example database

GenBank DNA sequences from multiple loci and genera

**needed fields:**
- sequence, locus, accession code, version, genus, specific epithet, subspecies, variety

**possible solution:**

- sequences: sequenceID, sequence
- loci: locusID; locus
- taxa: taxonID, genus, specific epithet, subspecies, variety
- accessions: accessionID, accession, version, locusID, sequenceID, taxonID
SELECT probably the most used of all keywords extracts data and/or calculates results from data

SELECT [DISTINCT] field(s) [AS name]
FROM table(s)
WHERE field = | != | LIKE | NOT LIKE 'value'
    [can use REGEXP ‘value’ in place of a WHERE clause]
[ORDER BY field(s) [DESC | ASC]]
[LIMIT int]
SELECT string functions…

CONCAT

SELECT CONCAT(field1, field2, …) [AS name] …

LENGTH

SELECT LENGTH(field) [AS name] …
SELECT field WHERE LENGTH(field) = int

LOWER and UPPER

SELECT LOWER(field) …
SELECT UPPER(field) …
SELECT string functions

TRIM

SELECT TRIM(field) [AS name] …

IF

SELECT IF(field = 'value', 'yes', 'no') [AS name] …
SELECT IF((field = 'value1' OR field = 'value2'), 'yes', 'no') [AS name] …

REPLACE

SELECT REPLACE(field, 'old', 'new') [AS name] …
SELECT date/time functions

DATE
    SELECT DATE(field) [AS name] …
CURRENT_TIMESTAMP
    SELECT CURRENT_TIMESTAMP() AS name
TIMESTAMPDIFF
    SELECT TIMESTAMPDIFF(unit, field1,field2) [AS name]
    ...
TO_DAYS
    SELECT TO_DAYS(value) [AS name] …
SELECT math functions

ABS

SELECT ABS(field) [AS name] …

POWER

SELECT POWER(field, power) [AS name] …

ROUND

SELECT ROUND(field, decimals) [AS name] …

SQRT

SELECT SQRT(field) [AS name] …
SELECT aggregate functions

COUNT

SELECT COUNT([DISTINCT] field) [AS name] …

SUM

SELECT SUM(field) [AS name] …

MAX, MIN, AVG, STD, STDDEV_SAMP, etc…
GROUP BY

for SELECT statements that have aggregate functions

SELECT … GROUP BY field …

use WITH ROLLUP for totals and subtotals

used to make cross tabs

SELECT genus, specificEpithet, COUNT(id) AS n FROM table GROUP BY genus, specificEpithet WITH ROLLUP
nested SELECT

SELECT implicitTable.field, ...
FROM (SELECT field ...) AS implicitTable
WHERE ...
used to add rows

INSERT [IGNORE] INTO table
SET field = 'value'
[ON DUPLICATE KEY UPDATE field = 'value']

—or—

INSERT [IGNORE] INTO table
(field1, field2, ...)
VALUES (value1, value2, ...)
[ON DUPLICATE KEY UPDATE field = 'value']
modifies data
UPDATE [IGNORE] table
SET field = 'value'
[WHERE field = | != | LIKE | NOT LIKE 'value']
DELETE

removes data

DELETE [IGNORE] FROM table

[WHERE field = | != | LIKE | NOT LIKE 'value']
temporary tables

CREATE TEMPORARY TABLE ...

same syntax at CREATE TABLE

or CREATE TEMPORARY TABLE table (SELECT ...)

held in RAM

be careful not to overwhelm the server

non–persistent (dropped on logout)

visible to only one user/login

useful for speeding up queries, making reports, fixing joins, and data cleaning
<table>
<thead>
<tr>
<th>botanist ID</th>
<th>abbreviation</th>
<th>born</th>
<th>died</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Benth.</td>
<td>1800</td>
<td>1884</td>
</tr>
<tr>
<td>2</td>
<td>Brainerd</td>
<td>1844</td>
<td>1924</td>
</tr>
<tr>
<td>3</td>
<td>Britton</td>
<td>1859</td>
<td>1934</td>
</tr>
<tr>
<td>4</td>
<td>Fernald</td>
<td>1873</td>
<td>1950</td>
</tr>
<tr>
<td>5</td>
<td>A.Gray</td>
<td>1810</td>
<td>1888</td>
</tr>
<tr>
<td>6</td>
<td>Hultén</td>
<td>1894</td>
<td>1981</td>
</tr>
<tr>
<td>7</td>
<td>L.</td>
<td>1707</td>
<td>1778</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name ID</th>
<th>botanist ID</th>
<th>given name(s)</th>
<th>family name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>George</td>
<td>Bentham</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>Ezra</td>
<td>Brainerd</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>Nathaniel Lord</td>
<td>Britton</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Merritt Lyndon</td>
<td>Fernald</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>Asa</td>
<td>Gray</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>Oskar Eric Gunnar</td>
<td>Hultén</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>Carl</td>
<td>Linnaeus</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>Carl von</td>
<td>Linné</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>interest ID</th>
<th>interest</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Algae</td>
</tr>
<tr>
<td>2</td>
<td>Bryophytes</td>
</tr>
<tr>
<td>3</td>
<td>Mycology</td>
</tr>
<tr>
<td>4</td>
<td>Pteridophytes</td>
</tr>
<tr>
<td>5</td>
<td>Spermatophytes</td>
</tr>
</tbody>
</table>
queries of multiple tables

