Big Data – Exercises

Fall 2023 – Week 6 – ETH Zurich

Data Models

Reading:

- (Mandatory) Melnik, S., et al. (2011). Dremel: Interactive Analysis of Web-Scale Datasets. In CACM, 54(6). [DOI]
- (Recommended) M. Droettboom, Understanding JSON Schema [online]
- (Recommended) Harold, E. R., & Means, W. S. (2004). XML in a Nutshell. [Available in the ETH library] [online] (Chapter 17 on XML Schema, except 17.3 on namespaces)
- (Optional) White, T. (2015). Hadoop: The Definitive Guide (4th ed.). O'Reilly Media, Inc. [Available in the ETH library] [online] (Chapters 12 (Avro) and 13 (Parquet)

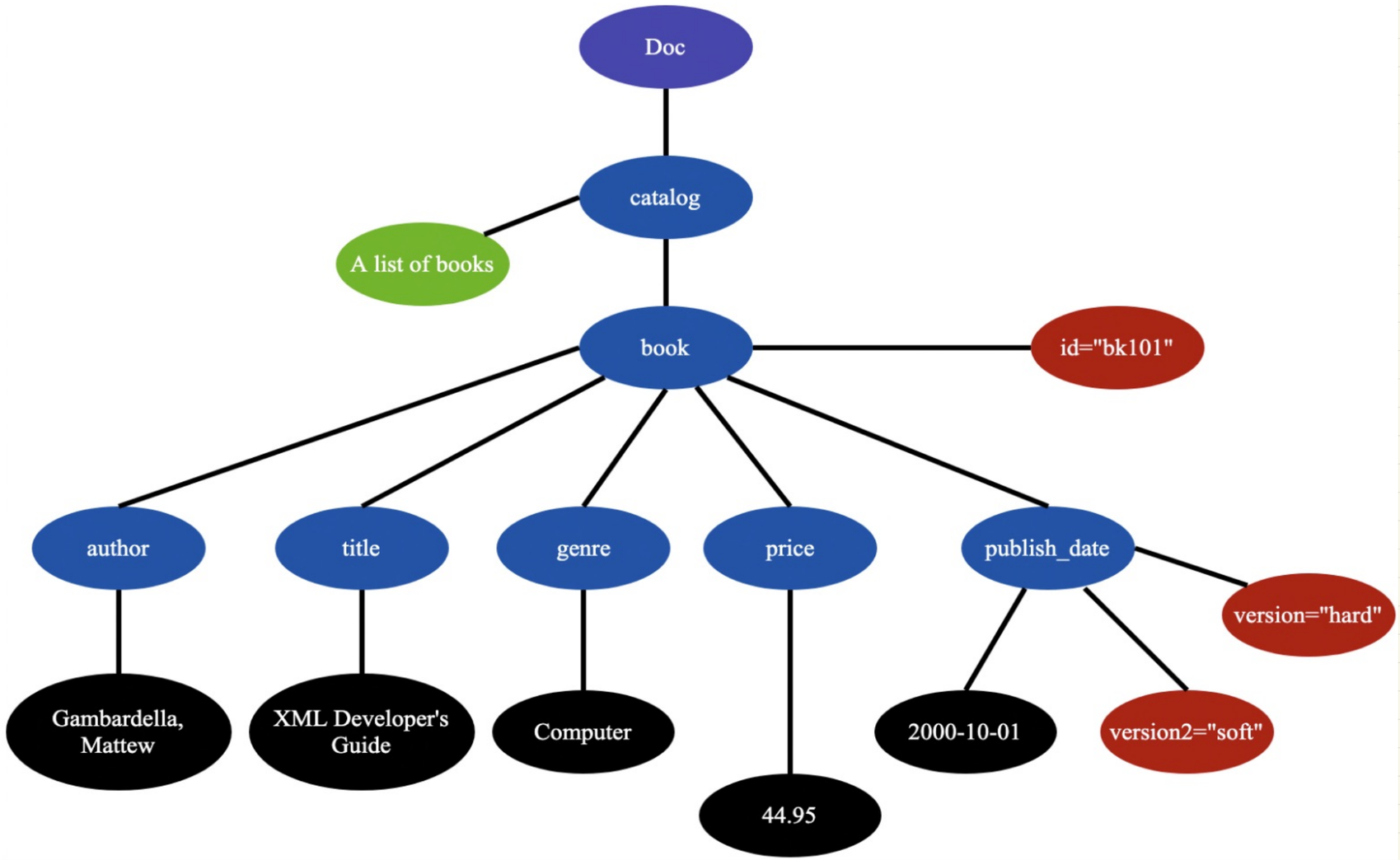
This exercise will consist of five main parts:

