Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

Corpus Processing Service: A Knowledge Graph Platform to perform deep data exploration on corpora.

Peter W J Staar, Michele Dolfi, Christoph Auer

[1]IBM Research, Saumerstrasse 4, 8820 Rueschlikon, Switzerland

Correspondence
Email: taa@zurich.ibm.com

Summary

Knowledge Graphs have been fast emerging as the de facto standard to model and explore knowledge in weakly structured data. Large corpora of documents constitute a source of weakly structured data of particular interest for both the academic as well as the industrial world. Key examples include scientific publications, technical reports, manuals, patents, regulations, etc. Such corpora embed many facts that are elementary to critical decision making or enabling new discoveries. In this paper, we present a scalable cloud platform to create and serve Knowledge Graphs, which we named *Corpus Processing Service*. Its purpose is to process large document corpora, extract the content and embedded facts, and ultimately represent these in a consistent knowledge graph that can be intuitively queried. To accomplish this, we use state-of-the-art natural language understanding models to extract entities and relationships from documents converted with our previously presented CCS platform. This pipeline is complemented with a newly developed graph engine which ensures extremely performant graph queries and provides powerful graph analytics capabilities. Both components are tightly integrated and can be easily consumed through REST APIs. Additionally, we provide user-interfaces to control the data ingestion flow and formulate queries using a visual programming approach. The CPS platform is designed as a modular microservice system operating on Kubernetes clusters. Finally, we validate the quality of queries on our truly end-to-end knowledge pipeline in a real-world application in the oil and gas industry. To date, the capabilities of CPS are successfully leveraged in more than 5 client engagements.

KEYWORDS:
keyword1, keyword2, keyword3, keyword4

1 | INTRODUCTION

As of 2015, Adobe estimated that there were 2.7 trillion PDF documents in circulation globally[1]. It is self-evident that this number has increased ever since. The explosive growth of documents one can observe since digital publishing became mainstream is posing a serious challenge to both the academic and corporate world. The increased publication rate of scientific articles makes it harder and harder for academics to keep aware of all the latest findings. Similarly, the ever-growing number of internal reports, documentation, patents, contracts, regulations, court filings etc. is for most corporations becoming simply unmanageable.

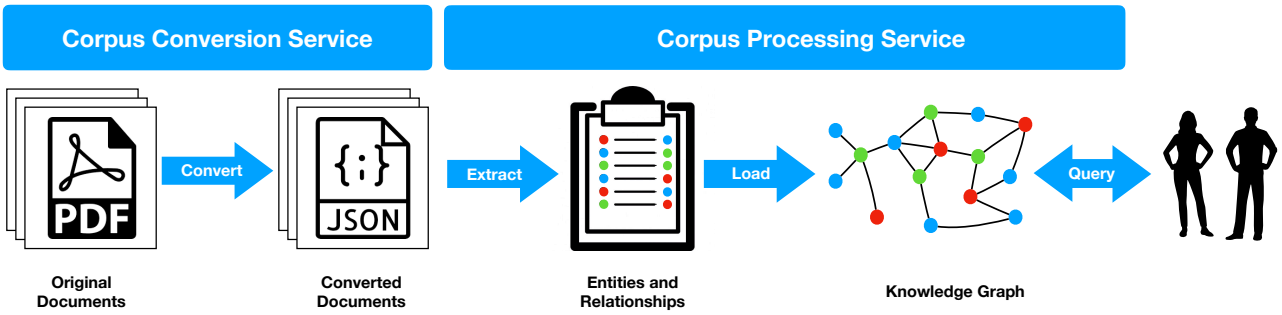

FIGURE 1 Sketch of the entire pipeline to perform deep data exploration on large corpora.

In a previous publication, we presented the Corpus Conversion Service (CCS)[2]. The CCS is a scalable could service, which leverages state-of-the-art machine learning to convert complex formats (e.g. PDF, Word, Bitmap) into a richly structured JSON representation of their content. As such, the CCS solves the first problem when confronted with a large corpus of documents, i.e. make the content of the documents programmatically accessible. Examples of the latter would be *List all images with their caption from the corpus* or *list all titles with their publication date*. The second problem is to obviously search or explore the content of the documents in a large corpus. For this problem, we have developed the Corpus Processing Service (CPS), which we present in this paper. The CPS is intended to create knowledge bases (KB's) from the converted JSON corpus and serve these KB's through in-memory knowledge graph stores. As such, the CPS is the natural extension of the CCS and has as an express purpose to make corpora of documents available for deep data exploration.

Currently, there exist only few well-integrated solutions to perform deep data exploration directly on large corpora [1]. Here, we define deep data exploration as the capability to ingest large corpora of documents into a scalable service and detect, extract and combine facts contained in these corpora in order to make new discoveries or support critical decision making. Currently, the most capable solutions to explore the knowledge in large corpora are restricted to keyword search (mostly for documents and embedded content) in combination with filtering tools.

It is key to understand that our goal of creating and querying Knowledge Graphs to enable deep data exploration goes beyond search in the spirit of rank and retrieve. While search is by no means trivial, many state-of-the art solutions exist for this purpose [2]. We argue however, that one needs query capabilities which allow for a combination of extracted facts and a fast, on-the-fly creation of new datasets to enable actual deep data exploration. Those datasets can then be used for further analysis, which might lead to new discoveries or support decision making.

To better distinguish this approach from conventional search, let us consider some example questions:

a) *"Definition of high temperature superconductor"*

b) *"Publications of Dr. Costas Bekas before year 2010".*

c) *"Maps of the Permian basin".*

d) *"Geological formations from the Miocene age with their depth, thickness, geographic location and composition".*

e) *"List all high-Tc superconductors with their known crystallographic and material properties?"*,

Question a) undoubtedly fits the classic search paradigm, since here one can expect a search engine to find a number sources with exact answers (i.e. definitions). Likewise, question b) can be easily answered through metadata based filter rules on a literature database. Question c) already requires some extent of domain knowledge to be encoded in a model to accurately classify the relevance of all known maps to the query, at least assuming no manual curation effort has been done. Question d) and e) ultimately impose query capabilities which are clearly infeasible to support through manual curation, and are very unlikely to be answered in any single data source. These questions require the system to return a more complex data structure (*e.g.* a table in which the rows list the formations or materials while the columns contain their respective properties).

[1] Elsevier Knovel (https://www.elsevier.com/solutions/knovel-engineering-information), GeoDeepDive (https://geodeepdive.org)
[2] For example ElasticSearch (https://www.elastic.co) and ApacheLucene (https://lucene.apache.org

Concluding from the above examples, we define the following qualifying criteria for a system that supports deep data exploration on corpora:

1. It can answer queries by combining different data elements from different sources into a new data structure.
2. It supports (1) by creating a knowledge model from a controlled, unstructured corpus in a mostly unsupervised way. It may profit from, but not require any manually curated data.
3. It may restrict supported queries to a specific domain (*e.g.* a technical field).

In this paper, we introduce Corpus Processing Service (CPS), which is a cloud platform to create and serve Knowledge Graphs (KGs) from unstructured corpora in order to enable deep data exploration.

