CS 61A Fall 2023

SQL, Final Review

Discussion 13: November 29, 2023

Note: For extended explanations of the concepts on this discussion, feel free to look at the **Appendix** section at the end of the worksheet.

SQL

SQL is a declarative programming language. Statements do not describe computations directly, but instead describe the desired result of some computation. It is the role of the query interpreter of the database system to plan and perform a computational process to produce such a result.

> For this discussion, you can test out your code at sql.cs61a.org. The `records` table should already be loaded in.

Select Statements

The following questions involve the `records` table:

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

To see the entire `records` table, go to sql.cs61a.org and enter `SELECT * FROM records;`. Note that you can answer the questions without seeing the entire table.

Q1: Oliver Employees

Write a query that outputs the names of employees that Oliver Warbucks directly supervises.

```
SELECT name FROM records WHERE supervisor = "Oliver Warbucks";
```

Q2: Self Supervisor

Write a query that outputs all information about employees that supervise themselves.

```
SELECT * FROM records WHERE name = supervisor;
```

Q3: Rich Employees

Write a query that outputs the names of all employees with salary greater than 50,000 in alphabetical order.

```
SELECT name FROM records WHERE salary > 50000 ORDER BY name;
```

Joins

The following questions involve the `records` and `meetings` tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q4: Oliver Employee Meetings

Write a query that outputs the meeting days and times of all employees directly supervised by Oliver Warbucks.

```
SELECT m.day, m.time FROM records AS r, meetings AS m WHERE r.division = m.division
AND r.supervisor = "Oliver Warbucks";
```

Q5: Different Division

Write a query that outputs the names of employees whose supervisor is in a different division.

```
SELECT e.name FROM records AS e, records AS s WHERE e.supervisor = s.name AND e.division
    != s.division;
```

Q6: Middle Manager

A middle manager is a person who is both supervising someone and is supervised by someone different. Write a query that outputs the names of all middle managers.

```
SELECT b.name FROM records AS a, records AS b WHERE a.supervisor = b.name AND b.
    supervisor != b.name;
```

Aggregation

The following questions involve the `records` and `meetings` tables:

records

Name	Division	Title	Salary	Supervisor
Alyssa P Hacker	Computer	Programmer	40000	Ben Bitdiddle
...	...	...	...	...
Robert Cratchet	Accounting	Scrivener	18000	Eben Scrooge

meetings

Division	**Day**	**Time**
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

Q7: Supervisor Sum Salary

Write a query that outputs each supervisor and the sum of salaries of all the employees they supervise.

```
SELECT supervisor, SUM(salary) FROM records GROUP BY supervivor;
```

Q8: Num Meetings

Write a query that outputs the days of the week for which fewer than 5 employees have a meeting. You may assume no department has more than one meeting on a given day.

```
SELECT m.day FROM records AS e, meetings AS m WHERE e.division = m.division GROUP BY m.
    day HAVING COUNT(*) < 5;
```

Q9: Rich Pairs

Write a query that outputs all divisions for which there is more than one employee, and all pairs of employees within that division that have a combined salary less than 100,000.

```
SELECT e1.division FROM records AS e1, records AS e2 WHERE e1.name != e2.name AND e1.
    division = e2.division
GROUP BY e1.division HAVING MAX(e1.salary + e2.salary) < 100000;
```

Final Review
Recursion, Sequences

Q10: Subsequences

A subsequence of a sequence `S` is a subset of elements from `S`, in the same order they appear in `S`. Consider the list `[1, 2, 3]`. Here are a few of its subsequences `[]`, `[1, 3]`, `[2]`, and `[1, 2, 3]`.

Write a function that takes in a list and returns all possible subsequences of that list. The subsequences should be returned as a list of lists, where each nested list is a subsequence of the original input.

