Nested Relational and Semi-Structured Databases

Databases with JSON Objects

Dirk Van Gucht[1]

[1]Indiana University

Motivation

- In the relational database model, each relation has a schema of attributes with atomic domains such as booleans, numbers, text strings etc
- In complex-object databases, data is permitted to have objects with domains such as arrays, sets, bags, objects of composite types (rows), relations, JSON objects, etc.

Motivation

- For such databases, we need mechanisms to

  1. define domains of complex-object types

  2. search (query) and manipulate databases with complex objects

- We will focus on and contrast two kinds of databases that are recursively defined in terms of

  1. complex object types: row (record) and set (bags);[1]
  2. the JSON type.[2]

[1] Such database are called nested relational databases.
[2] Such database are called semi-structured databases.

Complex-object types and nested relations

- Complex object types are inductively defined starting from atomic types and then using the row type and set type constructors:

  1. The types int, text, boolean, etc are atomic types
  2. If $A_1, \ldots, A_k$ are distinct attribute name and $\tau_1, \ldots, \tau_k$ are atomic types or set types, then

     $$(A_1 : \tau_1, \ldots, A_k : \tau_k)$$

     is a composite type[3]
  3. If τ is a composite type then $\{\tau\}$ is a set type

- If R is a relation name and τ is a composite type, then $R : \tau$ is a nested relation of type τ[4]

---

[3] A composite type is also called a record or row type.

[4] Sometimes also called a complex-object relation of type τ.

Complex-objects domain and nested relation instances

- Complex object domains are inductively defined starting from domains of atomic types:
  1. Each atomic type τ has a domain $dom(\tau)$ of atomic values such as integers, text, boolean etc.
  2. If τ is a composite type $(A_1 : \tau_1, \ldots, A_k : \tau_k)$, then
     $$dom(\tau) = dom(\tau_1) \times \cdots \times dom(\tau_k)$$
     wherein the i-th component of an object $o \in dom(\tau)$ can be identified as $o.A_i$ for each $i \in [1, k]$
  3. If $\{\tau\}$ is a set type, then $dom(\{\tau\})$ is the set of all finite subsets of $dom(\tau)$. So each object $o \in dom(\{\tau\})$ is a finite set of tuples of type τ
- If $R : \tau$ is a nested relation, then a nested relation instance of type $R : \tau$ is a finite set of tuples in $dom(\tau)$.

Examples of complex-object types

The following are examples of complex object types:

int	atomic type
text	atomic type
boolean	atomic type
(sid: int, sname: text, birthYear: int)	composite type
{(word: text)}	set type
(sid: int, gradeInfo: {(grade: text, courses: {(cno: int)})})	composite type

For succinctness, we will often omit the atomic types:

$(sid, sname, birthYear)$	composite type
$\{(word)\}$	set type
$(sid, gradeInfo\{(grade, courses\{(cno)\})\})$	composite type

Examples of nested relations

The following are examples of types of nested relations:

Student: (sid, sname, birthYear)

documentWords: (doc, words {(word)})

studentGrades: (sid, gradeInfo {(grade, courses {(cno)})})

Course: (cno, name, students {(sid,name,majors {(major)})}, teachers {(tid, name)})

Defining complex-object types in SQL

In (object-relational) SQL,

- Composite types are defined using composite type declaration:

  CREATE TYPE studentType AS (sid int, sname text, birthYear text);

  CREATE TYPE courseType AS (cno int);

  CREATE TYPE studentType AS (sid int);

- Set types are defined using the array type constructor

CREATE TYPE gradeCoursesType AS (grade text, courses courseType[ ]);

CREATE TYPE gradeStudentsType AS (grade text, students studentType[ ]);

Defining nested relations in SQL

```
CREATE TYPE gradeCoursesType AS (grade text, courses courseType[ ]);

CREATE TYPE gradeStudentsType AS (grade text, students studentType[ ]);
```