- XML data models
- XML Schemas
- JSON Schemas
- JSound
- Dremel

Document 2

```
<catalog>
   <!-- A list of books -->
   <book id='bk101'>
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date version='hard' version2='soft'>2000-10-01</publish_date>
   </book>
</catalog>
```

Solution



Document 1

```xml
<happiness xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="Schema.xsd"/>
```

Document 2

```xml
<happiness xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="Schema.xsd">
    <health/>
    <friends/>
    <family/>
</happiness>
```

Document 3

```xml
<happiness xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="Schema.xsd">
    3.141562
</happiness>
```

Document 4

```xml
<happiness xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="Schema.xsd">
    <health value="100"/>
    <friends/>
    <family/>
</happiness>
```

Document 5

```xml
<happiness xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:noNamespaceSchemaLocation="Schema.xsd">
    <health/>
    <friends/>
    <family/>
    But perhaps everybody defines it differently...
</happiness>
```

Schema 1

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="happiness">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="health"/>
                <xs:element name="friends"/>
                <xs:element name="family"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Schema 2

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="happiness">
        <xs:complexType mixed="true">
            <xs:sequence>
                <xs:element name="health"/>
                <xs:element name="friends"/>
                <xs:element name="family"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

Schema 3

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="happiness" type="xs:decimal"/>
</xs:schema>
```

Schema 4

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="happiness">
