A Domain-Independent Model for Capturing Annotation Data and Provenance
Modeling and Representation of CrowdTruth

THESIS

submitted in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

in

INFORMATION SCIENCES

by

Khalid Khamkham

Department of Computer Science
Faculty FEW, VU University Amsterdam
Amsterdam, the Netherlands
http://www.few.vu.nl/

Center for Advanced Studies
IBM Netherlands
Amsterdam, the Netherlands
http://www.ibm.nl
A Domain-Independent Model for Capturing Annotation Data and Provenance
Modeling and Representation of CrowdTruth

Khalid Khamkham
VU University Amsterdam
k.khamkham@gmail.com - 2005379

Abstract. With the advent of crowdsourcing as a viable method for data collection, the need for integrated annotation frameworks has increased. The relatively low cost and scalability of crowdsourcing methods allow for various types of annotation tasks to be run in different domains across different data modalities. From these tasks, a large amount of unprecedented data follows which needs to be processed and assessed. To facilitate these needs we propose a domain-independent model for use in the context of the CrowdTruth annotation framework. We present a solution for capturing and storing a variety of data and its provenance throughout the whole annotation workflow by utilizing the W3C PROV Model and the MongoDB NoSQL database.

1 Introduction

Data-driven machine learning, used to train and evaluate cognitive systems, requires input of a certain type of quality: ground truth data. Collecting such data is often a time-consuming and expensive process when done by domain experts [1], and may furthermore only result in limited datasets. Crowdsourcing is a viable alternative for this type of data collection, as it can provide to be a much more scalable and cheaper solution [2]. Crowdsourcing is a type of participative online activity wherein a task is proposed to a group of individuals which may then be voluntarily undertaken [3].

An example of a cognitive system that utilizes both human and machine intelligence is Watson QA: an artificially intelligent computer system developed by IBM that is capable of answering questions posed in natural language [4]. Watson is currently being adapted for question-answering in the medical domain [5][6][2], which requires large amounts of training and evaluation of medical texts for gathering the necessary ground truth data. This form of ground truth data collection typically requires human intelligence for the purposes of specialized text annotation like medical relation extraction, medical term correction and event annotation. However, because the usage of human expert resources can be time consuming, the ground truth data can be collected using a crowdsourcing approach instead.
Amazon Mechanical Turk (AMT)\(^1\) and CrowdFlower\(^2\) are examples of crowdsourcing platforms, which provide an environment wherein so-called workers can perform Human Intelligence Tasks (HITs), that are posted by so-called requesters. These HITs, otherwise known as microtasks, are sub-problems that multiple workers can independently perform often in return for a financial reward \(^7\). Crowdsourcing methods allow for a plethora of tasks to be run on different data modalities ranging from text to images and videos across different domains and data granularities. Because the input can be diverse in nature and because crowdsourcing follows the premise that a microtask can be performed by many contributors, a large amount of data follows from the results which needs to be assessed and interpreted.

In order to assess the quality and trustworthiness of the data collected, provenance of the data stored and processed needs to be captured. Provenance, or lineage, is information that describes the origins of an object and the various processes it went through before reaching its ultimate derivation \(^8\). It is metadata that can be used to reason over: reliability, quality, use and reusability, integrity and other facets of trust. As such, it is held in high regard within the scientific community as it provides a means for verifiability and justification of results.

Currently however, provenance is generally not captured when running an annotation task. We identify two main reasons for this: (i) the tools used throughout the annotation workflow generally work independently of each other, as such, they lack the interoperability to communicate the changes that occur at each transformation step; (ii) crowdsourcing platforms such as AMT and CrowdFlower generally encapsulate data and only provide limited analytics functionalities. Consider the example where the output of an arbitrary annotation task forms the input of another new annotation task, tracing back the ultimate derived result back to its initial state requires that provenance must be captured at every step in order to support such a data linkage.

We have approached our work in the context of the CrowdTruth framework \(^9\), which enables the gathering of human annotated data by combining machine-human computation into one unified end-to-end workflow. The main principles that underly the CrowdTruth framework are: the support for different data modalities, extendible task templates and platforms, disagreement analytics and provenance enabled storage.

Researchers of the CrowdTruth framework argue that there is an increasing need for such integrated annotation frameworks. First, the large amount of data used throughout the annotation workflow is difficult to interpret unless it is stored in one single place. Second, a typical annotation workflow consists of several steps such as: pre-processing of the input data, data collection, data analytics and post-processing. These steps are currently often performed independently from each other and can thus not provide a continuous automated collection of ground truth data unless they are integrated. Third, provenance of

\(^1\) https://www.mturk.com
\(^2\) http://www.crowdflower.com
the data stored and processed must be captured in order to assess its quality and trustworthiness.

In this paper we (the author) focus on the design of the CrowdTruth Data Model. More specifically, we research (i) how we can best model the data used in an annotation framework, such that it can support diverse data and provenance (ii) how we can store this data, to support the large amount of data and (iii) how we can access and query this data. We present our work in the form of a domain-independent Base Data Model which supports the W3C PROV Model \cite{10}, and the Instantiated Data Model implemented within the context of the CrowdTruth framework by utilizing the Laravel framework\(^3\) and the MongoDB NoSQL database\(^4\).

The rest of the paper is organized as follows. In section 2 we describe related work with regards to provenance. Section 3 details the design of the CrowdTruth Data Model. In section 4 we describe the implementation of our model inside the CrowdTruth framework. In section 5 we evaluate our data model by means of a qualitative analysis focusing on provenance, a comparative analysis focusing on specific modeling decisions, and a user study focusing on the perception of the end-user throughout the framework. Finally, we discuss future work in section 6 and conclude in section 7.

2 Related Work

This section outlines work which relates to our main research objectives. First, we describe provenance types and granularities. Next, we describe a W3C proposed recommendation for modeling provenance: PROV. Then, we look at work which captures provenance in a NoSQL environment and we describe an integrated annotation framework. Finally, we discuss the further improvements that can be made to integrated annotation frameworks.

2.1 Provenance types and granularities

Tan et al. \cite{11} identify two granularities of provenance: workflow (coarse-grained) provenance and data (fine-grained) provenance. “Workflow provenance refers to the record of the entire history of the derivation of the final output of the workflow.” This type of coarse-grained provenance allows for validation by repetition as its record enables experiments to be replicable throughout every step in the workflow. Typical to this type of provenance are the activities which record the input, processes and output that take place at each subsequent step. “Data provenance gives a relatively detailed account of the derivation of a piece of data that is in the result of a transformation step.” This type of fine-grained provenance does not allow transformed states to be divided into smaller pieces, and can thus give a detailed overview of every transformed state in the form of a self-contained entity.

\(^3\) http://www.laravel.com
\(^4\) http://www.mongodb.org
Buneman et al. [12] describe how fine-grained provenance can be further categorized into where- and why- provenance. Where-provenance denotes the exact location of the piece of data from which the derivative was obtained. Why-provenance justifies how one such derivative can be obtained while taking into consideration all sources that may have affected the transformation step.

Simmhan et al. [13] describe a taxonomy of provenance and its characteristics. Again, here the distinction is made between data-oriented and process-oriented provenance approaches. In essence, these types of provenance have also been mentioned by Tan et al. [11] and others [14,15,16]. The recurring theme, is that there is a need for supporting these different types of provenance. Our data model ensures that these different types or perspectives can be accessed individually, i.e. as partial provenance from a particular perspective, or as a hybrid/complete provenance.

Harth et al. [17] argue for adding a social dimension to the technical notion of provenance, “to associate provenance with the originator of a given piece of information”. An example given for such a model is “return all movies that are highly rated by my acquaintances, but exclude person X since I dislike her movie taste”. We agree with adding such an attribution dimension to our data model, as we may need to obtain similar information such as: “return all units annotated by a subset of crowdsourcing agents, but exclude the agents who have been identified as spammers”. Thus, in order to form judgments on the trustworthiness of a particular result, the reputation and reliability of its originators must be taken into account as a key requirement [18,19,20].

2.2 PROV Model

A more exhaustive list of related work with regards to provenance has been compiled by the World Wide Web Consortium (W3C) Provenance Incubator Group [21]. Based on their research, a set of specifications for modeling provenance on the Web has been produced: The PROV family of documents [10]. PROV is designed to be generic and domain-agnostic to support the diverse nature of the Web of Data. As such, it provides several classes which are abstract enough to be modeled within any application. Figure 1 shows the core PROV classes and their relations.

- **Entities**, are things which can be physical, digital, conceptual, or of another kind for which provenance may be described [22].
- **Activities**, are “how entities come into existence and how their attributes change to become new entities” [22].
- **Agents**, are entities that take “a role in an activity such that the agent can be assigned some degree of responsibility” [22].

The relations between the various classes define how they connect and depend on each other. We will elaborate upon these in Section 3, as the PROV model serves as a base for our data model. PROV furthermore describes how provenance can be viewed from different perspectives [22].
– **Agent-centered** provenance focuses on the people or organizations that were attributed to the creation of an entity and/or running of an activity.
– **Object-centered** provenance focuses on the lineage of entities by “tracing the origins of portions of a document to other documents”.
– **Process-centered** provenance focuses on the activities, i.e. the “actions and steps take to generate the information in question”.

The PROV family of documents thus intends to provide a standardization for modeling provenance to promote capturing of such metadata and interoperability between various applications.

---

![Core W3C PROV Classes](image)

**Fig. 1**: Core W3C PROV Classes

### 2.3 Provenance in systems

Sakka et al. [23] present a provenance management system (PMS) based on semantic web technologies. The semantic model used in this PMS is based on the Open Provenance Model [8], a previous iteration of the PROV model. Implementation and evaluation of this model was done on two storage systems: Sesame, an RDF triple store; and CouchDB, a NoSQL document-oriented database. Experiments were performed by running provenance queries on both systems. The Sesame RDF triple store showed to be highly expressive due to its SPARQL support, but limited in terms of scalability. CouchDB does not provide any SPARQL support, rather querying for provenance was done with the MapReduce model. Results for the document-oriented database showed good performance for aggregation and path queries, and a limited decrease in expressiveness. Furthermore,
NoSQL databases were deemed to be a good fit for provenance data, as such data is static and diverse in nature. The no schema design of NoSQL databases allow for such flexibility and its denormalization approach fits well with data that does not need to be updated.

