Semistructured Data Management
Part 2 - Graph Databases
Overview

• Schemas
• Graph Databases
• Schema Graphs
• Schema Extraction
• Indexing Semistructured Data

©2002, Karl Aberer, EPFL-SSC, Laboratoire de systèmes d'informations répartis
1. "To Schema or not to Schema?"

- **Benefits of schemas (e.g. DTD)**
  - Optimizing query evaluation and data storage
    - e.g. relational data storage for semi-structured data
  - Construction of Indexes
  - Understanding of data structure
    - e.g. browsing of databases, site map

- **Benefits of schema-less data (e.g. well-formed XML)**
  - no context information required to exchange data
    - e.g. context directly encoded into data (markup)
  - increased flexibility
    - e.g. adding or dropping dynamically structural elements such as attributes

Looking at Data on the Web we can make the following observations:

- The data is frequently schema-less. HTML and XML documents come along structured in an ad hoc manner. Structural properties (type information resp. classification) are implicit, hidden in the textual content of labels of structural elements, like links, titles, column names in tables etc. Having no schema has the advantage that users can exchange and process data without requiring a common context (the schema). Also it becomes easier to deal with the frequent changes of schemas on the Web. Of course the processing becomes less efficient and important meta information is possibly lost.

- Both a reason and consequence of data being schema-less is that the data is irregular, i.e. documents describing the same or similar content can be structured differently for every data instance.
Schemas on the other side are crucial for efficient data processing:

• For storage layout the database schema is used to partition the data: We have actually seen for the problem of relational XML storage, that knowing a schema is beneficial for determining more efficient storage schemes (both in terms of storage cost and data processing cost).

• For indexing and efficient query access the database schema is used to identify frequently used access paths: Knowing the schema allows to reason about queries without evaluating them. In conventional databases this corresponds to the process of semantically checking a query. In this process it is checked whether a query is correct with respect to the database schema, e.g. whether the attributes and relations used exist at all etc. As a result, it can be determined if an answer to the query is possible at all. If queries are semantically correct, by using the schema we may be able to decide which parts of a database need to be accessed at all in order to find an answer. The schema may also help modify the query by refining it to a query that is more precise and thus requires less data access.

Schemas are also extremely helpful in interfacing to users of the database, as the schema helps to summarize what kind of data and data structures can be expected in a database. This information in turn can be exploited to build more efficient human-computer interfaces, such as visual query interfaces or browsing interfaces.

So the problem we have to face is how can we apply techniques that take advantage of the availability of schemas without having a priori database schemas defined?
Schema Extraction

- Turning data with schema into schema-less data: trivial
  - requires embedding of schema information into data, see XML

- Turning schema-less data into data with schema: schema extraction
  - detect the structural regularities in a given database
  - exploit the embedded schema information
  - no well-defined solution: schemas might only partially represent the structure → no unique schema
  - non-trivial algorithmic task

- Study the problem for a generic data model: graph model
  - more general than specific models, such as XML, RDF, relational
  - avoid the "bells and whistles" built into the models
  - focus on the common algorithmic issues

- Basic primitive on the Web: navigation → directed graphs

The answer to the question of how to make schemas available, is that the schemas are to be extracted from the existing database. For semi-structured data models, such as XML, schematic information is directly embedded into the data. Schematic information is (meta-) data that is associated with data in order to make an interpretation possible, such as it is for example done in XML by enclosing content into element tags. Exploiting this embedded schematic information makes it possible to deduce from databases regularities in their occurrence and derive from that a-posteriori schema information. This process is called schema extraction. Schema extraction is a non-trivial task: first it requires a careful definition of what is a schema. This definition should allow some flexibility, since a schema should only describe only those parts of a semi-structured database that are actually regularly structured (otherwise the schema would just be a copy of the database itself). Extracting a schema is also a non-trivial algorithmic task. Given the difficulty of the task and for a wider applicability of the results the problem has been mostly investigated for generalizations of existing data models, that concentrate on their essential properties. A very abstract model that has been used for that purpose is the graph data model, which focuses essentially on navigation (or referencing) the most important way to link together information on the Web.
2. Graph Data Model

