Exploring Big Data with Helix: Finding Needles in a Big Haystack

Jason Ellis
IBM Research
jasone@us.ibm.com

Achille Fokoue
IBM Research
achille@us.ibm.com

Oktie Hassanzadeh
IBM Research
hassanzadeh@us.ibm.com

Anastasios Kementsietsidis
IBM Research
tasosk@ca.ibm.com

Kavitha Srinivas
IBM Research
ksrinivs@us.ibm.com

Michael J. Ward
IBM Research
michaeljward@us.ibm.com

ABSTRACT

While much work has focused on efficient processing of Big Data, little work considers how to understand them. In this paper, we describe Helix, a system for guided exploration of Big Data. Helix provides a unified view of sources, ranging from spreadsheets and XML files with no schema, all the way to RDF graphs and relational data with well-defined schemas. Helix users explore these heterogeneous data sources through a combination of keyword searches and navigation of linked web pages that include information about the schemas, as well as data and semantic links within and across sources. At a technical level, the paper describes the research challenges involved in developing Helix, along with a set of real-world usage scenarios and the lessons learned.

1. INTRODUCTION

Enterprises are captivated by the promise of Big Data, to develop new products and services, and to gain new insights into their customers and businesses. But with the promise of Big Data comes an expanded set of problems: *how do enterprises find the data they need for specific purposes*; *how do they make those data usable*; *how do they leverage learning about the data and expertise with the data across tasks*; and *how do they do all these in an efficient manner*. The research community has been quick in responding to enterprise needs but has mostly focused on the problem of efficiency. Indeed, significant effort has focused on providing scalability in the consumption and analysis of terabytes of data [1]. However, there has been little work on addressing in a practical manner the first set of problems, namely, how to help enterprises find *where* all the sources relevant to a task are located; *what* the relationships are between these sources; and *what* data in the sources are relevant to the task and what should be ignored. From our experience, these are the first problems with which customers are coming to us, and only after these problems are addressed it is possible to move to the next step of an efficient analytics solution.

This paper presents Helix, a system we have been actively developing for the last four years that has been used for exploratory analysis of data in various domains. Helix addresses the aforementioned problems by supporting users in iteratively exploring and combining data from multiple heterogeneous sources. Helix users may not have the background, interest, or time to understand all the schemas and data across all sources. However, they all understand what kind of data they are interested in and how they want to use it, so semantic technologies have the potential to bridge from user intention to data representation.

The first technical challenge we address in Helix is bringing together metadata from a variety of models that range across relational, XML, JSON, RDF and even Excel files. We are building a *semantic knowledge base* that describes the entire data corpus. Unlike existing approaches which focus primarily on pairwise integration of schemas within the same model [21], Helix considers multiple schemas across a variety of models. Source schemas may be readily available and can be extracted when sources are added to the system, or are automatically discovered by Helix when sources without schemas are incorporated. Since our approach is geared towards Big Data, even dealing with just the schemas to construct the knowledge base results in scalability issues which need to be addressed [9].

The second technical challenge we address is how to support users in finding relevant data. Keyword search is an obvious starting point, but presenting results from diverse sources differs from presenting document search results. For instance, if a keyword hit is a column name, what would the user like to see? Would other columns in the same ta-

ble be relevant? Or, would sample column values (and values from adjacent columns) suffice? What if the hit occurs in a table row, or in a value of a JSON tree? Do we show the entire row (with possibly hundreds of columns) or the entire JSON tree? There are both strong user experience and data management aspects to this challenge. Currently, Helix guides users in exploring the context in which search results occur using both *schematic* and *semantic* recommendations. The schematic recommendations are consistently presented regardless of the underlying data model. For semantic exploration, we leverage the constructed knowledge base to recommend other data elements, whether in the same or different sources, that may be worth exploring. As users initiate keyword searches and explore data sets through our interface, there is an underlying mechanism that uses the input keyword(s) and user navigational information to build (structured) queries that fetch relevant data from respective sources. In that manner, Helix users can focus on *how to accumulate and filter data of interest rather than focus on how to build queries in various dialects (e.g. SQL, SPARQL, Excel APIs).*

The third technical challenge we address is the discovery of interesting links between diverse sets of instance data. Existing works perform instance link discovery in a *batch* fashion. With huge data sets, and large numbers of sources, these methods generate every possible link, between all possible linkable entities. Generating *all* links not only requires substantial computation time and considerable storage space, but also requires substantial effort since the links must be verified and cleaned. Resources aside, it seems like a huge waste to generate all links only to discover later that actually only a small fraction of them is relevant to most tasks. In Helix, we address the shortcomings of existing approaches by providing a *dynamic* and *context-dependent* linking mechanism. Therefore, we allow the user to specify through metadata which types of links and which data collections she is interested in linking. We then generate at *run-time* links for these data using techniques we have developed for this purpose [15]. When a user has no idea as to what is linkable, we use our automatic identification of linkage points [16] to make recommendations.

The fourth challenge we address is how to assist users in building collections of data elements that are valuable to other users in other tasks. We leverage the internal knowledge base to build a recommendation system for users based on the specific data elements they find interesting. When users choose a specific piece of recommended data, and

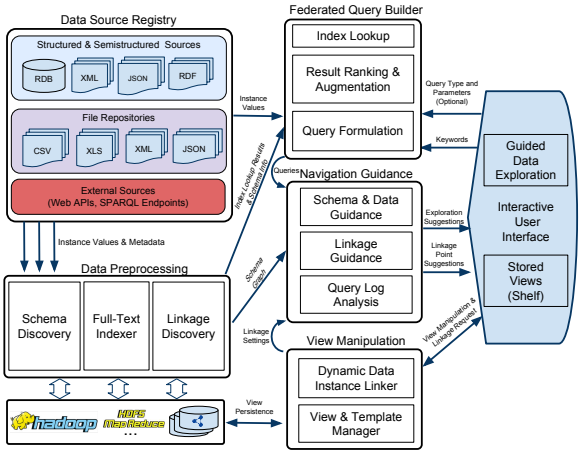

Figure 1: Helix Architecture

treat them as part of the same larger data set, we incrementally build up a high quality set of mappings across users. As more users interact with data, the system builds up knowledge about common portions of the data that get accessed together. This is a bootstrapping approach to building a common semantic model that applies across disparate data silos, and reflects a lower cost to building more formal semantic models/ontologies that can help serve a common view over disparate data.

