The Dow Jones Knowledge Graph

Ian Horrocks[1], Jordi Olivares[2], Valerio Cocchi[3], Boris Motik[1], and Dylan Roy[2]

[1] University of Oxford
[2] Dow Jones
[3] Oxford Semantic Technologies

Abstract. Dow Jones is a leading provider of market, industry and portfolio intelligence serving a wide range of financial applications including asset management, trading, analysis and bankruptcy/restructuring. The information needed to provide such intelligence comes from a variety of heterogeneous data sources. Integrating this information and answering complex queries over it presents both conceptual and computational challenges. In order to address these challenges Dow Jones have used the RDFox system to integrate the various sources in a large RDF knowledge graph. The knowledge graph is being used to power an expanding range of internal processes and market intelligence products.

1 Background and Motivation

Dow Jones is a leading provider of market, industry and portfolio intelligence serving a wide range of financial applications including asset management, trading, analysis and bankruptcy/restructuring.[4] Dow Jones supports businesses, governments and financial institutions with award-winning journalism, deep content archiving and indexing, robust data sets, and flexible information tools; it provides a portfolio of information solutions covering diverse customer needs including uncovering market advantage, integrating data into workflows and managing risk. The goal of Dow Jones is to deliver trusted news and data that can help businesses and society to make better decisions.

High quality market intelligence is critical to corporate decision making. For example, decision makers in a given company may need to be alerted to news items about competitor companies that operate in a related sector to themselves or one of their subsidiaries. The information needed to answer such questions can come from a wide range of heterogeneous sources, including structured sources such as company data and financial data, and unstructured sources such as news feeds. This data needs to be integrated so as to allow for suitable queries to be formulated across multiple sources. This can be very challenging: even if the data from all sources is loaded into a single database, the resulting schema can be very complex, and formulating suitable queries can be very difficult, requiring a combination of knowledge and expertise in the domain, the data sources and the query language. Moreover, the resulting queries can be computationally challenging for a typical database system.

[4] https://www.dowjones.com/

The solution adopted at Dow Jones is to use relevant information from multiple sources to construct a large RDF knowledge graph. This is achieved by using the standard direct mapping to transform structured sources into RDF triples [3], and by using an NLP process to extract relevant facts from news feed articles and transform them into triples. Fast loading and updating of the graph is critical to the feasibility of this approach: the graph currently consists of approximately 2.6 billion triples, and while some of these are derived from relatively static data sources (such as company data) others come from rapidly changing sources (such as news feeds).

The knowledge graph will power a wide range of internal processes and market intelligence products, with knowledge from the graph being accessed via SPARQL queries [6]. For example, Dow Jones researchers maintain data about competitor relationships between companies, and to support this they use an in-house tool to explore information about companies and their relationships retrieved from the graph via back-end SPARQL queries. Using SPARQL queries over the knowledge graph provides applications with both power and flexibility, but it means that fast SPARQL query answering is critical in order to provide acceptable response times in applications.

In order to meet these requirements, Dow Jones has chosen to use the RDFox knowledge graph system. RDFox provides fast parallelised data loading and is able to load all 2.6 billion triples in approximately 26 minutes using only a relatively modest 4 vCPU server; it also supports incremental updates, and can add/delete several thousand triples per second. RDFox also provides a highly optimised SPARQL engine which not only exploits novel in-memory data structures but also employs sideways information passing to optimise complex SPARQL queries; as a result, typical queries can be answered in milliseconds, and even hard "stress-test" queries can be answered in only a few seconds.

In the remainder of the paper we will provide more details about the construction of the knowledge graph and how it is used in applications (Section 2); review the relevant features of RDFox, and in particular the data loading and query answering capabilities that are critical in this setting (Section 3); present some data on system performance (Section 4); and discuss future plans for extending the system and its application (Section 5).

2 Knowledge Graph Construction and Applications

2.1 Knowledge Graph Construction

The knowledge graph integrates data from a wide range of sources that are maintained by and hosted in various different parts of the company. The majority of the data comes from the following sources:

– *Basic information about companies.* This is stored in a relational database and consists of basic information about more than 70 million companies including name, address, normalized code for the region of the address, industry codes (NAICS, SIC, NACE), other identifiers (such as DUNS or LEI), and other name aliases that might be found in news feeds.

- *Company hierarchy information.* This is stored in a relational database and consists of information that links a DUNS coded company to its parent company, forming a company hierarchy graph in relational form.
- *Executives.* This is stored in a relational database and consists of information about more than 140 million company executives including their name, the companies that they are associated with and the roles that they play in these companies.
- *Stock information about companies* This is stored in a relational database and consists of information about approximately 100,000 company stock market listings including their stock ticker (a unique identifier assigned to each security traded on a particular market), whether this is the main listing or not, and in which stock exchanges they are listed.
- *Stock exchanges.* This is stored in a CSV file and consists of information about stock exchanges including their name, location and relationship to other exchanges. The data is publicly available and can be accessed from https://www.iso20022.org/market-identifier-codes.
- *Geonames.* This is a public domain geographical names database derived from official public sources, and extended and improved via crowdsourcing. It consists of information about more than 25 million locations, including name variants, latitude, longitude, elevation, population, etc. The data is already available as RDF and can be accessed via https://www.geonames.org/.

