HTSQL -- a "Native" Web Query Language

Clark C. Evans
Prometheus Research, LLC
cce@clarkevans.com

Abstract

Hyper-Text Structured Query Language (HTSQL) is a standardizable middleware component that translates a HTTP request into an SQL query, performs the query against a relational database, and returns the result as XML, HTML, CSV, JSON, or YAML. HTSQL formalizes a URI-to-SQL translation, covering common database query constructs with a succinct, easy-to-learn syntax. HTSQL decouples the application from the datastore, putting the database itself "on the web".

Keywords: SQL, HTTP, URI, Relational Database

1. Introduction

Relational databases and the Internet are here to stay, and so is their combination. A tedious and recurrent aspect of Internet application development is database access: an HTTP request is sent from a web browser to an application server, which the server then translates into SQL, executes, and responds with the result-set formatted as CSV, HTML, XML, or increasingly, JSON or YAML. Current approaches to this challenge are often application specific, poorly documented, lack expressiveness, or are tightly coupled to a given programming framework. A reusable solution to this pattern will not only speed application development but will also enable a new class of inter-system collaboration opportunities.

Hyper-Text Structured Query Language is both a specification and an implementation of a "native" web query language. Its objective is to provide a near-complete mapping of URIs onto SQL while maximizing readability. HTSQL supports modern file formats and HTTP's best features: REST, error handling, request authentication, compression and encryption. Versions of HTSQL have been deployed in medical laboratories at Yale University for the past three years. The most recent deployment includes a two-tier solution pairing HTSQL with an entirely AJAX data-browser, DBGUI.

In a typical two-tier web architecture, exemplified by many PHP applications, URIs are associated with a page template. Each page description contains several SQL query templates, complete with place-holders for parameters. An HTTP request causes parameter substitution, query execution, and construction of the resulting HTML response based on the template. While it is straightforward to construct applications in this manner, the conflation of user interface with business logic and database query construction leaves much to be desired.

Three-tier architectures, such as Ruby on Rails or Enterprise Java Beans, create an intermediate business object layer between the database and the interface presentation. In these systems, objects are persisted via an object-to-relational mapper. The user interface of these applications is often a page templating system (directly accessing the intermediate objects) or client-side Javascript code using AJAX. As scalable as these solutions may be, database access is necessarily limited to what the business logic tier supports.

An emerging architecture is a hybrid that allows user access at multiple tiers. The first tier is a "native" web query language, such as Google Data or XML Query Language over HTTP. In this scenario, the user interface tier is often client-side Javascript using AJAX to interact with the datastore. A three-tier variant is also viable, where an intermediate business object layer uses HTTP requests instead of SQL to implement persistence. What this new approach enables is profound: direct user access to the full capabilities of the database tier. Unfortunately, current implementations of these datastores rely upon proprietary or non-relational database technology, unnecessarily increasing adoption cost and risk.

While direct access to the database tier (bypassing presentation and middle-tier business logic) is not always desirable, it can be incredibly useful. All too often, end-users of the system would like to get at their data without having to wade through the constraints of the application's chrome or they would like to hire someone else to write a different report or work-flow screen. Direct access to the datastore permits transparent application extensibility. Further, this approach provides enormous performance advantages to collaborative applications, or "mash-ups"; screen scraping and the maintenance of extracted data-sets for local processing are no longer necessary.

HTSQL works with modern web browsers and open source relational databases, such as PostgreSQL, (and soon) SQLite and MySQL. HTSQL is open source software. The specification and implementation are found at `http://htsql.org/`.

Example 1. A Trivial HTSQL Example

```
/project
```

```
SELECT p.*
  FROM project p
ORDER BY p.proj_id
```

In these examples we assume a simple task-tracking schema having a `project` and a `task` table. There is a many-to-one relationship between these tables such that every task has exactly one project. The primary key of the project table is a `proj_id` text column. The primary key of the task table is `proj_id` and `task_no` where proj_id refers to the corresponding column in the project table and `task_no` is an integer.

2. Design of HTSQL

The target audience for HTSQL is the *accidental programmer*, a person who customizes and tweaks software, a power-user. These individuals are not software engineers; instead, they are system administrators, business analysts -- even professional accountants or medical researchers. These people know their data inside and out. They are the ones who play with URIs just to "see what happens". It is an explicit goal for a URI-based SQL access language to be intuitive and helpful; completeness or even consistency is not nearly as important as usability.

