C. Evans
Prometheus Research, LLC.
August 2006

# HyperText Structured Query Language

## Abstract

HyperText Structured Query Language (HTSQL) is an extension to the HTTP/1.1 protocol that allows clients to remotely access a standard SQL database. This extension provides a mapping of HTTP requests into SQL statements, producing a response that corresponds to the result generated by the database. This document defines a URI scheme used for this mapping, together with a coherent set of HTTP methods, headers, and entity body formats.

# Table of Contents

# 1. Introduction

HyperText Structured Query Language (HTSQL) is a mechanism for accessing industry standard SQL [ISO9075-1992] data sources over HTTP [RFC2616]. This document specifies a URI scheme [RFC3986], HTTP methods and extensions which enable database access from standard web browsers. The principal advantage of HTSQL is the expression of queries in a concise web-friendly syntax: for for common database tasks, path-based URIs are both simple and intuitive. A secondary advantage of HTSQL is the integrated use of the HTTP protocol to provide authentication, data caching, encryption, content negotiation, and numerous other network operations. This approach to SQL-over-HTTP puts forth a reusable, application independent, and testable middleware layer which translates HTTP requests into SQL statements, returning the execution results to the user in a format their user-agent can handle.

## 1.1 Rationale

The target audience for HyperText Structured Query Language is not career programmers, nor is it casual users. HTSQL is designed for technical users including screen designers, database administrators, statisticians, medical researchers, and other "accidental programmers". HTSQL advances a human-friendly URI-based query syntax over traditional SQL queries, and HTTP over a more typical database access protocol. HTSQL offers the following:

URIs are instantly familiar to users who have been using the web for many years; human-readable URIs provide direct control over database information which is often lacking in traditional systems.

A database accessable via a web browser with persistent URIs allows query results to be bookmarked and emailed to collaborators; such an interface also enables easy navigation using web browser controls.

HTSQL offers greater flexibility than a purely graphical user interface, which necessarly limits the kinds of retrievals that can be specified. With even moderate exposure, an advanced user can learn to modify a URI to achieve results beyond what a graphical interface may provide.

HTSQL builds upon existing standards. By using standard HTTP, the database access protocol need not be burdened with encryption, signatures, cache control, content types, authentication, detailed audit-trails, or other network-related issues.

HTSQL provides increased security: an HTTPS interface enables encrypted database access, while at the same time providing a proxy for database activity monitoring.

Implicit in this approach is a compromise. While 90% of common database tasks can be expressed in human-readable URI format, the other 10% will necessarly be elusive. In those uncommon cases, client-side processing or server-side views are simple alternatives. While HTSQL is targeted for occassional programmers, it is also allows career programmers to quickly develop and deploy loosely coupled applications.

## 1.2 Objective

While access over HTTP is a very common feature of today's database systems, most implementations are application specific and fail to take full advantage of HTTP protocol. By providing a hybrid that is both application-independent and human-readable, HTSQL realizes the potential synergy of HTTP and SQL.

HTSQL should support most SQL-92 operations, although full coverage of SQL functionality is explicitly not a goal. Database views and/or triggers supplement HTSQL to satisfy unsupported needs.

For all but the most complicated database interactions, the corresponding URIs in HTSQL must be easy to read and understand.

Since HTSQL is meant to be used by casual programmers, error messages must be informative and layered so that casual users are not frightened and expert users are given the details they need.

HTSQL must use HTTP/1.1 features for well-known operations such as authentication, caching, range requests and content negotiation; HTSQL should extend or augment HTTP/1.1 only when necessary.

HTSQL must allow for fine-grained access permissions as allowed by SQL-92, mapping application users onto specific database accounts.

HTSQL requests must have a syntax that permits them to be embedded in languages such as Python and Java, or included into page templating languages such PhP.

HTSQL must support the standard [HTML] FORM element for common database operations using HTTP/1.0, requiring full HTTP/1.1 only when necessary.

HTSQL should support a wide variety of query result formats, including JavaScript Object Notation [JSON] and the eXtensible Markup Language [XML] for standard Javascript and DOM enabled web browsers, as well as Comma Separated Variable [CSV] for spreadsheets and data analysis tools.

HTSQL should minimize configuration using information available in the database catalogue; basic functionality should not require supplementary information.

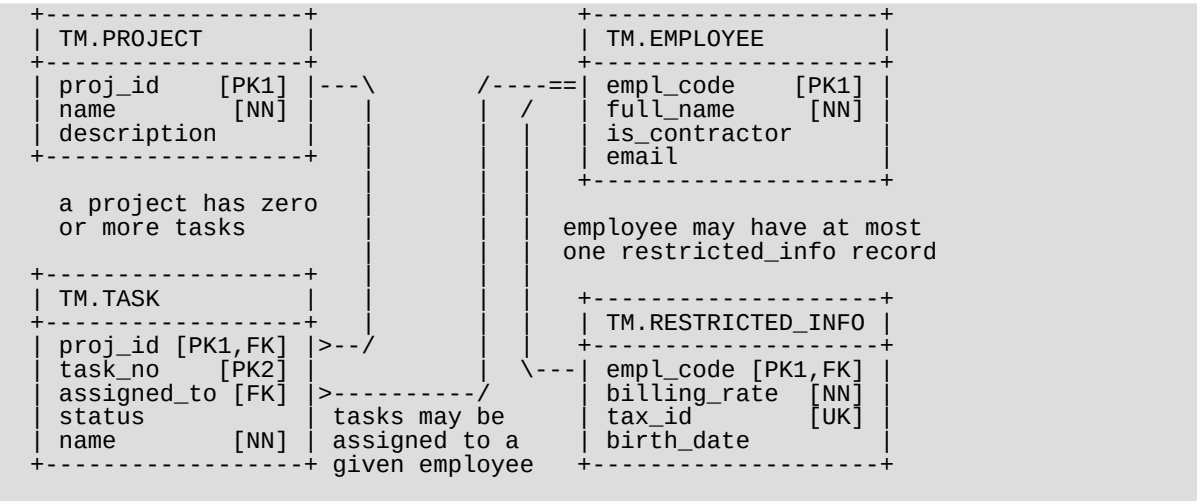
HTSQL should minimize server-side state; following as much as possible the principles of representational state transfer, [REST]. To allow usage of HTSQL from a browser's location bar, database updates may be submitted with the `GET` method; however, this relaxation can be disabled.

HTSQL must support [UNICODE] and use [ISO8601] style dates.

The HTSQL specification should permit usage of SQL-99 [ISO9075-1999] and SQL-2003 [ISO9075-2003] constructs when possible, but should not require these features for operation.

# 2. Preview

As a prelude to the formal specification, we provide a taste of HTSQL by presenting a limited set of URIs, associating each URI with an equivalent SQL expression. For these examples, we use a simple task-management schema representing projects, employees, and tasks.




In this `tm` schema, `project` and `employee` are top-level tables with primary keys `proj_id` and `empl_code` respectively. Each `task` is associated with a mandatory project, and optionally assigned to a given employee. Conversely, each employee may have multiple tasks assigned to them. While there may be more than one task per project, there is at most one `restricted_info` record per employee.

In this diagram, each table's **primary key** is marked with `[PK]` and any **unique key** is marked with `[UK]`. These *candidate keys* are a combination of columns which uniquely identify each row in the given table. Relationships between tables, drawn with a line, represent **foreign key** constraints. A foreign key defines a correspondence between rows in the source table (marked with `[FK]`) and rows in the referenced table. Mandatory columns are indicated `[NN]`; primary key columns are mandatory as well.

## 2.1 Simple Queries

An HTSQL **request** URI typically starts with a single *table* (optionally prefixed by a *schema*). Following a table expression is an optional filter expression, denoted with a question mark, to limit the rows returned.

`/tm:project`

This request selects all rows from the `project` table within the `tm` schema. By default, rows are ordered by the primary key, in this case, `proj_id`.

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

proj_id  |        name        |          description
---------+--------------------+-------------------------------
MEYERS   | Meyer's Residence  | insulation and winterizing
SSMall   | South Square Mall  | two new store fronts
THOM-LLP | Tom Thompson, LLP. | fix up room for new associate
...
```

`/tm:project?proj_id=='MEYERS'`

Scalar values, such as `'MEYERS'`, are single-quoted to distinguish them from column references, such as `proj_id`. In HTSQL, the equal-equal (`==`) operator represents **strict equality**.

```
SELECT * FROM tm.project
```

```
 WHERE proj_id = 'MEYERS'
ORDER BY proj_id

proj_id |       name        |        description
--------+-------------------+---------------------------
MEYERS  | Meyer's Residence | insulation and winterizing
```

`/tm:employee?empl_code<'B'`

Standard comparison operators less-than (`<`), greater-than (`>`), less-than-or-equal-to (`<=`), and greater-than-or-equal-to (`>=`) are translated to the equivalent SQL construct.

```
SELECT * FROM tm.employee
 WHERE empl_code < 'B'
ORDER BY empl_code

empl_code |  full_name   | is_contractor |       email
----------+--------------+---------------+-------------------
ADAM      | Adam O'Brien | FALSE         | adam@example.com
ARONSON   | Mary Aronson | FALSE         | mary2@example.com
...
```

`/tm:project?proj_id='ssmall'`

HTSQL's **usual equality** operator, indicated with a single equal (`=`), denotes a case insensitive comparison. Leading zeros and spaces are ignored for interoperability with fixed-width formats. This normalization also employs the underscore to match word separation characters such as the space and dash.

```
SELECT * FROM tm.project
 WHERE htsql_normalize(proj_id) = htsql_normalize('ssmall')
ORDER BY proj_id

proj_id |       name        |     description
--------+-------------------+---------------------
SSMall  | South Square Mall | two new store fronts
```

The usual equality operator provides a succinct syntax for common pattern-matching requirements. While not described in a SQL specification, the functional representation above, `htsql_normalize()`, has an unambiguous definition using SQL-92 constructs.

`/tm:employee?full_name~'Ron'`

In HTSQL, pattern matching is requested with [POSIX_1003.2] extended regular expressions, indicated by tilde-equal (`~=`) or with its case-insensitive variant, a single tilde (`~`).

```
SELECT * FROM tm.employee
 WHERE LOWER(full_name) LIKE '%ron%'
ORDER BY empl_code

empl_code |  full_name   | is_contractor |       email
----------+--------------+---------------+-------------------
ARONSON   | Mary Aronson | FALSE         | mary2@example.com
SMITH     | Ron Smith    | TRUE          | john@example.com
...
```

Since comparison using SQL92's LIKE or SQL2003's SIMILAR-TO is easily represented as a regular expression, HTSQL makes no provision to support these operators. For databases that do not natively support regular expressions, common cases can be translated into a corresponding SQL construct -- as shown in the example above.

`/tm:employee?full_name!~='Smith'`

Operators may be prefixed with exclamation (`!`) to negate comparison. In this example, a case-sensitive regular expression excludes rows where the `full_name` contains `'Smith'`. Simple regular expressions such as `'Smith'` can be expressed in SQL-92 using the LIKE operator.

```
SELECT * FROM tm.employee
 WHERE NOT full_name like '%Smith%'
ORDER BY empl_code

empl_code |  full_name   | is_contractor |       email
----------+--------------+---------------+-------------------
ADAM      | Adam O'Brien | FALSE         | adam@example.com
ARONSON   | Mary Aronson | FALSE         | mary2@example.com
...
```

`/tm:task?proj_id='Meyers'&task_no=1`

A conjunction (boolean AND) is indicated with ampersand (`&`). Observe that numeric literal values, such as `1`, need not be quoted.

```
SELECT * FROM tm.task
 WHERE htsql_normalize(proj_id) = htsql_normalize('meyers')
   AND htsql_normalize(task_no) = htsql_normalize('1')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status |        name
--------+---------+-------------+--------+--------------------
MEYERS  |       1 | ARONSON     | done   | Purchase Materials
```

`/tm:task?status=='done'|status=='review'`

Alternation (boolean OR) is indicated with the vertical bar (`|`).

```
SELECT * FROM tm.task
 WHERE status = 'done'
    OR status = 'review'
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status |        name
--------+---------+-------------+--------+---------------------
MEYERS  |       1 | ARONSON     | done   | Purchase Materials
MEYERS  |       2 | SMITH       | review | Strip Wall Paint
SSMall  |       1 | ADAM        | review | Install Slider Door
...
```

To enhance human-readability of URIs, HTSQL uses a number of characters from [RFC2396]'s *unwise* character set. Some browsers may change these characters (such as the vertical bar above) into their percent-encoded equivalent (in this case `%7C`). This does not affect the interpretation of the request.

`/tm:task?status='done','review'`

The comma (`,`) is used to construct a **list**. When compared to a scalar, the result is true if any of the pairings are true. Hence, the filter above is logically equivalent to `status='done'|status='review'`.

