# Semi-Structured Data: XML and JSON

**Instructor: Shel Finkelstein**

*Reference:*
*A First Course in Database Systems, 3rd edition,*
*Chapter 11.1-11.3, some of 11.4. a little of Chapter 12 (just for XML)*
*XML Slides from Prof. Jeffrey Ullman, Stanford University*

# Important Notices (Final)

CMPS 182 Final Exam is on **Monday June 10, 4:00 – 7:00pm**, in our usual classroom.

- **No** early/late Finals, **no** make-up Finals.
- **No** devices.
- Includes a Multiple Choice Section and a Longer Answers Section.
  - Bring Red Scantron sheets (ParSCORE form number f-1712) sold at Bookstore, and #2 pencils for Multiple Choice Section.
    - Ink and #3 pencils don't work.
- Covers entire quarter, with slightly greater emphasis on second half of quarter.
- You may bring in one double-sided 8.5 by 11 sheet, with anything that you can read unassisted printed or written on both sides of the paper.
  - **No sharing** of sheets is permitted.
  - Include name on top right of sheet. Sheets will be collected with Finals.
- You **must** show your UCSC ID at end of Final.
- Practice Final from Spring 2017 (2 Sections) is on Piazza under Resources-->Exams
  - Answers have also been posted (one file) in the same location..

# Important Notices

- Gradiance #5 was assigned on Tuesday, May 28, and is due on **Friday, June 7 by 11:59pm**.
  - You have enough information to complete this Gradiance Assignment.
  - Some of the questions may be difficult.
- There will be Lab Sections during the last week of classes.
  - These Lab Sections are an opportunity to get help with Gradiance #5, and also ask questions about Lab4 solution, Practice Final, and overall course material.
- Might have time to do a very brief review of one topic on Friday, June 7.
  - Probably will discuss BCNF and 3NF question from Practice Final.
- Spring 2019 Student Experience of Teaching Surveys - SETs are now open, and SETs close on Sunday June 9 at 11:59pm.
  - SETs is the new term for campus-wide student course evaluations.
  - Instructors **are not** able to identify individual responses.
  - Constructive responses help improve future courses.
- See Small Group Tutoring website for LSS Tutoring with Chandler Hawkins.

# Semi-Structured Data Models

- In the relational database management system, a schema must be defined *before* data can be stored.
  - Schema is known to the query processor.
  - Exploited to derive efficient implementations to access and update data.
- In a semi-structured data model (e.g., XML and JSON), a schema need not be defined prior to "data creation".
  - Flexible data model as the schema need not be defined ahead of time, and there may not be a structured schema associated with the data.
  - Semi-structured data tends to be "self-describing".
  - Also tends to be hierarchical.
  - Non-First Normal Form

# HyperText Markup Language (HTML)

- Lingua franca for publishing hypertext on the World Wide Web.
- Designed to describe how a Web browser should arrange text, images and push-buttons on a page.
- Easy to learn, but does not convey structure.
- Fixed tag set.

Opening tag

Text (PCDATA)

```
<HTML>
<HEAD><TITLE>Welcome to the XML course</TITLE></HEAD>
<BODY>

<H1>Introduction</H1>
<IMG SRC=”dragon.jpeg" WIDTH="200" HEIGHT="150” >

</BODY>
</HTML>
```

Closing tag

Attribute name

Attribute value

“Bachelor” tag

# The Structure of XML

- XML consists of *tags* and *text*

- Tags come in pairs <date> ...</date>

- They must be properly nested

<date> <day> ... </day> ... </date> --- good

<date> <day> ... </date>... </day> --- bad

(You can't do <i> ... <b> ... </i> ...</b> in HTML)

# Well-Formed XML

- Start the document with a *declaration*, surrounded by
  <?xml … ?> .
- Normal declaration is:
  <?xml version = ”1.0” standalone = ”yes” ?>
  - “standalone” = “no Data Type Definition (DTD) provided”
- The document starts with a *root tag* that surrounds nested tags.

