Proactive Annotation Management in Relational Databases*

Karim Ibrahim, Xiao Du, and Mohamed Y. Eltabakh
Computer Science Department, Worcester Polytechnic Institute
Worcester, MA, US
kaibrahim@cs.wpi.edu, xiaodu@cs.wpi.edu, meltabakh@cs.wpi.edu

ABSTRACT

Annotation management and data curation has been extensively studied in the context of relational databases. However, existing annotation management techniques share a common limitation, which is that they are all *passive* engines, i.e., they only manage the annotations obtained from external sources such as DB admins, domain experts, and curation tools. They neither learn from the available annotations nor exploit the annotations-to-data correlations to further enhance the quality of the annotated database. Delegating such crucial and complex tasks to end-users—especially under large-scale databases and annotation sets—is clearly the wrong choice. In this paper, we propose the *Nebula* system, *an advanced and proactive annotation management engine in relational databases*. Nebula complements the state-of-art techniques in annotation management by learning from the available annotations, analyzing their content and semantics, and understanding their correlations with the data. And then, Nebula proactively discovers and recommends potentially missing annotation-to-data attachments. We propose context-aware ranking and prioritization of the discovered attachments that take into account the relationships among the data tuples and their annotations. We also propose approximation techniques and expert-enabled verification mechanisms that adaptively maintain high-accuracy predictions while minimizing the experts' involvement. Nebula is realized on top of an existing annotation management engine, and experimentally evaluated to illustrate the effectiveness of the proposed techniques, and to demonstrate the potential gain in enhancing the quality of annotated databases.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems—*Relational Databases*

Keywords

Proactive Annotation Management; Keyword Search; Annotated Database.

1. INTRODUCTION

Data curation and annotation is becoming an integral component to modern relational database systems. This is due to several factors including: (1) Many emerging applications that use database systems, especially scientific applications [6, 10, 21, 28], rely on data curation for capturing various types of metadata information, e.g., attaching related articles to the data, highlighting erroneous or conflicting values, and capturing the data's lineage. (2) Annotations do not follow the rigid schema of relational databases, instead a single annotation can be attached to individual table cells, rows, columns, or arbitrary sets and combinations of them. And thus, annotation management gives end-users and higher-level applications the flexibility to annotate their data in a seamless way. And (3) The increasing scale of the generated annotations as well as their richness in semantics have triggered the need for querying the annotations in more systematic and efficient ways, and for incorporating them into the data processing cycle.

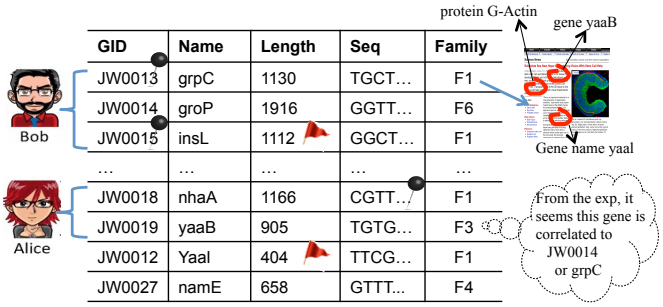
For the aforementioned reasons, annotation management has been extensively studied in the context of relational databases, e.g., [9, 13, 16, 18, 20]. These techniques address several aspects ranging from efficient storage and indexing [18, 20], abstract querying and propagation [9, 16, 20], summarization and summary-aware annotation propagation [22, 32], view-related annotations [13, 25], and semantic annotations [11, 12, 14, 16, 19]. However, the growing scale of the managed datasets and their annotations, and the increasing rate at which both the data and the annotations are dynamically changing, pose new challenges to annotation management that have not been addressed before. One of these challenges—which is the key focus of this paper—is that in many cases the database may become *"under annotated"*. That is, a given annotation may not be attached to all data tuples to which the annotation applies. The following scenarios highlight various reasons that cause a database to be under annotated.

Motivation Scenario 1– Embedded References to DB Objects: *Common uses of annotations include attaching documents or articles to specific data tuples, and linking free-text comments to data values. For example, referring to Figure 1, a biological scientist "Bob" investigating a set of genes may attach a related scientific article to one of his genes-under-investigation* `JW0013`. *Whereas, another scientist "Alice" may attach a comment to one of her genes-of-interest* `JW0019`. *Interestingly, each of these annotations apply to (and are related to) other tuples in the database beyond those to which they are manually attached. For example, the article also references gene names* `yaaB` *and* `yaaI`*, and refers to protein* `G-Actin`. *Similarly, Alice's comment references genes* `JW0014` *and* `grpC`. *Unfortunately, there are many reasons that may hinder Bob and Alice from creating these missing attachments, e.g., Bob and Alice may not know that the other objects actually exist in the database—A scientists usually focuses on a small subset of the data—, Bob may not have time to read the entire article, extract all its references to other objects, and then search the database to add the links, and Alice may not be willing to put any effort in attaching her comment to gene* `JW0014` *since that gene is not of interest to her. As a result, the database becomes under annotated.*

*This project is partially supported by NSF-CRI 1305258 grant.

SIGMOD'15, May 31–June 4, 2015, Melbourne, Victoria, Australia.

http://dx.doi.org/10.1145/2723372.2749435.



GID	Name	Length	Seq	Family
JW0013	grpC	1130	TGCT…	F1
JW0014	groP	1916	GGTT…	F6
JW0015	insL	1112	GGCT…	F1
…	…	…	…	…
JW0018	nhaA	1166	CGTT…	F1
JW0019	yaaB	905	TGTG…	F3
JW0012	Yaal	404	TTCG…	F1
JW0027	namE	658	GTTT...	F4

Figure 1: Example of Under-Annotated DB.

Motivation Scenario 2– Implicit Correlations Between Annotations and Data: *In real-world applications, correlations between the annotations and the data may exist. For example, continuing with our demonstration in Figure 1, some genes in the database may get annotated with "Rounded Flag" and "Triangle Flag" annotations as indicated in the figure. By analyzing the correlations between the annotations and the data, we may discover that most genes having value* `F1` *in the* `Family` *column have an attached "Rounded Flag" annotation. Such correlation suggests that gene* `JW0012` *is probably missing this annotations. Correlations may even exist among the annotations themselves. Unfortunately, manually defining correlation patterns between the annotations and the data, e.g., by DB admins, may not be feasible. This is because these correlations may not be known in advance, hard to capture and express, dynamically changing over time, or even not 100% conformed. As a result, links and associations between the annotations and the data can be easily missed without detection.*

These real-world scenarios are common in many applications, and they yield the database under annotated. As a result, crucial and valuable information can be lost, and the benefits from annotation management may decrease over time. Nevertheless, the state-or-art techniques in annotation management, e.g., [9, 15, 20, 32], are all *passive* systems that only manage the available annotations. The underlying assumption in these techniques is that the completeness in annotating the data is a task that is performed outside the annotation management engine, i.e., delegated to end-users and higher-level applications. Clearly, this assumption is error prone, impractical in some cases, and does not scale as motivated by the above scenarios. Instead, we need an advanced and *proactive* annotation management engine that is capable of learning from the available annotations, analyzing their associations and correlations with the data, and discovering and recommending potentially new attachments between the annotations and the data.

In this paper, we propose the *"Nebula"* system, *a proactive annotation management engine in relational databases*. We will focus only on addressing the challenges of the 1^{st} motivation scenario, i.e., the discovery and management of embedded references. Nebula extends the annotation management engine with advanced capabilities for analyzing and extracting semantics from annotations, identifying potential references and relationships to other database objects, and proactively providing recommendations for establishing these missing attachments. The key challenges to be addressed include:

(1) Identifying and extracting potential keywords from an annotation that possibly reference other objects in the database. This is a challenging task not only because the annotation itself can be a large document or free-text value, but also because the database objects are dynamically changing, too many to easily enumerate, and can be referenced in different ways.

(2) Searching the database for objects matching the identified keywords. DBMSs are optimized for structured query search, e.g., SQL queries. However, searching based on keywords requires advanced techniques and always involves uncertainty in the answer—not to mention that the generated keywords themselves inherit a degree of uncertainty since they are automatically generated.

(3) Ranking and prioritizing the predicted attachments. It is not a straightforward task especially because the ranking should not depend only on the confidence generated from the database search, but it should also capture the relationships between the data tuples and whether or not they have other annotations in common. Therefore, such annotation-related context need to modeled and integrated within the system.

(4) Verifying the predicted attachments. Depending solely on either of the system-generated confidence estimations or the domain experts to verify the attachments may not be a practical solution. We need a hybrid verification approach that offers high confidence in the attachments while minimizing the involvement of domain experts and DB admins.

The overall processing stages and key contributions of Nebula are presented in Appendix A- Figure 16. These contributions are summarized as follows:

- Introducing the new problem of *proactive annotation management*. We propose formal definition and modeling for the concept of *"annotated database"* and the problem of *"discovery and management of embedded references"*. We also propose quantification criteria for *"under-annotated databases"*. (Section 3)
- Developing techniques for extracting the potential keywords (embedded references) from the annotations and translating them to keyword queries to identify the corresponding database objects. Instead of re-inventing the wheel, we leverage the existing work in the area of *keyword search over RDBMSs* with substantial extensions and optimizations to solve the problem at hand. (Sections 4, 5)
- Proposing several refinement and optimization techniques that include: (1) Adjusting the predictions' ranking based on the annotations' context and the relationships among the data tuples, (2) Sharing the search execution among multiple concurrent queries, and (3) Developing approximate searching techniques that search only a very small—but highly promising—subset of the data records instead of searching the entire database. (Section 6)
- Developing assessment criteria and adaptive verification techniques for verifying the predicted attachments. The techniques learn from training datasets and previously verified annotations to balance between achieving a high prediction accuracy and minimizing the involvement of domain experts. (Section 7)
- Implementing the Nebula prototype engine on top of an existing annotation management system [18]. As such, Nebula builds on existing annotation-related functionalities, e.g., seamless storage, organization, and propagation of annotations, and adds the proposed proactive capabilities. The experimental evaluation using real-world workloads demonstrates the effectiveness of the proposed techniques in predicting the missing attachments, e.g., some of the proposed optimizations achieve up to 15x speedup without sacrificing the accuracy. (Section 8)

2. RELATED WORK

Data curation and annotation is an emerging research topic due to the increasing virtue and merit of the collected metadata in modern applications. In the context of relational databases, various techniques have been proposed to address different aspects of annotation management. The techniques in [18, 20] focus on providing efficient storage organization, indexing and compact representation for the annotations. Some techniques, e.g., [9, 16, 20], focus on extending the query algebra and operators to seamlessly propagate the annotations at query time. While other techniques focus on supporting special types of annotations, e.g., provenance annotations [11, 12, 16], hierarchical annotations [15, 23], belief annotations [19], and summary-based annotations [31, 32].

Despite the contributions offered by the above techniques, a common limitation to all of them is that they are all *passive* systems that manage only the available annotations. Detecting whether or not a database is under-annotated and discovering the missing attachments is beyond the scope of these techniques. The techniques proposed in [18, 25] offer mechanisms for automating the attachment of annotations to data tuples by enabling the curator (annotator) to specify *predicates* over the database as part of an annotation's definition. As such, newly added data tuples satisfying these predicates will have the corresponding annotation automatically attached to them. However, these techniques cannot be applied to the problem at hand since they deal only with structured predicates in the form of SQL queries (over the DB schema) without taking the annotations' content into account.

