Representation of structured, multi-level (hierarchical/nested), relational data using CSV (csv_ml)

Proposed by *Arundale R.*, *Siara Logics (cc)*
e-mail: *arun@siara.cc*, web: *http://siara.cc*

Overview

This article proposes the idea of using CSV format for defining structured relational data. The idea is nick named csv_ml and open source reference implementations are available under github repository https://github.com/siara-cc/csv_ml.

This idea germinated when looking for a compact format for representing and transferring structured data. The usual choices are XML and JSON, both of which waste a lot of space interleaving schema with data, while attempting to provide flexible data representation.

csv_ml attempts to provide a simple unambiguous format for representing structured data that includes schema definition.

In short, csv_ml is expected to
- save storage space (about 50% compared to JSON and 60-70% compared to XML)
- increase data transfer speeds
- be faster to parse compared to XML and JSON
- allow full schema definition and validation
- make schema definition simple, lightweight and in-line compared to DTD or XML Schema
- allow database binding
- be simpler to parse, allowing data to be available even in low memory devices

This document illustrates the idea starting with simple conventional CSV and proceeds to explain how it can be used to represent complex nested relational data structures with examples.

RFC4180 is taken as the basis for parsing of CSV.

Applications

- Enterprise Application Integration (EAI)
- Lightweight alternative to JSON or XML in Three-tier architecture
- Alternative to XML in transfer of data using AJAX
- Data storage and transfer format for embedded platforms such as Arduino and Raspberry PI.
- Data storage and transfer format for mobile/tablet devices based on Android, Windows or iOS.

Simple Single Level CSV data

Example 1.1: Conventional CSV

Although this article proposes using CSV for representing multi-level data, we start the discussion with conventional CSV example. However, the idea is not to represent just tabular data.

CSV is originally intended to represent tabular data. Consider the following example, which represents student data (name, subject, marks):

```
abc,physics,53
abc,chemistry,65
xyz,physics,73
xyz,chemistry,76
```

When we try to convert it to XML or JSON, we come across a problem, that is, it does not have any schema information, such as node name and attribute name. So we assume arbitrary names and transform as follows:

```
<?xml version="1.0" ?>
<nl c1="abc" c2="physics" c3="53"/>
<nl c1="abc" c2="chemistry" c3="65"/>
<nl c1="xyz" c2="physics" c3="73"/>
<nl c1="xyz" c2="chemistry" c3="76"/>
```

Here we come across another problem. There is no root node. So we also add an arbitrary root element, to make it well-formed XML. The below table shows CSV and XML representations of the same data:

CSV	abc,physics,53 abc,chemistry,65 xyz,physics,73 xyz,chemistry,76
XML	<?xml version="1.0" ?> <root> <nl c1="abc" c2="physics" c3="53"/> <nl c1="abc" c2="chemistry" c3="65"/> <nl c1="xyz" c2="physics" c3="73"/> <nl c1="xyz" c2="chemistry" c3="76"/> </root>

So an arbitrary root always forms the basis for all further examples.

Example 1.2: Conventional CSV with Header

Usually the first line of CSV data indicates the column names as in the following example:

```
name,subject,marks
abc,physics,53
abc,chemistry,65
xyz,physics,73
```

However, to parse this, a directive would be needed, to inform the parser that a header is present. Otherwise, the first line would also be considered as data. The directive is explained below:

```
csv_ml,1.0,UTF-8,root,no_node_name,inline
```

There are five columns in the directive, as explained below:

- `csv_ml` - indicates that this line is directive
- `1.0` - indicates version
- `UTF-8` - indicates encoding
- `root` - indicates what name should be used for the root element. It is `root` by default and can be changed. If omitted, `root` is used. If it is the same as the first data element name, the parser would attempt to make it the root. But if there are more than one siblings at the first level, a parsing error would be generated.
- `no_node_name` - indicates that node name is not mentioned in the header and it is to be assigned by the parser (n1 in this case). The other option would be `with_node_name`, which indicates that node name is used to link rows in data section with schema.
- `inline` - indicates that a header (schema) is present before data starts. The other option would be `external`, which indicates that schema is defined in an external file and the file name follows as next CSV field. The file name specification could be relative or absolute depending on Operation System conventions.