The relational data sources are transformed into RDF triples via the standard Direct Mapping of Relational Data to RDF [3]. A similar process is used to transform the CSV data into RDF. The Geonames data is already in RDF form. Integration of this data is relatively straightforward as the structured sources are well curated, and include industry standard identifiers such as ticker symbols, DUNS numbers, and NAICS codes. Some cleanup of "messy" identifiers may be required in the future if other data sources are added, but this is not currently an issue for Dow Jones.

The above sources yield a total of approximately 2.3 billion triples, and constitute about 90% of the triples in the knowledge graph. These sources are continuously curated and updated, but the rate of change is relatively low, and in order to simplify the system architecture the whole ETL process is simply repeated once per month.

In addition to this relatively static data, the graph also includes data extracted from financial news articles from several sources:

- Articles from Dow Jones publications including the Wall Street Journal,[5] Market Watch[6] and Baron's Magazine.[7] Approximately 7–10 thousand such articles are available at any time, and this set is constantly changing as old articles are deleted, new articles are added and existing articles are edited.

[5] https://www.wsj.com/
[6] https://www.marketwatch.com/
[7] https://www.barrons.com/

– Articles from the Dow Jones Factiva feed.[8] Approximately 150–250 thousand such articles are available at any time, and like the Dow Jones articles the set of available articles is constantly changing.

Each available Dow Jones article is represented by an entity in the knowledge graph, along with meta-data such as its title, news topics, and companies and regions mentioned. Some of this meta-data is available directly, but some, such as companies and regions mentioned, must be extracted from the text. This is done using a custom NLP process that extracts not only this kind of meta-data, but also so called *signals* that indicate relevant events such as earnings announcements, initial public offerings (IPOs), acquisitions, mergers and Chapter 11 bankruptcy filings. Each such signal is also represented by an entity in the graph. The data extraction process exploits domain knowledge stored in the graph and uses it to identify target entities (such as companies and regions), and is designed so as to be easily adaptable to capture any kind of entity or signal that might be of interest to Dow Jones customers, and that might help them to identify relevant news content. Article and signal meta-data is stored as triples in the knowledge graph associated with the relevant article and signal entities; additional triples link signal entities to relevant articles, companies, regions, etc.

Articles from the Factiva feed are processed in the same way, but due to the very large number of such articles they are only stored in the graph if they are found to contain relevant entities or signals.

The above process typically yields in the range of 4-5 thousand new signals each day, amounting to approximately 30–40 thousand triples. These are added to the knowledge graph incrementally, which takes only a few seconds. At the same time, triples relating to older articles that have been deleted from the relevant news-feeds are removed from the knowledge graph; this is again done incrementally, and again requires only a few seconds.

When signals are first added to the knowledge graph they are marked as "potential" by adding a suitable triple to the signal entity. Potential signals are checked and curated by human experts, and if confirmed the "potential" triple is deleted; otherwise the whole signal is deleted. These deletions are again achieved via incremental updates; such updates involve deleting only a small number of triples, which typically requires only a few milliseconds.

Finally, Dow Jones also maintains data about competitor relationships between companies. This data is actively curated on a continuous basis using an in-house tool that exploits knowledge graph queries to identify and analyse possible competitors. The resulting competitor relationship data is stored back into the knowledge graph. This is again realised via incremental updates; as in the case of signal curation, the number of triples involved in each update is relatively small and such updates can be performed in only a few milliseconds.

When all these sources are loaded into RDFox the resulting knowledge graph contains approximately 2.6 billion triples.

The resulting graph structure is very simple. The Direct Mapping of relational sources produces a structure that directly mimics the source tables; ar-

[8] https://factiva.com/

ticles and signals are represented by single entities, with attached (meta-) data triples; and triples are used to link signals, articles and other entities in the graph. Dow Jones have chosen to use the W3C Shape Expressions Language (ShEx) to describe this structure [18]. This could in principle be used for data validation, but it is used at Dow Jones simply to document the graph structure. Dow Jones application developers and knowledge engineers use the ShEx schema to help them to write queries, and they chose ShEx over SHACL [19] for this purpose because they find ShEx syntax to be more natural and easier to understand. For example, the following extract specifies the graph structure of stock listings:

```
cande-shex:StockListing {
 a [ cande:StockListing ] ;
 cande:lists_company IRI // orm:continuation cande-shex:Company ;
 cande:has_ticker_symbol xsd:string ;
 cco:designated_by @cande-shex:StockListingIdentifier * ;
 cande:listed_in IRI // orm:continuation cande-shex:StockExchange ;
 cande:is_primary_listing xsd:boolean ;
}

cande-shex:StockListingIdentifier {
 a [ cande:SEDOL cande:ISIN cande:CUSIP ] ;
 common:id_literal xsd:string ;
}
```

From this the developers and engineers can quickly identify the relevant predicates for accessing information about stock listings, e.g., they can access the ticker symbol via the `cande:has_ticker_symbol` predicate, and for navigating to other entities, e.g., they can navigate to the relevant company entity via the `cande:lists_company` predicate; moreover, they can see that the structure of companies is specified by the shape expression `cande-shex:Company`.