The overall Helix architecture is shown in Figure 1. We present details of different components of Helix (and respective challenges) by following logically the way the system is used in practice. So, in Section 2, we describe the pre-processing phase in Helix which includes schema discovery, linking, and indexing of input data sources. Section 3 presents the guided data exploration mechanism, while Section 4 presents several usage scenarios. Section 5 discusses related work, and Section 6 concludes the paper with a summary and a few interesting directions for future work.

2. PRE-PROCESSING PHASE

All input data sources in Helix are defined in the data source registry component. There are three classes of sources considered. The first class includes (semi-)structured sources with predefined schemas and query APIs, such as relational databases and triplestores. The second class is (online or local) file repositories, such as the ones published by governments (e.g., data.gov or data.gov.uk, or data sources published by U.S. National Library of Medicine), or in cloud-based file repositories (e.g., Amazon S3). Finally, the third class are those sources directly read from online Web APIs, e.g., data read using the Freebase API.

Figure 2: Example JSON Data

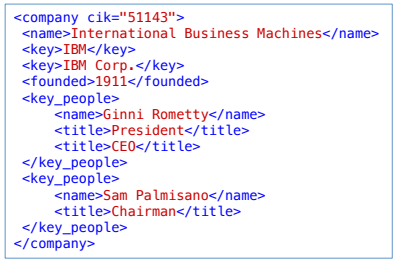

Figure 3: Example XML Data

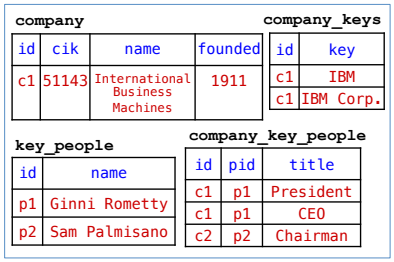

Figure 4: Example Relational Data

One of the goals in Helix is to process data based on explicit user needs and avoid unnecessary or expensive pre-processing given that we are dealing with Big Data. Therefore, the data pre-processing phase comprises only three *essential* steps, all performed in a highly scalable fashion implemented in the Hadoop ecosystem: (a) schema discovery, where each input source schema is represented in a common model in the form of a *local schema graph*; (b) full-text indexing, where data values and source metadata are indexed; and (c) linkage discovery, that incorporates instance-based matching and clustering of the (discovered) schemas. The outcome of the pre-processing phase is a *Global Schema Graph* which plays a key role in Helix. In the following, we discuss briefly each of the steps and how the Global Schema Graph is constructed.

2.1 Schema Discovery

The schema discovery process begins by constructing a *local schema graph* for each of the input sources. Intuitively, schema graphs are used as the common schema representation format that alleviates the differences across the models of different sources. We distinguish two types of nodes in the schema graphs: (a) *attributes*, which across models correspond to schema elements whose domain is literal values (e.g., column names in the relational model, PCDATA elements in XML, strings in JSON, etc); and *types*, which across models correspond to schema elements whose domain is defined (recursively) through other type or attribute nodes (e.g., tables or schemas in the relational model, intermediate elements in XML trees, etc.).

Given the wide range of sources considered in terms of data models, the local schema graph construction is customized to each data model. In more detail, for semistructured data with no provided schema, we first build a minimal schema graph (more precisely a tree) [17] that contains all the possible paths in the semistructured data instances. That is, nodes in the constructed local schema graph correspond to elements of the semistructured data, and a path in the schema graph

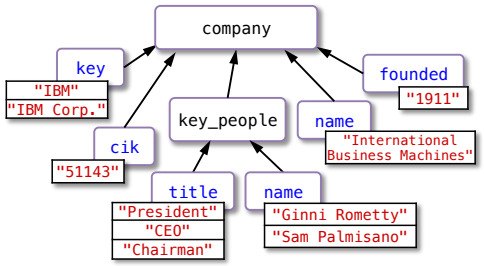

Figure 5: Example Local Schema Graph

exists if there is at least one data instance having that path. To illustrate, Figures 2 and 3 are two examples that show essentially the same data in both JSON and XML formats. Figure 5 shows a schema graph that can be extracted from either of them. In the graph, `company` and `key_people` are types, while `name` is an attribute. Our local schema graph construction is similar to that of approximate DataGuides [10] or representative objects [20]. Since our goal is a simple and concise representation of the structure of the data, we do not employ techniques that try to derive an accurate conceptual model of the input data. Instead, we use techniques that provide a schema that initially facilitates users' understanding of data, while providing ways to incrementally build more complex and accurate models through the user interaction and input.

For (semi-)structured data with a provided schema (e.g., through available metadata or DDLs), the schema graph can be constructed without analyzing instance values. So, for the relational source in Figure 4, the schema graph of Figure 5 can be constructed by using the table definitions and foreign key relationships. If schema definitions including foreign key information are not available, one type is created per table with table columns again becoming attributes of the type, and a foreign key discovery process [23] is performed along with link discovery (see Section 2.3) to connect the different tables.

For RDF (graph) data, each class is represented with a type, and its properties that have literal values become type attributes. Properties that link

two classes in RDF result in links between the corresponding types. Here again, if an RDF Schema is given, then it is used to construct the local schema graph. In the absence of an RDF schema, Helix infers the schema graph by going through all the instance data (RDF triples) and figuring out the resources, their classes (i.e., the `rdf:type` statements), their properties, and the properties between classes.

Once the schema graph for each source is constructed, a Uniform Resource Identifier (URI) is assigned to each node. The assignment is such that it not only captures the relative position of each node within the schema graph, but also captures the provenance of each node, e.g., the name of the source as well as its model. As an example, assume the schema graph in Fig. 5 is derived from a JSON source with name `compdata`. The URI for the property `cik` of object `company` is `json://compdata/company/cik`. Similarly, if the relational source in Fig. 4 is registered with name `cmpdt` and the tables are stored in a schema named `db2usr1`, `rdb://cmpdt/db2usr1/company/name` is the URI of the node corresponding to column `name` in table `company`.

2.2 Full-Text Index

In Helix, each instance value is also assigned a URI that can be used to locate that specific value. Figure 5 shows the instance values associated with each attribute in our example. We view instance values as documents and use a standard Information Retrieval (IR) engine to tokenize and index the instance values to allow standard keyword search across the input data sources with Boolean queries and fuzzy search capabilities. There are several ways to model instance values as documents:

- Treat each instance value as a document. So each instance value is assigned a URI that can be used to locate it. For example, URI `json://compdata/company[0]/key_people[0]/name[0]` is associated with "Ginni Rometty" in Figure 2.
- Treat all instance values of an attribute as a single document, with the attribute URI as the document identifier. For example, values "IBM" and "IBM Corp." in Figure 2 can form a document associated with URI `json://compdata/company/key`.
- Group values that appear together in one instance of a type and treat them as a single document. This results in *row-level* indexing for relational data, node-based grouping of instance values for tree-structured (e.g., XML) data, and object-level indexing for graph (RDF) data. In Figure 2, this approach results in one index entry for the `company` type instance, and two entries for `key_people` instances.