Accordingly, the CSV is parsed as shown below:

CSV	csv_ml,1.0,UTF-8,root,no_node_name,inline name,subject,marks abc,physics,53 abc,chemistry,65 xyz,physics,73 xyz,chemistry,76
XML	<root> <nl name="abc" subject="physics" marks="53"/> <nl name="abc" subject="chemistry" marks="65"/> <nl name="xyz" subject="physics" marks="73"/> <nl name="xyz" subject="chemistry" marks="76"/> </root>

Example 1.3 and 1.4: Conventional CSV with Header and Node name

The node name can be specified in the header as shown below. It would be used by the parser instead of assigning node name such as n1. Example 1.3 and 1.4 are equivalent and so produce the same output. However, the difference is explained below.

	Example 1.3	=	**Example 1.4**
CSV	csv_ml,1.0,UTF-8,root,with_node_name,inline student,name,subject,marks end_schema	=	csv_ml,1.0 student,name,subject,marks 1,abc,physics,53

	student,abc,physics,53 student,abc,chemistry,65 student,xyz,physics,73 student,xyz,chemistry,76		1,abc,chemistry,65 1,xyz,physics,73 1,xyz,chemistry,76
XML	<root> <student name="abc" subject="physics" marks="53"/> <student name="abc" subject="chemistry" marks="65"/> <student name="xyz" subject="physics" marks="73"/> <student name="xyz" subject="chemistry" marks="76"/> </root>		

While the output is the same, there are four differences between Example 1.3 and Example 1.4:

1. The directive `csv_ml,1.0,UTF-8,root,with_node_name,inline` is the same as `csv_ml,1.0`, because "`UTF-8`", "`root`", "`with_node_name`" and "`inline`" are default values if not specified.
2. In Example 1.3, the node name student needs to be specified in each line of data section. This is because more than one node could be defined at the same level in general (siblings in case of XML). The node name indicates which sibling the data corresponds to.
3. "`end_schema`" is required in Example 1.3 as there is no way of distinguishing where schema ends and data starts.
4. Once the schema is defined, the node names in data section can be indicated using index positions (in this case 1) instead of names. This also eliminates the need for "`end_schema`". In any case, node name (or) index position would be required in the data section, as there could be more than one node in the same level (in this case, under root).

Index positions would be used in subsequent examples (as in Example 1.4) as it further reduces space required.

Example 1.5: Multiple nodes under root

Multiple nodes can be defined under root element as shown below:

CSV	csv_ml,1.0 student,name,subject,marks faculty,name,subject 1,abc,physics,53 1,abc,chemistry,65 1,xyz,physics,73 1,xyz,chemistry,76 2,pqr,physics 2,bcd,chemistry
XML	<root> <student name="abc" subject="physics" marks="53"/> <student name="abc" subject="chemistry" marks="65"/> <student name="xyz" subject="physics" marks="73"/> <student name="xyz" subject="chemistry" marks="76"/> <faculty name="pqr" subject="physics"/> <faculty name="bcd" subject="chemistry"/> </root>

It can be seen that student node has index number 1 and faculty has 2.

Multi-Level CSV data

So far we have looked at single level CSV, but main objective of this exercise is to be able to define structured multi-level data as in XML or JSON.

Example 2.1: Multiple level CSV data

By using space characters at the beginning of each line, hierarchy can be specified using CSV format as shown in the below example:

CSV

```
csv_ml,1.0
student,name,age
 education,course_name,year_passed
  subject,name,marks
1,abc,24
 1,bs,2010
  1,physics,53
  1,chemistry,65
 1,ms,2012
  1,physics,74
  1,chemistry,75
1,xyz,24
 1,bs,2010
  1,physics,67
  1,chemistry,85
```

XML

```
<root>
  <student name="abc" age="24">
    <education course_name="bs" year_passed="2010">
      <subject name="physics" marks="53"/>
      <subject name="chemistry" marks="65"/>
    </education>
    <education course_name="ms" year_passed="2012">
      <subject name="physics" marks="74"/>
      <subject name="chemistry" marks="75"/>
    </education>
  </student>
  <student name="xyz" age="24">
    <education course_name="bs" year_passed="2010">
      <subject name="physics" marks="67"/>
      <subject name="chemistry" marks="85"/>
    </education>
  </student>
</root>
```

In the above example, the first node (student) does not have any space. The next node education begins with one space indicating that it is below the node student. The next node subject begins with two spaces indicating that it is below education.

The data section also follows the same pattern and the node is indicated using index position (1). Since there are no siblings, all the data rows start with index as 1.

Example 2.2: Multiple level CSV data with siblings

The index position is incremented within each level when siblings need to be defined. This is illustrated in the following example.