In order to accomplish this, you might first want to write a function `insert_into_all` that takes an item and a list of lists, adds the item to the beginning of each nested list, and returns the resulting list.

```
def insert_into_all(item, nested_list):
    """Return a new list consisting of all the lists in nested_list,
    but with item added to the front of each. You can assume that
     nested_list is a list of lists.

    >>> nl = [[], [1, 2], [3]]
    >>> insert_into_all(0, nl)
    [[0], [0, 1, 2], [0, 3]]
    """
    return [[item] + lst for lst in nested_list]

def subseqs(s):
    """Return a nested list (a list of lists) of all subsequences of S.
    The subsequences can appear in any order. You can assume S is a list.

    >>> seqs = subseqs([1, 2, 3])
    >>> sorted(seqs)
    [[], [1], [1, 2], [1, 2, 3], [1, 3], [2], [2, 3], [3]]
    >>> subseqs([])
    [[]]
    """
    if not s:
        return [[]]
    else:
        subset = subseqs(s[1:])
        return insert_into_all(s[0], subset) + subset
```

Trees

Q11: Long Paths

Implement `long_paths`, which returns a list of all *paths* in a tree with length at least `n`. A path in a tree is a linked list of node values that starts with the root and ends at a leaf. Each subsequent element must be from a child of the previous value's node. The *length* of a path is the number of edges in the path (i.e. one less than the number of nodes in the path). Paths are listed in order from left to right. See the doctests for some examples.

```
def long_paths(tree, n):
    """Return a list of all paths in tree with length at least n.

    >>> t = Tree(3, [Tree(4), Tree(4), Tree(5)])
    >>> left = Tree(1, [Tree(2), t])
    >>> mid = Tree(6, [Tree(7, [Tree(8)]), Tree(9)])
    >>> right = Tree(11, [Tree(12, [Tree(13, [Tree(14)])])])
    >>> whole = Tree(0, [left, Tree(13), mid, right])
    >>> for path in long_paths(whole, 2):
    ...     print(path)
    ...
    <0 1 2>
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 6 9>
    <0 11 12 13 14>
    >>> for path in long_paths(whole, 3):
    ...     print(path)
    ...
    <0 1 3 4>
    <0 1 3 4>
    <0 1 3 5>
    <0 6 7 8>
    <0 11 12 13 14>
    >>> long_paths(whole, 4)
    [Link(0, Link(11, Link(12, Link(13, Link(14)))))]
    """
    paths = []
    if n <= 0 and tree.is_leaf():
        paths.append(Link(tree.label))
    for b in tree.branches:
        for path in long_paths(b, n - 1):
            paths.append(Link(tree.label, path))
    return paths
```

Linked Lists

Q12: Deep Linked List Length

A linked list that contains one or more linked lists as elements is called a *deep* linked list. Write a function `deep_len` that takes in a (possibly deep) linked list and returns the *deep length* of that linked list. The deep length of a linked list is the total number of non-link elements in the list plus the total number of elements contained in all contained lists. See the function's doctests for examples of the deep length of linked lists.

Hint: Use `isinstance` to check if something is an instance of an object.

```
def deep_len(lnk):
    """ Returns the deep length of a possibly deep linked list.

    >>> deep_len(Link(1, Link(2, Link(3))))
    3
    >>> deep_len(Link(Link(1, Link(2)), Link(3, Link(4))))
    4
    >>> levels = Link(Link(Link(1, Link(2)), \
            Link(3)), Link(Link(4), Link(5)))
    >>> print(levels)
    <<<1 2> 3> <4> 5>
    >>> deep_len(levels)
    5
    """
    if lnk is Link.empty:
        return 0
    elif not isinstance(lnk, Link):
        return 1
    else:
        return deep_len(lnk.first) + deep_len(lnk.rest)
```

Generators

Q13: Powers

Implement `powers`, a generator function that takes positive integers `n` and `k`. It yields all integers `m` that are both powers of `k` and whose digits appear in order in `n`.

