/13/introducing-arrow-flight/. [Online; accessed 29-September-2022].

Melnik, S., et al. (2010). Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, *3*(1–2), 330–339. http://dx.doi.org/10.14778/1920841.1920886.

MongoDB (2022). MongoDB: The developer data platform. URL: https://www.mongodb.com/. [Online; accessed 11-October-2022].

Mörzinger, B. (2019). *Accessing manufacturing data through virtual knowledge graphs: on the value and semantic peculiarities of time series data* (Ph.D. thesis), TU Wien.

Mörzinger, B., et al. (2018). A large-scale framework for storage, access and analysis of time series data in the manufacturing domain. *Procedia CIRP*, *67*, 595–600. http://dx.doi.org/10.1016/j.procir.2017.12.267.

MTConnect Institute (2022). MTConnect. URL: https://www.mtconnect.org//. [Online; accessed 12-October-2022].

Neal Richardson, et al. (2022). Working with arrow datasets and dplyr • arrow R package. URL: https://arrow.apache.org/docs/r/articles/dataset.html. [Online; accessed 13-December-2022].

Ontop VKG (2022). Ontop with dremio. URL: https://ontop-vkg.org/tutorial/federation/dremio/. [Online; accessed 13-December-2022].

OPC Foundation (2018a). *OPC 10000-11 unified architecture part 11 historical access*: *Technical report*, OPC Foundation, URL: https://reference.opcfoundation.org/v104/Core/docs/Part11/. [Online; accessed 3-October-2022].

OPC Foundation (2018b). *PC 10000-1 unified architecture part 1 overview and concepts server to interactions*: *Technical report*, OPC Foundation, URL: https://reference.opcfoundation.org/Core/Part1/6.3.7/. [Online; accessed 24-October-2022].

OPC Foundation (2022). *OPC 10000-1 UA part 1: Overview and concepts*: *Technical report*, OPC Foundation, URL: https://opcfoundation.org/developer-tools/documents/view/158. [Online; accessed 13-December-2022].

OSIsoft (2022). PI system - connecting data, operations & people. URL: https://www.osisoft.com/pi-system. [Online; accessed 12-December-2022].

Özçep, Ö. L., Möller, R., & Neuenstadt, C. (2014). A stream-temporal query language for ontology based data access. In *Proceedings of the 37th annual German conference on AI (KI 2014)* (pp. 183–194). Springer, http://dx.doi.org/10.1007/978-3-319-11206-0_18.

Padia, A., Finin, T., Joshi, A., et al. (2015). Attribute-based fine grained access control for triple stores. In *Proceedings of the 3rd society, privacy and the semantic web-policy and technology workshop co-organised with 14th international semantic web conference (ISWC 2015)*.

Pedone, G., & Mezgár, I. (2018). Model similarity evidence and interoperability affinity in cloud-ready industry 4.0 technologies. *Computers in Industry*, *100*, 278–286. http://dx.doi.org/10.1016/j.compind.2018.05.003.

Perzylo, A., et al. (2019). OPC UA NodeSet ontologies as a pillar of representing semantic digital twins of manufacturing resources. In *Proceedings of the 24th IEEE international conference on emerging technologies and factory automation (ETFA 2019)* (pp. 1085–1092). IEEE, http://dx.doi.org/10.1109/indin41052.2019.8972102.

Petrova, E., et al. (2019). In search of sustainable design patterns: Combining data mining and semantic data modelling on disparate building data. In *Proceedings of the 35th conference on IT in design, construction, and management (CIB W78 2018)* (pp. 19–26). Springer, http://dx.doi.org/10.1007/978-3-030-00220-6.

Pfaffel, S., Faulstich, S., & Rohrig, K. (2017). Performance and reliability of wind turbines: A review. *Energies*, *10*(11), http://dx.doi.org/10.3390/en10111904.

Poggi, A., Lembo, D., Calvanese, D., Giacomo, G. D., Lenzerini, M., & Rosati, R. (2008). Linking data to ontologies. In *Journal on data semantics X* (pp. 133–173). Springer.