We support all the above indexing mechanisms as options, and our experience shows that the first approach is superior to others, with respect to keyword search during guided exploration (see Section 3). However, a particular challenge with this approach is the size of the index, given the number of instance values and the fact that instance URIs are long strings. As a result, we have devised a simple compression mechanism to index each *distinct* value only once for each attribute. This does not impact guided exploration, whereas it results in much smaller index size in practice. Regardless of the indexing mechanism, the attribute URIs are also stored in the index to facilitate attribute-based grouping of the keyword search results. Unlike previous work on the indexing of heterogeneous data [8] which are limited to the indexing of values, our index is further extended with indexing of metadata, i.e., with the types and attribute names themselves to provide a unified keyword search index.

2.3 Linkage Discovery

The last phase in pre-processing is discovering links between different types and attributes *within* as well as *across* the schema graphs of different sources. Traditional schema-based matching is not effective in matching highly diverse and automatically-constructed schemas where the labels of schema elements are not always representative of their contents, and data come from several sources that use different models and representations. Therefore, our approach is to perform an all-to-all instance-based matching of all the attributes. Scaling the matching process for a large number of attributes and large number of instances per attribute is a major challenge. We address this problem by casting it into the problem of computing document similarity in information retrieval [9]. Specifically, we treat each attribute node as a document, and we consider the instance values for that attribute as the set of terms in the document. To scale the computation of pairwise attribute similarity, we use Locality Sensitive Hashing (LSH) techniques, as is done in computing document similarity. Briefly, we construct a fixed small number of *signature* values per attribute, based on MinHash [3] or Random Hyperplane [4], in a way that a high similarity between the set of signatures guarantees high similarity among instance values. This results in efficient comparison of instance values between attributes. We then create small buckets of attributes so that similar attributes are guaranteed to be in the same bucket with a high probability. This is similar to a common indexing technique used in

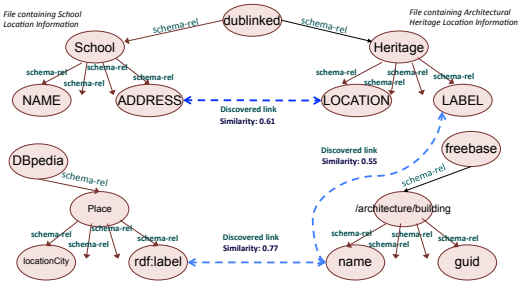

Figure 6: Sample Schema Graph

record linkage known as *blocking* [5]. Our experiments on large data sources show that our approach is very effective in reducing the number of pairwise attribute comparisons required for an all-to-all attribute matching [9].

In our evaluation, we found that the precision and recall of linkages between attributes with textual values is very good [9]. However, linkages between attributes with numeric or date/time values tend to have little semantic value, even when the similarity of their instances is high. Currently, we optionally filter attributes with these data types. We are investigating the scalability of *constraint-based instance matching* [21] for discovering linkages between such attributes.

The attribute-level linkages found within and across data sources are used not only for guided navigation of the sources (see Section 3), but also to find type-level linkages and grouping (clustering) of types. In more detail, type clustering is performed to group types that have the same or highly similar attribute sets. For example, all 'address' types of an XML source might create a single cluster, in spite of these addresses appearing at different levels and under different elements of the tree. Type-level linkages induce a similarity graph, where each node represents a type and the weight of an edge connecting two types reflects their similarity. This similarity is the average of (a) the instance-based similarity between the attributes of the two types; and (b) the Jaccard similarity between the sets of attribute labels of the types. An unconstrained graph clustering algorithm [13] is then used to find clusters of types in the similarity graph.

2.4 Global Schema Graph

The schema graphs of all the input sources along with discovered attribute and type linkages are all used to build the *Global Schema Graph*. This graph provides a unified view over the input sources, enables navigation, and allows the discovery of related attributes and types through schema and similarity-based linkages. Figure 6 shows a portion of a global schema graph constructed for one of our use cases. In this example, a data set on national heritage sites in the city of Dublin is linked to a data set in the same source containing school locations, based on the similarity of the address/location attributes in the two data sets. The data set is also linked to a type in a Web knowledge base that contains information on architectural buildings, which itself is linked to another knowledge base containing information about public locations (*Place* type in an ontology). These links implicitly show that these data sets contain information about locations, and that there is potentially a connection between school locations and national heritage sites in the city of Dublin, one of many exploration capabilities of Helix. In the figure, we distinguish two sorts of links, namely *explicit* links (drawn in solid black lines) that are inferred by looking at individual sources through schema discovery (see Section 2.1), and *discovered* links (drawn in dashed blue lines) that require additional logic and consider multiple sources (see Section 2.3). For discovered links, we add annotations to capture their computed similarity, as well as the method by which the link was discovered (e.g., MinHash, user generated, etc.).

The global schema graph is a key structure in Helix since it governs and guides user interactions (more details in Section 3). What is less obvious though is that there are technical challenges in terms of managing the graph itself. Helix is geared towards Big Data scenarios, and as more and more sources are incorporated into the system, the global schema graph very quickly becomes quite large. As the system continuously queries, updates, and augments the graph, it is important that all these operations are performed efficiently; otherwise the global schema graph ends up being a bottleneck to the system performance. To address these challenges, we store the global schema graph in our own graph store, called DB2RDF, which has been proven to outperform competing graph stores in a variety of query workloads using both synthetic and real data [2]. Our graph store supports the SPARQL 1.0 graph query language [24] and interactions with the global query graph are automatically and internally translated to SPARQL queries.