To meet the objectives defined earlier, CPS implements and tightly integrates two essential components. The first component is a scalable Knowledge Graph creation pipeline, which is used to automatically process text, tables and images through state-of-the-art segmentation and Natural Language Understanding (NLU) models and extract entities and relationships from them. The second component serves the created KG, enabling users to perform deep queries and advanced graph analytics in real-time[3]. This is supported through an underlying, highly optimised graph engine we developed to specifically address requirements for deep data exploration.

While CCS is purely purposed to interpret the (visual) document structure, CPS analyses the extracted content in order to further extract the facts and create a Knowledge Graph. The full processing pipeline is outlined in Fig. 1. To make the complementary nature of CCS and CPS evident, let us take captioned figures and tables as an example. Here, CCS extracts and links the caption to the proper figure or table structure, and CPS later exploits this information to accurately interpret the content of such figure or table and index the mentioned concepts or quantities.

The CPS platform has been deployed and successfully applied in multiple real-world scenarios in material science[4] and oil and gas industries[5] since we released first prototypes in early 2018.

In the remainder of this paper, we discuss in detail the technical aspects and implementation details of the two main components of the CPS. In Section **??**, we present in depth how the platform extracts facts from corpora at a massive scale. In Section **??**, we go into detail of designing deep queries and show how we compute them in a very efficient way with our high-performance graph engine. Later, in Section **??**, we will discuss in detail how both components are deployed and interacting on the cloud. Finally, in Section **??**, we present the complete system in a real world case study and benchmark its accuracy.

2 | SCALABLE KNOWLEDGE GRAPH CREATION

In CPS a Knowledge Graph is defined as a collection of entities and their relationships forming the graphs nodes and edges. Entities can have a wide variety of types. A basic scenario includes types such as documents, document components, keywords and authors. In addition, there can be more specific types tied to domain verticals, such as materials and properties in material science, or geological ages, formations, rocks, minerals, structures etc. for oil and gas exploration. Relationships in the KG are strictly defined between the entities. Similarly to the entities, the relationships are typed (*"has-material-property"* or *"has-geological-age"*). Also, relationships in the KG can be weighted, for example to represent the trustworthiness of a fact that the relationship represents.

In typical cases, we start from a collection of documents in different formats. Sometimes, documents are available in semi-structured, machine-interpretable formats such as JSON, XML or HTML. However, in the vast majority of cases this does not apply, especially for proprietary documents of companies and organisations. The latter are very often scanned or programmatic PDF documents. Using the CCS[2], these types of documents are converted into structured JSON files. Those provide easy access to the meta-data (*e.g.* title, abstract, references, authors) and the document body. The latter is structured by subtitles (of various levels), paragraphs, lists, tables (with internal row and column structures), figures and linked captions. Once the corpus is present in a structured, machine-processable format, the KG is created by applying three distinct tasks, namely **extraction**, **annotation** and **aggregation**. The inherent dependencies between these three tasks are defined through a Directed Acyclic Graph (DAG). We will refer to this DAG of tasks as a *Data Flow* (DF). In the next subsections, we establish the concept of Data Flows and discuss the details for each Data Flow task.

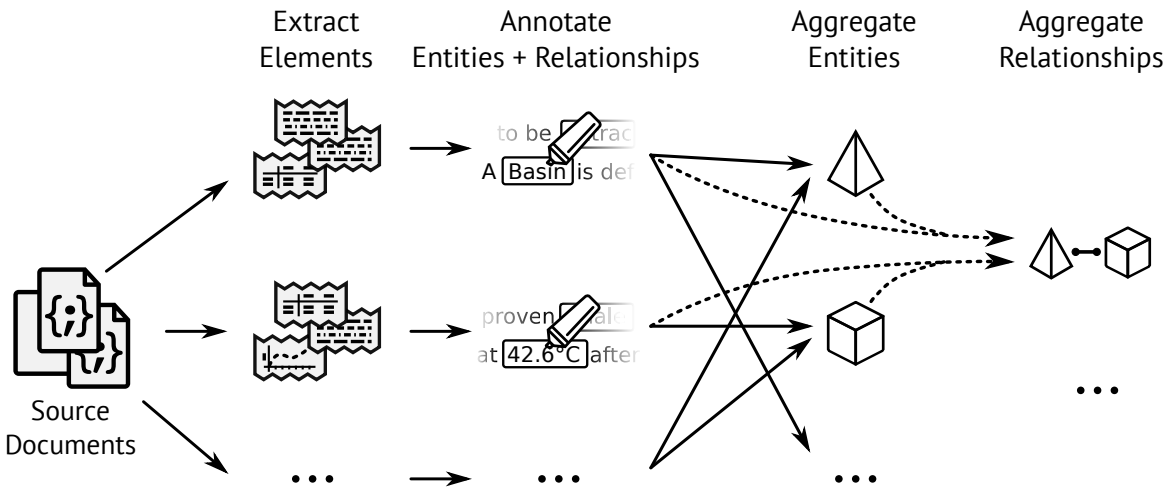

FIGURE 2 Schematic of a Data Flow for the creation of a Knowledge Graph. The Data Flow consists of three main task types: extraction of document elements (abstracts, paragraphs, tables, figures, etc.), annotation of these elements to detect entities and their relationships and finally aggregation of these entities and their relationships. For every task, we keep complete provenance, such that we can always trace back to a specific document or element that embeds a certain entity or relationship.

2.1 | Data Flow tasks

In Fig. 2, we sketch a minimal Data Flow, in which each of the three tasks are used consecutively in order to generate entities and relationships for a generic KG. We will use Fig. 2 to illustrate the purpose and implementation of each DF task.

2.1.1 | Extraction

In an extraction task, we generate new data entities (*e.g.* document components) from an original set of source entities (*e.g.* documents). During this process, new links are created which connect these newly generated data entities to their original source entity. Typical examples of such extraction tasks are the extraction of abstracts, paragraphs, tables or figures from the structured document files.

From a scalability point of view this task is embarrassingly parallel, which makes it extremely easy to implement on loosely interconnected environments such as a cloud. We simply iterate in parallel over all source entities in the backend database, extract the desired components and then insert those components as new data entities back into the database. Extraction tasks have no internal synchronisation points.

One particular benefit of this task is to make the query capability on the Knowledge Graph more fine grained by being able to provide provenance information on the result. For example this would let the user explore all the paragraphs, tables or figures that embed a certain fact.

2.1.2 | Annotation

In the annotation task, we apply Natural Language Understanding (NLU) methods to detect language entities and their relationships within a single data entity. Here, data entities can be as simple as a snippet of text (*e.g.* a paragraph) or more complex structures such as tables or figures. The main goal of the annotation task is to obtain all relevant information from the data entity with regard to the domain of the corpus. Since different technical fields require different annotations, our annotation task is modular, allowing language entities to be annotated for material science, oil and gas or more basic entities (*e.g.* noun phrases, abbreviations, unit and values, etc).

From a technical perspective, the language entities are detected and annotated using multiple NLU methods, ranging from complex regular expressions[3] to LSTM networks[7,8]. We employ state-of-the-art NLU toolkits such as Spacy[9] or NLTK[4] to

[3]Most language entities from a technical field are typically represented in a very specific, rigorous way that can be easily captured by regular expressions. We found that in practice, regular expressions often outperform DL models, since we can simply encode these representations.