- **Graph Data Model**
  - technically simpler to treat than for example XML
  - databases and schemas are treated in the same framework: graphs

- **Graph Data Model Definition**
  - a data graph is a labeled, rooted graph
  - leaf nodes contain atomic and typed data values

Formally: A data graph $D=(V, E, R)$ is a labeled rooted graph, where

- $V \subseteq N$ is a finite set of nodes,
- $E \subseteq V \times L \times V$ is a set of labeled edges,
- $R \subseteq V$ is a set of root nodes and
- All nodes in $V$ are reachable from some root in $R$.

Formally:
- $A \subseteq V$ is a set of leaf nodes.
- $C \subseteq A \times U \times T$ is the data stored in $D$, where $U$ is the domain of all data values and $T$ is the domain of data types.
- The nodes $A$ are called atomic nodes.

The least common denominator with respect to structural properties of any data model is the fact that we can view all data as a labeled graph. This is true for XML/HTML documents where the tree structure can be seen as a graph and the labels are determined through the element types, as well as when looking at the Link structure of the Web, where the labels are derived from the textual content of the links.

For processing large quantities of Web data we need however to be able to apply techniques from database management. These exploit typically some regularity in the data, which are normally captured by database schemas.

The approach we will introduce in the following is:

- We consider Web data as data graph
- We construct graphs that can be used as schemas for the data graph, by extracting any regular (i.e. repeating) structures in the data graph
- We use these schema graphs for efficient processing of queries.

A graph database consists not only of structure (the labeled graph) but also of data content. The data content is attached to leaf nodes of the data graph, i.e. nodes with no outgoing edges. Data content can be of any atomic type, like integer, string etc.. We do not further specify the types here, we assume that a finite number of types is given. Since the nodes contain atomic data they are called atomic nodes.

In some cases, when it will play no role, we will omit the data content and focus on the structural part of graph databases only.
This is an example of a data graph. Compared to an XML structure there exist some subtle differences.

Instead of the nodes the edges are labeled.

The graph can have multiple roots and it is not a tree, but a general directed graph.

The labels of the edges are used in order to determine the meaning of the content of atomic nodes.
Graph Databases Properties

- Any database can be interpreted as graph database, e.g. relational, XML, object-oriented ...

- Extends and restricts the XML data model
  - Extension as arbitrary graphs are allowed
    (if one ignores ID/IDREF and Xlink)
  - Restriction: no order, no data types etc.

- Useful abstraction to study specific aspects of processing semi-structured data related to the processing of paths
  - Path indexing, path queries, schemas

- Establishes an application of automata theory to data management
  - e.g. XPath expressions are regular expressions; regular expressions are equivalent to finite state automata

Any database can be interpreted as graph database. On the other side the graph database model is much weaker in terms of expressible constraints and behavior.

The relationship between XML and the graph database model can be described as follows.

- In one sense the graph data model is an extension of XML as the core XML model only considers trees whereas graph databases allow general graphs. This is true for core XML. However, applications on top of XML (such as XLink, the successor of html links) support mechanisms to model graphs through referencing.

- In another sense the graph data model is more limited. Besides a number of minor details, like data types and constraints, the main restriction is the lack of ordering of elements (resp. nodes) which is inherent in XML documents.

The graph database model is however a useful abstraction to investigate many questions and techniques that have to do with the processing of semi-structured data, in particular aspects related to the use of paths in the data graph (e.g. processing of path queries as they can be formulated in XPath). It is also defined in a way that it makes standard automata theory immediately applicable, which allows to derive a number of important properties and algorithms both of practical and theoretical interest.
A simple method to look at the structure of a graph database is to analyze which paths occur in it. We could for example enumerate them and the set of possible paths describes the type of the data graph. However, this is a redundant representation.
A representation of the set of paths is by constructing another graph that represents all these paths, but does not repeat any of these paths as it is the case in the data graph.