# <Tags>

- Tags are normally matched pairs, as <FOO> … </FOO>.
- XML tags are case-sensitive.
  - E.g., <FOO> … </foo> does not match.
- Tags may be nested arbitrarily.
- XML has only one basic type, which is text.

# Example: Well-Formed XML

```
<?xml version = “1.0” standalone = “yes” ?>
<BARS>
  <BAR><NAME>Joe’s Bar</NAME>
    <BEER><NAME>Bud</NAME>
      <PRICE>2.50</PRICE></BEER>
    <BEER><NAME>Miller</NAME>
      <PRICE>3.00</PRICE></BEER>
  </BAR>
  <BAR> ...
</BARS>
```

A NAME subelement

Root tag

A BEER subelement

Tags surrounding a BAR element

# More Terminology

- The segment of an XML document between an opening and a corresponding closing tag is called an *element*.

```
<person>
      <name> Benedict Cumberbatch </name>
  <tel> (831) 898 4321 </tel>
  <tel> (831) 898 1099 </tel>
  <email> bcumberbatch@ucsc.edu </email>
</person>
```

element

element,
a sub-element of

not an element

# Using XML to Specify a Tuple

```
<person>
  <name> Benedict Cumberbatch</name>
  <tel> (831) 898 4321 </tel>
  <email> bcumberbatch@ucsc.edu </email>
</person>
```

# Using XML to Specify a List

- We can represent a list by using the *same* tag repeatedly:

```
<addresses>
  <person> ... </person>
  <person> ... </person>
  <person> ... </person>
  ...
</addresses>
```

# Example:
# Two Ways of Representing a DB

projects:

| title | budget | managedBy |
| --- | --- | --- |
| | | |

employees:

| name | ssn | age |
| --- | --- | --- |
| | | |

# Project and Employee Relations in XML

```xml
<db>
  <project>
    <title> Pattern recognition </title>
    <budget> 10000 </budget>
    <managedBy> Joe </managedBy>
  </project>
  <employee>
    <name> Joe </name>
    <ssn> 344556 </ssn>
    <age> 34 < /age>
  </employee>
  <employee>
    <name> Sandra </name>
    <ssn> 2234 </ssn>
    <age> 35 </age>
  </employee>
  <project>
    <title> Auto guided vehicle </title>
    <budget> 70000 </budget>
    <managedBy> Sandra </managedBy>
  </project>
  :
</db>
```

Way 1: Projects and employees are intermixed.

# Project and Employee Relations in XML (cont’d)

```xml
<db>
  <projects>
    <project>
        <title> Pattern recognition </title>
        <budget> 10000 </budget>
        <managedBy> Joe </managedBy>
    </project>
    <project>
        <title> Auto guided vehicles </title>
        <budget> 70000 </budget>
        <managedBy> Sandra </managedBy>
    </project>
   :
  </projects>
  <employees>
    <employee>
      <name> Joe </name>
      <ssn> 344556 </ssn>
      <age> 34 </age>
    </employee>
    <employee>
      <name> Sandra </name>
      <ssn> 2234 </ssn>
      <age>35 </age>
    </employee>
  :
  </employees>
</db>
```

Way 2: Employees follow projects.

# Attributes

- An (opening) tag may contain *attributes*. These are typically used to describe the content of an element.
- Attributes cannot be repeated within a tag.

```
<entry>
  <word language = “en”> cheese </word>
  <word language = “fr”> fromage </word>
  <word language = “ro”> branza </word>
  <meaning> A food made … </meaning>
</entry>
```

# Attributes (cont'd)

- Another common use for attributes is to express dimension or type.

```
<picture>
  <height dim= "cm"> 2400 </height>
  <width dim= "in"> 96 </width>
  <data encoding = "gif" compression = "zip">
    M05-.+C$@02!G96YEFEC ...
  </data>
</picture>
```

- A document that obeys the "nested tags" rule and does not repeat an attribute within a tag is said to be *well-formed* .