```
SELECT * FROM tm.task
 WHERE htsql_normalize(status) in ('done','review')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status |        name
--------+---------+-------------+--------+---------------------
MEYERS  |       1 | ARONSON     | done   | Purchase Materials
MEYERS  |       2 | SMITH       | review | Strip Wall Paint
SSMall  |       1 | ADAM        | review | Install Slider Door
...
```

`/tm:task?status&!assigned_to`

To test for a non-empty value, column references (such as such as `status` and `assigned_to`) can be used in a filter without an explicit comparison operator. This particular example returns tasks which have been given a status, but have not yet been assigned to an employee.

```
SELECT * FROM tm.task
 WHERE (status IS NOT NULL AND status != '')
   AND (assigned_to IS NULL OR assigned_to = '')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  |     name
--------+---------+-------------+---------+--------------
MEYERS  |       3 | (NULL)      | planned | Remove Refuse
...
```

Expressions which evaluate to the empty string, to a zero value, or to NULL are considered `false`. In the example above, `status` is equivalent to `(!isnull(status)&status!='')`.

```
/tm:task?assigned_to&(status='done'|status='review')
```

Parentheses can be used to group boolean expressions. This request returns tasks that are not only assigned to an individual, but also have a `status` of `'done'` or `'review'`.

```
SELECT * FROM tm.task
 WHERE (assigned_to IS NOT NULL AND assigned_to != '')
   AND (htsql_normalize(status) = 'done'
     OR htsql_normalize(status) = 'review')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status |        name
--------+---------+-------------+--------+---------------------
MEYERS  |       1 | ARONSON     | done   | Purchase Materials
MEYERS  |       2 | SMITH       | review | Strip Wall Paint
SSMall  |       1 | ADAM        | review | Install Slider Door
...
```

As in SQL, conjunction has higher precedence than the alternation. The parentheses above are necessary: if omitted, the results would return tasks with `'review'` status regardless of whether they had been assigned.

With moderate exposure to these sorts of URIs and the corresponding results, the occasional programmer should be able to combine operators in a generative manner without assistance. Filter expressions can be combined with boolean operators and grouped with parentheses to generate arbitrarily complex predicates.

## 2.2 Specifiers

HTSQL provides a mechanism for referencing columns not only in the current table, as seen in prior examples, but also columns from related tables. In SQL, relationships between tables are declared with a foreign key constraint; a **specifier** associates rows from related tables by joining on these constraints. Specifiers are written as sequence of column and/or table names separated by periods, each period representing a join. Specifiers with two or more periods reflect a transitive join, forming a path from one table to another through intermediates.

```
/tm:task?assigned_to.is_contractor=true()
```

This request returns tasks that are assigned to a contractor. Because there is a foreign key from `assigned_to` in the task table to `empl_code` of the employee table, the HTSQL processor automatically constructs the appropriate join logic.

```
SELECT t.*
  FROM tm.task AS t
       JOIN tm.employee AS e
         ON (e.empl_code = t.assigned_to)
 WHERE e.is_contractor IS TRUE
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status |       name
--------+---------+-------------+--------+-----------------
MEYERS  |       2 | SMITH       | review | Strip Wall Paint
...
```

Since `is_contractor` is a boolean value, it can be tested directly without an operator; hence, this example could be written `/tm:task?assigned_to.is_contractor`. Further, when only one foreign key reference exists to a table, the name of the referenced table (`employee`) can be used instead of the referencing column (`assigned_to`), i.e. `/tm:task?employee.is_contractor`.

`/tm:employee?restricted_info.billing_rate>20`

This request returns employees who have a billing rate of more than 20. The automatic join here is possible since the `restricted_info` table has a foreign key which refers to the `employee` table.

```
SELECT e.*
  FROM tm.employee AS e
       JOIN tm.restricted_info AS r
       ON (r.empl_code = e.empl_code)
 WHERE r.billing_rate > 20
ORDER BY e.empl_code

empl_code |  full_name   | is_contractor |       email
----------+--------------+---------------+--------------------
ARONSON   | Mary Aronson | FALSE         | mary2@example.com
SMITH     | Ron Smith    | TRUE          | john@example.com
...
```

Tables, such as `restricted_info`, that have an optional one-to-one relationship with a primary table, like `employee`, are called *facets*. Facets allow the handling of sparse data sets and cases where particular sets of information must have additional security constraints.

`/tm:task?assigned_to.restricted_info.billing_rate>20`

The query above returns tasks assigned to employees having a billing rate greater than 20. This is accomplished by transitive application of the previous two rules. First a join is established from `task` to `employee`, and and then down to the `restricted_info` facet.

```
SELECT t.*
  FROM tm.task AS t
       JOIN tm.employee AS e
         ON (e.empl_code = t.assigned_to)
       JOIN tm.restricted_info AS r
         ON (r.empl_code = e.empl_code)
 WHERE r.billing_rate > 20
ORDER BY t.proj_id, t.task_no

proj_id | task_no | assigned_to | status |        name
--------+---------+-------------+--------+--------------------
MEYERS  |       1 | ARONSON     | done   | Purchase Materials
MEYERS  |       2 | SMITH       | review | Strip Wall Paint
...
```

In both of the preceding requests, the cardinality of the resulting join is *singular*: for each row in the `task` table there is at most one corresponding row in the `employee` table and for each employee, there is at most one row in the `restricted_info` table -- consequently, there is at most one `billing_rate` for every task.

`/tm:employee?task.status='done'`

This request returns employees who have at least one assigned task with a `status` of `'done'`. Since there may be more than one task assigned to a given employee, the comparison here is true if any of the task rows match this criteria.

```
SELECT e.empl_code, e.full_name, e.is_contractor, e.email
  FROM tm.employee AS e
 WHERE EXISTS
   (SELECT *
      FROM tm.task AS t
     WHERE t.assigned_to = e.empl_code
       AND htsql_normalize(t.status) = 'done')
ORDER BY e.empl_code
```

```
empl_code |  full_name    | is_contractor |       email
----------+---------------+---------------+------------------
ARONSON   | Mary Aronson  | FALSE         | mary2@example.com
...
```

Unlike the previous examples, the cardinality of this join is *plural*: there may be more than one task assigned to a given employee. When used in this manner, such a specifier is called plural and it checks for existence of at least one matching row.

```
/tm:employee?count(task;status='done')>4
```

This request returns employees who have been assigned more than 4 completed tasks. The **parameter** filter (`;`) limits the related tasks, and the `count()` *aggregate* function converts this correlated sub-query into a *scalar* value for comparison.

```
SELECT e.empl_code, e.full_name, e.is_contractor, e.email
  FROM tm.employee AS e
 WHERE
   (SELECT count(*)
      FROM tm.task AS t
     WHERE t.assigned_to = e.empl_code
       AND htsql_normalize(t.status) = 'done') > 4
ORDER BY e.empl_code


empl_code |  full_name    | is_contractor |       email
----------+---------------+---------------+------------------
ARONSON   | Mary Aronson  | FALSE         | mary2@example.com
...
```

HTSQL's specifier mechanism enables intuitive construction of complicated join criteria. Not only are *singular* (one-to-one or many-to-one) joins allowed, but *plural* (one-to-many) joins are also supported.

## 2.3 Selectors

A sequence of specifiers enclosed in curly brackets, called a **selector**, represents a set of correlated columns from related tables. When a selector immediately follows a table reference, it specifies which columns should be returned. Each specifier in a selector may be followed by a plus or a minus sign to indicate an ascending or descending sort order.

```
/tm:employee{is_contractor+,email,empl_code-}
```

The selector above names three columns, ordered ascending by `is_contractor` and then descending by `empl_code`. A third column, `email`, is returned, but is not used for sorting. HTSQL makes no provision to order results by columns that are not returned, or to list columns in an order that differs by their appearance in the sort criteria.

```
SELECT is_contractor, email, empl_code
  FROM tm.employee
ORDER BY is_contractor asc, empl_code desc;

is_contractor |       email       | empl_code
--------------+-------------------+----------
FALSE         | mary2@example.com | ARONSON
...
```

```
/tm:task{project.name,task_no,employee.full_name}
```

The above query uses multi-part specifiers to return, for each task, the corresponding project's name and the assigned employee's full name. Implicit joins are created in order to fetch the corresponding information. In the corresponding SQL, an outer join is used so that tasks lacking an assigned employee are still returned.

```
SELECT p.name AS "project.name", t.task_no,
       e.full_name AS "employee.full_name"
  FROM tm.task AS t
       JOIN tm.project AS p
         ON (t.proj_id = p.proj_id)
       LEFT OUTER JOIN tm.employee AS e
         ON (e.empl_code = t.assigned_to)
ORDER BY t.proj_id, t.task_no;

  project.name    | task_no | employee.full_name
------------------+---------+--------------------
Meyer's Residence |       1 | Mary Aronson
Meyer's Residence |       2 | Ron Smith
Meyer's Residence |       3 |
South Square Mall |       1 | Adam O'Brien
...
```

`/tm:task{proj_id,task_no,employee{full_name,is_contractor}}`

In this example, both the `full_name` and `is_contractor` columns from the `employee` table are requested. Using a nested selector in this manner avoids duplicating table references. The selector `employee{full_name,is_contractor}` is short-hand for `employee.full_name,employee.is_contractor`.

```
SELECT t.proj_id, t.task_no,
       e.full_name AS "employee.full_name",
       e.is_contractor AS "employee.is_contractor"
  FROM tm.task AS t
       LEFT OUTER JOIN tm.employee AS e
         ON (e.empl_code = t.assigned_to)
ORDER BY t.proj_id, t.task_no;

proj_id | task_no | employee.full_name | employee.is_contractor
--------+---------+--------------------+-----------------------
MEYERS  |       1 | Mary Aronson       | FALSE
MEYERS  |       2 | Ron Smith          | TRUE
...
```

`/tm:task{*,employee.*}`

In a manner like SQL, all columns can be requested in a selector using the asterix (*). This is also the default when a table is provided without an explicit selector.

```
SELECT t.*,
       e.empl_code AS "employee.empl_code",
       e.full_name AS "employee.full_name",
       e.is_contractor AS "employee.is_contractor",
       e.email AS "employee.email"
  FROM tm.task AS t
       LEFT OUTER JOIN tm.employee AS e
         ON (e.empl_code = t.assigned_to)
ORDER BY t.proj_id, t.task_no;

proj_id | task_no | . | employee {empl_code}
--------+---------+ . +---------------------
MEYERS  |    1    | . | ARONSON
MEYERS  |    2    | . | SMITH
MEYERS  |    3    | . | (NULL)
...
```

`/tm:project{proj_id,task.task_no}`

If a plural specifier (which causes a one-to-many join) is used within a selector, an ARRAY is returned. This example returns a row for each project, and for each row, an array of associated task numbers.

```
SELECT p.proj_id,
       ARRAY(SELECT t.task_no
               FROM tm.task t
              WHERE t.proj_id = p.proj_id
```

```
                  ORDER BY t.proj_id, t.task_no)
                    AS "task{task_no}"
  FROM tm.project AS p
ORDER BY p.proj_id;

proj_id  | task{task_no}
---------+-----------------------------
MEYERS   | {1,2,3}
SSMall   | {1}
THOM-LLP | {}
...
```

`/tm:employee{full_name,isnull(email),count(task)}`

Functions may be used in the context of a selector. In this example, `isnull()` is a *scalar* function while `count()` is an *aggregate* function. Aggregate functions may be applied to plural specifiers, such as `task`.

```
SELECT e.full_name,
       (e.email is null) AS "isnull(email)",
       count(t.*) AS "count(task)"
  FROM tm.employee AS e
       LEFT OUTER JOIN tm.task AS t
       ON (t.assigned_to = e.empl_code)
GROUP BY e.empl_code, e.full_name, (e.email is null)
ORDER BY e.empl_code;

full_name       | isnull(email) | count(task)
----------------+---------------+-------------
Adam O'Brien    | FALSE         | 23
...
Alfred Smith    | TRUE          | 0
...
```

In general, most SQL operations such as IS NULL are available in HTSQL. However, we use a simplified function notation that is more familiar to occasional programmers; our syntax includes sequential argument passing an optional keyword parameters.

`/tm:task{assigned_to,@status|count(*)}`

Data pivoting can be requested using the asterix (@) immediately preceding a column, such as `status`. In this case, distinct values in that column become the headers in the result set, and remaining columns become the values within respective buckets.