[4]https://www.nltk.org

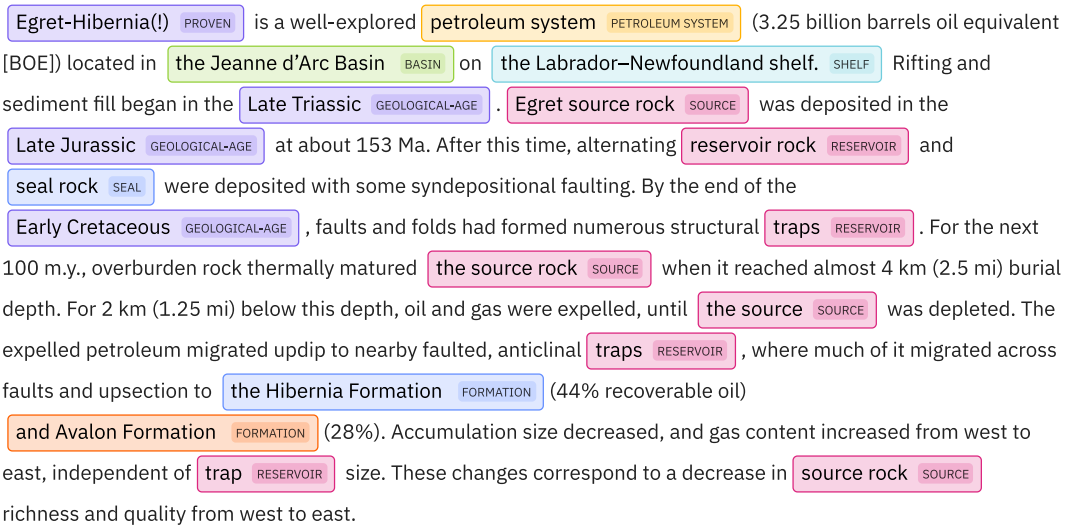

FIGURE 3 Illustration of various detected language entities in a particularly rich snippet of an AAPG abstract[6]. The language entities here are all related to geological concepts in the domain of oil and gas exploration.

train and apply custom named entity recognition models. A detailed investigation of these NLU annotators unfortunately goes beyond of the scope of this paper. However, in Fig. 3, we show the different types of named (geological) entities found in a paragraph by our oil and gas annotation model.

In Listing. 1, we also show an excerpt of how the annotations (both language entities and relationships) are stored in the backend. It is noteworthy here that relationships are stored as (weighted) links between two entity references[5]. The usage of references reduces data duplication and more importantly ensures that the relationships are always defined between two known entities in the KG. The latter simplifies the aggregation of the relationships significantly, since no new entities need to be created in the KG in order to aggregate the relationships (see section 2.1.4).

From a scaling perspective, this task is again embarrassingly parallel. Unlike the extraction task, the annotation task is not creating new data entities, but rather appending new data associated with an existing data entity. We simply apply the desired entity and relationship annotators on all document components (paragraphs, tables, etc) in parallel by distributing the operations on all available compute resources. Annotation tasks have no internal synchronisation points. From a corpus of about 100'000 documents we typically extract about three million paragraphs. Assuming unlimited resources, the annotation task could be distributed to potentially three million independent workers.

2.1.3 | Aggregation of entities

The aggregation task for entities is similar to an extraction task, in the sense that we create new entities and link them each to the source they were mentioned in. In addition to extraction, the entity aggregation task also applies a similarity metric[6] between the entities during extraction. This similarity metric will define if two entities refer to the same language concept and thus need to be represented by a single entity in the KG, rather than remaining separated. In Fig. 2, we have illustrated the aggregation task for two types of entities across many different document components. These entity types could be for example materials and properties or geological formations and geological ages. The links connecting the new entities to their source entity are weighted according to the frequency of the match, i.e. we set a higher weight if the the language entity has been found multiple times. From an implementation point of view, the aggregation task for entities is non-trivial. In distributed computing, it corresponds to a reduction operation. Our implementation distributes the iteration of the source elements among all available computational resources. The aggregation is first performed in a local buffer, which is then synchronised with the backend

[5] We follow the standard JSON-schema for references.

[6] A rather simple similarity metric is to perform a fuzzy comparison of the names of the newly found entities (i.e. the *name* field found in Listing. 1). A more sophisticated approach is to use word embeddings to identify if two concepts are similar.

Listing 1 Excerpt of the annotated abstract from an AAPG paper[6] with its original text and the detected entities and relationships. Note that relationships are typed (encoded in the field name) and weighted. The weight reflects the confidence of the language annotation model during extraction. Relationships are always defined on detected entities, and will therefore use references defining a link between two entities.

```
%\begin{code}
{
  "text": "Egret-Hibernia(!) is a well-explored petroleum system located in the Jeanne d'Arc
   ↪  Basin on the Labrador Newfoundland shelf. Rifting and sediment fill began in the Late
   ↪  Triassic. ...",
  "entities": {
   "petroleum-system": [
   { "name": "Egret-Hibernia Petroleum System"},
   ... ] ,
   "geological-structure": [
    { "name": "Jeanne D'Arc Basin"},
    { "name": "Labrador-Newfoundland Shelf"},
    ... ]
  },
  "relationships": {
    "located-in": [
      ["#/entities/petroleum-system/0", "#/entities/geological-structure/0", 1.0],
      ["#/entities/petroleum-system/0", "#/entities/geological-structure/1", 1.0],
      ["#/entities/geological-structure/0", "#/entities/geological-structure/1", 1.0],
      ... ]
  }
}
```

database only when it reaches a maximum size. The synchronisation step is a simple atomic update into an existing (or a newly created) database object. The synchronisation for updates from each worker task does not collide with the others.

2.1.4 | Aggregation of relationships

The aggregation of relationships introduces new links between the entities that were aggregated in the previous aggregation operation. In Fig. 2 this task is depicted as the last operation, where entities with an annotated relationship are explicitly linked together. For example we create an edge between the *Egret-Hibernia Petroleum System* and *Jeanne D'Arc Basin* from Listing 1.

Similar to the aggregation of entities, the aggregation task for relationships is a reduction operation. Two independent document components could describe the same relationship between two entities. To minimise the synchronisation lookup operation with the backend database, this task also utilises a local buffer which accumulates the changes to be committed to the KG until the maximum size is reached. This approach allows to distribute the computation among all the source document components and perform very few blocking operations in the backend database.

2.2 | Data Flows

The purpose of a Data Flow is to provide an execution plan for the task types detailed above in a meaningful order to generate or update a specific KG. When instantiating a DF, one has the possibility to define in a declarative way:

1. Which document components should be extracted from a converted corpus to form source entities (*e.g.* extract all paragraphs, tables, figures and captions from the AAPG articles)

2. Which annotator model(s) to use on which type of source entity (*e.g.* run the geology or material science annotators on paragraphs)

3. Which entity and relationship aggregations to perform on which set of annotated language entities

The DFs can thus be seen as blueprints for processing the corpus into a defined graph topology. Notably, our implementation of DFs and their tasks retains the flexibility of processing not only source documents of a well-known data schema such as from CCS, but virtually any structure that can be transformed to a JSON representation, including data entities from pre-curated

databases. We designed the CPS platform to support export and import of DFs on entirely new datasets without the burden of recreating it from scratch.

