SEMISTRUCTURED DATA is often explained as "schemaless" or "self-describing," terms that indicate that there is no separate description of the type or structure of data. Typically, when we store or program with a piece of data, we first describe the structure (type, schema) of that data and then create instances of that type (or populate) the schema. In semistructured data we directly describe the data using a simple syntax. We start with an idea familiar to Lisp programmers, association lists, which are nothing more than label-value pairs and are used to represent recordlike or tuplelike structures:

```
{name: "Alan", tel: 2157786,  email: "agb@abc.com"}
```

This is simply a set of pairs such as `name: "Alan"` consisting of a label and a value. The values may themselves be other structures as in

```
{name: {first: "Alan",  last: "Black"},
 tel: 2157786,
 email: "agb@abc.com"
}
```

We may represent this data graphically as a node that represents the object, connected by edges to values (see Figure 2.1).

However, we depart from the usual assumption made about tuples or association lists that the labels are unique, and we will allow duplicate labels as in

```
{name: "Alan, tel:  2157786, tel: 2498762 }
```

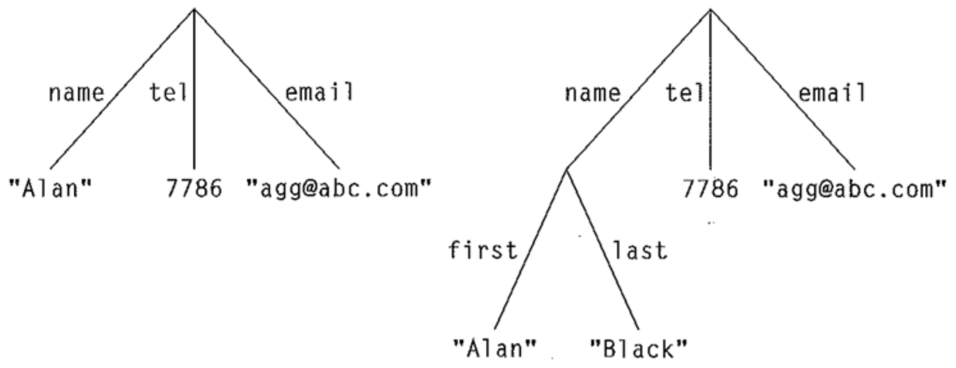


Figure 2.1 Graph representations of simple structures.

The syntax makes it easy to describe sets of tuples as in

```
{ person:
     {name: "Alan", phone: 3127786, email: "agg@abc.com"},
  person:
     {name: "Sara", phone: 2136877, email: "sara@math.xyz.edu"},
  person:
     {name: "Fred", phone: 7786312, email: "fds@acme.co.uk"}
}
```

We are not constrained to make all the `person` tuples the same type. One of the main strengths of semistructured data is its ability to accommodate variations in structure. While there is nothing to prevent us from building and querying completely random graphs, we usually deal with structures that are "close" to some type. The variations typically consist of missing data, duplicated fields, or minor changes in representation, as in

```
{person:
      {name: "Alan", phone: 3127786, email: "agg@abc.com"},
 person:
      {name: {first: "Sara", last: "Green"},
       phone: 2136877,
       email: "sara@math.xyz.edu"
      },

 person:
     {name: "Fred", Phone: 7786312  Height: 183}
}
```

In a programming language such as C++, it may be possible, by a judicious combination of tuple types, union types, and some collection type such as a list, to give a type to each of the examples above. However, such a type is unlikely to be stable. A small change in the data will require a revision of the type. In semistructured data, we make the conscious choice of forgetting any type the data might have had, and we serialize it by annotating each data item explicitly with its description (such as `name`, `phone`, etc.). Such data is called *self-describing*. The term *serialization* means converting the data into a byte stream that can be easily transmitted and reconstructed at the receiver. Of course self-describing data wastes space (if naively stored), since we need to repeat these descriptions with each data item, but we gain interoperability, which is crucial in the Web context. On the other hand, no information is lost by dropping the type, and we show next how easily familiar forms of data can be represented by this syntax.