In some domains, e.g., scientific applications, publically available databases may go under a curation process [1, 2, 3, 29]. In this process domain experts manually curate the data, ensure correct and high-quality annotations are attached to the data, and potentially add or remove further attachments between the annotations and the data. Certainly, this curation process is very time consuming, error prone and does not scale well, and more importantly consumes valuable cycles from domain experts and scientists. The proposed Nebula system is not intended to entirely eliminate the curation process. Instead, it is a complementary system by which a significant effort in discovering missing attachments and relationships between the data and the annotations can be automated. And hence, freeing domain experts to do what they know best, which is scientific experimentation.

Another related area of research is the *keyword search in relational database*. It is related to the proposed work because the embedded references are ultimately a set of keywords scattered within the annotations. Existing keyword search techniques, e.g., [5, 7, 27, 30], have made it feasible to execute unstructured queries (a.k.a. keyword queries) over structured databases. Some techniques require a pre-processing phase over the data and building appropriate indexes before answering queries [5, 27, 30]. Other techniques rely on metadata information, e.g., schema information, data types, domain constraints, in generating candidate SQL queries that may answer the keyword query [7]. Nebula will leverage such extensive work and existing techniques without re-inventing the wheel. However, as we will discuss in Section 4, a naive approach that relies solely on keyword search techniques would not be a practical solution and will not lead to the desired results. Therefore, Nebula uses such techniques as one component—which can be black box—within a bigger processing framework.

3. PROBLEM STATEMENT

In this section, we formally define an annotated database, and formulate the problem of proactive annotation management (*Stage 0* in Figure 16).

Definition 3.1 (Annotated Database)**.** *An annotated database $\mathcal{D}$ is modeled as a weighted bipartite graph $\mathcal{D} = \{\mathcal{A}, \mathcal{T}, \mathcal{E}\}$, where set $\mathcal{A}$ consists of nodes representing the available annotations, set $\mathcal{T}$ consists of nodes representing the data tuples in the database, and the edges $\mathcal{E}$ represent the associations and attachments between the annotations and the data tuples in a many-to-many relationship.*

Figure 2 illustrates an example of an annotated database $\mathcal{D}$. Each edge $e(\mathtt{a} \in \mathcal{A}, \mathtt{t} \in \mathcal{T}) \in \mathcal{E}$ is assigned a weight $e.w \in [0, 1]$ representing the confidence in this edge. $\mathcal{E}$ consists of two types of edges: (1) Solid-line edges (called *True Attachments*), and these are the attachments manually established by external sources, e.g., end-users, DB admins, and curators. These edges are assumed to be correct and 100% accurate, i.e., have $e.w = 1$, as illustrated in Figure 2. And (2) Dotted-line edges (called *Predicted Attachments*), and these are the edges that Nebula proactively predicts to

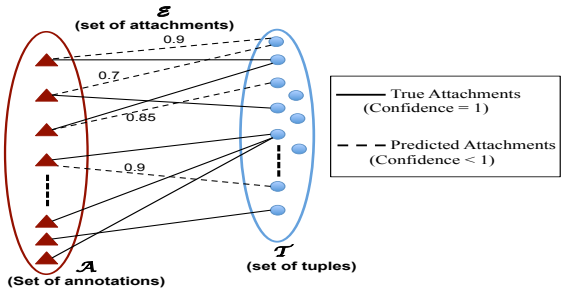


Figure 2: Modeling of an Annotated Database $\mathcal{D}$.

be missing attachments, and assigns an estimated confidence value ($e.w < 1$) to each of them. When a *Predicted Attachment* edge e is verified and accepted by the system (Section 7), e changes its type to a *True Attachment* edge with $e.w = 1$.

In an ideal world represented by an ideal database $\mathcal{D}_{ideal} = \{\mathcal{A}, \mathcal{T}, \mathcal{E}_{ideal}\}$, each annotation $\mathtt{a} \in \mathcal{A}$ is attached to all (and only those) data tuples in $\mathcal{T}$ to which $\mathtt{a}$ is related. And hence, $\mathcal{E}_{ideal}$ contains neither false-negative (denoted by F_N) nor false-positive (denoted by F_P) edges, i.e., $\mathcal{D}_{ideal}.F_N = \mathcal{D}_{ideal}.F_P = 0$. In our context, an annotation is said to be related to a data tuple according to the following definitions.

Definition 3.2 (Embedded Reference)**.** *An embedded reference to a tuple t within an annotation $\mathtt{a}$ is defined as a set of keywords $\{k_1, k_2, ..., k_m\}$ not necessarily consecutive within $\mathtt{a}$'s text that collectively refer to and identify t.*

Definition 3.3 (Annotation-to-Data Relationship)**.** *An annotation $\mathtt{a}$ is said to be related to data tuple t if $\mathtt{a}$ is manually attached to t, or $\mathtt{a}$ contains an embedded reference to t.*

For example, Alice's comment in Figure 1 contains two embedded references, which are "gene...JW0014" and "gene...grpC". And thus, Alice's comment is related to the JW0019 gene tuple (since it is manually attached to it), and to the JW0013 and JW0014 gene tuples (since it contains an embedded reference to each of them).

Unlike the ideal database $\mathcal{D}_{ideal}$, a real-world database $\mathcal{D}$ may diverge from $\mathcal{D}_{ideal}$. The following two evaluation metrics will measure the quality of $\mathcal{D}$ w.r.t $\mathcal{D}_{ideal}$ using the set-difference semantics between the two sets $\mathcal{E}_{ideal}$ and $\mathcal{E}$ as follows:

$$\mathcal{D}.F_N = |\mathcal{E}_{ideal} - \mathcal{E}| \;/\; |\mathcal{E}_{ideal}|, \quad \in [0,\ 1] \tag{1}$$

$$\mathcal{D}.F_P = |\mathcal{E} - \mathcal{E}_{ideal}| \;/\; |\mathcal{E}|, \quad \in [0,\ 1] \tag{2}$$

where $\mathcal{D}.F_N$ captures the ratio of the false-negative edges in $\mathcal{E}$, i.e., the edges in $\mathcal{E}_{ideal}$ that are missing in $\mathcal{E}$. Whereas, $\mathcal{D}.F_P$ captures the ratio of the false-positive edges in $\mathcal{E}$, i.e., the edges in $\mathcal{E}$ that are not present in $\mathcal{E}_{ideal}$. Based on Equations 1 & 2, an annotated database $\mathcal{D}$ without any predicted edges, i.e., without dotted-line edges, is guaranteed to have $\mathcal{D}.F_P = 0$, but $\mathcal{D}.F_N$ might be large. Our goal is thus, to minimize $\mathcal{D}.F_N$ by discovering the missing attachments, while keeping $\mathcal{D}.F_P$ as small as possible.

Definition 3.4 (Discovery of Embedded References)**.** *Given an annotation $\mathtt{a} \in \mathcal{A}$, which is manually attached to data tuples $\{t_1, t_2, ..., t_k\} \subset \mathcal{T}$, our goal is to discover all other data tuples $\{t'_1, t'_2, ..., t'_m\} \subset \mathcal{T}$ (if any) to which $\mathtt{a}$ is related, i.e., $\exists\, e(\mathtt{a}, \mathtt{t'_i}) \in \mathcal{E}_{\mathtt{ideal}}$ & $\mathtt{e(a, t'_i)} \notin \mathcal{E}$, $\forall\, \mathtt{1} \leq \mathtt{i} \leq \mathtt{m}$.* [1]

[1]We assume that *spam-like* annotations, e.g., an annotation that references all (or most) data tuples, do not exist. Detecting and handling this type of annotations is beyond the scope of this paper [26].

Definition 3.5 (Annotation's Focal)**.** *The focal of a given annotation* a $\in \mathcal{A}$ *is denoted as* $Foc(\texttt{a})$*, and defined as* $Foc(\texttt{a}) = \{t_1, t_2, ..., t_k\} \subset \mathcal{T}$*, where* $t_1, t_2, ..., t_k$ *are the data tuples to which* a *is manually attached.*

For example, referring to Bob's annotation in Figure 1, the article's focal is the JW0013 gene tuple. Moreover, according to Definition 3.4, Bob's article may have three potential missing attachments to the JW0019 and JW0012 gene tuples, as well as a missing attachment to the G-Actin protein tuple (if they exist in the database). As will be presented later, the annotation's focal—as part of the annotation's context—will play a key role in the discovery process w.r.t searching and ranking .

4. NAIVE BASELINE APPROACH

A naive approach to solve the problem at hand is to directly apply any of the existing keyword search techniques to discover the data tuples related to a given annotation a. The naive approach involves the following two main steps: (1) Passing annotation a as input to a keyword search technique, i.e., the annotation's content becomes a sequence of keywords to the search technique, and (2) The search technique would search the entire database to identify the data tuples satisfying the input keywords. Without loss of generality, Nebula uses the keyword search technique proposed in [7] as it has shown superiority over other techniques, and it incorporates metadata repositories—which are already part of Nebula's system—into its computations.

As a brief overview, the algorithm in [7] starts by assigning weights to each of the input keywords capturing whether a keyword has a potential mapping to a schema item, e.g., a table name or column name, or a database value. It is possible to have several of these potential mappings for each keyword. From these mappings, the algorithm constructs what are called *configurations*, where each configuration captures one possible semantics of the keyword query. And then, each configuration maps to one or more SQL queries over the database. Each query has an assigned confidence weight that captures the algorithm's estimated confidence of matching this query with the intended semantics. Finally, the produced data tuples from the SQL queries represent the answer to the keyword query, each tuple inherits the confidence of its query.

Limitations of Naive Approach: Clearly, the naive approach has severe limitations and drawbacks that affect not only the performance, but also the accuracy in producing the expected results. These limitations include:

• *Vague and Imprecise Input*: Keyword search techniques expect as input a short sequence of related keywords—usually less than 5 keywords. In contrast, in our case, an input annotation can be a general comment or big article (Refer to Bob's and Alice's annotations in Figure 1). As a result, the keyword search technique will encounter significant degradation in accuracy due to the imprecise input, produce very noisy results that most likely will be worthless, and will take substantial time in processing many input keywords. Therefore, the challenge is to pre-process the input annotation, identify potential keywords that are more likely to be embedded references, and then form meaningful and concise keyword query(s) that existing techniques can handle efficiently.

• *Absence of Annotation-Related Context:* Keyword search techniques are oblivious to the annotation-related context. For example, in keyword search, an input query is not attached to any data tuples. And hence, the query has no starting or focal point that may affect the produced answerset. In contrast, each annotation has a focal, which may affect the desired answerset, e.g., affecting the selection of the candidate attachments as well as their ranking and confidence weights. For example, since Alice's comment is attached to one of the genes, this increases the chance that the highlighted keyword "JW0014" from the comment is also referring to another

***ConceptRefs* Table (Part of *NebulaMeta*)**

Concept	TableName	Referenced BY
Gene	Gene	Gene.ID | Gene.Name
Gene Family	Gene	Gene.Family
Protein	Protein	PID | (PName & PType)
…	….	….

DB concepts that may be referenced within annotations

The expected ways to reference the concept inside annotations

Figure 3: Example of Nebula's Auxiliary Information.

gene— especially if such pair of genes has other annotations in common. Therefore, the challenge is to incorporate and integrate the annotation-related context into the search process to the extent that searching the entire database can be avoided.