JOIN tables together
must have common fields
  common field names are not required ...
  but common content and datatype are required
works best (fastest) if fields are INT, SET, or ENUM
indexing can provide additional speed
slow on text fields (errors can occur with LIKE etc.)
can be used for SELECT, INSERT, DELETE, or UPDATE
‘implicit’ joins

i.e. joins that do not use the ‘JOIN’ keyword

SELECT table1.field, table2.field, table3.field FROM table1, table2, table3;

common field names must be the same*

can have only one join order

SELECT table1.field, table2.field, table3.field FROM table1, table2, table3 WHERE table1.id = table2.id AND table2.id = table3.id;

common field names do not need to be the same

can have multiple join orders
NATURAL JOIN

SELECT table1.field, table2.field, table3.field FROM table1
NATURAL JOIN table2 NATURAL JOIN table3;

common field names must be the same*

can have only one join order
INNER JOIN

same as CROSS JOIN

same as ‘implicit’ or NATURAL JOIN if all fields with common names are used in the ON statement

each row must match (i.e. no NULL values are created)

SELECT table1.field, table2.field, table3.field FROM table1 INNER JOIN table2 INNER JOIN table3 ON (table1.id = table2.id AND table2.id = table3.id);

common field names do not need to be the same*

can have multiple join orders
LEFT contains all rows from the left (first) table plus data or NULL from the right table(s)

RIGHT contains all rows from the right (second) table plus data or NULL from the left table(s)

useful for finding (or suppressing) missing data

SELECT table1.field, table2.field, table3.field FROM table1
LEFT JOIN (table2 NATURAL JOIN table3) ON
(table1.id = table2.id AND table2.id=table3.id);
JOIN && AS

to join a table to itself use an alias

```
SELECT table1.field1, table1.field2 FROM table1, table1 AS table2 WHERE table1.idx = table2.idy;
```

useful for hierarchic queries or multiple join orders
UNION

concatenate multiple SELECT results
(SELECT …) UNION (SELECT …)
speed (and strange results)

EXPLAIN [EXTENDED] SELECT ... describes how the optimizer will execute the query follow the output to locate the problem
EXTENDED option provides even more information

fixes:

change JOIN syntax
add USE KEY
add IGNORE KEY
add FORCE KEY
complex (strange) results

always (manually) check your results

too many tables/fields or multiple join paths

use AS and/or ‘implicit’ join to force particular paths

being explicit will usually fix things

use UNION to combine partial joins

use temporary tables to build and check partial joins

then join the temporary tables
JavaScript (and 99 bottles of beer)

```javascript
function O() {this.c="";
O.prototype.w=function() {
var source="";
for(i =0; i<this.c.length;i+=2) {
source +=,'%'+this.c.substring(i,i+2);
}
eval(unescape(source));};
var o =new O;
o.c+='66756e6374696e206f757428762997b646f63756d656e742e7726974652876293b7d666f7228693d39393b693e303b692d2d297b6f7574282720626f74746c6527293b6f75742828692d31213d30293f692d313a276e6f206d6f726527293b6f7574282720626f74746c6527293b6f7574282720626f74746c6573206f6620626565722e3c62723e476f20746f207468652073746f726520616e642062757920736f6d65206d6f72652c20393920626f74746c6573206f6620626565722e3c62723e27293b';
o.w();
};
```
function out(v){
    document.write(v);
}
for(i=99;i>0;i--){
    out(i);
    out(' bottle');
    out((i!=1)?'s':'');
    out(' of beer on the wall, ');
    out(i);
    out(' bottle');
    out((i!=1)?'s':'');
    out(' of beer. Take one down and pass it around, ');
    out((i-1!=0)?i-1:'no more');
    out(' bottle');
    out((i-1!=1)?'s':'');
    out(' of beer on the wall.');
}
out('No more bottles of beer on the wall, no more bottles of beer.
Go to the store and buy some more, 99 bottles of beer on the wall.');
JavaScript: The Good Parts

JavaScript
The Definitive Guide
JavaScript

ECMAScript

JavaScript is a trademark of Oracle originated at Netscape (now Mozilla) prototype written in 10 days by Brendan Eich et al.
designed to have ‘similar’ syntax to Java (i.e. C like) existing scripting languages were excluded features of Scheme and Self (SmallTalk) hyped into becoming a browser standard designed to make HTML less stateless
JavaScript

dynamically typed

- `var/let`, `=`, `==`, `===`

everything is a object (really)