        <xs:complexType>
            <xs:sequence/>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

empty

Schema 5

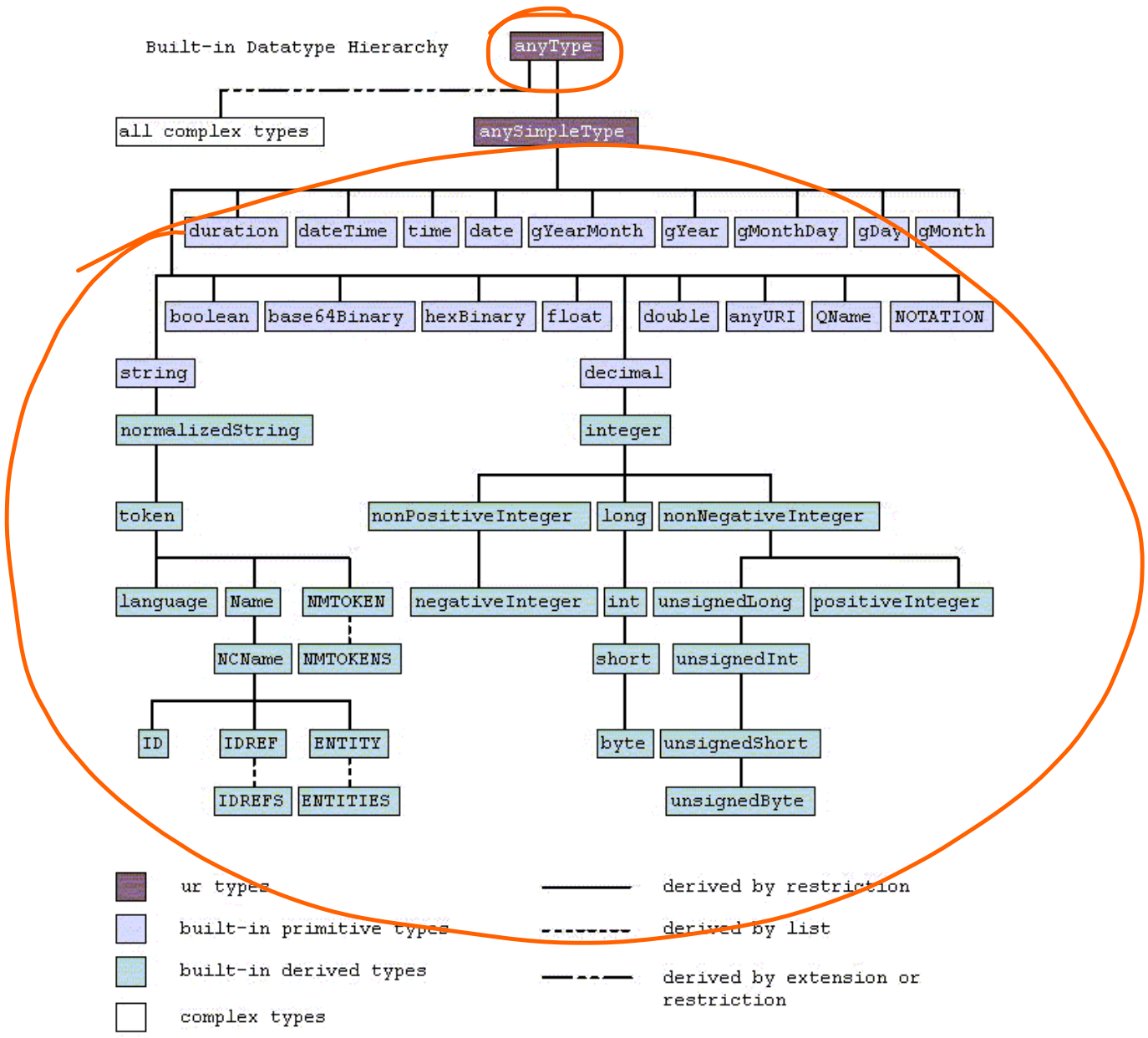
```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="happiness">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="health">
                    <xs:complexType>
                        <xs:attribute name="value" type="xs:integer" use="required"/>
                    </xs:complexType>
                </xs:element>
                <xs:element name="friends"/>
                <xs:element name="family"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>
</xs:schema>
```

XML Schema

https://cutt.ly/xml

👤 Created by: Hongyu Hè
ℹ️ Only for teaching purposes at ETH Zurich
💡 Based on an amazing explanation here: https://www.xml.com/pub/a/2001/08/22/easyschema.html

1. There are two categories of types in XML: `simpleType` and `complexType`.
   - ▼ They are the only two direct children of the root type `anyType` (for a figurative representation of the type hierarchy)



- ▼ In general, there are the following three kinds of types in the "*DataFrame type system*"
  i. Atomic types

Strings
Numbers
Booleans
Dates and Times
Time Intervals
Binaries
Null

  ii. Structure types: List, Map
  iii. Cardinality type — Sequence

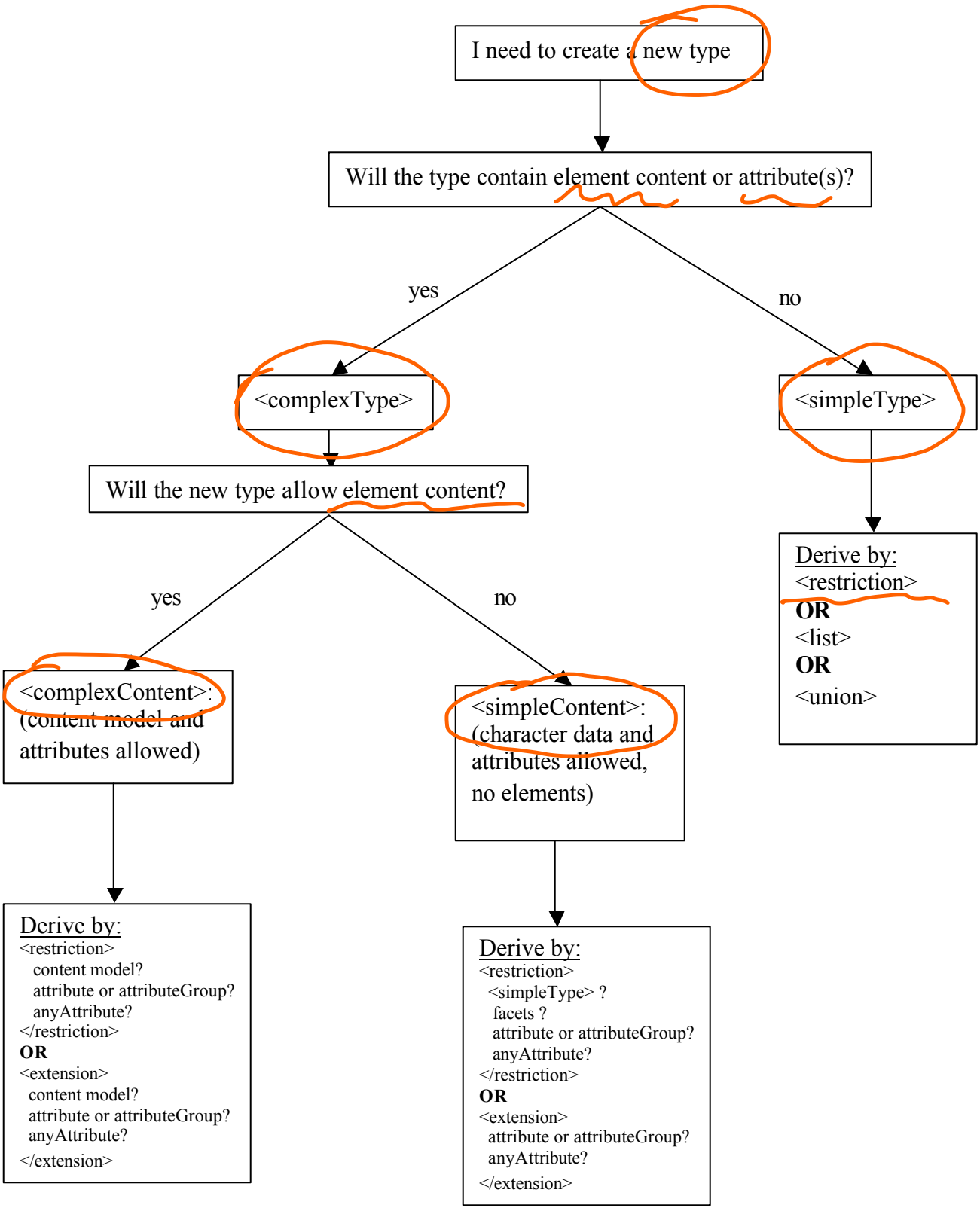
2. New types can be created from a base type by *deriving* and *extending*.
   a. By default, any type can be *derived* from the base type: `anyType` 🙂
      i. There are four ways to derive a new type: *restriction*, *extension, list,* and *union*.
   b. *Extending* any type will always result in a `complexType`.

   (See here for a graphical decision tree)

3. `simpleType` ⇒ user-defined `simpleType`

ℹ️ Following examples may contain an arbitrary namespace: `dc:`

Schema Type Syntax Decision Tree



▼ Via restriction