As mentioned above, ShEx could in principle be used for data validation, but it is not supported by RDFox. However, it would be an easy matter to translate ShEx into SHACL, which is supported by RDFox, if data validation were required.

2.2 Knowledge Graph Applications

The knowledge graph can be used to answer questions that would be difficult or impossible to answer without integrating multiple data sources. For example, given a company C specified by `<companyIri>`, the following query Q_1 retrieves competitor companies that are listed in the stock exchange and are in the same or related sector as C or that are in the exact same sector as one of C's direct subsidiaries:

```
SELECT DISTINCT ?competitor ?industryCode ?industryCodeType
WHERE {
  BIND(?company AS <companyIri>)
  {
    ?company cande:has_industry_code/skos:relatedMatch/^skos:relatedMatch
```

```
      ?industry .
  } UNION {
    [] cco:is_subsidiary_of ?company ;
       cande:has_industry_code ?industry .
  }

  ?industry a ?industryCodeType ;
              cande:has_id ?industryCode .
  FILTER(?industryCodeType IN (djid:DJIDCode, djn:DJNCode, naics2017:NAICSCode))
  ?competitor cande:has_industry_code ?industry .
  [] cande:lists_company ?competitor ;
}
```

Answering this query requires integrating basic company data, company hierarchy data, competitor relationships data and stock listings data. Such queries can be relatively easily constructed by consulting the ShEx specification outlined in Section 2.1 above.

The knowledge graph can power a wide range of internal processes and market intelligence products. One such internal process is the construction of the knowledge graph itself, and in particular the extraction of signals from news articles. Here the knowledge graph is used to support validation and disambiguation; for example, if the we find a potential signal of the form A is buying B, then A and B should both be companies, and should be identified with specific companies represented in the knowledge graph.

Another example is the identification of competitor relationships between companies. As already mentioned in Section 2.1, data about competitor relationships is stored in the knowledge graph and is presented to customers in "quote pages" which provide detailed information about given companies. Dow Jones researchers continuously curate this competitor data using a tool that supports identifying and exploring possible competitors. Users can specify a range of different search parameters and filters, and these are converted into SPARQL queries over the knowledge graph which return (details about) relevant companies; see, for example, query Q_1 above. Queries are constructed using templates whose slots can be filled with values derived from the user-specified search and filter parameters; in the case of Q_1, the company of interest can be specified in `<companyIri>`. The system is designed so that it is easy to add new parameters, filters and query templates as needed to meet user requirements.

An example of a product in which the knowledge graph will be used is the Wall Street Journal (WSJ) Bankruptcy Pro.[9] This product provides a searchable archive of relevant articles, and supplements articles with important data such as competitor analyses, risk factor identification, capital structure, credit ratings and recent filings. Users can specify a range of different search parameters and filters, and these will be converted into SPARQL queries over the knowledge graph which return (pointers to) relevant articles and data. Queries will be constructed using templates in the same way as for the competitor research tool described above.

[9] https://wsjpro.com/

The knowledge graph will also enable a range of new and more powerful applications that are currently under development including, e.g., personalised recommendations for customers, including recommending relevant authors and news articles, analysis of investment risk factors, and checks on regulatory compliance. Many of these applications will involve heavy use of RDFox's reasoning capabilities.

3 RDFox

As we have seen in Sections 2.1 and 2.2, construction and maintenance of the knowledge graph depends on fast loading and updating of triples, while applications of the knowledge graph depend on fast responses to SPARQL queries. These were the main considerations that led Dow Jones to select the RDFox system.

RDFox is a high performance knowledge graph and semantic reasoning engine. Originally the result of research at the University of Oxford [14], RDFox is now developed and marketed by Oxford Semantic Technologies.[10] RDFox exploits a patented in-memory architecture and parallelised computation to provide high performance for data loading, reasoning and query answering. Key features of RDFox include:[11]

- RDF triples, rules, and OWL 2 [17] and SWRL [8] axioms can be imported either programmatically or from files in a range of formats including turtle, datalog and OWL. RDF data can also be validated using the SHACL constraint language.
- Information can be accessed directly from external data sources, such as CSV files, relational databases, and Apache Solr.[12]
- Triples, rules and axioms can be exported into a number of different formats, and the contents of the system can also be (incrementally) saved into a binary file, which can later be used to restore the system's state.
- Multi-user support with ACID transactional updates [5].
- Access control allows for individual information elements in the system to be assigned different access permissions for different users.
- Full support for SPARQL 1.1, and functionality for monitoring query answering and accessing query plans.
- Materialization-based reasoning, where all triples that logically follow from the triples and rules in the system are materialized as new triples.
- Incremental update of materialized graphs: reasoning does not need to be performed from scratch when the information in the system is updated.
- Explanation of reasoning results: RDFox is able to return a proof for any new fact added to the store through materialization.