# Using IDs and IDRefs

```
<family>
    <person  id="jane" mother="mary" father="john">
        <name> Jane Doe </name>
    </person>
    <person id="john" children="jane jack">
        <name> John Doe </name>
    </person>
    <person id="mary" children="jane jack">
        <name> Mary Doe </name>
    </person>
        <person  id="jack" mother="mary" father="john">
        <name> Jack Doe </name>
    </person>
</family>
```

# An Example

```
<db>
  <movie id=“m1”>
    <title>Waking Ned Divine</title>
    <director>Kirk Jones III</director>
    <cast idrefs=“a1 a3”></cast>
    <budget>100,000</budget>
  </movie>
  <movie id=“m2”>
    <title>Dragonheart</title>
    <director>Rob Cohen</director>
    <cast idrefs=“a2 a9 a21”></cast>
    <budget>110,000</budget>
  </movie>
  <movie id=“m3”>
    <title>Moondance</title>
    <director>Dagmar Hirtz</director>
    <cast idrefs=“a1 a8”></cast>
    <budget>90,000</budget>
  </movie>
  :
  <actor id=“a1”>
    <name>David Kelly</name>
    <acted_In idrefs=“m1 m3 m78” >
    </acted_In>
  </actor>
  <actor id=“a2”>
     <name>Sean Connery</name>
     <acted_In idrefs=“m2 m9 m11”>
     </acted_In>
     <age>68</age>
  </actor>
  <actor id=“a3”>
      <name>Ian Bannen</name>
      <acted_In idrefs=“m1 m35”>
      </acted_In>
  </actor>
   :
</db>
```

# DTD Structure

<!DOCTYPE <root tag> [

<!ELEMENT <name>(<components>)>

. . . more elements . . .

]>

# Document Type Descriptors

- Document Type Descriptors (DTDs) impose structure on an XML document, much like relation schemas impose a structure on relations.
- The DTD is just a *syntactic* specification.
  - Not a semantic specification

# Example:  Address Book

```
<person>
    <name> MacNiel, John </name>
    <greet> Dr. John MacNiel </greet>
    <addr>1234 Huron Street </addr>
    <addr> Rome, OH 98765 </addr>
    <tel> (321) 786 2543 </tel>
    <fax> (321) 786 2543 </fax>
    <tel> (321) 786 2543 </tel>
    <email> jm@abc.com </email>
</person>
```

Exactly one name

At most one greeting

As many address lines as needed (in order)

Mixed telephones and faxes

As many emails as needed

# Specifying the Structure

The structure of a person entry can be specified by:

name, greet?, addr*, (tel | fax)*, email*

XML uses a form of Regular Expression (described later).

# A DTD for Address Book

```
<!DOCTYPE addressbook [
  <!ELEMENT addressbook (person*)>
  <!ELEMENT person
     (name, greet?, address*, (fax | tel)*, email*)>
  <!ELEMENT name        (#PCDATA)>
  <!ELEMENT greet       (#PCDATA)>
  <!ELEMENT address     (#PCDATA)>
  <!ELEMENT tel         (#PCDATA)>
  <!ELEMENT fax         (#PCDATA)>
  <!ELEMENT email       (#PCDATA)>
]>
```

“Parsed Character Data” (i.e., text)

# Our Relational DB Revisited

projects:

| title | budget | managedBy |
|---|---|---|
| | | |

employees:

| name | ssn | age |
|---|---|---|
| | | |

# Two Potential DTDs for that Relational DB

```
<!DOCTYPE db [
   <!ELEMENT db          (projects, employees)>
   <!ELEMENT projects    (project*)>
   <!ELEMENT employees (employee*)>
   <!ELEMENT project     (title, budget, managedBy)>
   <!ELEMENT employee    (name, ssn, age)>
   ...
]>
```

```
<!DOCTYPE db [
   <!ELEMENT db         (project | employee)*>
   <!ELEMENT project    (title, budget, managedBy)>
   <!ELEMENT employee (name, ssn, age)>
   ...
]>
```