CSV

```
csv_ml,1.0
student,name,age
 education,course_name,year_passed
  subject,name,marks
 references,name,company,designation
1,abc,24
 1,bs,2010
  1,physics,53
  1,chemistry,65
 1,ms,2012
  1,physics,74
  1,chemistry,75
 2,pqr,bbb,executive
 2,mno,bbb,director
1,xyz,24
 1,bs,2010
  1,physics,67
  1,chemistry,85
```

XML

```
<root>
  <student name="abc" age="24">
    <education course_name="bs" year_passed="2010">
      <subject name="physics" marks="53"/>
      <subject name="chemistry" marks="65"/>
    </education>
    <education course_name="ms" year_passed="2012">
      <subject name="physics" marks="74"/>
      <subject name="chemistry" marks="75"/>
    </education>
    <references name="pqr" company="bbb" designation="executive"/>
    <references name="mno" company="bbb" designation="director"/>
  </student>
  <student name="xyz" age="24">
    <education course_name="bs" year_passed="2010">
      <subject name="physics" marks="67"/>
      <subject name="chemistry" marks="85"/>
    </education>
  </student>
</root>
```

The nodes `education` and `references` are siblings and they are also children of node `student`. So in the data section, they are referred using the index numbers 1 and 2 respectively.

General aspects of parsing

Example 3.1: Node attributes and content

While the data elements are equivalent to node attributes in XML, any data after the last column can be treated as CData content of the node.

For example, consider the following csv_ml:

```
csv_ml,1.0
student,name,age
1,a,24,His record is remarkable
```

would be equivalent to:

```
<?xml version=”1.0“?>
<root>
  <student name="a” age=”24“><![CDATA[His record is remarkable]]></student>
</root>
```

If data after the last column includes comma, more than one CData section is created:

```
csv_ml,1.0
student,name,age
1,c,23,His record is remarkable,His performance is exemplary
```

would be equivalent to:

```
<?xml version=”1.0“?>
<root>
  <student name="c” age=”23“><![CDATA[His record is remarkable]]>
                            <![CDATA[His performance is exemplary]]></student>
</root>
```

While this situation is not expected in any practical implementation, this is mentioned to remove any ambiguity about how the parser should handle such data.

If less number of columns appear in data, the remaining columns are to be set with empty value, as shown below:

```
csv_ml,1.0
student,name,age
1,a
```

would be equivalent to:

```
<?xml version=”1.0“?>
<root>
  <student name="a” age=”“></student>
</root>
```

All the above situations are explained using the following example:

CSV	csv_ml,1.0 student,name,age 1,a 1,b,23,His record is remarkable 1,c,24,His record is remarkable,His performance is exemplary
XML	<root> <student age="" name="a"/> <student age="23" name="b"> <![CDATA[His record is remarkable]]></student> <student age="24" name="c"> <![CDATA[His record is remarkable]]> <![CDATA[His performance is exemplary]]></student> </root>

Example 3.2: Node content

The data elements (attributes) are optional. Consider the following example:

CSV

```
csv_ml,1.0
student
 name
 age
1
 1,a
 2,23
```

XML

```
<?xml version="1.0"?>
<root>
  <student>
    <name>a</name>
    <age>34</age>
  </student>
</root>
```

Example 3.3: Quote handling

Double quotes are handled according to RFC4180. The different scenarios are illustrated using the following example:

CSV

```
csv_ml,1.0
sample,text
1,No quote
1, No quote with preceding space
1,With quote (")
1,"With quotes. and ""comma"""
1, "With quotes, (space ignored)"
1, """Enclosed, with double quote"""
1, """Single, preceding double quote"
1, "Double quote, suffix"""
1, "Double quote, ("") in the middle"
1, "More
than
one
line"
```

XML

```
<root>
  <sample text="No quote"/>
  <sample text=" No quote with preceding space"/>
  <sample text="With quote (&amp;quot;)"/>
  <sample text="With quotes. and &amp;quot;comma&amp;quot;"/>
  <sample text="With quotes, (space ignored)"/>
  <sample text="&amp;quot;Enclosed, with double quote&amp;quot;"/>
  <sample text="&amp;quot;Single, preceding double quote"/>
  <sample text="Double quote, suffix&amp;quot;"/>
  <sample text="Double quote, (&amp;quot;) in the middle"/>
  <sample text="More
than
one
line"/>
</root>
```

Example 3.4: Inline comments and empty lines

Although RFC4180 does not specify about the possibility of comments and empty lines within CSV, it would be desirable to be able to include comments, to improve readability. It is also to be noted that XML allows inline comments and spaces.

The following example illustrates the various ways in which comments can be included in a CSV file.