```
:
SELECT t.assigned_to,
   SUM(CASE WHEN t.status = 'done' THEN 1
            ELSE 0 END) AS "done",
            ...
   SUM(CASE WHEN t.status = 'review' THEN 1
            ELSE 0 END) AS "review",
  FROM tm.task AS t
GROUP BY t.assigned_to
ORDER BY t.assigned_to;

assigned_to | done | . | review
------------+------+ . +--------
ADAM        |    0 | . |  1
ARONSON     |    4 | . |  2
...
(NULL)      |    0 | . |  0
```

This translation need not be done in SQL as shown above, since a pivot operation is strictly a visualization mechanism. The equivalent SQL above reflects an idiom for this sort of item. If more than one summary calculation or pivot is requested, there may be 2 or more header rows.

`/tm:restricted_info{empl_code,birth_date.year,tax_id[-4:]}`

HTSQL provides support for operations on structured column values, such as dates. Standard slicing syntax is also supported to extract substrings. The slice example above returns the last four digits of the employee's tax id.

```
SELECT empl_code,
       EXTRACT(YEAR from birth_date) AS "birth_date.year",
       SUBSTRING(tax_id FROM (1+(LENGTH(tax_id)-4))
                        FOR 4) AS "tax_id[-4:]"
  FROM tm.restricted_info
ORDER BY empl_code

empl_code | birth_date.year | tax_id[-4:]
----------+-----------------+------------
ADAM      | 1961            | 1492
...
```

In the more verbose functional notation, the slice example above, `tax_id[-4:]`, could be written as `substring(tax_id,from=(1+(length(tax_id)-4)),for=4)` directly emulating the corresponding SQL92 query.

HTSQL's selector syntax allows fault-free construction of straight forward control over what value expressions are returned in the result set. Invoking SQL functions and doing string manipulation is well supported.

## 2.4 Identifiers and Locators

HTSQL provides explicit support for selecting particular rows of a given table using primary key columns. When using this syntax, each value associated with a primary key column is called a **label**, and a dotted sequence of labels is called an **identifier**. Labels are compared via *usual equality* as described above. HTSQL uses square brackets to enclose a comma-separated list of identifiers that locate rows within the database. A sequence of identifiers enclosed in square brackets is called a **locator**.

`/tm:project[meyers]`

This request returns the `meyers` project using the locator syntax. Besides being shorter, this syntax asserts that exactly one row is returned. Except for the assertion, this request is the same as `/tm:project?proj_id='meyers'`. Comparison via the *usual equality* operator guards against labels that differ only by capitalization.

```
SELECT * FROM tm.project
 WHERE htsql_normalize(proj_id) = 'meyers'
ORDER BY proj_id

proj_id  | name               | description
---------+--------------------+----------------------------
MEYERS   | Meyers' Residence  | insulation and winterizing
```

If the corresponding SQL query returns an empty result set, a "404 Not Found" error is returned. It is also possible that more than one row are matched. In this case, a "300 Multiple Choices" is returned rather than a result set with multiple rows.

`/tm:task[meyers.1]`

The full-stop (`.`) is used used to separate labels in cases where the primary key includes more than one column. This example is similar to `/tm:task?proj_id='meyers'&task_no=1`

```
SELECT * FROM tm.task
 WHERE htsql_normalize(proj_id) = 'meyers'
   AND htsql_normalize(task_no) = '1'
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+---------------------+---------+-------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
```

`/tm:task[meyers.*]`

Components in an identifier may be unknown, replaced instead with the wildcard (`*`). In these cases, the cardinality of the result set can be one or more. Although an empty result set is still a "404 Not Found".

```
SELECT * FROM tm.task
 WHERE htsql_normalize(proj_id) = 'meyers'
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+-----------------------+---------+--------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
MEYERS  |       2 | SMITH       | review  | Strip Wall Paint
MEYERS  |       3 | (NULL)      | planned | Remove Refuse
...
```

`/tm:task[meyers.1,meyers.2,ssmall.0001]`

More than one identifier may be provided within square brackets if a specific enumeration of rows is known. In this multiple-identifier case, exactly one row is expected for each identifier provided. If any one of the identifiers is not matched, a "404 Not Found" is returned with the offending identifier.

```
SELECT * FROM tm.task
 WHERE (htsql_normalize(proj_id) = 'meyers' AND
        htsql_normalize(task_no) in ('1','2'))
    OR (htsql_normalize(proj_id) = 'ssmall' AND
        htsql_normalize(task_no) = '0001')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+-----------------------+---------+--------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
MEYERS  |       2 | SMITH       | review  | Strip Wall Paint
SSMall  |       1 | ADAM        | review  | Install Slider Door
```

`/tm:task{id(),name}`

A row identifier can returned using a built-in function, `id()` to return the primary key columns of the table concatenated with a full stop (`.`).

```
SELECT (t.proj_id || '.' || t.task_no) AS "id()",
       t.name AS "name"
  FROM tm.task AS t
ORDER BY t.proj_id, t.task_no;;

id()     | name
---------+------------------
MEYERS.1 | Purchase Materials
MEYERS.2 | Strip Wall Paint
...
```

`/tm:employee{empl_code,task}`

If a plural specifier naming a table is mentioned in a selector, then an ARRAY of associated identifiers is generated. In this example, a row for each employee is returned, and for each row, an array of associated task identifiers is listed.

```
SELECT e.empl_code,
       ARRAY(SELECT (t.proj_id || '.' || t.task_no)
               FROM tm.task t
              WHERE t.assigned_to = e.empl_code
             ORDER BY t.proj_id, t.task_no) AS "task"
  FROM tm.employee AS e
ORDER BY e.empl_code

empl_code | task
----------+----------------------------
ADAM      | {ssmall.1, ... }
```

```
ARONSON   | {meyers.1, meyers.5, ... }
...
SMITH-A   | {}
...
```

Identifiers provide a handy notation for resource location: they are concise and unique. Identifiers can be explicitly requested in the selector using `id()`; furthermore, they can be used within a locator to return exactly one row.

## 2.5 Commands

In HTSQL, a command denoted by parentheses may be used in the last (right most) path-segment of the URI to invoke a specific database operation or extension function. In the previous examples, the command `select()` was implicit.

`/tm:employee/select(offset=10,limit=2)`

The explicit `select()` command has two optional keyword/value arguments which can be used to return a sliding window over a result set. For this particular example, the result set starts at the 11th row and returns `2` rows.

```
SELECT * FROM tm.employee
ORDER BY empl_code
OFFSET 10 LIMIT 2              /* PostgreSQL Syntax */

empl_code | full_name     | is_contractor | email
----------+---------------+---------------+-------------
HUCK      | Ed Huckington | TRUE          | huck@example.com
JACK      | Jack Winters  | FALSE         | jack@example.com
```

Unfortunately, ISO SQL does not have a provision for offset/limit as implemented by PostgreSQL and other databases. However, the comments section in the SQL2003 hints that this feature is forthcoming. Since this feature is extremely useful and has implementations in just about every database, it is included in the HTSQL specification.

`/tm:project/insert()?proj_id:='ALBE'&name:='Alberca'`

Other commands for insert, update, and delete follow a similar syntax, using colon-equal (`:=`) to indicate an assignment. The result from an insert statement is a "201 Created" with a content body containing the URI(s) of the objects inserted.

```
INSERT INTO tm.project (proj_id, name)
  VALUES ('ALBE','Alberca');

201 Created
/tm:project[albe@1]
```

For RESTful behavior, the POST method with query arguments in a standard entity body format will also work; the GET method is permitted so that standard features can be used directly in a web browser's location bar.

`/tm:project/update()?proj_id='ALBE'&description:='Leaky Pool'`

This request updates the `description` of the 'ALBE' project. Note that only columns using the assignment operator (`:=`) are changed; the remaining column references are used to limit the rows affected.

```
UPDATE tm.project
   SET description = 'Leaky Pool'
 WHERE htsql_normalize(proj_id) = 'albe'

201 Created
/tm:project[albe@2]
```

Like the insert() operation, update() returns a "201 Created" when successful, listing the URIs of the affected resources. Even though the affected rows are actually modified, they constitute a new resource: implementations could permit access to previous versions of the modified row.

`/tm:project/delete(expect=3)?description~'pool'`

If a data modification request would change more than one row (or less than one row), an `expect` keyword argument is needed. In this example, the request expects exactly 3 rows to be deleted.

```
DELETE FROM tm.project
 WHERE LOWER(description) LIKE '%pool%';

204 No Content
```

Deleting a row does not return content. If any more or fewer rows would be affected, a "417 Expectation Failed" is returned and the data modification request is aborted.

`/tm:project[able]/merge()?name:='SouthWest%20Alberca'`

The `merge` operation provides a succinct syntax for adding or updating a resource based on its identifier. If the row already exists, it is updated, otherwise it is created. The result of this command is a "201 Created" when the corresponding insert or update succeeds.

```
MERGE INTO tm.project
USING tm.project ON htsql_normalize(proj_id) = 'albe'
WHEN MATCHED THEN UPDATE
  SET name = 'SouthWest Alberca'
WHEN NOT MATCHED THEN INSERT
  (proj_id, name) VALUES
  ('albe', 'SouthWest Alberca');

201 Created
/tm:project[albe@3]
```

The MERGE syntax above comes from SQL2003. However, the concept is so useful in a web setting that the corresponding SQL1992 transaction can be simulated if this feature is not natively supported. While HTSQL does not require spaces to be encoded as `%20`, many user-agents do.

`/tm:task{task_no,status}/parse()?employee='adam'`

The `parse()` command returns a parse tree representing the request. It is useful for clients which provide a graphical query builder. This command is also helpful when debugging because it shows how the HTSQL server is interpreting a given URI.

```
<parse xmlns="http://htsql.org/2006/">
  <context schema="tm" table="task">
    <selector>
      <specifier column="task_no" />
      <specifier column="status" />
    </selector>
  </context>
  <operation name="parse" />
  <query>
    <comparison type="equality">
      <specifier table="employee">
      <literal value="adam" />
    </comparison>
  </query>
</parse>
```

HTSQL provides the standard SQL commands, `insert()`, `update()` and `delete()`. The `merge()` function is particularly useful in a web environment where the status of an object is being "reset" regardless of

its previous state.

## 2.6 Path Contexts

In HTSQL, more than one table can be directly mentioned in a request, provided that the tables are related (perhaps transitively) by a foreign-key relationship. Each table, together with associated selectors and parameters is called a **context**.

`/tm:employee/task`

This request returns tasks, grouping by assigned employee. This particular result is possible since there is an `assigned_to` foreign key from `task` to `employee`. Employees which do not have corresponding tasks, or tasks which are not assigned to an employee are not returned.

```
SELECT e.empl_code     AS "employee{empl_code}",
       e.full_name     AS "employee{full_name}",
       e.is_contractor AS "employee{is_contractor}",
       e.email         AS "employee{email}",
       t.proj_id       AS "task{proj_id}",
       t.task_no       AS "task{task_no}",
       t.assigned_to   AS "task{assigned_to}",
       t.status        AS "task{status}",
       t.name          AS "task{name}"
  FROM tm.task AS t
       JOIN tm.employee AS e
         ON (t.assigned_to = e.empl_code)
ORDER BY e.empl_code, t.proj_id, t.task_no;


employee{empl_code} | . | task{proj_id} | task{task_no} | .
--------------------+ . +---------------+---------------+
ADAM                | . | MEYERS        |       7       | .
ADAM                | . | SSMall        |       2       | .
ARONSON             | . | MEYERS        |       1       | .
...
```

`/tm:employee[aronson]/task`

The example above returns tasks that have been assigned to the employee `aronson`; duplicating information about this employee for each row. While this query is similar to `/tm:task?assigned_to='aronson'`, it asserts that exactly one employee with identifier `aronson` is matched.

```
SELECT <all columns from employee and task>
  FROM tm.task AS t
       JOIN tm.employee AS e
         ON (t.assigned_to = e.empl_code)
 WHERE htsql_normalize(e.empl_code) = 'aronson'
ORDER BY e.empl_code, t.proj_id, t.task_no

employee{empl_code} | . | task{proj_id} | task{task_no} | .
--------------------+ . +---------------+---------------+
ARONSON             | . | MEYERS        |       1       | .
...
```

`/tm:project[meyers]/task[1]`

Identifiers may be shortened when the context can be used to fully-qualify them. In this example, the identifier for the requested task is `meyers.1`.

