Gradation: A Pay-as-You-Go Style Hybrid Query Language for Structured and Text Data

Yui Yasunaga
University of Tsukuba
yui@slis.tsukuba.ac.jp

Atsuyuki Morishima
University of Tsukuba
mori@slis.tsukuba.ac.jp

Hiroki Sodeyama
University of Tsukuba
hiroki.sodeyama.2009b@mlab.info

Masateru Tadaishi
University of Tsukuba
tada@slis.tsukuba.ac.jp

Shigeo Sugimoto
University of Tsukuba
sugimoto@slis.tsukuba.ac.jp

Abstract

There is an increasing number of Web data which consist of text and structured data, such as the combination of Wikipedia pages and DBpedia data. To issue queries to such data, we must choose one of the followings: (1) submit keyword queries against textual data part, or (2) submit structured queries written in structured query languages like SPARQL, against structured data part. Keyword queries are easy for casual users to write, but they do not have expressive powers enough to fulfill the user's information needs. On the other hand, structured queries are more expressive than keyword queries, but are not easy for casual users to write. This paper proposes a hybrid query language that seamlessly integrates the two types of queries, allowing us to write queries in a "pay-as-you-go" fashion.

Keywords: query languages, web search, structured data

Introduction

There is an increasing number of Web data which consist of text and structured data. An example is the combination of Wikipedia pages and DBpedia data (http://www.dbpedia.org/). DBpedia is RDF data that describe information appearing in the Wikipedia articles in a structured way. Another example is the combination of the Bible Ontology (http://home.bibleontology.com/) and text pages explaining the people who appear in the Bible[1]. Given the current activities such as Linking Open Data community projects (http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData) at W3C, we expect that in the near future it is common that many textual data have structured data (annotation) associated to them.

To issue queries to such data, you need to choose one of these types of queries: *keyword queries* against text pages, or *structured queries*, like SPARQL, against structured data. Keyword queries including Boolean queries are so simple and widely used by casual users. However, they cannot express complex conditions. In contrast, structured queries can express complex conditions, but are difficult for many users to write since it is required for users to know the complex syntax and schema-level information for writing queries.

This paper proposes *Gradation query language* (shortly, Gradation), a hybrid query language for the Web data that consist of *text pages* (Web pages whose main components are text) and *structured data* (mainly RDF data in the paper). In short, Gradation supports keyword queries with *options* to specify conditions on the associated structured data, and allows a seamless integration between keyword and structured queries. The design was developed based on an interesting fact that casual users often use

[1] It is easy to associate the people that appear in texts with objects in the Bible Ontology.

Acknowledgements: The authors are grateful to Prof. Sakaguchi and Prof. Nagamori for the discussion in seminars. This research was partially supported by PRESTO from the Japan Science and Technology Agency.
Yasunaga, Y., Morishima, A., Sodeyama, H., Tadaishi, M., & Sugimoto, S. (2013). Gradation: A pay-as-you-go style hybrid query language for structured and text data. *iConference 2013 Proceedings* (pp. 209-222). doi:10.9776/13178
Copyright is held by the authors.

"search options" in Web search, such as "`site:edu`." In fact, a survey (internet.com K.K. (Japan), 2009) reported that about 25% of users of Web search engines have experience of using options.

The design of Gradation allows users to write not only both pure keyword and structured queries, but also *hybrid queries* that intermix conditions on text data and associated structured data. Thus, Gradation allows us to write queries in a "Pay-as-you-go" fashion, i.e., describe various types of queries according to the tradeoff between the acceptable cost of writing queries and the required preciseness of the query description.

Here is an example. When you search for text pages containing "Tom," the query is:

```
Tom
```

If the associated structured data contain the age of people, a query to get text pages containing "Tom" that correspond to those people who are over 40 is [2]:

```
Tom age>=40
```

As shown above, you can not only just submit simple keyword queries, but also add more complex conditions on structured data if required.

The contributions of the paper are as follows: First, we propose a unified query language that covers both simple keyword queries and complex structured queries. Next, we prove that the query language is relationally complete, because the relational completeness is a well-known criterion for discussing the expressive power of structured query languages. Finally, we show the experimental results to prove that that language allows us to write many hybrid queries in addition to pure keyword and relational queries.

In general, users need to know the schema-level information (e.g. attribute and class names) to write structured queries. An interesting application of our hybrid language is to help the user construct complex structured queries: The user first submits simple keyword queries and the system shows hints to transform the query into more precise ones. The detailed discussions on the application are beyond the scope of this paper and will appear in forthcoming papers.

The remainder of this paper is organized as follows: First, we describe related work. Second, we explain the processing model for Gradation query language. Third, we explain Gradation queries. Next, we give the formal semantics and prove that Gradation is relationally complete. Then, we show some experimental results to illustrate Gradation allows us to write many queries other than pure keyword queries and relational queries. Finally, we describe the conclusion.

Related Work

Since Gradation is a hybrid query language for text and associated structured data, there are a lot of related work and tools that address queries for text and structured data in different ways. They include: (1) natural language queries for structured data, (2) keyword queries for structured data, (3) structured query languages that have built-in keyword search mechanisms, (4) full-text search engines that allow users to specify conditions on structured metadata and (5) sophisticated search systems.

(1) Natural language queries. It would be ideal if users can write queries in natural languages. Therefore, there have been a lot of attempts (Androutsopoulos, Ritchie, & Thanisch, 1995). A typical approach is to translate natural language queries into structured ones. However, it is difficult for computers to perfectly understand what users intended with the queries, and this is one of the reasons why we have artificial and formal query languages including SQL, SPARQL, and Gradation. Essentially, processing natural language queries is to map such queries to well-defined formal abstractions. In that sense, researches on natural language query processing and formal query languages are complementary to each other.

(2) Keyword queries for structured data. There are many researches on keyword queries for structured data (Chen, Wang, Liu, & Lin, 2009). For example, SPARK (Luo, Wang, & Lin, 2008) is a keyword search engine against structured data, which allows users to mention attributes or table names in queries. SPARK is different from Gradation in that (1) it is a query language for structured data only, and (2) the relationships among tuples the queries can represent are limited thus SPARK is not relationally complete. Another approach is to try to find the semantics of keyword queries in the context of given structured data (Sarkas, Paparizos, & Tsaparas, 2010), in which they try to find which attribute corresponds to each given keyword. Again, Gradation is different from such researches, in that it does not try to find the intention of keyword queries, and is complementary to them.