In SQL, nested relations are defined using the CREATE TABLE statement

```
CREATE TABLE student(sid int, sname text, birthYear text);

CREATE TABLE course(cno int, cname text, dept text);

CREATE TABLE enroll(sid int, cno int, grade text);

CREATE TABLE studentGrades(sid int, gradeInfo gradeCoursesType[ ]);

CREATE TABLE courseGrades(cno int, gradeInfo gradeStudentsType[ ]);
```

Specifying nested relation instances using INSERT INTO (Example)

```
CREATE TYPE fooType AS (C int, D int);
CREATE TYPE barType AS (A int, B fooType[]);

CREATE TABLE tableTest (X int, Y barType[]);
```

```
insert into tableTest values (1, ARRAY[]::barType[]);
insert into tableTest values (2, ARRAY[(2,ARRAY[(3,4)]::fooType[])]::barType[]);
insert into tableTest values (3, ARRAY[(3,ARRAY[(4,5)]::fooType[]),(4,ARRAY[(5,6),(6,7)]::fooType[])]::barType[]);
```

We obtain the following nested relation instance:

tableTest

X	Y
1	{}
2	{(2,{(3,4)}}
3	{(3,{(4,5)}), (4,{(5,6), (6,7)})}

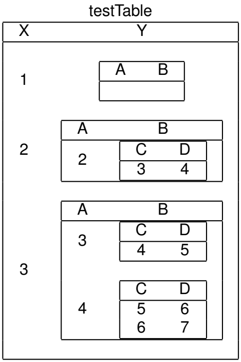
Nested relations in SQL

A better visualization of the nested relation instance is as follows:

testTable

X	Y
1	{}
2	{(2,{(3,4)}}
3	{(3,{(4,5)}), (4,{(5,6), (6,7)})}

nested relation visualization
$\rightarrow$



Nested Relation instance: JSON representation

An alternative specification of the example nested relation instance is as the following JSON object:

```
[{"X": 1,
  "Y": []},
 {"X": 2,
  "Y": [{"A": 2,
         "B":[{"C": 3,
               "D": 4}]}]},
 {"X": 3,
  "Y": [{"A": 3,
         "B":[{"C": 3,
               "D": 4}]},
        {"A": 4,
         "B":[{"C": 5,
               "D": 6},
              {"C": 6,
               "D": 7}]}]}]
]
```

Example: Populating a nested relation instance with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We can then populate the studentGrades nested relation with an INSERT INTO statement using a query with repeated grouping (nesting) and array aggregation:

```
INSERT INTO        studentGrades
WITH E AS          (SELECT sid, grade, array_agg(row(cno)::courseType) AS courses
                    FROM Enroll
                    GROUP BY (sid, grade)),

     F AS          (SELECT sid, array_agg(row(grade, courses)::gradeCoursesType) AS gradeInfo
                    FROM E
                    GROUP BY(sid));

SELECT sid, gradeInfo
FROM    F
```

Example: Populating a nested relation with an INSERT INTO Query

We have achieved the following:

Enroll

sid	cno	grade
100	200	A
100	201	B
100	202	A
101	200	B
101	201	A
102	200	B
103	201	A
101	202	A
101	301	C
101	302	A
102	202	A
102	301	B
102	302	A
104	201	D

double grouping:
group by (sid,grade)
group by (sid)
$\rightarrow$
double nesting

studentGrades

sid	gradeInfo
100	{(A,{(200),(202)}),(B,{(201)})}
101	{(B,{(200)}),(A,{(201),(202),(302)}),(C,{(301)})}
102	{(A,{(202),(302)}),(B,{(200),(301)})}
103	{(A,{(201)})}
104	{(D,{(201)})}

- For example, student 100 obtained two types of grades: ‘A’ and ‘B’
- She received an ‘A’ in courses 200 and 202, and a ‘B’ in course 201

Querying nested relations

- Just as in the case for ordinary relations, nested relations can be queried in SQL using its standard query constructs
- Because of the hierarchical structure of such relations, most queries need to navigate in accordance with this hierarchy
- Frequently, the complex-objects of set-type need to be unnested to reveal their inner data
- In addition, and depending on the output structure of the query, grouping and array (set) aggregation (nesting) needs to be applied