CSV

```
/* You can have comments anywhere,
   even at the beginning
*/
csv_ml,1.0

/* And empty lines like this */

sample,text1,text2
1,/* This is a comment */ "hello", "world" /* End of line comment */
1,/* This is also a comment */, "/* But this isn't */"

1,"third", "line" /* Multiline
comment */
/* Comment at beginning of line */1, "fourth" , "line"
```

XML

```
<?xml version="1.0"?>
<root>
  <sample text1="hello" text2="world"/>
  <sample text1="" text2="/* But this isn't */"/>
  <sample text1="third" text2="line"/>
  <sample text1="fourth" text2="line"/>
</root>
```

Comments within double quotes are not recognized as comment, but recognized as data.

Example 3.5: Changing root node

The root node can be change to something other than root, using the directive as shown below.

CSV

```
csv_ml,1.0,UTF-8,data
student,name,age
1,a,24
```

XML

```
<?xml version="1.0"?>
<data>
  <student name="a" age="24"/>
</data>
```

If the specified root element name is the same as the first data element, the first data element is made as root, as shown below:

CSV

```
csv_ml,1.0,UTF-8,student
student,name,age
1,a,24
```

XML

```
<?xml version="1.0"?>
<student name="a" age="24"/>
```

However, if there are more than one record for the first element, or if there are more than one siblings, then a parser error is generated. For example, both the examples given below will not parse successfully, when trying to convert to XML:

```
csv_ml,1.0,UTF-8,student
student,name,age
1,a,24
1,b,35
```

```
csv_ml,1.0,UTF-8,student
student,name,age
faculty,name,age
1,a,24
2,b,45
```

But when trying to convert the above two examples to JSON format, there would be no error as JSON does not have the concept of unique root element.

Example 3.6: Namespaces

As in XML, it is also possible to use namespaces, by using a colon character in node name or attribute. An example is given below:

CSV

```
csv_ml,1.0
our:student,his:name,age,xmlns:his,xmlns:our
1,a,24,http://siara.cc/his,http://siara.cc/our
1,b,26,http://siara.cc/his,http://siara.cc/our
```

XML

```
<?xml version="1.0"?>
<root>
  <our:student his:name="a" age="24"
    xmlns:his="http://siara.cc/his" xmlns:our="http://siara.cc/our"/>
  <our:student his:name="b" age="26"
    xmlns:his="http://siara.cc/his" xmlns:our="http://siara.cc/our"/>
</root>
```

The URI can be defined at the root level as shown below:

CSV

```
csv_ml,1.0,UTF-8,root/our='http://siara.cc/our' his='http://siara.cc/his'
our:student,his:name,age
1,a,24
1,b,26
```

XML

```
<?xml version="1.0"?>
<root xmlns:our="http://siara.cc/our" xmlns:his="http://siara.cc/his">
  <our:student his:name="a" age="24"/>
  <our:student his:name="b" age="26"/>
</root>
```

Note that the namespace definitions start after the node name and a / (forward slash). Also if there are more than one namespaces, they are separated by space character.

The root node itself can have a namespace as shown in the following example:

CSV

```
csv_ml,1.0,UTF-8,xsl:stylesheet/xsl='http://www.w3.org/1999/XSL/Transform'
xsl:stylesheet
 xsl:template,match
  xsl:value-of,select
1
 1,student
  1,@name
  1,@age
```

XML

```
<?xml version=”1.0“?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="student"/>
     <xsl:value-of select="@name"/>
     <xsl:value-of select="@age"/>
  </xsl:template>
</xsl:stylesheet>
```

The above example shows how a stylesheet could be represented using CSV.

Example 3.7: Re-using node definitions

In XML Schema definitions, node definitions can be re-used. With csv_ml, re-usable nodes can be defined and referred to, with a special prefix of 0. All the re-usable nodes are to be defined at the beginning and linked to the structure under each node.

In the below example, `xsl:value-of` is a node that can appear both under `xsl:template` and `xsl:for-each`.