(3) Structured query languages with keyword search mechanisms. XQuery and XPath Full-Text (Case et al., 2011) extends the syntax and semantics of XQuery and XPath to realize full-text search on XML. Many SQL databases support full-text search functions. Semplore is an IR-based system that has a query language for text and structured data. Because they are extensions of structured languages or a novel query language, users cannot write even simple queries without knowing the syntax. In contrast, Gradation is a natural extension of keyword queries, and allows us to write queries in a "Pay-as-you-go" fashion.

[2] You can quote character strings (attribute values, or keywords) to distinguish them from reserved words or attribute names if necessary.

(4) Full-text search engines allowing queries on structured metadata. An example is Lucene (http://lucene.apache.org/core/), a search engine given as a Java library to develop full-text search systems. Programmers, who use Lucene to implement a search engine, define queryable fields for given text data in advance, and then users of the search engine can use advanced search functions to specify the fields. Unlike Gradation, Lucene does not define how to write queries since it is not a query language. In addition, the functions for structured search given by Lucene are limited and do not provide the relational completeness.
(5) Sophisticated search systems. There is an increasing number of sophisticated search systems, such as faceted search systems (Tunkelang, 2009) and fielded search systems (e.g., LexisNexis (http://www.lexisnexis.com/en-us/)). Gradation is different from the systems in that it is a relationally complete language that seamlessly integrates keyword search and structured queries.

In addition, the ranking of query results is an important issue of in Text retrieval and Web search (Selvan, Sekar, & Dharshini, 2012). Gradation has a set-based semantics and is neutral on the ranking issue. The development of ranking schemes for Gradation is an interesting future work.

Query Processing Model

This section explains the query processing model for Gradation. We first define the data model of Gradation, and then, explain the input and output of the query processing model.

The Data Model

We define the data D as a triple: $D = (T, G, f)$. Figure 1 illustrates an example of D.

- T is a set of text pages. For example, $T = \{$Samurai_t, M:I_t, ToyStory_t, Ken_t, Tom_t, Johnny_t$\}$ is in Figure 1 [3].
- G is the RDF data associated to T. RDF (Resource Description Framework) (http://www.w3.org/RDF/) is a framework for describing metadata. RDF describes metadata in terms of a set of resources, a set of values, and the relationship among them. RDF data consists of a set of triples, each of which has the form of `(Subject, Predicate, Object)`, in which `Subject` is a resource, and `Object` is either a resource or a value. For example, given a resource Tom_r[4], assertions on Tom_r such as "Tom_r is 50 year-old" or "Tom_r belongs to Actor class" can be described using a set of triples: {(Tom_r, age, 50), (Tom_r, type[5], Actor)}. Given a set of RDF triples, we can express it as a labeled directed graph $G = (N, P, E)$ (Figure 1 (top)). Here, N is a set of nodes, P is a set of predicates, and E is a set of edges. Each edge in E is a triple (n_i, p, n_j) where $n_i, n_j \in N$ and $p \in P$. Triples of RDF data are mapped to edges, in which resources and values are mapped to nodes, and predicates are mapped to edge labels. In Figure 1, N = {Tom_r, 48, ToyStory_r, Film} and E = {(Tom_r, age, 48), (ToyStory_r, type, Film)}.
- $f : T \to N$ is an injective function that expresses the relationship between text pages in T and nodes of G. If $t \in T$ is associated to $n \in N$, the fact is represented by $f(t) = n$. For example, $f($Tom_t$) =$ Tom_r in Figure 1.

Inputs and Outputs

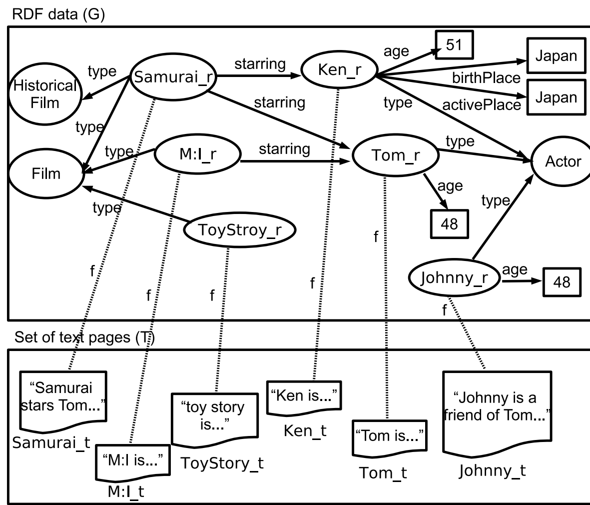
Figure 2 illustrates the query processing model for Gradation. The model consists of the following components: (1) D is the data explained in the previous section, (2) q is a submitted query. (3) R is the relation to represent the result of applying q to D.

The schema of relation R is determined by q. In the simplest case, R is a unary relation whose only attribute is "t_uri" which means the URI of the text pages. For example, if q is "`Tom`", R is a unary relation contains a set of URIs of text pages containing "Tom". Other cases will be explained in the next section.

Gradation Query Language

This section explains Gradation queries with examples. We start with simple queries that appear in casual searching and gradually show more precise ones. Therefore, this section illustrates the flexibility of the language. All of the examples take as input the data shown in Figure 1. The purpose of the section is to give intuitive explanation. The formal semantics are given in the next section, and the detailed syntax will be shown in the Appendix.

[3] Here, strings ending with "_t" denote URIs of text pages.
[4] Here, strings ending with "_r" denote URIs of resource nodes.
[5] "type" will be used to refer to "rdf:type".



Strings ending with ``_r'' are URIs of resource nodes, and strings ending with ``_t'' are URIs of text pages.