Our backend engine can exploit the DAG defined through the DF to massively distribute the individual tasks on all compute resources, because independent branches of the DAG each containing a chain of tasks can execute in parallel. The achievable level of parallelism changes throughout the execution. A practical example is a DF which extracts paragraphs and abstracts from all documents in the corpus, then annotates them and finally aggregates all entities. Here the extraction tasks are distributed only over all documents, then in the annotation tasks we increase the parallelism to all document components. Any synchronisation points thus can be pushed back into the aggregation tasks.

2.3 | Cloud architecture

For the KG creation pipeline, we implemented an asynchronous compute scheme we already use in our CCS solution[2]. The system is exposed to the user via an API frontend which communicates to the compute workers through a message broker and a result backend. The workers operate on the data, which is hosted on a NoSQL database and a cloud object store for data blobs. These workers are dynamically scaled by the cloud orchestrator to best match the current load of the platform.

The processing of the KG creation typically starts with the user submitting the DF to the frontend API. The DAG of operations is then interpreted as described in the previous section and fine-grained tasks are submitted to the broker, *e.g.* the whole corpus is split in many independent chunks. The user receives an overall status from the API and is notified when the DF processing has completed.

3 | DEEP DATA EXPLORATION USING KNOWLEDGE GRAPHS

We will now look into the requirements to perform deep data exploration on a populated Knowledge Graph. A deep data exploration requires two fundamental capabilities:

1. perform *deep queries* on the graph, i.e. queries that require multi-hop traversals
2. perform *graph analytics* on the full graph or subsets of it on-the-fly

Deep queries are essential to dynamically combine independent facts together in the given query context. This would apply for example to explorational queries aimed to characterise petroleum system elements, as detailed in our case study (see section **??**). Graph analytics can further reveal hidden structure in the KG topology. Examples of advanced graph-analytical operations are page rank, node centralities[10,11], node clustering, spectral analysis and label propagation.

Both deep queries and graph analytics have in common that they are inherently expensive to compute on conventional graph databases, due to a rapid expansion of the number of visited nodes as a function of the graph-traversal depth. This is a major obstacle in providing reasonable time-to-solution in the aforementioned cases, as we demonstrate later in 3.2. Virtually all established graph database products on the market today[7] fall victim to this, as was also reported in multiple sources[12,13]. Due to the poor performance we observed with available graph databases, we developed a new graph engine for the CPS platform. This graph engine is able to execute advanced graph-analytics[3] as well as evaluate deep queries with multi-hop traversals on large graphs (>1B edges) extremely fast.

In the remaining part of this section, we elaborate on our newly developed graph engine. In subsection 3.1, we discuss the implementation design. In subsection 3.2, we discuss performance results and compare it to Neo4J. Later, in subsection 3.3, we will explain how the deep queries are formulated and evaluated in the graph engine. Finally, in subsection 3.4 we illustrate the cloud architecture serving the KG.

3.1 | Design of the graph engine

In computer science, two prevalent implementation schemes for graphs have emerged, one using *adjacency lists* and one relying on *adjacency matrices*[14,15]. In the adjacency list format, every node is essentially an object which contains a set of indices

[7]For example Neo4J, Titan, JanusGraph, Amazon Neptune and Arangodb

representing its neighbours[8]. The edges are therefore stored as a property of the node. In the adjacency matrix approach, all nodes obtain an identifier (typically an unsigned integer) and the edges are stored as a list of node-identifier tuples.

It is commonly known that most graph operations can be translated into matrix-operations using linear algebra[14]. For example, consider the graph-traversal $\mathscr{V} \xrightarrow{\mathscr{A}} \mathscr{W}$, in which we start from a set of nodes $\mathscr{V}$ and traverse the edge $\mathscr{A}$ in order to obtain a new set of nodes $\mathscr{W}$. This can be directly translated into linear algebra as

$$\vec{w} = A\,\vec{v} \quad \text{with} \quad \vec{v}_i = \begin{cases} 1 & \text{if} \quad \text{node i} \in \mathscr{V} \\ 0 & \text{if} \quad \text{node i} \notin \mathscr{V}, \end{cases} \tag{1}$$

and with A being the adjacency matrix representation of the edge $\mathscr{A}$. Translating single graph-traversals into linear algebra operations significantly simplifies the job of deeper graph traversals. For example, to obtain the k-order neighbourhood of node set $\mathscr{V}$, one simply needs to evaluate Eqn (1) k times recursively, as in

$$\vec{w} = A^k\,\vec{v} = A\,(A\,(\ldots\,(A\,\vec{v}))). \tag{2}$$

Therefore, deep queries can be implemented efficiently as long as Eqn (1) can be evaluated efficiently. Over the past decades, lots of research has been conducted in the High Performance Computing community on the acceleration and parallelisation of Eqn (1) in the context of graphs. In this context, the matrix A is sparse and the linear operation of Eqn (1) is referred to as a sparse matrix vector multiplication (SpMV), for which highly optimised implementations have been developed[16,17]. Notably, most advanced graph-analytical operations can be formulated using SpMV operations. The most trivial case is page-rank, in which one recursively executes Eqn (1) in combination with a renormalisation until $\vec{w}$ is equal to $\vec{v}$. In our previous work[3], we have also shown in detail that advanced graph-analytical operations such as node centralities and spectral analysis of the graph can be done effectively with only SpMV operations.

Since both deep queries and advanced graph analytics hugely benefit from a fast SpMV kernel, we have opted to design the graph engine in the CPS platform to work entirely with the adjacency matrix format.

3.2 | Memory architecture and performance optimisation

Both adjacency lists and adjacency matrices based graph implementations have specific advantages and disadvantages. The adjacency list format is very well suited for node-centric operations since it exploits data-locality for local graph operations, such as first order traversals. However it proves suboptimal for global scale graph operations, which are required for deep queries and the advanced graph analytics. Here, one typically has to perform graph-traversals starting from many (or even all) nodes and accumulating the weight in the resulting nodes. In an adjacency list format, this often leads to many cache misses during execution, resulting in low performance. Furthermore, parallelising global graph-traversals in the adjacency list format suffers significantly from concurrent write conflicts between threads during execution. In the adjacency matrix format, these problems are not encountered. The graph-traversals can be directly translated into a SpMV or even a sparse-matrix sparse-vector multiplication (SpMSpV). It has also been well established how to execute the SpMV effectively in a multithreaded fashion, and how to minimise cache-misses by applying a clever sorting of the tuples list[18].

To illustrate the advantages of the adjacency matrix format for our needs, we show the time-to-solution (TTS) for queries with increasing order of traversals for Neo4J[9] and our graph engine in Fig. 4. We computed a k-hop traversal query on the graph500[10] (64M edges) and twitter-graph[11] (1.5B edges). Two important observations can be made. Firstly, our graph engine is able to run easily 3rd, 4th and even higher-order graph traversals. With Neo4J, this proves very difficult, as the TTS grows upwards of one hour. Secondly, our graph engine shows minimal variance in the TTS between all runs of the k-order graph-traversals. This is in stark contrast to Neo4J, where the TTS strongly depends on which node(s) one starts from.

