In [7]:

```
users = [[0, "Hero", 0],
         [1, "Dunn", 2],
         [2, "Sue", 3],
         [3, "Chi", 3]]
```

In [8]:

```
from typing import Tuple, Sequence, List, Any, Callable, Dict, Iterator
from collections import defaultdict
```

In [9]:

```
# A Few Type Aliases We'll Use Later
Row = Dict[str, Any]                            # A database row
WhereClause = Callable[[Row], bool]             # Predicate for a single row
HavingClause = Callable[[List[Row]], bool]  # Predicate over multiple rows
```

In [10]:

```
class Table:
    def __init__(self, columns: List[str], types: List[type]) -> None:
        assert len(columns) == len(types), "The number of columns must equal the number

        self.columns = columns          # Names of columns
        self.types = types              # Data types of columns
        self.rows: List[Row] = []       # (No data yet)

    def col2type(self, col: str) -> type:
        idx = self.columns.index(col)      # Finds the index of the column and returns
        return self.types[idx]

    def insert(self, values: list) -> None:
        # Check for the Right Number of Values
        if len(values) != len(self.types):
            raise ValueError(f"You need to provide {len(self.types)} values")

        # Check for the Right Types of Values
        for value, typ3 in zip(values, self.types):
            if not isinstance(value, typ3) and value is not None:
                raise TypeError(f"Expected type {typ3}, but got {value}")

        # Add the Corresponding Dictionary as a Row
        self.rows.append(dict(zip(self.columns, values)))

    def __getitem__(self, idx: int) -> Row:
        return self.rows[idx]

    def __iter__(self) -> Iterator[Row]:
        return iter(self.rows)

    def __len__(self) -> int:
        return len(self.rows)

    def __repr__(self):
        """Represents the table by columns then rows"""
        rows = "\n".join(str(row) for row in self.rows)

        return f"{self.columns}\n{rows}"

    def update(self,
               updates: Dict[str, Any],
               predicate: WhereClause = lambda row: True):
        # First, Make Sure the Updates Have Valid Names and Types
        for column, new_value in updates.items():
            if column not in self.columns:
                raise ValueError(f"invalid column: {column}")

            typ3 = self.col2type(column)
            if not isinstance(new_value, typ3) and new_value is not None:
                raise TypeError(f"expected type {typ3}, but got {new_value}")

        # Now Update
        for row in self.rows:
            if predicate(row):
                for column, new_value in updates.items():
                    row[column] = new_value

    def delete(self, predicate: WhereClause = lambda row: True) -> None:
        """Delete all rows matching predicate"""
        self.rows = [row for row in self.rows if not predicate(row)]

    def select(self,
               keep_columns: List[str] = None,
               additional_columns: Dict[str, Callable] = None) -> 'Table':

        if keep_columns is None:         # If no columns are specified, then return all
            keep_columns = self.columns

        if additional_columns is None:
            additional_columns = {}

        # New Column Names and Types
        new_columns = keep_columns + list(additional_columns.keys())
        keep_types = [self.col2type(col) for col in keep_columns]

        # This is How to Get the Return Type from a Type Annotation
        # It Will Crash if `calculation` Doesn't Have a Return Type
        add_types = [calculation.__annotations__['return']
                     for calculation in additional_columns.values()]

        # Create a New Table for Results
        new_table = Table(new_columns, keep_types + add_types)

        for row in self.rows:
            new_row = [row[column] for column in keep_columns]
            for column_name, calculation in additional_columns.items():
                new_row.append(calculation(row))
            new_table.insert(new_row)

        return new_table

    def where(self, predicate: WhereClause = lambda row: True) -> 'Table':
        """Return only the rows that satisfy the supplied predicate"""
        where_table = Table(self.columns, self.types)
        for row in self.rows:
            if predicate(row):
                values = [row[column] for column in self.columns]
                where_table.insert(values)
        return where_table
    def limit(self, num_rows: int) -> 'Table':
        """Return only the first `num_rows` rows"""
        limit_table = Table(self.columns, self.types)
        for i, row in enumerate(self.rows):
            if i >= num_rows:
                break
            values = [row[column] for column in self.columns]
            limit_table.insert(values)
        return limit_table

    def group_by(self,
                 group_by_columns: List[str],
                 aggregates: Dict[str, Callable],
                 having: HavingClause = lambda group: True) -> 'Table':

        grouped_rows = defaultdict(list)

        # Populate Groups
        for row in self.rows:
            key = tuple(row[column] for column in group_by_columns)
            grouped_rows[key].append(row)

        # Result Table Consists of group_by Columns and Aggregates
        new_columns = group_by_columns + list(aggregates.keys())
        group_by_types = [self.col2type(col) for col in group_by_columns]
        aggregate_types = [agg.__annotations__['return']
                           for agg in aggregates.values()]
        result_table = Table(new_columns, group_by_types + aggregate_types)

        for key, rows in grouped_rows.items():
            if having(rows):
                new_row = list(key)
                for aggregate_name, aggregate_fn in aggregates.items():
                    new_row.append(aggregate_fn(rows))
                result_table.insert(new_row)

        return result_table

    def order_by(self, order: Callable[[Row], Any]) -> 'Table':
        new_table = self.select()       # Make a copy
        new_table.rows.sort(key = order)
        return new_table

    def join(self, other_table: 'Table', left_join: bool = False) -> 'Table':

        join_on_columns = [c for c in self.columns           # Columns in both tables
                           if c in other_table.columns]

        additional_columns = [c for c in other_table.columns # Columns only in right ta
                              if c not in join_on_columns]

        # All Columns from the Left Table Plus Additional Columns from the Right Table
        new_columns = self.columns + additional_columns
        new_types = self.types + [other_table.col2type(col)
                                  for col in additional_columns]

        join_table = Table(new_columns, new_types)

        for row in self.rows:
            def is_join(other_row):
                return all(other_row[c] == row[c] for c in join_on_columns)

            other_rows = other_table.where(is_join).rows

            # Each Other Row That Matches This One Produces a Result Row
            for other_row in other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [other_row[c] for c in additional_columns])

            # If no Rows Match and It's a Left Join, Then Output with Nones
            if left_join and not other_rows:
                join_table.insert([row[c] for c in self.columns] +
                                  [None for c in additional_columns])

        return join_table
```