Assume the following functions are defined:

- `is_power(base, s)`: `is_power` takes in a positive integer `base` and a non-negative integer `s` and returns `True` if there is some number `n` where `pow(base, n) = s`
- `curry2`: `curry2 = lambda f: lambda x: lambda y: f(x, y)`

Hint: `filter(func, seq)` returns an iterator that yields all the values `x` in `seq` where `func(x)` is truthy.

```
def powers(n, k):
    """Yield all powers of k whose digits appear in order in n.

    >>> sorted(powers(12345, 5))
    [1, 5, 25, 125]
    >>> sorted(powers(54321, 5))  # 25 and 125 are not in order
    [1, 5]
    >>> sorted(powers(2493, 3))
    [3, 9, 243]
    >>> sorted(powers(2493, 2))
    [2, 4]
    >>> sorted(powers(164352, 2))
    [1, 2, 4, 16, 32, 64]
    """
    def build(seed):
        """Yield all non-negative integers whose digits appear in order in seed.
        0 is yielded because 0 has no digits, so all its digits are in seed.
        """
        if seed == 0:
            yield 0
        else:
            for x in build(seed // 10):
                yield x
                yield x * 10 + seed % 10
    yield from filter(curry2(is_power)(k), build(n))
```

Scheme

Q14: Group by Non-Decreasing

Define a function `nondecreaselist` that takes in a scheme list of numbers and outputs a list of lists, which overall has the same numbers in the same order, but grouped into lists that are non-decreasing.

For example, if the input is a stream containing elements

```
(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1)
```

the output should contain elements

```
((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1))
```

Note: The skeleton code is just a suggestion; feel free to use your own structure if you prefer.

```
(define (nondecreaselist s)

    (if (null? s)
        nil
        (let ((rest (nondecreaselist (cdr s)) ))
            (if (or (null? (cdr s)) (> (car s) (car (cdr s))))
                (cons (list (car s)) rest)
                (cons (cons (car s) (car rest)) (cdr rest))
            )
        )
    )
)

(expect (nondecreaselist '(1 2 3 1 2 3)) ((1 2 3) (1 2 3)))

(expect (nondecreaselist '(1 2 3 4 1 2 3 4 1 1 1 2 1 1 0 4 3 2 1))
        ((1 2 3 4) (1 2 3 4) (1 1 1 2) (1 1) (0 4) (3) (2) (1)))
```

Programs as Data

Q15: Or with Multiple Args

Recall `make-or` from Homework 9. Implement `make-long-or`, which returns, as a list, a program that takes in any number of expressions and `or`'s them together (applying short-circuiting rules). This procedure should do this without using the `or` special form. Unlike the `make-or` procedure from Homework 9, the arguments will be passed in as a list named `args`.

The behavior of the `or` procedure is specified by the following doctests:

```
scm> (define or-program (make-long-or '((print 'hello) (/ 1 0) 3 #f)))
or-program
scm> (eval or-program)
hello
scm> (eval (make-long-or '((= 1 0) #f (+ 1 2) (print 'goodbye))))
3
scm> (eval (make-long-or '((> 3 1))))
#t
scm> (eval (make-long-or '()))
#f
```

```
(define (make-long-or args)
    (cond
        ((null? args) #f)
        (else
            `(let ((v1 ,(car args)))
                (if v1 v1 ,(make-long-or (cdr args))))
        )
    )
)
```

Appendix: Explanation of Material

Select Statements

A `SELECT` statement creates a table. The following statement creates a single-row table with columns named `first` and `last`.

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last;
Ben|Bitdiddle
```

A `UNION` statement creates a table that consists of the rows of two tables with the same number of columns.

```
sqlite> SELECT "Ben" AS first, "Bitdiddle" AS last UNION
...> SELECT "Louis", "Reasoner";
Ben|Bitdiddle
Louis|Reasoner
```

A `FROM` clause specifies an existing table from which information can be drawn. The following statement creates a table that consists of the `name` and `division` columns from an existing table `records`.

```
sqlite> SELECT name, division FROM records;
Alyssa P Hacker|Computer
...
Robert Cratchet|Accounting
```

The special syntax `SELECT *` selects all columns from an existing table. It's an easy way to display the contents of a table.

```
sqlite> SELECT * FROM records;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge
```

The general syntax of a `SELECT` statement is as follows:

```
SELECT [columns] FROM [tables]
WHERE [condition] ORDER BY [columns] LIMIT [limit];
```

- `SELECT [columns]`
  - Specifies the columns of our output table; `[columns]` is a comma-separated list of column names, and `*` selects all columns
- `FROM [tables]`
  - Specifies the existing tables from which we select columns; `[tables]` is a comma-separated list of table names
- `WHERE [condition]`
  - Filters the output table to include only rows which satisfy the `[condition]`, a boolean expression
- `ORDER BY [columns]`
  - Orders the rows of the output table by the given comma-separated list of columns
- `LIMIT [limit]`
  - Limits the number of rows in the output table to the integer `[limit]`

The following `SELECT` statement lists all information about employees with the "Programmer" title.