```
<!--Define type-->
<simpleType name="myNameType">
  <restriction base="string">
    <enumeration value="Don Smith" />
  </restriction>
</simpleType>

<!--Associate it with an element-->
<element name="employee" type="dc:myNameType" />
<!--Use it-->
<dc:employee>Don Smith</dc:employee>
```

4. `complexType` has two subtypes: (1) `complexType + simpleContent` and (2) `complexType + complexContent`

   ▼ `complexType + simpleContent`: By adding **only** *attributes* to a `simpleType` will create this type.

```
<!--Define type-->
<complexType name="myNewNameType">
  <simpleContent>
    <extension base="dc:myNameType">

     <attribute name="position" type="string" />

    </extension>
  </simpleContent>
</complexType>

<!--Associate it with an element-->
<element name="employee" type="dc:myNewNameType" />
<!--Use it-->
<dc:employee position="postdoc">Don Smith</dc:employee>
```

   ▼ `complexType + complexContent`: Besides attributes, if we **also** add children *elements*, this creates this type.

```
<!--Define type-->
<complexType name="myNewNameType">
  <complexContent>
    <restriction base="anyType">

      <sequence>
        <element name="name" type="string" />
        <element name="location" type="string" />
      </sequence>

      <attribute name="position" type="string" />

    </restriction>
  </complexContent>
</complexType>

<!--Associate it with an element-->
<element name="employee" type="dc:myNewNameType" />
<!--Use it-->
<dc:employee position="postdoc">
<dc:name>Don Smith</dc:name>
<dc:location>Dallas, TX</dc:location>
</dc:employee>
```

   ▼ Default setting: `complexType + complexContent + <restriction base="anyType">`, so below is the equivalent of the above.