```
SELECT <all columns from project and task>
  FROM tm.project AS p
       JOIN tm.task AS t
         ON (t.proj_id = p.proj_id)
 WHERE htsql_normalize(p.proj_id) = 'meyers'
   AND htsql_normalize(t.task_no) = '1'
ORDER BY p.proj_id, t.proj_id, t.task_no
```

```
project{proj_id} | . | task{proj_id} | task{task_no} | .
-----------------+ . +---------------+---------------+ .
MEYERS           | . | MEYERS        |       1       | .
```

`/tm:project[meyers]/employee[aronson]/task/insert()?task_no=4&name='Clean Up'`

When more than one table is provided for an `insert()` statement, un-ambiguous links to the rows identified in the context are assumed. In this case, the new task will be part of the `meyers` project and will be assigned to `aronson`.

```
INSERT INTO tm.task (proj_id, assigned_to, task_no, name)
VALUES ((SELECT proj_id FROM tm.project
           WHERE htsql_normalize(proj_id) = 'meyers'),
         (SELECT empl_code FROM tm.employee
           WHERE htsql_normalize(empl_code) = 'aronson'),
           '4', 'Clean Up')
```

In HTSQL, whitespace between tokens (but not within single or double quotes) is not significant. To enhance readability, we broke the request above onto two lines.

`/tm:employee{full_name}/task{project.name,task_no}?status='done'`

This request returns tasks that are `'done'` together with detail regarding the assigned employee's full name, the project's name, and the task number. That tasks that are not assigned are not returned with this request.

```
SELECT e.full_name AS "employee{full_name}",
       p.name      AS "task{project.name}",
       t.task_no   AS "task{task_no}"
  FROM tm.task AS t
       JOIN tm.employee AS e
         ON (t.assigned_to = e.empl_code)
       JOIN tm.project AS p
         ON (p.proj_id = t.proj_id)
 WHERE htsql_normalize(t.status) = 'done'
ORDER BY e.empl_code, t.proj_id, t.task_no

employee{full_name} | task{project.name}  | task{task_no}
--------------------+---------------------+--------------
Mary Aronson        | Meyers' Residence   |       1
...
```

`/tm:employee//task`

To suppress the automatic joins, a double-slash (`//`) may be used; the result is a cross product. In this example, the usual join using the foreign key `assigned_to` is explicitly suppressed to return all permutations of employee and task.

```
SELECT <all columns from employee and task>
  FROM tm.task AS t,
       tm.employee AS e
ORDER BY e.empl_code, t.proj_id, t.task_no;

employee{empl_code} | . | task{proj_id} | task{task_no} | .
--------------------+ . +---------------+---------------+
ADAM                | . | MEYERS        |       1       | .
ADAM                | . | MEYERS        |       2       | .
..
ARONSON             | . | MEYERS        |       1       | .
...
```

`/tm:$a:=project//$b=:project?a.proj_id[0]==b.proj_id[0]`

When using more than one copy of a table, a table alias created by `$var:=` is required. Once aliased, subsequent usage of the table is then referenced with `var`. This contrived example returns pairs of

projects which have the same first letter in their project identifier.

```
SELECT a.proj_id     AS "a{proj_id}",
       a.name        AS "a{name}",
       a.description AS "a{description}",
       b.proj_id     AS "a{proj_id}",
       b.name        AS "a{name}",
       b.description AS "a{description}"
  FROM tm.project AS a,
       tm.project AS b
  WHERE SUBSTRING(a.proj_id FROM 1 FOR 1) =
        SUBSTRING(b.proj_id FROM 1 FOR 1)
 ORDER BY a.proj_id, b.proj_id;

a{proj_id} | . | b{proj_id} | .
-----------+ . +------------+ .
MEYERS     | . | MEYERS     | .
MEYERS     | . | MIVDA      | .
MIVDA      | . | MEYERS     | .
...
```

`/tm:(+)employee/task`

In HTSQL, an *outer join* is indicated with a plus (`(+)`) surrounded by parenthesis preceding the table name. The above request, for example, returns tasks even if they have not been assigned an employee. In these cases, the employee columns are NULL.

```
SELECT <all columns from employee and task>
  FROM tm.task AS t
       LEFT OUTER JOIN tm.employee AS e
         ON (t.assigned_to = e.empl_code)
ORDER BY e.empl_code, t.proj_id, t.task_no;


employee{empl_code} | . | task{proj_id} | task{task_no} | .
--------------------+ . +---------------+---------------+
ADAM                | . | SSMall        |       2       | .
ARONSON             | . | MEYERS        |       1       | .
...
(NULL)              | . | MEYERS        |       3       | .
...
```

`/tm:employee{full_name};is_contractor/(+)task{project.name,task_no}`

This request returns each contractor in the `employee` table and their associated tasks, if any. Filters on a particular context are indicated with a semi-colon (`;`), the context's **parameters**. In this case, the `is_contractor` filter applies to the `employee` table.

```
SELECT e.full_name AS "employee{full_name}",
       p.name      AS "task{project.name}",
       t.task_no   AS "task{task_no}"
  FROM tm.employee e
       LEFT OUTER JOIN tm.task AS t
         ON (t.assigned_to = e.empl_code)
       LEFT OUTER JOIN tm.project AS p
         ON (p.proj_id = t.proj_id)
 WHERE e.is_contractor IS TRUE
ORDER BY e.empl_code, t.proj_id, t.task_no

employee{full_name} | task{project.name} | task{task_no}
--------------------+--------------------+---------------
Ron Smith           | Meyer's Residence  |             2
Alfred Smith        | (NULL)             |  (NULL)
...
```

This set of examples illustrates the large difference in readability between HTSQL URIs and the corresponding SQL. While SQL may be more expressive, for common needs, HTSQL is more succinct and understandable.

The "path" based metaphor of URIs, together with the parameter syntax using the semi-colon, allows relatively common joins to be easily specified.

## 2.7 Transactions and Locking

HTSQL supports database transactions and record locking. Besides explicit row locking, bulk updates and *optimistic locking* are possible. Optimistic locking is accomplished by returning a row version; and then updating a row only when this version matches. Pessimistic locking requires either a *session* mechanism as provided by the application or its transaction is limited to the scope of the current HTTP connection (using "Connection: Keep-Alive" for HTTP/1.0 clients).

`/tm:task{id(),tag(),idtag(),name}`

The `tag()` function returns, for each row, a string value that can be used to test when an update has occurred. In a subsequent request, if the value of `tag()` changes, then the row has been updated. The return value of `tag()` is usually a row version number or a timestamp that is changed on each update. The `idtag()` function returns a row's identifier and tag as a single value, separated with an at-sign (@).

```
SELECT (t.proj_id || '.' || t.task_no) AS "id()",
       <tag> AS "tag()",
       (t.proj_id || '.' || t.task_no ||
        '@' || <tag>) AS "idtag()",
       t.name AS "name"
  FROM tm.task AS t
ORDER BY t.proj_id, t.task_no;;

id()     | tag() | idtag()    | name
---------+-------+------------+--------------------
MEYERS.1 | 3     | MEYERS.1@3 | Purchase Materials
MEYERS.2 | 7     | MEYERS.2@7 | Strip Wall Paint
...
```

Since the notion of a *tag* is not specified by SQL92, its implementation may be dependent upon the specific database, schema, and configuration used. The only constraint placed upon `tag()` is that all future versions of updated rows must return a different value. If the HTSQL processor is unable to fulfill these requirements, it must respond with "501 Not Implemented".

`/tm:task[meyers.2@7]`

This request attempts to return version 7 of the task identified by `meyers.2`. As above, if the task does not exist, then a "404 Not Found" is returned. If the `tag()` of the requested row has since changed, a "301 Moved Permanently" is issued, giving the location of the current version.

```
301 Moved Permanently
/tm:task[meyers.2@8]
```

`/tm:project[meyers@2]/update()?name:='Changed%20Name'`

If an update or delete operation is applied to an object that has the incorrect version, then the update request would fail with a "409 Conflict". This message will include a reference to the most recent version, as well as a body with the change history (if available) since the version requested.

```
409 Conflict
/tm:project[albe@3]
```

`/interaction()`

The `interaction()` operation takes a POST body of type "text/htsql", each line being an HTSQL request. The requests are executed together as part of a single transaction; if any request fails with a 4xx or 5xx result code, then the entire transaction is rolled back. Following is an example request body which inserts a project and three tasks as a single transaction.

```
/tm:project[waterfall]/insert()?name:='Waterfall Example'
```

```
/tm:project[waterfall]/task[1]/insert()?name:='Feasibility'
/tm:task[waterfall.2]/insert()?name:='Analysis Work'
/tm:task[$(project).3]/insert()?name:='Design Phase'
```

Within text/htsql document, variables like `$(table)` resolves to the identifier for the most recent insert or merge statement of the given table. Hence, `$(project)` in this example is `waterfall`.

`/tm:project[waterfall]/task/insert()`

Bulk inserts can also be performed using a CSV file in the POST body. The first line of the file gives the columns, and remaining lines correspond to each row being insert. In this case, the project, `waterfall` is obtained from the parent context and doesn't need to be repeated in the CSV file.

```
task_no,status,name
4,planned,Implementation
5,planned,Testing
6,planned,Deployment
```

A similar file can be provided for `update()` or `merge()`, however, these actions require at least one `id()` or `idtag()` column to uniquely locate the row being affected.

`/begin()`

This starts a transaction. An active transaction can be finished with either `commit()` or `rollback()`. The transaction is automatically abandoned if the current HTTP connection or the active session ends.

```
BEGIN TRANSACTION

204 No Content
```

`/tm:project/select(lock='update nowait')`

This select parameter places a row lock on the returned rows. If a lock cannot be obtained, it immediately returns a "408 Request Timeout" error. This command must be preceded with a `begin()`.

```
SELECT *
FROM tm.project
FOR UPDATE NO WAIT
ORDER BY proj_id

408 Request Timeout
```

Locking for `update` is assumed when either `lock="wait"` or `lock="nowait"`. An explicit lock is dropped after an explicit `commit()` or `rollback()`.

HTSQL provides for standard optimistic locking, bulk transaction operations, as well as standard transactions with explicit locking.

## 2.8 Resources and Formats

Normally, when a web browser makes a request, it provides a list of result formats it will accept. HTSQL provides mechanisms to override this content negotiation and explicitly request a particular result format. When a table (with optional locator and selector) is followed by an extension, the output format is determined via typical association to well-known MIME types. If the extension is not known, it is assumed to be `text/ext` where `ext` is the extension provided in the URI.

Further, a path segment starting with the resource indicator (~) means that the request is not to be processed by HTSQL, but instead should be delegated to an underlying application.

`/tm:project[meyers]{id(),name}.csv`

The result format can be specified as an argument to `select` or other operations. In this case, the

standard comma-separated variable format is chosen. The output format can also be provided as an argument to the `select(format='text/csv')` command.

```
id(),name
MEYERS,Meyers' Residence
```

In the CSV format, results exactly match the SQL query (and are appropriately de-normalized). The double-quote is a delimiter used if a column value contains a comma; two adjacent double-quotes are used to escape a double-quote occurring in the query results.

### `/tm:project{name}/employee{}/task{task_no,status}.xml`

In the eXtensible Markup Language ("XML") format, result elements are named after the table they represent. For each path context, a hierarchical relation is established, duplicating intermediate nodes (such as "aronson" above) as required. If a given table or column is not a valid element or attribute name, there are work-arounds using the `htsql` namespace.

```
<?xml version="1.0" encoding="utf-8"?>
<htsql:result xmlns:htsql="http://htsql.org/2006/"
              htsql:schema="tm">
  <project htsql:id="meyers@3"
           name="Meyer's Residence">
    <employee htsql:id="aronson@11">
      <task htsql:id="meyers.1@1"
            task_no="      01" status="done" />
      <task htsql:id="meyers.23@1"
            task_no="      23" status="review" />
      <!-- ... -->
    </employee>
    <employee htsql:id="adam@5">
      <task htsql:id="meyers.11@4"
            task_no="      11" status="done" />
      <!-- ... -->
    </employee>
    <!-- ... -->
  </project>
  <project htsql:id="ssmall@9"
           name="South Square Mall">
    <employee htsql:id="aronson@11">
      <task htsql:id="ssmall.13@1"
            task_no="      01" status="started" />
      <!-- ... -->
    </employee>
    <!-- ... -->
  </project>
  <!-- ... -->
</htsql:result>
```

### `/tm:project{name}/employee{}/task{task_no,status}.yaml`

The YAML format for HTSQL consists of two sections: a context and an assembly. For each table mentioned in the request, the context lists each row returned for that table. The assembly then represents the drill-down relationships between these rows. For large results, the content returned by YAML is much smaller than the equivalent XML, since each row occurs only once in the context rather than being duplicated for each occurrence in the hierarchical assembly.