GATE [24], a General Architecture for Text Engineering, is a framework and graphical development environment for language engineering. GATE is comprised of three main elements: The GATE Document Manager provides a central repository to store all information about the data being processed, A Collection of Reusable Objects for Language Engineering allow for processing of various language tasks, and, the GATE Graphical Interface adds support for visualization of the data and management of corpora. In addition to this, GATE can be extended with plugins. The GATE Crowdsourcing plugin [25] allows for mapping of documents to crowdsourcing units which can be used in tasks on platforms such as AMT and CrowdFlower. Provenance is tracked by grouping annotations in separate annotation sets, such as: pre-annotated annotations, crowdsourced judgements and consensus sets. The plugin furthermore adds support for reusable interfaces for NLP classification and selection tasks.

In this section we looked at different facets of provenance and how they interconnect with each other. Specifically, we can see that the so-called workflow provenance and data provenance identified by Tan et al. [11] are also mentioned in the PROV model as respectively: process-centered provenance and object-centered provenance. A previous iteration of the PROV model: the Open Provenance Model, was shown to be used successfully in a NoSQL environment by Sakka et al. [23]. GATE comes close to what can be considered an integrated annotation framework. However, it only supports the annotation of text corpora. Furthermore, the support of provenance is only briefly mentioned via an external plugin and does not seem to follow a recognized standard such as PROV.

We therefore believe there is room for improvement in the form of an annotation framework that can support a variety of data modalities, rather than just text alone. In addition, capturing provenance allows for trust to be assessed by e.g. comparing and replicating experiments, attributing results to its originators, tracing the lineage of derivatives, etc. Furthermore, the use of a recommended provenance standard, such as PROV, will aid in interoperability and exchange of information between various applications. To facilitate these needs we have developed our own model in the context of the CrowdTruth framework[^1] which we describe in the following section.

[^1]: <http://www.crowdtruth.org>
[^2]: <https://github.com/CrowdTruth/CrowdTruth>
3 Design of the CrowdTruth Data Model

In this section we detail the design and development for our data model. As the model lies at the core of the CrowdTruth framework, we first begin by outlining the main components of the framework. Next, we define the key requirements for our data model and explain how they were obtained. Then, we present our model in the form of a Base Data Model (BDM) and Instantiated Data Model (IDM). We distinguish between these two variants to show the genericity of the data model at a conceptual level, and how it can be populated with a diversity of data at an instantiated level.

3.1 CrowdTruth

The CrowdTruth framework integrates a set of open-source components to provide an end-to-end workflow combining machine-human computation for gathering annotated data. Figure shows an overview of the main components.

![CrowdTruth Overall Architecture](image)

**Fig. 2: CrowdTruth Overall Architecture**

- **Data Pre-Processing** provides functionalities for ingestion of data on different modalities (text, images and videos) via direct fileupload and querying of online resources via API interfaces. In addition, pre-processing of data can be applied to filter and specify the appropriate input for microtasks thus optimizing the time and cost spent on gathering annotations.
- **Job Configuration** takes the aforementioned filtered input in the form of a batch, and creates a job with specific job settings such as: task template that is to be used, payment options, and the running platforms for the job.
– **Data Collection** enables the gathering of annotated data from external crowdsourcing platforms via push and polling methods. Data is stored along with its provenance using the PROV model, allowing for deep interlinked analysis between the units, workers and resulting annotations.

– **Data Post-Processing** enables the novel CrowdTruth disagreement metrics, which are based on fostering disagreement between individual annotators, providing insight into vagueness and ambiguity of the input being annotated which may be lost in the traditional inter-annotator agreement method [1].

– **Data Analytics** provides a visualization suite specifically tailored for use with the CrowdTruth metrics. As such, it provides functionalities for evaluating results through graphical views at both individual and aggregated levels.

### 3.2 Data Characteristics

We have previously established that crowdsourcing methods can be applied on a variety of data modalities and domains. Some examples of these different use cases are: Accurator and Steve Museum (image labeling), Waisda? (video), and annotation of text for the purposes of categorization, translation, moderation etc. Here, we emphasize on the diversity of the data used in annotation settings. In addition to this, the CrowdTruth metrics foster disagreement of annotators to capture ambiguity and vagueness of data. This new type of Crowd Truth has proven to be a more effective method for gathering annotated data [26]. To support this generation of ground truth data, all annotations on a particular unit are captured and stored rather than just the inter-annotator agreement alone. This type of collection, further increases the size or scale of the data captured.

In section 2 we have mentioned how provenance is a key metadata for the purposes of trust assessment. Therefore, we identify provenance as a (meta)data characteristic which needs to be captured throughout the annotation workflow. By capturing provenance we may reason over:

– **Object.** The *Entity* that a provenance statement is about.

– **Attribution.** The *Agents* that are responsible for the creation of an *Entity*.

– **Process.** The *Activities* that took place to generate a particular *Entity*.

### 3.3 Model and Storage requirements

Based on the diversity, scale, and need for linkage of the data, we have defined three main requirements for our data model: (i) a high level of abstraction is needed to support different content modalities, (ii) adequate specificity, i.e. semi-structure, is needed to still be able to query the data, and (iii) provenance of the data must be captured. In addition to this, the storage system must support these requirements as well.

Traditional Relational Database Management Systems (RDBMSs) rely on pre-defined schemas to model their data [27,28], often called blueprints or migrations. However, the structure of the data in an annotation framework can
often not be defined upfront, as new unknown data items in other domains will most likely be of a different structure. Therefore, we believe a dynamic flexible schema is needed. In addition, RDBMSs typically normalize the data stored to eliminate redundancy and allow for easy updating, as unique data records then only have to be modified in one single place \cite{29,28}. On the other hand, the so-called NoSQL database storage systems follow a no schema design approach wherein the data stored does not have to adhere to a predefined schema and where denormalization is recommended to support Big Data analytics with models such as MapReduce \cite{30,31}.

We believe that a NoSQL approach is suitable for provenance data as such data is generally static, i.e. not updated, which means that normalization becomes less important. A similar conclusion was drawn by Sakka, et al \cite{23}. Furthermore, normalization requires that join queries are performed before the data results in self-contained objects or entities. A denormalization approach, as used in NoSQL systems, means that no expensive join queries have to be performed. Rather data can be stored as self-contained objects directly, which can then be linked to each other for provenance purposes. The trade-off here is that updates become more expensive in favor of reads. However, as provenance data is static, such updates are generally not performed. In addition, the no schema design allows for great flexibility which support the diversity of the data, and unknown future data, well.

Based on these reasons we opted for a NoSQL approach and we chose for MongoDB\footnote{http://www.mongodb.org} in particular. MongoDB is a NoSQL document-oriented database which does not rely on a predefined schema, rather its schema can be defined dynamically at any point in time. Its BSON\footnote{http://www.mongodb.com/json-and-bson} design allows for any JSON structure to be stored easily with practically no conversion. In addition, most programming languages provide built-in functions to convert their array/dictionary data structures to JSON structures.

A MongoDB deployment hosts a number of databases. A database holds a set of collections. A collection holds a set of documents. A document is a set of key-value pairs. Documents have dynamic schema. Dynamic schema means that documents in the same collection do not need to have the same set of fields or structure, and common fields in a collections documents may hold different types of data\footnote{http://www.mongodb.org}.

The design of our approach follows a slight variation of the official MongoDB documentation for modeling tree structures with parent references\footnote{http://docs.mongodb.org/manual/tutorial/model-tree-structures-with-parent-references/}. In the example given, children (derivatives) are linked to their singular parents via a parent attribute, which allows the formation of tree structures. However, provenance as described in PROV also allows for derivatives to have more than one parent, essentially forming a directed acyclic graph. Many other workflow systems such
as Chimera \[32\], Taverna \[33\] and Kepler \[34\] also record provenance as a directed acyclic graph. Thus, in order to support entities which are derived from multiple parents, we have slightly altered the example given in the official MongoDB documentation by substituting the string parent attribute with an array parents attribute. Both the advantages and limitations for this approach will be discussed in the evaluation section later on.

### 3.4 Base Data Model

Figure 3 shows the CrowdTruth Base Data Model, which is a conceptual domain-agnostic model that can be instantiated with a variety of data. The BDM uses the PROV model as its base, as three of the four models are also core provenance classes: **Entity**, **Agent** and **Activity**. The fourth added **SoftwareComponent** model, allows for information to be stored about the various components used in the framework.

We have implemented the models to make use of the Active Record pattern \[35\]: models are implemented as classes in the application layer, which correspond to their respective collections in the database layer. For example, the **Entity** model corresponds to the entities collection. An instance of such a class in the application layer, corresponds to one document within the database layer. By using this pattern we are able to both instantiate these classes directly and extend them into their domain specific classes. Furthermore, the Active Record pattern provides a straightforward implementation for CRUD operations, as they can be performed directly on these models.

**Relations** In PROV, an entity is derived from zero, one, or many entities. Zero meaning that there was no prior reference to a parent entity, or unknown to the framework. We resolve this wasDerivedFrom relation by storing references to the immediate parents using the parents attribute. The transitive property of the parents attribute can then be used across every generation to reconstruct and trace data lineage. Furthermore, an entity wasGeneratedBy one and only one activity. We resolve this reference with the activity_id, which is stored within the entity itself. The wasAttributedTo relation connects an entity to the one agent responsible for its creation via the agent_id attribute.

Activities are how entities come into existence. One particular activity may use zero, one, or more entities for the generation of new entities. The input to an activity is captured by the used attribute, which references other entities. Furthermore, we have modeled activities to be composed of two other models: agent and software component. An agent is responsible for the running of an activity and uses a particular software component in this process. The agent_id and softwareComponent_id attributes capture the references to these models inside the activity.

Agents are responsible for the creation of entities and running of activities in the framework. An agent can be of a different type e.g. user, software or crowd. Furthermore, an agent worksOn one particular platform such as AMT.
or CrowdFlower. A platform is also modeled as software component within the framework and is referenced by the `softwareComponent_id` inside the agent model.