```
<complexType name="myNewNameType">
  <sequence>
    <element name="name" type="string" />
    <element name="location" type="string" />
  </sequence>
  <attribute name="position" type="string" />
</complexType>
```

5. Empty type is `complexType + complexContent`

   a. Both `simpleType` and `complexType + simpleContent` allow data/textual contents.

   b. But `complexType + complexContent` does not allow data/textual contents by default.

mixed = true

i. It's *optional* to have elements.

ii. It's possible to have data/textual contents through `<xs:complexType mixed="true">`

- But even with the attribute, the data/textual contents are still **optional**.

▼ Example

```
<!--Full version-->
<complexType name="processingHook">
  <complexContent>
    <restriction base="anyType">
    </restriction>
  </complexContent>
</complexType>

<!--(Simplified) using the default-->
<complexType name="processingHook">
</complexType>

<!--Use it-->
<element name="callMyApp" type="dc:processingHook" />
```

6. Nitty-gritty

a. If an element is defined to be empty by the schema, then it has to be in the instances.

b. The `sequence` element

i. **It has default attributes value `minOccurs` and `maxOccurs` to be 1.**

ii. The order in which the elements appear has to be obeyed by the instances.

c. Additional attributes can appear in `complexType + *` without defining them in advance.

d. Optional/additional elements are NOT allowed by XML schema due to the Unique Particle Attribution rule.

e. `ref` vs `type` here

i. `ref` copy-pastes existing **non-type** definitions, i.e., element, attribute, etc.

f. It's possible to uniquely identify an element within a document (see).

Task 2.3

Continuing the topic of the Great Language Game, provide an XML Schema which will validate the following document:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<attempts>
    <attempt country="AU" date="2013-08-19">
        <voiceClip>48f9c924e0d98c959d8a6f1862b3ce9a</voiceClip>
        <choices>
            <choice>Maori</choice>
            <choice>Mandarin</choice>
            <choice>Norwegian</choice>
            <choice>Tongan</choice>
        </choices>
        <target>Norwegian</target>
        <guess>Norwegian</guess>
    </attempt>
    <attempt country="US" date="2014-03-01">
        <voiceClip>5000be64c8cc8f61dda50fca8d77d307</voiceClip>
        <choices>
            <choice>Finnish</choice>
            <choice>Mandarin</choice>
            <choice>Scottish Gaelic</choice>
            <choice>Slovak</choice>
            <choice>Swedish</choice>
            <choice>Thai</choice>
        </choices>
        <target>Slovak</target>
        <guess>Slovak</guess>
    </attempt>
    <attempt country="US" date="2014-03-01">
        <voiceClip>923c0d6c9e593966e1b6354cc0d794de</voiceClip>
        <choices>
            <choice>Hungarian</choice>
            <choice>Sinhalese</choice>
            <choice>Swahili</choice>
        </choices>
        <target>Hungarian</target>
        <guess>Sinhalese</guess>
    </attempt>
</attempts>
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

    <!-- lets declare first some custom types -->
    <xs:complexType name="choicesType">
        <xs:sequence>
            <xs:element name="choice" minOccurs="1" maxOccurs="unbounded" type="xs:string"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="attemptType">
        <xs:sequence>
            <xs:element name="voiceClip" type="xs:string"/>
            <xs:element name="choices" type="choicesType"/>
            <xs:element name="target" type="xs:string"/>
            <xs:element name="guess" type="xs:string"/>
        </xs:sequence>
        <xs:attribute name="country" type="xs:string"/>
        <xs:attribute name="date" type="xs:date"/>
    </xs:complexType>

    <!-- lets declare the elements appearing in the document -->
    <xs:element name = "attempts">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="attempt" type="attemptType" maxOccurs="unbounded"/>
            </xs:sequence>
        </xs:complexType>
    </xs:element>

</xs:schema>
```

3. JSON Schemas

JSON Schema is a vocabulary that allows you to annotate and validate JSON documents. It is used to:

- Describe your existing data format(s).
- Provide clear human- and machine- readable documentation.
- Validate data, i.e., automated testing, ensuring quality of client submitted data.