# Summary of XML Regular Expressions

- A The tag A occurs
- e1,e2 The expression e1 followed by e2
- e* 0 or more occurrences of e
- e? Optional -- 0 or 1 occurrences
- e+ 1 or more occurrences
- e1 | e2 either e1 or e2
- (e) grouping, e.g.,

  <!ELEMENT Address Street, (City | Zip)>

# Specifying Attributes in the DTD

- Bars can have an attribute `kind`, a character string describing the bar.

```
<!ELEMENT BAR (NAME, BEER*)>
   <!ATTLIST BAR kind CDATA   #IMPLIED>
```

Character string type; no tags

Attribute is <u>optional</u>, as opposed to: #REQUIRED

# Example of Attribute Use

- In a document that allows BAR tags, we might see:

```
<BAR kind = ”sushi”>
   <NAME>Homma’s</NAME>
   <BEER><NAME>Sapporo</NAME>
       <PRICE>5.00</PRICE></BEER>
   ...
</BAR>
```

# Specifying ID and IDREF Attributes in a DTD

```
<!DOCTYPE family [
  <!ELEMENT family   (person)*>
  <!ELEMENT person  (name)>
  <!ELEMENT name    (#PCDATA)>
  <!ATTLIST person
          id          ID       #REQUIRED
          mother   IDREF   #IMPLIED
          father    IDREF   #IMPLIED
          children  IDREFS  #IMPLIED>
]>
```

id is an ID attribute

# An XML Document That Conforms to the DTD

```
<family>
    <person  id="jane"  mother="mary" father="john">
        <name> Jane Doe </name>
    </person>
    <person id="john" children="jane jack">
        <name> John Doe </name>
    </person>
    <person id="mary" children="jane  jack">
        <name> Mary Doe </name>
    </person>
        <person  id="jack"  mother=”mary" father="john">
        <name> Jack Doe </name>
    </person>
</family>
```

# Consistency of ID and IDREF Attribute Values

- ID stands for identifier. The values across all IDs must be distinct.

- IDREF stands for identifier reference. If an attribute is declared as IDREF, then …
  - the associated value must exist as the value of some ID attribute (i.e., no dangling "pointers").

- IDREFS specifies "several" (0 or more) identifiers.

- IDREFs are a lot like Foreign Keys … except that IDREFs don't have data types!

# movieschema.dtd

```
<!DOCTYPE db [
  <!ELEMENT   db        (movie+, actor+)>
  <!ELEMENT   movie     (title, director, cast, budget)>
    <!ATTLIST           movie   id   ID   #REQUIRED>
  <!ELEMENT   title     (#PCDATA)>
  <!ELEMENT   director  (#PCDATA)>
  <!ELEMENT   cast      EMPTY>
    <!ATTLIST cast      idrefs   IDREFS   #REQUIRED>
  <!ELEMENT   budget    (#PCDATA)>
```

# moviechema.dtd (cont'd)

```
  <!ELEMENT   actor       (name, acted_In, age?, directed*)>
  <!ATTLIST   actor       id      ID       #REQUIRED>
  <!ELEMENT   name        (#PCDATA)>
  <!ELEMENT   acted_In    EMPTY>
     <!ATTLIST  acted_In          idrefs   IDREFS  #REQUIRED>
  <!ELEMENT   age         (#PCDATA)>
  <!ELEMENT   directed    (#PCDATA)>
]>
```

# Connecting the Document with its DTD

- In line:

```
<?xml version="1.0"?>
<!DOCTYPE db [<!ELEMENT ...> ... ]>
<db> ... </db>
```

Includes everything from movieschema.dtd

- Another file:

```
<!DOCTYPE db SYSTEM ”movieschema.dtd">
```

Note word SYSTEM

- A URL:

```
<!DOCTYPE db SYSTEM
        "http://www.schemaauthority.com/movieschema.dtd">
```