This section covers the design of the HTSQL language, explaining how various features of SQL are covered, why particular decisions were made and what is their impact upon usability. We start with the most tedious detail, quoting and escaping.

2.1. Character Set, Escaping & Strings

By definition, HTSQL is a URI scheme, and therefore it must follow the syntax of RFC 2396 and its successor, RFC 3986. For readability, HTSQL employs several characters in the *unwise* or *reserved* categories. Conversely, some characters HTSQL uses for operators or delimiters are *unreserved*. Therefore, HTSQL does not use percent-encoding for syntactic escaping as anticipated by RFC 3986. Instead, it relies upon standard UTF-8 percent-encoding solely for transmission, such as inclusion in an HTML `href` attribute.

Following SQL's lead, HTSQL has two kinds of strings: literal values and catalog identifiers. Literal strings in HTSQL are single-quoted (`'`), doubling the single quote when one occurs in a value, `'O''Reilly'`. As in SQL, numeric literals need not be quoted. Note that "C"-style escaping is not required since URI percent-encoding can be used to represent non-printable or non-ASCII characters. However, a single quote cannot be escaped in this manner since decoding is performed before parsing.

Catalog identifiers, e.g., table or column names, can always be double-quoted (`"`), although this is seldom necessary. Following SQL's precedent, HTSQL catalog names can be left unquoted when a case-insensitive comparison uniquely matches a given table or column name. Furthermore, the underscore (_) can be used to match a space, dash, or any other non-alphanumeric character; `test_3` matches tables named `TEST-3` and `Test 3`. This matching scheme permits HTSQL identifiers to serve as valid Javascript identifiers. It also permits HTSQL URIs to be used in double-quoted XML/HTML attributes without percent-encoding.

Earlier passes of the HTSQL grammar limited indicators to the reserved *sub-delims* character set as described in RFC 3986 and was solely dependent upon percent-encoding for string escaping. This solution used spelled out functions for basic operators (e.g less-than() instead of the unwise `<` character). Common queries were verbose and confusing, requiring casual users to code expressions with reverse polish notation rather than the more usual infix notation. Worse, common URI-handling functionality in browsers and URI fetching APIs were often uncooperative: they would percent-encode (or un-encode) characters where the encoding was significant. After significant hair loss, HTSQL uses percent-encoding solely for transport-level encoding.

2.2. Predicate Expressions

The usual URI query format, HTML FORM submission, is a sequence of `name=value` pairs delimited by an ampersand (`&`). While this model is very successful, it intuitively limits queries to a conjunction of comparisons between column names and values. Although one could work creatively to introduce full predicate power into this limited syntax, the result would require variable definition and references and might be generally unreadable. A break from this tradition is needed.

Query parameters in HTSQL emulate the HTML FORM tradition in that the ampersand (`&`) signifies conjunction and the equal sign (`=`) signifies equality. The similarity stops there. Following the "C" language tradition, HTSQL adds the vertical bar (`|`) for alternation, the exclamation mark (`!`) for negation, and parentheses to indicate scoping. Further, the right-hand side vs. left-hand side distinction is also discarded, permitting expressions based on single-quoted literals or unquoted column names on either side of an operator.

In addition to the differences above, HTSQL adds a function call syntax, e.g., `lower()` as well as intro-

ducing the standard comparison and mathematical operators. POSIX regular expressions are specified via a tilde (~); a single tilde is case-insensitive and a double tilde is case-sensitive. If a given database does not directly support POSIX regular expressions, an HTSQL processor should attempt to convert regular expressions to the corresponding SQL92 `LIKE` or SQL99 `SIMILAR TO` expressions. Direct support for these SQL operators is not provided.

Example 2. Filtered List of Tasks

```
/task?status~'done'|hours>0.5
```

```
SELECT t.* FROM t.task
WHERE LOWER(t.status) LIKE '%done%'
   OR t.hours > 0.5
ORDER BY t.proj_id, t.task_no
```

Although HTSQL URIs do not follow HTML FORM semantics, they can be constructed with Javascript and submitted via XMLHTTPRequest. To support standard HTML form submission, HTSQL honors POST requests having the `multipart/form-data` (RFC2388) or `application/x-www-form-urlencoded` (HTML32) mimetype. In these cases, the left-hand side of each query-argument pair is assumed to be a column name; the right-hand side, a literal. Following the column name is an optional operator, e.g., `proj_id~` requests a case-insensitive regular-expression search on `proj_id`.