3. GUIDED EXPLORATION

Once a global schema graph is built, one might think a user could somehow use the global schema graph directly to query the data and navigate through related pieces of information. However, despite several efforts on our part to help users directly explore the schema graph (our first prototype followed this approach [14]), comprehending

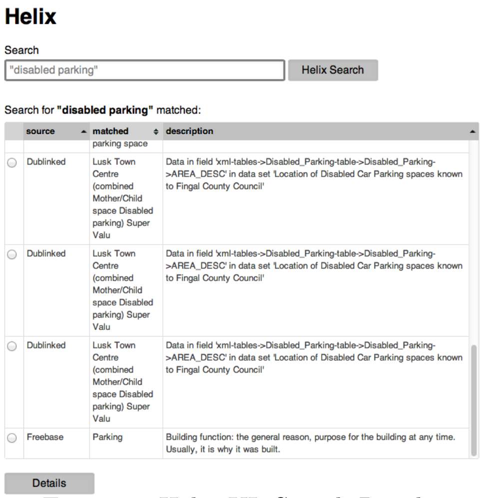

Figure 7: Helix UI: Search Results

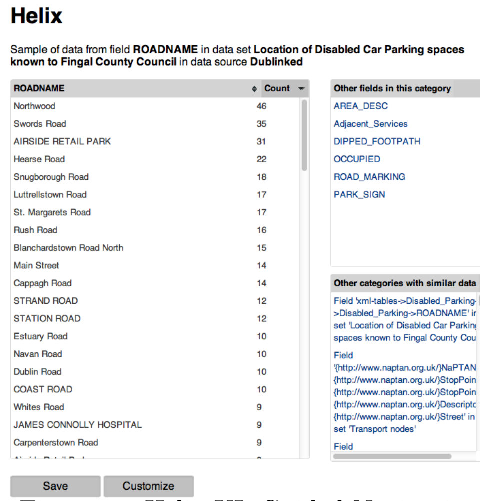

Figure 8: Helix UI: Guided Navigation

the myriad of schema elements and their connections turned out to be too much for users to grasp. We also tried using keyword searches directly on the global schema graph to help users construct structured queries as in [26], but this technique did not help the construction of complex queries required by some of our use case scenarios (Section 4). We have thus arrived at the approach of guided data exploration, which allows users to construct complex queries *iteratively* by building on some basic building blocks. Guided exploration in Helix has four components: (1) keyword search, (2) guided navigation, (3) construction of virtual views, and (4) integration across virtual views to build more complex queries.

3.1 Keyword Search

Users initiate their guided exploration with a keyword search over the index described in Section 2.2. Our search engine results (see Figure 7) are customized in such a manner that the result set contains not only the hits from the global schema graph for the input keyword(s) (with each hit being either a type or an attribute hit, and shown in the column labeled "matched" in Figure 7), but also the name of the data source in which each hit appears (column "source") as well as the precise location of the hit within the source (column "description"). As with any search, the search results are rank ordered, but they can be sorted by values in any of the columns. The search process is, by itself, not novel compared to those found in the literature. We use it simply to initiate an exploration in the system.

3.2 Guided Navigation

Guided data navigation is an iterative process that assists users in confirming that data that they are viewing are relevant to their tasks and in discovering closely related data elements. Users initiate navigation by choosing a hit on the search engine results page. Helix displays a screen with three primary components, a table containing sample data for the chosen hit and two lists of links to related attributes and types (see Figure 8). Clicking on one of these links takes the user to a new data navigation screen, this time populated with data and links for the attribute or type clicked on. The presentation is uniform regardless of the data model of the underlying data source. The user is guided to structurally and semantically related attributes and types, reinforced at each step with sample data. She can focus on the data itself rather than how it is represented and accessed. We describe the main features of the guided navigation interface next.

- Data sample: A sample of popular instance values are presented when a user drills down on a hit on an attribute. The objective is to help users decide if the selected hit is relevant to their task. If the hit is on a type, it is difficult to determine which of the attributes for the type should be displayed. In this case, the user can explore data associated with that type using the schema links described next.
- Schema links: Helix displays links found during the schema discovery process (see Section 2.1) for two reasons. First, schema links inform users about *other* attributes and types in the *vicinity* of the hit (i.e., in the same source) and may be relevant for their task. Second, we have observed that this sort of navigation often guides users to the right data even if the original hit was not on the right attribute. The list of schema links are navigable, and ordered by the number of instances in the attribute.
- Discovered links: For a given attribute or type

hit, we show links to other attributes or types that were discovered during pre-processing (see Section 2.3). These links are similar to recommendations for relevant data. Our evaluation of these links over Web-based data sources [9] indicates that the precision and recall over enterprise data sources is high as well. This list is ordered initially by the similarity score calculated during pre-processing. A user validates a discovered link indirectly by clicking on it and using the relevant data from it (as described in Section 3.3). When this occurs, we boost the link between the two types or attributes (the one in the hit and the one the user navigated to from the hit) to the maximal similarity value of 1.0, and we annotate the link to indicate that it is user validated. These links are subsequently ranked higher when users again browse the same hits and their associated links.

We have observed that the simple navigation techniques described here often help users find relevant data after a few navigation steps. Occasionally a hit is exactly what the user is looking for; at other times, the data of real interest are for a different attribute or type in the same source; and sometimes one of the discovered links guides the user to more relevant data in another source. The key message here is that, in practice, users rarely find what the are looking for in *one shot*. This is even the case in web searches. What is important is that Helix provides a whole infrastructure to help users *zero in* on the data they want to use. Our experience from different usage scenarios shows that this process is typically faster than using a pure-text search-based approach that lacks both the context of the hits and the connections with other related areas of the search space.

3.3 Virtual View Construction

When a user has found an interesting type, she can construct a virtual view on that type, and save it on a *data shelf*. The steps to accomplish the creation of a virtual view involve customizing the type on the guided navigation screen. At the interface level, the user chooses various attributes of a type in tabular form, and is never aware of the actual data format. The user actions available in this step include simple projections, filtering and ordering on any of the attributes of a selected type (see Figure 9). In the back end, because our internal representation of the global schema is a large graph, a 'virtual' view corresponds to a Basic Graph Pattern (BGP) in SPARQL, which (in its most basic form) starts with the template `?x type <T>`, where

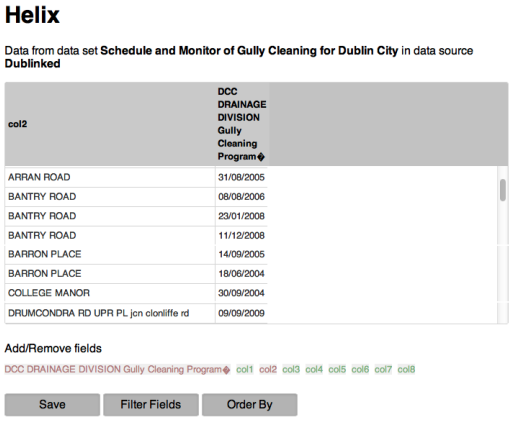

Figure 9: Helix UI: Virtual View Construction

`<T>` is a type, and adds statements of the form `?x <P> ?y`, as the user projects attribute `P`. Note that at the interface level, the user is not required to be aware of any formal structured query language. We build up the structured query as the user interacts with a simplified version of the data in its tabular form. Filtering and ordering are likewise internally represented using equivalent SPARQL operators.