Figure 1. Example of D

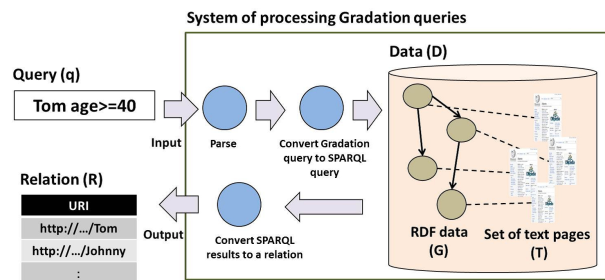


Figure 2. Query Processing Model of Gradation

A *query* consists of *keywords* and *non-keyword components* (Table 1). Non-keyword components are used to write more complex queries that require path expressions, class specifications, and arithmetic comparisons. Gradation allows us to write queries with various combinations of keywords and non-keywords components, covering both of simple keyword queries and complex structured ones. The user can choose appropriate combinations according to the tradeoffs between the acceptable cost of writing queries and the required preciseness of query description. In the followings, we first explain queries consisting only of keywords. Then, we explain queries having non-keyword components.

Queries with One Keyword

The section explains queries consisting of only one keyword, and their semantics. In Gradation, a query to get the text pages containing "Tom" can be written as:

`Tom` (Q1)

As another example, a query to get the text pages containing "Samurai" is:

`Samurai` (Q2)

The results of the queries Q1 and Q2 are shown in Figures 3 and 4, respectively.

In general, if a query consists of one keyword k, the query processor performs a Boolean full-text search against T with k, and the result is a set of URIs of text pages containing k. Formally, the result is $\{t | t \in T, match(t, keyword)\}$.

Gradation supports the wildcard ("$*$") as a special type of keyword. The result of the query consisting only of "$*$" is T, namely, the set of all the text pages. The wildcard can be used at the place in which predicates or string values are required (i.e., at places in which class names, attribute names, or string values are required).

Intersection, Union and Difference

In order to make a query to produce the intersection, union, and difference of the results of subqueries (Lines 1-3 of Table 1), users use non-keyword components `and`, `or`, and `-`, respectively. For example, a query to get the text pages containing "Tom" and "Samurai" is:

`Tom and Samurai` (Q3)

The result of query Q3 is shown in Figure 5. This is the intersection of the results of Q1 and Q2 (see Figures 3 and 4). In Gradation, the intersection is the default interpretation to connect subqueries. Therefore, `and` can be omitted and the query "`Tom Samurai`" is interpreted as "`Tom and Samurai`".

Table 1
List of Non-keyword Components of Queries

Query with a non-keyword component	Explanation
q_1 `and` q_2	**Intersection.** Returns the intersection of the results of queries q_1 and q_2.
q_1 `or` q_2	**Union.** Returns the union of the results of queries q_1 and q_2.
q_1 `-` q_2	**Difference.** Returns the set of tuples that appear in the result of q_1 and do not appear in that of q_2. If q_1 is omitted, "*" is used for q_1 (Here, "*" is a wildcard for keywords. See Section "Queries with One Keyword").
`(` $q_1 \ldots q_i$ `)` $q_{i+1} \ldots q_n$	**Grouping.** $q_1 \ldots q_i$ are evaluated before the others.
`class:`c	**Class Instances.** Here, c is a class name in RDF data. The query returns the set of instances of class c, namely the set of resourse nodes have `rdf:type` predicate connected to c or to a class reachable from c through subclass properties.
q `[`$attribute_1$`, ...,` $attribute_n$`]`	**Make Relational Attributes Appear in the Result (Projection).** Here, $attribute_i$ is one of the followings: (1) `t_uri`, which means the URI of text pages, (2) `r_uri`, which means the URI of resource nodes, and (3) a predicate that appears in the given RDF data.
$q_1 . p . q_2$	**Path Traversal.** Let N_i be a set of RDF graph nodes corresponding to the result of q_i. The query returns the set of (t_1, t_2), in which $(n_1, p, n_2) \in E$, $n_i \in N_i$, $n_i = f(t_i)$ and $t_i \in T$.
$attribute_1\ \theta\ value$ or $attribute_1\ \theta\ attribute_2$	**Attribute-based Selection.** This component returns the set of URIs of text pages each of which is connected to RDF graph node s satisfying one of the following conditions: (1) There exist $(s, attribute_1, value) \in E$ s.t. $attribute_1 \theta value$ holds. (2) There exist $(s, attribute_1, o_1), (s, attribute_2, o_2) \in E$, s.t. $o_1 \theta o_2$ holds. For θ, we can use >, <, >=, <=, =, ==, or !=.
q_1 `*` q_2	**Cartesian Product.** The query returns the Cartesian Product of q_1 and q_2 (Note that each query returns a relation in the Gradation query processing model.)
q_1 `as` $\$a$ $q_2 \ldots q_n$	**Alias.** In $q_2 \ldots q_n$, $\$a$ can be used as the name to refer to the result of q_1.

t_uri
Samurai_t
Tom_t
Johnny_t

Figure 3. Output of Q1

t_uri
Samurai_t

Figure 4. Output of Q2

t_uri
Samurai_t

Figure 5. Output of Q3

Grouping of Subqueries

Users can use `(...)` to group subqueries (Line 4 of Table 1), to affect the evaluation order of subqueries. For example, the following queries have different evaluation orders of subqueries.

`(Tom or Ken) Scientist` (Q4)

`Tom or (Ken Scientist)` (Q5)

Here, Q4 returns the intersection of the set of text pages containing "Tom" or "Ken," and the set of text pages containing "Scientist." On the other hand, Q5 returns the union of the set of text pages containing "Tom," and the set of text pages containing "Ken" and "Scientist."

Restricting to Instances of a Particular Class

Gradation has a non-keyword component `class:`c in order to restrict the query results to instances of a particular class c (Line 5 of Table 1). For example, a query to get the set of text pages corresponding to `Tom` who is an actor is:

`Tom class:Actor` (Q6)

t_uri
Tom_t
Johnny_t

Figure 6. Output of Q6

t_uri
Tom_t
Ken_t
Johnny_t

Figure 7. Output of `class:Actor`

t_uri	age
Tom_t	48
Ken_t	51
Johnny_t	48

Figure 8. Output of Q7

t_uri	age
Tom_t	48
Ken_t	51
Johnny_t	48

Figure 9. Output of Q8

t_uri
Tom_t
Ken_t
Johnny_t

Figure 10. Output of Q9

age
48
51
48

Figure 11. Output of Q10

The result of Q6 is shown in Figure 6. The semantics of the query is the intersection of the results of subqueries `Tom` and `class:Actor` (Figure 3 and 7). In general, the result of `class:`c is the set of text pages corresponding to resource nodes which are instances of c. Here, we say a resource node is an instance of c if the node has `rdf:type` predicate connected to c or to a class reachable from c through subclass properties in the RDF data G. Therefore the subquery `class:Actor` returns the set of URIs of text pages that corresponds to RDF resource nodes having `rdf:type` predicate connected to `Actor` class or to a class reachable from `Actor` class through subclass properties in the RDF data G.