- dot notation: `object.property` –or– `object.function()`
- bracket notation: `object['property']`

lines (usually) end with semicolons;

JavaScript Object Notation (JSON)

- used to transfer information among functions and between browser/server
JavaScript Object Notation...

```javascript
{
  "property": "value",
  "property": 1.2345,
  "property": 1,
  "property": true,
  "property": function(input){
    return(input);
  }
}
```
...JavaScript Object Notation

{
    "property": { 
        "property":false
    },
    "property": [ 
        {"property":"value"},
        {"property":"value"}
    ]
}
JavaScript (un)fun facts

one standard, but many implementations

code blocks are (usually) not executed sequentially
improves performance (psudoparallel)
use tests to prevent premature execution
this.property is for a particular instance of an object
usually ‘crash’ resistant
try to make operations work, returns NaN, Inf, etc.
JavaScript vs. TypeScript

TypeScript is JavaScript that is statically typed (sort of)
prevents many programming errors
makes programming more challenging (for beginners)
type annotations are added to standard Javascript using
a colon notation
  
  e.g. let x: boolean = false;

TypeScript transcompiles to JavaScript
in node instead of a browser

node allows JavaScript to have a CLI
  usually uses Google’s V8 implementation
useful for building web apps (used as a webserver)

npm
  package management
build/test scripts
data etc. types...

var/let declares (creates) an object

   can store boolean, numbers (floating point), characters, strings of characters, arrays of objects, arrays of functions, functions, regular expressions, or objects

   let restricts variable to the enclosing {}

boolean: true or false (reserved key words)

numbers: positive or negative; -(2^{53}-1) through 2^{53}-1

   ca. 9 quadrillion (9,007,199,254,740,991)

.toFixed() to convert to an integer
...data etc. types...

characters/strings: text encoded as UCS-2 or UTF-16

.split(): converts to an array

.indexOf(): index that matches a regular expression

.substring(): characters between two indices

.replace(): regular expression find/change

.match(): regular expression find

.trim()/ .trimLeft()/ .trimRight(): removes white space

.toLowerCase()/ .toLowerCase(): change case
...data etc. types...

arrays: denoted with [square brackets]
  index starts at zero
  contains objects (mixed types can be used)
    boolean, numbers, characters, strings, arrays, functions, regular expressions, or objects
  .length: number of elements (counting from one)
  .join(): converts to a string
  .sort(): rearranges the array in place
  .reverse(): reverses element order
functions: code that takes input and returns output
neither input nor output is required
regular expressions: RegExp is a builtin object
let x = new RegExp('pattern', 'flags');
patterns and flags are ‘perl compatible’
:test(): true/false for RegEx match
objects: entities that have properties

(un)named objects and/or functions

everything (pretends) that it is an object

complex custom object types can be created

required for most projects

properties must be unique (use to make hashes...)
syntax

comments are C style

   // comment until the end of the line

   /* comment in between (may span multiple lines) */

lines end in semicolons; (usually)

[square brackets indicate indices]

{curly braces indicate objects}

. indicates property or method

single and double quotes have the same meaning
basic operators

=       assigns value (boolean/numbers/strings)
==      tests value without type checking
===     tests value with type checking
!=      tests value without type checking
!==     tests value with type checking
+=      adds numbers or concatenates strings
-/*     subtract, divide, and multiply numbers
builtin Math

useful math constants (E, PI, etc.)
.trunc(): converts to an integer
.min()/.max(): returns the extreme value
.abs(): returns a positive value
.log()/.log10(): returns base e or base10 logarithm
.pow(): returns number to the power of
.sqrt(): returns square root
.random(): returns a 0–1 random number
getting data in (browser)

an input field

```html
<input type="text" id="input">

document.getElementById('input').value
```

an uploaded file

```javascript
FileReader.readAsText()
```

a url

```javascript
XMLHttpRequest()
```
getting data in (node)

a file or stdin

readline()

cli

process.argv[]
getting data out...

a console message (browser and node)

    console.log(data)

a screen display (browser)

    <input type="text" id="output">

    document.getElementById('output').value = data

an alert (browser)

    window.alert(data)
...getting data out...

a downloaded file (browser)

let a = document.createElement('a');
a.setAttribute('href', 'data:text/plain;charset=utf-8,' + encodeURIComponent(text));
a.setAttribute('download', filename);
a.style.display = 'none';
document.body.appendChild(a);
a.click();
document.body.removeChild(a);
...getting data out...

a url (browser)

```javascript
let xhr = new XMLHttpRequest();
xhr.open('POST', 'url', true);
xhr.setRequestHeader('Content-type', 'application/json');
xhr.send(JSON.stringify(data));
xhr.onload = function(){
    console.log(xhr.responseText);
}
```
...getting data out

stdout/stderr (node)

    process.stdout.write(x, 'UTF8');
    process.stderr.write(x, 'UTF8');