2.3. Join Specifiers

In typical database usage, foreign key relationships denote a valid join. Although versions of SQL up to SQL 2003 do not provide an explicit syntax for automatic join construction, the comments to the SQL 2003 specification describe this idea. In HTSQL, a *specifier* is a reference to a column of the current table, or it is a series of links to a related table or to a column in a related table. These links are delimited by the period (`.`), where each period represents an SQL join to be constructed. Every link has a a name and is defined as a pairing of columns needed to construct the corresponding join.

The HTSQL reference implementation uses SQL92's `INFORMATION_SCHEMA` to discover foreign key constraints and to define links based on the implied relationships. When possible, a foreign key constraint is named for the table it references. If more than one link to the referenced table is possible, the links are named after the corresponding columns. If this method does not yield a unique name, then the foreign key constraint's name is used. This default configuration can be manually overridden.

Example 3. Filtering by a Specifier

```
/task?project.is_closed
```

```
SELECT t.* FROM task t
  JOIN project p
    ON (p.proj_id = t.proj_id)
    WHERE p.is_closed
ORDER BY t.proj_id, t.task_no
```

While foreign key constraints from the referencing table to the referent provide a *singular* relationship, it is possible to reverse the join direction. In this case, for each row being be returned in the result set, more than one row in the referencing table is matched. This is a *plural* join specifier. HTSQL's semantics evaluate plural cardinality up the expression tree as far as possible, permitting queries that test for a condition in a subordinate table. Eventually, to be evaluated, a plural specifier must become singular. This is either done with an explicit aggregate function such as `sum()` or implicitly via `exists()`.

Given `/project?task.status='done'`, an HTSQL processor returns all `project` rows where there exists at least one corresponding `task` having a `status` of `'done'`. This behavior requires moderately complex bookkeeping to determine contexts where evaluation is occurring. Each context corresponds to a correlated sub-query in the generated SQL. The generated SQL will vary according to the placement of functions that convert a plural to a singular value. The following example returns projects that have a total of at least three hours of time spent on all of their subordinate tasks.

Example 4. Projects Filtered by Aggregate

```
/project?sum(task.hours)>3.0
```

```
SELECT p.* FROM project p
 WHERE (SELECT SUM(t.hours) FROM task t
    WHERE p.proj_id = t.proj_id) > 3.0
ORDER BY p.proj_id
```

2.4. Column Selectors and Projections

In HTSQL, result-set column selection is accomplished using curly braces, for example, `project{name,title}`. By default, result sets are sorted according to the table's primary key. If an expression has a trailing plus (`+`) or minus (`-`), then the sort is ascending or descending, respectively, on that expression. A selector lacking a preceding table can be used to return a scalar, e.g., `{now()}`.

Aggregate expressions are indicated in a column selector with a vertical bar (`|`). Expressions before the

vertical bar are selected and used in the corresponding `GROUP BY` clause. While the general SQL syntax does not conflate selecting a column and grouping by it, HTSQL does because the two operations are highly correlated in practice and superfluous columns can be easily ignored.

We call the vertical bar *projection* since it causes the correlation between the set of rows returned and rows from the referenced table to differ. In the following example, rows are returned for each unique pair of `project.name` and `status` specifiers. A `DISTINCT` result is therefore obtained by using the projection indicator without providing an aggregate expression, `task{project.name,status|}`. This syntax overloading requires that alternations must be surrounded by parentheses to be used in a selector.

Example 5. Selecting Aggregate Expressions

```
/task{project.name,status-|sum(hours)}
```

```
SELECT p.name, t.status, sum(t.hours)
  FROM task t JOIN project p
    ON (p.proj_id = t.proj_id)
GROUP BY p.name, t.status
ORDER BY t.status DESC, p.name
```

2.5. Row Locators

The most common operation in a database application is the retrieval of a row by primary key. Since an HTSQL processor is configured to use primary key constraints, naming the columns is redundant. Therefore, a request such as `task?proj_id='MEYERS'&task_no=1` can be shortened significantly to `task[MEYERS.1]`.

This *locator* syntax is inspired by the domain name system (DNS). A row locator is indicated with matching square brackets (`[ ]`) and contains a comma-separated list of row locations. Each location is a series of primary key column values ("*label*s"), delimited by a period (`.`). Unquoted labels are compared in a case-insensitive manner, similar to unquoted catalog identifiers. If a case-sensitive match is needed to uniquely identify a given row, then the label should be single quoted.