Specifying Relational Attributes to Appear in the Output

Each Gradation query can have a sequence of attribute names following the query (i.e., q [$attribute_1$, ..., $attribute_n$]) in order to provide each relational attribute with a tag "to appear" meaning that the attribute appears in the final result (Line 6 of Table 1). When the attributes correspond to RDF predicates, the attributes will contain values in the given RDF data G. For example, a query to return a set of binary tuples with (1) URIs of text pages that contains 'Tom' corresponding to resources who are of actor type, and (2) the values of their ages is:

`(Tom class:Actor)[t_uri, age]` (Q7)

Here, `t_uri` is the attribute to contain URIs of test pages (explained next). The result of Q7 is shown in Figure 8. The query returns a relation which has two relational attributes, `t_uri` and `age`.

Here, $attribute_i$ can be one of the followings:

1. `t_uri`: URIs of text pages that are contained in the result of q.
2. `r_uri`: URIs of resource nodes corresponding to the values of `t_uri` (i.e., r_uri = f(t_uri)).
3. Predicate p (e.g., `age`), that is connected to the RDF nodes in `r_uri`. If the RDF nodes do not have p, the values of the relational attribute of the final result will be null.

There are two points to note. First, `t_uri`s are tagged as "to appear" by default (i.e., query q without [$attribute_1$, ..., $attribute_n$] is interpreted as q`[t_uri]` by default). For example, query "`Tom`" is interpreted as "`Tom [t_uri]`." Second, the expression can be nested in queries and the tags are overwritten by the outer ones (i.e., if an attribute that is given "to appear" tag by an inner one is not specified in the outer one, the tag is removed from the attribute). For example, the following nested queries (Q8-Q10) return the results in Figures 9-11.

`((Tom class:Actor)[t_uri, age])[t_uri, age]` (Q8)

`((Tom class:Actor)[t_uri, age])[t_uri]` (Q9)

`((Tom class:Actor)[t_uri, age])[age]` (Q10)

Traversing Paths in the RDF Graph

Gradation allows users to traverse the paths in the given RDF data (Line 7 of Table 1). For example, a query to get the set of pairs (binary tuples) of text pages, in which each tuple has a text page corresponding to a film and a text page corresponding to an actor who stars in the film, is:

`class:Film.starring.class:Actor` (Q11)

t_uri(Film)	t_uri(Actor)
M:I_t	Tom_t
Samurai_t	Tom_t
Samurai_t	Ken_t

Figure 12. Output of Q11 and Q12

t_uri(Film)	t_uri(Actor)
M:I_t	Tom_t
Samurai_t	Tom_t

Figure 13. Output of Q13

The result of Q11 is shown in Figure 12 [6].

In general, query $q_1 . p . q_2$ returns a set of binary tuples, each of which represents a one-length path to traverse a link to connect the results of queries q_1 and q_2. In other words, let T_i be the set of `t_uri` values of the result of q_i, and let $N_i = \{n | n = f(t), t \in T_i\}$. Then, the result of $q_1 . p . q_2$ is a set of binary tuples $\{(t_1, t_2) | \forall i \in \{1, 2\} (n_i \in N_i, n_i = f(t_i)), (n_1, p, n_2) \in E\}$. Since the default attribute for the result is `t_uri`, the query $q_1 . p . q_2$ is interpreted as $q_1 . p . q_2$`[t_uri(`q_1`), t_uri(`q_2`)]` if not explicitly specified.

For example, consider query Q11 and assume that N_1 = {M:I_r, Samurai_r, ToyStory_r} and N_2 = {Tom_r, Ken_r, Johnny_r}, are the set of instances of Film class and that of Actor class, respectively. Then, the result of Q11 is {(M:I_t, Tom_t), (Samurai_t, Tom_t), (Samurai_t, Ken_t)}.

As a natural extension, users can write arbitrary length of path traversal $q_1 . p_1 . q_2 . p_2 . q_3 . \ldots . p_{n-1} . q_n$. In this case, let N_i be the set of RDF nodes that are `r_uri` values for the result of q_i. Then, the result of the query is a set of m-ary tuples $\{(t_1, t_2, \ldots, t_m) | \forall 1 \leq i \leq m (n_i = f(t_i)), \forall 1 \leq i < m (n_i \in N_i, (n_i, p_i, n_{i+1}) \in E)\}$

Combining Keyword Queries with Path Traversals

This section shows an example of a query having a keyword and a path traversal:

```
Tom class:Film.starring.class:Actor
```
(Q12)

Note that the query is an example of a hybrid query (neither a pure keyword nor a pure relational query). A subtle point here is that the results of `Tom` and `class:Film.starring.class:Actor` are not union compatible, because the former returns a unary relation and the latter returns a binary relation. Therefore, we cannot compute the intersection of the two results. However, as will be explained in a later section (Resolution Rules for Union Incompatibility), Gradation has a resolution rule for the case in which one subquery is a keyword query. Let R_1 is the result of a keyword subquery (with a keyword k, such as `Tom`) and R_2 is the non-unary relation computed by the other query q_2. In this case, the result of k `and` q_2 contains a non-unary tuple $t \in R_2$ if a text page addressed by one of the `t_uri` attribute values of t contains k.

For example, let R_{path} be the result of the second subquery of Q12, and assume that R_{path} ={(M:I_t, Tom_t), (Samurai_t, Tom_t), (Samurai_t, Ken_t)}. Then, since every tuple in R_{path} has at least one URI of text pages containing "Tom," the result of Q12 is the same as the result of Q11 (shown in Figure 12).