```
%YAML 1.1
---
schema: tm
context:
  project:
  - !project &1
    =: meyers@3
    name: Meyer's Residence
  - !project &2
    =: ssmall@9
```

```
      name: South Square Mall
    #...
    employee:
    - !employee &3
      =: aronson@1
    - !employee &4
      =: adam@5
    #...
    task:
    - !task &5
      =: meyers.1@1
      task_no: 01
      status: done
      project: *1
      employee: *3
    - !task &6
      =: meyers.23@1
      task_no: 23
      status: review
      project: *1
      employee: *3
      # ...
    - !task &7
      =: meyers.11@4
      task_no: 23
      status: done
      project: *1
      employee: *4
      # ...
    - !task &8
      =: ssmall.13@1
      task_no: 13
      status: started.
      project: *2
      employee: *3
    #...
assembly: [*1: [*3: [*5, *6],
                *4: [*7]],
           *2: [*3: [*8]]]
...
```

`/tm:project{name}/task{task_no,status}/~some+resource`

Whenever a path segment starts with a tilde ~, it indicates a user resource. Once found, the entire URI is not processed by HTSQL, but is instead passed on to the application. The application can then choose to return a resource specific to the path, or what ever it wishes. Note that relative paths work as expected, even for "static" resources. For example, a relative `./select()` reference in the resource above would produce a list of tasks.

```
(application defined resource for "some+resource")
```

`/tm:project{name}/task{task_no,status}.html~bing`

Following a request for XML or HTML output, a style-sheet can be requested as shown above. If the style-sheet lacks an extension (such as "xsl"), then "css" is assumed.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
<HTML>
  <HEAD>
    <LINK type="text/css" rel="stylesheet"
     href="/tm:project{name}/task{task_no,status}/~bing.css">
  </HEAD>
  <BODY>
    <TABLE>
      <COLGROUP ID="project">
        <COL CLASS="project project_name" ID="project:name">
```

```
          </COLGROUP>
          <COLGROUP ID="task">
            <COL CLASS="task task_task_no" ID="task:task_no">
            <COL CLASS="task task_status" ID="task:status">
          </COLGROUP>
          <THEAD>
            <TR>
              <TH COLSPAN="1" SCOPE="colgroup"
                  CLASS="project">Project</TH>
              <TH COLSPAN="2" SCOPE="colgroup"
                  CLASS="task">Task</TH>
            </TR>
            <TR>
              <TH>Name</TH>
              <TH>Task No</TH>
              <TH>Status</TH>
            </TR>
          </THEAD>
          <TBODY>
            <TR ID="task:meyers.1">
              <TD ROWSPAN="2" SCOPE="rowgroup" VALIGN="top">
                Meyers' Residence</TD>
              <TD>      01</TD>
              <TD>done</TD>
            </TR>
            <TR ID="task:meyers.2">
              <TD>      02</TD>
              <TD>done</TD>
            </TR>
            <!-- ... -->
            <TR ID="task:ssmall.1">
              <TD>South Square Mall</TD>
              <TD>      01</TD>
              <TD>review</TD>
            </TR>
            <!-- ... -->
          </TBODY>
        </TABLE>
      </BODY>
    </HTML>
```

The varied output formats and customizable resources, together with stylesheet linkage, create a flexible mechanism for constructing user interfaces.

## 2.9 HTML FORM Compatibility

Standard HTML form encoding presents several challenges to HTSQL. While it is possible to provide almost any name (such as one corresponding to a specifier) for an INPUT control, user-provided values are always percent-encoded instead of being single-quoted as HTSQL requires. Further, HTML only supports a flat expression structure, conflating the ampersand's meaning of conjunction with simply implying that additional form elements are provided. HTSQL defines a **literal value** syntax and **variable** substitution to provide direct support for HTML forms without requiring client-side processing.

`/tm:project?name@~south+square`

Any operator may be preceded by the at-sign (`@`) to indicate that the right-hand side is a percent-encoded value. A literal value provided in this syntax terminates at the next ampersand character or at the end of the request string. This request is equivalent to `/tm:project?name~'south square'`.

```
    SELECT * FROM tm.project
     WHERE LOWER(name) LIKE '%south square%'
    ORDER BY proj_id

    proj_id  | name               | description
    ---------+--------------------+-----------------------------
```

```
SSMall   | South Square Mall  | two new store fronts
...
```

This literal value syntax permits values to be included according to standard web usage without requiring cumbersome single quotes. In this syntax, the plus sign is used to encode the space (`%20`) and reserved characters must be percent-encoded.

`/tm:task?status@=done&status@=review`

When using the literal value syntax, multiple items joined by an ampersand are treated as a simple alternation. This is an ugly exception to the normal grammar, however it is needed to meet the expectations of standard HTML form usage. The request above is equivalent to `/tm:task?status='done','review'`.

```
SELECT * FROM tm.task
 WHERE htsql_normalize(status) in ('done','review')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+---------------------+---------+------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
...
```

Since the comma is a reserved character, this syntax also allows expressions such as `/tm:task?status@=done,review` with an identical interpretation.

`/tm:task?status='done',other&$other@:=review`

In HTSQL, variables are declared with a dollar-sign. This example uses variable substitution to return tasks that are either done or ready for review. The variable `other` is first referenced as part of the filter on the `status` column. Following the ampersand, this variable is defined with the value "review", using the assignment operator.

```
SELECT * FROM tm.task
 WHERE htsql_normalize(status) in ('done','review')
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+---------------------+---------+------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
...
```

This query could be submitted via a HTML form with a hidden input `status` having value `'done',other`, and another input named `$other@:` with user-provided value `review`. After variable resolution, this request is equivalent to `/tm:task?status='done','review'`.

`POST /tm:task?assigned_to=='ARONSON'`
with URL-encoded post body `status@=review&status@=done`

When POST is used with a "multipart/form-data" or "application/x-www-form-urlencoded" MIME types, key/value pairs are integrated as if they used an ampersand.

```
SELECT * FROM tm.task
 WHERE htsql_normalize(status) in ('done','review')
   AND assigned_to = 'ARONSON'
ORDER BY proj_id, task_no

proj_id | task_no | assigned_to | status  | name
--------+---------------------+---------+------------------
MEYERS  |       1 | ARONSON     | done    | Purchase Materials
...
```

The request above is equivalent to
`/tm:task?assigned_to=='ARONSON'&status='done','review'`.

By merging POST arguments with a URI using literal-value syntax and with clever use of variable substitution, it should be possible to send just about any HTSQL query using standard HTML form submission.

# 3. Design

TODO: this section needs to be complete re-written.

## 3.1 Concepts

HTSQL relies upon meta-data regarding an SQL schema in order to provide a mapping of URLs onto SQL statements. In particular, the *INFORMATION_SCHEMA* as defined by SQL-92 and implemented by most modern databases is mostly sufficient for this purpose. Lacking an compliant information schema, a particular implementation could, of course, provide a replacement mechanism.

A reasonably simple HTTP URI access mechanism requires a restrictive data access and representation model. This chapter outlines the conceptual model which HTSQL supports. While this model can be implemented with an SQL database, not every SQL construct will have a corollary in this model.

## 3.2 Workspace

A *workspace* refers to a coarse level of information storage and authorization. In a relational database, a workspace typically corresponds to a database instance or schema. While it is possible for two workspaces to have the same structure, all workspaces in a system need not be structured identically. Besides identifying the data repository, the workspace can be used for access control, limiting particular users to given workspaces.

## 3.3 Class

A *class* refers to a collection of data items (or *entities*) having the same structure. An entity is a single instance of a class, such as a particular person or a given object. In a relational database, a class is often associated with a *table* and each entity in the class is stored as a *row*. Each class belongs to exactly one workspace and has a name that is unique within its workspace. By convention, the name of a class is singular, e.g., 'person' not 'persons' or 'people'.

## 3.4 Field

A *field* refers to descriptive detail of a class, such as a person's name or gender. Each entity in a class may have at most one *value* for each of its fields. If a value is not provided, the field is said to be *null*. In a relational database, a field is usually associated with a *column* of the class's table and a particular entity's value is often called a *cell*. There are three sorts of fields, scalars, references, and special.

A *scalar* field is value represented as a sequence of characters.

The *text* field type represents a Unicode string, each text field has a specified length and is typically shown on the screen with a text box.

The *memo* field type represents a multi-line Unicode string, without a set length. It is usually shown on the screen as a text area.

The *date* field type represents a date, which is typically shown in [ISO8601], YYYY-MM-DD format. A date field is usually displayed as a single text box, but this can vary depending upon the application.

The *integer* field type can hold 32 bit integer values.

The *sort* is a numeric field that is used primarly for overriding the sort in a list.

The *boolean* field type can hold a true or false value and is usually rendered with a checkbox.

The *code* field type is used for unique keys. In HTSQL, codes are alpha-numeric, admitting the dash. Codes are case-insensitive, but case-preserving. Codes also ignore leading 0's.

A *reference* field is used to record links from one entity to another. Implementation details of this mechanism are not specified, but HTSQL uri's use reference fields extensively to associate entities.

There are a few *special* field types supported by HTSQL which are neither scalar values nor references.

The *file* field type is used to store files which have been attached to a given entity.

The *choice* field type represents a list of values, at most one of which may be selected. This field is usually represented on the screen using a single-select drop-down list. The specific value for each choice is a *code*. Choice items are associated with a title, note, and sort. When shown in a list box, the items are sorted according to sort, and the title is shown in the list box. The actual code value is what is stored and shown in tabular output.

## 3.5 Identity

Besides the system provided handle, HTSQL requires that each entity in the system have a unique, human-readable *identifier*. These identifiers are constructed using a dotted-style concatenation of the relevant code fields.

# 4. Syntax Notation

This section is about one revision outdated, with the exception of the very first section which has an updated version of the productions. Remaining sections need to be completely re-written.

This specification uses the Augmented Backus-Naur Form (ABNF) notation of [RFC2234], including the following core ABNF syntax rules defined by that specification: ALPHA (letters), CR (carriage return), DIGIT (decimal digits), DQUOTE (double quote), HEXDIG (hexadecimal digits), LF (line feed), and SP (space). The complete URI syntax is collected in Appendix A.

The HTSQL specification is a URI scheme [RFC3986] intended for use over the HTTP protocol [RFC2616]. Hence, the productions in the HTSQL syntax follow the generic syntax for uniform resource identifiers. Besides strict compliance with RFC 3986, the design of HTSQL syntax is influenced by several important considerations:

the need to represent Unicode characters to permit usage in locales where ASCII is insufficient

compatibility with HTML forms and modern browsers, with particular attention to `form-urlencoded` enctype as defined by RFC 1866

ability of HTSQL URIs to be embedded in host languages in a natural way, especially a higher level transaction syntax

when not conflicting with other goals follow the precedent of SQL-92 syntax and other modern programming languages

Here are the HTSQL productions, in all their glory.

```
input           ::= request ('/' command?)? format? ('?' test?)?

request         ::= ('/' role)? ('/' segment)*

role            ::= '~' name

segment         ::= modifier? (alias? qname locator?)?
                    selector? (';' test?)?

modifier        ::= '/' | '+' | '*' | ( '(' ('/' | '+' | '*') ')' )

alias           ::= '$' name ':='

locator         ::= '[' location (',' location)* ']'
nested_locator  ::= '(' location (',' location)* ')'
location        ::= label ('.' label)*
label           ::= '*' | literal | locator | nested_locator

selector        ::= '{' selections? ('|' selections?)? '}'
selections      ::= selection (',' selection)*
selection       ::= alias? expr (selector | '+' | '-')?

command         ::= name '(' cmd_args? ')'
cmd_args        ::= literal (',' cmd_args)* | cmd_kwds
cmd_kwds        ::= name '=' literal (',' cmd_kwds)

format          ::= '.' name ('(' cmd_args? ')')?

test            ::= and_test ('|' and_test)*
and_test        ::= not_test ('&' not_test)*
not_test        ::= '!' not_test | assign

assign          ::= name ':=' expr | comp

comp            ::= expr ( ( '==' | '=' | '<=' | '<' | '>=' | '>' |
                             '~=' | '~' | '!==' | '!=' | '!~=' | '!~' )
                           expr (',' expr)* )?

expr            ::= term (('+'|'-') term)*
term            ::= factor (('*'|'/'|'%'|'div'|'mod') factor)*
factor          ::= ('+'|'-') factor | power
```

```
power           ::= atom ('^' power)

atom            ::= literal | '(' (test | request) ')' | locator | specifier

specifier       ::= '*'
                  | qname ( '.' qname | call )* ('.' '*')?

call            ::= '(' (args | request)? ')'
args            ::= expr (',' args)* | kwds
kwds            ::= name '=' expr (',' kwds)*

name            ::= NAME
qname           ::= (NAME ':')? name
literal         ::= STRING | NUMBER
```

TODO: Remaining sections here are out-of date.