Example: Querying nested relations

The following query illustrates path navigation along a path in the nested relation hierarchy

- "Find the grade information of each student who received an 'A' in some course."

```
SELECT    sid, gradeInfo
FROM      studentGrades sg
WHERE     'A' IN (SELECT grade
                  FROM   UNNEST(sg.gradeInfo))
```

- Notice the UNNEST(sg.gradeInfo) operation
- This UNNEST operation permits us to access the tuples that reside in the sg.gradeInfo set object.
- Thus, in particular, we can access the grade attribute of these tuples and compare their grade components with the grade 'A'

Example: Querying nested relations

- "Find the grade information of each student who received an 'A' in some course."

```
SELECT   sid, gradeInfo
FROM     studentGrades sg
WHERE    'A' IN (SELECT grade
                 FROM   UNNEST(sg.gradeInfo))
```

- We obtain the following result:

sid	gradeInfo
100	{(A,{(200),(202)}),(B,{(201)})}
101	{(B,{(200)}),(A,{(201),(202),(302)}),(C,{(301)})}
102	{(A,{(202),(302)}),(B,{(200),(301)})}
103	{(A,{(201)})}

Example: Querying nested relations

- "Find the grade information of each student who received an 'A' in some course."
- We can express this query also as follows:

```
SELECT   sid, gradeInfo
FROM     studentGrades sg, UNNEST(sg.gradeInfo) g
WHERE    g.grade = 'A'
```

Example: Querying nested relations

The following query illustrates extraction of sub-objects inside other objects:

- "Find for each student the set of courses in which he or she received an 'A' "

```
SELECT   sg.sid, g.courses
FROM     studentGrades sg, UNNEST(sg.gradeInfo) g
WHERE    g.grade = 'A'
```

- We obtain the following result:

sid	courses
100	{(200),(202)}
101	{(201),(202),(302)}
102	{(202),(302)}
103	{(201)}

Example: Querying nested relations

The following query illustrates "path" navigation along a path in the nested relation hierarchy

- “Find the grade information of each student who enrolled in course 301."

```
SELECT   sid, gradeInfo
FROM     studentGrades sg,
           UNNEST(sg.gradeInfo) g,
             UNNEST(g.courses) c
WHERE    c.cno = 301;
```

- In this case we need to UNNEST twice since the cno information resides at the 2nd level in the hierachy
- Notice how this query follows the hierarchical structure of the studentGrades relation

Example: Querying nested relations

The following query illustrates "path" navigation along a path in the nested relation hierarchy where, in addition, conditions are checked at nodes in the path:

- “Find the grade information of each student who received a ‘B’ in course 301."

```
SELECT    sid, gradeInfo
FROM      studentGrades sg,
            UNNEST(sg.gradeInfo) g,
              UNNEST(g.courses) c
WHERE     g.grade = ‘B’ AND c.cno = 301;
```

  I.e, in studentGrades, navigate along each path with the following structure and conditions:

$$sg \rightarrow g[grade = \text{‘B’}] \rightarrow c[cno = 301]$$

  In Xpath (Unix-like) path notation:

$$sg/g[grade = \text{‘B’}]/c[cno = 301]$$

Example: Querying nested relations

The following query illustrates path navigation followed by object construction:

- "For each student, find the set of courses in which he or she is enrolled."

```
SELECT      sid, array_agg(c.cno) as courses
FROM        studentGrades sg,
              UNNEST(sg.gradeInfo) g,
                UNNEST(g.courses) c
GROUP BY    (sg.sid)
```

- The result is as follows:

sid	courses
100	{200,201,202}
101	{200,201,202,301,302}
102	{200,202,301,302}
103	{201}
104	{201}

Example: Querying nested relations

The following query illustrate path navigation, joining nested relations, object construction, and conditions checking:

“For each student who majors in ‘CS’, list his or her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should by grouped by the department in which they are offered."