2.2 REPRESENTING RELATIONAL DATABASES

A relational database is normally described by a schema such as `r1(a,b,c)` `r2(c,d)`. In these expressions, `r1` and `r2` are the names of the relations, and `a,b,c` and `c,d` are the column names of the two relations. In practice we also have to specify the types of those columns. An instance of such a schema is some data that conforms to this specification, and while there is no agreed syntax for describing relational instances, we typically depict them as rows in tables (see Figure 2.2).

We can think of the whole database instance as a tuple consisting of two components, the `r1` and `r2` relations. Using our notation, we describe this as {`r1`: i_1, `r2`: i_2}, where i_1 and i_2 are representations of the data in the two relations, each of which can be described as a set of rows:

```
{r1: {row: {a: a1, b: b1,  c: c1},
      row: {a: a2, b: b2,  c: c2}
      },
 r2: {row: { c: c2,  d: d2},
      row: { c: c3,  d: d3},
      row: { c: c4,  d: d4}
      }
}
```

It is worth remarking here that this is not the only possible representation of a relational database. Figure 2.3 shows tree diagrams for the syntax we have just given and for two other representations of the same relational database.

r1:

a	b	c
a1	b1	c1
a2	b2	c2

r2:

c	d
c2	d2
c3	d3
c4	d4

Figure 2.2 Examples of relations.

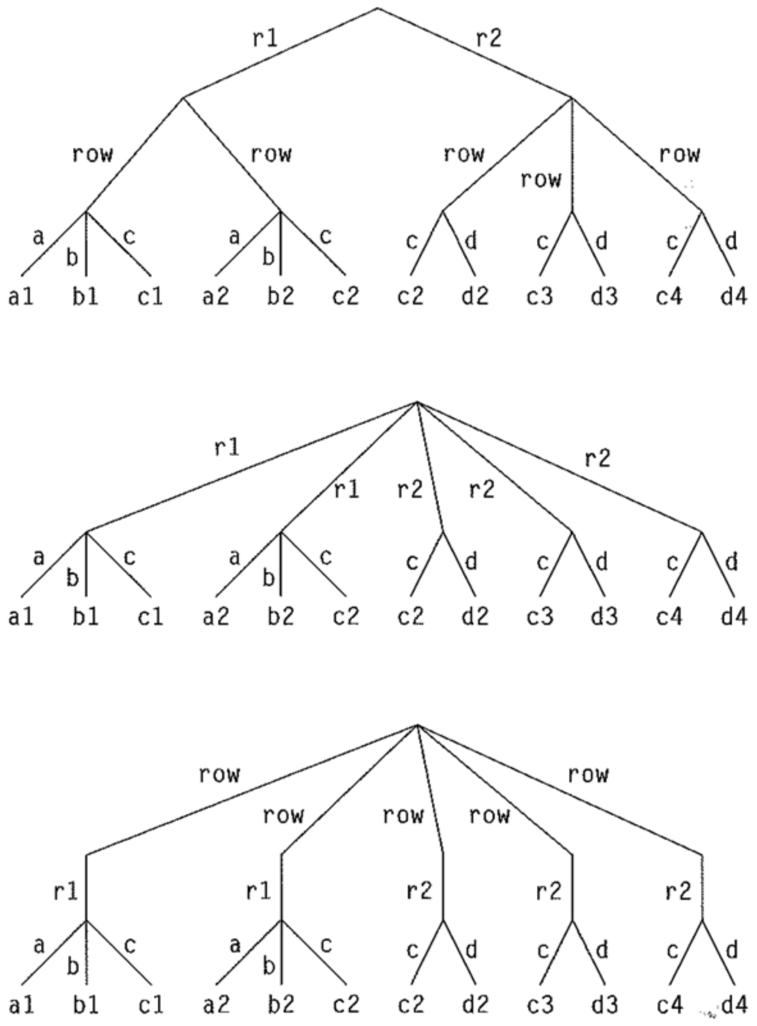


Figure 2.3 Three representations of a relational database.