Preimesberger, C. (2017). Dremio launches self-service data analytics for data scientists. URL: https://www.eweek.com/big-data-and-analytics/dremio-launches-self-service-data-analytics-for-data-scientists/. [Online; accessed 11-October-2022].

Przyjaciel-Zablocki, M., et al. (2013). Map-side merge joins for scalable SPARQL BGP processing. In *2013 IEEE 5th international conference on cloud computing technology and science, Vol. 1* (pp. 631–638). IEEE.

RDS 81346 Technique ApS (2022). RDS 81346-10 (RDS-PS) – A Wind Farm & A WTG. URL: https://www.81346.com/s/RDS-PS-Wind-Farm-Example-hk82.pdf. [Online; accessed 29-September-2022].

Schiekofer, R., & Weyrich, M. (2019). Querying OPC UA information models with SPARQL. In *Proceedings of the 24th IEEE international conference on emerging technologies and factory automation (ETFA 2019)* (pp. 208–215). IEEE, http://dx.doi.org/10.1109/indin41052.2019.8972102.

Schiekofer, R., et al. (2019). A formal mapping between OPC UA and the semantic web. In *Proceedings of the 17th international conference on industrial informatics (INDIN 2019)* (pp. 33–40). IEEE.

Schleipen, M., et al. (2016). OPC UA & industrie 4.0-enabling technology with high diversity and variability. *Procedia Cirp*, *57*, 315–320. http://dx.doi.org/10.1016/j.procir.2016.11.055.

Singh, G., Bhatia, S., & Mutharaju, R. (2020). OWL2bench: A benchmark for OWL 2 reasoners. In *LNCS*: *Vol. 12507*, *Proceedings of the 19th international semantic web conference (ISWC 2020)* (pp. 81–96). Springer, http://dx.doi.org/10.1007/978-3-030-62466-8_6.

Skjæveland, M. G. (2022). Terse syntax for reasonable ontology templates (stOTTR). URL: https://dev.spec.ottr.xyz/stOTTR/. [Online; accessed 30-September-2022].

Skjæveland, M. G., et al. (2021). OTTR: Formal templates for pattern-based ontology engineering. In *Proceedings of the workshop on ontology design and patterns* (pp. 349–377). IOS Press, http://dx.doi.org/10.3233/SSW210025.

Soylu, A., et al. (2018). OptiqueVQS: A visual query system over ontologies for industry. *Semantic Web*, *9*(5), 627–660. http://dx.doi.org/10.3233/SW-180293.

Soylu, A., et al. (2022). TheyBuyForYou platform and knowledge graph: Expanding horizons in public procurement with open linked data. *Semantic Web*, *13*(2), 265–291. http://dx.doi.org/10.3233/SW-210442.

Steindl, G., Frühwirth, T., & Kastner, W. (2019). Ontology-based OPC UA data access via custom property functions. In *Proceedings of the 24th IEEE international conference on emerging technologies and factory automation (ETFA 2019)* (pp. 95–101). IEEE, http://dx.doi.org/10.1109/ETFA.2019.8869436.

Swartz, A., et al. (2020). RDFLib. URL: https://github.com/RDFLib/rdflib. [Online; accessed 29-September-2022].

Tang, S., et al. (2019). A review of building information modeling (BIM) and the internet of things (IoT) devices integration: Present status and future trends. *Automation in Construction*, *101*, 127–139. http://dx.doi.org/10.1016/j.autcon.2019.01.020.

Tanon, T. (2022a). Oxrdf. URL: https://github.com/oxigraph/oxigraph/tree/main/lib/oxrdf. [Online; accessed 12-December-2022].

Tanon, T. (2022b). Spargebra. URL: https://github.com/oxigraph/oxigraph/tree/main/lib/spargebra. [Online; accessed 29-September-2022].

Tantik, E., & Anderl, R. (2017). Integrated data model and structure for the asset administration shell in industrie 4.0. *Procedia Cirp*, *60*, 86–91.

The Apache Software Foundation (2022a). Apache arrow. URL: https://arrow.apache.org/. [Online; accessed 29-September-2022].

The Apache Software Foundation (2022b). Apache parquet. https://parquet.apache.org/. [Online; accessed 20-November-2022].