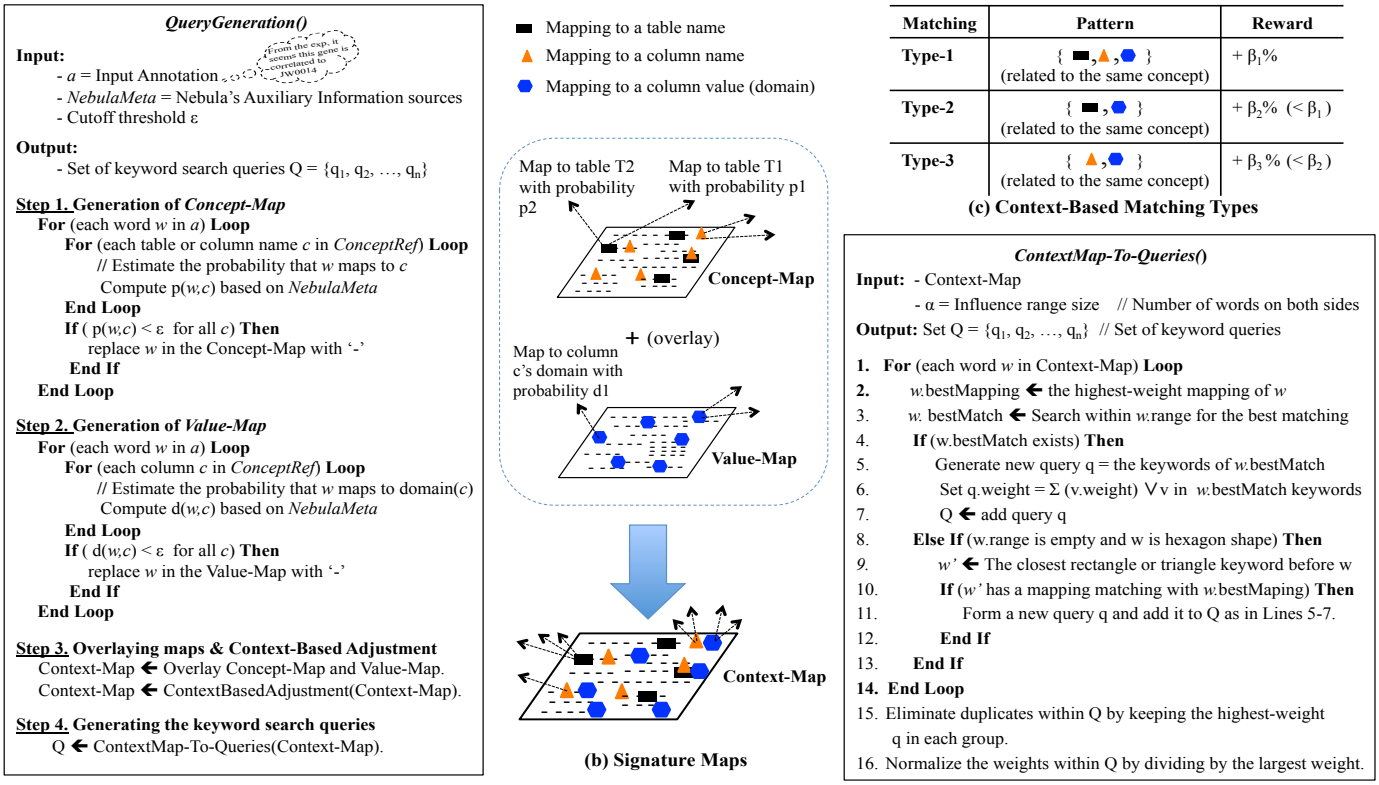
• *Lack of Optimizations and Verifications*: Our problem involves several optimization opportunities and design issues that are beyond the scope of the standard keyword search techniques. For example, existing techniques are designed to answer one query at a time. In contrast, in our problem, a single annotation may generate multiple keyword queries at once. Therefore, handling these multiple queries as a group opens new opportunities not only for optimizing their execution, but also for correlating and ranking their results. Moreover, the predicted attachments between the annotations and the data need to be verified through some mechanisms—depending on the application's sensitivity, the verification may either be fully-automated or require experts' involvement.

5. FROM ANNOTATIONS TO KEYWORD QUERIES

In this section, we present techniques for pre-processing a given annotation, identifying the potential embedded references, and forming keyword search queries from those references (*Stage 1* in Figure 16). To construct meaningful keyword queries, Nebula relies on a variety of auxiliary information sources, e.g, domain knowledge, lexical and semantic database repositories, syntactic patterns and descriptions in the data, etc. The goal is to assign weights to the different words in the annotation capturing the probability that a word would be part of an embedded reference vs. being a regular English word. And then, based on these probabilities, we generate potential keyword queries.

5.1 Nebula's Auxiliary Information

Nebula integrates a variety of sources into its metadata repository, called *"NebulaMeta"*, which includes: (1) Publically available lexical and semantic knowledge databases, e.g., WordNet [4], which maintain synonyms and hyponyms of the English words. (2) Domain and expert knowledge capturing the synonyms and equivalent names of the database tables and columns—especially that the table and column names may have abbreviations without obvious semantics, e.g., a column name "GID" will have an equivalent name like "Gene ID" that is more semantically clear. (3) Domain information for the different columns in the database as well as any available ontologies and vocabularies, e.g., the values within a Gene.Function column may follow a specific ontology. In this case, these ontologies will be stored—along with their links to the database columns—in NebulaMeta. And then, during the search phase, estimating whether or not a given keyword w references a database column, e.g., Gene.Function, will depend on whether or not w is present in the corresponding ontology. (4) Syntactic description and patterns of the column values, e.g., the values in the Gene.ID column in Figure 1 conform with the regular expression of *JW[0...9]*[4], whereas the values in the Gene.Name column usually consist of four letters following the regular expression of *[a...z]*[3]*[A...Z]*. These patterns can be even extracted using automated techniques, e.g., [8]. (5) Samples drawn randomly from specific database columns (See the *ConceptRefs* ta-



(a) Pseudocode generating keyword queries from annotation *a*

(b) Signature Maps

(c) Context-Based Matching Types

(d) From Context-Map to Keyword Queries

Figure 4: Extracting Keyword Search Queries From Annotation a.

ble below) if they have no attached ontology or syntactic patterns. And (6) Domain and expert knowledge capturing the key concepts in the database, and the most probable ways, i.e., columns' combinations, for referring to these concepts within the annotations.

As an example, the *ConceptRefs* system table in Figure 3 is part of *NebulaMeta*. It stores the key concepts in a biological database and the most probable ways for referencing them inside the annotations. For example, among the key concepts are *Gene*, *Protein*, and *Gene Family* (1^{st} column). Concepts do not have to belong to separate database tables, e.g., the genes and gene families are stored in the same *Gene* table (2^{nd} column). The 3^{rd} column captures the most probable ways by which a concept may be referenced within an annotation, e.g., a gene is most probably referenced by either its GID or Name columns, while a protein is more likely to be referenced using either its PID column, or the combination of PName and PType columns. In the context of this paper, we assume that populating the *ConceptRefs* table is performed by the domain experts [2].

5.2 Signature Maps and Queries Generation

5.2.1 Generation of Signature Maps

We outline the key steps in the query-generation algorithm (called *QueryGeneration()*) in Figure 4(a). The algorithm's inputs include: (1) an input annotation a (which is newly inserted into the database), (2) the *NebulaMeta* auxiliary information, and (3) a cutoff threshold ϵ. The algorithm starts by generating two types of maps, called *signature maps*, which emphasize and highlight the important keywords within annotation a.

[2]In extreme cases, a module can be developed for learning from the available annotations the key concepts in the database that they frequently reference, and by which column(s). However, this is beyond the scope of this paper.

Step1 in the algorithm generates the *Concept-Map*. This map highlights the words that are more likely to reference a table name or a column name included in the *ConceptRefs* auxiliary table. The algorithm loops over each word w in the annotation (outer loop), and each table name and column name c included in *ConceptRefs* (inner loop), and estimates a matching weight between them, i.e., $p(w, c)$. To compute $p(w, c)$, we consider whether or not: (1) w is exactly matching c, (2) w is matching equivalent names of c (defined by domain experts), or (3) w is matching synonyms of c. The first two matching types give higher weight than the third type, otherwise the weight is zero. If w does not have a good matching with any of the concepts, i.e., $p(w, c) < \epsilon$, $\forall c$, where ϵ is an input threshold, then w is ignored and replaced with '−' in the *Concept-Map* as depicted in Figure 4(b). For the ease of illustration and further discussion, we use a *rectangle*, and *triangle* shapes to indicate potential mappings from the word to either of a table name, or a column name, respectively. Notice that each of the emphasized words may have multiple potential mappings.

The second generated map is called *Value-Map* (Step 2 in the *QueryGeneration()* algorithm). This map highlights the keywords that are more likely to reference a value within the database—more specifically a value in any of the columns included in the *ConceptRefs* auxiliary table. The algorithm loops over each word w in the annotation (outer loop), and each of the target columns c, and computes a weight indicating the probability that w belongs to the domain of c, i.e., $d(w, c)$. To compute $d(w, c)$, we consider whether or not: (1) w has the same data type as c, (2) w belongs to c's ontology (if any), (3) w follows c's regular expression pattern (if any), and (4) w has good matching with c's drawn sample (if c has no associated ontology or regular expression pattern). The more positive matching of these factors, the higher the weight. Similar to Step 1, if $d(w, c) < \epsilon$, $\forall c$, then w is ignored and replaced with '−' in the *Value-Map* as depicted in Figure 4(b). Each emphasized word in

Value-Map—highlighted as a *hexagon* shape—may have multiple potential mappings to different column domains, each with different $d(w, c)$ probability.

In Step 3, the algorithm overlays the two maps on top of each other such that the emphasized words will be put into the others' context. The generated map is referred to as the *Context-Map* as illustrated in Figure 4(b). The next step is to adjust the assigned weights—either $p(w, c)$ or $d(w, c)$—depending on the surrounding context and how well the mappings will match with the neighbors' mappings. Therefore, *QueryGeneration()* will call Function *ContextBasedAdjustment()* to adjust the probabilities as will be described in the next section.

5.2.2 Context-Based Weight Adjustment

Before describing how the weights are adjusted based on the surrounding context, we will first introduce the matching types used in the adjustment process. We identify three types of matching based on the context, which are summarized in Figure 4(c). *Type-1* matching (the strongest match) means that we are able to form a matching that consists of a table name, column name, and a value within that column. This is indicated in Figure 4(c) by having a set of three shapes {rectangle, triangle, hexagon}. For example, if words *{"gene", "Id", "JW0018"}* are within one context, then they form a *Type-1* matching. As a result, the probabilities of mapping the 1^{st} word to the gene table, the 2^{nd} word to the ID column within that table, and the 3^{rd} word to a value within this column should all be rewarded and increased. The *Type-2* matching is a weaker match but still common. It means that a matching can be formed consisting of a table name and a database value (without a column name), e.g., a comment may include "gene yaaB". Again the mappings that result in such a match will be rewarded, but with a smaller benefit compared to *Type-1* matching. Finally, the Type-3 matching in which the formed matching consists of the column name and a value (without a table name). This is the weakest match and its rewarding is smaller compared to the other types.

Based on these context-based matching types, the *ContextBasedAdjustment()* Function works as follows (A pseudocode of the function is presented in Appendix A- Figure 17). The function loops over each word *w* in *Context-Map* and creates an influence range around *w*, called *w.range*. The influence range *w.range* is α words to the left and to the right of *w*, where α is an input parameter to the function. This range represents the surrounding context of *w* within which the matching patterns are most likely to be found. Therefore, for each potential mapping of *w*, referred to as *w.mapping*, the function will search within *w.range* if any of Type-1 matchings can be formed. If any, then for each match, the weight of *w.mapping* will be increased by β_1 precent. Otherwise, the search continues to the lower-ranked matches, i.e., Type-2 followed by Type-3 matches. The best match of these types will increment the weight of *w.mapping* by β_2, or β_3 percent, respectively, where $\beta_3 < \beta_2 < \beta_1$ as indicated in Figure 4(c).