CSV

```
csv_ml,1.0,UTF-8,xsl:stylesheet/xsl='http://www.w3.org/1999/XSL/Transform'
01,xsl:value-of,select
02,xsl:for-each,select
 01
xsl:stylesheet,version
 xsl:template,match
  01,02
1,1.0
 1,//student
  01,@name
  01,@age
  02,education
   01,@course_name
   01,@year_passed
```

XML

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0">
  <xsl:template match="//student">
     <xsl:value-of select="@name"/>
     <xsl:value-of select="@age"/>
     <xsl:for-each select="education">
       <xsl:value-of select="@course_name"/>
       <xsl:value-of select="@year_passed"/>
     </xsl:for-each>
  </xsl:template>
</xsl:stylesheet>
```

The first two lines define the nodes `xsl:value-of` and `xsl:for-each` as re-usable by using the prefix 0 (`01` and `02`). The third line indicates that `xsl:value-of` can be a child of `xsl:for-each`. While the next two lines are normal schema definitions for `xsl:stylesheet` and `xsl:template`, the sixth line indicates (`01,02`) that both `xsl:for-each` and `xsl:value-of` can be children of `xsl:template`.

Schema definition and database binding

The essential purpose of schema definition is data validation. With csv_ml, it can also be used to bind with a database. Following are the different basic data types that are recognized:

text
numeric
integer
real
date
datetime

If not specified, the default data type would be text. For binding with a database, if the type name is not the same as any of the above, it can be mapped to one of the above.

In the below sections, each feature of schema definition is explained with an example and how it is bound to a database by showing the corresponding DDL (Data Definition Language) and DML (Data Manipulation Language).

Using DDL and DML is only one of the ways of binding csv_ml to a database. Alternatively, a program could be used instead of just generating statements. The examples shown here can be used as a reference.

Example 4.1: Specifying Type and Length

The type and length of a particular attribute can be specified as in the example below.

CSV	csv_ml,1.0 student,name(40)text,subject(30)text,marks(3)integer 1,abc,physics,53 1,xyz,physics,73
DDL	CREATE TABLE student (name text(40) NOT NULL, subject text(30) NOT NULL, marks integer(3) NOT NULL, id integer primary key autoincrement);

Example 4.2: Default value

An attribute can be assigned a default value as in the below example:

CSV	csv_ml,1.0 student,name(40)text,subject(30)text=physics,marks(3)integer 1,abc,pqr,maths,53 1,xyz,,chemistry,73

DDL	CREATE TABLE student (name text(40) NOT NULL, subject text(30) DEFAULT 'physics', marks integer(3) NOT NULL, id integer primary key autoincrement);

In the above example, subject has a default value of 'physics'.

Example 4.3: Null values

An attribute is assumed to be NOT NULL by default. This behaviour can be modified as follows.

CSV	csv_ml,1.0 student,name(40)text,nick(30)text=null,subject(30)text,marks(3)integer 1,abc,pqr,physics,53 1,xyz,,physics,73
DDL	CREATE TABLE student (name text(40) NOT NULL, nick text(30), subject text(30) NOT NULL, marks integer(3) NOT NULL, id integer primary key autoincrement);

In the above example, nick allows null values. In general, usage of NULL value is discouraged as the values become incomparable. It is more convenient and less confusing to use a default value as in the below example. This also facilitates when two nullable columns are joined.

CSV	csv_ml,1.0 student,name(40)text,nick(30)text=,subject(30)text,marks(3)integer 1,abc,pqr,physics,53 1,xyz,,physics,73
DDL	CREATE TABLE student (name text(40) NOT NULL, nick text(30) DEFAULT '', subject text(30) NOT NULL, marks integer(3) NOT NULL, id integer primary key autoincrement);

Example 4.4: Precision and Scale

Numeric precision and scale can be specified as follows:

CSV	csv_ml,1.0 student,name(40)text,subject(30)text,"marks(6,2)numeric" 1,abc,physics,53.34 1,xyz,physics,73.5
DDL	CREATE TABLE student (name text(40) NOT NULL, subject text(30) NOT NULL, marks numeric(6,2) NOT NULL, id integer primary key autoincrement);

In the above example, marks column has a precision of 6 and scale of 2.

Example 4.5: Date and Time

Date and time values are specified using ISO-8601 format (Basically YYYYMMDD and YYYYMMDDHHMISS)

CSV	csv_ml,1.0 student,name,subject,marks,birth_date()date,join_date_time()datetime 1,abc,physics,53.34,19820123,20140222093000 1,xyz,physics,73.5,19851112,20140224154530
DDL	CREATE TABLE student (name text NOT NULL, subject text NOT NULL, marks text NOT NULL, birth_date date NOT NULL, join_date_time datetime NOT NULL, id integer primary key autoincrement);
DML	INSERT INTO student (name, subject, marks, birth_date, join_date_time) VALUES ('abc', 'physics', '53.34', '1982-01-23', '2014-02-22 09:30:00'); INSERT INTO student (name, subject, marks, birth_date, join_date_time) VALUES ('xyz', 'physics', '73.5', '1985-11-12', '2014-02-24 15:45:30');
SQL sample	select strftime('%Y', birth_date) as year_of_birth, strftime('%s', join_date_time) as seconds_elapsed_since_join from student;

Example 4.6: Special column id

For specifying primary key for the record, a special column id is used. If it is not specified as part of the schema, it is included when generating DDL.

This can be seen in any of the above examples (`id integer primary key autoincrement`).

Apart from being used as primary key, it can be used to INSERT, UPDATE or DELETE if specified as part of the schema.

CSV	csv_ml,1.0 student,id,name,subject,marks 1,,abc,physics,53 1,,abc,chemistry,54 1,3,xyz,physics,73 1,*4,xyz,physics,73
DDL	CREATE TABLE student (id text NOT NULL primary key autoincrement, name text NOT NULL, subject text NOT NULL, marks text NOT NULL);
DML	INSERT INTO student (name, subject, marks) VALUES ('abc', 'physics', '53'); INSERT INTO student (name, subject, marks) VALUES ('abc', 'chemistry', '54'); UPDATE student SET name = 'xyz', subject = 'physics', marks = '73' WHERE id=3; DELETE FROM student WHERE id=4;

As can be seen from the DML generated above, rows that have empty id values generate INSERT statements, rows having id values generate UPDATE statements and rows that have a * symbol before the id value generate DELETE statements.

Example 4.7: Special column parent_id

In addition to id column, a `parent_id` column is automatically added to those nodes that have a parent node.

Whenever an INSERT statement is generated, those having parent nodes fill the `parent_id` column with the id generated for the parent row. This is shown in the following example:

CSV