[10] https://www.oxfordsemantic.tech/

[11] See https://arxiv.org/pdf/2102.13027.pdf for a survey of RDF stores and their features.

[12] https://solr.apache.org/

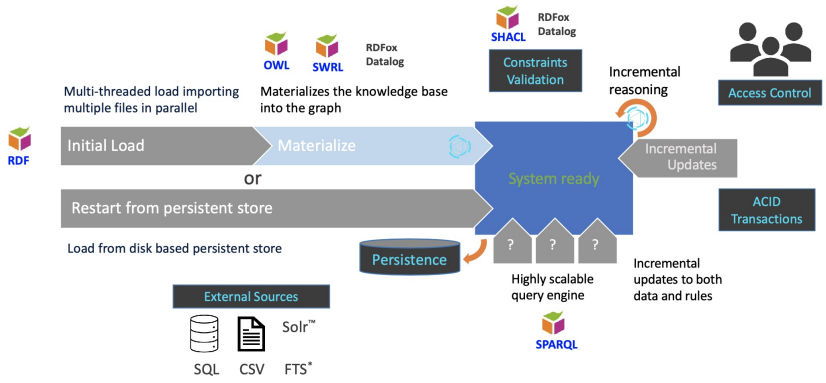

Fig. 1. RDFox Architecture

Figure 1 illustrates RDFox's basic features and functionality. At startup, RDFox can load data, rules, axioms and constraints in a range of different formats as described above. It is also possible to import data directly from external legacy sources including relational databases, CSV files and Apache Solr. Alternatively, the system can be restored from a previous state saved in a binary file. An important feature in the Dow Jones application is that RDFox can import multiple sources in parallel, and we will discuss this in more detail below.

After loading, RDFox performs materialization-based reasoning and constraint validation using a parallelized variant of the seminaïve algorithm [1, 13] (see Section 3.1). Once the initial materialization process is complete the store is ready for subsequent operations including querying and incremental updating. Access control and ACID transactions allow for control over user access to data and ensure predictable behaviour when multiple users are updating the store. The state of the system can also be saved in a binary file for subsequent reloading.

Incremental updates can include deletion and addition of data, and also deletion and addition of rules, axioms and constraints. RDFox deals with such updates using FBF, a novel extension of the delete and rederive (DRed) view maintenance algorithm that avoids excessive overdeletion [11, 12]. Like data loading, incremental updates are parallelized for improved performance. RDFox uses a highly optimised SPARQL engine with sideways information passing; this is another important feature in the Dow Jones application that we will discuss in more detail below. Each query is evaluated on a single thread, but multiple queries can be evaluated in parallel using multiple threads.

3.1 Parallelized Materialization

As already mentioned, RDFox materializes all implied triples using a parallelized variant of the seminaïve algorithm [1, 13]. The triples that make up the RDF

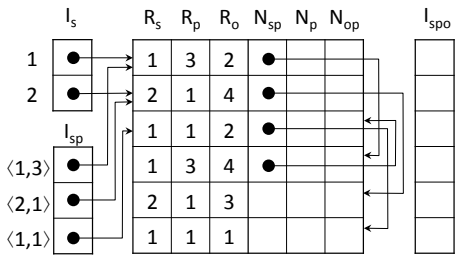

Fig. 2. Data Structure for Storing RDF Triples

graph are stored in a table. The triples are considered one at a time and matched to the rules, with parallelization being achieved by assigning triples to available threads. For example, given the following rules

$$\langle ?x, C, ?y\rangle \wedge \langle ?y, E, ?z\rangle \rightarrow \langle ?x, D, ?z\rangle \tag{R1}$$

$$\langle ?x, D, ?y\rangle \wedge \langle ?y, E, ?z\rangle \rightarrow \langle ?x, C, ?z\rangle \tag{R2}$$

and a triple $\langle a, E, b\rangle$, a thread will match it to the triple $E(?y, ?z)$ in rule (R1) and evaluate subquery $\langle ?x, C, a\rangle$ to derive triples of the form $\langle ?x, D, b\rangle$, and it will handle rule (R2) analogously. We thus obtain independent subqueries, each of which is evaluated on a distinct thread. The difference in subquery evaluation times is irrelevant because of the large number of queries (i.e., proportional to the number of triples) so threads are fully loaded.

A naïve application of this idea would be inefficient: if we have triples $\langle a, C, b\rangle$ and $\langle b, E, c\rangle$, then we would derive the triple $\langle a, D, c\rangle$ twice—that is, we would consider the same rule instance twice. To address this source of inefficiency, the seminaïve algorithm evaluates subqueries only over the triples that appear *before* the triple being processed. For example, if $\langle a, C, b\rangle$ is processed first, then $\langle b, E, c\rangle$ will not be visible to the subquery and $\langle a, D, c\rangle$ will not be derived; however, when $\langle b, E, c\rangle$ is processed, $\langle a, C, b\rangle$ will be visible to the subquery and $\langle a, D, c\rangle$ will be derived.