Fields are optional by default! "required": [ ]

Task 3.1

Provide an JSON Schema which will validate the following document.

```
{
  "firstName": "John",
  "lastName": "Doe",
  "age": 21
}
```

Solution

```
{
  "$id": "https://example.com/person.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Person",
  "type": "object",
  "properties": {
    "firstName": {
      "type": "string",
      "description": "The person's first name."
    },
    "lastName": {
      "type": "string",
      "description": "The person's last name."
    },
    "age": {
      "description": "Age in years which must be equal to or greater than zero.",
      "type": "integer",
      "minimum": 0
    }
  }
}
```

Task 3.3

Based on the given Json schema, can you give an instance of it? HINT: We defined array of things.

```
{
  "$id": "https://example.com/arrays.schema.json",
  "$schema": "http://json-schema.org/draft-07/schema#",
  "description": "A representation of a person, company, organization, or place",
  "type": "object",
  "properties": {
    "fruits": {
      "type": "array",
      "items": {
        "type": "string"
      }
    },
    "vegetables": {
      "type": "array",
      "items": { "$ref": "#/definitions/veggie" }
    }
  },
  "definitions": {
    "veggie": {
      "type": "object",
      "required": [ "veggieName", "veggieLike" ],
      "properties": {
        "veggieName": {
          "type": "string",
          "description": "The name of the vegetable."
        },
        "veggieLike": {
          "type": "boolean",
          "description": "Do I like this vegetable?"
        }
      }
    }
  }
}
```

Tip: You can refer to a definition elsewhere with `$ref`

For example, here is an example JSON schema `schema2.json` including **cross-schema references** to types in `schema1.json`, a schema in the same directory, and **in-schema references** to types and properties in the same schema:

```
{
  "$id": "http://release.niem.gov/niem/niem-core/4.0/",
  "$schema": "http://json-schema.org/draft-04/schema#",
  "type": "object",
  "additionalProperties": false,
  "definitions": {
    "nc:Amount": {
      "$ref": "./schema1.json#/definitions/xs:decimal",
      "description": "An amount of money."
    },
    "nc:AmountType": {
      "additionalProperties": false,
      "description": "A data type for an amount of money.",
      "properties": {
        "nc:Amount": {
          "description": "An amount of money.",
          "$ref": "#/definitions/nc:Amount"
        },
        "nc:CurrencyCode": {
          "description": "A unit of money or exchange.",
          "$ref": "#/definitions/nc:CurrencyCode"
        },
      },
    },
    "nc:CurrencyCode": {
      "$ref": "./schema1.json#/definitions/xs:token",
      "description": "A unit of money or exchange."
    }
  },
  "properties": {
    "nc:ItemValueAmount": {
      "$ref": "#/definitions/nc:AmountType",
      "description": "A monetary value of an item."
    }
  },
  "required": ["nc:ItemValueAmount"]
}
```

Dremel : Two F's

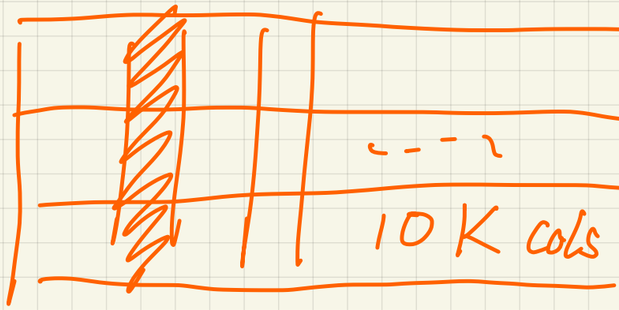
* Facts: ① Backend of Google BigQuery.
② Petabyte-scale NoSQL search engine.
③ "Constant" (!) execution time regardless of data size.
④ Faster than MapReduce on the peta-scale.
⑤ Powers Google Analytics, AdSense, etc.

* Features:
① Data model : Protocol buffers v1 (protobuf)
⇒ Today!
② Query execution : Server tree execution plan
⇒ CMU Advanced Database Systems Spring '23 Lecture 19

Protobuf

* Objectives:

Why? → Wide column stores



① Convert JSON-like data (tables w/ nested, missing fields)

to columnar representation

② Convert ↓ back to the original format

⇓

Lossless, i.e., preserve missing fields (NULL)

⇓

Blow up the columnar tables back!