**Base Attributes and ID Structure** Specific to the entity model are the following attributes: `format`, denotes whether an entity is of a `text`, `image` or `video` type; `domain`, e.g. `medical`, `news`, `art`, also extendible with other domains; `documentType`, e.g. `IBM-medical-sentences`, `NYT-news-article`, `Rijks-image`; `parents`, references immediate parent identifiers to capture data provenance; `content`, contains the JSON structure specific to that `documentType`; `tags`, e.g. `unit`, `segment`, `frame`, which indicates aggregation level, granularity or annotation stage of an entity; `hash`, computed on the content to prevent duplication in the database; `cache`, e.g. `batchCount`, `jobsCount` for query optimization; `agent_id` and `activity_id` to capture provenance references. In the previous section we have already explained most of the attributes for the agent, activity and software component models as these attributes are used to resolve the relations. IDs of a model are composed of specific key attributes.
An **Entity-ID**, is composed of:
{modelName}/{format}/{domain}/{documentType}/{incrementer}.
For example: entity/text/medical/ibm-medical-sentence/1.

An **Activity-ID**, is composed of:
{modelName}/{softwareComponent_id}/{incrementer}.
For example: activity/fileuploader/1.

An **Agent-ID**, is composed of:
{modelName}/{type}/{softwareComponent_id}/{platformAgentId}.
For example: agent/user/crowdtruth/khamkham.

A **SoftwareComponent-ID**, is simply the name of a particular SoftwareComponent as it is always unique.

### 3.5 Instantiated Data Model

The Instantiated Data Model captures domain specific information about the input, actions and output of a particular annotation workflow. We explain the IDM by means of IBM Watson Medical Use Cases [9]. Because these use cases are fairly extensive, they can be used as an example for many other domain specific annotation workflows.

**IBM Watson Medical Use Cases**

- **FactSpan: Correction Factor Span.** The crowd is given a sentence with two highlighted factors. For each factor, the crowd is asked to determine whether it is complete. If it is not, the workers highlight the words in the sentence that would complete the factor.

- **RelEx: Relation Type Identification.** The crowd is given a sentence with two highlighted factors and a set of 12 target relation types. The crowd is asked to select all relation types that are expressed in the sentence between the given factors.

- **RelDir: Relation Direction Identification.** The crowd is given the output of RelEx - a sentence, two highlighted factors, and a relation between the factors - and are asked to choose the direction of the relation. Since this is an easy task, we use golden units to keep the workers honest.

- **RelExDir: Relation Extraction & Direction Identification.** The crowd is given the combined task of relation extraction and direction on the output from FactSpan. As with RelEx, the workers are shown a sentence with the two highlighted factors from the FactSpan task, and then are asked to check all relations that apply between them. On each selected relation its direction is also asked.

**Unit states** At the core of an annotation workflow lie the units which are to be annotated in a particular task. Within the CrowdTruth framework units exists in various states, which are defined by the tags attribute.
– **Raw-Unit.** Initial state of a unit upon ingestion in the framework. Raw denotes that no changes have been made to the original file upon ingestion.
– **Base-Unit.** Unique piece of content stored in the framework, e.g. image or sentence.
– **Media-Unit.** A base-unit with some metadata, e.g. sentence and terms, from which specific task-units can be created.
– **Task-Unit.** A unit with a minimal subset of properties from one or more media-unit, which is used as the specific input for one type of annotation task.
– **Worker-Unit.** Results from an annotation task performed by one worker on one task-unit.

Ground truth data is captured for IBM Watson by annotating so-called *ibm-medical-sentences* with various tasks. The initial state, before entering the CrowdTruth framework, of *ibm-medical-sentences* is a CSV file. Such a file typically has a structure as can be seen in Table 1.

<table>
<thead>
<tr>
<th>Row Nr.</th>
<th>Seed Relation</th>
<th>Term 1</th>
<th>Term 2</th>
<th>Sentence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>cause</td>
<td>lyme disease</td>
<td>borrelia</td>
<td>Lyme disease is a multi-system organ disease caused by Borrelia burgdorferi</td>
</tr>
<tr>
<td>2</td>
<td>cause</td>
<td>lyme disease</td>
<td>borrelia burgdorferi</td>
<td>Lyme disease is a multi-system organ disease caused by Borrelia burgdorferi</td>
</tr>
<tr>
<td>3</td>
<td>cause</td>
<td>viremia</td>
<td>virus</td>
<td>Perinatal HIV infection is characterized by a sustained high-level viremia and a high risk of rapid progression to AIDS, indicating a failure of immunologic containment of the virus</td>
</tr>
</tbody>
</table>

Table 1: Example IBM-medical-sentences CSV file

**Raw-Unit** An *ibm-medical-sentences* file is ingested in the framework with the *FileUploader* software component. Domain specific attributes for this initial file are specified by the agent upon upload through the web interface, such as the *text format*, *medical domain* and *ibm-medical-sentences document-Type*. The *documentType* denotes that the content of this file contains a
certain structure as can be seen in Table 1 with the various columns. With the specified attributes and the tags: raw, unit; an entity 6 is created in the database along with a reference to its responsible agent 8 and the activity 7 that generated this particular entity. The purpose of a raw-unit is to capture the initial state of a document, without any changes, that was ingested into the framework.

**Base-Unit** The next step in the annotation workflow is to identify the unique (base) records of content that were previously uploaded within the raw-unit. An important distinction to be made here, is that the data layer is not responsible for the specific information being stored. Rather it is the concern of the application layer, e.g. Agents and SoftwareComponents, to decide exactly what information is stored and what can be considered as unique content values.

For this particular use case, we have defined that the sentence column in the raw-unit is the base content value. In Table 1 we can see that there are three rows but only two unique sentences. An agent uses the ibm-base-sentence-generator software component 12 which takes as input the initial raw-unit, and stores a new base-unit entity for every unique sentence in the entities collection. These new base-unit entities refer to the initial raw-unit as their parent with the parents attribute. The activity 11 that generates these new entities, records the agent responsible, and the specific input entities and software component used. The purpose of a base-unit entity, is to be able to identify and aggregate all units that belong to an initial piece of unique content throughout various annotation workflows.

**Media-Unit** Next, we generate so-called media-units, which are base-units along with some extra metadata from which different specific task-units can be created. In order to generate media-units, we use the ibm-media-sentence-generator software component 15. Previously, when generating base-units, we had only used the sentence column from the initial raw-unit. For the generation of media-units, we use all the information (columns) available in each row from the raw-unit. As one row is one sentence with some extra metadata such as: term one, term two and the seed relation.

In addition, the software component generates some extra properties which can be used for filtering later on: relationInSentence, specifies whether the stemmed seed relation is mentioned in the sentence; relationBetweenTerms, specifies whether the relation is mentioned between the two terms; relationOutsideTerms, the inverse of the previous property; commaSeparatedTerms, to identify whether terms are mentioned in a list; overlappingTerms, to identify whether two terms share a same word; wordCount, to enable filtering based on a wordcount range; parenthesisAroundTerms and semiColonBetweenTerms.

We store these media-units as new entities with the tags: media, unit, and we reference the raw-unit in the parents attribute. In addition, we reference the previously created unique base-unit with the baseUnit_id attribute. Note
that the base-unit was not modeled as the parent for the media-unit because all the information available in the media-unit wasDerivedFrom from the initial raw-unit. Rather, the purpose of the baseUnit_id is to be able to aggregate various entities which reference the same base-unit, regardless of the generation level. In addition to storing these new media-unit entities, we store the activity [14] which records the input and software component [15] used, and the responsible agent [8] that ran the activity [14].

**Task-Unit** In order to specify what the exact input to one type of annotation task is, we create so-called task-units. A task-unit is a type of entity, which only contains the minimal data needed to perform one particular type of annotation task on a crowdsourcing platform. FactSpan, is the first annotation task in the IBM Watson medical use cases. The minimal input needed for a FactSpan task is an ibm-medical-factspan-sentence which only contains the sentence and one term. This task-unit entity is derived from a previously created ibm-medical-media-sentence, which is a media-unit. We use the aforementioned generated properties in the media-unit to specify and filter the appropriate input for the task-unit e.g. a sentence no longer than 20 words, by using the wordCount property; a sentence wherein the terms are not mentioned in a list, with the commaSeparatedTerms property; etc. These properties help us define the appropriate input, thus optimizing the time and cost spent on an annotation task.

We generate these new task-unit entities, with the tags: task, unit, which refer to the media-unit [13] as their parent using the parents attribute. Again we store the same baseUnit_id that was previously stored in the media-unit as well, because the unique content value is the same. This now allows us to easily aggregate the media-units and task units that belong to a particular base-unit. The activity [17] that generates these task-units, references the input and the specific software component [18] used, and the agent [8] responsible for the creation of these entities [16].

**Batch Entity** In PROV, a collection is set of entities, whose membership can be expressed using the hadMember[11] relation. However, because collection is already used within MongoDB to define a set of documents, we have chosen to use the alternative name batch to model a PROV collection. Batches within the CrowdTruth framework are used for easy reusability. Imagine running the same set of task-units on a new job with different job settings (payment, running platform, etc.), an agent then only needs to specify an existing batch as a parameter for that new job. Batch entities are modeled with the specific documentType batch. A batch references entities via the member attribute. Additionally, extra information can be placed in the content attribute such as the title and description.

Previously, we had created a set of task-units, which were derived from the aforementioned media-units based on some filtered properties. These task-units, as described above.

with the documentType `ibm-medical-factspan-sentence`, are then aggregated into a batch entity which will be used in a crowdsourcing job. The activity 20 records the input, software component 21 and agent 8 associated with the creation of this entity 19.

**Job Entity** A job entity, created by the `jobcreator` software component, specifies the parameters or configuration that will be used for running a particular annotation task on a crowdsourcing platform. Specific to the job entity is the documentType `job`. Example parameters for a job are: payment options, number of judgments on a particular unit, country/channel selection, etc. Most of these options are specific to the running crowdsourcing platform that will be used for the job. The data model does not set any limitations on the amount or naming of these parameters. However, consistency of the JSON structure across jobs is required for easy query-ability.