5.2.3 Generation of Keyword Search Queries

The last step of the *QueryGeneration()* algorithm is to form potential keyword queries from the *Context-Map* (Step 4). The algorithm is outlined in Figure 4(d). The algorithm will first loop over each keyword *w*, and only *w*'s highest-weight mapping will be considered, say *w.bestMapping* (Line 2). Based on this mapping, the algorithm will form the best possible matching within *w*'s influence rage (*w.range*), i.e., forming Type-1 match, and if not then forming Type-2 match, and if not then forming Type-3 match (Line 3). This best match (if any) will form a keyword search query $q = \{k_1, k_2, k_3\}$ (for Type-1 matching), or $q = \{k_1, k_2\}$ (for Type-2 or Type-3 matching). We then set the weight of q to be the sum of weights assigned to its keywords' selected mappings, and then add it to the output set Q (Lines 5-7).

IdentifyRelatedTuples()

Input: Q = { q_1, q_2, …, q_n} // Set of keyword queries,
D ← the annotated database

Output: T = {t_1, t_2, …, t_x} // Set of candidate tuples

Step 1: Executing the keyword queries

1. **For** (each query *q* in Q) **Loop**
2. *q*.answer = KeywordSearch(*q*, *D*) //each tuple *t* has *t*.conf [0, 1]
3. **For** (each tuple *t* in *q*.answer) **Loop**
4. *t*.conf = *t*.conf x *q*.weight //incorporate the query's weight
5. **End Loop**
6. T ← add *q*.answer to T
7. **End Loop**

Step 2: Group tuples and reward the frequent ones

9. Group tuples in T based on tuples content
10. For (each group *g* of tuple *t*) Loop
11. *t*.conf = Σ t_i.conf, forall t_i in *g*

Step 3: Normalize the weights

12. maxC = maximum confidence in T
13. **For** (each *t* in T) **Loop**
14. *t*.conf /= maxC
15. **End Loop**

Figure 5: Identifying DB tuples Related to annotation a.

We consider one important special case while generating the keyword queries, in which the concept keyword (a rectangle or triangle shape), may appear earlier in the text and it may not repeat with each of the following values (a hexagon shape) belonging to this concept. This case is common in human writing as studied in [17, 24]. For example, in Alice's comment the keyword "gene" is not repeated before the keywords "JW0014" or "grpC" since the context is already around genes. To address this special case (Lines 8-12), the algorithm checks if a hexagon-shape keyword *w* has an empty influence range *w.range*. If that is the case, then the algorithm searches backward (starting from *w*'s position) until it finds the closest concept, i.e., a rectangle- or triangle-shape keyword, say *w'*. If *w.bestMapping* and a mapping from *w'* can form a Type-2 or Type-3 match, then a keyword query $q = \{w', w\}$ will be formed. Otherwise, *w* will be ignored. The last steps of the *ConceptMap-To-Queries()* Function (Lines 15-16) are to eliminate any duplicates within Q by keeping only the highest-weight q from each group, and then normalize the weights of the remaining queries to be between [0, 1].

6. EXECUTION, REFINEMENT, AND OPTIMIZED PROCESSING

The next step after generating the set of keyword queries Q is to execute them (*Stage 2* in Figure 16). In the following, we will present the basic version of the execution algorithm, and then propose several refinement and optimization strategies to enhance its performance and accuracy. Due to space limitations, we will omit one of the proposed optimizations, which concerns the *shared query execution* among the keyword queries in Q. In summary, the underlying keyword search technique in [7] will generate for each query $q \in Q$ one or more candidate SQL query(s), each captures a potential semantic of q (Refer to Section 4). Therefore, Nebula exploits possible sharing opportunities among the SQL queries instead of executing them in isolation.

6.1 Basic Execution Algorithm

The *IdentifyRelatedTuples()* algorithm in Figure 5 outlines the key execution steps. In Step 1, each keyword query $q \in Q$ is submitted to the search technique in [7] for execution—As indicated before, any other technique can be used (Line 2). Each tuple t

in the answerset will be assigned a confidence value *t.conf* that is computed by internal criteria specific to the underlying search technique. Nevertheless, Nebula adjusts this confidence according to the query's weight $q.weight$ by multiplying them together (Lines 3-6). And then, the answerset is added to the final output T. In the next step (Step 2), the algorithm rewards the tuples that appear in the answerset of more than one query. The intuition is that if a tuple t satisfies multiple queries generated from the same annotation, then this increases the probability that t is actually related to the annotation. Therefore, the algorithm groups the tuples, and sums the confidences within each group (Lines 9-11). Finally, in Step 3, the confidences are normalized relative to the largest confidence value (Step 3). By the end of this step, T will contain the final output set of the candidate data tuples that are likely to be referenced by annotation `a`. Each tuple t carries a confidence $t.conf \in [0, 1]$ reflecting Nebula's confidence of this attachment.

It is important to highlight that while searching for the relevant tuples given a keyword query, the underlying search algorithm internally leverages the FK-PK relationships among the database tables to produce meaningful related tuples (Line 2). That is why Nebula does not re-examine or adjust the weights based on these relationships. Moreover, Nebula does not explicitly take the structural similarity among the data tuples into account because, by default, all tuples inside a single table will have the same structure. And thus, this property cannot effectively discriminate between tuples in the database. Instead, we incorporate another, more effective, property based on the annotations' focal as described in the next section.

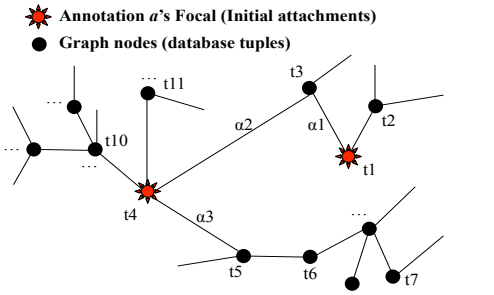
6.2 Focal-Based Confidence Adjustment

An annotation's focal (Def. 3.5) can be leveraged in different ways to enhance Nebula's accuracy in discovering the embedded referenced. We will first introduce a new data structure, called *"Annotations Connectivity Graph"* (a.k.a ACG). ACG is illustrated in Figure 6, where each annotated tuple in the database is represented by a node in the graph. An edge $e(t_i, t_j)$ connects two tuples t_i and t_j iff there are common annotations between the two tuples. Each edge has a weight $e.weight$ (indicated by α_i in Figure 6) representing the ratio between the common annotations to the total number of annotations attached to both t_i and t_j tuples. The ACG structure is incrementally built offline by Nebula as more annotations are attached to the data tuples.

The intuition behind using ACG to adjust the confidence of the candidate data tuples is as follows. Assume annotation `a` has two tuples in its focal, i.e., $Foc(\mathtt{a}) = \{\mathtt{t_1}, \mathtt{t_4}\}$ as illustrated in Figure 6. After analyzing `a` and extracting the potential embedded references, the data tuples produced from the *IdentifyRelatedTuples()* Function are $T = \{t_3, t_5, t_7, t_x\}$, where t_x is not even present in ACG. Certainly, the fact that t_3 is connected to both t_1 and t_4, i.e., already share other annotations with them, increases Nebula's confidence that `a` might also be referencing t_3—especially if α_1 and α_2 are closer to 1. The same applies to t_5 but with slightly less confidence degree since t_5 shares some annotations with only one focal tuple, i.e., t_4. Whereas, for t_7 and t_x, the ACG structure does not suggest an increase in their confidence.

To reflect the ACG structure on the candidate tuples' confidence, we extend Step 2 in *IdentifyRelatedTuples()* Function (Refer to Figure 5) by adding the following lines of code after the grouping step, i.e., after the loop in Lines 10-11.

```
For (each t ∈ T) Loop
    For (each e(t, f) ∈ ACG, ∀ f ∈ Foc(a)) Loop
        t.conf += e.weight x t.conf
    End For
End For
```



Each edge $e(t_i, t_j)$ indicates that t_i & t_j share common annotations. The e.weight (α) is the ratio between the common annotations to the total annotations attached to t_i & t_j.

Figure 6: Annotations Connectivity Graph (ACG).

That is, for each direct edge each $e(t, f)$ between tuple t and one of the annotation's focal f, t's confidence will be increased according to the e's weight. The higher the weight and the larger the number of direct edges to focal tuples, the higher the reward that tuple t receives. The adjustment model can be easily extended to take into account the shortest path—in terms of the number of hops—between t and each focal tuple instead of only the direct edges. In this case, tuple t_7, for example, will get some reward based on its 4-hop distance from t_4, e.g., by multiplying the weights of the in-between edges. However, Nebula currently incorporates only the direct edges to adjust the weights since the latter extension is semantically weaker and may cause model overfitting.

6.3 Approximate Searching with Focal-Based Spreading

The key intuition here is that as the ACG structure gets mature over time, i.e., models more and more annotations, the graph gets relatively stable and not many new edges are added. In this case, there is a good chance that the embedded references in a given annotation will be referring to direct (or close) neighbors of the annotation's focal. And thus, instead of searching the entire database—which is a very expensive operation—, we can focus the search only on the neighbors of the annotation's focal. Towards this goal, we first introduce the notion of ACG stability as follows.

Definition 6.1 (ACG Stability)**.** *The ACG structure is said to be stable iff for the most recent batch of annotations of size B with total number of attachments to database tuples M ($M \geq B$) , the number of newly added edges to ACG is N, where $N/M < \mu$.*

where the batch size B, and the stability threshold $\mu < 1$ are configuration parameters. The ACG's stability property—which is a Boolean value— changes from one batch to another. For ease of computations, the system considers non-overlapping batches (not sliding batches). That is, when the current batch collects B annotations, the ACG's stability property will be re-computed based on that batch. And then, the counters are reset for the next batch. When the ACG structure is marked as stable, then Nebula follows the intuition described above, and searches only neighbors of the annotation's focal.

We investigated several variants of implementing this intuition inside Nebula. One variant, called *Fixed-Scope*, is to search a fixed *K-hop* neighbors of the annotation's focal. This is performed by extending Step 1 in Figure 5 and replacing Lines 1- 2 by the following lines:

```
- miniDB = Form a materialized view of the K-hop
      neighbors of tuple f in ACG, ∀f ∈ Foc(a)
- For (each query q ∈ Q) Loop
    - q.answer = KeywordSearch(q, miniDB)
    ...
```

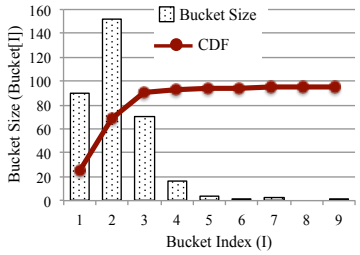


Figure 7: Metadata Profile for Dynamic Selection of K.

That is, the system will perform the keyword search operation over a mini database *miniDB* consisting of only the *K-hop* neighbors of the annotation's focal, where K is a parameter that specifies the radius around the focal tuples. For example, referring to Figure 6, the 1-hop neighbors of annotation a consists of $\{t_2, t_3, t_5, t_{10}, t_{11}\}$. Each tuple in *miniDB* will follow the schema of its own table, and thus creating a materialized mini version of the original database.