Another big advantage of using the adjacency matrix format is that we can exploit advanced compression methods[19] such as CSR or blocked COO. This reduces significantly the memory footprint of the graph and allows bigger graphs to be hosted entirely in-memory. In our case, we have opted to represent the edges by blocked matrices of a fixed size, in which each block matrix is of type COO. We chose the size of the block-matrix to be $2^{16} = 65536$, allowing a pair of indices to be compactly

[8] This memory architecture is clearly documented for Titan (http://s3.thinkaurelius.com/docs/titan/current/data-model.html) and Neo4J (http://key-value-stories.blogspot.com/2015/02/neo4j-architecture.html).

[9] We chose Neo4J as a reference since it is currently the most popular graph database solution, see https://db-engines.com/en/ranking_trend/graph+dbms

[10] http://graph500.org/

[11] https://snap.stanford.edu/data/higgs-twitter.html

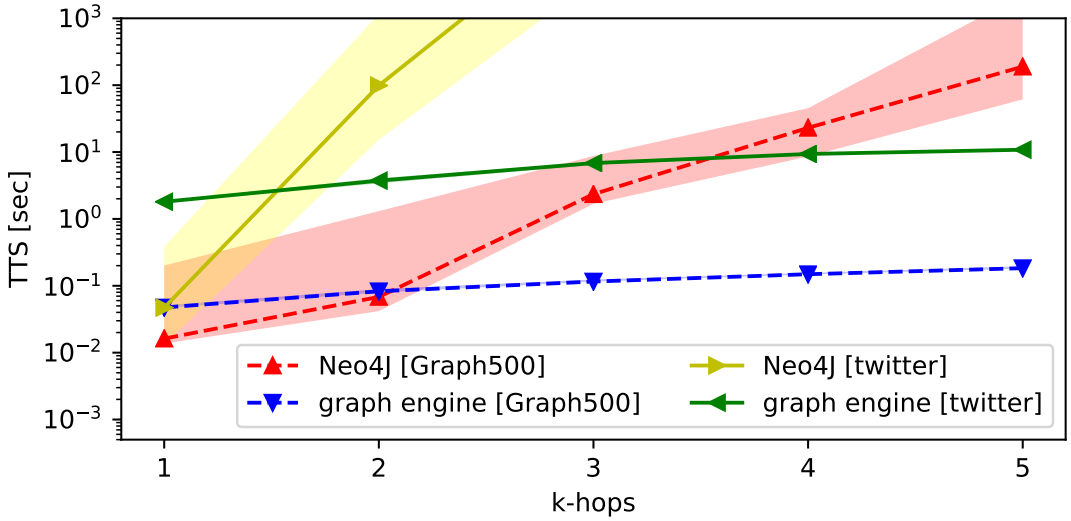

FIGURE 4 The time-to-solution for k-hop graph traversal for Neo4J and our new graph engine. The results were obtained for the graph500 and twitter benchmark graphs. The 10th and 90th percentiles are represented by the shaded regions, the median is shown by the markers.

represented by two unsigned short integers. Consequently, an edge has a memory footprint of only 4 bytes (equivalent to a single 32-bit integer), while a weighted edge a footprint of 8 bytes[12]. This is a significant reduction in memory footprint compared to Neo4J graph databases, which use 33 bytes for unweighted edges[13]). Consequently, we can host graphs of close to 8 billion edges on a virtual machine with 32 GB of free memory, and even close to one trillion edges on a bare-metal POWER9 node with 4TB of memory.

3.3 | Formulation and evaluation of Deep Queries

The goal of querying a KG is to answer complex questions. As such, users need to be provided with a functionality to formulate complex queries on the KG and quickly evaluate them.

In order to avoid imposing a complex query language onto users, we have devised a way to define complex graph queries in a declarative format, which we call a *workflow*. Workflows are represented as a DAG of operations and are conceptually related to Data Flows. Unlike the former, the nodes of workflow DAGs do not represent data-transformation tasks, but specific graph operations which mutate an input (or intermediate) set of nodes into another set. We call these operations *worktasks*. For further convenience, we have developed a graphical user interface (UI) which allows to define such workflows in a visual programming approach (see Fig. 5) .

Currently, we support four fundamental types of worktasks: *node-retrieval*, *traversal*, *logical operators* and *transform functions*. In the following sections, we will discuss in detail how the worktasks are implemented in the context of our adjacency matrix design.

Node retrieval

This task finds a set of nodes which satisfy certain search criteria. This can range from finding a single node by its (approximate) name or exact node identifier, to finding nodes that satisfy a particular property. The task constructs a node vector $\vec{v}$, such that

$$\vec{v}_i = \begin{cases} 1 & \text{if} \quad \text{node i} \in \mathscr{S} \\ 0 & \text{if} \quad \text{node i} \notin \mathscr{S}, \end{cases} \tag{3}$$

where $\mathscr{S}$ represents the set of nodes that satisfy the search criteria.

[12] We assume the weight can be represented by a float value.
[13] https://neo4j.com/developer/guide-sizing-and-hardware-calculator/