Example 6. Locating rows via Primary Key

```
/task[meyers.1]
```

```
SELECT t.*
  FROM task t JOIN project p
    ON (p.proj_id = t.proj_id)
 WHERE htsql_norm(p.proj_id)
     = htsql_norm('meyers')
   AND t.task_no = 1

htsql_norm(x) :=
TRANSLATE(TRIM(BOTH ' ' FROM
 LOWER(CAST($1 AS TEXT)> )),
 ' ~`!@#$%^&*()-_={}[]|\:;"<>,.?/''',
 '--------------------------------')
```

When a table's primary key contains a foreign-key reference to another table, then those columns are not included in the locator. Instead, the referenced table's locator is used. This extra join permits the referenced table's locator to differ from the primary key columns of the referent, although the extra join can be optimized away in the SQL query above. If the referenced table's locator has more than one label, then it is grouped in parentheses. For example, a cross-product `task_dependency` table with a primary key having exactly two `task` table references might have a locator like `(MEYERS.1).(SMITH.8)`.

In this locator syntax, unquoted (case-insensitive) labels are limited to alphanumeric characters plus the hyphen (`-`), which matches a single whitespace or non-alphanumeric character. This choice of wildcard is informed by the usual occurrence of the hyphen in product codes or serial numbers (e.g., `EH-348-X`). The label syntax is also compatible with ISO 8601 dates (e.g., `2007-06-25`), encouraging use of this standard date format.

The `id()` function returns the locator for the current row. Text columns are single-quoted by default, since there may be some rows which differ only by case. Integer, date, and boolean columns, or columns explicitly configured otherwise, are not quoted.

2.6. Path Segments

Relative URIs, indicated with the forward-slash `/`, intuitively support drill-down behavior. In this resource navigation technique, a new path segment is appended to the URI as one traverses deeper into a hierarchical structure. For HTSQL, each path segment corresponds to a table reference, together with an optional specifier, row locator, and filter expression. Filter expressions in a path segment context are indicated by the semi-colon (`;`) instead of the question mark.

When two or more path segments are used, they must be connectable via a *primary* link, where the descendant's primary key contains a reference to a candidate key of the parent table. The resulting SQL joins both tables, asserting that exactly one possible link exists.

Example 7. Joining via Path Segments

```
/project;!is_closed/(+)task
```

```
SELECT p.proj_id AS "project.proj_id",
       p.name    AS "project.name",
       ...
       t.proj_id AS "task.proj_id",
       t.task_no AS "task.task_no",
       t.hours   AS "task.hours",
       ...
  FROM project p
       LEFT OUTER JOIN task t
    ON (p.proj_id = t.proj_id)
 WHERE NOT p.is_closed
ORDER BY p.proj_id,
  t.proj_id, t.task_no
```

This syntax has optional join modifiers. If a segment's table is prefixed by "`(+)`" then an `OUTER JOIN` is used. If "`(*)`" is indicated, the tables are not joined and a cross product is performed instead. With the *mask* modifier, "`(!)`", the path segment is ignored unless a subsequent path segment can be connected via a series of links. This mask syntax facilitates an unobtrusive yet effective row-level permission mechanism.

Path segments can also have their own selector. By using an empty column selector, a path segment can be used to filter without returning results. For example, `/project[meyers]{}/task` returns tasks in the `meyers` project but does not return columns for that project. In HTSQL 1.0, segment selectors containing projections are not permitted.

2.7. Commands, Assignment, and Lookups

The default operation for an HTSQL request is the `select()` command without arguments, corresponding to an SQL `SELECT` statement. An explicit command can instead be provided as the last segment of a URI. For example, rows 100-149 of a result set can be returned using the explicit, parameterized `SELECT` command, `select(limit=50,offset=100)`. Besides `select()` command, HTSQL also supports `insert()`, `update()`, and `delete()`, among others.

Example 8. Insert Statement

```
/project[newprj]
  /insert()?name:='New Project'
```

```
INSERT INTO project (proj_id, name)
VALUES ('newprj', 'New Project')
```

To support data modification, the assignment operator (`:=`) is used. In an `update()`, the traditional equality operator is used to limit the affected roles while the assignment operator is used to set values. In an `insert()` command, locators imply assignment rather than a test for equality.