Furthermore, we have modeled job entities to be composed of two other entities: a batch entity containing the specific task-units which are to be annotated and a template entity which defines the crowdsourcing task-template. Both of these are referenced by the job with the `batch_id` and `template_id` attributes. The activity 23 records the input (batch and template) and software component 23 used, along with the associated agent 8 that was responsible for creating the job.

**Worker-Unit** A worker-unit is the result of an annotation task performed by one worker on one task-unit. One task-unit can have many worker-units, however, a worker-unit always belongs to one specific task-unit. The structure of the results of an annotation done by a worker on a specific unit depends on the task-template that was used. Thus, the documentType for this particular worker-unit is: `factspan-workerunit`, we also add the tags: worker and unit.

In addition, we store a reference to the task-unit that was annotated by the worker inside the parents attribute, as we have modeled a task-unit to be the parent of a worker-unit. Furthermore, we store the mediaUnit_id from which the task-unit was derived and the baseUnit_id which specifies the base content value used in this task. The job_id attribute specifies what job this worker-unit belongs to. Lastly, the worker-unit may store platform specific information such as the HITid (task ID on AMT) or CrowdFlower channel from which this worker-unit originated. The activity 26 stores a reference to the task-unit 16, task-template and software component 27 (platform) used by the crowd-agent 28 that generated the worker-unit entity 25.

**Worker-Annotation-Vector** A worker-annotation-vector contains the datastructure, i.e. the vector, based on which the CrowdTruth disagreement metrics can be computed. One worker-annotation-vector always belongs to one particular worker-unit. Because there is a one-to-one mapping between the worker-unit and the worker-annotation-vector, one might argue that this
vector could have been stored directly in the worker-unit itself. Although this is true, we are interested in capturing the transformation step that occurred when generating this particular vector for provenance purposes. Therefore, we store a new entity \(29\) the so-called worker-annotation-vector, for every previously created worker-unit. This allows us to capture information about the activity \(30\) and the agent \(32\) that generated this particular worker-annotation-vector.

The content of a worker-annotation-vector depends on the input, i.e. the worker-unit, and the task-template that was used during the annotation. Thus, the documentType for this particular entity is factspan-wav. Next, we add the tags: worker, annotation, vector. The parents attribute records the reference to the particular worker-unit \(25\) that this worker-annotation-vector belongs to. In addition, we store all the previous references that were available in the worker-unit parent as well, such as the baseUnit_id, mediaUnit_id and taskUnit_id attributes. The crowdtruth-factspan-adapter software component \(31\) uses the worker-unit and task-template from a particular job to generate the worker-annotation vectors. The agent \(32\) responsible is the crowdtruth-system software-agent, as the running of this activity may be triggered by an automatic autonomous process. Finally, the activity \(30\) records the agent, and the input and software-component used during this process.

Corrected Media-Units We have previously established that a media-unit is a base-unit along with some metadata from which specific task-units can be created. A corrected media-unit is a base-unit along with some piece of metadata which is the result of a crowdsourcing annotation task. In the case of a FactSpan task, the metadata would be the factors deemed correct by the crowd and subsequently the metrics, which identify the high quality annotations. Because the results of annotation tasks are often used as the input for new annotation tasks, we create new media-units in order to maintain consistency. As specific task-units are always derived from media-units. These newly generated media-units however, are corrected because they contain a corrected piece of metadata.

Thus, we create the corrected media-unit entities with the tags: media, unit, corrected. In addition, we add a new tasks attribute to this entity, wherein we store the name of the task, FactSpan, to be able to identify what annotation task this media-unit resulted from. The media-unit entity \(33\) references the worker-units with the parents attribute to resolve the wasDerived-From relation. Furthermore, we store references to the base-unit and the previous media-unit in the lineage, with respectively the baseUnit_id and mediaUnit_id attributes. This allows us to easily perform aggregation and provenance queries to see how the particular unit changed after every annotation task.

The activity \(34\) which generates this new corrected media-unit, needs to have knowledge about the input used during the annotation task, i.e. task-unit \(16\) the result from the annotation, i.e. worker-unit \(24\) and finally the previous media-unit \(13\) to which the corrected result can be added. Note, that we do not add the corrected metadata to the previous media-unit \(13\) but rather only the
newly generated media-unit Finally, we store a reference to the responsible agent via the agent_id attribute in both the entity and activity. The ibm-media-sentence-generator software component is referenced in the activity.

We have now completed one annotation workflow, wherein we have run a FactSpan task. To recap, we initially ingested a raw ibm-medical-sentences CSV file, and stored this as a raw-unit entity. Next, we identified the base-units, i.e. the unique records of content, and the media-units available in the initial raw-unit. From the media-units we created specific FactSpan task-units based on filtered properties, which were then aggregated into a batch entity. After which, we created a job entity, with specific parameters such as the aforementioned batch, task-template, and specific job settings. The crowdsourced results from this job are worker-unit entities from which we generate worker-annotation-vectors, which are used by the CrowdTruth disagreement metrics. Finally, based on the high quality annotations, identified by the CrowdTruth metrics, we generate new corrected media-units which now contain the input for any new specific annotation tasks, i.e. task-units.

At every transformation step, i.e. generation of new entities, we store and reference the associated activities and responsible agents, to capture provenance as defined by PROV. For the next IBM Watson medical use case, Relation Extraction (RelEx), the specific RelEx task-units would be derived from the newly generated media-units. The rest of the steps in the annotation workflow would be similar as the one just described, until the next set of corrected media-units are created which result from the RelEx annotation task.
3.6 Virtual Entities (Combined Tasks)

*Virtual Entities* are entities which are created via a simulated model, in order to maintain the cardinalities and relations between the various entities as we have described in the previous sections. Specifically, these are:

- A *task-unit*, always only contains the input to **one** type of annotation task.
- A *job*, is similarly always associated with only **one** type of annotation task.
- A *worker-unit*, always belongs to only **one** specific *task-unit*.

Let us now look at combined tasks, which violate these properties. In essence, the use of combined tasks is to allow a worker to simultaneously perform more than one type of annotation task, because there may be some overlapping or dependent elements such as the input used. Thus, by combining such tasks, less time may be spent and costs can be decreased, as e.g. the input only has to be read once. We identify two types of combined tasks:

- **Dependent Combined Tasks.** The output of one type of annotation task forms the input of some other type of annotation task. I.e. the latter task is dependent on the former.
- **Independent Combined Tasks.** The different tasks are independent of each other. Rather, they are only combined because they may share some overlapping elements such as the input, e.g. text or image used.

**Dependent Combined Tasks** Let us consider the example of a *RelExDir* task, wherein the crowd is given the combined task of relation extraction and direction on the output from *FactSpan* [9]. *RelExDir* is an example of a dependent combined task, because identifying the direction of a particular relation can only be done after said relation has been extracted. As we have already elaborately explained all the unit-states in the previous sections before, we will only focus on the additional properties that are needed to support combined tasks. And also, how the violated properties in this combined annotation workflow can be amended.

The input to a *RelExDir* task is an *ibm-medical-relexdir-sentence*, which is *task-unit* entity. We derive this entity from the previously corrected *media-unit* that resulted from the *FactSpan* task. Because this new combined *task-unit* contains the input to **more than one** task, we add the **tags**: *task*, *unit*, *combined*. Similarly, we add the tag **combined** to the associated: *batch*, *job* and resulting **worker-units** for this annotation workflow.

Note, that we do not create the **worker-annotation-vectors** for combined **worker-units**. Rather, the combined tag implies that the entities in this particular annotation workflow need to be separated into their own singular task entities before we can create the **worker-annotation-vectors**. Thus, the resulting singular task entities are generated via a virtual simulated model. Specific to a *virtual* entity is the addition of the tag *virtual*, and a reference to the
combined entity from which it was simulated with the additional virtualOf
attribute.

We begin in order by creating the first virtual task-unit, RelEx, from the
combined RelExDir task-unit. The resulting ibm-medical-relex-sentence
task-unit holds the tag virtual and only contains the minimal content needed
to perform a RelEx task. In addition, we store a new attribute: virtualOf, which
references the combined task-unit from which it was generated. Note, that the
combined task-unit is not the parent to the virtual task-unit, rather the
virtual task-unit is still derived from the same corrected media-unit. The rest
of the properties in the virtual task-unit stay the same as they were in the
combined task-unit.

Next, we create a virtual batch which aggregates the ibm-medical-relex-sentence
virtual task-units. Similarly, we create a virtual RelEx job from the combined
job. For both the batch and job we store the tag virtual, and reference their
respective combined entities with the virtualOf attribute. We also update any
references if needed: e.g. the virtual job actually references the virtual batch
with the batch_id and not the combined batch.

From the resulting combined RelExDir worker-units, we can now create
the virtual RelEx worker-units. Again we store the virtual tag, virtualOf
attribute, and we update any specific references such as the specific virtual
job and task-unit to which this worker-unit belongs to. Once this is done,
we can now create any specific worker-annotation-vectors from the virtual
worker-units as they now only contain the annotation for a single task. Based
on these worker-annotation-vectors, the CrowdTruth disagreement metrics
can be computed, high quality annotations can be identified, and we can subsequently create new media-units from which specific task-units can be derived
for the next annotation task.

The observant reader will have noticed that we have only created virtual
entities for one task, RelEx, up until this point. The reason for this is that the
next annotation task, RelDir, is dependent upon the results from RelEx. Thus,
by simulating a virtual RelEx annotation workflow, we can now generate the
corrected media-units from which we can derive the task-units for the next
task: RelDir.

We continue explaining dependent tasks by focusing on the second task,
RelDir, in the combined RelExDir task. From the corrected media-units which
resulted from the virtual RelEx annotation workflow, we can now create specific RelDir task-units. These task-units hold the tag virtual and refer to
the combined task-unit with the virtualOf attribute. The parent to these
task-units is the corrected media-unit from which it was derived.

We aggregate the RelDir task-units, into a virtual batch. A newly generated virtual job uses this virtual batch along with the specific RelDir task-
template. The resulting virtual RelDir worker-units from this job, are used
to create the specific RelDir worker-annotation-vectors. The CrowdTruth
disagreement metrics can now run on these worker-annotation-vectors, iden-
tify the high quality annotations, and subsequently allow us to generate new corrected media-units which result from the RelDir task.