```
csv_ml,1.0
student,name,age
 education,course_name,year_passed
 references,name,company,designation
1,abc,24
 1,bs,2010
 1,ms,2012
 2,pqr,bbb,executive
 2,mno,bbb,director's secretary
```

DDL

```
CREATE TABLE student (name text NOT NULL, age text NOT NULL, id
integer primary key autoincrement);

CREATE TABLE education (course_name text NOT NULL, year_passed text
NOT NULL, id integer primary key autoincrement, parent_id integer);

CREATE TABLE references (name text NOT NULL, company text NOT NULL,
designation text NOT NULL, id integer primary key autoincrement,
parent_id integer);
```

DML

```
INSERT INTO student (name, age) VALUES ('abc', '24');

INSERT INTO education (course_name, year_passed, parent_id) VALUES
('bs', '2010', (select seq from sqlite_sequence where
name='student'));

INSERT INTO education (course_name, year_passed, parent_id) VALUES
('ms', '2012', (select seq from sqlite_sequence where
name='student'));

INSERT INTO references (name, company, designation, parent_id) VALUES
('pqr', 'bbb', 'executive', (select seq from sqlite_sequence where
name='student'));

INSERT INTO references (name, company, designation, parent_id) VALUES
('mno', 'bbb', 'director''s secretary', (select seq from
sqlite_sequence where name='student'));
```

The id/parent_id combination creates a foreign key relationship between tables. However, the DDL for foreign key is not generated, as the database will generate an error when a parent row is deleted. The parent and child rows can be deleted by indicating * in the id value.

Single Quotes

Any single quote found in the data is encoded with double single quotes in the value section as shown in Example 4.7 above (`'director''s secretary'`).

Retrieval of data from database

Once the data is stored into tables using DDL/DML scripts, it is possible to retrieve it back using reference to the id of the topmost element. Following sections show how the data can be retrieved for the examples already discussed:

Retrieving data for Example 2.1

In this case three tables would be created using the DDL generated, namely, student, education and subject. The scripts generated and data stored in SQLite database when running them are shown below:

CSV

```
csv_ml,1.0
student,name,age
 education,course_name,year_passed
  subject,name,marks
1,abc,24
 1,bs,2010
  1,physics,53
  1,chemistry,65
 1,ms,2012
  1,physics,74
  1,chemistry,75
```

Generated DDL

```
CREATE TABLE student (name text NOT NULL, age text NOT NULL, id
integer primary key autoincrement);

CREATE TABLE education (course_name text NOT NULL, year_passed text
NOT NULL, id integer primary key autoincrement, parent_id integer);

CREATE TABLE subject (name text NOT NULL, marks text NOT NULL, id
integer primary key autoincrement, parent_id integer);
```

Generated DML