2.3 REPRESENTING OBJECT DATABASES

Modern database applications handle objects, either through an object-relational or an object database. Such data can be represented as semistructured data too. Consider for example the following collection of three persons, in which `Mary` has two children, `John` and `Jane`. Object identities may be used when we want to construct structures with references to other objects:

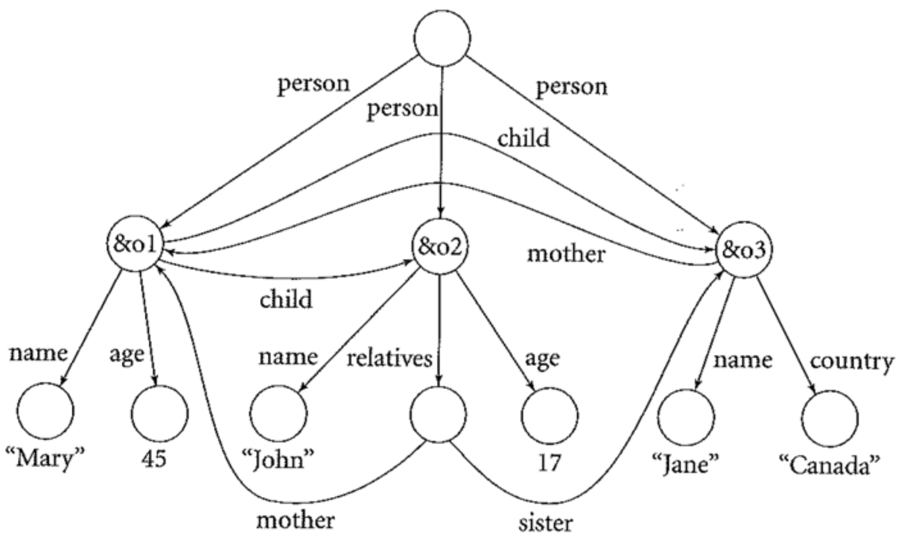


Figure 2.4 A cyclic structure.

```
{person: &o1{name:   "Mary",
             age:    45,
             child:  &o2,
             child:  &o3
            },
 person: &o2{name:   "John",
             age:    17,
             relatives: {mother: &o1,
                         sister: &o3}
            },
 person: &o3{name:  "Jane",
             country: "Canada",
             mother: &o1
            }
}
```

The presence of a node label such as `&o1` before a structure binds `&o1` to the identity of that structure. We are then able to use that label—as a value—to refer to that structure. In our graph representation we allow ourselves to build graphs with shared substructures and cycles, as shown in Figure 2.4. The names `&o1`,

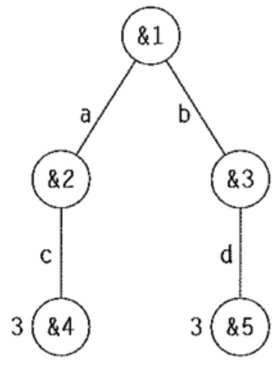


Figure 2.5 A structure with object identities.

&o2, &o3 are called *object identities*, or oids. In this figure, we have placed arrows on the edges to indicate the direction, which is no longer implicitly given by the treelike structure.

At this point the data is no longer a tree but a graph, in which each node has a unique oid. We shall freely refer to nodes as *objects* throughout this book. Note that even the terminal nodes have identities. In Figure 2.5, we have drawn circles around the object identities to differentiate the values stored at terminal nodes from the object identities for those nodes. This suggests that we could also place data (i.e., atomic values) at internal nodes in a structure. We shall not do this in our initial formalism for semistructured data; atomic values will be allowed only on leaves, since this simple formalism suffices to represent any data of practical interest.

Since oids are just strings or integers, their meaning is restricted to a certain domain. Although oids have a logical meaning, it is important to understand what they denote physically. For data loaded in memory, an oid can be a seen as a pointer to the memory location corresponding to a node and is only valid for the current application. For data stored on disk, an oid may be an address on the disk for the node, or an entry in a translation table that provides the address. In this case, the same oid is valid across several applications, on all machines on the local network sharing that disk. In the context of data exchange on the Web, however, an object identifier becomes part of a global namespace. We may use, for instance, a URL followed by a query that, at that URL, would extract the object. Indeed, we need this or another means of *locating* an object. In the same data instance, oids from different domains may coexist and may be translated during data exchange. For example, the oid of an object on the Web may be translated into a disk-local oid when that object is fetched on the local disk, then translated into a pointer when it is loaded in memory.