**Independent Combined Tasks** Independent combined tasks are similar to dependent combined tasks in the sense that they too are composed of combined and virtual entities. However, in the case of an independent combined task there is no notion of an order, as no task relies on the output of another task.

Let us consider a Location Extraction (LocEx) task and a Participant Extraction (PartEx) task, from some arbitrary news article. These two tasks share the same input, and can thus be combined into one task to save time spent on reading the input and subsequently cost spent on the whole crowdsourcing annotation task. The annotation workflow for this combined task starts with a task-unit which has the documentType: locex-partex-newsarticle. We follow a similar annotation workflow, like before, by creating the associated batch, job and resulting worker-units for this combined annotation workflow. We now have violated properties in the combined entities that hold more than one task-type. We amend these violated properties by generating a virtual annotation workflow for every task in the combined annotation workflow: LocEx and PartEx. We begin by creating the virtual task-unit entities, locex-newsarticle and partex-newsarticle which are simulated from the combined task-unit. Similarly, we create the associated virtual batch, job and worker-units for the two separate tasks in the combined task. Based on the virtual worker-units, we can create the respective virtual worker-annotation-vectors which hold the vectors for each type of annotation task. Finally, the CrowdTruth disagreement metrics can identify the high quality annotations and new corrected media-units can be generated.

4 Implementation

In order to ensure extensibility and openness we have implemented the CrowdTruth framework with open web standards in mind. CrowdTruth is built on top of the open-source PHP framework, Laravel\(^\text{12}\) which uses the Model-View-Controller (MVC) pattern to decouple application logic, data and presentation. It leverages built-in packages for authentication, routing, creation of templates and APIs. In addition, Laravel ships with the Eloquent\(^\text{13}\) object-relational-mapper (ORM), which allows for objects in the application layer to be mapped with entries in the database layer for easy CRUD operations via a pattern called Active Record. Because the built-in Eloquent ORM only supports a select amount of relational databases, we make use of an external Laravel-MongoDB package called Moloquent\(^\text{14}\) to add support for MongoDB databases.

\(^{12}\) [http://laravel.com/](http://laravel.com/)

\(^{13}\) [http://laravel.com/docs/4.2/eloquent](http://laravel.com/docs/4.2/eloquent)

\(^{14}\) [https://github.com/jenssegers/Laravel-MongoDB/](https://github.com/jenssegers/Laravel-MongoDB/)
In addition, we make use of *Twitter Bootstrap*\(^{15}\), *jQuery*\(^{16}\) and *Handlebars*\(^{17}\) for the front-end of the framework. Furthermore, all PHP packages are pulled, or consumed, via a PHP dependency management tool called *Composer*\(^{18}\). Finally, we have implemented the CrowdTruth framework and all its dependent packages to run in a virtual *Vagrant*\(^{19}\) environment, which ensures that the framework can run on any operating system with the exact same dependencies.

We have previously made mention of how every *model* in the application layer, corresponds with a *collection* in the database layer. The Active Record pattern used within the framework specifies that the models are the only entry points to anything that is related to data manipulation. For example, an instance of the *entity* model in the framework, results in a data object which corresponds to a specific document in the *entities* collection in the database. Thus, to resolve the relations between the various documents stored in the database, the models in the application layer must have relationships defined between them.

MongoDB makes use of two types of relationships\(^{20}\): documents may *embed* each other, or documents may *reference* each other. More specifically, this means that a model: *embedsOne*, *embedsMany*, *referencesOne* or *referencesMany* other model(s). Figure 4 shows the CrowdTruth BDM and the types of MongoDB relationships that are used to model the various PROV relations. We elaborate upon these relations in the next evaluation section.

---

**Fig. 4: CrowdTruth Base Data Model: MongoDB relationship types and PROV relations**

\(^{15}\) [http://getbootstrap.com/]

\(^{16}\) [http://jquery.com/]

\(^{17}\) [http://handlebarsjs.com/]

\(^{18}\) [https://getcomposer.org/]

\(^{19}\) [http://www.vagrantup.com/]

\(^{20}\) [http://docs.mongodb.org/manual/core/data-model-design/]
5 Evaluation

In order to assess the accessibility, extensibility and usability of the CrowdTruth Data Model, we perform three types of evaluation. First, we perform a qualitative analysis to assess whether we indeed have a provenance enabled storage as defined by PROV. Next, we perform a comparative analysis between two iterations of the CrowdTruth Data Model, to show its evolution and to explain why particular modeling decisions were made. Finally, we conduct a user study with end users of the framework, our goal here is to assess the usability and extensibility of the framework and indirectly the data model.

5.1 Qualitative Analysis: PROV

To ensure that we have indeed captured provenance as defined by PROV, our data model must support the querying of any PROV class (Entity, Activity and Agent). Moreover, the relations between these PROV classes, or models, must be resolvable from all three provenance perspectives. Finally, it may be the case that we are only interested in querying partial provenance. To assess whether this is possible, we must define our experimental tasks to the finest granularity possible. The focus of this experiment is to assess the accessibility and extensibility of our model.

Experimental Setup We can specify the provenance requirements as defined by PROV with the following experimental tasks. Again, here we make the distinction between the various PROV classes (models), to assess whether accessibility is possible from every provenance perspective (object, process and agent). We furthermore omit any domain specific knowledge for these experimental tasks to ensure they are domain-independent.

- **Entity**
  - T1.S1: How can we retrieve a specific Entity?
  - T1.S2: From which Entities was the current Entity derived?
  - T1.S3: Which Activity generated this Entity?
  - T1.S4: What Agent was attributed to the creation of this Entity?

- **Activity**
  - T2.S1: How can we retrieve a specific Activity?
  - T2.S2: Which Entities were used as input during this Activity?
  - T2.S3: What SoftwareComponent was used during this Activity?
  - T2.S4: What Agent was associated with performing this Activity?
  - T2.S5: Which Entities resulted from running this particular Activity?

- **Agent**
  - T3.S1: How can we retrieve a specific Agent?
  - T3.S2: For which Entities was this Agent responsible for its creation?
  - T3.S3: For which Activities was this Agent responsible for its running?
  - T3.S4: What platform does this Agent work on?
Results In this section, we translate the aforementioned experimental tasks into valid queries to retrieve the necessary data. The output of these queries are very similar to the JSON examples in the appendix. However, because our goal is to assess whether these experimental tasks will work for any domain, we focus on the domain-independent queries instead rather than the JSON output results. In the next sections, we discuss how these queries work, after which we discuss why this was done in this particular way.

- Entity
  - T1.S1: $entity = Entity::where('key', 'operator', 'value')->first();
  - T1.S2: Entity::with('wasDerivedFrom')->get();
  - T1.S3: Entity::with('wasGeneratedBy')->get();
  - T1.S4: Entity::with('wasAttributedTo')->get();

- Activity
  - T2.S1: $activity = Activity::where('key', 'operator', 'value')->first();
  - T2.S2: Activity::with('used')->first();
  - T2.S3: Activity::with('usedSoftwareComponent')->first();
  - T2.S4: Activity::with('wasAssociatedWith')->first();
  - T2.S5: Entity::where('activity_id', '=', $activity->id)->get();

- Agent
  - T3.S1: $agent = Agent::where('key', 'operator', 'value')->first();
  - T3.S2: $entities = Entity::where('agent_id', '=', $agent->id)->get();
  - T3.S3: $activities = Activity::where('agent_id', '=', $agent->id)->get();
  - T3.S4: Agent::with('worksOn')->first();

Analysis In this section, we specifically explain how the exact queries work and how the relations are resolved.

- Entity
  - T1.S1: In order to retrieve a specific Entity, we must make use of the Entity model and specify any required properties in the where-clause. A key, in the where-clause, specifies the JSON attribute which will be checked for the given operator and value. Finally, we may retrieve the first Entity which satisfies this clause by using the first() method, or all entities by using the get() method.
  - T1.S2: The with method specifies that there is a relation that must be resolved. Specifically for this query, it is the wasDerivedFrom relation. As we can see in Figure 4, this relation ReferencesMany other entities. To resolve this relation we specify a wasDerivedFrom function in the Entity model which will be resolved when it is used in the with method. This function defines that the parents attribute in the local Entity model, ReferencesMany foreign Entity models via their _id attributes.
  - T1.S3: To resolve the wasGeneratedBy relation, we specify a function in the Entity model which ReferencesOne activity. The Entity model is associated to the Activity model via the local activity_id and a foreign _id attribute.
- T1.S4: The wasAttributedTo relation is resolved by a function in the Entity model. The local agent.id attribute in the Entity model ReferencesOne_id attribute in the Agent model.

- Activity
  - T2.S1: A specific Activity can be retrieved by specifying where-clauses, like before, and subsequently appending either a first() or get() method.
  - T2.S2: The used relation is resolved by a function in the Activity model. The local used attribute in the Activity model ReferencesMany_id attribute in the Entity model.
  - T2.S3: The usedSoftwareComponent relation is resolved by a function in the Activity model. The local softwareComponent.id attribute in the Activity model ReferencesOne_id attribute in the SoftwareComponent model.
  - T2.S4: The wasAssociatedWith relation is resolved by a function in the Activity model. The local agent.id attribute in the Activity model ReferencesOne_id attribute in the Agent model.
  - T2.S5: An Entity is always generated by some particular Entity. Thus to retrieve a subset of entities which were generated by some particular Entity, we merely have to add a where-clause which specifies a particular activity.id when querying for entities.

- Agent
  - T3.S1: A specific Agent can be retrieved by specifying where-clauses, like before, and subsequently appending either a first() or get() method.
  - T3.S2: An Entity is always attributed to a particular Agent. Thus to retrieve a subset of entities which were attributed to some particular Agent, we merely have to add a where-clause which specifies a particular agent.id when querying for entities.
  - T3.S3: An Entity is always associated to a particular Agent. Thus to retrieve a subset of activities which were associated to some particular Agent, we merely have to add a where-clause which specifies a particular agent.id when querying for activities.
  - T3.S4: The worksOn relation is resolved by a function in the Agent model. The local softwareComponent.id attribute in the Agent model ReferencesOne_id attribute in the SoftwareComponent model. Platforms, such as AMT and CrowdFlower, are modeled as SoftwareComponents within the CrowdTruth framework.