## 4.1 Character Model

### 4.1.1 Percent Encoding and Unwise Characeters

Characters not allowed in a particular context can be expressed using an escape sequence consisting of the percent character "%" followed by two hexadecimal digits representing the octet code. For example, "%20" is the escaped encoding for the US-ASCII space character. For the purposes of this specification, we casually assume all hexadecimal digits are upper-case although lower-case or mixed-case is permitted.

```
  escaped = "%" HEXDIG HEXDIG
```

For various reasons (see [RFC1738]), certain characters have been excluded from direct use within a URI. Strictly speaking, these characters must be percent-encoded. However, if the host environment (say, an embedded programming language) allows it may be acceptable to not encode spaces or other key characters.

```
  delims = "<" | ">" | "#" | "%" | '"'
  unwise = "{" | "}" | "|" | "\" | "^" | "`"
  control = <US-ASCII coded characters 00-1F and 7F hexadecimal>
  space = <US-ASCII coded character 20 hexadecimal>
```

### 4.1.2 Unicode Character Set

To allow entity names from languages not supported by the ASCII character set, HTSQL permits unicode characters using a UTF-8 representation. As traditional to XML and similar serializaiton languages, the actual character ranges allowed are limited to:

```
  unicode-xtra = [#xA0-#xD7FF] | [#xE000-#xFFFD] | [#x10000-#x10FFFF]
  unicode-char = unicode-xtra | [#x20-#x9F] | #x9 | #xA | #xD
```

Strictly speaking, one must percent-encode `unicode-extra` characters since octets in the range of 80-FF are not included in either the `reserved` or `unreserved` character set. However, all HTSQL servers should support octets in this range so that percent-encoding is not necessary.

### 4.1.3 Reserved Characters

RFC 3986 defines reserved characters as a combination of general delimiters together with scheme specific delimiters. HTSQL obviously builds upon and complies with these productions.

```
  reserved = gen-delims / sub-delims

  gen-delims = ":" / "/" / "?" / "#" / "[" / "]" / "@"

  sub-delims = "!" / "$" / "&" / "'" / "(" / ")"
             / "*" / "+" / "," / ";" / "="
```

In previous versions of this specification (2396), the right and left square brackets were in the `unwise` character set. In the most recent specification, they are only used for IPv6 addressing in the host part of the URI and are not used in the general path or query portions of the URI. One would hope that they could be added to the pchar production in future HTSQL revisions. In HTSQL we use these characters to delimit an array (following SQL-99 syntax) -- although strictly speaking these two characters must be percent-encoded.

Another character of interest is the colon, ":", which is permitted in any path segment so long as it isn't the first relative segment. This character is useful in-combination with equality symbol, following the Pascal language's notation for assignment.

### 4.1.4 Reserved Characters ala HTML Forms

An express goal of HTSQL is to be compatible with HTML form submission, in particular the `form-urlencoded` enctype first defined in RFC 1866. Query strings constructed with this mechanism percent-encode all characters which are not explicitly unreserved, using "+" to represent the space character (%20). Field submission is then an ordered list of `name "=" value` triplets, separated by "&". Hence, the following `sub-delims` have specific meaning within HTSQL:

```
reserved-html = "=" / ";" / "&" / "+"
```

In some cases the distinction between these characters or their percent-encoded variant are not significant, in those cases the following productions are used:

```
char-equals = "%3D" / "="
char-amp = "%26" / "&"
char-plus = "%2B" / "+"
char-semi = "%3B" / ";"
```

### 4.1.5 Additional Reserved Characters

Besides `reserved-html` characters, HTSQL reserves three other `sub-delims` with specific meaning. The dollar-sign, "$", is used for variable substitution purposes. The parenthesis "(" and ")", are used to differentiate between literal expressions and computed expressions.

```
reserved-expr = "$" / "(" / ")" / ","
```

Note that the usage of these three characters is somewhat problematic, in RFC 2396, they are explicitly included in the `mark` set of unreserved characters. Thankfully, most browsers do not go out of their way to percent-encode these characters when they are included in a form's `action` attribute.

```
char-dollar = "%25" / "$"
char-lpar = "%28" / "("
char-rpar = "%29" / ")"
char-comma = "%2C" / ","
```

In the cases where the distinction between the literal character and its percent-encoded variant is not important, we use the productions above.

### 4.1.6 Unreserved Characters

RFC 3986 defines unreserved characters as alpha-numeric plus a few common word connectors. As recommended, the percent-encoded versions of these characters are normalized before processing begins.

```
unreserved = ALPHA / DIGIT / "-" / "." / "_" / "~"
```

HTSQL goes beyond this set and additionally treats three more characters in the `sub-delims` set as unreserved. In many browsers the `*` is almost never percent-encoded when it should be, so this simply cannot be used reliably. Furthermore, there doesn't seem to be a pressing need to reserve the exclamation mark.

Finally, some browsers tend to always encode the single-quote no matter what the context; rendering its use as a reserved character impratical.

```
unreserved-htsql = unreserved / "'" / "*" / "!"
```

Putting the single-quote in this unreserved class deserves further discussion since it is used to delimit literal strings, and an escape mechanism is needed for single-quotes used within those strings. However, the double-quote, which is used in an analogous way for table and column names has the same problem, and it is "unwise" so it must always be percent-encoded. Following SQL-92's syntax, these delimited strings encode the quote delimiter with two adjacent quote marks. While it may be inconvenient, it is safe and consistent.

Please note that while these characters are `unreserved` in the sense of RFC 3896, they cannot be used willy-nilly. In most contexts, each of these characters has very specific meanings, regardless of their status of percent-encoded or otherwise.

### 4.1.7 Cautious Characters

There are several characters which are either `unwise`, `delims`, or `gen-delims` sets but are otherwise used by HTSQL. In practice these characters may often be used in a web-environment without percent-encoding, and hence, an HTSQL processor must be able to handle these characters appearing without being percent-encoded. expressions.

```
char-quot = "%22" / '"'
char-colon = "%3A" / ":"
char-commat = "%40" / "@"
char-verbar = "%7C" / "|"
char-lt = "%3C" / "<"
char-gt = "%3E" / ">"
char-div = "%2F" / "/"
char-mod = "%25" / "%"
```

### 4.1.8 HTSQL Characters

In HTSQL it is not necessary to use all of the `sub-delims` which RFC 3986 makes available. In particular, the single-quote, asterix, and exclamation are treated as unreserved characters.

```
reserved-htsql = gen-delims / reserved-html / reserved-expr
```

Hence, the following productions are used in HTSQL to indicate that non-reserved characters have been decoded (including UTF-8 content):

```
normal-char = unreserved-char | reserved-htsql

unreserved-char = (unicode-char - reserved-htsql - "%")
                | pct-encoded
```

The `normal-char` production signals cases where chararacters not in the reserved set are decoded. Hence, "%20" is normalized to " ", however, "%3D" is not normalized to "=" since it is reserved and may be used to delimit components of an HTSQL request. Most productions in this specification are based on normal-char.

## 4.2 Labels, Identifiers and Specifiers

The syntatic building blocks for HTSQL expresions are sequences of characters, joined by periods to indicate hierarchy or path navigation. This notation is borrowed directly from RFC 1035's "domain names": the components of a domain name are called labels.

In HTSQL, we have two sorts of labels: names and codes. Whereas names refer only to database entities such as tables and columns; codes more generally refer to any value used within a unique or primary key column. Other than having different quoting rules and slightly different syntax constraints, they behave in a similar

manner.

Broadly speaking, each kind of label (name or code), has both a regular and a delimited form. The regular variant is not only unquoted, but brings with it case-insensitive comparison rules motivated by SQL92's `regular identifier`. In contrast, the delimited form requires that the value be quoted and further implies literal comparison.

### 4.2.1 Label Comparison

For label comparisons, let us define a normal form with the following operations:

```
- leading and trailing spaces are stripped
- the content is converted to a lower-case form
- space and dash characters are converted to an underscores
- leading 0's are trimmed from all but the last character
```

When comparing labels, if either label was constructed using the "regular" unquoted syntax, then equivalence is defined as equality on their normalized representations. Otherwise, comparison is defined as equality of literal values. Unlike SQL, the space, dash and underscore are collapsed for "regular" comparison purposes: this allows for the common usage of spaces in a mixed-case table-name, yet allowing these tables to be referred to without quoting.

In HTSQL, special consideration is given to numeric values which have leading zero's used for padding. Often times the input and output of database are text-based programs which require fixed-width strings; this treatment provides the greatest compatibility with those tools at quite a minimal cost in processing complexity.

## 4.3 Specifiers and Names

A `specifier` is a dotted sequence of names which uniquely determine a path in the database structure. The names refer to either tables or columns; the dots represent joins between tables. Each component of a specifier is mandatory; although the last component may be '*' to indicate "all" columns within the given structure.

```
specifier = spec-wild | name ("." name)* ("." spec-wild)?
spec-wild = "*"
```

The components within a specifier are called names; these are labels for tables, columns, and other database entities. There are three syntax forms for an HTSQL name:

```
name = regular-name / delimited-name / special-name
```

The "regular" form uses case-insensitive comparison with a limited character-set, while the "delimited" variant is universal but requires special quoting. Special names are meant for HTSQL and proprietary extensions which are not necessarily handled via the normal interpretation rules.

### 4.3.1 Regular Names

The `regular-name` production reflects `regular identifier` in the SQL-92 specification. This syntax requires that the identifier start with a letter, provides the underscore for a word separator, and defines case-insensitive comparison. The HTSQL definition extends this notion by additionally allowing Unicode characters:

```
regular-name = regular-name-start *regular-name-char
regular-name-start = ALPHA / unicode-extra
regular-name-char = ALPHA / DIGIT / "_" / unicode-extra
```

Note that this production is awfully close to the `unreserved` production of RFC 3968, less the dash "-", period ".", and tilde "~". This is intentional as these other characters, while they are not reserved, are given specific meanings in HTSQL. If a database object name must include one of these characters, the delimted form is needed.

### 4.3.2 Delimited Names

The `delimited-name` production is a catch-all mechanism for names that don't fit into the prior category; it is motivated by the `delimited-identifier` production in the SQL-92 specification, using the double-quote for a delimiter. Including a double-quote character in a delimited name is done with two adjacent double-quotes (and not with a C-style \" escape).

```
delimited-name = DQUOTE delim-name-char+ DQUOTE
delim-name-char = unreserved-char - DQUOTE | DQUOTE DQUOTE | "+"
```

Within a delimited name, all `reserved` characters are forbidden, with the exception of the `+` symbol, which is treated as a space character (%20).

The use of the double-quote in this manner requires comment. This character is not not a RFC 3986 "reserved" character, neither is it "unreserved". In prior URI specifications it is listed as "unwise". Depending on the context in which an HTSQL URI is used, the double-quote character may need to be percent-encoded (%22). This character was chosen since it directly maps onto SQL's syntax, and since the single-quote (a reserved character) is needed to delimit literal scalar values. Since the usage of this form should be relatively uncommon, this should not be a huge complication.

### 4.3.3 Special Names

The `special-name` production is meant to provide an extension mechanism for HTSQL URIs. Following the example of the Python programming language, where a leading underscore indicates that a name is given special treatment:

```
special-name = metadata-name / reserved-name / extension-name
metadata-name = "_"
extension-name = "_" regular-name
reserved-name = "__" regular-name
```

As in the `regular-name` production above, special names are compared in a case-insensitive manner. Names beginning with a double underscore are reserved for HTSQL's usage, while those starting with a single underscore are encouraged for extension modules. The single underscore is allocated for a top-level entity like SQL92's `INFORMATION_SCHEMA`.