* How to do ①?

Explain by an example from the paper

Example from the Dremel paper

```
DocId: 10                          D1
Links
  Forward: 20
  Forward: 40
  Forward: 60
Name
  Language
    Code: 'en-us'
    Country: 'us'
  Language
    Code: 'en'
  Url: 'http://A'
Name
  Url: 'http://B'
Name
  Language
    Code: 'en-gb'
    Country: 'gb'
```

```
DocId: 20                          D2
Links
  Backward: 10
  Backward: 30
  Forward: 80
Name
  Url: 'http://C'
```

NULL

```
message Document {
  required int64 DocId;
  optional group Links {
    repeated int64 Backward;
    repeated int64 Forward; }
  repeated group Name {
    repeated group Language {
      required string Code;
      optional string Country; }
    optional string Url; }}
```

root

Name . Language . ~~Code~~ (r, r)

	Value	r	d
D1	en-us	0	2
	en	2	2
	NULL	1	1
	en-gb	1	2
D2	NULL	0	1

Name . Language . Country (r, r)

	Value	r	d
D1	us	0	3
	NULL	2	2
	NULL	1	1
	gb	1	3
D2	NULL	0	1

Document . Name . Url (r)

	value	r	d
D1	//A	0	2
	//B	1	2
	NULL	1	1
D2	//C	0	2

(1), (2) → Index

Document . Links . Forward (r)

	value	r	d
D1	20	0	2
	40	1	2
	60	1	2
D2	80	0	2

Document . Links . Backward (r)

	value	r	d
D1	NULL	0	1
D2	10	0	2
	30	1	2

Solution

DocId

value	r	d
10	0	0
20	0	0

Name.Url

value	r	d
http://A	0	2
http://B	1	2
NULL	1	1
http://C	0	2

Links.Forward

value	r	d
20	0	2
40	1	2
60	1	2
80	0	2

Links.Backward

value	r	d
NULL	0	1
10	0	2
30	1	2

Name.Language.Code

value	r	d
en-us	0	2
en	2	2
NULL	1	1
en-gb	1	2
NULL	0	1

Name.Language.Country

value	r	d
us	0	3
NULL	2	2
NULL	1	1
gb	1	3
NULL	0	1

Protobuf : Messy data → Columnar tables

① Write down the full path to the field (e.g., Doc.FieldA.FieldB)

② Ignore "required" fields — Why? Forced by schemas. ⇒ Definitely there!

③ Index the remaining fields (e.g., ~~Doc~~.FieldA.FieldB.~~FieldC~~)

④ Index/mark "repeated" fields

#1 = r

r

1 2

d=2

Repetition Level : Index of the last duplicated "repeated" field.

Definition Level : Index of the last defined (non-required) field.

optional / repeated

```
message Catalog {
    repeated group Book {
        required string Title;
        repeated string Author;
        optional uint32 Year;
        repeated group Version {
            required string Type;
            optional uint32 Pages;
        }
    }
}
```

Use this to convert the following message to column storage:

```
Book
    Title: "Because Internet"
    Author: "McCulloch, Gretchen"
    Year: 2019
    Version
        Type: "soft"
Book
    Title: "Hitchhiker's Guide to the Galaxy"
    Author: "Adams, Douglas"
    Year: 1979
Book
    Title: "XML in a Nutshell"
    Author: "Harold, Elliotte Rusty"
    Author: "Means, Scott W."
    Year: 2005
    Version
        Type: "soft"
        Pages: 718
```

Catalog.Book.Title

Catalog.Book.Author

Catalog.Book.Year

Catalog.Book.Version.Type

Catalog.Book.Version.Pages

Solution

Catalog.Book.Title

Because Internet	0	1
Hitchhiker's Guide to the Galaxy	1	1
XML in a Nutshell	1	1

Catalog.Book.Author

McCulloch, Gretchen	0	2
Adams, Douglas	1	2
Harold, Elliotte Rusty	1	2
Means, Scott W.	2	2

Catalog.Book.Year

2019	0	2
1979	1	2
2005	1	2

Catalog.Book.Version.Type

soft	0	2
NULL	1	1
soft	1	2

Catalog.Book.Version.Pages

NULL	0	2
NULL	1	1
718	1	3