**Module-5**

**Finding Candidate Keys and Closure of attribute sets:**

**Basic Information:**Implementing the algorithm for closure of attributes and identifying candidate keys involves understanding the relationships expressed by functional dependencies in a relation. The closure of attributes, denoted as X⁺, captures all attributes functionally determined by a given set X. To find it, iterate through the functional dependencies, adding attributes determined by X until no more can be added. Candidate keys are minimal superkeys (sets of attributes that uniquely identify all others). By systematically combining attributes and checking for minimality, we can identify candidate keys.

1. **Understanding Functional Dependencies (FDs)**: Given a set of functional dependencies, understand that it expresses the relationships between attributes in a relation. For example, if we have an FD A -> B, it means that for every unique value of A, there is a unique value of B associated with it.
2. **Closure of Attributes**: The closure of attributes of a set of attributes X, denoted as X⁺, is the set of all attributes that are functionally determined by X. To compute the closure of attributes:
   * Start with the given set of attributes X.
   * Check each functional dependency to see if the attributes on the left side are contained in X. If they are, add the attributes on the right side to X.
   * Repeat this process until no new attributes can be added to X. The resulting set is the closure of X.
3. **Identifying Candidate Keys**:
   * To identify candidate keys, we need to find the minimal superkeys.
   * Start with each attribute individually and find its closure.
   * Combine attributes and find their closure until no additional attributes can be added without changing the closure.
   * The minimal superkeys are the smallest sets of attributes that uniquely identify all other attributes in the relation.
   * To identify candidate keys, check if removing any attribute from a minimal superkey results in a set that is no longer a superkey. If so, the original minimal superkey is a candidate key.
4. **Algorithm Implementation**:
   * Implement functions to compute the closure of attributes given a set of functional dependencies.
   * Implement a function to generate all possible combinations of attributes to find the minimal superkeys.
   * Implement a function to check if a minimal superkey is a candidate key by testing its minimality.

**Algorithm Used:**

function find\_closure(attributes, fds)

closure = attributes

while changes

changes = false

for each fd in fds

if fd.lhs is subset of closure and fd.rhs is not subset of closure

closure = closure union fd.rhs

changes = true

return closure

function find\_candidate\_keys(attributes, fds)

keys = []

for each subset of attributes as key\_candidate

if find\_closure(key\_candidate, fds) == all\_attributes

keys.add(key\_candidate)

minimal\_keys = []

for each key in keys

if no other key is subset of key

minimal\_keys.add(key)

return minimal\_keys

**Code:**

from itertools import combinations

class FunctionalDependency:

    def \_\_init\_\_(self, lhs, rhs):

        self.lhs = lhs

        self.rhs = rhs

    def \_\_repr\_\_(self):

        return f"{','.join(self.lhs)} -> {','.join(self.rhs)}"

def find\_closure(attributes, fds):

    closure = set(attributes)

    changed = True

    while changed:

        changed = False

        for fd in fds:

            if all(attr in closure for attr in fd.lhs) and not all(attr in closure for attr in fd.rhs):

                closure.update(fd.rhs)

                changed = True

    return closure

def find\_candidate\_keys(attributes, fds):

    candidate\_keys = []

    all\_attributes = set(attributes)

    for i in range(1, len(attributes) + 1):

        for combo in combinations(attributes, i):

            closure = find\_closure(combo, fds)

            if closure == all\_attributes:

                candidate\_keys.append(combo)

    minimal\_candidate\_keys = []

    for key in candidate\_keys:

        is\_minimal = True

        for other\_key in candidate\_keys:

            if key != other\_key and set(key).issubset(set(other\_key)):

                is\_minimal = False

                break

        if is\_minimal:

            minimal\_candidate\_keys.append(key)

    return minimal\_candidate\_keys

attributes = ['A', 'B', 'C', 'D', 'E', 'F']

functional\_dependencies = [

    FunctionalDependency(['A'], ['B']),

    FunctionalDependency(['B'], ['C', 'D']),

    FunctionalDependency(['D'], ['A']),

    FunctionalDependency(['C', 'E'], ['F']),

    FunctionalDependency(['F'], ['B'])

]

print("\nList of Candidate Keys is :")

candidate\_keys = find\_candidate\_keys(attributes, functional\_dependencies)

for key in candidate\_keys:

    print(','.join(key))

print("\nList of Closure of Attributes is :")

for attr in attributes:

    closure = find\_closure([attr], functional\_dependencies)

    print(f"{attr}+ = {','.join(closure)}")