It is recommended that organizations that wish to augment HTSQL with proprietary extensions should use an extension-name with a reverse domain name style package name, `_com_clarkevans_myextension`, to avoid naming conflicts.

## 4.4 Identifiers and Codes

An `identifier` is a dotted sequence of components, with a correspondence to the columns in the table's primary key. Each component is either a `code` or, recursively, an `identifier`.

```
identifier = ident-regular | ident-paired
ident-paired = (identifier '.') + code
ident-regular = ident-component ('.' ident-component)*
ident-component = '(' identifier ')' | code
```

Codes are specific labels intended to match against unique or primary key column values; there are three syntax forms for an HTSQL code:

```
code = regular-code / delimited-code / array-code
```

Like the `name` productions, the "regular" form uses case-insensitive comparison with a limited character-set, while the "delimited" variant requires quoting.

### 4.4.1 Regular Codes

The `regular-code` reflects the `label` production of 1035, but extends the notation to permit numbers and allowing the underscore synonymously with the dash as a word separator; like the `name` production, Unicode characters are also granted.

```
regular-code-char = ALPHA / DIGIT / unicode-extra / "-" / "_"
regular-code = 1*regular-code-char
```

This range explicitly excludes all unprintable and "reserved" characters from RFC 3986 ("Uniform Resource Identifiers"). Furthermore, this range does not include the period "." or tilde "~" as these characters have explicit meaning in HTSQL and are particularly problematic since they cannot be escaped via percent-encoding. Note that any value matching the `regular-name` production also matches `regular-code`

### 4.4.2 Delimited Codes

The `delimited-code` reflects the `string literal` production in the SQL92 specification and is a catch-all for codes which do not fit into the prior category. It is identical to the HTSQL `literal` production using the single-quote character for its delimiter.

```
delimted-code = literal
```

### 4.4.3 Array Codes

The `array-code` production meets the requirement of SQL99's ARRAY data-type, which could possibly be used as foreign or unique key column. It's definition is straight-forward, permitting recursion and insignificant whitespace:

```
array-xtra = iwsp "," iwsp code
array-code = "[" iwsp code *array-xtra "]"
```

The use of the right and left brackets requires commentary. These two characters are neither "reserved" nor "unreserved" according to RFC 3986; and are listed as "unwise" in prior specifications. Depending on the context in which the HTSQL URI is used, these characters might need to be percent encoded, %5B for the left bracket and %5D for the right bracket. In practice, this is not usually a problem.

### 4.4.4 Identification Rules

The construction of an identifier is non-trivial, but intuitive once you get the hang of it. Primary key column values (aka codes) that are not used to refer to another row in the system are included directly in the identifier.

If columns in a table's primary key participate in a foreign key such that the target columns are exactly a unique key of the target table, then those columns collectively are represented by a single `identifier` as defined by the referenced table. Note that the actual column values need not be included in the referenced identifier; for example, a sequence number may be a unique key of a parent table, referenced in the primary key of the child table, yet, not included in the parent table's primary key. The position of this identifier is the position of the first column appearing in the foreign key constraint.

If the column is a SQL99 `REF` data-type or the HTSQL SQL92 compatible equivalent, then the column is represented by an identifier of the referenced table. The actual values of these `REF` data types are internal and should never be accessable via the HTSQL interface. If these columns are selected, then the corresponding identifier of the referenced object should be returned as the column value.

When an `identifier` occures as a component in another identifier, it is serialized using parenthesis to indicate boundary. Parenthesis may be omitted in one of two cases: when the referenced table has an identifier with exactly one code, and when the current table's identifier is composed of exactly one identifier (from a parent table) and a code. Segments are given the same order in the identifier as the corresponding columns are ordered in the table's primary key. The canonical form of an identifier always uses parenthesis and delimited

codes.

When used in matching contexts, each component of an identifier may be replaced with a wild-card component, `*`, which matches any corresponding value within the database. In all other cases, the `*` value is forbidden since it is not a valid `code`.

## 4.5 Syntax Elements

This section contains other low-level productions for variables, string literals, etc.

### 4.5.1 Variables

The variable construct in HTSQL can be used to break complicated logical steps into more managable components. Variable substution is done in a the HTSQL processor and is not passed on to the underlying database.

#### 4.5.1.1 Variable Declaration

Variable declaration is done using the ":=" assignment operator. A variant of this operator, ":@=" is used when the right-hand-side should be treated as an evaluated expression. Since both ":" and "@" are reserved characters, they may be, but are not required to be percent-encoded. To permit variable declarations within a HTML form, the dollar sign may also be percent-coded as required.

```
var-decl = char-dollar regular-name (var-decl-norm / var-decl-eval)
var-decl-norm = char-colon "=" expr-norm
var-decl-eval = char-colon char-commat "=" expr-eval
```

Note that `expr-eval` is terminated by the end of request, a reserved character such as the semi-colon ";", parenthesis ")" or amperstand "&".

#### 4.5.1.2 Variable Substution

Variables can be used as expressions directly delimited by a space character. They can also be used **within** a any quoted value, given that the dollar-sign may be percent-encoded if this is not the intended outcome; however, in this case, parenthesis are use to futher delimit the scope of the variable.

```
var-expr = char-dollar regular-name
var-subst = "$(" regular-name ")"
```

Note that in the latter case, only the reserved form of the character and surrounding parenthesis is admitted. Variables can be declared in either the current HTSQL request, or by the enclosing environment. Unresolved variables are an error.

### 4.5.2 Literal Values

In HTSQL numeric values can be expressed without any special presentation. However, regular character literals must either be quoted or put into a context bounded by reserved characters.

```
expr-literal = quoted-literal / numeric-literal
```

#### 4.5.2.1 Numeric Literals

Numbers, including the decimal point and sign, may be included without escaping in all HTSQL contexts. This is your standard definition, including the exponent.

```
numeric-literal = DIGIT+ numeric-fraction? numeric-exponent?
numeric-exponent = ( "e" / "E" ) numeric-sign? DIGIT+
numeric-fraction = "." DIGIT+
numeric-sign = char-plus / "-"
```

#### 4.5.2.2 Plain Literals

These sorts of literals are found on the right-hand-side of a comparison expression, they are percent-encoded values that end at the first reserved character (excepting "+" and "$" as detailed below).

```
plain-literal = literal-char*
literal-char = unreserved-char | "+" | var-replace
```

Within a character literal, a plus sign is treated as a space character (the actual plus sign is percent-encoded). Furthermore, variable substutiton is performed with character sequences such as `$(variable)`.

#### 4.5.2.3 Quoted Literals

In some circumstances when a plain-literal is not permitted a quoted literal form is allowed. The syntax of a quoted literal followes the precedent of SQL-92 where literal values are delimited by a single-quote character (%27). Within a quoted literal, a single-quote can be represented using two adjacent single-quotes in a manner similar to SQL-92. Note that the single-quote is treated as an unreserved character in HTSQL even though it is included in the `sub-delims` production.

```
quoted-literal-char =  literal-char - "'" | "'" "'"
quoted-literal = "'" quoted-literal-char* "'"
```

In a manner similar to plain literals, the "+" symbol is treated as a space, and variable substutiton is done with sequences like `$(variable)`.

## 4.6 Comparison Expressions

By comparison, we mean expressions which take arbitrarily typed values and return a boolean result. Comparison expressions in HTSQL substantially differ from their corresponding SQL-92 syntax, there are `evaluated` and `normal` comparison forms:

```
comparison = expression iwsp (comp-eval / comp-norm)
```

### 4.6.1 Evaluated Comparison

The intent of the evaluated comparison form is to permit HTSQL syntax to be used in an HTML query form. The value provided to the HTSQL server is percent-encoded, and it is the expectation that once this is decoded, the result matches the `expression` production for the given context.

```
comp-eval = char-commat comp-oper eval(plain-literal)
```

For example, `?ship_date@=purchase_date` would result in a comparison of the "ship_date" column with the "purchase_date" column. An example with function invocation might be `?ship_date@=today%28%29`, which represents a user typing in `today()` in a HTML field with name `ship_date@`. If the resulting expression is invalid, it is an error.

It should be noted that the at-sign, "@", is a `gen-delims` reserved character so it may or may not be percent-encoded, either form should work for this production. Note that the un-encoded form is safe to use in any place where this production would appear in an HTSQL URI. The reason this is reserved character is to delimit user-name in the authority component of the HTTP request.

### 4.6.2 Normal Comparison

While the above comparison form gives the most flexibility, users more commonly enter data they expect to be treated literally. Further, we wish to allow function calls on the URI line without requring them to be percent-encoded. This is handled with the following production:

```
comp-norm = comp-oper comp-rhs ("," comp-rhs)*
comp-rhs = expr-???? iwsp / plain-literal
```

The problem with the `comp-rhs` production is that it is ambiguous; however, we mean to choose the first alternative `expr-????` first, if it fails, then use `plain-literal` which always succeeds.

### 4.6.3 Literal Comparison

Comparisons are complicated by a historical fact: HTTP queries via the GET method are already quite well defined. When submitting a form, the right-hand-side of an equality operator is percent-encoded, substituting a "+" in lieu of "%20" for readability. This syntax is both well suported and intutive: so regardless of its divergence from SQL, HTSQL supports it directly.

For example, a query fragment such as `?department=math` treats the left-hand-side as a specifier for the column "department", and the right-hand-side as a literal value, 'math'. This behavior differs subtantially from SQL, where both sides would name columns. While this may seem problematic at first, it actually fits the common and generally accepted usage pattern. Most of the time when columns are being compared in SQL, they are part of a join clause and these sorts of comparisons are automated by HTSQL. The remaining cases when one wishes to compare one column to another are far less common. While the SQL syntax is more general, this historical pattern is more common.

### 4.6.4 Right Hand Side Expressions

In order to facilitate a more full comparison mechanism we must signal that the right-hand-side is not a `plain-literal`. Luckily, this is possible in HTSQL since the parenthesis are reserved `sub-delims` and thus they will be percent-encoded if they appear in a plain literal. Further, since the `expression` production allows any expression to be enclosed in parenthesis, we have a clean mechanism for distinguishing between these two cases.

For example, the query fragment `?purchase_date=(ship_date)` will treat the right-hand-side as a specifier for the "ship_date" column. The use of a function will also trigger this behavior, for example, `?name=lower(name)` will return all rows where the "name" column is lower case. Similarly, since the dollar-sign is reserved, variable references can be used, as well as expressions based on variables.

While the balance here is somewhat awkward, it supports the common cases with the least amount of characters and it makes the less common cases possible, if not straight-forward.

### 4.6.5 Inclusion Lists

For comparison operators, the right-hand-side can optionally be a list of values. This makes checking for inclusion a handy operation. For example, `?department=math,english` will result in an SQL fragment similar to `"department" IN ('math','english')`. The actual meaning of the inclusion list depends upon the particular operator; but it is intended to be an alternation where the left-hand-side need match only one of the options on the right.

### 4.6.6 Comparison Operators

Comparison operators are restricted to three cases: (a) where the right-most character is reserved, or where (b) the current character is a "modifier" and hence we know the operator is not finished. An exhaustive list of supported comparison operators:

```
comp-oper =  ">=" /  "<=" / "==" /  "~=" /  "~"
          / "!>=" / "!<=" / "!=" / "!~=" / "!~"
          /   "<" /   ">" / "="
comp-norm = comp-oper comp-rhs ("," comp-rhs)*
comp-rhs = plain-literal / iwsp expression iwsp
```

In the above listing, "==" means equality, while "!=" means inequality. The greater and less than symbols have

their usual meaning, as well as ">=" and "<=" which include the end-points in the comparison. The "!<=" operator is equivalent to ">", only that it can be used in an HTML form. For example, "?column>value" can be encoded using an input name of "column!<". During form submission, the equal-sign is appended giving "column!<=value". While it is ugly, it works well and the user only sees the input box and what ever label is attached to it.

The tilde "~" indicates a POSIX regular expression. If the underlying database does not support regular expressions using the POSIX syntax, it should convert the value (if possible) to an equivalent LIKE or SIMILAR expression. Direct support for the LIKE operator isn't that helpful given the standardization and power of regular expressions.