In our simple syntax for semistructured data, we allow both nodes with explicit oids and nodes without oids: the system will explicitly assign a unique oid automatically, when the data is parsed. Thus `{a:&o1{b:&o2 5}}` and `{a:{b:5}}` denote isomorphic graphs, as does `{a:&o1{b:5}}`.

This convention requires us to take some care about what is denoted by this syntax. Consider the simple structure

```
{a: {b: 3}, a: {b: 3}}
```

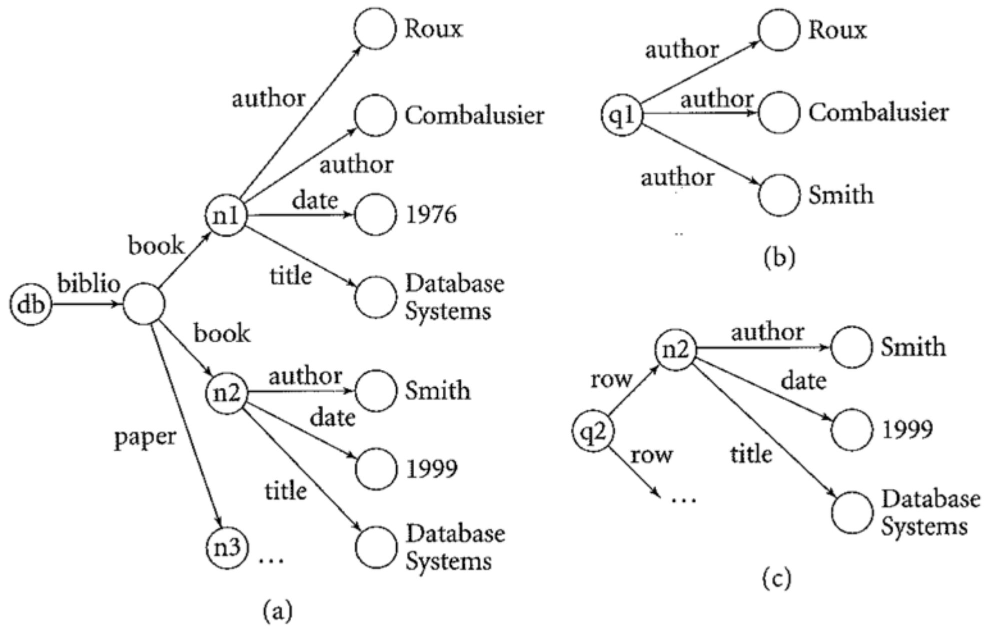
Since a separate object identifier is assigned to each component, each of the repeated `{b: 3}` components will have a different identifier. Thus this represents a set with *two* elements (there are two edges from the root in the graph representation). If this were a representation of a relation, we might treat this as a structure with *one* tuple, equivalent to `{a: {b: 3}}`. To start with, we shall assume that each node in the graph is given a unique oid when loaded by the system, so that this is a set of two objects. Later, in Chapter 6, we shall examine an alternative semantics in which object identities are not added.

4.1 PATH EXPRESSIONS

One of the main distinguishing features of semistructured query languages is their ability to reach to arbitrary depths in the data graph. To do this they all exploit, in some form, the notion of a *path expression*. Recall our model of data as an edge-labeled directed graph. A sequence of edge labels $l_1.l_2.\ldots.l_n$ is called a *path expression*. We view a path expression as a simple query whose result, for a given data graph, is a set of nodes. For example, the result of the path expression `"biblio.book"` in Figure 4.1(a) is the set of nodes {`n1`, `n2`}. The result of the path expression `"biblio.book.author"` is the set of nodes with the associated set of strings {`"Combalusier"`, `"Roux"`, `"Smith"`}.

In general, the result of the path expression $l_1.l_2.\ldots.l_n$ on a data graph is the set of nodes v_n such that there exist edges $(r, l_1, v_1), (v_1, l_2, v_2), \ldots, (v_{n-1}, l_n, v_n)$ in the data graph, where r is the root. Thus, path expressions result in sets of nodes. They do not (yet) result in pieces of semistructured data.