Discussion As can be seen in Figure[4], we have favored referencing to resolve the various PROV relations over embedding the various models. Embedding is a form of denormalization which fits data well that is read often and updated scarcely. Provenance data generally fits these characteristics, as such data is also static and not updated. At the same time, however, provenance also has hierarchical and graph characteristics such as nodes (documents) and edges (relations). Thus, in order to still maintain maximum query-ability we have modeled the various models to reference rather than embed each other.
Let us consider the example of entities embedding other entities with every transformation, rather than having the various entities reference each other. Such a model would result in deeply nested complex structures, which are difficult to query, as the JSON keys may now be n-levels (unknown) deep. The application layer needs to have knowledge of the location of the JSON keys in order to be able to query for particular documents. By having the documents reference, rather than embed each other, we can always assume that they are first-class citizens of their respective collections. This allows us to assume that the keys described in the CrowdTruth Data Model are always available as first level attributes.

Note, that referencing is considered a form of normalization. However, we only normalize the four models mentioned in the CrowdTruth BDM: Entity, Activity, Agent and SoftwareComponent. More specifically, this means that the structure of an individual document can be denormalized to any extent long as it remains a first-class within its own collection. In this way we are still able to make use of the static nature of provenance data, while still remaining completely accessible.

Next, unnecessary duplication has been prevented by having the various models reference rather than embed each other. For example, one might argue that the activity model could have been embedded inside the entity model, as an entity always has a one-to-one mapping with an activity. Although this is true, an activity may generate n-entities. This would mean that all the generated entities would now embed the exact same activity, resulting in unnecessary duplication. The same argument can be made for the agent model, as an agent is always associated with an activity and always attributed to an entity. Again, embedding here would cause redundancy, as updating a particular agent attribute, e.g. email address, would require updating every single entity and activity. Furthermore, because we are also interested in querying for partial provenance, referencing is preferred. As the various PROV classes are then stored as self-contained documents which do not have to be sliced when accessing the data, as is the case when they are embedded.

In terms of extensibility, keeping the models as distinct separate collections is also preferred. For example, a user permission/role system within a framework requires the validation of so-called allowed permissions and required permissions. Thus, within the agent model we can store the allowed permissions to perform a certain activity, and within e.g. a particular software component model we may store the required permissions for specific access controls.

Thus, by keeping the core PROV classes as distinct models and collections which reference each other, we have a good balance between normalization and denormalization. The normalization between the various models allow for complex queries to be performed as every PROV class is in its own self-contained document, however, within such a self-contained document denormalization is still allowed to fit any type of data. This results in completely accessible, i.e. query-able, provenance which is both extendible in the application and in the database layer.
5.2 Comparative Analysis

In order to explain the evolution and specific modeling decisions of our design, we have conducted a comparative analysis between two iterations of the CrowdTruth Data Model. For the sake of brevity, we have previously only made mention of the final iteration of the data model. Thus, we first begin by outlining the differences between the two iterations in the experimental setup, and then discussing the consequent results afterwards.

Experimental Setup  Within the entity model, the parents attribute was previously modeled as a string type attribute called parent_id. Furthermore, the tags attribute was not captured. Also, throughout the Instantiated Data Model, we did not store any baseUnit_id, mediaUnit_id, and taskUnit_id attributes. Next, the current Agent model did previously not exist. Rather, it was split into three individual models per agent-type: UserAgent, SoftwareAgent, CrowdAgent; which mapped to their own respective MongoDB collections. Finally, within the activity model, the used attribute was not captured.

Results and Discussion  In the previous iteration of the data model, we had made the assumption that an entity, could only be derived from one and only other entity. Although this is generally the case, there are exceptions where an entity may be derived from more than just one entity. To support such functionality, we replaced the initial parent_id string attribute which referencesOne entity, with the parents array attribute which referencesMany entities.

Next, because we previously had no notion of a tags attribute, it was not possible to distinguish which entities were units and which were not. Rather, only the application layer had knowledge of which specific documentTypes were units. For example, an ibm-medical-media-sentence is a unit, however an entity with the documentType job is not a unit. Thus, in essence, the application layer had too much knowledge of the data being stored inside the database. This resulted in static, tightly coupled, code in the application layer which is not good for extensibility. To resolve this we introduced the tags attribute in the data model, which allows us to specify exactly which entities are units within the database itself in an abstract manner, without inferring this from the documentTypes in the application layer.

The tags attribute proved to be furthermore useful for labeling the various states of the entities, such as: raw, base, media, task, worker. Previously, these states were inferred from their documentTypes, like before, or they were simply not captured at all. Consider the example of an ibm-medical-media-sentence, which contains a sentence and two terms. For a FactSpan task, however, only the sentence and one term is used. Thus, it was not possible to refer to an entity in the provenance lineage to identify exactly what the input was to that particular annotation task, as the so-called task-units did not exist yet. The tags attribute, makes the identifying of these various states possible in an easy and extendible way.
The addition of the `baseUnit_id`, `mediaUnit_id` and `taskUnit_id` attributes allow for easy aggregation by only performing one query, which was previously not possible. Let us consider the following query:

```php
Entity::where('baseUnit_id', '=', 'value')->get();
```

This particular query, will retrieve all entities which have a `baseUnit_id` field with the specified operator and value. Because this key is stored in various entities throughout a particular annotation workflow, we are able to aggregate the different unit states with only one query. In fact, we may even further filter the aggregated results by specifying some of the aforementioned tags:

```php
Entity::where('baseUnit_id', '=', 'value')->where('tags', 'all', ['corrected', 'media', 'unit'])->get();
```

This particular query retrieves the `corrected media-units`, which have the same `baseUnit_id`. Corrected media-units are entities which have a corrected piece of metadata which is the result of a crowdsourcing annotation task. The resulting entities from the last query, subsequently hold a `mediaUnit_id` attribute, wherein they refer to the previous `media-unit` in the lineage. Thus, the order of the tasks that were run is also preserved, without having to infer this from the transitive `parents` attribute.

We had previously split up the current `Agent` model into the three agent-type models: `UserAgent`, `SoftwareAgent`, `CrowdAgent`; as we believed these three types to be distinctive enough to be stored into their own MongoDB collections. However, this initial assumption made modeling the PROV relations unnecessarily difficult, as the relations now had to be specified further for every agent-type. Consider the PROV `wasAttributedTo` relation, because of the three previous agent-type models, this relation now similarly needed to be split up into three distinctive `wasAttributedToUserAgent`, `wasAttributedToCrowdAgent`, `wasAttributedToSoftwareAgent` relations. In essence, this meant we were slightly changing the original PROV relation. We resolved this by combining all the agents into one `Agent` model and distinguishing between them via a `type` attribute. Again, here the flexibility of MongoDB shows, as documents with (semi) different structure are allowed to be stored in the same collection.

Finally, in the previous iteration of the data model, we did not capture the input `used` for an activity, as we believed that this was already captured within the `parents` attribute inside the entity. Thus, by omitting the `used` attribute inside the activity, we could potentially avoid redundant references. This assumption, however, was only partly correct. Any references inside the `parents` attribute within an entity are also `used` by the associated activity. However, the activity may store additional references which are not stored within the `parents` attribute, such as the task-template used. In essence, this meant that we were capturing object-centered provenance but only incomplete process-centered provenance. We amended this wrong assumption by capturing the complete input `used` inside the activity with the `used` attribute, even though this often overlaps with the `parents` attribute in the resulting generated entities.
5.3 User Study

The previous two evaluation experiments primarily focused on gaining technical insight into the CrowdTruth Data Model. However, we are also interested in gauging the perception of the end-user, as this may be different from that of the developer. To achieve this, we have conducted a user study, focusing on the various components used throughout the annotation workflow, which (in)directly make use of the CrowdTruth Data Model.

Experimental Setup To evaluate the design described in Section 3, we set up an experiment with both internal and external users of the CrowdTruth framework. The goal of the experiment was to assess the usability, extendibility and usefulness of the CrowdTruth framework and indirectly the CrowdTruth Data Model. To achieve this, the users were asked to perform one complete annotation workflow, as similarly described in the instantiated data model Section 3.5, with a think-aloud protocol.

Participants In total, there were seven active members collaborating on the CrowdTruth project. However, four members were heavily involved during the design and development process, and were therefore not eligible as objective participants for the evaluation. Thus, the user study consisted of three internal users and six external users who were invited to participate because of their experience in the crowdsourcing field.

Settings The user study lasted for about a half an hour and was composed of four parts:

(a) survey to assess the experience of the users with annotation systems. The survey was filled in online by the participants prior to the experiment (10 minutes)
(b) a brief introduction of the CrowdTruth framework (5 minutes)
(c) performing a set of tasks within the CrowdTruth framework which constituted the completion of one annotation workflow (10 minutes)
(d) assessment survey, filled in after completing the set of tasks (5 minutes)

The pre-test survey, gives insight into the level of experience the participants have with crowdsourcing methods. This allowed us to assess whether they were eligible to participate in the experiment, and to what extent the results were generalizable. Because all participants had access to the internet, we opted for an online survey, this allowed them to fill in the survey within their own schedule and at their own pace.

The rest of the user study was performed actively together with the participants. Six of the nine participants performed the user study offline in person, and the other three online due to distance constraints. In order to similarly understand the behavior of the online three online participants, they were asked to share their screen and to also make use of a think-aloud protocol.
The users were first presented with a brief introduction wherein the purpose of the interview, the CrowdTruth framework and the CrowdTruth disagreement metrics were explained. Next, the participants were asked to perform a set of tasks which correspond with the completion of one annotation workflow. The set of tasks ensured some control over the users’ activity while still allowing them to explore the system throughout every phase.

Finally, in order to avoid any observer bias and to capture direct feedback from the participants, a post-test survey was filled in to obtain information about the set of tasks that were performed. The survey was semi-structured, with open-ended questions, and focused on the usability, extendibility and usefulness of the various components used throughout the annotation workflow.