```
INSERT INTO student (name, age) VALUES ('abc', '24');
INSERT INTO education (course_name, year_passed, parent_id) VALUES
('bs', '2010', (select seq from sqlite_sequence where
name='student'));
INSERT INTO subject (name, marks, parent_id) VALUES ('physics', '53',
(select seq from sqlite_sequence where name='education'));
INSERT INTO subject (name, marks, parent_id) VALUES ('chemistry',
'65', (select seq from sqlite_sequence where name='education'));
INSERT INTO education (course_name, year_passed, parent_id) VALUES
('ms', '2012', (select seq from sqlite_sequence where
name='student'));
INSERT INTO subject (name, marks, parent_id) VALUES ('physics', '74',
(select seq from sqlite_sequence where name='education'));
INSERT INTO subject (name, marks, parent_id) VALUES ('chemistry',
'75', (select seq from sqlite_sequence where name='education'));
```

SQLite dump with actual id and parent_id

```
INSERT INTO "student" VALUES('abc','24',1);
INSERT INTO "education" VALUES('bs','2010',1,1);
INSERT INTO "education" VALUES('ms','2012',2,1);
INSERT INTO "subject" VALUES('physics','53',1,1);
INSERT INTO "subject" VALUES('chemistry','65',2,1);
INSERT INTO "subject" VALUES('physics','74',3,2);
INSERT INTO "subject" VALUES('chemistry','75',4,2);
```

CSV retrieved back	csv_ml,1.0 student,name,age,id education,course_name,year_passed,parent_id,id subject,name,marks,parent_id,id end_schema /* Select name, age, id From student Where id='1' */ student,abc,24,1 /* Select course_name, year_passed, parent_id, id From education Where parent_id='1' */ education,bs,2010,1,1 /* Select name, marks, parent_id, id From subject Where parent_id='1' */ subject,physics,53,1,1 subject,chemistry,65,1,2 education,ms,2012,1,2 /* Select name, marks, parent_id, id From subject Where parent_id='2' */ subject,physics,74,2,3 subject,chemistry,75,2,4

It can be seen from the above example how the data stored into the database can be retrieved back with a single API call.

The SQLite special table sqlite_sequences stores the auto-generated sequence number of the last INSERT statement. So this feature is used to insert data into the parent_id columns of the child tables.

From the SQLite dump, the values generated for id and parent_id columns can be seen.

The SQL statements given within comment are used to retrieve the subsequently generated node data. The id and parent_id columns are also included in the retrieved csv.

Retrieving data for Example 1.5

In example 1.5, there are two nodes student and faculty and the first level. So for retrieving data for this, id value for each node needs to be passed separately as shown below:

CSV	csv_ml,1.0 student,name,subject,marks faculty,name,subject 1,abc,physics,53 2,pqr,physics
Generated DML	INSERT INTO student (name, subject, marks) VALUES ('abc', 'physics', '53'); INSERT INTO faculty (name, subject) VALUES ('pqr', 'physics');
SQLite dump with actual id and parent_id	INSERT INTO “student” VALUES ('abc', 'physics', '53', 1); INSERT INTO faculty VALUES ('pqr', 'physics', 1);

CSV retrieved back	csv_ml,1.0 student,name,subject,marks,id faculty,name,subject,id end_schema /* Select name, subject, marks, id From student Where id='1' */ student,abc,physics,53,1 /* Select name, subject, id From faculty Where id='1' */ faculty,pqr,physics,1

For this case id value ‘1' needs to be passed separately to student and faculty nodes. In real life situation, the values could be different.

The API call for id in this case would be an array containing two values. The API details are given in the subsequent section.

API

The reference implementation provides visual demo programs developed on Swing, Android, Javascript and Applet.

API (Application-Program-Interface) for Java and Javascript are shown below:

Basic parsing of csv_ml to DOM with input as string

Java

```
MultiLevelCSVParser csv_ml_parser = new MultiLevelCSVParser();
StringReader sr
     = new StringReader("csv_ml,1.0\nstudent,name,age\n1,abc,23");
Document dom = csv_ml_parser.parseToDocument(sr, false);
if (csv_ml_parser.getEx().getErrorCode() > 0) {
   System.out.println(csv_ml_parser.getEx().get_all_exceptions());
   return;
}
// Do something with dom, such as
// String c1 = dom.getDocumentElement().getChildNodes().item(0)
//             .getAttributes().getNamedItem("name").getNodeValue();
```

Java script