Since the set of possible paths is a set of strings over the alphabet consisting of all element names, we might use a trie structure in order to represent this set and use the trie as a schema. This is actually already very close to the notion of schemas for graph databases that we will develop.
A more compact representation would combine those nodes that share common properties. For example, all string contents are of the same nature and can be considered as a common node. Nodes that are reached by the same paths also can be combined.
Simulation

- Why is $S_1$ a schema for $D$?
  - For every node $d$ in $D$ reached by a path $p$ starting from the root there exists a corresponding node $s$ in $S_1$ reachable by the same path and the types of the leaf nodes are the same in case $d$ is a leaf node.
  - Relationship $R$ between nodes of $D$ and $S_1$: $d R s$
  - $S_1$ simulates $D$, denoted as $D \sim S_1$ or $D \lessdot R S_1$

Formally:
Simulation: Given Graphs $G_1, G_2$ and a relation $R \subseteq V_1 \times V_2$, then $R$ is a simulation if for all labels $l \in L$ and for all $x_1, y_1 \in V_1$ and for all $x_2 \in V$ holds:
  - If $x_1 \rightarrow l y_1$ and $x_1 R x_2$ then there exists $y_2 \in V_2$ such that $y_1 R y_2$ and $x_2 \rightarrow l y_2$
  - We write then $G_1 \sim G_2$

Formally:
Rooted Simulation: for the roots $r_1$ and $r_2$ it holds $r_1 R r_2$; then $G_1 \sim_{\text{root}} G_2$
Typed Simulation: for all $x, y$: if $x R y$ and $y$ is an atomic type then $x$ must be an atomic node with content of that type.
Wildcards and alternate labels:
  - $x R _$ and $x R (x|y|…)$

A schema graph $S$ is said to be a schema for a data graph $D$ if $S$ is a rooted, typed simulation of $D$. $S$ may contain wildcards and alternate labels.

We want now to formalize what it means that a graph is a schema graph for a data graph. To that extent we introduce the notion of simulation. A simulation is characterized by a relation $R$. The definition of simulation expresses, that whenever we have in the data graph an edge and the node from which the edge emerges is related through the simulation to a node in the schema graph, then in the schema graph there exists an edge to a node that is related to the node to which the edge in the data graph connects to. The simulation relationship induces a partial order on graphs.

To make schema graphs more flexible, we extend first the definition of a data graph in two ways. We allow not only single labels, but also sets of labels to occur at the edges in the schema graph. Either we allow any possible edge (a wildcard, using the same notation as for path queries) or a specific set of tables (alternate labels). The definition of the simulation condition has to be extended in the straightforward manner:

If $x_1 \rightarrow_1 y_1$ and $x_1 R x_2$ then there exists $y_2 \in V_2$ such that $y_1 R y_2$ and $x_2 \rightarrow L' y_2$ and $l$ is contained in the set of labels specified by $L'$.

A further extension is required to deal with the typed contents of a data graph. For that purpose we introduce special nodes that are used to represent atomic content of a specific type. For each type of atomic content occurring in the data graph there exists exactly one node in the schema graph. Having extended the notion of data graphs to schema graphs, we have also to extend the definition of simulation, to take care of two properties of data graphs, namely roots and atomic nodes. We require that roots of the data and schema graph are related by the relation $R$ (rooted simulation). And we require that atomic nodes in the data graph are related to atomic type nodes of the same type in the schema graph.

This completes the definition for a schema graph for a semi-structured database $D$!
Asymmetry of the Simulation Relationship

- Example
  - \( D < S2 \) and \( S2 < D \) (bisimulation)
  - \( D < S1 \) but not \( S1 < D \)

For every node \( d \) in \( D \) reached by a path \( p \) starting from the root there exists a corresponding node \( s \) in \( S1 \) reachable by the same path and the types of the leaf nodes are the same in case \( d \) is a leaf node; but not vice versa!