Rather than specify a path completely, we may want to specify it by some property. This property may be a property of the path (e.g., the path must traverse a `paper` edge) or it may be a property on an individual edge label (e.g., the edge label contains the substring `"bib"`.) We shall use *regular expressions*, at two levels,



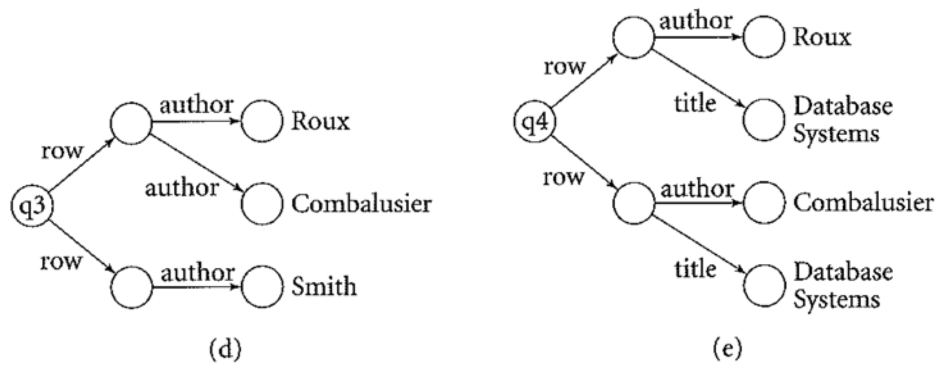


Figure 4.1 Database and answers.

both on the alphabet of edge labels and on the alphabet of (ASCII) characters that comprise labels, to describe such path properties. Starting with the alphabet of edge labels, a regular expression such as `book|paper` matches either a `book` edge or a `paper` edge. In the database of Figure 4.1, it could be used in the query `biblio.(book|paper).author`

Another useful pattern is a "wild card," which matches any edge label. We shall use the symbol _. Thus `biblio._.author` matches any path consisting of a `biblio` edge followed by any edge followed by a `author` edge. We can also consider a Kleene closure operation, denoted `*`, which specifies arbitrary repeats of a regular expression. For instance, `biblio._*.author` specifies nodes at the end of a path that starts with a `biblio` label, ends with an `author` label, and has an arbitrary sequence of edges between.

The general syntax of regular expressions on paths is

$$e ::= l \mid \epsilon \mid \_ \mid e'.'e \mid '('e')' \mid e'—'e \mid e'*' \mid e'+' \mid e'?'$$

where l ranges over labels, e over expressions, and ϵ is the empty expression. The expression $e*$ stands for 0 or more repeats of e, that is, for

$$\epsilon \mid e \mid e.e \mid e.e.e \mid e.e.e.e \mid \ldots$$

Also, $e+$ stands for one or more repeats, and $e?$ for zero or one occurrence of e.

So far, we can either specify completely a label or use the _ wild card. To specify more complex label patterns, we shall use the established syntax of grep. For example, the pattern

```
((s|S)ection|paragraph)(s)?
```

matches any six labels: `section`, `Section`, `sections`, `Sections`, `paragraph`, and `paragraphs`. To avoid ambiguities between the regular expressions for labels and those for path expressions we enclose the former in quotation marks. (Note that the period is used as a character wild card in grep, but it is used as the concatenation operator for path expressions.) As an example of a generalized path expression that contains both forms of matching, the pattern

```
biblio ._*.section.("[tT]itle" | paragraph.".*heading.*")
```

matches any path that starts with a `biblio` label and ends with a `section` label followed by either a (possibly capitalized) `title` or a `paragraph` edge followed by an edge that contains the string `heading`.

In some models of semistructured data, the edge labels are not limited to being strings. They may be integers or some other category of strings. A query language that makes use of path expressions may coerce all these types to an underlying string representation and match a regular expression to this representation.

When the data graph has cycles, it is possible to specify paths of arbitrary length. Referring to Figure 2.7, the path expressions

```
cities.state-of
cities.state-of.capital
cities.state-of.capital.state-of
cities.state-of.capital.state-of.capital
```

all match the data graph. Moreover the regular expression _* matches an infinite number of paths whenever there is a cycle. The set of nodes specified by this path expression is still finite, being a subset of the nodes in the data graph, but we need to exercise more care when computing the result of a regular exprssion on a data graph with cycles.