Once a virtual view has been constructed, the user can save it on a *data shelf* which just saves a SPARQL query in the user's profile. The user can also annotate the virtual view with terms that more accurately describe it. These terms are also added to the full text index (where the 'document' being indexed is now a query), such that subsequent searches in guided exploration can lead to hits on virtual views as well as original datasets. As stated in the prior section, we consider saving a virtual view an indication of the usefulness of the guided navigation step and boost the similarity values of discovered links followed during the construction of the virtual view.

3.4 Integration of Virtual Views

The real power of Helix is that these simple virtual views can now be used as building blocks for more complex views. To build such views, we provide two operators which correspond loosely to a JOIN and a UNION operation. The difference between a standard JOIN operation and our notion of JOIN is that our version is actually a *semantic* JOIN that corresponds to an instance linkage operation between the instance data of different virtual views. The UNION operation is more straightforward as it has the usual semantics. Two issues require further explanation. First, we need to explain how we actually fetch that data for the virtual views that are to be joined (or unioned). Second, we need to explain how the actual join occurs.

In Helix, because the data sources being joined may not even be in the same format, and not all of the formats have uniform query engines (or even have an engine), we use the global schema graph as a semantic model that unifies distributed data sources with *direct schema mappings* from the global schema to each of the data sources schema. Differences in query engines (and languages) are then accommodated by using the global schema to local source schema mappings and knowledge of the query language/API associated with the underlying source to translate the view specification in SPARQL to an appropriate query (e.g., SQL for relational, XPath for XML) that is executed over the underlying source to fetch the instance data. Note that for data formats that do not support a query engine (like CSV or Excel), we house the data in a key-value store to account for variable schema across multiple datasets.

With the instance data in place, we need to perform the actual JOIN operation. Unless specified by the user, Helix recommends a set of attribute pairs on which the views may be linked. This recommendation is not purely based on the links found during the linkage discovery step described in Section 2.3, since attribute and type similarity does not always imply that instance-level linkages exist. For instance, two types may be linked because of similarities at the term level, but actual instance values might differ (e.g., *Bank of America* versus *Apple Bank*). Our recommendation of pairs of attributes for linkage is based on our work on automatic identification of linkage points [16]. Briefly, we recommend a pair of attributes as a linkage point if they can effectively be used to link instances of the two types. The linkage points are ranked based on their *strength* [16], which reflects the ability to establish a reasonable number of one-to-one links between the instances. Note that we consider all the attributes of the type in the view, not just those explicit in the stored query. The user can then choose a pair (or pairs) of the recommended attributes. Helix invokes its Dynamic Data Instance Linker (see Figure 1) that uses fuzzy matching techniques [15] to link the views and presents the results to the user, together with a confidence score (see Figure 10). The user can accept the results by saving them to the shelf or go back and choose different or additional linkage points and try again. When a new view is saved, internally the SPARQL algebra describing the view also records the selected linkage points. As an aside, since each view is a BGP in SPARQL, linkage points might occur between variables that don't necessarily have the same name in the SPARQL. The linkage is therefore expressed as a filter that calls a JOIN operation on the two variables that are being joined.

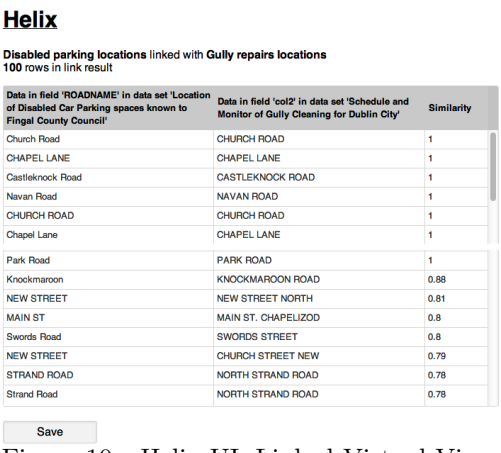

Figure 10: Helix UI: Linked Virtual Views

4. USAGE SCENARIOS

The design and implementation of the Helix system has gone through extensive evaluation using several usage scenarios in different domains. The majority of the usage scenarios are inspired by our interactions with customers, in trying to understand their needs in data exploration and help them with the first steps of their data analytics tasks. In this section, we describe two such usage scenarios and some of our key observations and lessons learned. We first describe details of usage scenarios using data published by the city of Dublin, Ireland. Extracting relevant information from online public data repositories such as those published by government agencies is a frequent request within enterprises. We then describe a customer relationship management (CRM) use case as an example enterprise data exploration scenario. Finally we share some of the lessons learned through these and other applications of Helix. Note that our goal here is not to perform a scientific study of the effectiveness of the algorithms implemented in the system (such as the study we have performed on accuracy of attribute-level linkages [9] and linkage point discovery [16]). Nor do we intend to evaluate the effectiveness of our user interface through a large-scale user study, which is a topic of future work and beyond the scope of this paper.

Table 1 provides a summary of the source characteristics in the two scenarios, and Table 2 provides the total number of links found across these sources. Each source is in itself composed of multiple data sets. We therefore provide a summary of the number of links between data sources, as well as the

Table 1: Summary of Data Sources

Data Source	Types	Instances	Tables/files
Bug reports	201	7M	1
Bug fixes	95	121M	7
Freebase	1,069	44M	NA
DBpedia	155	2M	NA
Dublinked	1,569	22M	485

Table 2: Links Across Data Sources' Types

Data Src/Data Src	#Links	Data Src/Data Src	#Links
Bug fixes/Bug fixes	1,510	Bug reports/Freebase	298
Bug fixes/Bug reports	1,209	Bug reports/Bug reports	316
Bug fixes/DBpedia	25	Dublinked/Dublinked	288,045
Bug fixes/Freebase	1,216	Dublinked/DBpedia	225
Bug reports/DBpedia	4	Dublinked/Freebase	2,351

summary of links within a single data source (e.g., a single data source like Dublinked is composed of several hundred files). The number of links is provided to demonstrate that the system computes a large number of them. It is not our intent here to characterize them by the standard metrics of precision and recall (cf. [9]). As the links are used primarily within the context of a rather focused search, we illustrate in the use cases below how sample links may help data discovery and analysis.