One might wonder why the simulation relationship is asymmetric. The reason is that the "larger", the schema graph, may contain more structures than the data graph. In other words the data graph has to conform structurally to the schema graph but not vice versa. In case a simulation exists on both directions, we are talking of a bisimulation. Having a bisimulation means that the graphs are structurally equivalent (contain exactly the same kinds of paths).
Classification by Schema Graphs

- The relationship $R$ between nodes from the data graph ("data instances") and nodes from the schema graph ("data classes") implies a classification of the nodes of the data graph
- Example: data graph $D$ and schema graph $S_1$

<table>
<thead>
<tr>
<th>$R$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class</td>
</tr>
<tr>
<td>$s_1$</td>
</tr>
<tr>
<td>$s_2$</td>
</tr>
<tr>
<td>$s_3$</td>
</tr>
<tr>
<td>$s_4$</td>
</tr>
<tr>
<td>STRING</td>
</tr>
</tbody>
</table>

- Attention
  - Classification is not unique, i.e. a data node may belong to multiple classes
  - Classification can be ambiguous, i.e. a data node may belong to one or more of several classes

The natural way to classify data instances, which are the nodes of the data graph, is now to consider the node in the schema graph to which they are related via the simulation as the class to which the nodes in the data graph belong to.
Multiple classification occurs when the same node in the data graph can be reached by different paths and these paths are associated with different nodes (classes) in the schema graph. We illustrate this by slightly modifying our schema graph example. Note that this change does not affect the property that the schema graph is a simulation of the data graph.
Whereas multiple classification was required in the previous example in order to satisfy the simulation property, it can even lead to ambiguity in classification. Using the same schema graph, different simulations are possible leading to different classifications. In the example we could have for the simulation relation R, that the instance c2 is member of class s3 and s6 or only of class s3. In both cases we would have a correct simulation relationship since the definition of simulation only states that there exists at least one successful continuation in the schema graph, and that the nodes need to be related to all possible successful continuations.

This is of course an undesirable situation: given a database schema we cannot uniquely determine to which class a data object belongs. Consider for example a situation where the database schema is used to organize the storage and data is classified differently at the time when it is inserted to the database and when it is searched for. It could not be found. To remedy this situation we have to find a possibility to guarantee a unique classification.
Maximal Simulations

- Given two simulations $R_1$ and $R_2$ between a data graph $D$ and a schema graph $S$ the following holds
  
  $D \prec_{R_1} S$ and $D \prec_{R_2} S$ then $D \prec_{R_1 \cup R_2} S$

- Consequently there exists a maximal simulation from $D$ to $S$

- Unambiguous Classification: a data object belongs to a class if it is accordingly classified by means of the maximal simulation

- The maximal simulation can be computed as a fixpoint iteration

  Formally: Input data graph $D$; schema graph $S$;
  
  $R' := \emptyset$;
  $R := \{ (o, c) \mid o \in D, c \in S \}$;
  
  while $R \neq R'$
  
  $R' := R$;
  
  $R := \{ (o, c) \mid$ either $o$ is atomic value and $c$ it’s type
  
  or for all $o \rightarrow o' \in D$ there exists $(o',c') \in R'$ and $c \rightarrow c' \in S \}$

In order to arrive to a unique classification we first make an important observation on simulations: if two (different) simulations are given by means of two relations, then taking the union of the relations we obtain again a simulation. This fact was already illustrated in the previous example where one simulation was a subset of the other.

Due to this property a maximal simulation must exist, which is the largest relation $R$ that defines a simulation between a data graph $D$ and schema graph $S$. This largest relation is simply the union of all possible relations that define simulations.