A key question in the focal-based spreading search is: *How to specify the value of K?* Blindly setting K to a large value may cause unnecessarily processing overhead, while setting it to a small value may cause the accuracy to be unacceptably low. To help guiding this decision, Nebula creates and maintains a profile on how the ACG structure changes over time. The profile is a simple histogram structure consisting of an array of integers, called *Bucket*, and is built as follows. Assume an annotation `a` with focal tuples $Foc(\texttt{a})$, for which Nebula predicts a set of True Attachment tuples $T = \{t_1, t_2, ..., t_x\}$. This means that `a` will be attached to each of these tuples, and hence an edge will be added to ACG between each t_i and each of the focal tuples $Foc(\texttt{a})$ (unless the edge already exists). Before adding these edges, Nebula will perform the following step.

For `(each t ∈ T)` **Loop**
- `S = shortest path from t to any of Foc(a)`
  `assuming un-weighted ACG.`
- `//Update the profile`
- `profile.Bucket[S.length] += 1;`

End For

While computing S, we assume un-weighted ACG since we care about the smallest number of hops from t to reach any of the annotation's focal. And then, we update the profile by incrementing the bucket corresponding to $S.length$, i.e., Bucket[S.length] is incremented by 1. The intuition is that, if we were to discover t by searching only the neighbors of the annotation's focal, then K would need to be at least equal to $S.length$ (or larger), otherwise t would have been missed. By accumulating more points in the profile over time, the distribution of points across the different buckets would give us a good guidance on where to set K (See Figure 7). For example, according to the example profile illustrated in Figure 7, by setting by setting $K = 2$, or $K = 3$, we expect to discover 71%, or 93% of the candidates, respectively. The reason is that based on the history, 71%, and 93% of the candidates produced from the entire database search were 2-hop, and 3-hop away from at least one of the focal tuples, respectively. The profile can be then used either by the DB admins to manually select K, or by the system to automatically select K given a desired accuracy.

7. VERIFICATION AND ASSESSMENT

So far, we presented various techniques and optimizations for discovering and predicting the missing attachments of a given annotation `a` to a set of data tuples $T = \{t_1, t_2, ..., t_x\}$. In this section, we address how these predicted attachments are verified and how the different techniques are assessed (*Stage 3* in Figure 16).

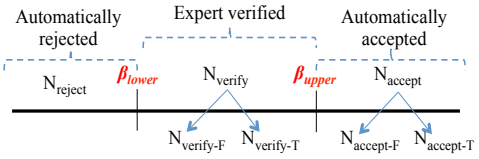


Figure 8: Verification Bounds and Assessment Variables.

Definition 7.1 (Verification Task)**.** *A verification task v is defined as v = (v_{id}, `a`, t, confidence, evidence), where v_{id} is a unique system-generated identifier,* `a` *is an annotation, t is a database tuple to which Nebula predicted a missing attachment, confidence is the estimated confidence of the attachment, and evidence is the set of evidences supporting Nebula's prediction. The result of v is a Boolean decision of either accepting the attachment (becomes a True Attachment), or rejecting and discarding the attachment.*

In the current implementation, *v.evidence* contains all the keyword queries q generated from `a` for which t is part of the answer. The objective from reporting *v.evidence* is to help the DB admins in the verification process. Since manual verification of all verification tasks—which can be many—is not a practical solution, Nebula relies on two thresholds β_{lower} and β_{upper} as highlighted in Figure 8, where a verification task v having *v.confidence* $< \beta_{lower}$ will be automatically rejected (discard from the system). Whereas, if *v.confidence* $> \beta_{upper}$, then v will be automatically accepted (becomes a True Attachment with 100% confidence). In contrast, if $\beta_{lower} \leq$ *v.confidence* $\leq \beta_{upper}$, then v would require expert involvement for verification. In this case, v is called a *"pending verification task"*, and stored in a system table for further processing. This table can be queried by database admins (or authorized users) to report the pending tasks and verify them. To verify a task, the following extended SQL command is introduced:

`[Verify | Reject] Attachement <`v_{id}`>;`

The new command is needed because if the attachment is verified and accepted, then a sequence of actions will be performed by the system transparently from end-users. These actions include: (1) Attaching annotation `a` to tuple t (it becomes a True Attachment), (2) Updating the ACG structure to reflect the new attachment (Refer to Figure 6), and (3) Updating the metadata profile that guides the focal-based spreading technique (Refer to Figure 7). Notice that the same actions will be performed for each automatically accepted task whose *v.confidence* $> \beta_{upper}$.

Assessment: The verification decisions will be assessed according to four criteria. We assume that the number of attachments between `a` and the data tuples in the ideal world D_{ideal} is denoted by N_{ideal}. The predictions of the system can be categorized into five categories according to the β_{lower} and β_{upper} bounds as depicted in Figure 8. The number of automatically rejected predictions is denoted by N_{reject}. The number of manually-verified predictions is denoted by N_{verify}, which is further divided into $N_{verify-T}$ (if the verification is positive), or $N_{verify-F}$ (if the verification is negative), such that $N_{verify} = N_{verify-T} + N_{verify-F}$. Finally, the number of automatically accepted predictions is denoted by N_{accept}, which is further divided into $N_{accept-T}$ (if the prediction is correct), or $N_{accept-F}$ (if the prediction is wrong), such that $N_{accept} = N_{accept-T} + N_{accept-F}$. Based on these variables, the assessment criteria is defined as follows.

Definition 7.2 (Assessment Criteria)**.** *For a given annotation* `a`*, the prediction of* `a`*'s missing attachments is assessed based on four criteria: (1) The false-negative ratio F_N, (2) The false-positive ratio F_P, (3) The manual effort involved in the verification process M_F, and (4) The manual hit ratio (conversion ratio) M_H. The criteria factors are defined as:*

BoundsSetting()

Input:
- Training dataset $D_{Training}$, Tolerance threshold Ψ (e.g., 0.1)

Output: Setting values for β_{lower} and β_{upper}

Step 1: Distort the training dataset and generate $D_{incomplete}$

1. **For** (each annotation *a*) **Loop**
2. - Remove all of *a*'s attachments to the data tuples
3. which are referenced within *a*, except one.
4. **End Loop**

Step 2: Compute the assessment criteria

5. **For** (each annotation *a* in $D_{incomplete}$) **Loop**
6. - Let Nebula predicts the missing attachments
7. **For** (each possible setting of β_{lower} and β_{upper}) **Loop**
8. - Compute F_N, F_P, M_F, and M_H (w.r.t $D_{Training}$)
9. **End For**
10. **End For**

Step 3: Select the best β_{lower} & β_{upper}

11. **For** (each possible setting of β_{lower} and β_{upper}) **Loop**
12. - Compute average over all annotations: G_{FN}, G_{FP}, G_{MF}, G_{MH}
13. **End For**
14. Select the settings of β_{lower} and β_{upper} that minimizes G_{MF}
15. while keeping $G_{FN} < \Psi$ and $G_{FP} < \Psi$.

Figure 9: Adaptively Adjusting β_{lower} & β_{upper} Bounds.

$$F_N = \frac{N_{ideal} - (N_{verify-T} + N_{accept-T} + N_{focal})}{N_{ideal}}$$

$$F_P = \frac{N_{accept-F}}{(N_{verify-T} + N_{accept} + N_{focal})}$$

$$M_F = N_{verify}$$

$$M_H = N_{verify-T} \ / \ N_{verify}$$

The F_N and F_P factors capture the accuracy and completeness of the prediction, whereas M_F and M_H capture the required effort by domain experts in the verification process. Notice that the only source producing false positives, i.e., $F_P > 0$, is the automatically accepted predictions that are actually wrong $N_{accept-F}$.

Adaptively Adjusting the β_{lower} and β_{upper} Bounds: The specification of the two bounds β_{lower} and β_{upper} is clearly an important factor since they affect the accuracy of the accepted/rejected predictions as well as the balance between automating the verification decision vs. involving the domain experts in the process. For example, if $\beta_{upper} = 1$, then no predictions will be automatically accepted and they must be manually verified. In contrast, if $\beta_{lower} = \beta_{upper}$, then all the predictions will be decided automatically without any expert involvement. The optimal scenario is to minimize F_N and F_P with zero involvement from domain experts, i.e., $M_F = 0$. This is due to the potential long delays in human actions, the inability to scale to many verification tasks, and more importantly their valuable time that should not be consumed in the verification process. However, entirely eliminating the experts from the process may not be feasible due to the uncertainty involved in deciding whether or not an annotation is related to a data tuple. Therefore, β_{lower} and β_{upper} should be adjusted to generate acceptable F_N and F_P while minimizing M_F.

To help adjusting β_{lower} and β_{upper} to proper values, we periodically deploy the algorithm presented in Figure 9. Nebula uses a training dataset $D_{Training}$ in which each annotation `a` is attached to all (and only those) tuples related to `a`. $D_{Training}$ is built by selecting a subset of the annotations from the underlying database, e.g., selecting few 100s of the available annotations, and then manually verifying their attachments and adding any missing ones. And then, the algorithm distorts the dataset by considering each annotation `a` and removing all its attachments to the data tuples except one. The resulted distorted dataset is called $D_{incomplete}$ (Step 1 in Figure 9). The system will then try to discover the missing attachments (Line 6), and then assess such prediction for different settings of β_{lower} and β_{upper} (Lines 7-9). Notice that all the variables defined in Figure 8 can be automatically computed (including $N_{verify-F}$ and $N_{verify-T}$) since $D_{Training}$ dataset is available. For example, $N_{verify-F}$ will correspond to the verification tasks located between β_{lower} and β_{upper} that do not have a matching in $D_{Training}$. In Step 3, we average the assessment measures over all the annotations for each setting for β_{lower} and β_{upper}, and then take the best setting that minimizes the expert involvement (M_F) while keeping F_N and F_P within an acceptable range.

Further enhancements can be applied to the *BoundsSetting()* algorithm—which are omitted from Figure 9 for simplicity. These enhancements include: (1) In Step 1, we may consider different degrees of dataset distortion, e.g., we may remove all attachments of a given annotation except 2 or 3 links, which creates less-distorted versions of the dataset. And (2) In Step 3, Lines 7-9, instead of blindly exploring different values of β_{lower} and β_{upper}, we can use M_H factor to guide the adjustment of the bounds. For example, if M_H is high (closer to 1), then this means that most manually-verified predictions are accepted. In this case, the algorithm may lower β_{upper} (moving it to the left) to allow accepting more predictions automatically without passing them for verification.

It is worth mentioning that in practice Nebula does not require the D_{ideal} dataset to be present—which is the typical cases in real-world applications. In this case, the assessment criteria can be computed periodically by the domain experts as follows: (1) Given a set of m newly added annotations, the domain experts will examine them and identify the True attachments, i.e., N_{ideal}, (2) Nebula will provide its predictions for each annotation in m divided into the three categories depicted in Figure 8, i.e., *Reject*, *Verify*, and *Accept*, and (3) The domain experts will compute the assessment criteria factors for each annotation, and can even compute additional statistics, e.g., min, max, and average, across the m annotations.

8. EXPERIMENTS

Nebula is implemented on top of an existing annotation management system [18], which offers end-to-end annotation management functionalities, e.g., mechanisms for adding annotations, transparent storage and indexing, and automated propagation of annotations along with the queries' answers at query time. The experiments are conducted using an AMD Opteron Quadputer compute server with two 16-core AMD CPUs, 128GB memory, and 2 TBs SATA hard drive. Figure 10 summarizes the key experimental parameters, which are described in the following section.