To support this idea in practice, RDFox uses patented data structures that support both efficient evaluation of subqueries and efficient parallel updates [13, 9]. Like systems such as Hexastore [20] and RDF-3X [16], RDFox maintains indexes over stored triples to support efficient (sub)query evaluation; RDFox, however, uses hash-based indexes that allow for efficient 'mostly' *lock-free* parallel updates [7]: most of the time, at least one thread is guaranteed to make progress regardless of the remaining threads.

RDFox stores triples in a six-column *triple table* as shown in Figure 2. As usual in RDF systems, resources are encoded as integer IDs using a dictionary, with IDs produced by a counting sequence so they can be used as array indexes. Columns R_s, R_p, and R_o contain the integer encodings of the subject, predicate, and object of each triple. Each triple participates in three linked lists: an *sp*-list connects all triples with the same R_s grouped (but not necessarily sorted) by

R_p, an *op*-list connects all triples with the same R_o grouped by R_p, and a *p*-list connects all triples with the same R_p without any grouping; columns N_{sp}, N_{op}, and N_p contain the next-pointers. Triple pointers are implemented as offsets into the triple table.

RDFox maintains various indexes to support matching triples with different binding patterns (i.e., different configurations of variables in the triple). For example, index I_s maps each s to the head $I_s[s]$ of the respective *sp*-list; to match a triple $\langle s, ?y, ?z \rangle$ in I, we look up $I_s[s]$ and traverse the *sp*-list to its end; if $?y = ?z$, we skip triples with $R_p \neq R_o$. Index I_{sp} maps each s and p to the first triple $I_{sp}[s, p]$ in an *sp*-list with $R_s = s$ and $R_p = p$; to match a triple $\langle s, p, ?z \rangle$ in I, we look up $I_{sp}[s, p]$ and traverse the *sp*-list to its end or until we encounter a triple with $R_p \neq p$. Index I_{spo} contains each triple in the table, and so it can match fully specified triples $\langle s, p, o \rangle$. Other indexes include I_p, and I_o and I_{op}. Indexes I_s, I_p, and I_o are realised as arrays indexed by resource IDs. Indexes I_{sp}, I_{op}, and I_{spo} are realised as open addressing hash tables storing triple pointers.

Lock-freedom is achieved using compare-and-set (CAS) instructions: $\mathsf{CAS}(loc, exp, new)$ loads the value stored at location *loc* into a temporary variable *old*, stores the value of *new* into *loc* if $old = exp$, and returns *old*; hardware ensures that all steps are atomic (i.e., without interference). CAS can be used directly to update the linked lists in the triple table. For example, if thread T^1 has added a triple $\langle 1, 3, 6 \rangle$ to the table and is trying to add it to the N_{sp} list after the triple $\langle 1, 3, 2 \rangle$, then T^1 will set the N_{sp} pointer of the $\langle 1, 3, 6 \rangle$ entry to point to the N_{sp} pointer from the $\langle 1, 3, 2 \rangle$ entry and will use a CAS instructions to try to set the N_{sp} pointer from the $\langle 1, 3, 2 \rangle$ entry to point to the $\langle 1, 3, 6 \rangle$ entry; if the CAS instruction fails, then some other thread must have changed the N_{sp} pointer, in which case T^1 repeats the insertion procedure.

The process of adding a new triple to the table is more complex as one must atomically query I_{spo} (to check for duplicates), add the triple to the table, and update I_{spo}. To do this, RDFox implements a form of localised locking: if a thread does not find the new triple in I_{spo}, then it identifies a suitable empty bucket and tries to lock it by using a CAS instruction to store a special marker in the bucket. If this fails then some other thread may have already inserted the same triple, and so the whole operation is repeated beginning with the query to I_{spo}. If the CAS instruction succeeds, then we can add the new triple to the table, store it in the bucket (effectively releasing the localised lock), and then update all remaining indexes. In the meantime, we make sure that other threads do not skip over the bucket until the marker is removed.

3.2 Parallelized data loading

Although originally designed to support parallelized materialization, the lock-free data structures described in Section 3.1 also allow for the parallelization of data loading. This can be achieved simply by assigning a thread to each data source to be loaded. Each thread can then add triples to the triple table in the same way as the multiple threads used for materialization.

Additionally, when data is being loaded from files containing RDF triples in turtle format, each file can use one thread for parsing and multiple threads for adding parsed triples to the triple table. Parsing is single threaded because the syntax of IRIs makes it difficult to parallelize, and in any case parsing is typically much faster than adding triples to the data structures, so a single parser thread can keep several data addition threads fully occupied. If the data is split into multiple files, then these can be loaded in parallel using multiple threads.