Given that, we have identified a unique simulation among the set of all possible simulations. In order to avoid ambiguity we therefore decide to classify data objects based on this unique maximal simulation and to thus avoid ambiguity.

The maximal simulation can be computed as a fixpoint as follows: we start from the total relation between $D$ and $S$, i.e. all data objects and classes are related. Then we stepwise eliminate those pairs in $R$ that violate the simulation condition. That is, whenever a pair should be contained in $R$ we check whether a link exists in the data graph. In practice there exist more efficient algorithms to perform classification.

This classification method is for example required when we want to match an existing database to a given schema in the best possible way.
Example

- Example of a data and schema graph that would required multiple iteration steps for computing the maximal simulation.

```
1
  a   a
2  a   3
  a   a
  5  6  7
b  c  c
  8  9  0
```
```
c1
  a   a
  c1
  a   a
  c2  c3
  a   a
  c4  c5
c6
```

Actually the classification process in most practical cases completes in one step. This is an example where convergence does not occur in one step. The reason is, that in order to classify the nodes at the first level correctly the classification at the second level has to be performed in a first iteration. Only after we « know » to which classes 5,6,7 belong we can correctly relate 2,3,4 to c2 and c3.
Up to now we have only considered the situation where a schema is given and investigated the question of how a database conforms to this schema by introducing the notion of simulation. In practice the situation will occur that one starts from a fairly simple, and very general schema, and refines it over time, by adding structural information that is observed from the data. Of course, while doing that we would prefer not to loose conformance of the existing data with the new schema. Therefore we require that the refined schema must conform to all databases that conform to the original schema. A refined schema is called to subsume the schema that it refines.

Subsumption expresses that the refined schemas contains not only the constraints imposed by the original schema but also some additional ones that have been added.
Schema Subsumption

- What does it mean that one schema is a refinement of another?
  - All databases conforming to the schema S conform also to the refined schema S'.
  - This is exactly the case if S' < S!
  - In other words: if we can simulate schema S using S' then S' is a refinement, we say S' subsumes S.
  - Defines a partial order on all schemas!
  - Queries against more general schemas can also be answered against more special schemas, but not vice versa.

Formally:

For simulation relationships R and R':

\[ G_1 \sim_R G_2 \text{ and } G_2 \sim_R G_3 \text{ then } G_1 \sim_{R \circ R'} G_3 \]

where \( x (R \circ R') z \) iff exists \( y \) such that \( (x R y \text{ and } y R' z) \)

Consequence: if S' < S and D < S' then also D < S.

Formally: we observe that for simulations the following property holds: if we compose two relations defining simulations the resulting relation again defines a simulation relation. The composition of relations is defined in an analogous way to the join of binary relations with one matching attribute.

We use this property of composition of relations then as follows: if S is a schema for D (i.e. \( D < S \)) and S' subsumes S (i.e. \( S < S' \)) then we can conclude that \( D < S' \) and therefore S' is also a schema for D. Thus we have a well defined notion of schema subsumption.

Remark: schema subsumption is not only considered as an important concept for graph databases, but as well for other data models, in particular the relational model. There it is known that deciding whether one schema subsumes another is known to be an important (and hard) problem.
We illustrate schema subsumption by the two schemas S1 and S2 we have devised for D. The table illustrates on the one hand how the nodes in D are classified, and on the other hand the relation between nodes of S1 and S2 that specify the simulation between the two schemas. Schema S2 is obviously less refined as it induces a coarser classification of data. It allows to accommodate more different data graphs than S1.
Summary

- Why are schemas useful for databases?
- Which characteristics of a data graph is captured by a schema graph?
- What is the concept of rooted, typed simulation needed for?
- Does there exist more than one schema graph for a data graph?
- What is the difference between the notions non-unique and ambigue classification?
- What can we obtain from a maximal simulation?
- What is schema subsumption?
- Why is the notion of schema subsumption important?