sid	sname	courseInfo
101	Nick	{(CS,{(301,AI),(200,PL),(202,Dbs)}),(Math,{(201,Calculus)}),(Philosophy,{(302,Logic)})}
102	Chris	{(CS,{(200,PL),(202,Dbs),(301,AI)}),(Philosophy,{(302,Logic)})}
103	Dinska	{(Math,{(201,Calculus)})}
105	Vince	{}

Example: Nested relations restructuring

“For each student who majors in ‘CS’, list her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should by grouped by the department in which they are offered."

```
WITH E AS (SELECT sid, cno
           FROM   studentGrades sg,
                    unnest(sg.gradeInfo) g,
                       unnest(g.courses) sc),

     F AS (SELECT sid, dept, array_agg((cno,cname)) as courses
          FROM   E NATURAL JOIN Course
           GROUP  BY(sid, dept))

SELECT sid, sname, ARRAY(SELECT (dept, courses)
                         FROM    F
                         WHERE   s.sid = F.sid) AS courseInfo
FROM   student s
WHERE  sid IN (SELECT sid
              FROM  major m
              WHERE major = 'CS');
```

Example: Nested relations restructuring

"For each student who majors in 'CS', list her sid and sname, along with the courses she is enrolled in. Furthermore, these courses should by grouped by the department in which they are offered."

We obtain the following result:

sid	sname	courseInfo
101	Nick	{(CS,{(301,AI),(200,PL),(202,Dbs)}),(Math,{(201,Calculus)}),(Philosophy,{(302,Logic)})}
102	Chris	{(CS,{(200,PL),(202,Dbs),(301,AI)}),(Philosophy,{(302,Logic)})}
103	Dinska	{(Math,{(201,Calculus)})}
105	Vince	{}

JSON (JavaScript Object Notation) objects

A JSON object (self-) describing a person:

```
{ "firstName": "John",
  "lastName": "Smith",
  "age": 25,
  "isAlive": true,
  "address": { "streetAddress": "21 2nd Street",
               "city": "New York",
               "state": "NY",
               "postalCode": "10021-3100" },
  "phoneNumbers": [
    { "type": "home",
      "number": "212 555-1234" },
    { "type": "mobile",
      "number": "123 456-7890" } ],
  "children": [],
  "spouse": null
}
%
```

JSON basic, array, and null values

JSON objects are inductively defined using atomic, array, **null**, and object values:

- Atomic values:
  - Number: a signed decimal number (double-precision floating-point format)
  - Strings: a sequence of zero or more characters. Strings are delimited with double-quotation marks
  - Boolean: either of the values **true** or **false**
- Array values:
  - An ordered list of zero or more values, each of which may be of any type
  - Arrays use square bracket notation and array elements are comma-separated
- null value: An empty value, using the word **null**

JSON object values

- Object value:
  - An unordered collection of key-value pairs where the keys are strings
  - Objects are delimited with curly brackets and use commas to separate each pair, while within each pair the colon ':' character separates the key from its value
  - Since objects are intended to represent associative arrays (maps), typically each key is unique within an object

XML documents

- Just like a JSON value, an XML document is a self-describing data object
- In contrast with the predefined type-structured nature of relations and nested relations, JSON values and XML documents are examples of semi-structured data
  - In JSON, the structure is revealed in the keys of the objects
  - In XML, the structure is revealed in the labeled tags of the document

Example: XML document

- The labeled tags (in red) provide the structure of the document
- The values in the document appear in green

```
<person>
   <firstName> John </firstName>
   <lastName> Smith </lastName>
   <age> 25 </age>
   <address>
      <streetAddress> 21 2nd Street </streetAddress>
      <city> New York </city>
      <state> NY </state>
      <postalCode> 10021 </postalCode>
   </address>
   <phoneNumber>
      <type> home </type>
      <number> 212 555-1234</number>
   </phoneNumber>
   <phoneNumber>
      <type> fax </type>
      <number> 646 555-4567 </number>
   </phoneNumber>
   <gender>
      <type> male</type>
   </gender>
</person>
```