8.1 Experimental Setup and Workloads

Curated Biological Datasets: We use a subset of the UniProt real-world annotated biological database [3]. UniProt offers a comprehensive repository for protein and functional information for various species. We extracted three main tables including *Protein*, *Gene*, and *Publication*. The tables are connected through the following relationships: The *Protein* table has a many-to-one relationship with *Gene*, and many-to-many relationships with *Publication*. The *Gene* table has also a many-to-many relationship with *Publication*. The dataset consists of approximately 750,000 protein records ($\approx$ 4.7GBs), 1.3 x 10^6 gene records ($\approx$ 8GBs), and 12 x 10^6 publication records ($\approx$ 4.5GBs). Thus, the total size of the dataset is approximately 18GBs.

We divide the dataset to form three experimental subsets with different sizes, which are: (1) *Small-Size Dataset* (D_{small}) in which we select 10% of the proteins, genes, and their related publications ($\approx$ 2GBs), (2) *Mid-Size Dataset* (D_{mid}) in which we select 50% of the proteins, genes, and their related publications ($\approx$ 9GBs), and (3) *Large-Size Dataset* (D_{large}) in which we use the entire extracted records ($\approx$ 18GBs). The *NebulaMeta* repository

Symbol	Description
D_{small}, D_{mid}, D_{large}	The databases with different sizes of 2.5GB, 10GB, and 20GB, respectively.
$L_{i\text{-}j}$	A set of publications (annotations) with number of links between *i* and *j*. Takes values: $L_{1\text{-}3}$, $L_{4\text{-}6}$, $L_{7\text{-}10}$
L^m	A set of 15 annotations, each of a max size *m* bytes, where *m* in {50, 100, 500, 1000}. L^m contains 5 annotations from each of $L_{1\text{-}3}$, $L_{4\text{-}6}$, $L_{7\text{-}10}$.
ε	The cutoff threshold for signature-map generation. Takes values 0.4, 0.6, or 0.8.
Δ	Distortion degree. Takes values 1, 2, or 3. Δ = *x* means dropping all links of an annotation except *x* links.
K	# of hops in focal-based spreading search

Figure 10: Summary of Workload Parameters.

is manually populated by adding the two concepts of "Gene" and "Protein" to the *ConceptRefs* table, and identifying their `ID`, and `Name` columns to be the referencing columns. We also specified regular expression patterns over the values in `Gene.ID` and `Gene.Name` columns.

Annotation Workload: We use the publication records in the dataset to represent the annotations over the gene and protein records. To create the workload for each of the three datasets, i.e., $D = \{D_{small}, D_{mid}, or D_{large}\}$, the following procedure is followed (A visual representation of the datasets and the workload mixture is presented in Appendix A- Figure 18):

(1) The annotated dataset D is assumed to be an ideal and complete dataset, i.e., when verifying Nebula's recommendations, D will be treated as the reference D_{ideal} dataset.

(2) A workload over D is created by selecting a mixture of annotations, which will act as the new annotations to be inserted into the database. The workload consists of four distinct sets, denoted as $L^{50}, L^{100}, L^{500}, L^{1000}$, where each L^m consists of a total of 15 annotations, each of a max size of m bytes.

(3) The 15 annotations within each L^m are selected such that they are divided equally across three distinct subsets (5 from each subset), which are denoted as L_{1-3}, L_{4-6}, and L_{7-10}. Set L_{i-j} contains annotations having a number between i and j (inclusive) of embedded references to the `Gene` and/or `Protein` tables (See Figure 18). For example, "$L^{100}.L_{4-6}$" is a set in the workload that contains 5 annotations, each of a max size of 100 bytes, and each contains between 4 to 6 embedded references to the gene and/or protein records. This step involves a manual effort to identify the publications that have the desired number of embedded references under each group [3]. The workload is designed in such way to ensure the inclusion of annotations with various sizes as well as diverse number of embedded references.

(4) The ACG structure for dataset D is built from the FK-PK relationships between the publication records and the gene and protein records, i.e., two tuples—from any of the `Protein` and `Gene` tables—having common a publication will have an edge between them in ACG. The only exception is that the annotations used in the workload, i.e., part of the L^m sets, do not participate in building the ACG. This is because they will be treated as new annotations. The ACG is built at once and not in an incremental fashion.

As summarized in Figure 10, the other important parameters include the cutoff threshold ϵ, which is used in generating the signature maps, the distortion degree Δ, where an annotation `a` with $\Delta = x$ means that all of `a`'s links to the gene and protein records will be dropped except x links, and parameter K, which represents the number of considered hops in the focal-based spreading search.

[3]Set "$L^{50}.L_{7-10}$" was always empty since not as many embedded references can fit within a 50-char annotation. Therefore, we substituted the missing annotations by additional ones added to L_{1-3} and L_{4-6} subsets.

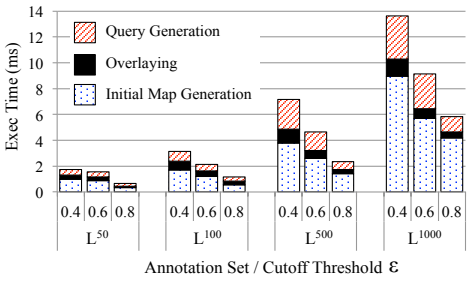
8.2 Performance Evaluation

Generation of Keyword Queries: In Figure 11, we focus on studying the performance of generating the keyword queries from a given annotation. These experiments are independent of the database size since they focus only on analyzing the annotation's content. Therefore, we use the workload created from the largest dataset D_{large}. Each experiment is repeated 15 times, one for each annotation in L^m, and the average values are presented in the figures. In the *Naive* approach, the entire annotation is assumed to be a single query, and thus we assume the needed time to generate the query in this case is zero. In Nebula , the generation goes through three phases, which are: (1) The generation of the *Concept* and *Value* signature maps, (2) The overlaying and context-based weight adjustment, and (3) The generation of the keyword queries. The execution time taken by each of these phases is illustrated in Figure 11(a). We study the performance under various cutoff threshold (ϵ) and annotation sets (L^m), which are presented on the x-axis. As illustrated in Figure 11(a), the first phase (map generation) takes around 2/3 of the execution time, and then the other two phases consume around 1/3 of the time. As the cutoff threshold gets larger, less number of keywords qualify to be added to the signature maps, and thus less work is needed to align them and generate candidate keyword queries.

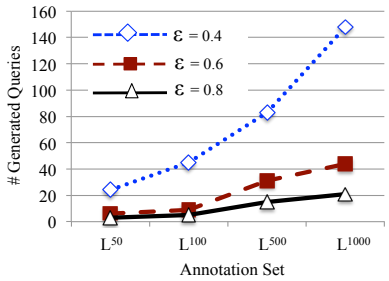
In Figure 11(b), we present the number of generated keyword queries from the previous experiment under the different L^m and ϵ values. The cutoff threshold $\epsilon = 0.4$ is clearly very low and causes the number of generated queries to be relatively high—Recall that each annotation has less than 10 embedded references, and thus in the ideal case, we should expect around 10 keyword queries. This large number of generated queries is due to the fact that many keywords from the annotation will pass the 0.4 threshold. In contrast, for $\epsilon = 0.6$ & 0.8 the thresholds are tighter and less number of queries are generated. Although this is a good sign, this figure does not capture whether or not the queries represent the actual embedded references within the annotation. Therefore, we manually investigate the generated SQL queries and check whether a query refers to something other than an embedded reference (False Positive), or some embedded references are not captured by any query (False Negatives). We report the results in Figure 11(c).

The presented numbers in Figure 11(c) indicate that for $\epsilon = 0.4$ no embedded references were missed, i.e., the false negative percent is zero. However, very large percentage of the generated queries search keywords that are not embedded references, e.g., 91% of the generated queries in the case of L^{1000} are false positives. The 0.6 cutoff is superior over the 0.4 value since the 0.6 threshold also has zero false negatives but significantly less false positives (as well as less number of total queries as illustrated in Figure 11(b)). The tightest threshold 0.8 misses few embedded references, yet its false positive results are very low compared to the other values, e.g., it has zero false positives under L^{50}. Even under larger annotation sizes, e.g., L^{1000}, where the false positive percentage reaches 54% for $\epsilon = 0.8$, the effect is not as severe as in the cases of $\epsilon = 0.4$ *or* 0.6 because: (1) The number of queries for the 0.8 threshold is relatively very small (Refer to Figure 11(b)), and (2) As the number of false-positive queries becomes smaller, less noise propagates to the subsequent phases, i.e., the tuple-level weight adjustment (Section 6.1), and the focal-based confidence adjustment (Section 6.2). And thus, these phases become more effective and produce better-quality results.

In general, there is no golden value for ϵ that works the best in all cases. However, as our experiments show, and has been also reported in literature in related keyword search problems [7], values between 0.5 and 0.8 usually yield good results. In the rest of the experiments, we will exclude the 0.4 threshold from the comparison as it has no advantage over the 0.6 threshold.



(a) Overhead of Query Generation Phases

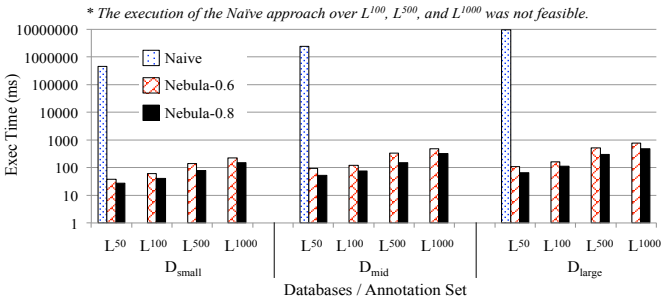


(b) Number of Generated Keyword Queries

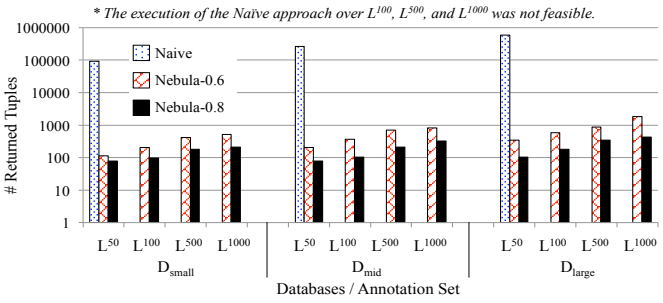
Annotation Set	False Negative %			False Positive %		
	Cutoff Threshold ε			Cutoff Threshold ε		
	0.4	0.6	0.8	0.4	0.6	0.8
L^{50}	0	0	33.3	83	50	0
L^{100}	0	0	0	84	66	40
L^{500}	0	0	12	90	72	53
L^{1000}	0	0	10	91.5	77	54

(c) Query-Level False Positives and Negatives

Figure 11: Performance of Keyword Query Generation Phases.



(a) Execution Time Overhead



(b) Number of Returned Tuples

Figure 12: Performance of Candidate Tuple Generation.

Execution of Keyword Queries: After generating the keyword queries, we study their execution performance in Figure 12. The x-axis presents the different database sizes as well as the various annotation sets L^m under each database. The comparison includes the *Naive* approach in which the keyword query includes the entire content of the annotation. In Figure 12(a), we measure the total execution time, i.e., the time of one query in the case of the *Naive* approach, and the sum of the execution times from the n queries generated from Nebula. As indicated in the figure, the *Naive* approach is five orders-of-magnitudes slower than the other two cases; *Nebula*-0.6 (using $\epsilon = 0.6$), and *Nebula*-0.8 (using $\epsilon = 0.8$). Moreover, under larger annotation sets, e.g., L^{100} or larger, it was not even feasible to execute the *Naive* approach.