Applying Grouping in Various Types of Queries

Users can apply grouping (explained in an earlier section (Grouping of Subqueries)) to various types of queries including non-keyword queries. For example, a query to get the set of pairs of text pages, in which each binary tuple has (1) a text page containing "Tom" that corresponds to an actor and (2) a text page for a film in which he stars in is:

```
class:Film.starring.(Tom class:Actor)
```
(Q13)

The result of Q13 is shown in Figure 13. As with the case of keyword queries, the subqueries in `()` is evaluated before the other subqueries. In Q13, the subquery `(Tom class:Actor)` is evaluated first and returns the set of URIs of text pages having "Tom" and corresponding to the resources are actor type. Therefore, the result of Q13 (shown in Figure 13) is different from that of Q12 (shown in Figure 12), and does not have the tuple (Samurai_t, Ken_t).

Selection Based on Attribute-Values

The following is an example of a query to get the URIs of text pages corresponding to actors who are over 50:

```
class:Actor age>=50
```
(Q14)

[6] If at the output relation has different attributes with the same attribute names, the names would be changed appropriately.

t_uri
Ken_t

Figure 14. Output of Q14

t_uri(Film)	t_uri(Actor1)	t_uri(Actor2)
Samurai_t	Tom_t	Ken_t

Figure 15. Output of Q15

t_uri(Film)	t_uri(Actor)
Samurai_t	Ken_t

Figure 16. Output of `$a.starring.(class:Actor Ken)`

The result of Q14 is shown in Figure 14. Attribute-based selection (Line 8 of Table 1) can be specified as (1) $attribute_1\ \theta\ value$ or (2) $attribute_1\ \theta\ attribute_2$ where $value$ is a constant. Here, the former is the comparison between values of two attributes and the latter is the comparison between values of an attribute and a constant. The output of $attribute_1\ \theta\ value$ is the relation that contains URIs of text pages each of which is connected to RDF graph node s s.t. $(s, attribute_1, value) \in E$ (See an earlier selection (The Data Model)) and $attribute_1 \theta value$ holds (for example, when q is "`age>=50`," s = Ken_r). Similarly, the output of $attribute_1\ \theta\ attribute_2$ is the relation that contains URIs of text pages each of which is connected to RDF graph node s s.t. $(s, attribute_1, o_1), (s, attribute_2, o_2) \in E$, and $o_1 \theta o_2$ holds (For example when q is "`birthPlace==activePlace`," s can be Ken_r).

Alias

Gradation allows users to give alias names to subqueries (that returns relations) so that they can write complex queries, such as queries having the same path traversal multiple times (Line 9 of Table 1). For example, assume we want to know the films such that (1) two actors star in the same film, (2) one of them has the text page that contains "Tom" and (3) the other has the text page that contains "Ken." Then, the query is:

```
(class:Film as $a).starring.(class:Actor Tom) $a.starring.(class:Actor Ken)      (Q15)
```

The result of Q15 is shown in Figure 15. It has three attributes: (1) the URIs of text pages containing "Tom" and corresponding to RDF nodes of actor type, (2) the URIs of text pages containing "Ken" and corresponding to RDF nodes of actor type, and (3) the URIs of text pages corresponding to films they star in, because the `t_uri`s are the default attributes to appear in the result (See an earlier section (Specifying Relational Attributes to Appear in the Output))

There is one point that users have to be careful in using aliases. As explained in an earlier section (Intersection, Union and Difference), the default interpretation of a sequence of subqueries $(q_1\ \ q_2\ \ \ldots)$ is the intersection of the subquery results. However, for subqueries that are related to each other with the same alias, the interpretation is the *join* of the subqueries. For example, the result of Q15 is the equi-join on `$a` of the results of `(class:Film as $a).starring.(class:Actor Tom)` (the same as the result of Q10 (Figure 13)) and `$a.starring.(class:Actor Ken)` (Figure 16). The attributes of the result relation of Q15 are `t_uri`s of the two subqueries.

Formal Semantics and Relational Completeness

This section defines the formal semantics of Gradation Language. Then, we show that Gradation is relationally complete. Actually, the expressive power of Gradation is equivalent to the relational algebra with text containment predicate, which means that the queries can be translated into relationally complete structured query languages with text containment predicates (such as SPARQL).

This section is organized as follows: First, we map the data D (explained in The Data Model) to an equivalent universal relation. Second, we give the formal semantics for queries that we explained (see Gradation Query Language) only intuitively. Finally, we show that Gradation is relationally complete in the sense that it can express any relational queries on the universal relation.

Mapping Text and RDF Data to a Relation

This section explains how to map D (text and RDF data) to a universal relation U that preserves the information in D. For the explanation, we use the data shown in Figures 17 and 18.

The data in Figure 17 have two text pages Samurai_t and Ken_t. Figure 19 shows the result of mapping it to U. Let $D = (T, G, f)$. Then, U contains one tuple for each resource node n_t in G. Attributes of U are `r_uri`, `t_uri`, `text`, and attributes corresponding to all of the predicates that appear in G. (e.g.,

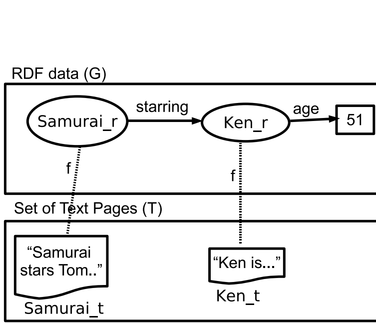


Figure 17. A part of Figure 1

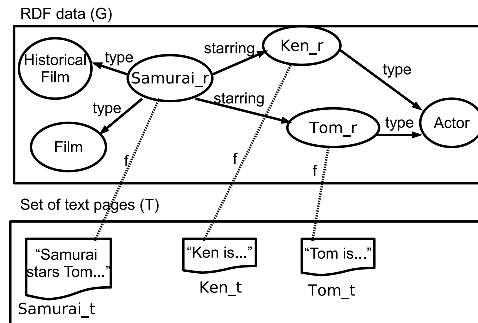


Figure 18. Another part of Figure 1

r_uri	text	t_uri	starring	age
Samurai_r	"Samurai stars Tom..."	Samurai_t	Ken_r	NULL
Ken_r	"Ken is..."	Ken_t	NULL	51

Figure 19. Universal relation U for the data in Figure 17

"starring" and "age"). Each tuple contains values of the `r_uri` of n_t, `t_uri` of t s.t. $n_t = f(t)$, and attribute values each of which is o s.t. $(n_t, p, o) \in G$. The values of attributes that do not correspond to predicates on n_t are null.