**Output:**

**A computer screen shot of white text

Description automatically generated**

**Boyce Codd Normal Form:**

**Basic Information:**Boyce-Codd Normal Form (BCNF) is a refinement of the more commonly known Third Normal Form (3NF) in database normalization. It deals with the issue of functional dependencies within a relation. A relation is in BCNF if and only if every determinant (attribute or set of attributes that uniquely determines other attributes) is a candidate key. In simpler terms, BCNF ensures that there are no non-trivial functional dependencies where a non-prime attribute (not part of any candidate key) determines another non-prime attribute. Achieving BCNF minimizes redundancy and dependency issues in a database schema, enhancing data integrity and efficiency. However, not all relations can be decomposed into BCNF without loss of information, requiring careful consideration during the normalization process.

**Algorithm Used:**

Function compute\_closure(attributes, fds):

closure = set(attributes)

changed = True

while changed:

changed = False

for each functional dependency (X -> Y) in fds:

if X is subset of closure and Y is not subset of closure:

closure = closure union Y

changed = True

return closure

Function find\_candidate\_keys(attributes, fds):

candidate\_keys = []

for each subset of attributes in the power set of attributes:

if compute\_closure(subset, fds) is equal to attributes:

candidate\_keys.append(subset)

return candidate\_keys

Function decompose\_to\_bcnf(attributes, fds):

bcnf\_relations = []

candidate\_keys = find\_candidate\_keys(attributes, fds)

for each candidate key ck in candidate\_keys:

remaining\_fds = fds copy

for each functional dependency (X -> Y) in fds:

if X is subset of ck:

remove (X -> Y) from remaining\_fds

add (ck, compute\_closure(ck, fds), remaining\_fds) to bcnf\_relations

return bcnf\_relations

**Code:**

import itertools

def compute\_closure(attributes, fds):

    closure = set(attributes)

    changed = True

    while changed:

        changed = False

        for fd in fds:

            if fd[0].issubset(closure) and not fd[1].issubset(closure):

                closure |= fd[1]

                changed = True

    return closure

def is\_superkey(attributes, fds, candidate\_key):

    return compute\_closure(candidate\_key, fds) == set(attributes)

def find\_candidate\_keys(attributes, fds):

    candidate\_keys = []

    for i in range(1, len(attributes) + 1):

        for subset in itertools.combinations(attributes, i):

            if is\_superkey(attributes, fds, set(subset)):

                candidate\_keys.append(set(subset))

    return candidate\_keys

def decompose\_to\_bcnf(attributes, fds):

    bcnf\_relations = []

    candidate\_keys = find\_candidate\_keys(attributes, fds)

    for ck in candidate\_keys:

        remaining\_fds = fds.copy()

        for fd in fds:

            if fd[0].issubset(ck):

                remaining\_fds.remove(fd)

        bcnf\_relations.append((ck, compute\_closure(ck, fds), remaining\_fds))

    return bcnf\_relations

attributes = ['A', 'B', 'C', 'D']

fds = [({'A'}, {'B', 'C'}), ({'B'}, {'D'})]

bcnf\_relations = decompose\_to\_bcnf(attributes, fds)

for relation in bcnf\_relations:

    print("Relation Number:", bcnf\_relations.index(relation) + 1)

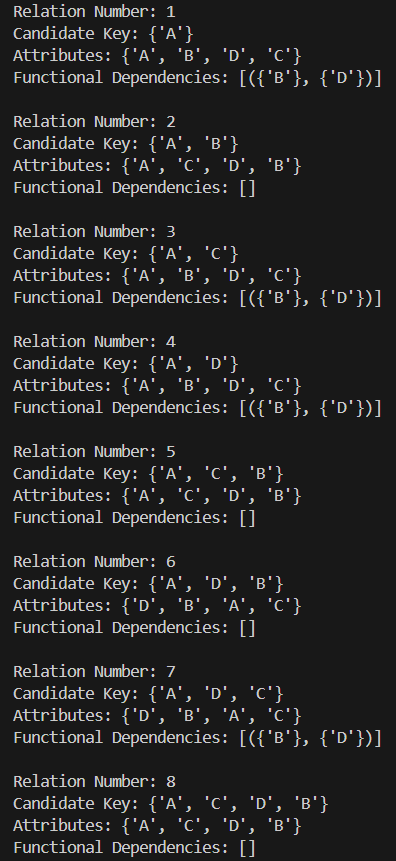
    print("Candidate Key:", relation[0])

    print("Attributes:", relation[1])

    print("Functional Dependencies:", relation[2])

    print()

**Output:**

Top of Form