The performance between *Nebula*-0.6 and *Nebula*-0.8 is relatively comparable. The differences in execution time are mainly due to the number of queries being executed as well as the number of produced tuples, which is depicted in Figure 12(b). The number of produced tuples from the queries again confirm that the *Naive* approach is useless as it returns 100s of thousands of tuples under the smallest annotation set L^{50}. This is due to the keyword query itself being very imprecise, and hence the underlying search technique returns a significant portion of the database. It is important to highlight that for *Nebula*-0.6 and *Nebula*-0.8 the number of returned tuples does not grow in the same ratio as the database size grows. This is because many of the queries are on columns with unique values, and thus the increase in the database size does not necessarily reflect on the number of returned tuples.

Referring back to Figure 12(a), the reported execution time of Nebula's techniques represent the sum of times from executing each keyword query in isolation, i.e., no multi-query sharing. This is the default mode, especially if the underlying keyword search technique is used as black box. Nevertheless, as we highlighted in Section 6, if we enable shared execution among the SQL queries generated from the keyword search technique, then significant savings can be achieved. For completeness, we report in Figure 13 the performance gain from such query sharing. We repeat the same experiment as in Figure 12(a) while comparing *Nebula*-0.6 and *Nebula*-0.8 with their sharing-enabled variants. The results indicate around 40% to 50% speedup in execution time while producing the same number of output tuples.

Finally, the performance of executing the keyword queries using the approximate focal-based spreading technique is presented in Figure 14. In this experiment, we set the database to the largest one D_{large}, the cutoff threshold to $\epsilon = 0.6$ (as it has zero false negatives), and the annotation set to L^{100} (an average-size set). Moreover, there is no sharing while executing the queries. This experiment is sensitive to the number of initial attachments between an annotation and the database records (the focal points). Therefore, we vary the distortion degree Δ over the x-axis, which controls the number of focal points that an annotation will have, and measure the performance under various values of K, i.e., the number of hops around each focal point. The results in Figure 14(a) illustrate that the execution time is around 15x faster than the basic search without sharing. Moreover, when compared with multi-query sharing, it is around 8x faster. This is mostly due to searching a small—but highly-promising—subset of the database.

Clearly, as the two key parameters Δ and K increase, the number of the searched tuples will increase, which will reflect on increasing both the execution time (Figure 14(a)), and the number of produced tuples (Figure 14(b)). By comparing Figures 12(b) and 14(b), we observe that the number of produced tuples is significantly less in the latter case. Therefore, the advantages of the focal-based spreading search are not limited only to speeding up the execution time, but also to reducing the verification overhead, which will be studied next. We repeated the experiments in Figure 14 using different dataset sizes and cutoff ϵ thresholds, and the insights from the results are almost the same, i.e., significant gain in execution time and an order-of-magnitude reduction in the candidate tuples.

Verification and Assessment: Since the ideal database D_{ideal} is known in our experiments, i.e., for a given annotation a we know all of its attachments to the database tuples, then the assessment criteria introduced in Definition 7.2 can be easily computed. Even the *expert-verified* factors $N_{verify-F}$ and $N_{verify-T}$ can be automatically computed, i.e., $N_{verify-T}$ represents all the candidate

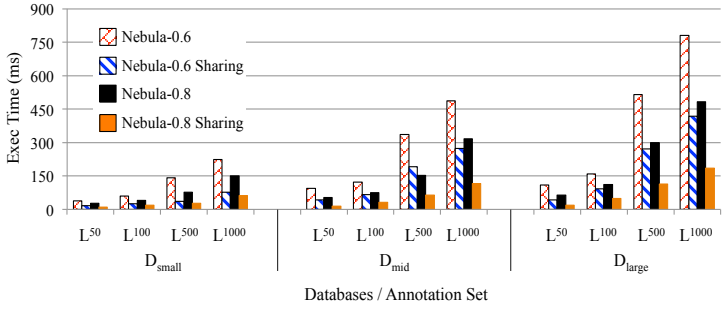


Figure 13: Shared Query Execution.

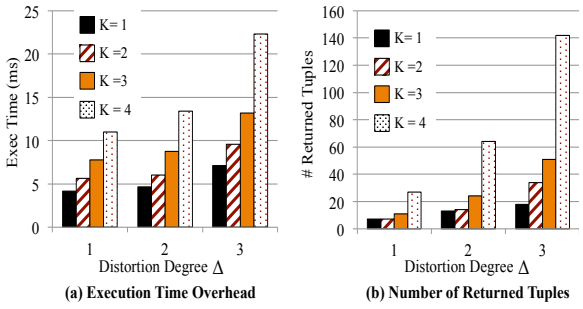


Figure 14: Focal-Spreading Approximate Search.

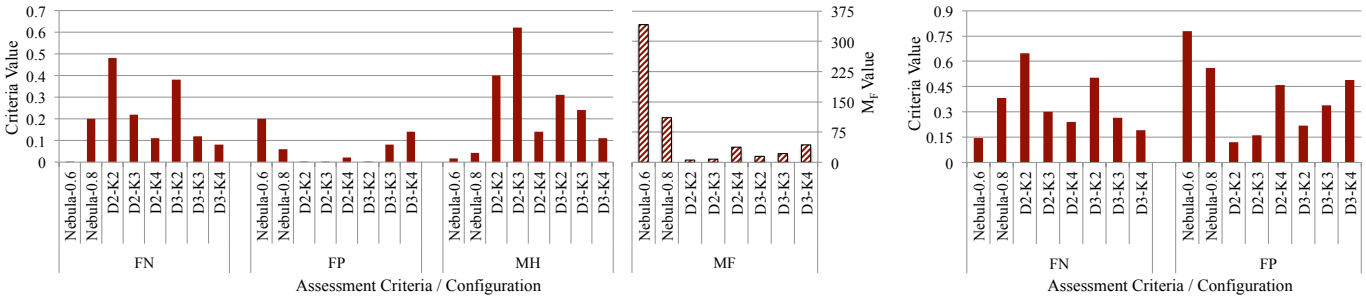


Figure 15: Evaluation of the Verification and Assessment Criteria.

tuples that lie between β_{lower} and β_{upper} and their links to a exist in D_{ideal}, while $N_{verify-F}$ represents those tuples that lie between β_{lower} and β_{upper} and their links to a do not exist in D_{ideal}. This is under the assumption that experts do not make errors. The evaluation is repeated 15 times, once for each annotation in L^{100}, and the average values are presented in the figures.

In Figure 15, we fix the annotation set to L^{100}, and the database to the largest one D_{large}. We then evaluate each of the four assessment criterion under eight different configurations (the x-axis), which include the basic algorithm without approximation under two cutoff thresholds *Nebula*-0.6 and *Nebula*-0.8, and six configurations for the focal-based approximate searching with various Δ and K parameters. In Figure 15(a), the verification bounds β_{lower} and β_{upper} are automatically adjusted using the algorithm presented in Section 7. The $D_{Training}$ dataset is built by randomly selecting 500 annotations from the database, and manually verifying their attachments. The algorithm has set the bounds to $\beta_{lower} = 0.32$ and $\beta_{upper} = 0.86$. The results show that there is no single configuration that dominates the others in all criteria factors. Nevertheless, in general, *Nebula*-0.8 is performing better than *Nebula*-0.6, especially when considering the manual effort needed from the experts' side (the M_F factor). Yet, *Nebula*-0.8 introduces around 20% false negatives. The second observation is that the focal-based approximate techniques are performing very well, especially under $K = 3$ or $K = 4$.

In the above experiment, we exclude the evaluation of the Naive approach w.r.t. the assessment criteria since it does not scale to annotation sizes beyond L^{50} (Refer to Figure 12). However, to confirm the infeasibility of this approach, we computed the assessment factors $\{F_N, F_P, M_F, M_H\}$ for the L^{50} case in which 587,040 tuples have been returned (Figure 12(b)), and the results are $\{0, 0.93, 318427, 1.6e-5\}$, respectively. This means that for a single annotation, the domain experts will need to verify 318,427 annotations from which only 5 will be accepted. These numbers provide a clear evidence that *Nebula* enables a new functionality in annotated databases that is not possible otherwise.

In Figure 15(b), we evaluate an extreme case in which we manually set the verification bounds $\beta_{lower} = \beta_{upper} = 0.5$. In this case, there will be no expert involvement in the verification process. The results show that the F_P values get significantly higher, and hence the techniques create many wrong associations between the annotation and the data tuples. Moreover, the F_N values also get higher by a noticeable percentage. We repeated the same experiment by setting β_{lower} and β_{upper} to the same value (but different from 0.5), and in all cases the F_N and F_P were relatively very high compared to the results in Figure 15(a). Our conclusion is that it may not be feasible to entirely exclude the experts from the verification process while achieving good results. However, the results in Figure 15(a) are promising and show that with a reasonable involvement, we can achieve acceptable results.

9. CONCLUSION

In this paper, we presented the un-addressed problem of *proactive annotation management in relational databases*. We focused on one sub-problem, which is the discovery and management of embedded references within the annotations. To address this problem, we proposed the *Nebula* engine that complements the state-of-art in annotation management with proactive capabilities. The key contributions introduced by Nebula are: (1) Proposing techniques for analyzing the annotations' content and discovering the embedded references, (2) Extending the state-of-art in RDBMS-based keyword search techniques to efficiently search and find the data tuples corresponding to the embedded references, (3) Proposing various annotation-aware optimizations including focal-based weight adjustment, approximate searching with focal-based spreading, and multi-query shared execution, and (4) Developing an expert-enabled verification mechanisms that adaptively minimize the experts' involvements while maintaining high-accuracy predictions. The experimental evaluation illustrates the significant gain in enhancing the quality of annotated databases, the effectiveness of the proposed optimization techniques, and the reduction in the time- and resource-consuming process of manual curation.

10. REFERENCES

[1] EcoliHouse: A Publicly Queryable Warehouse of E. coli K12 Databases. http://www.porteco.org/.

[2] PubChem- Database of Chemical Molecules. https://pubchem.ncbi.nlm.nih.gov.

[3] The Universal Protein Resource Databases (UniProt) . http://www.ebi.ac.uk/uniprot/.

[4] WordNet: Lexical Database. . *http://wordnet.princeton.edu/*.

[5] DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, pages 5–16, 2002.

[6] P. Alper, K. Belhajjame, C. Goble, and P. Karagoz. Small Is Beautiful: Summarizing Scientific Workflows Using Semantic Annotations. In *IEEE BigData Congress*, pages 318–325, 2013.

[7] S. Bergamaschi, E. Domnori, F. Guerra, R. Trillo Lado, and Y. Velegrakis. Keyword Search over Relational Databases: A Metadata Approach. In *SIGMOD*, pages 565–576, 2011.

[8] S. Bergamaschi, C. Sartori, F. Guerra, and M. Orsini. Extracting Relevant Attribute Values for Improved Search. *IEEE Internet Computing*, 11(5):26–35, 2007.

[9] D. Bhagwat, L. Chiticariu, and W. Tan. An annotation management system for relational databases. In *VLDB*, pages 900–911, 2004.

[10] S. Bowers and B. LudŁscher. A Calculus for Propagating Semantic Annotations through Scientific Workflow Queries. In *In Query Languages and Query Processing (QLQP)*, 2006.

[11] S. Bowers, T. Mcphillips, S. Riddle, M. K. Anand, and B. Ludascher. Kepler/pPOD: Scientific Workflow and Provenance Support for Assembling the Tree of Life. In *Provenance and Annotation Workshop (IPWA)*, pages 70–77, 2008.

[12] P. Buneman, J. Cheney, W.-C. Tan, and S. Vansummeren. Curated databases. In *Proceedings of the 27th ACM symposium on Principles of database systems (PODS)*, pages 1–12, 2008.