©2002, Karl Aberer, EPFL-SSC, Laboratoire de systèmes d'informations répartis
3. Schema Extraction

• Construction of a Schema Graph from the Data Graph
  – Automatically
  – There exists no unique schema graph

• Goal: construct a schema graph that is
  – Accurate: every path that occurs in the schema graph occurs in the data graph and vice versa
  – Concise: every path occurs only once

• This schema graph is called Data Guide

Up to now we have looked at the questions (1) how to classify data given a schema graph and (2) how to compare two schema graphs.

In both cases the assumption was that a schema is given. If we look at the situation we encounter on the Web we will often have very large and complex semi-structured databases without schemas, and it will be very difficult to devise a correct schema manually. Therefore we go now one step further and look at the problem whether a schema can also be automatically generated. Being able to generate schemas automatically can have many benefits. Besides the usual benefits already mentioned earlier, this can be particularly useful for data integration, where an extracted schema may be used to define mappings between the data in different heterogeneous databases.

Of course, we have also seen that many different schemas for the same database may exist. For an automatic schema extraction from a database we need thus a well-defined criterion for determining which schema to generate out of the set of possible schemas. To that end we require the following:

- The schema should be accurate. That means a path that occurs in the schema must also be present in the data graph (the mechanism must not "invent" paths -> correctness of the schema) and any path that occurs in the data graph must be represented in the schema (the mechanism must not "forget" data paths, -> completeness of the schema)
- The schema should be concise: the schema should contain no redundancy, i.e. every path that occurs in the schema should occur only once.

Actually observing these criteria leaves only one possible schema to be constructed, which is then called the data guide.
Data Guide Construction

• Nodes of the data guide correspond to subsets of data graph nodes
• The set containing the data graph root is the root of the data guide
• For each data guide node constructed so far and for each edge label of an edge leaving some node from the node set of the data guide node do the following
  - form the set of nodes in the data graph that are reached by this edge label
  - either the set exists already in the data guide: create a labeled edge to it
  - otherwise create a new data guide node and connect it by the edge
• Continue till no more new data guide nodes are created

1 dg := new node in DG; R := {({root}, dg)}; Make({root}, dg);
2 Make(s1, d1)
3 { p := { (l, o) | o' ->, o in D and o’ in s1};
4   for each l occurring in p
5     s2 := {o | (l, o) in p};
6     if exists (s2, d2) in R add an edge d1 ->, d2 to DG
7     else d2 := new node in DG;
8       R := R union {(s2, d2)};
9       add an edge d1 ->, d2 to DG;
10      Make(s2, d2); }

We now give an algorithm for constructing the data guide and we will analyze its properties subsequently.

The basic idea of the algorithm is to construct, starting from the root of the data graph, for every possible path in the data graph the set of data objects that are reached by exactly this path.

Let us analyze the algorithm in more detail. It starts by creating a root node in the data guide. The root node in the data guide is related to the root node of the data graph, and we store this relationship in R (line 1). The relationship R is the simulation relationship. Then we start the recursive procedure of constructing the data guide by calling Make (line 2) with a given set of nodes (class) and the corresponding representation of it in the data guide.

When Make is called we investigate all labeled edges that start from a node of the class we are currently looking at (s1) and store them in a set p (line 3). For each different label that we find in p we have now to check how to represent the new path (that is the extension of the path we have reached so far by an edge with label l), whether we have already a class in R that represents exactly the nodes that can be reached by this path (these are the nodes s2 from line 5) or whether a new class needs to be created in the data guide. If already a class exists (line 6) then we have simply to add a new edge to the data guide. If it does not exist (line 7), then we have to create a new node in the data guide, add the relation between this new node and the set s2 into R (line 8) and add the correct edge into the data guide (line 9). Only if a class has been newly created we have to invoke the recursive procedure again for this new class (line 10) in order to create necessary extensions of the data guide as a result of the creation of the new class.