```
var csv_ml_parser
   = new CSV_ML_Parser("csv_ml,1.0\nstudent,name,age\n1,abc,23");
var dom = csv_ml_parser.parse("dom", false);
if (csv_ml_parser.ex.error_code > 0) {
   display_exceptions();
   return;
}
// Do something with dom, such as
// alert(dom.documentElement.childNodes[0].getAttribute("name"));
```

Basic parsing of csv_ml to JSON with input as string

Java

```
MultiLevelCSVParser csv_ml_parser = new MultiLevelCSVParser();
StringReader sr
   = new StringReader("csv_ml,1.0\nstudent,name,age\n1,abc,23");
JSONObject jso = csv_ml_parser.parseToJSO(sr, false);
if (csv_ml_parser.getEx().getErrorCode() > 0) {
   System.out.println(csv_ml_parser.getEx().get_all_exceptions());
   return;
}
// Do something with jso, such as
// String c1 = jso.getJSONArray("student")
//                  .item(0).getJSONString("name");
```

Java script

```
var csv_ml_parser
   = new CSV_ML_Parser("csv_ml,1.0\nstudent,name,age\n1,abc,23");
var jso = csv_ml_parser.parse("jso", false);
if (csv_ml_parser.ex.error_code > 0) {
   display_exceptions();
   return;
}
// Do something with jso, such as
// alert(jso["student"][0]["name"]);
```

API for advanced (pull) parsing of csv_ml to JSON with input as string

The reference implementation supports advanced Pull parsing, which means that as soon as parsing for a element is complete, the control returns to the API caller. The caller could process the data in the element and destroy the contents to conserve memory. This is in contrast to parsing the entire file and returning a memory intensive Document object.

This is especially useful when parsing huge streams of data such as passing data from one system to another as in Enterprise Application Integration (EAI).

An example is given below:

Java

```
MultiLevelCSVParser parser = new MultiLevelCSVParser();
// Usually a stream is used (java.io.InputStream or java.io.Reader)
StringReader sr
    = new StringReader("csv_ml,1.0\nstudent,name,age\n1,abc,23");
short targetObject = ParsedObject.TARGET_W3C_DOC; // or TARGET_JSON_OBJ
parser.initParse(targetObject, is, toValidate);
for (ParsedObject parsedObject = parser.parseNext();
   parsedObject != null; parsedObject = parser.parseNext()) {
   // parseNext() returns for each Node successfully constructed.
   // parsedObject.getCurrentElement() or parsedObject.getCurrentJSO()
   // can be called to have access to the successfully constructed
   // node, such as:
   //
   // if (parser.getEx().getErrorCode() == 0) {
   //    Element parsedElement = parsedObject.getCurrentElement();
   //    System.out.println(parsedElement.getAttribute("name"));
   //    // Do something with parsedElement here
   //    // and if necessary free memory by deleting attributes
   //    // // or even remove parseElement itself from
   //    // // the document object after use
   // }
}
```

API for generating DDL and DML

The API for generating DDL and DML needs a schema object generated from parsing the csv_ml. This can be obtained by just calling initParse, if only DDL is needed or by completely parsing the given csv_ml data if both DDL and DML are needed. This is shown in the example below:

Java

```
MultiLevelCSVParser parser = new MultiLevelCSVParser();
StringReader sr
      = new StringReader("csv_ml,1.0\nstudent,name,age\n1,abc,23");
DBBind.generateDDL(parser.getSchema(), out_str); // generate DDL
if (out_str.length() > 0) {
   out_str.append("\r\n");
   DBBind.generate_dml_recursively(parser.getSchema(),
         doc.getDocumentElement(), "", out_str); // generate DML
} else {
   out_str.append("No schema");
}
```

API for retrieving data

The API for generating DDL and DML needs a schema object generated from parsing the csv_ml. This can be obtained by just calling initParse, if only DDL is needed or by completely parsing the given csv_ml data if both DDL and DML are needed. This is shown in the example below:

Java

```
Connection con = DriverManager.getConnection("jdbc:sqlite:example.db");
MultiLevelCSVParser parser = new MultiLevelCSVParser();
StringReader sr
    = new StringReader("csv_ml,1.0\nstudent,name,age\n1,abc,23");
parser.initParse(ParsedObject.TARGET_W3C_DOC,  // just parse the schema
    new InputSource(new StringReader(egString)), false);
StringBuffer out_str = new StringBuffer();
String[] arr_id = new String[] {"1"}; // pass id as '1'
DBBind.generateSQL(parser.getSchema(), out_str, con, arr_id);
// The above call generates the SQL, runs it and stores
// the output in the form of csv_ml in out_str
```

API for database binding is not available in javascript as it is not possible to access database directly from Javascript (client-side).

- End of document -