The plain equal-sign "=" all by itself has special meaning in HTSQL, the expression on the left-hand-side and right-hand-side are compared after the following transformation:

```
  COALESCE(NULLIF(
     TRANSLATE(LOWER(
       TRIM(LEADING '0' FROM
         TRIM(BOTH ' ' FROM
           CAST(expression AS TEXT)
         ))),
     '- ','__'),
  ''),'0')
```

While this pattern is a bit esoteric, it matches typical expectations of a case-insensitive match which ignores leading 0's and treats dashes, underscores and spaces as equivalent. If the left-hand-side is a literal, it can be prepared by the HTSQL translator; if searches on a particular column become common, the result of this transform can be indexed to speed up performance. This particular transform is quite useful for comparison of "codes", an extremely common use case.

## 4.7 Expressions

In HTSQL, expressions support functions, together with unary and binary operators. Expressions are recursively defined:

```
expression = function / expr-literal / specifier / expr-paren /
           / expression expr-continue

expr-continue = expr-cont-numeric



expr-compound = expression expr-comp-numeric

           /

expression = expr-paren / expr-predicate / expr-numeric
           / specifier / quoted-literal / function / expr-comp

expr-comp = "(" iwsp ( expression iwsp / comparison ) /  ")"
expr-paren = "(" iwsp expression iwsp ")"
```

Expostssions are complicated by comparisons which terminate only with an unreserved character and hence must be enclosed with parenthesis in its common form.

HTSQL uses not only symbolic operators common in programming languages, but also text versions of the same operators. To use the text versions they must be followed by and/or preceded with either whitespace or parenthesized expressions.

```
expr-rhs = wspc+ expression / iwsp expr-paren / iwsp expr-comp
expr-lhs = expression wspc+ / expr-paren iwsp / expr-comp iwsp
```

### 4.7.1 Predicates

Logical expressions are comparisons and other expressions that are joined logical conjunction, disjunction, and negation. Both symbolic and named version of these operators are provided.

```
expr-predicate = expr-negation
               / expr-alternation
               / expr-conjunction

expr-negation = "!" iwsp expression
                | "not" iwsp expr-rhs

expr-conjunction = comparison char-amp iwsp expression
                 | expression iwsp char-amp iwsp expression
                 | comparison char-amp comparison EOS
                 | expression iwsp char-amp comparison EOS
                 | expr-lhs "and" expr-rhs

expr-alternation = expression iwsp char-verbar iwsp expression
                 | expr-lhs "or" expr-rhs
```

Note that the conjunction operator permits a unparenthesized comparison on its left-hand-side; this is permitted since "&" is a reserved character and thus does not introduce ambiguity. The unparenthesized comparison is also permitted on the right-hand-side via the end of stream or segment EOS, which matches but does not consume "/" or ";".

### 4.7.2 Numeric Expressions

Conjunction of numeric values by an operator follows the SQL-92 syntax modified for URLs. Since the plus "+" character is a `sub-delims` reserved character, we allow for its literal inclusion and also for percent-encoding. Since the soldius "/" character is a `gen-delims` it must either be percent-encoded or, "div" may be used in its place.

```
expr-numeric = numeric-literal / expr-negation
             / expr-expr-addition / expr-subtraction
             / expr-division / expr-multiplication
             / expr-modulo

expr-negation = "-" expression

expr-subtraction = expression iwsp "-" iwsp expression
                 | expr-lhs "less" expr-rhs

expr-addition = expression iwsp "+" iwsp expression
              | expr-lhs "plus" expr-rhs

expr-multiplication = expression iwsp "*" iwsp expression
                    | expr-lhs "times" expr-rhs

expr-division = expression iwsp "%2F" iwsp expression
              | expr-lhs "div" expr-rhs

expr-modulo = expression iwsp "%25" iwsp expression
            | expr-lhs "mod" expr-rhs
```

Even though the SQL-92 specification does not include the modulo operator, it is provided here since many database implementations provide for this functionality. It should be used only when needed since an underlying database may not support its usage.

Operator precedence, of course, follows SQL-92; left-to-right evaluation with multiplication and division binding more tightly than addition and subtraction. These constructs are directly translated into SQL, there is no special handling done by HTSQL.

### 4.7.3 Function Application

Functions in HTSQL do not follow SQL-92 syntax, which have optional parameters and a sort-of keyword argument mechanism. Instead, HTSQL functions are modeled more after Python and other languages which

have by-position and keyword arguments. The value arguments of a function call is a normal expressions - that is, they are treated as specifiers and not as literal values unless single quoted.

```
function = regular-name iwsp "(" iwsp func-args? iwsp ")"
func-pair = regular-name iwsp "=" expression
func-item = func-pair / expression
func-args = func-item ( iwsp "," iwsp func-item )*
```

Note that the equal sign, parenthesis, and comma are reserved `sub-delims` charaters and that the percent-encoded variants are not delimiters in these productions (and indeed must be percent-encoded if they occur in a value expression).

In some cases, a particular function may optionally treat particular `specifier` arguments as values in a pre-defined enumeration rather than as a column specifier. For example, the `trim` function may use an argument value of `both` not as a column specifier, but rather as a flag to trim spaces on both sides of the given value.

## 4.8 Request URIs

HTSQL requests are a specialization of RFC 3986's `hier-part` and `query` assuming that relative resolution has already been performed.

At the top level, a request can either be a user defined resource, or an HTSQL action. In an HTSQL context, either the slash "/" or the semi-colon ";" may be used to separate path segments. An HTSQL system treats them as synonymously.

```
request = "/" context? ( user-resource / action query? )
context = schema-name (separator segment)* separator
separator = "/" / ";"
schema-name = name
```

The `schema-name` production is simply a `name` which refers to a schema in the underlying database system.

### 4.8.1 User Resources

While HTSQL provides access to a database, many static or semi-static resources are requried for screens, and other purposes. Anywhere within an HTSQL request, a path segment that begins with the tilde "~" indicates a resource the meaning of which is out-of-scope of this specification.

The user resource production must exactly follow RFC 3986; in particular, the `path-rootless` and `query` productions are used here.

```
user-resource = "~" rfc3986-path-rootless ("?" rfc3986-query)?
```

Note that a user resource may be contextualized, that is, they can be prefixed with a schema, tables, and other parameters. It is intended that user-resources may be dependent upon not only preceding content but the meta-data associated with this information.

### 4.8.2 Query Fragments

The query fragment in HTSQL are expressions with a boolean value, alternatively permitting variable declarations. Note that query fragments are either terminated with a reserved character, or the end of the request.

```
query-part = comparison / expr-predicate / var-decl
query = query-part (";" query-part)*
```

Query components are resolved with respect to the left-most table mentioned in the request's path segments.

### 4.8.3 Path Segment

Path segments are either query components included within the actual request path, or they are table names and column specifiers.

```
segment = query / table-name selector? picker?
selector = iwsp "(" iwsp expression (iwsp "," iwsp expression)* ")"
picker = "!" identifier
```

# 5. Semantics

This chapter describes the translations between HTSQL and the resulting SQL queries. The result of an HTSQL query is a flat list, having at most one row from each member of the right-most context. This result set is filtered in several ways. A set of links from the driving class outward, as part of the filter is used to restrict values.

## 5.1 Context

...

# 6. References

## 6.1 Normative References


[ISO9075-1992] International Standards Organization, " Database Language SQL, 1992 ", ISO/EIC 9075:1992, July 1992.

[RFC2234] Crocker, D.H. and P. Overell, " Augmented BNF for Syntax Specifications: ABNF", RFC 2234, November 1997.

[RFC2616] Felding, R., Gettys, J., Mongul, J., Frystyk, H., Mastiner, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, " Uniform Resource Identifiers (URI): Generic Syntax", RFC 3986, January 2005.


## 6.2 Informative References


[CSV] "", 2004.

[HTML] "", 2004.

[HTML4] "", 2004.

[ISO8601] "", 2004.

[ISO9075-1999] International Standards Organization, " Database Language SQL, 1999 ", ISO/EIC 9075:1999, September 1999.

[ISO9075-2003] International Standards Organization, " Database Language SQL, 2003 ", ISO/EIC 9075:2003, August 2003.

[JSON] "", 2004.

[PGSQL] "", 2004.

[POSIX_1003.2] "", 2004.

[REC-XML] "", 2004.

[REST] Fielding, R., " Architectural Styles and the Design of Network-based Software Architectures. ", 2000.

[RFC1738] "", 2004.

[RFC1808] "", 2004.

[RFC2086] "", 2004.

[RFC2396] "", 2004.

[RFC2396BIS] "", 2004.

[RFC2732] "", 2004.

[SQL98] "", 2004.

[UNICODE] "", 2004.

[XML] "", 2004.

[XSL] "", 2004.

[YAML] "", 2004.

## Author's Address

**Clark C. Evans**
Prometheus Research, LLC.
315 Whitney Ave.
New Haven, CT 06511
US
Phone: +1 734 418 8644
EMail: info@clarkevans.com
URI: http://clarkevans.com

# A. Collected ABNF for URI

## B. Sample Database Schema

```
--
-- SAMPLE SCHEMA, SQL-92 CONSTRUCTS ONLY
--
DROP SCHEMA tm CASCADE;
CREATE SCHEMA tm;

-- PostgreSQL, not SQL99, Syntax
CREATE OR REPLACE FUNCTION htsql_normalize(text)
  RETURNS text
  AS 'SELECT COALESCE(NULLIF(
              TRANSLATE(LOWER(
                TRIM(LEADING ''0'' FROM
                  TRIM(BOTH '' '' FROM
                    CAST($1 AS TEXT)
                  ))),
              ''- ''''__''),
           ''''),''0'');
  ' LANGUAGE SQL IMMUTABLE STRICT;

CREATE TABLE tm.project (
    proj_id        VARCHAR(16),
    name           VARCHAR(64) NOT NULL,
    description    VARCHAR(2000),
    CONSTRAINT project_pk
      PRIMARY KEY (proj_id)
);

CREATE TABLE tm.employee (
    empl_code      VARCHAR(16),
    full_name      VARCHAR(64) NOT NULL,
    is_contractor  BOOLEAN,
    email          VARCHAR(64),
    CONSTRAINT employee_pk
      PRIMARY KEY (empl_code)
);

CREATE DOMAIN tm.status AS VARCHAR(8);
ALTER DOMAIN  tm.status
   ADD CONSTRAINT status_check
     CHECK (VALUE IN ('done','review','planned'));

CREATE TABLE tm.task (
    proj_id        VARCHAR(16)
      REFERENCES tm.project(proj_id),
    task_no        INTEGER,
    assigned_to    VARCHAR(16)
      REFERENCES tm.employee(empl_code),
    status         tm.status,
    name          VARCHAR(64) NOT NULL,
    CONSTRAINT task_pk
      PRIMARY KEY (proj_id, task_no)
);

CREATE TABLE tm.restricted_info (
    empl_code      VARCHAR(16)
      REFERENCES tm.employee(empl_code),
    billing_rate   INTEGER NOT NULL,
    tax_id         VARCHAR(16) UNIQUE,
    birth_date     DATE,
    CONSTRAINT restricted_info_pk
      PRIMARY KEY (empl_code)
);

INSERT INTO tm.project VALUES ('MEYERS',
  'Meyer''s Residence', 'insulation and winterizing');
INSERT INTO tm.project VALUES ('SSMall',
  'South Square Mall', 'two new store fronts');
INSERT INTO tm.project VALUES ('THOM-LLP',
  'Tom Thompson, LLP.', 'fix up room for new associate');
```

```
INSERT INTO tm.employee VALUES ('ADAM',
  'Adam O''Brian', FALSE, 'adam@example.com');
INSERT INTO tm.employee VALUES ('ARONSON',
  'Mary Aronson', FALSE, 'mary2@example.com');
INSERT INTO tm.employee VALUES ('SMITH',
  'Ron Smith', TRUE, 'john@example.com');
INSERT INTO tm.employee VALUES ('SMITH-A',
  'Alfred Smith', TRUE, NULL);

INSERT INTO tm.task VALUES ('MEYERS',1,
  'ARONSON','done','Purchase Materials');
INSERT INTO tm.task VALUES ('MEYERS',2,
  'SMITH','review','Strip Wall Paint');
INSERT INTO tm.task VALUES ('MEYERS',3,
  NULL,'planned','Remove Refuse');
INSERT INTO tm.task VALUES ('SSMall',1,
  'ADAM','review','Install Slider Door');

INSERT INTO tm.restricted_info VALUES ('ARONSON',
  26, '222-22-1492', '03-01-1961');
INSERT INTO tm.restricted_info VALUES ('SMITH',
  22, '444-44-4444', '08-15-1965');
```

# Index