# First Example

```
<?xml version = “1.0” standalone = “no” ?>
<!DOCTYPE BARS [
    <!ELEMENT BARS (BAR*)>
    <!ELEMENT BAR (NAME, BEER+)>
    <!ELEMENT NAME (#PCDATA)>
    <!ELEMENT BEER (NAME, PRICE)>
    <!ELEMENT PRICE (#PCDATA)>
]>
<BARS>
    <BAR><NAME>Joe’s Bar</NAME>
      <BEER><NAME>Bud</NAME> <PRICE>2.50</PRICE></BEER>
      <BEER><NAME>Miller</NAME> <PRICE>3.00</PRICE></BEER>
    </BAR>
    <BAR> …
</BARS>
```

“no” means that there is a DTD

The DTD

The document

# Second Example

- Assume the BARS DTD is in file bar.dtd.

```
<?xml version = “1.0” standalone = “no” ?>
<!DOCTYPE BARS SYSTEM ”bar.dtd”>
<BARS>
    <BAR><NAME>Joe’ s Bar</NAME>
     <BEER><NAME>Bud</NAME>
          <PRICE>2.50</PRICE></BEER>
     <BEER><NAME>Miller</NAME>
          <PRICE>3.00</PRICE></BEER>
    </BAR>
    <BAR> …
</BARS>
```

Get the DTD from the file bar.dtd

# Well-Formed and Valid Documents

- We say that an XML document is *well-formed* if the document (with or without an associated DTD) has proper nesting of tags and the attributes of every element are all unique.

- We say that an XML document x is *valid* with respect to a DTD D if x conforms to D. That is, if the document x conforms to the regular expression grammar and constraints given by D.

# DTDs versus Schemas (or Types)

- By database (or programming language) standards DTDs are rather weak specifications.
  - Only one base type -- PCDATA
  - No useful "abstractions" e.g., no sets
  - IDREFs are untyped. They allow you to reference something, but you don't know what!
  - Few constraints. E.g., "Local keys" as opposed to global IDs.
  - Tag definitions are *global.*

- XML Schema:
  - An extension of DTDs that allows one to impose a schema or type on an XML document.

# XML Schema

- A more powerful way to describe the structure of XML documents.

- XML-Schema declarations are themselves XML documents.
  - They describe "elements" and the things doing the describing are also "elements".
  - See textbook, Section 11.4.

# Query Languages for XML

- XPath: Language for navigating through an XML document.
  - See textbook, Section 12.1.
- XQuery: Query language for XML, similar in power to SQL.
  - See textbook, Section 12.2.
- XSLT: Language for extracting information from an XML document and transforming it.
  - See textbook, Section 12.3.

# JSON: The Basics

Jeff Fox
@jfox015

**Built in Fairfield County:**
**Front End Developers Meetup**
**Tues. May 14, 2013**

# What is JSON?

# JSON is…

- A lightweight text based data-interchange format

- Completely language independent

- Based on a subset of the JavaScript Programming Language

- Easy to understand, manipulate and generate

# JSON is NOT…

- Overly Complex
- A “document” format
- A markup language
- A programming language

# Why use JSON?

- Straightforward syntax
- Easy to create and manipulate
- Can be natively parsed in JavaScript using **eval()**
- Supported by all major JavaScript frameworks
- Supported by most backend technologies

# JSON vs. XML

# Much Like XML


- Plain text formats
- “Self-describing“ (human readable)
- Hierarchical (Values can contain lists of objects or values)

# Not Like XML


- Lighter and faster than XML
- JSON uses typed objects. All XML values are type-less strings and must be parsed at runtime.
- Less syntax, no semantics
- Properties are immediately accessible to JavaScript code

# Knocks against JSON

- Lack of namespaces
- No inherent validation (XML has DTD and templates, but there is JSONlint)
- Not extensible
- It's basically just ***not*** XML

# Syntax

# JSON Object Syntax

- Unordered sets of name/value pairs
- Begins with { (left brace)
- Ends with } (right brace)
- Each name is followed by : (colon)
- Name/value pairs are separated by , (comma)