Tasks  Previously in Section 3 and more specifically in the Instantiated Data Model Section 3.5, we have described how data is modeled at different stages of the annotation workflow. In order to gauge the perception of the end-user about these very same stages, we opted to have the participants complete a similar annotation workflow. The set of tasks consisted of using a series of software components within the CrowdTruth framework to ingest data, up until creating a job on a crowdsourcing platform. Specifically these tasks, related to their respective components, are:

- **T1 : Ingestion of data**
  - The user is asked to upload an ibm-medical-sentences CSV file by means of the FileUploader SoftwareComponent. On the upload page, the user is presented with an interface where he/she has to define the specific format, domain and documentType for this document.

- **T2 : Pre-processing of the ingested data**
  - The user is asked to pre-process his previously ingested document with the ibm-media-sentence-generator SoftwareComponent on the pre-processing page.

- **T3 : Filtering of the pre-processed data**
  - The user is asked to:
    - **T3.S1** Order the pre-processed ibm-medical-media-sentences by their highest number of words (descending order)
    - **T3.S2** Set the range of the allowed number of words in sentences between 20 (minimum) and 40 (maximum) words.
    - **T3.S3** Filter/find the sentences which have:
      - Relation between terms.
      - Without parenthesised terms.

- **T4 : Exporting the filtered data**
  - Export the previously filtered data into a CSV file.

- **T5 : Creation of a batch with the filtered data**
  - The user is asked to create a batch and to specify a title and description for his/her batch.

- **T6 : Creation of a crowdsourcing job**
  - The user is asked to create a crowdsourcing job, with his previously created batch. Here, the user is given freedom over the various parameters that are available for creating a job, such as the annotations, units and reward per task.
Results and Discussion

Participants’ pre-test survey In order to confirm the eligibility of both the internal and external users for participating in the experiment, we conducted a pre-test survey. Table 2 gives an overview of the summarized results from the survey. We can see that several annotation systems are used ranging from: Amazon Mechanical Turk (AMT), CrowdFlower (CF), but also custom frameworks such as Accurator, Waisda?, etc. The use cases for using these various systems range from: collecting ground truth data for machine learning purposes, entity recognition, enriching RDF description, research of the trustworthiness of the crowd, etc. Based on the diversity of the annotation systems, data modalities and scopes of analysis, the participants were deemed eligible for the experiment.

Performance on tasks

- **T1 : Ingestion of data**
  The FileUploader software component, was found to be very straightforward and usable. However, it lacked in terms of extendibility, as the user was only able to select some of the pre-defined options for the possible formats, domains and documentTypes. The initial reasoning for this limitation, was that only administrator/moderators should be able to extend this list with possible valid options. However, due to the feedback of the users, we plan on relaxing this constraint by allowing them to specify their own new options. Furthermore, we have previously mentioned how a documentType corresponds with the structure of the uploaded document. However, upon selecting a particular documentType, the user was not made aware of what this particular structure should look like, rather it was implied to be known. Thus, we plan on presenting examples of the different structures when uploading specific documentTypes, to make this more clear and meaningful.

- **T2 : Pre-processing of the ingested data**
  In a similar fashion, the ibm-media-sentence-generator software component was found to be straightforward and usable. However, it lacked in terms of extendibility, as this specific software component was only usable for ibm-medical-sentences. Despite that some of the generated properties, e.g. identifying the number of words in a particular CSV column, could be applied to other CSV documents as well.
  In order to alleviate this problem, we plan on creating a dynamic pre-processing component that does not rely on the given structure of the initially uploaded document. Rather, the user will be able to map specific functions, such as the word-count identifier, to specific columns in the initially uploaded file, regardless of its documentType.

- **T3 : Filtering of the pre-processed data**
  Filtering of the data was done by the crowtruth-search software component. This specific component was found to be very usable as it provides functionality for string text search, sorting, hiding and displaying of information, and filtering on specific properties. Rather, the main feedback given
<table>
<thead>
<tr>
<th>Participant number</th>
<th>Annotation frameworks used</th>
<th>Formats used in annotation tasks</th>
<th>Scope of analysis</th>
<th>Main activities inside the annotation systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Steve museum – Waisda? – Accurators</td>
<td>– text</td>
<td>Research to analyze trustworthiness of tags for images</td>
<td>– filter annotations based on quality – filter workers based on behavior – find user expertise for tasks – employing provenance information for trust</td>
</tr>
<tr>
<td>2</td>
<td>CrowdFlower</td>
<td>– text</td>
<td>Enriching RDF of TV program descriptions</td>
<td>– detection of low/high quality workers – identify the inter-annotator agreement</td>
</tr>
<tr>
<td>3</td>
<td>Gamified crowdsourcing application/framework</td>
<td>– text – binary data</td>
<td>Training of IBM Watson in the medical domain</td>
<td>– identify the inter-annotator agreement – find the amount of different answers to a question – track the progress of tasks that are related – view the annotations of one worker – compare the annotations of one worker with the crowd</td>
</tr>
<tr>
<td>5</td>
<td>– CrowdSearcher – CroKnow – Custom processing pipelines and scripts – R as a tool for statistical analysis.</td>
<td>– images – videos – text – binary data</td>
<td>Factual and subjective annotation of mostly photographs and prints for the purposes of ground truth data creation</td>
<td>– filtering – outlier detection</td>
</tr>
<tr>
<td>6</td>
<td>Accurator – Cliopatria (Prolog backend)</td>
<td>– images of artworks</td>
<td>To create annotated data for a data collection management system and search on museum websites</td>
<td>– trust assessment</td>
</tr>
<tr>
<td>7</td>
<td>CrowdFlower (CF), Amazon Mechanical Turk (AMT) and CrowdTruth metrics for analysis</td>
<td>– text</td>
<td>Training medical data for: – question-answering – named entity recognition – relation extraction for medical domain</td>
<td>– find low quality workers – filter outliers – analyze the worker agreement – analyze the quality of input data with annotations</td>
</tr>
<tr>
<td>8</td>
<td>Crowdtruth metrics for analysis and Crowdflower, Amazon Mechanical Turk to gather data</td>
<td>– text</td>
<td>Align questions and passages that can contain the answer for IBM Watson</td>
<td>– filter out passages based on their length and contents – sort passages based on their length – group passages – find low quality workers – find answers – find relations</td>
</tr>
<tr>
<td>9</td>
<td>CrowdTruth (and indirectly AMT and CF)</td>
<td>text</td>
<td>Research of data from PatientsLikeMe website</td>
<td>identify if annotated and unannotated sentences contain terms from vocabularies</td>
</tr>
</tbody>
</table>

Table 2: Summarized results of the pre-test survey
here was that performed searches should be savable into a configuration, which can be reused in the future.

- **T4 : Exporting the filtered data** The crowdtruth-export SoftwareComponent currently allows filtered data to be exported into CSV format. The users were generally satisfied with this, as crowdsourcing platforms, such as CrowdFlower and Amazon Mechanical Turk, primarily use the CSV format for their input data as well. However, there currently is no support for specifying the CSV delimiter and encoding for the output documents. We plan on amending this by allowing the users to specify these parameters, and to also allow for export in JSON format, as it is already saved in the database with such a structure.

- **T5 : Creation of a batch with the filtered data** The batchcreator SoftwareComponent allows the aforementioned filtered results to be aggregated into a batch entity which may then be reused across jobs. The users currently only have to specify a batch title and description, which was found to be sufficient. The main feedback given here was that provenance information, related to the previous search filters applied, should be automatically saved as well.

- **T6 : Creation of a crowdsourcing job** The jobcreator SoftwareComponent was found to be straightforward and usable from the end-user’s perspective. The main feedback here was that selecting a batch was somewhat troublesome, as the current list of batches does not provide any search functionality like the one in the crowdtruth-search software component. Rather, the list of batches are only ordered by their date, which meant the users needed to scroll until they found their batch. Here, we may reuse the crowdtruth-search software component, to allow for searching of batches based on title/description filtering and sorting.

**Users’ assessment of the framework** As we already have enumerated the specific feedback received for every task in the previous section, we now summarize the users’ feedback for the three different perspectives and give average scores on a scale from 1 (lowest) to 5 (highest).

The various software component used throughout the set of tasks were found to be very usable and received a score of four. In essence, the purpose of every component was self-explanatory and straightforward. Rather, there was an initial learning curve, as the users needed some time to get familiar with every component. The extendibility, received a score of three, which leaves room for improvement. We identify the main limiting factors for the extendibility to be the user interfaces which restricted upload options to a certain limit, and (pre-processing) software component which were too domain specific. The CrowdTruth Data Model in and of itself does not pose any restrictions on the data stored or accessed. Rather, the user interfaces and the software component that lie in between the data stored and the end-user were not completely extendible enough. Finally the usefulness received a score of four. Although the various tasks and software components were found useful, more documentation throughout the framework was needed.
5.4 Limitations

We have previously made mention of how our approach is similar to the official MongoDB documentation for modeling tree structures with parent references\footnote{http://docs.mongodb.org/manual/tutorial/model-tree-structures-with-parent-references/}. Our approach differs in that it allows for referencing of multiple parents rather than just one parent, this way provenance can still be captured for entities that were derived from more than one parent entity. However, this approach has a downside in that it limits some built-in querying functionality that was previously possible, namely the direct querying via an ancestor path.

Let us consider a provenance lineage wherein the entities can at most have one parent entity. In such a model, a singular string \texttt{parent} attribute would suffice, rather than the array \texttt{parents} attribute which we are currently using. Based on the parent references throughout that provenance lineage, we may then generate and store an array \texttt{ancestor} attribute which captures the complete provenance or ancestors for that particular entity. This is also called modeling trees with an array of ancestors\footnote{http://docs.mongodb.org/manual/tutorial/model-tree-structures-with-ancestors-array/}. Such a model allows for easy querying of descendants which are derived from some particular ancestor(s), because one element in the \texttt{ancestors} attribute represents one parent generation. With the multiple array \texttt{parents} attribute which we use in our approach to capture complete provenance, a simple array \texttt{ancestors} attribute cannot be generated anymore. This is because the assumption that one element in the \texttt{ancestors} attribute always represents one and only one previous parent does not hold anymore, as a previous generation may now be comprised of more than one parent. In order to alleviate this problem we have opted to store the \texttt{baseUnit\_id}, \texttt{mediaUnit\_id} and \texttt{taskUnit\_id} to still allow for direct querying of descendants which belong to a particular \texttt{base-unit}, \texttt{media-unit} or \texttt{task-unit}. For the other cases, we may always still fall back on traversing provenance via the transitive \texttt{parents} attribute.