In general, given a regular expression e and a data graph D, the result of e on D is computed as follows. First, construct some automaton A that is equivalent to the regular expression e; any nondeterministic automaton would do.[1] Let $\{x_1, x_2, \ldots\}$ be the set of nodes in D, with x_1 being the root, and let $\{s_1, s_2, \ldots\}$ be the set of states in A, with s_1 being the initial state. Compute a set *Closure* as follows. Initially $Closure = \{(x_1, s_1)\}$. Repeat the following, until *Closure* does not change any more. Choose some pair $(x, s) \in Closure$, and consider some edge $x \xrightarrow{a} x'$ in the datagraph D, and some transition $s \xrightarrow{a} s'$ in A, labeled with the same label (denote here a); add the pair (x', s') to *Closure*. After *Closure* reaches a fixpoint, the result of e on D consists of the set of nodes x such that *Closure* contains some pair (x, s), with s a terminal state in A. Note that this procedure always terminates. In the worst case, it has to visit each node x in the data graph as many times as states in the automaton A. However, in practice, it will visit most nodes at most once and may not visit portions of the database at all.

4.2 A CORE LANGUAGE

Path expressions, although they are an essential feature of semistructured query languages, only get us some distance toward a query language. They can only return a subset of nodes in the database. They cannot construct new nodes, they cannot perform the equivalent of a relational database join, and they cannot test values stored in the database. Path expressions form the basic building block of the language. Standard query language features will do the rest.

The syntax of languages that have been developed so far is in a state of flux. We shall adopt a rather conservative approach and then deal with various extensions. We shall base our core syntax on Lorel and UnQL, which, to within very minor syntactic details, agree on this core.

4.2.1 The Basic Syntax

As in Lorel, our basic syntax is based on OQL and illustrated with the following query:

```
% Query q1
select author: X
from   biblio.book.author X
```

which computes the set of book authors. Informally, the semantics of the query is to bind the variable `X` to each node specified by the path expression. The effect of the query is to form one new node and connect it with edges labeled `author` to the nodes resulting from the evaluation of the path expression `biblio.book.author`. The result of the query is a new data graph shown in Figure 4.1(b) whose output corresponds to the ssd-expression

```
{author: "Roux", author: "Combalusier", author: "Smith"}
```

In our next example, we restrict the output by use of a condition in the `where` clause of the query:

```
% Query q2
select row: X
from   biblio._ X
where "Smith" in X.author
```

whose output shown in Figure 4.1(c) corresponds to the ssd-expression

```
{row: {author: "Smith",
       date: 1999,
       title: "Database Systems"}, ...}
```

The notation `X.author` is also a path expression whose root is taken as the node denoted by `X`. In general, the result of this expression is a set, and the predicate `"Smith" in X.author` is a predicate that tests for set membership.

Assuming a predicate `matches` for matching strings to regular expressions, we can write the following:

```
select author:Y
from   biblio._ X,
       X.author Y,
       X.title Z
where  matches(".*(D|d)atabase.*", Z)
```

This collects all the authors of publications whose title contains the word "database". Note how each line of the `from` clause introduces a new variable.

In general, the semantics of a query `select E from B where C` is defined in three steps. The first deals with `from`, the second with `where`, and the last with `select`. Assume that `B` defines three variables, `X, Y, Z`. In the first step, we find the set of all bindings of these variables specified by `B`; each binding maps the variables to oids in the data graph. In the second step, we filter the bindings that satisfy `C`. Let this set be

$$\{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}.$$

Here each value $x_1, \ldots, z_n$ is an oid from the input graph. In our illustration, this is a ternary relation, but in general it is a relation whose arity is the number of variables defined in `B`. In the last step, we construct the ssd-expression that forms the result

$$\{\mathtt{E}(x_1, y_1, z_1), \ldots, \mathtt{E}(x_n, y_n, z_n)\}.$$

Each $\mathtt{E}(x_i, y_i, z_i)$ denotes `E` in which x_i, y_i, z_i are substituted for `X, Y, Z`. Finally, the result corresponds to an ssd-expression.