- In the remainder of this lecture, we will no longer focus on XML

Review of populating a nested relation with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We can then populate the studentGrades relation with an INSERT INTO statement with a query that use repeated grouping and array aggregation:

```
INSERT INTO         studentGrades
WITH E AS           (SELECT sid, grade, array_agg(row(cno)::courseType) AS courses
                     FROM Enroll e
                     GROUP BY (sid, grade)),

     F AS           (SELECT sid, array_agg(row(grade, courses)::gradeCoursesType) AS gradeInfo
                     FROM E
                     GROUP BY(sid))
SELECT sid, gradeInfo
FROM F;
```

Populating a JSON objects relation with an INSERT INTO Query

- We assume that data has been inserted into the Enroll relation
- We create the following table of type JSONB

```
CREATE TABLE jStudentGrades (studentInfo JSONB)
```

- We then restructure the Enroll data and insert it into this table

```
INSERT INTO    jstudentGrades
WITH E AS      (SELECT e.sid, e.grade,
                array_to_json(array_agg(json_build_object('cno',e.cno))) as courses
                FROM Enroll e
                GROUP BY (e.sid, e.grade)),

     F AS      SELECT json_build_object('sid', sid, 'gradeInfo',
                array_to_json(array_agg(json_build_object('grade', grade, 'courses', courses)))) as studentInfo
               FROM E
               GROUP BY(sid))
SELECT *
FROM F;
```

Populating a JSON objects relation with an INSERT INTO Query

We obtain the following table consisting of 5 JSON objects:

F
{ "sid" : 100, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 200},{ "cno" : 202}]},{ "grade" : "B", "courses" : [{ "cno" : 201}]}]}
{ "sid" : 101, "gradeInfo" : [{ "grade" : "B", "courses" : [{ "cno" : 200}]},{ "grade" : "A", "courses" : [{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}]},{ "grade" : "C", "courses" : [{ "cno" : 301}]}]}
{ "sid" : 102, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 202},{ "cno" : 302}]},{ "grade" : "B", "courses" : [{ "cno" : 200},{ "cno" : 301}]}]}
{ "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 201}]}]}
{ "sid" : 104, "gradeInfo" : [{ "grade" : "D", "courses" : [{ "cno" : 201}]}]}

JSON versus Nested Relation Construction Operators

```
INSERT INTO    jstudentGrades
WITH E AS      (SELECT sid, grade,
                array_to_json(array_agg(json_build_object('cno',cno))) as courses
                FROM Enroll
                GROUP BY (sid, grade)),

     F AS      SELECT json_build_object('sid', sid, 'gradeInfo',
               array_to_json(array_agg(json_build_object('grade', grade, 'courses', courses)))) as studentInfo
               FROM E
               GROUP BY(sid))
SELECT *
FROM F;
```

Operator	JSON	Operator	Nested Relations
object construction	json_build_object()	row construction	row()
array construction	array_to_json(array_agg())	array construction	array_agg()

Example: Querying the JSON relation

- "Find the grade information of each student who received an 'A' in some course."

```
SELECT   sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'
FROM     jstudentGrades sg,
           jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g
WHERE    g -> 'grade' = ' "A" ';
```

Operator	JSON	Operator	Nested Relations
extraction	jsonb_array_elements()	extraction	UNNEST()

- Notice the similarity with the query expressed on the studentGrades nested relation

```
SELECT   sid, gradeInfo
FROM     studentGrades sg, UNNEST(sg.gradeInfo) g
WHERE    g.grade = 'A'
```

Example: Querying the JSON relation

- "Find the grade information of each student who received an 'A' in some course."

```
SELECT   sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'
FROM     jstudentGrades sg,
            jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g
WHERE    g -> 'grade' = ' "A" ';
```

- We obtain the following result:

```
{ "sid" : 100,
    "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 200},{ "cno" : 202}]},{ "grade" : "B", "courses" : [{ "cno" : 201}]}]}

{ "sid" : 101,
    "gradeInfo" : [{ "grade" : "B", "courses" : [{ "cno" : 200}]},{ "grade" : "A",
       "courses" : [{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}]},{ "grade" : "C", "courses" : [{ "cno" : 301}]}]}

{ "sid" : 102,
   "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 202},{ "cno" : 302}]},{ "grade" : "B",
     "courses" : [{ "cno" : 200},{ "cno" : 301}]}]}

{ "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 201}]}]}
```