The Apache Software Foundation (2022c). Apache spark™ - unified engine for large-scale data analytics. URL: https://spark.apache.org/. [Online; accessed 11-October-2022].

The Apache Software Foundation (2022d). DataFrame API — Arrow DataFusion documentation. URL: https://arrow.apache.org/datafusion/user-guide/dataframe.html. [Online; accessed 13-December-2022].

The Apache Software Foundation (2022e). Pyarrow - apache arrow python bindings. URL: https://arrow.apache.org/docs/python/index.html. [Online; accessed 29-September-2022].

The Modbus Organization (2012). MODBUS application protocol specification V1.1b3. URL: https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. [Online; accessed 17-October-2022].

The pandas development team (2020). Pandas-dev/pandas: Pandas. http://dx.doi.org/10.5281/zenodo.3509134.

Van Der Aalst, W. (2012). Process mining. *Communications of the ACM*, *55*(8), 76–83. http://dx.doi.org/10.1145/2240236.2240257.

van Gool, S., Yang, D., & Pauwels, P. (2021). Integrating sensor and building data flows: a case study of the IEQ of an office building in the netherlands. In *Proceedings of the 13th European conference on product and process modeling (ECPPM 2020-2021)* (pp. 328–333). CRC Press.

Villalobos, K., et al. (2020). A three level hierarchical architecture for an efficient storage of industry 4.0 data. *Computers in Industry*, *121*, Article 103257. http://dx.doi.org/10.1016/j.compind.2020.103257.

Vink, R. (2022a). perf(rust): sort and unsort join key if other side is sorted #5069. URL: https://github.com/pola-rs/polars/pull/5069. [Online; accessed 15-December-2022].

Vink, R. (2022b). Polars. URL: https://www.pola.rs/. [Online; accessed 3-October-2022].

Weintraub, G., Gudes, E., & Dolev, S. (2021). Needle in a haystack queries in cloud data lakes. In *EDBT/ICDT workshops*.

World Wide Web Consortium (2013a). *SPARQL 1.1 federated query*: *Technical report*, World Wide Web Consortium, URL: https://www.w3.org/TR/sparql11-federated-query/.

World Wide Web Consortium (2013b). *SPARQL 1.1 protocol*: *Technical report*, World Wide Web Consortium, URL: https://www.w3.org/TR/sparql11-protocol/.

World Wide Web Consortium (2013c). *SPARQL 1.1 query language*: *Technical report*, World Wide Web Consortium, URL: https://www.w3.org/TR/sparql11-query/.

Xiao, G., et al. (2018). Ontology-based data access: A survey. In *Proceedings of the twenty-seventh international joint conference on artificial intelligence (IJCAI 2018)* (pp. 5511–5519). http://dx.doi.org/10.24963/ijcai.2018/777.

Xiao, G., et al. (2019). Virtual knowledge graphs: An overview of systems and use cases. *Data Intelligence*, *1*(3), 201–223. http://dx.doi.org/10.1162/dint_a_00011.

Xiao, G., et al. (2020). The virtual knowledge graph system ontop. In *LNCS*: *Vol. 12507*, *Proceedings of the 19th international semantic web conference (ISWC 2020)* (pp. 259–277). Springer, http://dx.doi.org/10.1007/978-3-030-62466-8_17.

Zheng, Z., et al. (2022a). Executable knowledge graphs for machine learning: A bosch case of welding monitoring. In *LNCS*: *Vol. 13489*, *Proceedings of the 21st international semantic web conference (ISWC 2022)* (pp. 791–809). Springer, http://dx.doi.org/10.1007/978-3-031-19433-7_45.

Zheng, Z., et al. (2022b). ExeKG: Executable knowledge graph system for user-friendly data analytics. In *Proceedings of the 31st ACM international conference on information & knowledge management (CIKM 2022)* (pp. 5064–5068). ACM, http://dx.doi.org/10.1145/3511808.3557195.

Zheng, Z., et al. (2022c). Towards a statistic ontology for data analysis in smart manufacturing. In *CEUR workshop proceedings*: *Vol. 3254*, *Proceedings of the ISWC 2022 posters, demos and industry tracks*. CEUR-WS.org.