Example 9. Update Statement

```
/project[newprj]
  /update()?name:='Different Name'
```

```
UPDATE project
   SET name = 'Different Name'
 WHERE htsql_norm(proj_id)
     = htsql_norm('newproj')
```

HTSQL provides a mechanism to make assignments by specifier, which then affects the foreign key columns of the corresponding link. This feature removes the need to track each table's primary key and foreign key columns, reducing error and improving readability. To make this work, HTSQL has a lookup operator (`@`). This syntax is particularly powerful for multi-column foreign-key assignments or in cases where the foreign key of the referencing table does not directly correspond to the primary key of the referent.

Example 10. Assignment via Lookup

```
/task[meyers.1]/update()?
    assigned_to:=@employee[john]
```

```
UPDATE task
   SET assigned_to =
     (SELECT empl_code
        FROM employee
        WHERE htsql_norm(empl_code)
            = htsql_norm('john'))
WHERE (proj_id, task_no) IN
  (SELECT t.proj_id, t.task_no
     FROM task t
     JOIN project p
       ON (p.proj_id = t.proj_id)
    WHERE htsql_norm(p.proj_id)
        = htsql_norm('meyers')
      AND t.task_no = 1)
```

2.8. Users, Roles, and Conflicts

Since HTSQL uses HTTP as a base protocol, it assumes that the `REMOTE_USER` gateway variable corresponds to an actual database user and executes `SET SESSION AUTHORIZATION` to shed any administrator privileges for the subsequent query. The user's `role`, on the other hand, can be set via HTSQL URI using a tilde in the first path segment, such as `/~clerk/project`. This causes `SET ROLE 'clerk'` be performed before any query is executed. Note that tables, columns, domains, and any transitively associated links are affected by setting the user and role. Role usage in this manner provides stable and sharable URIs that reference the same resource across users who would normally have different permission sets.

In HTSQL, if a table is configured to have a "version" column, optimistic locking can be used. In this case, each locator can be followed with a version tag, delimited with the semi-colon (`;`). For example, `[meyers.1;4]` might refer to the 4th revision of the a given task. When a command uses a locator having the version tag, the HTSQL processor additionally verifies that the column (usually a sequence or timestamp) associated with the version tag matches. The `id(version=1)` function returns a row locator with its version tag.

2.9. Types and NULLs

HTSQL has a powerful type system, designed to be intuitive for casual users. Quoted literal values and `null()` are not assigned a type immediately; instead, the type is inferred from context. Automatic type conversion only occurs in the presence of boolean operators (including the implicit top level conjunction). As in Python, boolean conversion treats empty strings, arrays lacking elements, zero intervals, and zero numeric values as `FALSE`. This careful balance yields succinct expressions without sacrificing type safety.

Boolean operators treat `NULL` values according to SQL rules. HTSQL extends this behavior by defining the boolean value of a non-boolean `NULL` expression to be `FALSE`. This permits concise foreign-key checking, e.g., `/task?!assigned_to` would include rows from the `task` table where the foreign key `assigned_to IS NULL`. However, for boolean-valued columns, this transformation does not apply. Hence, `/project?!is_closed` will not include rows where `is_closed IS NULL`.

To facilitate `NULL` comparison, HTSQL employs a doubled equal sign (`==`) to mean SQL99 `IS NOT DISTINCT FROM` operator. In the above example, one could check for both `FALSE` and `NULL` rows on a boolean-valued column with the expression `project?is_closed!==true()`. HTSQL uses functions such as `true()`, `false()`, and `null()` to represent SQL constants. Functions such as `coalesce()` or `is_null()` reflect the corresponding SQL construct.

2.10. Namespaces and Objects

Namespaces in HTSQL follow XML's lead, using the colon (`:`) as a delimiter. By default, function, command, and table references need not include the namespace if they are otherwise unique. However, if a given user has access to two different tables named `project`, a reference should use the schema's catalog identifier, e.g., `tm:project` to specify the `project` table in the `tm` schema. Standard functions reside within the `htsql:` namespace and user-defined functions or commands can be registered via plugins and accessed through other namespaces.

HTSQL support methods and attributes on objects. For example, calling SQL's `CHARACTER_LENGTH` function is accomplished through a `length()` method on a text column, e.g., `project{proj_id,name.length()}`. For the `DATE` data type, the `day`, `month`, and `year` are accessed through attributes, rather than using the unwieldy `EXTRACT` function. For example, `task{due_date.year|count()}` lists the number of tasks grouped by the year they are due.

User-defined types are supported with a direct translation from HTSQL into its SQL equivalent. In HTSQL, ARRAYs are constructed using `array()`, and ROWs through `row()`. Assignment to a user-defined composite type is done by pairing a selector listing field labels to a selector listing the values. For example, `some_table?udt_column{field_one, field_two} := {'value_one', 'value_two'}`.