Example: Querying the JSON relation

- “Find the grade information of each student who enrolled in course 301."

```
SELECT   sg.studentInfo -> 'sid', sg.studentInfo -> 'gradeInfo'
FROM     jstudentGrades sg,
           jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,
             jsonb_array_elements(g -> 'courses') c
WHERE    (c -> 'cno')::jsonb = '301';
```

- Notice the similarity with the query expressed for the studentGrades nested relation

```
SELECT   sid, gradeInfo
FROM     studentGrades sg,
           UNNEST(sg.gradeInfo) g,
             UNNEST(g.courses) c
WHERE    c.cno = 301;
```

Example: Querying the JSON relation

- “For each student, find the set of courses in which he or she is enrolled."

```
SELECT      sg.studentInfo -> 'sid' as sid , array_to_json(array_agg(c -> 'cno')) as courses
FROM        jstudentGrades sg,
                jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,
                    jsonb_array_elements(g -> 'courses') c
GROUP BY    (sg.studentInfo -> 'sid')
```

- The result is as follows:

sid	courses
100	[200,201,202]
101	[200,201,202,301,302]
102	[200,202,301,302]
103	[201]
104	[201]

Example: Querying the JSON relation

- “For each student, find the set of courses in which he or she is enrolled."

```
SELECT      sg.studentInfo -> 'sid' as sid , array_to_json(array_agg(c -> 'cno')) as courses
FROM        jstudentGrades sg,
              jsonb_array_elements(sg.studentInfo -> 'gradeInfo') g,
                jsonb_array_elements(g -> 'courses') c
GROUP BY    (sg.studentInfo -> 'sid')
```

- Notice the similarity with the query expressed for the studentGrades nested relation

```
SELECT      sid, array_agg(c.cno) as courses
FROM        studentGrades sg,
              UNNEST(sg.gradeInfo) g,
                UNNEST(g.courses) c
GROUP BY    (sg.sid)
```

MongoDb

- MongoDb is a semi-structured data and programming model to store, search, and manipulate collections of documents
- A collection is an array of documents represented as JSON objects:

db.jstudentGrades

```
[
{ "sid" : 100,
    "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 200},{ "cno" : 202}]},{ "grade" : "B", "courses" : [{ "cno" : 201}
{ "sid" : 101,
    "gradeInfo" : [{ "grade" : "B", "courses" : [{ "cno" : 200}]},{ "grade" : "A",
       "courses" : [{ "cno" : 201},{ "cno" : 302},{ "cno" : 202}]},{ "grade" : "C", "courses" : [{ "cno" : 301}]}]},
{ "sid" : 102,
   "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 202},{ "cno" : 302}]},{ "grade" : "B",
     "courses" : [{ "cno" : 200},{ "cno" : 301}]}]},
{ "sid" : 103, "gradeInfo" : [{ "grade" : "A", "courses" : [{ "cno" : 201}]}]},
{ "sid" : 104, "gradeInfo" : [{ "grade" : "D", "courses" : [{ "cno" : 201}]}]}
]
```

MongoDb (Queries)

- Queries and updates typically take the following form

```
db.<collection>.<method>(<filter>, <options>)
```

`db`	name of the database
`collection`	name of a collection in `db`
`method`	operation on (objects in) `collection`
`filter`	conditions that selects objects in `collection`
`options`	each method has certain options for what is will do with documents that match the filter condition

MongoDb (Path queries)

- Find the student grade info for student with sid = '102':

  `db.<jstudentGrades>.find({'sid': 102})`

- Find the student grade info for students who received an 'A' in some course:

  `db.<jstudentGrades>.find({'gradeInfo.grade': 'A'})`

  Here 'gradeInfo.grade' is a path.

- Find the student grade info for students who took course 301:

  `db.<jstudentGrades>.find({'gradeInfo.courses.cno': 301})`

  Here 'gradeInfo.courses.cno' is a path.