Just-in-Time Information Extraction using Extraction Views

Amr El-Helw
EMC Corp.
amr.elhelw@emc.com

Mina H. Farid
University of Waterloo
mfathy@uwaterloo.ca

Ihab F. Ilyas
Qatar Computing Research Institute (QCRI)
ikaldas@qf.org.qa

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.4 [**Database Management**]: Systems—*Textual Databases*

General Terms

Algorithms, Design, Performance

Keywords

Information extraction, text databases, extraction views

1. INTRODUCTION

Modern real-world applications often rely on large amounts of data that grow at rapid rates. While some of this data is stored in a structured form, unstructured text documents such as Web pages, email messages, news articles and reports contain rich information that can be very useful if extracted on time. The ability to answer structured SQL queries over this kind of unstructured data allows for more complex analysis and better insights into that data.

Example applications that benefit from structured queries over unstructured textual data include reputation management systems, which download Web pages to track the "buzz" around companies and products; comparative shopping agents, which locate e-commerce Web sites and index the products offered; and other information extraction applications, which retrieve documents and extract structured relations from the unstructured text. For example, in business intelligence, a company about to release a product may want to know whether similar products are well received by consumers, and find out the average price of such products. This can be achieved by extracting information from online shopping Web sites, and aggregating and users' ratings of similar products.

In the absence of automatic means of extracting such information, one way to complete such tasks is using keyword search to retrieve documents that are relevant to the query at hand, followed by manually identifying the relevant data in the returned documents. This can be an expensive process which fails to leverage complex extraction techniques to find semantically relevant documents that do not contain the exact text of the keyword query. The simplest automated approach is the *extract-transform-load* (ETL) approach, which means applying information extraction (IE) techniques on the text documents to extract the data and transform it into a structured format, then loading it into a data warehouse. Structured queries using SQL or similar languages can then be posed on the data warehouse whenever needed. This approach may be suitable for applications that do not require recent and up-to-date results. However, for frequently changing data, this snapshot will become stale quickly, and will not accurately reflect the original documents.

Just-in-time information extraction is an integration between information extraction and database engines to allow extracting only timely and relevant data within reasonable run-time constraints, by performing extraction as part of query processing rather than as a separate offline process. Variations of this approach include SQoUT [4, 5, 6], System-T [7], And XLog [8].

We present a lightweight implementation of just-in-time extraction that does not require fundamental modifications to the DBMS. Our framework integrates information extraction with traditional query processing through view matching techniques. We introduce *extraction views*, database views whose data is obtained by running information extractors on specific document collections, rather than by running SQL queries on relational data. Extraction views leverage the current view infrastructure available in most commercial DBMSs. They can be used in queries in the same way as traditional database views, which allows the DBMS to exploit well-developed query optimization opportunities that are applicable to relational data, including pushing down predicates, and using different access paths, join methods and join orders. These different optimizations, in addition to which extraction views to use for a given query constitute the plan space for that query. The query optimizer uses a cost model to determine the best plan to answer the query.

Our system demonstrates:

- a solution inspired by the data integration paradigm (specifically the *local-as-view* approach [3]); Extraction views encapsulate the IE tasks, and use extractors as black boxes with minimum exploitation of IE metadata.
- extending the traditional query optimizer, enabling it to explore the full optimization space for queries involving such views and choose the best plan for the defined cost model.

We propose to demonstrate a visual front-end to enable the user to design an application schema, and pose queries to the database. The interface presents the user with the query results and the execution plan chosen by the optimizer.

2. SYSTEM OVERVIEW

In this section we describe our integrated extraction and querying framework, in which information extraction is query-driven. Our

SIGMOD '12, May 20–24, 2012, Scottsdale, Arizona, USA.

Table 1: Example extractors

Extractor	Domain(s)
E_1	company
E_2	company, city
E_3	name
E_4	date
E_5	name, company, position
E_6	color
E_7	time
E_8	address, city, country
E_9	email

proposed framework supports SQL queries on both relational data and data extracted from text documents. Data is extracted during query processing, based on a schema that is defined on demand.

