# `fn:parse-csv()` update

## QT4 CG meeting, 11 July 2023

Matt Patterson, Saxonica

# Terminology changes

`records`, `header fields`, & `fields` -> `rows`, `columns`, & `fields`

# Parsing functions

`fn:parse-csv()`
`fn:csv-to-xdm()`
`fn:csv-to-xml()`

# `fn:parse-csv()`

Does nothing more than handle parsing rows according to the delimiter and quoting rules set.

Returns `array(xs:string)*`

# fn:csv-to-xml()

Uses `parse-csv()` and returns a predefined XML structure

```
<csv>
    <header>
        <column>first-name</column>
        ...
        <column>last-name</column>
    </header>
    <row>
        <field column="first-name">value</field>
        ...
        <field column="last-name">value</field>
    </row>
    ...
</csv>
```

# fn:csv-to-xdm()

Uses `parse-csv()` and returns a `parsed-csv-record` record:

```
record(
    header as map {
        "columns": map { 1: "first-name", ... },
        "row": ("first-name",...)
    },
    rows as map {
        "field": function(
            $key as union(xs:integer, xs:string)
        ) as xs:string?
        "fields": xs:string*
    }*
)
```

# Input options

All the functions have the same basic signature

```
function($input as xs:string?, $options as map())
```

The options map shares a core record type:

```
record(
    field-separator? as xs:string,
    record-separator?   as xs:string,
    quote-character? as xs:string,
)
```

# Input options

This record type can be shared with any functions that generate CSV.

All the parse function options maps also have:

| | |
|---|---|
| **`trim-whitespace?`** | **`xs:boolean`** |

# Input options

`fn:csv-to-xml()` and `fn:csv-to-xdm()` additionally take options for filtering the input, and extracting column names from a header row.

```
column-names? union(xs:boolean, map(xs:integer, xs:string))
filter-columns? xs:integer+
number-of-columns? union(enum('first-row'), xs:integer)
```

`filter-columns` and `number-of-columns` are mutually exclusive.

# `column-names`

Controls extraction from a header row, or explicit setting, of column names. Passed an `xs:boolean`, enables extraction of column names from the first row, or switches them off. Passed a map of column number to name, allows explicit setting of column names.

## `filter-columns`

If passed, only returns the specified column numbers in rows.

## `number-of-columns`

Controls the length of returned rows, truncating overlong rows and padding out shorter rows

# Using `csv-to-xdm()`'s response

```
for $row in fn:csv-to-xdm($csv-string) return {
    <tr>{
        for $field in $row?fields return { <td>{ $field }</td> }
    }</tr>
}
```

Or

```
for $row in fn:csv-to-xdm($csv-string) return {
    <tr>{
        <td>{ $row?field('name') }</td><td>{ $row?field('amount') }</td>
        <td>{ $row?field(42)}</td>
    }</tr>
}
```

# using `csv-to-xml()`'s response

```
<xsl:apply-templates select="csv-to-xml($csv-string)"/>
<xsl:template match="csv">
    <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="row">
    <tr><xsl:apply-templates</tr>
</xsl:template>
```

# `csv-to-xml()` XML structure

I am asserting that the schema for the returned XML should be fixed and not changeable by the user.

I am not sure whether it should be put into a special namespace or returned without namespace, because I am not clear on what (if any) negative consequences of no-namespace would be.

# `csv-to-xml()` XML structure

The big question for me is whether to use column names directly in the row-level XML, as already shown, or to use id refs:

```
<csv>
    <header>
        <column id="col-1">first-name</column>
        <column id="col-2">last-name</column>
    </header>
    <row>
        <field ref="col-1">value</field>
        <field ref="col-2">value</field>
    </row>
</csv>
```

```
<xsl:variable name="csv-data" select="csv-to-xml($csv-string)"/>
<xsl:variable name="vip-column" select="$csv-data/header/column[. = 'most important']/@id"/>
<xsl:apply-templates select="$csv-data"/>
...
<xsl:template match="field[@ref=$vip-column]">
    ...
</xsl:template>
```

My preference is for the `@id-@ref` version, not least because I can't see how you could make the `@column` approach work without horrible string mangling.

# `fn:parse-csv()` output

In this version of the proposal, `fn:parse-csv()` simply handles the basic parsing of the CSV string into fields and rows, for the other functions to make use of, and as the basis for users to build more complex / non-standard processing:

```
(
    ['r1f1', 'r1f2'],
    ['r2f1', 'r2f2']
)
```

Based on our previous conversation, it seems that sequence-of-arrays is preferred for the basic format because of potential streamability, and general ease of use with current looping operators and expectations.

# `fn:parse-csv()` data input

It would be nice to be able to not rely on a string for input, given that CSVs can be very large.

`unparsed-text-lines()` doesn't include the line endings, which could be a problem given that line endings can occur inside a field given quoting.

Allowing `fn:parse-csv()` to accept `xs:string*` with the caveat that implementations must not assume that the strings in the sequence represent meaningful chunks (like a whole row).