Illustrating the example above, in the first step we find the set of all values for `X, Y, Z` such that `biblio._ X`, `X.author Y`, `X.title Z`. In the second step, we keep only those such that `matches(".*(D_d)atabase.*",Z)`. Let

$$\{(x_1, y_1, z_1), \ldots, (x_n, y_n, z_n)\}$$

be this set. In the last step, we construct the result: `{author:`y_1`, author:`y_2`, . . . , author:`y_n`}`. The query therefore creates just one new node in the data graph (the root).

Some queries create more than one new node. For instance, consider

```
select row: { title: Y, author: Z}
from   biblio.book X, X.title Y, X.author Z
```

Here the result will be constructed as

`{row:{title:`y_1`, author:`z_1`},...., row:{title:`y_n`, author:`z_n`}}`.

A new node is used for the entire result. Plus for each binding (x_i, y_i, z_i), a new node is constructed to represent the row `{title:`y_i`, author:`z_i`}`.

Another means of creating many new nodes is by nesting subqueries in the select clause since each instance of the nested subquery will potentially create new objects. This is illustrated in the following two queries:

```
% Query q3
select row:( select author: Y
             from   X.author Y)
from   biblio.book X


% Query q4
select row: ( select author: Y, title: T
              from   X.author Y,
                     X.title  T)
from  biblio.book X
where "Roux" in X.author
```

The output of q3 corresponds to the ssd-expression:

```
{ row: {author:  "Roux", author: "Combalusier"},
  row: {author: "Smith"} }
```

The data graph of the result is shown in Figure 4.1(d) and should be compared with the output from query q1.

The more complex query q4 implements a generalized form of projection and selection. The output is shown in Figure 4.1(e).

We can also express joins. Consider for instance a relational schema r1(a,b), r2(b,c), for which the following ssd-expression represents an instance:

```
{ r1: { row: {a: 1, b: 2},
        row: {a: 1, b: 3} },
  r2: { row: {b: 2, c: 4},
        row: {b: 2, c: 3} } }
```

Consider the query

```
% Query q-join
select a: A, c: C
from   r1.row X,
       r2.row Y,
       X.a A, X.b B, Y.b B', Y.c C
where  B = B'
```

where X and Y range, respectively, over the rows in r1 and r2. The query computes the join of the two relations on their B attributes, that is, their natural join, and projects on the *A* and *C* attributes.

Suppose that rows are allowed multiple B values. This query also computes the "join." But we have to be careful about what we mean by a join. Two rows match if they have some common B value. As another example of a join with a more sophisticated condition, the following query asks for authors that are referred to at least twice in some paper with "Database" in the title:

```
select row: W
from   biblio.paper X, X.refers-to Y, Y.author W,
       X.refers-to Z
where  NOT ( Y = Z )
  and  W in Z.author
  and  matches("*Database*", X.title)
```

4.3 MORE ON LOREL

The core language is, as we have noted, a fragment of Lorel, the query language in Lore (Lightweight Object REpository), a system for managing semistructured data. Lorel was essentially derived by adapting OQL to querying semistructured data. We now complete the description of Lorel, which provides for terser queries through the use of syntactic shortcuts and coercions that may make the language more attractive to, say, an SQL programmer.

The first of these concerns the omission of labels. Consider

```
select X
from   biblio.book.author X
```

Since we have not placed any label in the select clause, it is not a priori clear how this query is to produce semistructured data. Lorel adds a default label. To be consistent with other examples, we shall assume here that row is chosen as the default. (The default in Lorel would be answer). The result of the query is therefore

```
{row: "Roux", row: "Combalusier", row: "Smith"}.
```

Lorel also allows us to use path expressions in the select clause.

```
% Query q3′
select X.author
from   biblio.book X
```

Observe that for each X, X.author denotes a set of nodes. Indeed, we may understand X.author as the nested query

```
select author: Y
from   X.author Y
```

In general, an expression of the form X.p.l, where p is an arbitrarily complex path expression, is understood as the nested query

```
select l: Y
from   X.p.l Y
```

Now, in general, the semantics associated to path expressions in the select clause is understood by rewriting them into nested queries. In particular, query q3′ above corresponds to

```
select row: (select author: Y
             from   X.author Y)
from  biblio.book X
```

Thus, q3′ is the same query as q3.