3.3 SPARQL query answering

The indexed triple table described in Section 3.1 is designed to support efficient (sub)query evaluation during materialisation and so already supports efficient join evaluation in SPARQL query answering. However, SPARQL queries can be (heavily) nested; i.e., the outer level query can have sub-queries as components. A simple example is a query $Q = Q1$ MINUS $Q2$. In this case query Q is made up of two sub-queries $Q1$ and $Q2$, with the answer to Q being the answer to $Q1$ minus the answer to $Q2$. Note that $Q1$ and $Q2$ could themselves contain sub-queries and that this nesting of queries can continue to arbitrary depth. In order to make query answering be more efficient and to use less memory we want to evaluate the query "top-down", that is, starting with the outer level queries and working inwards. In our example, a naïve "bottom-up" method would compute the answers to $Q1$ and $Q2$, and then subtract the answer to $Q2$ from the answer to $Q1$; however, this would require computing and storing the full answers to both sub-queries. In our "top-down" method we would iterate through the answers to $Q1$, and for each such answer we would check if it is also an answer to $Q2$, retaining it as an answer to Q only if it is not an answer to $Q2$. This requires very little storage, and only requires us to check $Q2$ for tuples that we already identified as answers to $Q1$. This technique is known in the literature as Sideways Information Passing (SIP) [1]; in our example, information about answers to $Q1$ is passed "sideways" to $Q2$.

The above example is relatively simple, but SPARQL is a large language containing many operators for modifying and combining queries (Filter, Bind, And, Union, Minus, Distinct, Project, etc.) as well as a large number of built-in functions for manipulating values including, e.g., arithmetic functions (plus, minus, etc.), aggregation (sum, max, min, etc.) and string manipulation (concatenate, sub-string, etc.). It is extremely challenging to design a SPARQL query evaluation algorithm that maximises the efficiency benefits of SIP while at the same time guarantees to conform to the SPARQL semantics, i.e., to compute the same answers as would be computed by a naïve bottom-up method. Neumann and Weikum presented a SIP algorithm for basic SPARQL pattern matching queries [15], but this doesn't consider nested queries using some or all of the above mentioned features. RDFox uses a patented algorithm that extends SIP optimisation to arbitrary queries by compiling the query into a tree and introducing variable normalisation and expansion nodes as needed to ensure safe application of SIP [10]. The combination of SIP and the optimised data structures discussed in Section 3.1 allow for extremely efficient evaluation of SPARQL queries: most

queries used in applications of the Dow Jones knowledge graph can be answered in only a few milliseconds, and even the most complex queries require only a few seconds (see Section 4).

4 Performance

In this section we present some performance data for RDFox using both standard benchmarks and the Dow Jones knowledge graph.

4.1 Test Data and Environments

For standard benchmarks we used both LUBM and WatDiv [4, 2]. We used a version of LUBM with 10,000 universities (LUBM-10k), which comprises approximately 1.3 billion asserted triples, with a further approximately 0.5 billion triples added via materialisation of (rules derived from) the LUBM ontology; the graph for query answering therefore comprises approximately 1.8 billion triples. We used WatDiv 100M, which comprises approximately 150 million asserted triples; WatDiv does not have an ontology. Each benchmark comes with a standard set of test queries. These tests used RDFox 5.4.0 running on a c5.18xlarge AWS instance with 3.0 GHz Intel Xeon processors, 72 vCPUs and 144 GiB of RAM.

For the knowledge graph tests we used the Dow Jones Knowledge Graph (DJKG) described in Section 2.1, which comprises approximately 2.6 billion triples, and a set of three test queries:

Q1 retrieves all the signals and their properties that were derived from an English language article that was published between 2020-05-24 and 2020-05-26, and that talks about either Africa or North America.
Q2 retrieves all the signals and their properties that were derived from an English language article that was published between 2020-01-01 and 2020-09-28, and that talks about a company with a given identifier.
Q3 retrieves the number of different companies in the knowledge graph grouped by identifier type, industry, and country.

Q1 and Q2 are typical application queries; Q3 is not a realistic application query but is designed to stress-test SPARQL query engines. The SPARQL for these queries is too verbose to be given here, but they are available at https://bit.ly/3qGJS9I along with all non-confidential data. These tests used RDFox 4.0.0 running on a Google Cloud N1 with 4 vCPUs and 125 GB of RAM.

4.2 Data Loading

The data was split into multiple files to facilitate parallel loading. In the case of WatDiv and LUBM the data was split into 72 files and loaded using 72 threads; in the case of DJKG the data was split into 4 files and loaded using 4 threads. Table 1 shows the loading time for the three data sets (Time) as well as the

Table 1. Data loading times

Dataset	Time	Threads	T/s	T/T/s	Speedup
DJKB	1,560.0	4	1,666,667	416,667	—
LUBM	273.0	72	4,761,905	66,138	—
WatDiv	272.3	1	400,285	400,285	1.00
WatDiv	85.7	4	1,271,852	317,963	3.18
WatDiv	26.6	16	4,097,658	256,104	10.24
WatDiv	17.5	32	6,242,710	195,085	15.56
WatDiv	17.5	64	6,228,441	97,319	15.56
WatDiv	16.8	72	6,487,959	90,111	16.21

number of threads (Threads), the loading rate in triples per second (T/s), the relativised loading rate in triples per thread per second (T/T/s), and the speedup relative to a single thread (Speedup) in the WatDiv case.