In [11]:

```
def main():
    # Constructor Requires Column Names and Types
    users = Table(['user_id', 'name', 'num_friends'], [int, str, int])
    users.insert([0, "Hero", 0])
    users.insert([1, "Dunn", 2])
    users.insert([2, "Sue", 3])
    users.insert([3, "Chi", 3])
    users.insert([4, "Thor", 3])
    users.insert([5, "Clive", 2])
    users.insert([6, "Hicks", 3])
    users.insert([7, "Devin", 2])
    users.insert([8, "Kate", 2])
    users.insert([9, "Klein", 3])
    users.insert([10, "Jen", 1])

    assert len(users) == 11
    assert users[1]['name'] == 'Dunn'

    assert users[1]['num_friends'] == 2             # Original value

    users.update({'num_friends' : 3},               # Set num_friends = 3 in rows where
                 lambda row: row['user_id'] == 1)

    assert users[1]['num_friends'] == 3             # Updated value

    # SELECT * FROM users;
    all_users = users.select()
    assert len(all_users) == 11

    # SELECT * FROM users LIMIT 2;
    two_users = users.limit(2)
    assert len(two_users) == 2

    # SELECT user_id FROM users;
    just_ids = users.select(keep_columns = ["user_id"])
    assert just_ids.columns == ['user_id']

    # SELECT user_id FROM users WHERE name = 'Dunn';
    dunn_ids = (
        users
        .where(lambda row: row["name"] == "Dunn")
        .select(keep_columns = ["user_id"])
    )
```

```python
assert len(dunn_ids) == 1
assert dunn_ids[0] == {"user_id": 1}

# SELECT LENGTH(name) AS name_length FROM users;
def name_length(row) -> int: return len(row["name"])

name_lengths = users.select(keep_columns = [],
                            additional_columns = {"name_length": name_length})
assert name_lengths[0]['name_length'] == len("Hero")

def min_user_id(rows) -> int:
    return min(row["user_id"] for row in rows)

def length(rows) -> int:
    return len(rows)

stats_by_length = (
    users
    .select(additional_columns = {"name_length" : name_length})
    .group_by(group_by_columns = ["name_length"],
              aggregates={"min_user_id" : min_user_id,
                          "num_users" : length})
)



assert len(stats_by_length) == 3
assert stats_by_length.columns == ["name_length", "min_user_id", "num_users"]

def first_letter_of_name(row: Row) -> str:
    return row["name"][0] if row["name"] else ""

def average_num_friends(rows: List[Row]) -> float:
    return sum(row["num_friends"] for row in rows) / len(rows)

def enough_friends(rows: List[Row]) -> bool:
    return average_num_friends(rows) > 1

avg_friends_by_letter = (
    users
    .select(additional_columns = {'first_letter' : first_letter_of_name})
    .group_by(group_by_columns = ['first_letter'],
              aggregates = {"avg_num_friends" : average_num_friends},
              having = enough_friends)
)


assert len(avg_friends_by_letter) == 6
assert {row['first_letter'] for row in avg_friends_by_letter} == \
       {"H", "D", "S", "C", "T", "K"}

def sum_user_ids(rows: List[Row]) -> int:
    return sum(row["user_id"] for row in rows)

user_id_sum = (
    users
    .where(lambda row: row["user_id"] > 1)
    .group_by(group_by_columns = [],
              aggregates = {"user_id_sum" : sum_user_ids})
)
```

```
assert len(user_id_sum) == 1
assert user_id_sum[0]["user_id_sum"] == 54

friendliest_letters = (
    avg_friends_by_letter
    .order_by(lambda row: -row["avg_num_friends"])
    .limit(4)
)


assert len(friendliest_letters) == 4
assert friendliest_letters[0]['first_letter'] in ['S', 'T']

user_interests = Table(['user_id', 'interest'], [int, str])
user_interests.insert([0, "SQL"])
user_interests.insert([0, "NoSQL"])
user_interests.insert([2, "SQL"])
user_interests.insert([2, "MySQL"])

sql_users = (
    users
    .join(user_interests)
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns = ["name"])
)



assert len(sql_users) == 2
sql_user_names = {row["name"] for row in sql_users}
assert sql_user_names == {"Hero", "Sue"}

def count_interests(rows: List[Row]) -> int:
    """Counts how many rows have non-None interests"""
    return len([row for row in rows if row["interest"] is not None])

user_interest_counts = (
    users
    .join(user_interests, left_join = True)
    .group_by(group_by_columns = ["user_id"],
              aggregates = {"num_interests" : count_interests })
)

likes_sql_user_ids = (
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .select(keep_columns = ['user_id'])
)

likes_sql_user_ids.group_by(group_by_columns = [],
                            aggregates = {"min_user_id" : min_user_id})



assert len(likes_sql_user_ids) == 2

(
    user_interests
    .where(lambda row: row["interest"] == "SQL")
    .join(users)
    .select(["name"])
)

(
    user_interests
    .join(users)
    .where(lambda row: row["interest"] == "SQL")
    .select(["name"])
)
```

In [12]:

```
if __name__ == "__main__": main()
```