# JSON Example

```
var employeeData = {
  "employee_id": 1234567,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false
};
```

# Arrays in JSON

- An ordered collection of values
- Begins with [ (left bracket)
- Ends with ] (right bracket)
- Name/value pairs are separated by , (comma)

# JSON Array Example

```
var employeeData = {
  "employee_id": 1236937,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false,
   "random_nums": [ 24,65,12,94 ]
};
```

# Data Types

# Data Types: Strings

- Sequence of zero or more Unicode characters
- Wrapped in "double quotes"
- Backslash escapement

# Data Types: Numbers

- Integer
- Real
- Scientific
- No octal or hex
- No NaN (Not a Number) or Infinity – Use **null** instead.

# Let’s end with an example JSON

```
{     "firstName": "John",
      "lastName": "Smith",
      "age": 25,
      "address":    {
             "streetAddress": "21 2nd Street",
             "city": "New York",
             "state": "NY",
             "postalCode": "10021“
             },
      "phoneNumber": [
                 {
                 "type": "home",
                 "number": "212 555-1234“
                 },
                 {
                 "type": "fax",
                 "number": "646 555-4567“
                 }
             ]
}
```

# The same Example in XML

```
<Object>
<Property><Key>firstName</Key> <String>John</String></Property>
<Property><Key>lastName</Key> <String>Smith</String></Property>
<Property><Key>age</Key> <Number>25</Number></Property>
<Property><Key>address</Key> <Object> <Property><Key>streetAddress</Key>
<String>21 2nd Street</String></Property>
<Property><Key>city</Key> <String>New York</String></Property>
<Property><Key>state</Key> <String>NY</String></Property>
<Property><Key>postalCode</Key> <String>10021</String></Property>
</Object>
</Property> <Property><Key>phoneNumber</Key>
<Array> <Object> <Property><Key>type</Key> <String>home</String></Property>
<Property><Key>number</Key> <String>212 555-1234</String></Property></Object>
<Object>
<Property><Key>type</Key> <String>fax</String></Property> <Property><Key>number</
Key> <String>646 555-4567</String></Property> </Object> </Array>
</Property>
</Object>
```

# Where is JSON used today?

- Anywhere and everywhere **(even in 2013, much more now)**!

And many, many more!

# Some Resources

- Simple Demo on Github: https://github.com/jfox015/BIFC-Simple-JSON-Demo
- Another JSON Tutorial: http://iviewsource.com/codingtutorials/getting-started-with-javascript-object-notation-json-for-absolute-beginners/
- JSON.org: http://www.json.org/

# Google Protocol Buffers

**from:**

**F1: A Distributed SQL Database That Scales**

**http://dl.acm.org/citation.cfm?id=2536232**

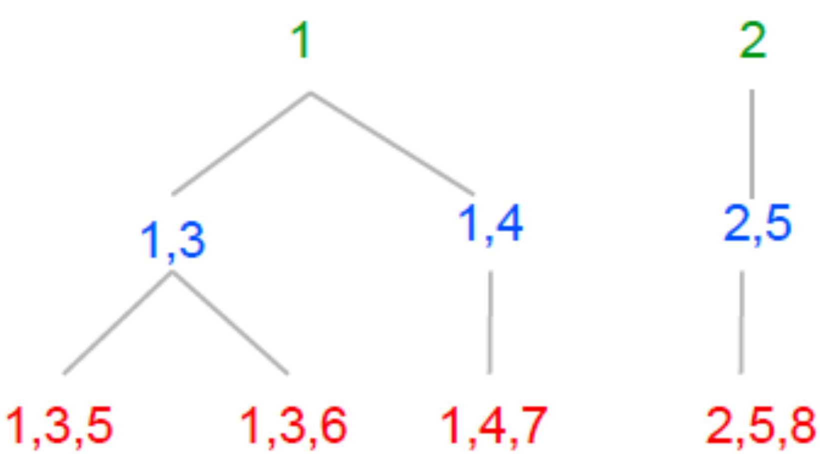
**Protocol Buffers: Not on Final!**

# Hierarchical Schema

Explicit table hierarchies. Example:

- Customer (root table): PK (CustomerId)
- Campaign (child): PK (CustomerId, CampaignId)
- AdGroup (child): PK (CustomerId, CampaignId, AdGroupId)

**Rows and PKs**



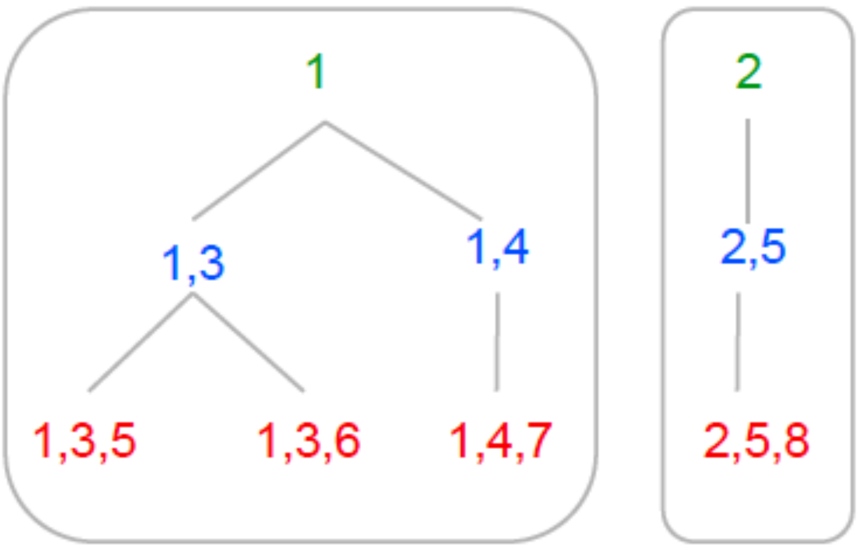

**Storage Layout**

```
Customer (1)
Campaign (1,3)
AdGroup  (1,3,5)
AdGroup  (1,3,6)
Campaign (1,4)
AdGroup  (1,4,7)
Customer (2)
Campaign (2,5)
AdGroup  (2,5,8)
```

# Clustered Storage

- Child rows under one root row form a **cluster**
- Cluster stored on one machine (unless huge)
- Transactions within one cluster are most efficient
- Very efficient joins inside clusters (can merge with no sorting)

**Rows and PKs**




**Storage Layout**

```
Customer (1)
Campaign (1,3)
AdGroup  (1,3,5)
AdGroup  (1,3,6)
Campaign (1,4)
AdGroup  (1,4,7)
```

```
Customer (2)
Campaign (2,5)
AdGroup  (2,5,8)
```

# Protocol Buffer Column Types

Protocol Buffers

- Structured data types with optional and repeated fields
- Open-sourced by Google, APIs in several languages

Column data types are mostly Protocol Buffers

- Stored like blobs in Spanner
- SQL syntax extensions for reading nested fields
- Coarser schema with fewer tables - inlined objects instead

Why useful?

- Protocol Buffers pervasive at Google -> no impedance mismatch
- Simplified schema and code - apps use the same objects
  - Don't need foreign keys or joins if data is inlined

# SQL on Protocol Buffers

```
SELECT CustomerId, Whitelist
FROM Customer
```

| CustomerId | Whitelist |
|---|---|
| 123 | feature {<br>  feature_id: 18<br>  status: ENABLED<br>}<br>feature {<br>  **feature_id: 269**<br>  **status: ENABLED**<br>}<br>feature {<br>  feature_id: 302<br>  status: ENABLED<br>} |

```
SELECT CustomerId, f.*
FROM Customer c
PROTO JOIN c.Whitelist.feature f
WHERE f.feature_id IN (269, 302)
  AND f.status = 'ENABLED'
```

| CustomerId | feature_id | status |
|---|---|---|
| 123 | **269** | **ENABLED** |
| 123 | 302 | ENABLED |