The algorithm terminates for cycles, since when arriving a second time at the same class in the cycle the corresponding class has already been created and we go to the condition of line 6, where no recursive calls will be made.

Observe that even in the presence of cycles such an algorithm must terminate, as the number of subsets of nodes in the data graph is finite. Thus in the worst case for each subset of the nodes of the data graph a node in the data guide is constructed. This gives an idea on a principal problem of data guides, namely complexity. However, in practice if we have regular data graphs with a lot of redundant structure this does not play a role.
We give the example of a data guide. When carefully analyzing the resulting
data guide one can observe several effects.

- The nodes typically occur in multiple places in the data guide
- Repeating structures in the data graph are reduced to a single occurrence in
  the data guide
- Nevertheless the data guide can be more complex than the data graph itself
- The data guide is deterministic. That means that all outgoing edges in the
  data guide are different.

In addition, if cycles are present in the data graph, they lead to cycles in the
data guide.
Properties of the Data Guide

- **Observations**
  - cycles in the data graph lead to cycles in the data guide
  - nodes may occur in multiple places in the data guide
  - Repeating structures in the data graph are reduced to a single occurrence in the data guide
  - the data guide can be more complex than the data graph itself
  - The data guide is deterministic: all outgoing edges in the data guide have different labels

- **It can be shown that:**
  The data guide is the *minimal, deterministic schema graph*
  - Every other deterministic schema graph is subsumed by the data guide

The property of being the minimal, deterministic schema graph provides a concise criterion for defining data guides.
Semistructured Data Indexing

- Each node in the data guide is a hash table
  - The outgoing label is the key
  - The address of the data guide node reached by the labelled edge is stored in the hash table entry
  - The list of nodes in the data graph reachable via the label is stored in the hash table entry

- Query processing
  - If in a path query a label is encountered, then the respective label is looked up
  - If another label follows, the next node in the data guide is looked up
  - If it is the last node of the query the data objects are returned

- A problem with data guides is their potential complexity
  - Can be much larger than the database

- Question: is it necessary to require a deterministic graph for indexing?

As a data guide summarizes precisely what kind of paths occur in a graph database and associates the nodes of the graph database with the paths, we can use the data guide as an index structure. With respect to evaluation of queries we basically evaluate the query against the data guide rather than against the database itself, and obtain the result nodes from the graph database only at the end when the query has been processed and an answer has been found in the data guide. To make a data guide a real index structure we have also to think about the way it is actually stored, such that it can be efficiently accessed. To that end a simple approach is to create for every node of the data guide a hash table with the outgoing labels as keys and the references to the nodes reached by the labeled edge as values.
Query Answering

- Evaluating /programmer/employee/leads
Node Equivalence

- Observation: If the possible paths leading to two nodes in the data graph are the same, the nodes are equivalent from the viewpoint of query processing
  - p1 and p3 are indistinguishable, whichever query is used to distinguish them
  - other examples: e1, e2, and e2, e4 are indistinguishable

Formally:
Language Equivalence
\[ L_x = \{ p=p_1 \ldots p_n \mid p \text{ is a path from the root to } x \} \]
\[ x \equiv y \text{ if } L_x = L_y \]
equivalence class of x denoted as \([x]\)

A problem with data guides is their potential space complexity. It results mainly from the requirement of having a deterministic graph (each node has only outgoing edges with different labels). It is a well-known fact that a deterministic graph (automaton) that is equivalent to a non-deterministic one is in the worst case exponential in the size of the non-deterministic graph (automaton). This leads to the idea that it might be worthwhile to consider the possibility of using non-deterministic schema graphs in order to construct indexes.