2.1 Attribute Domains and Lineage

Our framework assumes the existence of a set of extractors $\mathcal{E}$, which can be general purpose extractors such as UIMA [2] annotators or GATE [1] applications. Each extractor in $\mathcal{E}$ extracts data and returns it as objects with one or more attributes. All objects returned by a given extractor E have the same set of attributes, $attr(E)$. Hence, the output of each extractor can be viewed as a set of tuples sharing the same attributes. Given the set $\mathcal{E}$, the *domain universe* $\mathcal{D}$ is the set of all possible domains extracted by all extractors in $\mathcal{E}$, i.e. for $\mathcal{E} = \{E_1, ..., E_n\}, \mathcal{D} = attr(E_1) \cup ... \cup attr(E_n)$.

Example 1. Table 1 shows a group of 9 available extractors, and the domains that can be extracted by each. Based on these extractors, $\mathcal{D}$ = *{company, city, name, date, position, color, time, address, country, email}*

The set $\mathcal{D}$ represents the space of extractable information that is accessible to queries. The domain names in $\mathcal{D}$ are *unique*, i.e. there can be no two domains with the same name that extract different types of data. However, the same domain can be extracted by multiple extractors. The domain universe $\mathcal{D}$ can be expanded by adding new extractors that return different domains. These domains are independent of any particular queries or applications.

Given an extractor E, if the domains returned are $d_1, d_2, ..., d_m$, each returned tuple has the following:

- The values $v_1, v_2, ..., v_m$, where v_i belongs to the domain d_i; $1 \leq i \leq m$, and
- The *lineage* of each v_i. The lineage of an extracted attribute value is additional metadata that indicates where that value was extracted from.

The simplest and most common form of lineage information can be ⟨*docURL, span*⟩, where *docURL* is the URL of the document from which a particular attribute value is extracted, and *span* is the position of that value inside the document, usually defined as *(begin, end)*. We assume that all extractors expose this lineage information in addition to the actual extracted data. In our work, we found that many real-life extractors do in fact return this information as a minimum [1, 2, 7]. This lineage information is often useful in linking data produced by different extractors.

2.2 T-tables as Relational Wrappers

Our framework supports answering queries that reference both relational data and extracted data. The relational data is represented as traditional relational tables. A relational table consists of (a) an *intension*, which is the table definition (the set of attributes in the table, and the type of each attribute), and (b) an *extension*, the actual data in that table, stored as tuples. To represent the extracted data, we define a special type of tables, called *T-tables*. A T-table consists of only the table signature describing the attributes in that table and their domains. They provide a relational wrapper to the extracted data in order to be able to pose queries on that data. From the query point of view, a T-table is like a traditional relational table. The difference is the source of the data in each. In traditional tables, the data has to be loaded before any queries can be posed. This data is stored in a structured form (tuples), and read from disk during query execution. T-tables only have to be defined prior to issuing any queries, but no data is loaded into them. During query execution, data is directly extracted from text documents. Each T-table can only include data that is available for extraction, i.e. only include attributes that refer to the domains in $\mathcal{D}$. The T-table definition is in the form $T(a_1 : d_1, ..., a_n : d_n)$ where each a_i is an attribute name, and d_i is the domain to which a_i belongs.