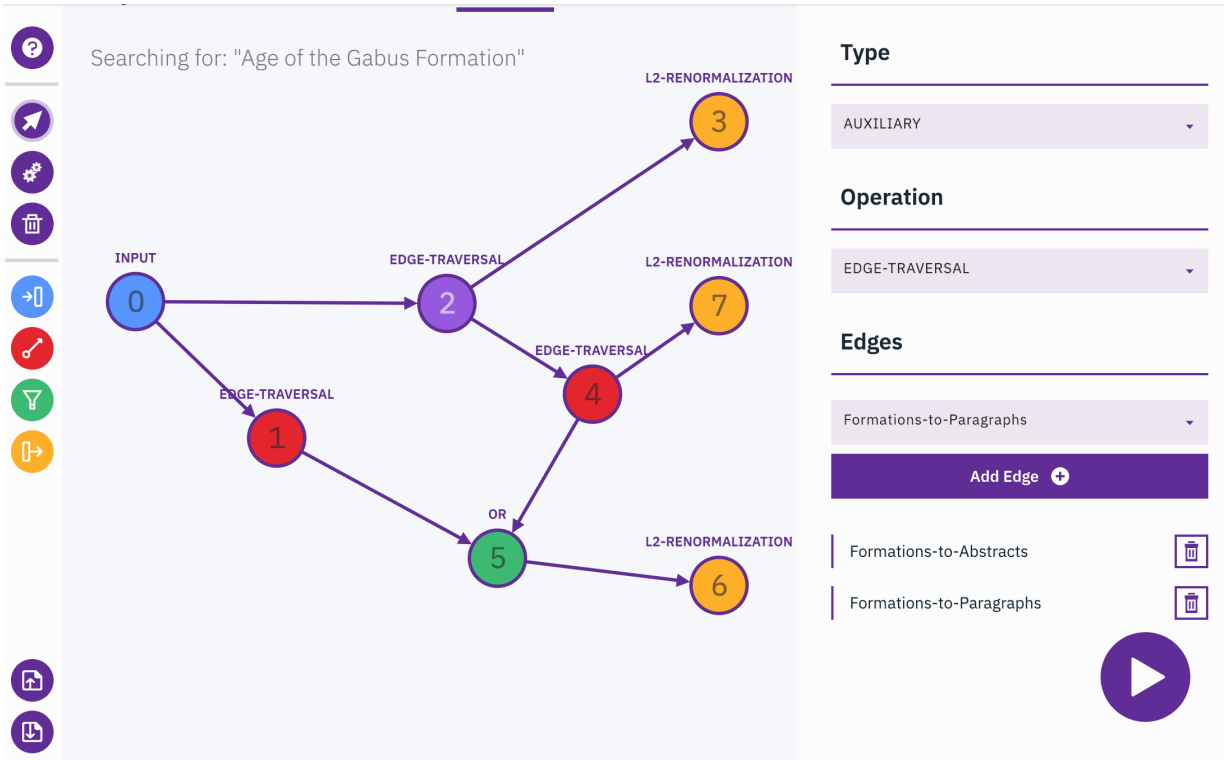

FIGURE 5 Visual workflow editor for deep queries in the CPS platform. The interface exhibits a left toolbar to pick specific graph operations, a main drawing area for the workflow DAG and a right panel to inspect and define parameters of each graph operation. Colours indicate different operation types such as input node-retrieval (blue), traversal (red), logical operators (green) and transform functions (yellow). Valid workflows can be executed using the *play* button.

Graph traversal

The simplest type of graph-traversal is the direct graph-traversal. As explained in detail in section 3.1, these can be implemented as a straightforward SpMV operation $\vec{w} = A\,\vec{v}$. In more advanced types of graph-traversals we evaluate all paths of different depth. Since the number of paths connecting two nodes might increase exponentially with the path-length, one typically reduces the contribution of each path by weighting it with the inverse factorial of the path-length. For example, consider the case in which we want to explore deeper, indirect paths as follows,

$$\vec{w} = \left(A + \frac{A^2}{2!} + \frac{A^3}{3!} + \dots\right)\vec{v} = \left(e^A - 1\right)\vec{v}. \tag{4}$$

In its most generic case, a graph-traversal can therefore be written down as a matrix-function applied on an edge, i.e. $\vec{w} = f(A)\,\vec{v}$. As discussed in detail in previous work[3], this type of operation can be evaluated extremely efficiently using a recursive Chebyshev polynomial expansion.

Logical operations

In logical operations, two sets of nodes are merged into one resulting set, each represented through a node vector. There are three common logical operations, AND, OR and NOT. In the AND and OR operations, we compute the geometric or the arithmetic mean respectively for each pairwise elements in the vectors. In the NOT operation, we inverse the sign for each element of the input vector.

Transform functions

Lastly, we implement operations which transform the weights associated with nodes. One such operation renormalises and ultimately ranks the nodes according to their weight.

With these four types of operations, we can express rich queries to answer complex questions, which can have multiple in- and outputs.

Let us now discuss how a workflow is evaluated within the graph engine. Once a workflow has been submitted, each worktask is initially assigned a vector. These vectors are all initialised to zero ($\vec{v}_i = 0$). Next, the graph will analyse the DAG of worktasks and identify which tasks can be run in parallel. This is achieved by performing a topological sort using depth-first traversal, which yields a list in which each item is a set of tasks that can be executed in parallel. The graph engine then proceeds with the parallel task computations. For each task, we obtain a set of nodes with corresponding weights by identifying the non-zero elements in the associated node vector. After executing the full workflow, we therefore obtain for each task a list of nodes which can be sorted according to their weights. The higher the weight of the node, the more relevant this node is. As such, we can also retrace which nodes were important in each stage of the workflow.

3.4 | Cloud Architecture

The KG data is distributed between three storage solutions: a NoSQL database, a cloud object storage (COS) and the graph engine. Each node is represented as a document in a NoSQL database which contains all the properties attached to the node, *e.g.* the text of a paragraph. If there is a binary object attached to the node, *e.g.* the PDF document or an image, this is stored on the COS. The graph engine contains only the minimal information needed to execute the queries, i.e. the connectivity of the graph and the properties which are indexed for filtering and search.

The graph engine is exposed to the user via a REST API which is able to aggregate results collected from the different storage sources. To ensure decent performance when serving queries of multiple users, the graph engine can be dynamically scaled horizontally. Most workflow queries execute fast enough such that they can be responded from a synchronous request. Others, especially the graph analytics computations, are more expensive and return large amounts of data. Thus, these queries are executed through an asynchronous API and the results are paginated and streamed back to the user on completion.

4 | CLOUD DESIGN

The primary deployment target for the CPS is a cloud environment orchestrated via Kubernetes. We package the full platform assets with a Helm chart for quick deployment on multiple setups. For example we can easily deploy the platform on the IBM Cloud or on-premise in an IBM Cloud Private instance, both on x86- and POWER-based nodes.

In Fig. 6 we show the high-level cloud design of the CPS. The platform allows to manage and instrument the corpus processing in a multi-tenant fashion, i.e. it handles multiple knowledge ingestion pipelines and it serves multiple knowledge graphs. We call each unit a *Knowledge Graph Space* (KGS), which consists of a dedicated instance of the graph engine, a dedicated MongoDB database and a bucket on a Cloud Object Store (COS). A dashboard allows each project owner to manage the access and the usage of resources. The KGS can be launched into multiple flavours to optimally balance the utilisation of the cluster. These flavours range from a virtual machine with small amount of memory to a full dedicated node including hardware acceleration with GPUs. Once a KGS is created, it can be paused and rescaled without loss of data or downtime.

5 | CASE STUDY: OIL AND GAS EXPLORATION

Oil and gas exploration is a complex, technical field of expertise, in which one tries to infer new potential geographic sites for oil and gas discoveries. To be successful in oil and gas exploration, one needs to have a deep understanding of geological processes in combination with lots of data associated with these geological processes.

Unfortunately, the data of many geological processes and entities is scattered across databases (public and proprietary) and corpora of documents, where it is often deeply embedded in text, tables and figures. For example, geographic information of geological structures can be found in NaturalEarthData[14], while their history, evolution and components (*e.g.* formations with their age, rock-composition and depth) are discussed in reports (governmental and proprietary) and scientific articles. As a consequence, experts in oil and gas exploration often need to read many documents in order to find all the information of a certain geographic area and get a good understanding of its underlying geology.