The construction of a non-deterministic automaton for indexing starts with a basic observation. If the set of paths that lead to a node in the data graph is the same for two nodes, they are indistinguishable from the viewpoint of processing a path query. Either they are always both in the result or not. Thus we can declare them as equivalent. This leads to the notion of language equivalence which can be formally defined and for which we introduce the corresponding notations.
Index Construction

- Based on the language equivalence classes
  - Nodes of Index I are the node equivalence classes of the data graph
  - Edge between equivalence classes if corresponding edge exists in the data graph for the members of the equivalence class (unambiguously determined)
  - Nondeterministic graph (multiple outgoing edges with same label)

Having introduced the notion of language equivalence we are almost done. We define the index by declaring every equivalence class as one node of the index graph and connecting the nodes with a labeled edge if such an edge exists between the members of the equivalence class in the data graph. This is an unambiguous definition, since the equivalence classes cannot be distinguished from the viewpoint of accessibility through paths. Thus either all members of two equivalence classes are connected by a labeled edge with a specific label or none (otherwise the classes would split). This completes the definition of the index graph! Similarly as for data guides we can now create a data organization based on hash tables for each node in the graph and query processing proceeds in the same way, with the only difference that probably from a given node multiple paths need to be explored for the same label. For obvious reasons the index cannot be larger than the database itself now!
Query Answering

- Evaluating /programmer/employee/leads
Computing Language Equivalence

• Drawback of using language equivalence
  - Checking for language equivalence is PSPACE complete

• Idea: approximate language equivalence by a computationally more feasible relation

• Approximation: a relation $\approx$ is an approximation of $\equiv$ if
  
  for all $x, y \ x \approx y \Rightarrow x \equiv y$

• Using a approximate relation for constructing the index graph
  - Queries will be correctly answered
  - Query evaluation more expensive (more nodes to check)

Building the index (or better schema used for indexing) based on language equivalence suffers from the fundamental problem that language equivalence is a very hard problem to solve in general. Thus for having a practically useable approach one considers relations on the nodes that are easier to determine but maintain the properties required for indexing. This is achieved by approximating the language equivalence relation, by refining it. Refining means that equivalence classes under language equivalence may be split in multiple classes. This does not influence the correctness of query answering on the schema graph (but it can increase the necessary amount of work since more nodes need to be inspected).

In the following we introduce a construction for an approximation relation that in practice delivers almost in all cases the same relation as the language equivalence relation (in other words it will be hard to find examples where the relations differ), but is more efficient.
Reverse Bisimulation

- Reverse Bisimulation relation ≈
  - Equivalent to the simulation relation in both directions with the edges of the graphs inverted
  - On any graph there exists a maximal reversed bisimulation relation which can be efficiently computed and
    for all x, y, x ≈ y ⇒ x ≡ y

Formally:
Reverse Bisimulation
if x ≈ y and x is the root then y is the root, and vice versa
if x ≈ y and there exists an edge x' → l x, then there exists an edge y' → l y and x' ≈ y', and vice versa

For introducing an approximation relation we come back to the notion of simulation: we perform simulation in the backwards direction (why?) and the simulation must be done in both directions in order to obtain an equivalence relation. This produces an approximation relation and there exist efficient algorithms to compute this relation, which we will not be able to discuss here.
Language Equivalence vs. Reversed Bisimulation

- Normally language equivalence and reverse bisimulation coincide
- Counterexample: $x \equiv y \equiv z$, but $x \not\approx y \not\approx z$

In practice reversed bisimulation and language equivalence coincide. We give here an example were this is not the case.
Recapitulation

• What is a data guide and how does a data guide compare to other schema graphs?

• What are properties of data guides with respect to the data graph they are constructed from?

• How is a data guide used for indexing?

• What is the approach to reduce the potential size of an index derived from a data guide?

• How is language equivalence used to construct an index?

• How is the problem that checking language equivalence is expensive circumvented?
References

• Course material based on
  - S. Abiteboul, P. Bunemann, D. Suciu: Data on the Web: From Relations to Semistructured Data and XML, Morgan Kaufman, 2000 (in particular chapters 4.1, 7.1, 7.3.3, 7.4.1)

• Relevant articles