In general, a resource node can have two or more values for one predicate. For example, in Figure 18, Samurai_r has two edges for each of "starring" and "type" predicates. Figure 20 shows the result of mapping it to U. When n_t has $m > 1$ values for the same predicate p, U contains m tuples that are the same as each other except that attribute values for p are different. This recursively applies to the general case in which two or more such predicates exist. For example, U has four tuples for Samurai_r (Figure 20), which are generated by combining {Ken_r, Tom_r} for "starring" predicate and {Film, HistoricalFilm} for "type" predicate.

Constructing the Universal Relation. This section explains the algorithm for translating D into U. The algorithm works with the simple case in which each RDF node has only one value for each predicate. The algorithm for the general case in which nodes have two or more values for the same predicate is a natural extension of the algorithm and thus omitted. The algorithm (Figure 21) works as follows. Given D, it generates a $tuple$ for each resource node $n_t \in N$ (Lines 2-3), stores attribute values in each $tuple$ (Line 4-13), and inserts the $tuple$ into U (Line 14). Here, $node_uri(n_t)$ in Figure 21 is a function to get the URI of resource node n_t. Similarly, $textpage_uri(n_t)$ is a reverse function of f to get the URI of text page corresponding to n_t (i.e., $n_t = f(textpage_uri(n_t))$), and $text(u)$ is a function to get the contents of the text page of URI u.

Theorem 1 *The algorithm in Figure 21 outputs* U *that preserves the information stored in* $D = (T, G, f)$. □

Proof. To prove the theorem, we have to show U preserves information stored in G, T, and f, respectively. First, we show that U preserves the information in G. Lines 10-13 guarantee that for every $n_t \in N$, there is a tuple in U that has information on every triple (n_t, p, o) in G. Each tuple in U contains $node_uri(n_t)$ in the attribute r_uri and contains o in the attribute p.

Next, we show that U preserves the information stored in function $f : T \rightarrow N$. Remember that f is a one-to-one mapping between text pages and RDF resource nodes and that for every text page there is one RDF node in N. Given the fact, Line 8 guarantees that for every n_t, there is a tuple that has URI of the corresponding text page (if any). Since f is an injective function and the URI of every n_t is stored in U, it preserves the information encoded in function f.

Finally, we show that U preserves the information in text pages in T. Line 9 guarantees that each tuple in U stores not only the URI of a text page but the contents of it, which means that U preserves the information stored in the contents of text pages in T. □

Formal Semantics

This section gives the formal semantics of queries that we explained only intuitively in an earlier section (Gradation Query Language). First, we give the formal semantics of Gradation queries in terms of relational algebra expressions on the universal relation U. Second, we explain the formal semantics for subtle cases in which the relations are not union compatible in Intersection, Difference or Union operations.

Semantics of Gradation Queries. Table 2 summarizes the semantics of Gradation queries in terms of relational algebra expressions on the universal relation U we explained in an earlier section (Mapping Text

r_uri	text	t_uri	starring	type
Samurai_r	"Samurai stars Tom..."	Samurai_t	Ken_r	HistoricalFilm
Samurai_r	"Samurai stars Tom..."	Samurai_t	Ken_r	Film
Samurai_r	"Samurai stars Tom..."	Samurai_t	Tom_r	HistoricalFilm
Samurai_r	"Samurai stars Tom..."	Samurai_t	Tom_r	Film
Ken_r	"Ken is..."	Ken_t	NULL	Actor
Tom_r	"Tom is..."	Tom_t	NULL	Actor

Figure 20. Universal relation U for the data in Figure 18

INPUT: Data $D = (T, G, f)$, where $G = (N, P, E)$
OUTPUT: Universal Relation U
1:$U = \phi$
2:**FOREACH** $n_t \in N$
3: Tuple $tuple$ = new Tuple();
4: **FOREACH** $A_i \in U's\ Attributes$
5: $tuple[A_i]$= NULL;
6: **ENDFOR**
7: $tuple$[r_uri] = $node_uri(n_t)$;
8: $tuple$[t_uri] = $textpage_uri(n_t)$;
9: $tuple$[text] = $text(textpage_uri(n_t))$;
10: **FOREACH** $(n_t, el, v2) \in E$
11: **IF** ($v2$ has URI) $tuple[el]$ = $node_uri(v2)$;
12: **ELSE** $tuple[el]$ = $v2$;
13: **ENDFOR**
14: $U.insert(t)$;
15:**ENDFOR**
16:**RETURN** U;

Figure 21. Algorithm to create the relation U

and RDF Data to a Relation). In Table 2, q is a Gradation query and $sem(q)$ is a relational algebra expression that is equivalent to q (modulo the relational attributes that appear in the output). U is the universal relation, and $R.a$ is the attribute a of relation R.

We use the table to define the semantics of q as follows: Given q, the semantics of q is defined as $\pi_{attrs}(sem(q))$, in which $sem(q)$ is given in Table 2, and $attrs$ are the set of attributes that are tagged as "to appear" by $q_1[attributte_1, \ldots, attribute_n]$ in q. Therefore, we first construct a relation $R = sem(q)$ with Table 2. During the process, we compute $attrs$ when we encounter $q_1[attributte_1, \ldots, attribute_n]$ in q. As explained in an earlier section (Specifying Relational Attributes to Appear in the Output), the expression can be nested and $attrs$ will be overwritten by the outer one. Finally, we project out the attributes that are not contained in $attrs$ from R.

Resolution Rules for Union Incompatibility. This section explains the resolution rules for union incompatibility in Intersection, Difference and Union operations. An informal explanation was already given in an earlier section (Combining Keyword Queries with Path Traversals).

Union. When two relations for the union are not union compatible, the union operation returns the outer union (Navathe & Elmasri, 2002) of the two relations.

Intersection. In general, the two relations $sem(q_1)$ and $sem(q_2)$ for the intersection must be union compatible. However, Gradation defines the result of the intersection if one of q_is is a keyword query k (say, $q_1 = k$). In that case, we interpret the k and q_2 as selecting tuples in $sem(q_2)$ that contains k in any of `text` attributes of $sem(q_2)$. Formally, the result of query "k `and` q_2" is $R = \sigma_{text_1\ contains\ k \vee \ldots \vee text_n\ contains\ k}(sem(q_2))$. Here, $text_i$ is a `text` attribute of $sem(q_2)$. Note that the intersection is commutative thus "q_2 `and` k" returns the same result of "k `and` q_2."