4.1 Dublinked

The city of Dublin has a set of data from different government agencies that is published in a number of different file formats (see: `http://dublinked.ie/`). At the time of this writing, Helix could access 203 collections. Each collection consists of multiple files, resulting in 501 files with supported formats that broke down into 206 XLS, 148 CSV, 90 DBF, and 57 XML files. Helix indexed and pre-processed 485 files, but 16 files could not be indexed due to parsing errors. Our main use case here is data integration across the different agency data, but we also decided to connect the Dublinked data to Freebase and DBpedia, to determine if we could use the latter two sources as some form of generic knowledge. For the pre-processing step, we processed DBpedia and Freebase as RDF dumps.

The value of integrating information across files and across government agencies is obvious, but we illustrate here a few examples, based on links discovered in our pre-processing step, in Table 3. Here are some examples of questions that a city official can now construct queries for, based on Helix discovered linkages in the data shown in the table:

1. Find schools that are polling stations, so that the city can prepare for extra traffic at schools during voting periods.
2. Find disabled parking stations that will be affected by pending gully repairs, to ensure accessibility will be maintained in a specific region of the city.
3. Find recycling stations that handle both cans and glass to route waste materials to the right stations.

Table 3: Sample Links for the Dublinked Scenario

Property Pairs	Score
`xml://School-Enrollment/Short-Name` → `xml://Polling-Stations-table/Name`	0.82
`csv://DisabledParkingBays/Street` → `csv://GullyRepairsPending/col2`	0.68
`xls://CanRecycling/col0` → `xls://GlassRecycling/col0`	0.71
`csv://PostersPermissionsOnPoles/Org` → `csv://CandidatesforElection2004/col2`	0.54
`csv://CandidatesforElection2004/col1` → `csv:/CandidatesforLocalElection2009/col5`	0.97
`csv://PlayingPitches/FACILITY-NAME` → `csv://PlayAreas/Name`	0.40
`csv://FingalNIAHSurvey/NAME` → `http://rdf.freebase.com/architecture/structure/name`	0.56
`dbf://Nature-Development-Areas/NAME` → `http://rdf.freebase.com/sports/golf-course/name`	0.55
`csv://ProtectedStructures/StructureName` → `http://dbpedia.org/HistoricPlace/label`	0.42

Table 4: Type Clusters for the Dublinked Scenario

Type Clusters

`xml://SchoolEnrollment20092010-1304`
`csv://SchoolEnrollment20102011-2139`
`xml://SchoolEnrollment20102011-2146`
`csv://SchoolEnrollment20082009-1301`
`xml://Schoolenrollment20082009-1303`

`csv://GullyCleaningDaily2004-11CENTRALAREA-1517`
`csv://GullyCleaningDaily2004-11NORTHCENTRALAREA-1518`
`csv://GullyCleaningDaily2004-11NORTHWESTAREA-1518`
`csv://GullyCleaningDaily2004-11SOUTHEASTAREA-1519`

4. Find organizations who have the most number of permissions to put posters on poles, to assess organizations with maximal reach to citizens.

In general, links alert users to the possibility of related data that could be pooled before any analytics is performed. For instance, any analytics on play areas would likely need to include the data in PlayAreas file as well as the Play pitches file. Similarly, time series analysis of election data would likely include the 2004 file as well as the 2009 file. Finally, links to external data sets can easily imbue the data with broader semantics. As examples, the Name column in the FingalNIAHSurvey file refers to architectural structures, but another column also called 'Name' in the Nature-Development-Areas file is really about golf courses or play areas. Similarly, the StructuredName column in the ProtectedStructures file is about historic structures.

Table 4 shows two sample type clusters that Helix discovered in the Dublinked data. Recall that type clusters are based on similarity of the schema elements in the type, as well as the instance similarity of each of those elements. The first cluster (files starting with `SchoolEnrollment*`) groups data by year despite changes in the data format. The second cluster (files starting with `GullyCleaningDaily2004*`) discovered by Helix groups data by area, as is apparent from the titles of the files. Following our design goals in Helix, the system itself is not trying to interpret the semantics of each discovered cluster. It will provide a tool for a knowledge worker to specify data sets for meta-analysis.

4.2 CRM

In most enterprises, maintaining a consistent view of customers is key for customer relationship management (CRM). This task is made difficult by the fact that the notion of a customer frequently changes with business conditions. For instance, if an enterprise has a customer *"IBM"* and also a customer *"SoftLayer"*, they are distinct entities up until the point that one acquires the other. After the acquisition, the two resolve to the same entity. The process of keeping these entities resolved and up to date in the real world is often a laborious manual process, which involves looking up mergers and acquisitions on sites like Wikipedia and then creating scripts to unify the companies involved. Our second scenario targets this use case. The real world sources involved are (a) a relational database with a single table that tracks defects against products (b) a relational database that tracks fixes for the defects in the defect tracking database (with 7 tables – one per product), (c) an RDF version of Wikipedia, from Freebase/DBpedia.

The query that the knowledge worker is interested in is a picture of the number of defects fixed for each customer (where each customer is grouped or resolved by merger and acquisition data). We highlight the features of Helix that help the user build this query. We illustrate what steps a user would take in Helix if her intent is to build a table of customer records of bugs and their corresponding fixes, accounting for the latest mergers and acquisitions. Note that because much of this data is proprietary, we do not display confidential results.

Step 1 The user issues a keyword search on a customer name, such as 'IBM', to see what they can find. The hits returned include records in the bug database, as well as nodes in the Freebase/DBpedia RDF graph which match IBM (e.g., IBM, IBM AIX, etc). The user then clicks a particular hit in the bug reports database to explore the record in the context of the original table/graph. The user sees the larger context for the table (other records in the column that contains IBM, and other columns in the table that are related to the `CUST-NAME` column within the same table). More importantly, the user finds other properties that also contain similar data (as an example, see Table 5 that shows some real links found by Helix). If the user browses the `CUST-NAME` column in the bug reports database, Helix recommends the `CUSTOMER` column in the bug fixes database, and the `/business/organization` type in the RDF Freebase graph based on the links.