[14] https://www.naturalearthdata.com/

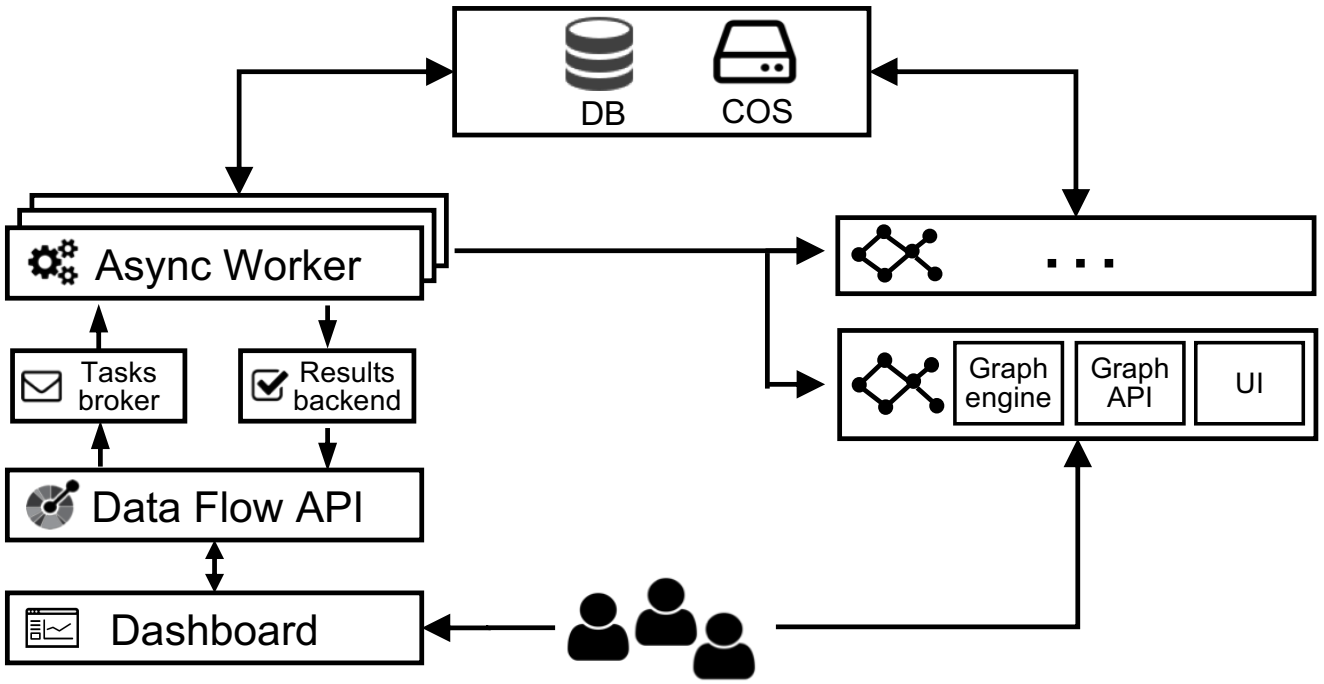

FIGURE 6 The architectural design of the CPS platform. On the left, we show the Data Flow processing architecture orchestrated through an asynchronous REST API. On the right, we sketch the multi-tenant KG serving facility which provides a dedicated environment for each project.

The main tasks of the experts working in oil and gas exploration is to identify potential new exploration sites. This is typically done by describing a basin or one of its sub-regions. In practice, *"describing a basin"* boils down to identifying all geological formations with their properties in the basin and investigating if these formations constitute a petroleum system[20]. In its most minimalistic form, a petroleum system is defined by three components: source, reservoir, and seal. The source is the rock formation in which the oil or gas was created. Once created, the oil or gas typically migrates to a porous reservoir rock, which holds the oil and gas. In order for the oil and gas not to escape, the reservoir needs to be covered by an impermeable rock formation which is called the seal. Each one of these components is comprised of one or more formations, with a certain age and rock composition. To identify a petroleum system in a certain geographical area, one has to find a candidate formation for each component (i.e. reservoir, seal and source) and observe that the properties of these components satisfy some well-established constraints. For example, the reservoir formation has to have a lower depth than the seal formation. Another example of such constraints is that the age of the seal and reservoir has to be older than the source.

In order for the CPS platform to help the oil and gas explorationalists in their day-to-day job effectively, it needs to meet two objectives. On the one hand, it needs to create a consistent Knowledge Graph from a document corpus. This Knowledge Graph has to contain all geological formations with their respective properties (*e.g.* geographical locations, depth, age, rock composition). On the other hand, CPS needs to provide fast query responses, such that one can automatically retrieve potential components of petroleum systems and apply the constraints to filter out promising candidates.

During the development and implementation of custom NLU annotators in CPS for oil and gas exploration, the client team worked hand in hand with the IBM Research team to set up a controlled accuracy benchmark in which the key capabilities of the CPS can be quantified. The goal of the benchmark was to test the entire pipeline depicted in Fig. 1, i.e. from PDF document ingestion to a final, queryable KG. The key components of this specific pipeline are,

1. the conversion of PDF documents into JSON through CCS
2. the creation of the KG in the CPS from the JSON documents
3. the querying of the KG served by CPS to identify petroleum systems elements with their properties

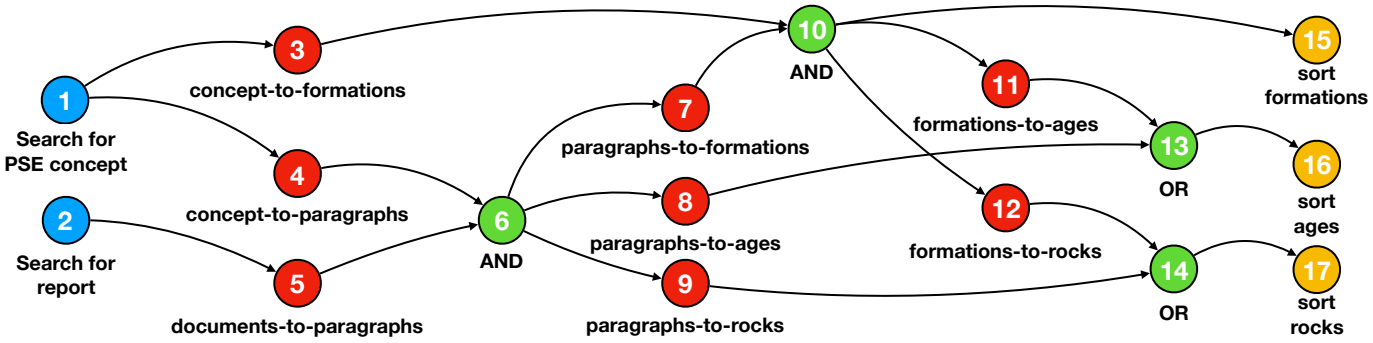

FIGURE 7 The evaluation workflow to identify the petroleum system elements (PSE) in an article and infer its properties. It starts by searching for all petroleum system elements of a certain type (*e.g.* source, reservoir or seal) and a particular report (worktasks 1 and 2). By successive graph traversals (worktasks 3-5, 7-9, 11, 12) along specific edges and logical operations (worktasks 6, 10, 13, 14), we are able to obtain a list of candidate formations (worktask 15), ages (worktask 16) and rocks (worktask 17), ranked by their accumulated weight. Execution of this query takes less than 18 ms on average.

TABLE 1 Top-k accuracies validation of KG query results. Numbers represent the fraction in which any of the *k* highest ranked answers matches the expected answer.

PSE	property	top-1	top-2	top-3	top-5
reservoir	age	0.82	0.96	0.98	1.00
	formation	0.93	0.98	1.00	1.00
	rock	0.62	0.80	0.87	0.94
seal	age	0.73	0.91	0.94	0.97
	formation	0.82	0.94	0.97	0.98
	rock	0.82	0.92	0.95	0.97
source	age	0.75	0.92	0.96	0.97
	formation	0.89	0.96	0.97	0.98
	rock	0.83	0.92	0.95	0.96

On the suggestion of the experts in the client team, the entire pipeline was run on the 1051 *Field Evaluation Reports* from the C&C Reservoirs[15] dataset. The advantage of using this dataset for an accuracy benchmark is that each report includes two parts. One part is verbose text describing the history, evolution and composition of the fields. The language used is of similar complexity to standard geological publications and thus a realistic challenge for our KG creation pipeline. The second part at the end of each report is comprised of tables which summarise the text and provide us the elements of the petroleum systems with their properties. Therefore, we ingest these reports into CCS and extract both text and tables. Then, by generating a KG only from the text and keeping the tables as ground-truth to compare answers of the KG queries against, we obtain a well-controlled, end-to-end accuracy benchmark.