Difference. Similarly, the two relations $sem(q_1)$ and $sem(q_2)$ for the difference must be union compatible. However, Gradation defines the result of the difference if one of q_i is a keyword query. Formally, the semantics of query "$k\ -\ q_2$" is $\sigma_{text_1\ not\ contains\ k \wedge \ldots \wedge text_n\ not\ contains\ k}(R_k)$. Here, k is a keyword, and $text_i$ is a `text` attribute of $sem(R_k)$. The semantics of query $q_1 - k$ is $\sigma_{text_1\ not\ contains\ k \wedge \ldots \wedge text_n\ not\ contains\ k}(R_1)$.

Table 2
Formal Semantics of Gradation Queries

Query q	Description	$Sem(q)$
$keyword$	**Keyword Query.**	$\sigma_{t_uri\ contains\ keyword}(U)$
q_1 `and` q_2	**Intersection.**	$sem(q_1) \cap sem(q_2)$
q_1 `or` q_2	**Union.**	$sem(q_1) \cup sem(q_2)$
q_1 `-` q_2	**Difference.**	$sem(q_1) - sem(q_2)$. If q_1 is omitted, $sem(q_1) = U$.
`class:`c_1	**Class Instances.**	$\sigma_{type=c_1 \vee \ldots \vee type=c_n}(U)$ where $c2, \ldots, c_n$ are the types reachable from c_1 through subclass properties.
$q_1 . p_1 . q_2 . p_2 . \ldots . p_{n-1} . q_n$	**Path Traversal.**	Let $R_i = sem(q_i)$ and assume that each q_i has only one t_uri attribute with "appear tag." Then, $sem(q) = (((R_1 \bowtie_{R_1.p_1=R_2.r_uri} R_2) \bowtie_{R_2.p_2=R_3.r_uri} R_3) \ldots) \bowtie_{R_{n-1}.p_{n-1}=R_n.r_uri} R_n$.
$attribute_1\ \theta\ value$ or $attribute_1\ \theta\ attribute_2$	**Attribute-based Selection.**	$R = \sigma_{attribute_1\ \theta\ value}(U)$ or $\sigma_{attribute_1\ \theta\ attribute_1}(U)$. For θ, we can use >, <, >=, <=, =, ==, or !=.
q_1 `*` q_2	**Cartesian Product.**	$sem(q_1) \times sem(q_2)$
q_1`[`$attribute_1$`, ...,` $attribute_n$`]`	**Attributes to appear in the output**	$attributes_i$s are tagged as those to appear in the final output

Operations	Relational Algebra	Gradation
Union	$R \cup S$	$R\ or\ S$
Difference	$R - S$	$R - S$
Cartesian Product	$T \times V$	$T * V$
Projection	$\pi_{A_1,A_2,\ldots,A_n}(R)$	$R[A_1, A_2, \ldots, A_n]$
Selection	$\sigma_{A_i\ \theta\ c}(R)$ or $\sigma_{A_i\ \theta\ A_j}(R)$	$R\ \ A_i \theta c$ or $R\ \ A_i \theta A_j$

Figure 22. Gradation queries that are equivalent to five relational operations

Relational Completeness

This section proves the following theorem.

Theorem 2 *Gradation is relationally complete on the universal relation U shown in an earlier section (Mapping Text and RDF Data to a Relation).* □

Proof. We show that Gradation can express queries that are equivalent to five relational algebra operations: Union, Difference, Cartesian product, Projection and Selection on U. In the discussion, we assume that:

1. Relation R, S, T and V are any relations that can be derived by applying relational algebra expressions to U,
2. R and S are union compatible,
3. The schema of R is $R(A_1, A_2, \ldots, A_i, \ldots, A_j, \ldots, A_n)$,
4. θ is one of $<, \leq, >, \geq, =, \neq$, and
5. c is a constant.

Then, Figure 22 shows how the five operations can be expressed in Gradation. □

Evaluation

As shown in an earlier section (Formal Semantics and Relational Completeness), the expressive power of Gradation is equivalent to the relational algebra with text containment predicate. The unique point of Gradation is its syntax - some queries can be written as pure keyword queries, some queries can be

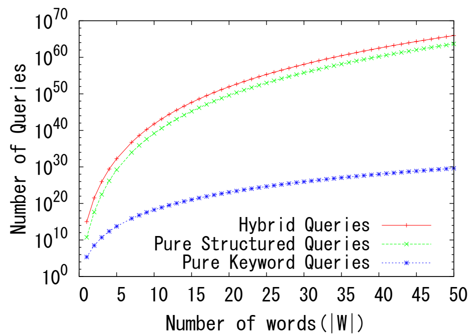


Figure 23. Experimental Result 1 ($height = 8$)

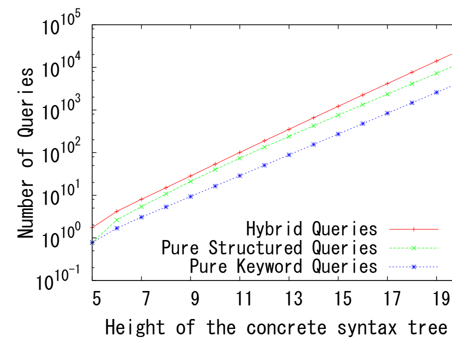


Figure 24. Experimental Result 2 ($|W| = 1$)

written by combining keywords and non-keyword components, and others can be written as pure structured queries, allowing us to write queries in a pay-as-you-go fashion. In order to confirm that Gradation allows us to write many keyword queries and hybrid queries other than pure structured queries, we counted the number of each type of queries that can be written in Gradation.

Query Styles

We define three query styles to which each Gradation query expression belongs. Note that only the syntax matters to define the styles. It is possible that query expressions in different styles are equivalent in their results.