The same DJKB loading test was repeated using several other knowledge graph systems. RDFox was at least an order of magnitude faster than any of these other systems; unfortunately the licence conditions of these systems mean that we are not able to present their results here.

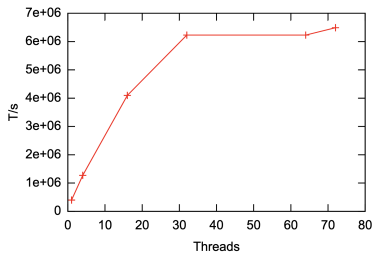

Fig. 3. Number of threads vs. loading speed (triples per second)

Even the relativised (T/T/s) loading rates are not directly comparable across different datasets as there may be a large difference in, e.g., the cost of parsing, which can depend on many factors (such as the structure of URIs). In order to give a clearer idea of the effectiveness of parallel loading we therefore repeated the WatDiv loading test using different numbers of threads; these results are also presented in Table 1, and we have additionally plotted T/s against the number of threads in Figure 3. As we can see, the speedup from 1–32 threads is relatively consistent, with 32 threads giving a nearly 16 times speedup, but there is little or no additional speedup after that. The reasons for this are not fully understood, and are difficult to investigate in a cloud computing environment; however, we believe that there are only 36 physical cores, with the 72 vCPUs coming from hyper-threading, so significant speedup beyond 36 times is not necessarily to be expected.

Table 2. Results on WatDiv 100M benchmark (times in ms)

Query	#ans	R1	R2	R3	R4	R5	Avg	ms/ans
L1	2	1	1	1	1	1	1	<1
L2	595	16	210	24	60	7	63	<1
L3	24	1	1	1	1	1	1	<1
L4	603	20	11	9	8	7	11	<1
L5	958	16	41	5	12	1	15	<1
S1	6	1	1	1	1	1	1	<1
S2	249	10	12	19	50	5	19	<1
S3	0	29	26	33	30	29	29	–
S4	13	113	333	233	15	30	145	11
S5	68	15	13	11	17	10	13	<1
S6	81	4	15	11	1	6	7	<1
S7	0	1	1	1	1	1	1	–
F1	7	7	6	11	11	4	8	1
F2	58	1	6	9	4	1	4	<1
F3	128	1	7	12	5	1	5	<1
F4	382	6	5	4	5	6	5	<1
F5	43	1	1	1	1	1	1	<1
C1	201	30	23	34	23	20	26	<1
C2	22	140	65	75	62	59	80	4
C3	4,244,261	1,830	1,640	1,380	1,360	1,450	1,532	<1

4.3 Query Answering

The results on the WatDiv queries are presented in Table 2. For each query we give the number of answers (#ans), the time to return all answers in 5 separate runs (R1–R5), and the average time (Avg); we also give the average time per answer (ms/ans). All times are in milliseconds. As can be seen, RDFox answers most queries in only a few milliseconds; query C3 takes an average of 1,532ms, but this is mainly due to the time taken to return over 4 million answers. The average time per answer is less than 1ms in most cases, and never more than 11ms.

Table 3. Results on LUBM 10k benchmark (times in ms)

Query	#ans	t-1	t-10	t-100	t-all	ms/ans
q1	4	2	1	1	1	0.21
q2	2,528	1,440	2,210	17,800	459,000	181.57
q3	6	1	1	1	1	0.11
q4	34	1	1	1	1	0.01
q5	719	1	1	1	1	0.00
q6	104,403,077	1	1	1	68,933	0.00
q7	67	1	1	1	1	0.01
q8	7,790	1	1	1	39	0.01
q9	2,721,773	1	1	5	128,000	0.05
q10	4	1	1	1	1	0.13
q11	224	1	1	1	1	0.00
q12	15	2	1	1	1	0.03
q13	46,366	1	1	1	303	0.01
q14	79,211,095	1	1	1	37,733	0.00

The results on the LUBM queries are presented in Table 3. For each query we give the number of answers (#ans), the time to return the first answer (t-1), the first 10 answers (t-10), the first 100 answers (t-100) and all answers

(t-all); we also give the average time per answer (ms/ans). All times are in milliseconds. Most of the queries are relatively easy for RDFox, with all answers being returned within 1s, and in most cases in less than 1ms. Queries q6, q9 and q14 take several seconds to fully evaluate, but this is only because of the very large numbers of answers, ranging from 2.7 million up to more than 104 million; the times to return the first 100 answers, and the times per answer, are still in the (sub) millisecond range. Query q2 is the only query that can be considered non-trivial; this is a "triangle" query, where there is no query plan that can avoid computing a very large intermediate result that is subsequently pruned by other query atoms. Even on this query, RDFox returns the first answer in only 1.4s, and returns subsequent answers at a rate of approximately one every 180ms.

For the three DJKB test queries, the average execution time for RDFox was 300ms for Q1, 12ms for Q2 and 10,700ms for Q3. As mentioned in Section 4.1, Q3 is not a realistic query but has been designed as a stress test. The same test was again repeated with several other knowledge graph systems; RDFox was always at least an order of magnitude faster and in some cases several orders of magnitude faster.