Example 2. Consider the domain universe from Example 1, and suppose we need to pose queries related to companies and employees. For this particular application, we can define the following T-tables:

- *Comp (cname: company, addr: address, cty: city, cntry: country)*
- *Emp (ename: name, birthdate: date, job: position, ecomp: company, hiredate: date)*

Both the *cname* and *ecomp* attributes belong to the *company* domain in $\mathcal{D}$. Similarly, the *birthdate* and *hiredate* attributes belong to the *date* domain. Note that the T-table *Emp* cannot include a *salary* attribute for example, since we have no *money* domain that can be extracted. Defining the T-tables allows the user to pose queries such as:

```
SELECT job, min(hiredate) FROM Comp, Emp
WHERE cname = ecomp AND cntry = 'Italy'
```

2.3 Extraction Views and Joiners

Even though the definition of the T-table specifies the attribute domains, this definition does not specify where the data is obtained from (which document collection) or how to obtain it (which extractors to use). This is accomplished using *extraction views* and *joiners*. An extraction view v is defined as $v(T, A, C, E)$, where:

- T is a T-table on which v is defined
- A is the set of attributes in T that v covers
- C is the document collection that is mapped to v
- E is the extractor that will be used to extract the tuples of v from the documents in C

Extraction views are a way of packaging extractors into logical entities. The extraction view definition tells the system where and how to obtain the data. It establishes the mapping between the T-table attributes, the document collection, and the extractor. Different extraction views can cover different groups of attributes (possibly overlapping) in the same T-table, each using a different extractor and/or a different document collection. An extraction view $v(T, A, C, E)$ is valid only if (a) all the attributes in A must belong to the T-table T, and (b) the domains of the attributes in A (as specified in the definition of T) must be among those produced by the extractor E.

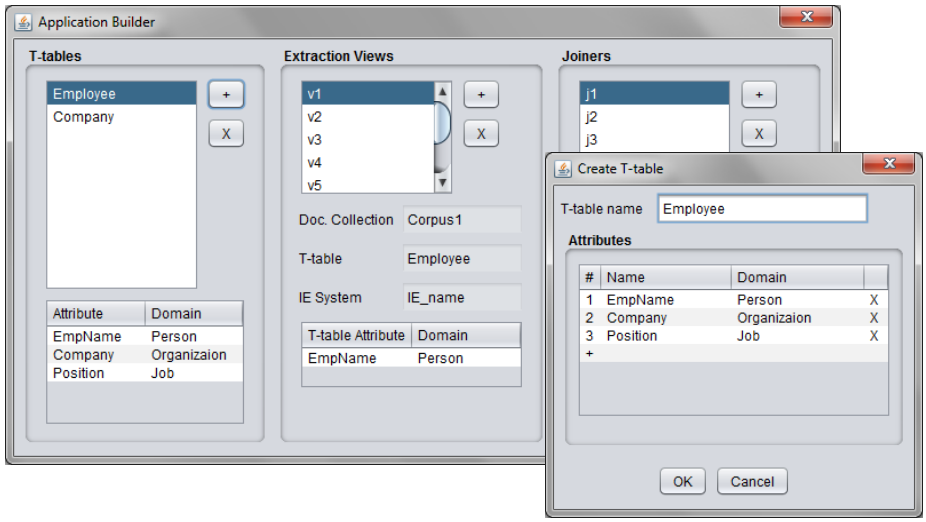

Figure 1: Application builder window, with a child window to create a new T-table

Example 3. Suppose that we have two document collections: C_1 contains company information, and C_2 contains employee information. Given the T-tables defined in Example 2, and given the extractors in Table 1, we can define the following extraction views (among others):

- $v_1(Comp, \{cname\}, C_1, E_1)$
- $v_2(Emp, \{ecomp\}, C_2, E_1)$
- $v_3(Emp, \{ename\}, C_2, E_3)$
- $v_4(Emp, \{birthdate\}, C_2, E_4)$
- $v_5(Emp\{ename, ecomp, job\}, C_2, E_5)$
- $v_6(Emp, \{hiredate\}, C_2, E_4)$
- $v_7(Comp, \{addr, cty, cntry\}, C_1, E_8)$

The definition of v_1 means that the attribute *cname* in T-table *Comp* can be obtained by invoking extractor E_1 on document collection C_1. Note that some views overlap, e.g. v_3 and v_5 have the *ename* attribute in common.