Step 2 The next step is the creation by the user of multiple virtual views that are placed on the data shelf (see Figure 11). Step 2 is a direct outcome of the data exploration conducted by the user in Step 1, where the user finds relevant data, and now wants to subset it for their task. For this example, we assume the user creates 3 virtual views. The first view contains a subset of bug reporting data with the columns `CUSTOMER` and `BUG NUMBER`, the second contains a subset of the bug fixes data with the columns `BUG NO`, `FIX NO`, and the third contains a subset of Freebase data, with `/business/employer`, and its relationship to its acquisitions through the `/organization/companiesAcquired` attribute.

Table 5: Sample Links for the CRM scenario

Property Pairs	Score
`rdb://Bug-Reports/CUST-NAME` → `rdb://Bug-Fixes/Product1/CUSTOMER-NAME`	0.75
`rdb://Bug-Reports/CUST-NAME` → `rdb://Bug-Fixes/Product2/CUSTOMER-NAME`	0.74
`rdb://Bug-Reports/CUST-NAME` → `http://rdf.freebase.com/business-operation/name`	0.47
`rdb://Bug-reports/CUST-NAME` → `http://dbpedia.org/Company/label`	0.28

Step 3 This step involves using semantic joins to build more complex views customized for the user's task. Here, the user likely joins Views 1 and 2 on `BUG NUMBER` and `BUG NO` to create View 4 of bugs that were fixed for different customers. Then, the user joins Views 3 and 4 on `CUSTOMER` and `/organization/companiesAcquired` to create View 5 of bugs and fixes by customer, where the customer record also reflects any companies acquired by customers in the bug report/fixing sources. At this point, the user could union View 4 with View 5 to find a count of bugs and fixes delivered to a customer and any of its acquisitions. Figure 11 shows all the steps in the process. In the figure, notice that a bug like 210 which normally would only be attributed to customer *"SoftLayer"* is now also counted as part of the bugs for customer *"IBM"* since the latter acquired the former. Knowledge of these acquisitions can be used to further *refine* the result by, say, removing all *"SoftLayer"* entries since they are already considered as part of *"IBM"*.

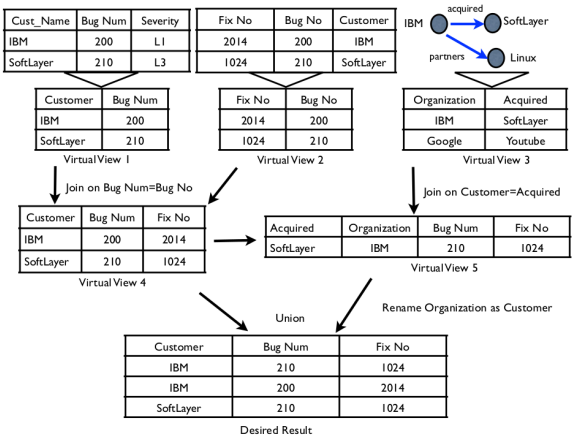

Figure 11: Steps in the CRM Scenario

4.3 Other Use Cases

In addition to the two use cases described above, we have used Helix in a number of other domains and use cases. We have found unexpected correlations among data sets that require examining instance linkages. An example is discovering a rather strong connection from company names whose products have been recalled to names of companies that make Peanut Butter products, compared to say, those that make H1N1 products or Vegetable Protein products. This scenario uses public data published by the U.S. Department of Health and Human Services. In such scenarios, the instance linkages performed through the user interface are examined to measure the relative importance of the discovered links. Another discovery is the use of unexpected labels, or changes to the type of data instances stored without a change in the schema. For example, in a scenario involving air quality data from various government agencies, we observed schema elements labeled as "zip" that contained string-valued addresses. Note that all these scenarios have been performed using a system that can be set up by non-expert users who have little or no technical background on data management technologies; they only need to specify data access mechanism for input data sources.

5. RELATED WORK

Our work builds upon and extends techniques from several research areas including data management, information retrieval, semantic web, and user interface design. In terms of the overall system, Helix can be seen as a novel DataSpace Support Platform (DSSP). It offers many of the features envisioned for DSSPs by Halevy et al. [12] including dealing with data and applications in a wide variety of formats, providing keyword search over heterogeneous sources, and providing "tools and pathways to create tighter integration of data in the space as necessary". To our knowledge, Helix is the first such system that allows generic "pay-as-you-go" integration mechanism that works on heterogeneous data across a wide variety of domains and applications.

Previous work has proposed systems that perform analysis and "pay-as-you-go" integration in specific domains. Kite [22] supports keyword search over multiple unrelated relational databases, but semantic integration is achieved using foreign key discovery techniques that are not well-adapted to tree- and graph-based data sources, and it does not scale well to a large number of databases. iTrails [27] accommodates semi-structured data sources, and its internal data model is similar to Helix's local schema graphs. Its integration semantics are provided through *trails*, query rewriting rules specified in an XPath-like language. These trails are written by the system provider, though the authors propose methods for automated generation. The Q system [25] has several features similar our system: it constructs a schema graph over structured and semi-structured data sources, with edges representing both structural and semantic relationships, including ones discovered through schema matching; and its users provide feedback to improve query results. Q's domain is primarily scientific data sources that are reasonably well-curated. It generates queries from keywords by computing top-k tree matches over the schema graph, then displays results directly to users, while Helix utilizes an iterative, link-following approach to incrementally build results. With Q, users provide explicit feedback on results based on both the answers and the queries (trees) that produced the answers, requiring a fair degree of technical sophistication; our system derives implicit feedback based on what users do with results. Google Fusion Tables [11] emphasizes data sharing, visualization, and collaboration on Web-based tabular data sources, with only simple data integration support. QuerioCity [18] is designed to catalog, index, and query city government data. Its approach is similar to our support of semistructured file repositories, although the specific domain allows certain tasks to be further automated (e.g., automatic linkage to DBpedia entities and types).

Our work on the user interface is related to research on providing non-expert users with search and query capability over standard database management systems. In particular, our goal is providing an *exploratory search* [19] mechanism over large heterogeneous data sources for users to not only perform "lookup", but also "learn" and "investigate". In terms of UI elements, relevant to our work is the Explorator system [7], that allows users to navigate through RDF data and run SPARQL queries by creation of facets and set operations. Our use of social guidance is also similar in nature to the social aspects of RExplorator [6], which extends Explorator with the ability to reuse results previously found by other users. Helix notably differs from the previously mentioned prior work [6, 7, 19, 28] in its support for navigation through heterogeneous data and its unique *online* linkage discovery capability.