- **Pure Keyword-query Style**: A query in this style has an appearance of Boolean keyword queries. An example is `(Tom or Thomas) and actor`.
- **Pure Structured-query Style**: A query in this style is a direct translation of a relational algebra expression into Gradation. An example is `Class:Actor[name]`, which is a direct translation of $\pi_{name}(\sigma_{Type=\text{``}Actor\text{''}}(U))$.
- **Hybrid Query Style**: A query in this style can be classified into neither pure keyword nor structured-query styles. An example is `Tom class:Actor`

Experiment

The purpose of the experiment is to confirm that Gradation allows us to write query expressions in various styles so that we can write queries in the pay-as-you-go fashion. In the experiment, we counted the number of Gradation query expressions in each of the three query styles. As explained in Introduction, support to change queries into more precise ones is one of our future work.

In the experiment, if we don't limit the number of words that appear in a query and the height of the concrete syntax trees of queries, it is obvious that every query style has a countable infinite number of queries and the comparison is meaningless. Therefore, we parameterize the number of words ($|W|$) and the height of the concrete syntax tree ($height$) in counting the number of queries. Here, W is a set of words used in queries. In the experiment, we assume that any $w \in W$ can appear at the places in which keywords, class names, attribute names, or string values can appear in a query.

We compared the numbers of queries in each of the three styles, varying the values of $height$ and $|W|$.

Results

Figure 23 shows the number of possible query expressions in each query style in the case of $height = 8$ and $1 \leq |W| \leq 50$. Figure 24 shows the number of possible query expressions in the case of $|W| = 1$ and $5 \leq height \leq 20$. The figures show that Gradation allows us to write many hybrid queries in addition to pure keyword and structured ones (Note that the Y-axis is in log scale).

Conclusion

This paper proposes Gradation, a hybrid query language for text and associated structured data that covers both keyword queries and structured queries. Users can add various query components to keywords

to constitute various kinds of queries in Gradation. We proved that Gradation is relationally complete and showed that Gradation allows us to write many hybrid queries in addition to pure keyword and structured queries. Therefore, Gradation allows us to write queries in a "Pay-as-you-go" fashion, i.e., describe various types of queries, according to the tradeoff between the acceptable cost of writing queries and the required preciseness of the query description.

Future work includes the development of efficient processing schemes for Gradation queries, the development of a scheme for ranking query results, and the development of a support system to show the user hints so that she can rewrite her imprecise queries (typically keyword queries) to more precise, structured ones.

References

Androutsopoulos, I., Ritchie, G. D., & Thanisch, P. (1995). Natural Language Interfaces to Databases - An Introduction. *Journal of Natural Language Engineering*, *1*, 29–81.

Case, P., Dyck, M., Holstege, M., Amer-Yahia, S., Botev, C., Buxton, S., ... Shanmugasundaram, J. (2011). XQuery and XPath Full Text 1.0 W3C Recommendation (REC-xpath-full-text-10-20110317 ed.) [Computer software manual]. <http://www.w3.org/TR/xpath-full-text-10/>.

Chen, Y., Wang, W., Liu, Z., & Lin, X. (2009). Keyword Search on Structured and Semi-Structured Data. In *Proceedings of the 35th sigmod international conference on management of data* (pp. 1005–1010). doi: 10.1145/1559845.1559966

internet.com K.K. (Japan). (2009). *An article from internet.com k.k. (japan) (in japanese).* <http://japan.internet.com/wmnews/20090908/5.html>.

Luo, Y., Wang, W., & Lin, X. (2008). SPARK: A Keyword Search Engine on Relational Databases. In *Data engineering, 2008. icde 2008. ieee 24th international conference on* (pp. 1552–1555). doi: 10.1109/ICDE.2008.4497619

Navathe, S. B., & Elmasri, R. A. (2002). *Fundamentals of Database Systems* (3rd ed.). Addison Wesley Longman.

Sarkas, N., Paparizos, S., & Tsaparas, P. (2010). Structured Annotations of Web Queries. In *Proceedings of the 2010 international conference on management of data* (pp. 771–782). NY, USA: ACM. doi: 10.1145/1807167.1807251

Selvan, M. P., Sekar, A. C., & Dharshini, A. P. (2012). Survey on Web Page Ranking Algorithms. *International Journal of Computer Applications*, *41*(19), 1–7. (Published by Foundation of Computer Science, New York, USA) doi: 10.5120/5646-7764

Tunkelang, D. (2009). Faceted Search. *Synthesis Lectures on Information Concepts, Retrieval, and Services*, *1*(1), 1-80. doi: 10.2200/S00190ED1V01Y200904ICR005

Appendix: The Syntax of Gradation Query Language

```
<query> ::= <exper>
<exper> ::= <term>
       | <exper> ' or ' <term>
<term> ::= <factor>
       | <term> (' and ' | ' ' | ' * ' | '.' <edge> '.') <factor>
       | 'not ' <term>
       | <term> ' as ' <alias>
       | <term> <cond>
       | <term> '[' <attrs> ']'
<factor> ::= '(' <exper> ')'
       | <keyword>
       | 'class:' <classname>
<edge> ::= <exper_edge>
<exper_edge> ::= <term_edge>
       | <exper_edge> ' or ' <term_edge>
<term_edge> ::= <factor_edge>
       | <term_edge> (' and ' | ' ' ) <factor_edge>
       | 'not ' <term_edge>
<fact_edge> ::= '(' <exper_edge> ')'
       | <edgename>
<cond> ::= <exper_cond>
<exper_cond> ::= <term_cond>
```

```
       | <exper_cond> ' or ' <term_cond>
<term_cond> ::= <factor_cond>
       | <term_cond> (' and ' | ' ' ) <factor_cond>
       | 'not ' <term_cond>
<fact_cond> ::= '(' <exper_cond> ')'
       | <attrname> <relational_op> ( <attrname> | <constant> )
<attrs> ::= <attrs> ',' <attrname>
       | <attrname>

<keyword> ::= <string> | <uri>
<constant> ::= <string>
<attrname> ::= '-'? ( <edgename> | 't_uri' | 'r_uri' )
<classname> ::= <identifier>
<edgename> ::= <identifier>
<alias> ::= '$' <identifier>

<relational_op> ::= '>' | '<' | '>=' | '<=' | '=' | '!='
<string> ::=  [0-9a-zA-Z\*\-\.]+
       | '"' [0-9a-zA-Z\*\-\.][0-9a-zA-Z\*\-\.\ ]* '"'
<identifier> ::= [a-zA-Z][a-zA-Z0-9_%]+
```