We often need to construct the tuples of a T-table from the tuples returned by multiple extraction views. In Example 3, v_1 and v_7 together cover all the attributes of the T-table *Comp*. However, with only the extraction views, it is not possible to determine which v_1 tuple belongs to which v_7 tuple. Thus a join condition is needed to determine which values actually belong to the same *Comp* entity. The concept of joining or linking the outputs of extractors based on positional information or other metadata already exists in some of today's publicly available IE frameworks. For example, UIMA [2] allows the definition of *aggregate analysis engines*, which operate on the outputs of extraction modules, potentially using extraction metadata to join them. Although in many cases, it is necessary to understand the semantics of the text to determine which values belong together, this can often be determined by analyzing the position of attribute values within the documents. This is especially true for documents that are semi-structured, e.g. Web pages displaying search results or containing HTML tables. This is where the lineage information produced by the extractors becomes useful. We exploit this information by defining *joiners*.A joiner j is defined as $j(T, A, C, P)$, where:

- T is a T-table on which j is defined
- A is the set of attributes in T that j covers
- C is the document collection on which j applies
- P is a predicate (or set of predicates) on the values and/or lineage of the attributes in A.

The joiner definition is specific for a particular document collection, since the document format in one collection can be different from another.

Example 4. Continuing our running example, we can define the joiner $j_1(Comp, \{cname, addr\}, C_1, P)$ where:
P = [cname.docURL = addr.docURL and addr.begin – cname.end < 30]

This joiner definition means that a tuple that contains a *cname* value (e.g. returned from v_1), and a tuple that contains an *addr* value (e.g. returned from v_7) belong to the same logical entity if the *addr* value is located less than 30 characters after the *cname* value within the same document. Note that in Example 3, some extraction views (e.g. v_4 and v_6) use the same extractor to extract two different attributes. On their own, these two views return the exact same date values. However, with the other views and with the proper joiners, these two views can contribute different information to queries.

3. DEMONSTRATION

The system is implemented by extending the data model and optimizer of Apache Derby to include the required data objects and algorithms. In addition to the core query engine, we developed a Java GUI with JDBC connection to demonstrate the system. We outline the different stages in the system's lifetime in Section 3.1, and then we describe the demonstration scenario in Section 3.2

3.1 System Lifetime

The various objects and components in our framework are defined and used at different points in time.

Extractor Registration Time

Registering an extractor tells our framework how to invoke the extractor and what kind of information it can extract. This updates the domain universe $\mathcal{D}$. Registering extractors is application-independent, and can be done at any time, causing the domain universe to expand as new kinds of extractable information become available.

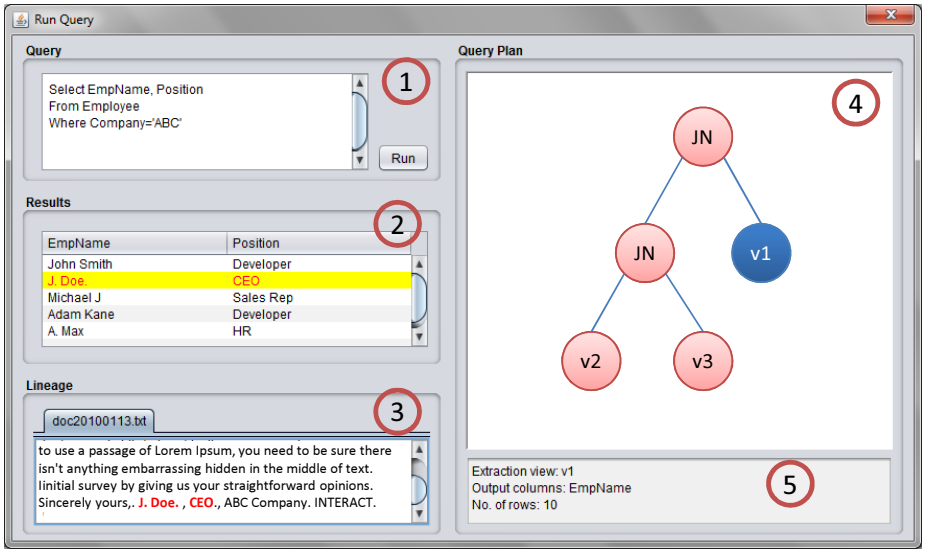

Figure 2: Query interface

Application Design Time

Given a particular application or query workload, this is where T-tables, extraction views, and joiners are defined, using the extractors and the attribute domains that are available to the framework. The application or workload cannot be started until this phase is completed.