2.11. Output Formats

HTSQL result sets are returned in industry-standard formats, including TXT, XML, CSV, JSON, HTML, and YAML. The default format is determined by HTTP Content-Type negotiation, or if that fails, is a whitespace-padded plain-text format for debugging and testing. An explicit format can be requested by using a standard "file extension" on the referenced table, e.g., `/task.xml`.

If there is more than one path segment provided, and if the result format is hierarchical (such as XML), then nested element structures are used. For data-manipulation operations, such as an `update()`, if a column selector is included with the referenced table, the affected rows are returned. This corresponds to PostgreSQL's `RETURNS` clause, a very helpful exten-

sion to SQL which the author hopes will be included in future versions of the standard.

3. Conclusion

Starting with an initial vision of a coherent URI-to-SQL translator over 4 years ago, HTSQL has matured in numerous ways and has proven itself to be a valuable component in a rapid web application development architecture. Beyond what was discussed above, HTSQL has facilities for details like transactions, as well as pluggable commands for things like mail merging and label generation. HTSQL has made development faster and more reliable than previous methods and facilitates re-usable modules and database schemas.

While it provides only anecdotal evidence, one user (a medical researcher at Yale University) is well known for "fiddling" with HTSQL URIs to get the data he wants. After a few months of using our system though the user interface, he now frequently ignores the HTML and Javascript UI and goes directly for the URIs. He did this without a manual and without any encouragement. The adoption of database-tier access by the accidential programmer will be tested more this year, as the system is deployed to additional sites. We think that HTSQL will rise to the challenge.

4. Acknowledgments and References

Support for this research was provided by a generous award from the Simons Foundation and by Prometheus Research. This work would not have been possible without Python, PostgreSQL, and the amazing group of people that support these projects. The author wishes to thank Kirill Simonov for his thoughtful contributions to the design of HTSQL and his excellent implementation. Finally, researchers at the Yale Child Study Center deserve thanks for braving earlier versions of HTSQL.

[1] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, and Larry Masinter. *Hypertext Transfer Protocol—HTTP/1.1 . IETF RFC*. 2616. June 1999.

[2] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform resource identifiers (URI): Generic syntax*. RFC 2396. 1998. http://www.ietf.org/rfc/rfc2396.txt.

[3] Tim Berners-Lee, Roy T. Fielding, and Larry Masinter. *Uniform resource identifier (URI): Generic syntax*. RFC 3986. January 2005. http://www.ietf.org/rfc/rfc3986.txt.

[4] David Raggett. *HTML 3.2 Reference Specification*. {W3C} recommendation. W3C. January 1997. http://www.w3.org/TR/REC-html32-19970114.

[5] Larry Masinter. *Returning Values from Forms: multipart/form-data*. RFC 2388. August 1998. http://www.ietf.org/rfc/rfc2388.txt.

[6] International Organization for Standardization. *ISO/IEC 9075:1992 Database Language SQL*. 1999.

[7] International Organization for Standardization. *ISO/IEC 9075-2:1999: SQL Part 2: Foundation*. 1999.

[8] International Organization for Standardization. *ISO/IEC 9075-2:2003 SQL Part 2: Foundation*. 2003.

[9] PostgreSQL Development Group. *PostgreSQL Relational Database*. http://postgresql.org.

[10] Python Development Community. *Python Language*. http://python.org.

[11] Douglas Crockford. *The application/json media type for JavaScript Object Notation (JSON)*. RFC 4627. July 2006. http://json.org/.

[12] Yakov Shafranovich. *Common format and mediit type for comma-separated values (CSV) files*. RFC 4180. October 2005.

[13] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. *YAML Ain't Markup Language (YAML) v 1.1*. December 2004. http://yaml.org/spec/.

[14] Tim Bray, Jean Paoli, and C.M. Sperberg-McQueen. *Extensible Markup Language (XML)*. Febuary 1998.

[15] ECMA. *ECMA-262: ECMAScript Language Specification*. ECMA (European Association for Standardizing Information and Communication Systems)December 1999. http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf.

[16] Google, Inc.. *Google Data APIs Protocol Reference*. Google, Inc. 2006. http://code.google.com/apis/gdata/reference.html.

[17] Scott Boag, Don Chamberlin, Mary Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. *XQuery 1.0: An XML Query Language*. W3C January 2007. http://www.w3.org/TR/xquery/.