6. CONCLUSION

In this paper, we described Helix, a system that allows knowledge workers and data scientists to *explore* a large number of data sources using a unified

intuitive user interface. Users can find portions of the data that are of interest to them using simple keyword search, navigate to other relevant portions of the data, and iteratively build customized views over one or more data sources. These features rely on highly scalable schema and linkage discovery performed as a pre-processing step, combined with online (and in part social) guidance on linkage and navigation. We demonstrated capabilities of our system through a number of usage scenarios.

We are currently working on extending Helix in a number of directions. On the user interface side, we are extending the social guidance feature through more complex query log analysis. Our goal is to predict a user's future steps through profiles based on similarity analysis of previous users' queries. We are also working on making user views accessible through a standard (RDF/SPARQL) API. This will allow non-expert users to build a custom and potentially complex knowledge graph, an alternative to the expensive, laborious task of building an enterprise-scale ontology for data analytics.

7. REFERENCES

[1] D. Agrawal, S. Das, and A. El Abbadi. Big Data and Cloud Computing: Current State and Future Opportunities. In *EDBT*, pages 530–533, 2011.

[2] M. A. Bornea, J. Dolby, A. Kementsietsidis, K. Srinivas, P. Dantressangle, O. Udrea, and B. Bhattacharjee. Building an Efficient RDF Store over a Relational Database. In *SIGMOD*, pages 121–132, 2013.

[3] A. Z. Broder. On The Resemblance and Containment of Documents. In *SEQUENCES*, pages 21–29. IEEE Computer Society, 1997.

[4] M. S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *STOC*, pages 380–388, 2002.

[5] Peter Christen. A Survey of Indexing Techniques for Scalable Record Linkage and Deduplication. *IEEE Trans. Knowl. Data Eng.*, 24(9):1537–1555, 2012.

[6] Marcelo Cohen and Daniel Schwabe. RExplorator - Supporting Reusable Explorations of Semantic Web Linked Data. In *ISWC Posters&Demos*, 2010.

[7] S.F.C. de Araújo and D. Schwabe. Explorator: A Tool for Exploring RDF Data through Direct Manipulation. In *LDOW2009*, 2009.

[8] X. Dong and A. Y. Halevy. Indexing Dataspaces. In *SIGMOD*, pages 43–54, 2007.

[9] S. Duan, A. Fokoue, O. Hassanzadeh, A. Kementsietsidis, K. Srinivas, and M. J. Ward. Instance-Based Matching of Large Ontologies Using Locality-Sensitive Hashing. In *ISWC*, pages 49–64, 2012.

[10] R. Goldman and J. Widom. Approximate dataguides. In *Proceedings of the Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats*, pages 436–445, 1999.

[11] H. Gonzalez, A. Y. Halevy, C. S. Jensen, A. Langen, J. Madhavan, R. Shapley, W. Shen, and J. Goldberg-Kidon. Google Fusion Tables: Web-Centered Data Management and Collaboration. In *SIGMOD*, pages 1061–1066. 2010.

[12] A. Y. Halevy, M. J. Franklin, and D. Maier. Principles of dataspace systems. In *PODS*, pages 1–9, 2006.

[13] O. Hassanzadeh, F. Chiang, R. J. Miller, and H. C. Lee. Framework for Evaluating Clustering Algorithms in Duplicate Detection. *PVLDB*, 2(1):1282–1293, 2009.

[14] O. Hassanzadeh, S. Duan, A. Fokoue, A. Kementsietsidis, K. Srinivas, and M. J. Ward. Helix: Online Enterprise Data Analytics. In *WWW*, pages 225–228, 2011.

[15] O. Hassanzadeh, A. Kementsietsidis, L. Lim, R. J. Miller, and M. Wang. Semantic Link Discovery over Relational Data. In *Semantic Search over the Web*, pages 193–224. 2012.

[16] O. Hassanzadeh, K. Q. Pu, S. Hassas Yeganeh, R. J. Miller, M. Hernandez, L. Popa, and H. Ho. Discovering Linkage Points over Web Data. *PVLDB*, 6(6):444–456, 2013.

[17] S. Hassas Yeganeh, O. Hassanzadeh, and R. J. Miller. Linking Semistructured Data on the Web. In *WebDB*, 2011.

[18] V. Lopez, S. Kotoulas, M. L. Sbodio, M. Stephenson, A. Gkoulalas-Divanis, and P. M. Aonghusa. QuerioCity: A Linked Data Platform for Urban Information Management. In *ISWC*, pages 148–163, 2012.

[19] G. Marchionini. Exploratory Search: From Finding to Understanding. *CACM*, 49(4):41–46, 2006.

[20] S. Nestorov, J. D. Ullman, J. L. Wiener, and S. S. Chawathe. Representative Objects: Concise Representations of Semistructured, Hierarchial Data. In *ICDE*, pages 79–90, 1997.

[21] E. Rahm and P. A. Bernstein. A Survey of Approaches to Automatic Schema Matching. *VLDB Journal*, 10(4):334–350, 2001.

[22] M. Sayyadian, H. LeKhac, A. Doan, and L. Gravano. Efficient Keyword Search Across Heterogeneous Relational Databases. In *ICDE*, pages 346–355. 2007.

[23] Y. Sismanis, P. Brown, P. J. Haas, and B. Reinwald. GORDIAN: Efficient and Scalable Discovery of Composite Keys. In *VLDB*, pages 691–702, 2006.

[24] SPARQL Query Language for RDF. `http://www.w3.org/TR/rdf-sparql-query/`.

[25] P. P. Talukdar, M. Jacob, M. S. Mehmood, K. Crammer, Z. G. Ives, F. Pereira, and S. Guha. Learning to Create Data-Integrating Queries. *PVLDB*, 1(1):785–796, 2008.

[26] T. Tran, H. Wang, S. Rudolph, and P. Cimiano. Top-k Exploration of Query Candidates for Efficient Keyword Search on Graph-Shaped (RDF) Data. In *ICDE*, pages 405–416, 2009.

[27] M. A. Vaz Salles, J.-P. Dittrich, S. K. Karakashian, O. R. Girard, and L. Blunschi. iTrails: Pay-as-you-go Information Integration in Dataspaces. In *VLDB*, pages 663–674. 2007.

[28] R. W. White, S. M. Drucker, G. Marchionini, M. A. Hearst, and m. c. schraefel. Exploratory Search and HCI: Designing and Evaluating Interfaces to Support. Exploratory Search Interaction. In *CHI Extended Abstracts*, pages 2877–2880, 2007.