Query Time

Queries are posed over the defined T-tables. During query optimization, the optimizer finds the best execution plan using a combination of the defined extraction views and joiners. During query execution, the extractors associated with the selected views are invoked, extracting information from the corresponding document collections and returning them as relational tuples.

3.2 Demonstration Scenario

Screenshots of our user interface are shown in Figures 1 and 2. The first step in our demo is application design, which takes place as follows:

1. The user launches the *application builder* (Figure 1), which has 3 areas, showing the existing T-tables, extraction views, and joiners respectively.
2. For each of the 3 object types, the user can add new objects by clicking the (+) button, which opens a new window to define the properties. Figure 1 shows the *Create T-table* window. Similar windows exist for extraction views and joiners.
3. In the application builder, selecting any of the existing objects displays information about that object at the bottom.

After designing the application, the user can start posing queries using the query interface (Figure 2), which is divided into query (area 1), results (area 2) , lineage (area 3), plan (area 4), and node information (area 5). We will demonstrate our system with various queries. The query scenario is as follows:

1. The user types the query in area 1, and clicks the "Run" button.
2. A table of the query results is shown in area 2.
3. The user can click on any tuple in the query results to see the lineage of this tuple (the original documents from which the data was extracted) in area 3. If the tuple is a result of joining multiple documents, then area 3 will have a separate tab for every document. The attribute values of the selected tuple are highlighted in the shown documents. In the shown example, both attributes of the selected tuple come from the same document, which is shown at the bottom, with the values highlighted in the text.
4. The execution plan chosen by the optimizer is displayed in area 4. This plan shows which extraction views are used, as well as the join types and join order. T-tables are not shown in the actual plan since they are only logical wrappers.
5. Clicking any node in the execution plan displays detailed information about that node in area 5, including the node's type (view, join, selection, etc.), the node's input and/or output attributes, among other information.

4. REFERENCES

[1] H. Cunningham, D. Maynard, K. Bontcheva, and V. Tablan. GATE: An Architecture for Development of Robust HLT Applications. In *ACL*, pages 168–175, 2002.

[2] D. Ferrucci and A. Lally. UIMA: An Architectural Approach to Unstructured Information Processing in the Corporate Research Environment. *Nat. Lang. Eng.*, 10:327–348, 2004.

[3] A. Y. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 10(4):270–294, 2001.

[4] A. Jain, A. Doan, and L. Gravano. Optimizing SQL Queries over Text Databases. In *ICDE*, pages 636–645, 2008.

[5] A. Jain, P. G. Ipeirotis, A. Doan, and L. Gravano. Join Optimization of Information Extraction Output: Quality Matters! In *ICDE*, pages 186–197, 2009.

[6] A. Jain, P. G. Ipeirotis, and L. Gravano. Building Query Optimizers for Information Extraction: The SQoUT Project. *SIGMOD Record*, 37(4):28–34, 2008.

[7] F. Reiss, S. Raghavan, R. Krishnamurthy, H. Zhu, and S. Vaithyanathan. An Algebraic Approach to Rule-Based Information Extraction. In *ICDE*, pages 933–942, 2008.

[8] W. Shen, A. Doan, J. F. Naughton, and R. Ramakrishnan. Declarative Information Extraction Using Datalog with Embedded Extraction Predicates. In *VLDB*, pages 1033–1044, 2007.