For step (1) of the pipeline, we ingested all 1051 PDFs into CCS and visually annotated the document structure on 300 (out of 46019) pages. This yielded a page model which accurately converted all documents to JSON format with a 99.7% recall and 99.3% precision in the converted structure. These numbers are in line with those reported in our previous works[2]. Importantly, very accurate conversion results are key to the resulting quality, since otherwise the language annotators will process incomplete data and eventually the relevance of query results will suffer.

In step (2), we create the Knowledge Graph by executing a Data Flow that will generate all the entities and relationships relevant to the geology domain. Our language annotator models trained for geology extract geographic areas, geological structures (*e.g.* basins), formations, ages, rocks, petroleum systems and their elements (PSE) (*e.g.* seal, source and reservoir). Overall, we

[15]https://www.ccreservoirs.com/

extracted a total of 4597 PSEs, 8811 formations, 471 geological ages and 64 rock types (relevant to the PSEs). The full processing performed at an average rate of 130 ms per page per worker core, on a system with 3 worker nodes each using 4 cores. Eventually, the KG included 679'296 edges connecting 116'662 nodes.

In step (3), we query the Knowledge Graph using a tailored evaluation workflow. This workflow allows us to identify PSEs and their connected properties in the Knowledge Graph, *e.g.* their age, formation and rock composition. In Fig. 7, we visualise the DAG of this workflow. The final node weights are accumulated throughout the branches on the workflow and represent the relevance score of each node.

To evaluate the correctness of the predicted PSE properties, we follow the standard practice of reporting the *top-k accuracy*. This is computed as the percentage in which any of the k highest ranked answers matches the expected answer, over all documents. In table 1, we show the *top-1*, *top-2*, *top-3* and *top-5* accuracy for all properties of each petroleum system element. One can make two distinct observations. First, the *top-1* numbers are in the range of 0.75-0.9, meaning that for 3 in 4 cases, the most relevant result predicted by the KG was correct (precision). Secondly, we observe that the *top-5* numbers are very high (≥ 0.97), showing that the system was able detect and aggregate most of the PSEs and their properties (recall). Thus, the recall of the language annotators in the KG creation pipeline was very satisfactory.

6 | CONCLUSIONS

With the introduction of the CPS platform, we demonstrate substantial benefit for domain experts and data scientists in exercising deep exploration of published knowledge in a fully integrated, yet modular cloud solution. CPS seamlessly connects to the *Corpus Conversion Service*, complementing it with a highly scalable, automated pipeline to build consistent domain knowledge models and an intuitive, powerful approach to explorational queries and graph-scale analytics. This is accomplished through three fundamental design considerations: 1. We do not require manual data curation or annotation. 2. We built a scalable, efficient architecture to support the ingestion, processing and query workloads, all embedded in a single platform and 3. We expose the capabilities through an intuitively consumable API and complementary UI tools.

In our oil and gas case study, we successfully verified our solution for a real-world application with the help of subject matter experts from a client team. Currently, CCS and CPS are actively used in more than 5 client engagements, most notably in the oil and gas industry as well as in the material science industry.

Future work will focus on processing public repositories such as the arXiv.org library, USPTO and PubMed in order to make their content available to deep data exploration.

References

1. Ydens P. This number originates from a keynote talk by Phil Ydens, Adobe's VP Engineering for Document Cloud. A video of the presentation can be found here: https://www.youtube.com/watch?v=5Axw6OGPYHw. 2015.

2. Staar PWJ, Dolfi M, Auer C, Bekas C. Corpus Conversion Service: A Machine Learning Platform to Ingest Documents at Scale. In: KDD '18. ACM; 2018; New York, NY, USA: 774–782

3. Staar PWJ, Barkoutsos PK, Istrate R, et al. Stochastic Matrix-Function Estimators: Scalable Big-Data Kernels with High Performance. In: ; 2016: 812–821

4. Manica M, Auer C, Weber V, et al. An Information Extraction and Knowledge Graph Platform for Accelerating Biochemical Discoveries. *ArXiv* 2019; abs/1907.08400.

5. Ruffo P, Piantanida M, Bergero F, Staar P, Bekas C. Application of Geocognitive Technologies to Basin & Petroleum System Analyses. 2019: 10. doi: 10.2118/197610-MS

6. Magoon LB, Hudson TL, Peters KE. Egret-Hibernia(!), a significant petroleum system, northern Grand Banks area, offshore eastern Canada. *American Association of Petroleum Geologists Bulletin* 2005; 89(9): 1203–1237. doi: 10.1306/05040504115

7. Lample G, Ballesteros M, Subramanian S, Kawakami K, Dyer C. Neural Architectures for Named Entity Recognition. In: ; 2016.

8. Chiu JPC, Nichols E. Named Entity Recognition with Bidirectional LSTM-CNNs. *TACL* 2016; 4: 357-370.

9. Honnibal M, Montani I. spaCy 2: Natural language understanding with Bloom embeddings, convolutional neural networks and incremental parsing. *To appear* 2017.

10. Estrada E. Subgraph centrality in complex networks. *Physics Review E* 2005; 71(5): 056103.

11. Estrada E, al. e. Network properties revealed through matrix functions. *SIAM Review* 2010; 52(4).

12. Labs R. Benchmarking RedisGraph 1.0. 2019.

13. TigerGraph . Real-Time Deep Link Analytics. 2018.

14. Kepner J, Gilbert J. *Graph Algorithms in the Language of Linear Algebra*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics . 2011.

15. Kepner J, Bader DA, Buluç A, Gilbert JR, Mattson TG, Meyerhenke H. Graphs, Matrices, and the GraphBLAS: Seven Good Reasons. In: . 51 of *Procedia Computer Science*. Elsevier; 2015: 2453–2462.

16. Buluc A, Gilbert JR. The Combinatorial BLAS: Design, Implementation, and Applications. *Int. J. High Perform. Comput. Appl.* 2011; 25(4): 496–509. doi: 10.1177/1094342011403516

17. Kepner J, Aaltonen P, Bader DA, et al. Mathematical foundations of the GraphBLAS. *2016 IEEE HPEC* 2016: 1-9.

18. Azad A, Jacquelin M, Buluç A, Ng EG. The Reverse Cuthill-McKee Algorithm in Distributed-Memory. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)* 2017: 22-31.

19. Shahnaz R, Usman A, Chughtai IR. Review of Storage Techniques for Sparse Matrices. *2005 Pakistan Section Multitopic Conference* 2005: 1-7.

20. Welte D, Horsfield B, Baker D. *Petroleum and Basin Evolution: Insights from Petroleum Geochemistry, Geology and Basin Modeling*. Petroleum and Basin Evolution: Insights from Petroleum Geochemistry, Geology, and Basin ModelingSpringer . 1997.