A second use of overloading in Lorel occurs in the use of comparison predicates, especially equality. For instance, consider

```
select row: X
from biblio.paper X
where X.author = "Smith"
```

Note that X.author is strictly speaking a *set* of authors. Here we are using an equality predicate on disparate types. The intended meaning is "Smith" in X.author, that is,

```
select row: X
from biblio.paper X
where exists Y in X.author ( Y = "Smith" )
```

A similar rewriting is performed for other predicates as well. For instance, the selection condition in

```
select row: X
from biblio.paper X
where X.year > 1993
```

is understood as `exists Y in X.year ( Y > 1993 )`.

The previous syntactic extensions can be viewed as providing some form of coercion. The user compares an atomic value `1993` to a set `X.year` and the system introduces a membership test. In the same spirit, the user should not have to care about exact atomic types, for example, whether a date is stored as an integer, a string, or in a particular `date` sort known by the system. So, the semantics of the language is that the comparison of two atomic values succeeds if both values can be coerced to the same type and the comparison succeeds in this type. The comparison `X.date` > `1995` thus succeeds for a date such as `1998`—an integer—or for the string `"1998"`.

We should also mention at this point that Lorel allows both label and path variables. These are described later (Section 4.5).

```
select title: X
from   biblio.paper Z,
       Z.title X,
       Z.year Y
where  Y > 1989
```

A pattern such as `{biblio: {paper: X}}` that describes a tree consisting of a single path is said to be *linear*. We shall adopt the syntax $l_1.l_2. \ldots l_n.e$ for linear patterns $\{l_1 : \{l_2 : \ldots \{l_n : e\} \ldots\}\}$. This is a *path pattern*. Thus `biblio.paper.X in db` is equivalent to `biblio.paper X`. As an example, the following UnQL query produces the titles of papers written by `"Smith"`:

```
select title: X
where biblio.paper.{author: "Smith", title: X} in db
```

Patterns are also useful for the multiple bindings that are needed to express joins:

```
select row: {a: X, c: Z}
where r1.row.{a: X, b: Y} in db,
      r2.row.{b: Y, C: Z} in db
```

This should be compared to query `q-join`.

4.5 LABEL AND PATH VARIABLES

The use of label variables allows some interesting queries that combine what would normally be regarded as schema and data information. Label variables are used in both Lorel and UnQL. In the following we shall continue our assumption that variables are capitalized (Lorel and UnQL use different conventions for flagging label variables). Consider the queries

```
select L: X
from   biblio._*.L X
where  matches(".*Shakespear.*", X)

select new-person: (select L: Y
                    from   X.L Y
                    where  not (L = salary))
from   db.person X
```

The first of these finds all the occurrences of a string in the database matching a regular expression. It returns a tuple showing the labels that lead to a string, for example,[3]

```
{author: "William Shakespear",
 title: "Shakespeare's Tragedies", ...}
```

The second of these "hides" the `salary` attribute of an employee. Unlike a conventional query language, we do not need to know the attributes that are to be preserved in order to formulate this query.

It is also convenient to turn labels into data and vice versa.

```
select publication: {type: L, title: T}
from   biblio.L X, X.title T
where  X.Year > 1989
```

This is a commonly required transformation. Rather than have the type of a publication node described by the label on the edge that leads to the node, we give a common label `publication` to all publications and describe the type of the publication by a subsidiary `type` edge. Typical output from this query would be

```
{ publication: {type: "book",
                title: "Database Systems"},
  publication: {type: "paper",
                title: "Semistructured data"},    ... }
```

which shows how a leaf node may carry label data rather than string data.

Another way to use labels as data is to effect a "group-by" operation. The following example groups papers under edges labeled by their year of publication.

```
select Y: (select X
           from   biblio.paper X
           where  X.year = Y)
from   biblio.paper.year Y
```

Note that some year `Y` may occur in the result several times, once for each paper published that year. All occurrences will be equal (each contains all papers published that year); hence, we need to perform duplicate elimination. We will discuss duplicate elimination in Section 6.4.3.