[13] P. Buneman and et. al. On propagation of deletions and annotations through views. In *PODS*, pages 150–158, 2002.

[14] P. Buneman, S. Khanna, and W. Tan. Why and where: A characterization of data provenance. *Lec. Notes in Comp. Sci.*, 1973:316–333, 2001.

[15] P. Buneman, E. V. Kostylev, and S. Vansummeren. Annotations are relative. In *Proceedings of the 16th International Conference on Database Theory*, ICDT '13, pages 177–188, 2013.

[16] L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. DBNotes: a post-it system for relational databases based on provenance. In *SIGMOD*, pages 942–944, 2005.

[17] R. Cole, J. Mariani, and H. Uszkoreit, editors. *Survey of the State-of-the-Art in Human Language Technology*. Cambridge University Press and Giardini, 1997.

[18] M. Y. Eltabakh, W. G. Aref, A. K. Elmagarmid, M. Ouzzani, and Y. N. Silva. Supporting annotations on relations. In *EDBT*, pages 379–390, 2009.

[19] W. Gatterbauer, M. Balazinska, N. Khoussainova, and D. Suciu. Believe it or not: adding belief annotations to databases. *Proc. VLDB Endow.*, 2(1):1–12, 2009.

[20] F. Geerts, A. Kementsietsidis, and D. Milano. Mondrian: Annotating and querying databases through colors and blocks. In *ICDE*, pages 82–93, 2006.

[21] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4):34–41, 2005.

[22] K. Ibrahim, D. Xiao, and M. Y. Eltabakh. Elevating Annotation Summaries To First-Class Citizens In InsightNotes. In *EDBT Conference*, 2015.

[23] E. V. Kostylev and P. Buneman. Combining dependent annotations for relational algebra. In *Proceedings of the 15th International Conference on Database Theory (ICDT)*, pages 196–207, 2012.

[24] R. Kumar and A. Tomkins. A Characterization of Online Browsing Behavior. In *Proceedings of the International Conference on World Wide Web*, pages 561–570, 2010.

[25] Q. Li, A. Labrinidis, and P. K. Chrysanthis. ViP: A User-Centric View-Based Annotation Framework for Scientific Data. In *SSDBM*, pages 295–312, 2008.

[26] X. Li, M. Zhang, Y. Liu, S. Ma, Y. Jin, and L. Ru. Search Engine Click Spam Detection Based on Bipartite Graph Propagation. In *Proceedings of the 7th ACM International Conference on Web Search and Data Mining*, pages 93–102, 2014.

[27] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: Top-k Keyword Query in Relational Databases. In *SIGMOD*, pages 115–126, 2007.

[28] G. Palma and et. al. Measuring Relatedness Between Scientific Entities in Annotation Datasets. In *International Conference on Bioinformatics, Computational Biology and Biomedical Informatics*, pages 367:367–367:376, 2013.

[29] N. P. Savchuk, K. V. Balakin, and S. E. Tkachenko. Exploring the chemogenomic knowledge space with annotated chemical libraries. *Current Opinion in Chemical Biology*, 8(4):412, 2004.

[30] S. Tata and G. M. Lohman. SQAK: Doing More with Keywords. In *SIGMOD*, pages 889–902, 2008.

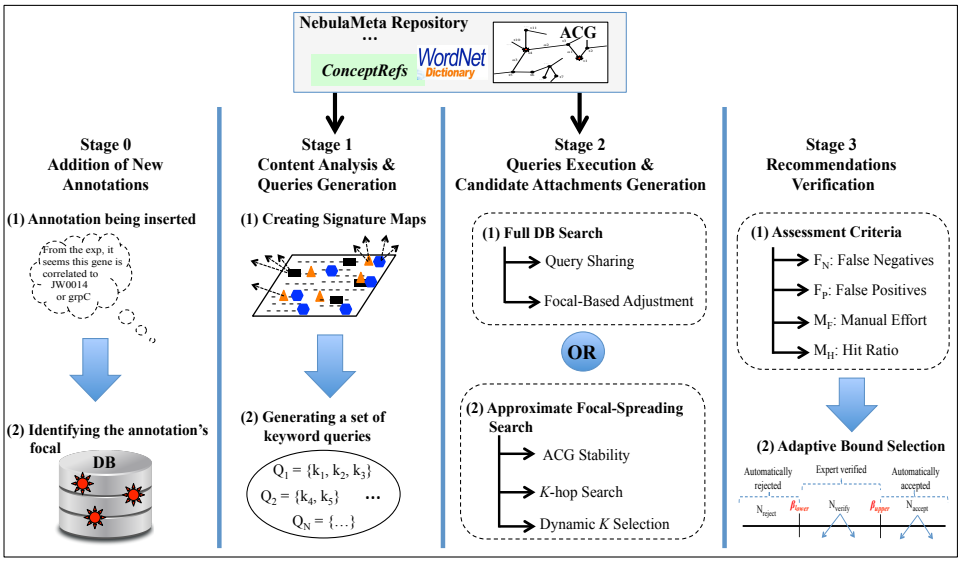
[31] D. Xiao, A. Bashllari, and T. M. M. Y. Eltabakh. Even Metadata is Getting Big: Annotation Summarization using InsightNotes. In *SIGMOD Conference*, 2015.

[32] D. Xiao and M. Y. Eltabakh. InsightNotes: summary-based annotation management in relational databases. In *SIGMOD Conference*, pages 661–672, 2014.

APPENDIX

A. SUPPORTING FIGURES

In this section, we include several supporting and illustrative figures. Figure 16 presents the main processing stages of the Nebula system along with the key contributions in each stage, which are: *Stage 0:* The modeling of an annotated database, the insertion of new annotations, and identifying the focal points (Section 3), *Stage 1:* The analysis of the annotation's content, constructing signature maps, and the generation of keyword-search queries (Section 5), *Stage 2:* The execution of the queries either over the entire database, or over a small subset surrounding the focal points (Section 6), and *Stage 3:* The verification of the generated candidate attachments and assessing the results' quality (Section 7). Figure 17 presents a detailed pseudocode for the *ContextBasedAdjustment()* Function, which is introduced in Section 5.2.2, and called from the *QueryGeneration()* Function (Figure 4(a)). The function adjusts the weights assigned to each candidate keyword in the annotation based on the keyword's surrounding context. Finally, Figure 18 presents a visual representation of the experimental datasets and the annotation workload based on which Nebula is evaluated (Section 8).



Nebula's Main Processing Stages. Nebula has three main processing stages triggered by the insertion of a new annotation (Stage 0), which are:

Stage 1: In which the annotation's content is analyzed, the metadata information (NebualMeta) is consulted, and signature maps are generated. The signature maps highlight the keyword combinations that potentially represent embedded references. Those combinations will create a pool of *keyword-search* queries.

Stage 2: In which the queries are executed to produce candidate attachments between the new annotation and other tuples in the DB. Nebual has two execution modes: (1) Full DB Search, and (2) Approximate Focal-Spreading Search. In both modes, NebulaMeta plays a key role in adjusting the confidence degree, and/or limiting the search space. The results are candidate attachments called "*verification tasks*".

Stage 3: In which the verification tasks are either automatically accepted, automatically rejected, or manually verified by experts (depending on their confidence scores). Nebula deploys adaptive algorithms to set the bounds between these three decisions. The goal is to achieve a high-quality prediction while maintaining experts' involvement to a minimal acceptable degree.

Figure 16: The Main Processing Stages of Nebula.

ContextBasedAdjustment()

Input: - Context-Map
- α = Influence range size // Number of words on both sides

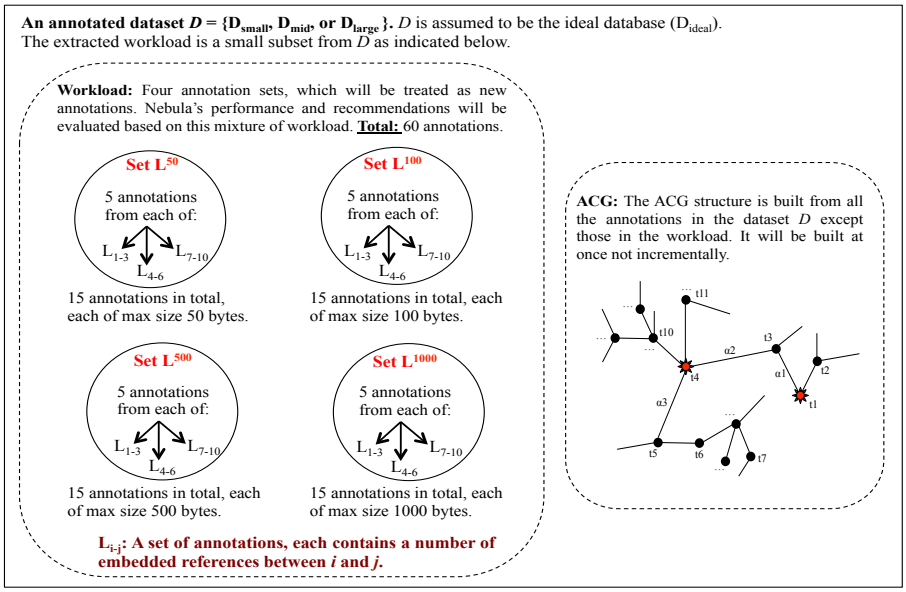
Output: Updated Context-Map

1. **For** (each word *w* in Context-Map) **Loop**
2. *w*.range = Form an influence range around *w* of size α (both sides)
3. **For** (each potential mapping *w*.mapping of *w*) **Loop**
4. **If** (*w*.mapping can form with other mappings in *w*.range a Type-1 match) **Then**
5. Increment *w*.mapping weight by β_1% for each match
6. **Else If** (*w*.mapping can form with other mappings in *w*.range a Type-2 match) **Then**
7. Increment *w*.mapping weight by β_2% for each match
8. **Else If** (*w*.mapping can form with other mappings in *w*.range a Type-3 match) **Then**
9. Increment *w*.mapping weight by β_3% for each match
10. **End If**
11. **End Loop**
12. **End Loop**

ContextBasedAdjustment() Function (Section 5.2.2). The function loops over each word *w* in *Context-Map* and creates an influence range around *w* called "*w.range*" (Line 2). The influence range *w.range* is α words to the left and to the right of *w*. This range represents the surrounding context of *w* within which the matching patterns are most likely to be found.

For each potential mapping of *w*, referred to as *w.mapping*, the function will search within *w.range* for any of *Type-1* matching. If any can be formed, then for each match, the weight of *w.mapping* will be increased by β_1% (Lines 4-5). Otherwise, the search continues to the lower-ranked matches, i.e., *Type-2* and *Type-3* matches, as illustrated in Lines 6-10. Any match of these types will increment the weight of *w.mapping* by β_2%, or β_3%, respectively.

Figure 17: Pseudocode of the *ContextBasedAdjustment()* Function.



Experimental Datasets and Workload (Section 8.1). Nebula is evaluated under three datasets D_{small}, D_{mid}, D_{large}. From each dataset, a workload is extracted as illustrated in the figure. The total number of annotations within a workload is 60 divided into 4 disjoint groups: L^{50}, L^{100}, L^{500}, and L^{1000}. The annotations in each group are further selected from 3 disjoint subsets: L_{1-3}, L_{4-6}, and L_{7-10}.

The ACG structure is built from all of the annotations in the dataset (D_{small}, D_{mid}, or D_{large}) excluding the annotations selected within the workload.

Figure 18: Visual Representation of the Experimental Datasets and the Extracted Annotation Workload.