5 Discussion and Future Directions

Using a knowledge graph at Dow Jones has had many advantages: it facilitates the integration of data from multiple heterogeneous sources, SPARQL queries provide a powerful and flexible mechanism for accessing information, and this can be used to power a wide range of internal processes and user facing products.

Constructing and maintaining a large knowledge graph can be computationally challenging, as can answering SPARQL queries over the graph. However, RDFox boasts several features that help it to perform well on these tasks, in particular lock-free data structures, parallelised data loading, incremental data updates and a highly optimised SPARQL engine. As a result it can load the entire 2.6 billion triple data set in only 26 minutes and can answer typical application queries in only a few milliseconds.

Currently, the majority of the data in the knowledge graph is kept up to date by simply reloading it on a regular basis (once per month). This is feasible given RDFox's fast loading time, but is clearly not ideal. Dow Jones developers are therefore working an a rearchitected system in which RDFox is connected directly to data sources (a feature already supported by RDFox) and the graph is automatically updated whenever the source data changes.

So far the knowledge graph has mainly been used as part of internal processes such as the extraction of signals from news feed articles and the maintenance of competitor relationships data. Work is underway to integrate the knowledge graph into a wider range of internal processes, for example to support the Risk and Compliance team, and into existing customer facing products. It is also planned to develop several new and more powerful applications that were previously infeasible due to data integration issues and/or query performance issues. One specific goal is to increase customer engagement by providing user specific recommendations for relevant articles in news feeds.

References

1. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison Wesley Publ. Co., Reading, Massachussetts (1995)
2. Aluç, G., Hartig, O., Özsu, M.T., Daudjee, K.: Diversified stress testing of RDF data management systems. In: ISWC (1). Lecture Notes in Computer Science, vol. 8796, pp. 197–212. Springer (2014)
3. Arenas, M., Bertails, A., Prud'hommeaux, E., Sequeda, J.: A direct mapping of relational data to RDF. W3C Recommendation (27 September 2012), available at http://www.w3.org/TR/rdb-direct-mapping/
4. Guo, Y., Pan, Z., Heflin, J.: LUBM: A benchmark for OWL knowledge base systems. J. of Web Semantics **3**(2–3), 158–182 (2005)
5. Härder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15**(4), 287–317 (1983)
6. Harris, S., Seaborne, A.: SPARQL 1.1 Query Language. W3C Recommendation (21 March 2013), available at https://www.w3.org/TR/sparql11-query/
7. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
8. Horrocks, I., Patel-Schneider, P.F., Boley, H., Tabet, S., Grosof, B., Dean, M.: SWRL: A semantic web rule language combining OWL and RuleML. W3C Member Submission (21 May 2004), available at http://www.w3.org/Submission/SWRL/
9. Motik, B., Nenov, Y., Horrocks, I.: Parallel materialisation of a set of logical rules on a logical database (US Patent 10817467, Oct 2020)
10. Motik, B., Nenov, Y., Horrocks, I.: Complex query evaluation using sideways information passing (US Patent 11216456, Jan 2022)
11. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Incremental update of datalog materialisation: the backward/forward algorithm. In: Proc. of the 29th Nat. Conf. on Artificial Intelligence (AAAI 15). pp. 1560–1568. AAAI Press (2015)
12. Motik, B., Nenov, Y., Piro, R., Horrocks, I.: Maintenance of datalog materialisations revisited. Artificial Intelligence **269**, 76–136 (2019)
13. Motik, B., Nenov, Y., Piro, R., Horrocks, I., Olteanu, D.: Parallel materialisation of Datalog programs in centralised, main-memory RDF systems. In: Proc. of the 28th Nat. Conf. on Artificial Intelligence (AAAI 14). pp. 129–137. AAAI Press (2014)
14. Nenov, Y., Piro, R., Motik, B., Horrocks, I., Wu, Z., Banerjee, J.: RDFox: A highly-scalable RDF store. In: Proc. of the 14th International Semantic Web Conference (ISWC 2015). Lecture Notes in Computer Science, Springer (2015)
15. Neumann, T., Weikum, G.: Scalable join processing on very large RDF graphs. In: SIGMOD Conference. pp. 627–640. ACM (2009)
16. Neumann, T., Weikum, G.: The RDF-3X engine for scalable management of RDF data. VLDB J. **19**(1), 91–113 (2010)
17. OWL 2 Web Ontology Language Overview (Second Edition). W3C Recommendation (11 December 2012), available at http://www.w3.org/TR/owl2-overview/
18. Prud'hommeaux, E., Boneva, I., Labra Gayo, J.E., Kellogg, G.: Shape expressions language 2.1. W3C Community Group Report (8 October 2019), available at http://shex.io/shex-semantics/
19. Shapes Constraint Language (SHACL). W3C Recommendation (20 July 2017), available at https://www.w3.org/TR/shacl/
20. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. PVLDB **1**(1), 1008–1019 (2008)