```
sqlite> SELECT * FROM records WHERE title = "Programmer";
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle
Cy D Fect|Computer|Programmer|35000|Ben Bitdiddle
```

The following `SELECT` statement lists the names and salaries of each employee in the accounting division, sorted in descending order by their salaries.

```
sqlite> SELECT name, salary FROM records
...> WHERE division = "Accounting" ORDER BY salary DESC;
Eben Scrooge|75000
Robert Cratchet|18000
```

All valid SQL statements are terminated by a semicolon.

An SQL statement may have any number of line breaks and any amount of whitespace. But keep in mind that consistent line-breaking and indentation make your code a lot easier to read!

Joins

Joining tables is a fundamental database operation.

So far, we've been able to examine and filter through the information in individual rows. But what if we want to reveal relationships between rows of the same table or with information in a different table? The tool we use for this is called a join, which involves considering every possible combination of rows from multiple tables. In SQL, a join is specified by a comma-separated list of input tables in the `FROM` clause of a `SELECT` statement.

For example, suppose we have a `meetings` table that records divisional meetings.

Division	Day	Time

Division	Day	Time
Accounting	Monday	9am
Computer	Wednesday	4pm
Administration	Monday	11am
Administration	Wednesday	4pm

The following statement joins the `records` table and the `meetings` table:

```
sqlite> SELECT * FROM records, meetings;
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Accounting|Monday|9am
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Computer|Wednesday|4pm
Alyssa P Hacker|Computer|Programmer|40000|Ben Bitdiddle|Administration|Monday|11am
...
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge|Administration|Monday|11am
Robert Cratchet|Accounting|Scrivener|18000|Eben Scrooge|Administration|Wednesday|4pm
```

There is one row in the joined table for each possible pair of a row from `records` and a row from `meetings`. Because `records` has 5 columns and `meetings` has 3 columns, the joined table has $5 + 3 = 8$ columns. The columns are named `Name`, `Division`, `Title`, `Salary`, `Supervisor`, `Division`, `Day`, `Time`. Because `records` has 9 rows and `meetings` has 4 rows, there are $9 * 4 = 36$ possible pairs of rows between the two tables; uncoincidentally, the joined table has 36 rows.

Sometimes, we join tables with overlapping column names. When this happens, we need to be able to disambiguate column names. The `AS` keyword gives an alias to a table listed in a `FROM` clause. Then we can use a dot expression to refer to a column in that table.

The following statement finds the name and title of Louis Reasoner's supervisor.

```
sqlite> SELECT b.name, b.title FROM records AS a, records AS b
...> WHERE a.name = "Louis Reasoner" AND
...> a.supervisor = b.name;
Alyssa P Hacker|Programmer
```

Aggregation

An aggregate function condenses information from multiple rows of a table into a single row. Some aggregate functions are `MAX`, `MIN`, `COUNT`, and `SUM`.

If we wanted to find the name and salary of the employee who makes the most money, we might type:

```
sqlite> SELECT name, MAX(salary) FROM records;
Lana Lambda|610000
```

The special syntax `COUNT(*)` counts the number of rows in a table. We can count the number of rows in `records` (which is the number of employees at our company).

```
sqlite> SELECT COUNT(*) FROM records;
9
```

The `GROUP BY [column name]` clause groups together all rows that have the same value in `[column name]`. Then aggregation is performed on each group so that there is exactly one row in the output table for each group.

The following statement finds the minimum salary earned in each division of our company.

```
sqlite> SELECT division, MIN(salary) FROM records GROUP BY division;
Computer|25000
Administration|25000
Accounting|18000
```

The `HAVING [condition]` clause filters the output table to include only the groups that satisfy the `[condition]`.

If we wanted to find all titles that are held by more than one person, we might type:

```
sqlite> SELECT title FROM records GROUP BY title HAVING COUNT(*) > 1;
Programmer
```