Furthermore, in order to support future unknown data sources, the CrowdTruth data model only imposes some semi-structure and allows for additional new JSON keys to be stored. This flexibility comes at a drawback in that the application layer needs to have knowledge about the new specific JSON keys which can be queried upon. We recognize this as an inherent drawback for allowing such flexibility to support new data modalities. A potential solution to this problem is that new data sources must be changed into a known structure via an adapter before they are stored, or the application layer must be given additional knowledge about the new JSON keys if they need to be queried upon.
6 Future Work

In this section we describe future work which can be designed and implemented to further improve the extensibility and usefulness of the CrowdTruth Data Model. Note that we have already mentioned specific improvements that are to be done on the various existing software components in the user study. Thus, here we focus on two main additions.

First, we have shown how we have captured provenance according to the PROV model and its various classes. However, the export of this provenance knows specific structures or serializations as well. These PROV serializations provide a standardized structure which enables the interchange of provenance information in heterogeneous environments, i.e. different implementations. Examples of PROV serializations are: PROV-DC, PROV-O, PROV-XML, PROV-N, PROV-JSON. Here, we may add support for one or all of these serializations within the CrowdTruth framework. Adding support for PROV-JSON would be the most simple, however, as our current data is stored in a similar (BSON) structure. The exported PROV-JSON serialization may then be converted to other PROV serializations by means of tools such as ProvToolbox.

Second, adding support for the Open Annotation Data Model (OA), which aims to provide a standard description to share Annotations between systems, may be a useful addition to the CrowdTruth Data Model as well. The OA model is similar to the PROV model in the sense that it too tries to enable interoperability and interchange of information (annotations) between heterogeneous systems. As can be seen in Figure the OA model consists of an Annotation Class which is associated with a Body and a Target. Here, the Body is the comment or descriptive resource, and the Target is the object which is being annotated. In essence, we have already captured these specific entities within the CrowdTruth Data Model. These are respectively: the task-unit (target) and the worker-unit (body). Thus, adding support for the OA model primarily becomes a matter of mapping specific entities to OA vocabulary items.

[^23]: http://www.w3.org/TR/2013/NOTE-prov-dc-20130430/
[^24]: http://www.w3.org/TR/2013/REC-prov-o-20130430/
[^25]: http://www.w3.org/TR/2013/NOTE-prov-xml-20130430/
[^26]: http://www.w3.org/TR/2013/REC-prov-n-20130430/
[^27]: https://provenance.ecs.soton.ac.uk/prov-json/
[^28]: http://lucmoreau.github.io/ProvToolbox/
[^29]: http://www.openannotation.org/spec/core/
7 Conclusion

This document describes the design and development of the CrowdTruth Data Model, which supports the capturing of a variety of data and its provenance by utilizing the PROV model [10]. We argue that capturing provenance for annotated data is a necessity as it provides a means for justification of results. More specifically, capturing provenance means capturing the input, actions and output of a particular process along with its responsible agents. Based on this metadata, replicability of results and assessments of trust based on e.g. high quality annotations becomes possible.

The requirements for the CrowdTruth Data Model were elicited from literature and are primarily based on the PROV model. The PROV model was particularly chosen as it is a proposed W3C recommendation which is based on extensive research done by the W3C Provenance Incubator Group[21]. By implementing such a standardized model, interoperability and exchange of provenance information between different heterogeneous applications becomes possible. Next to this, specific design decisions were added as additional requirements, such as the domain-independent nature of the data model and the use of open-source tools. For these particular design decisions, we did not impose any particular constraints on the data model. Rather, the challenge here became supporting a variety of unknown data types and domains for future use.

We presented the CrowdTruth Data Model in the form of a Base Data Model (BDM) 3.4 and Instantiated Data Model (IDM) 3.5. The BDM serves as a domain-independent model which only describes the abstract classes, relationships and semi-structure needed, to instantiate and capture any type of data along with its provenance. The IDM gives a detailed overview of how one such instantiation can be done with domain specific knowledge, along with its specific input, actions, output and responsible agents throughout an annotation workflow.
The NoSQL MongoDB database was chosen to store all data and provenance because of its flexible dynamic schema and denormalization approach. The flexible schema is especially useful for supporting future unknown data types and domains, and the denormalization approach used fits provenance data well as such data is of a static and self-contained nature. The data model was implemented within the context of the CrowdTruth framework, which is built on top of the open-source Laravel framework. Laravel was chosen for its openness, extensibility and support for data accessibility via its Active Record pattern.

Evaluation of the CrowdTruth Data Model was split into three parts: a qualitative analysis focusing on PROV, a comparative analysis focusing on the evolution of the data model, and finally a user study to gauge the perception of the end-user throughout the framework. The qualitative analysis showed how provenance is fully captured and accessible via all three PROV classes. Given any PROV class, it is possible to query for both complete provenance or partial provenance. The comparative analysis shows how the data model was improved after the first iteration of its design. Specifically, here we show how support for unit states was added via a tagging system, while not having to specify any domain specific knowledge. Furthermore, we show how aggregation queries can be used to retrieve a subset of units which belong to a particular unique piece of initial content. Finally, in the user study we have shown how we have implemented the CrowdTruth Data Model within an actual usable setting within the CrowdTruth framework. The conclusion of the user study was that while there are usability and restriction problems identified by the end-users, these all pertained to the user interfaces that were used rather than the data model itself.

In conclusion, the CrowdTruth Data Model provides an abstract yet specific enough design which supports the capturing of provenance as described by PROV, while remaining accessible, extendible and domain-independent. In the future we plan to extend the CrowdTruth Data Model further by adding support for various PROV serializations and the Open Annotation Data Model.
References

6. Dumitrache, A., Aroyo, L., Welty, C., Sips, R.J., Levas, A.: "dr. detective": combining gamification techniques and crowdsourcing to create a gold standard in medical text
8 Appendix

Note The following figures serve as minimal examples to convey an understanding of the data model and its relations. Therefore, we have omitted irrelevant attributes such as specific SoftwareComponent attributes.

Fig. 6: Entity: Initial IBM-medical-sentences CSV file after ingestion into the database

Fig. 7: Activity: Generated the raw-unit in Figure 6
```json
{
  "_id": "agent/user/crowdtruth/khamkham",
  "firstname": "Khalid",
  "lastname": "Khamkham",
  "platformAgentId": "khamkham",
  "softwareComponent_id": "crowdtruth",
  "type": "user",
  "updated_at": ",",
  "created_at": ""
}
```

Fig. 8: Agent(user): Khamkham

```json
{
  "_id": "fileuploader",
  "label": "The FileUploader SoftwareComponent..."
}
```

Fig. 9: SoftwareComponent: FileUploader

```json
{
  "_id": "entity/text/medical/ibm-medical-base-sentence/0",
  "format": "text",
  "domain": "medical",
  "documentType": "ibm-medical-base-sentence",
  "parents": [
    "entity/text/medical/ibm-medical-sentences/0"
  ],
  "tags": ["base", "unit"],
  "content": {
    "val": "Lyme disease is a multi-system organ disease caused by Borrelia burgdorferi",
    "properties": {
      "wordCount": 11
    }
  },
  "agent_id": "agent/user/crowdtruth/khamkham",
  "activity_id": "activity/ibm-base-sentence-generator/0",
  "updated_at": ",",
  "created_at": ""
}
```

Fig. 10: Entity: IBM-medical-base-sentence generated from the raw-unit in Figure 6
Fig. 11: Activity: Generated the base-unit in Figure 10

Fig. 12: SoftwareComponent: IBM-base-sentence-generator
Lyme disease is a multi-system organ disease caused by Borrelia burgdorferi.

Fig. 13: Entity: IBM-medical-media-sentence generated from the raw-unit in Figure 6.
Fig. 14: Activity: Generated the media-unit in Figure 13

Fig. 15: SoftwareComponent: IBM-media-sentence-generator
Fig. 16: Entity: IBM-medical-factspan-sentence generated from the media-unit in Figure 13
Fig. 17: Activity: Generated the task-unit in Figure 16

Fig. 18: SoftwareComponent: IBM-factspan-sentence-generator

Fig. 19: Entity: Batch aggregating a set of task-units (factspan-sentences) in Figure 13
Fig. 20: Activity: Generated the Batch Entity in Figure 19

Fig. 21: SoftwareComponent: BatchCreator

Fig. 22: Entity: Job
Fig. 23: Activity: Generated the Job Entity in Figure 22
Fig. 24: SoftwareComponent: JobCreator

Fig. 25: Entity: factspan-workerunit, result from an annotation task performed on the task-unit in Figure 16
Fig. 26: Activity: Generated the worker-unit in Figure 25

Fig. 27: SoftwareComponent: AMT

Fig. 28: Agent(crowd): amtworker_001
Fig. 29: Entity: factspan-wav, worker-annotation-vector generated on the worker-unit in Figure 25

Fig. 30: Activity: Generated the worker-annotation-vector in Figure 29

Fig. 31: SoftwareComponent: CrowdTruth FactSpan Adapter
Fig. 32: Agent(software): crowdtruth-system

```json
{
    "_id": "agent/software/crowdtruth/system",
    "platformAgentId": "system",
    "softwareComponent_id": "crowdtruth",
    "type": "software",
    "updated_at": "",
    "created_at": ""
}
```

Fig. 33: Entity: Corrected IBM-medical-media-sentence generated from some particular worker-units after completion of a crowdsourcing annotation task.

```json
{
    "_id": "entity/text/medical/ibm-medical-media-sentence/1",
    "format": "text",
    "domain": "medical",
    "documentType": "ibm-medical-media-sentence",
    "parents": ["entity/text/medical/factspan-workerunit/0"],
    "baseUnit_id": "entity/text/medical/ibm-medical-base-sentence/0",
    "mediaUnit_id": "entity/text/medical/ibm-medical-media-sentence/0",
    "tags": ["media", "unit", "corrected"],
    "tasks": ["FactSpan"],
    "content": "corrected media-unit after completion of a crowdsourcing annotation task."
}
```
Fig. 34: Activity: Generated the corrected media-unit in Figure 33