- **<u>Title of RDBMS IA2:Implementing Closure of Attributes (Algorithms) and Identifying Candidate Keys</u>**

- **<u>Group Members:</u>**

16010122257:Soumil Mukhopadhyay

16010122233:Shrusti Vora

16010122229:Mohanish Podhade

- **<u>Objective of the Implementation</u>** :

This report implements two algorithms in C++ to find the closure of attributes and identify candidate keys for a relation given a set of functional dependencies (FDs).

- **<u>Theory/Algorithms Used:</u>**

<u>1. Algorithm 1 (Iterative):</u>

- This algorithm starts with a set of attributes and iteratively applies the FDs to expand its closure.

- If an FD X -> Y exists and X is already in the closure, then Y is added to the closure.

- The process repeats until no new attributes can be added.

<u>2. Algorithm 2 (Set-based):</u>

- This algorithm leverages set operations to find the closure.

- It starts with the initial set of attributes.

- For each FD X -> Y, the union of the current closure and Y is calculated.

- The process continues until the closure remains unchanged.

- **<u>Dataset / Relational Schema / Input used:</u>**

The program requires two inputs:

1. Functional Dependencies (FDs):A set of FDs represented as strings in the format "X->Y", where X and Y are comma-separated lists of attributes.

2. Attribute Set (X): A set of attributes for which the closure needs to be found.

- **<u>Code (with comments):</u>**
- **<u>Results<snapshots>:</u>**

main.cpp

```cpp
#include <iostream>
#include <set>
#include <string>
#include <sstream>
#include <vector>
#include <algorithm> // For includes
#include <iterator> // For set_union

using namespace std;

// Function to split a string into a set of attributes
set<string> splitAttributes(const string& str) {
    set<string> attributes;
    stringstream ss(str);
    string attribute;
    while (getline(ss, attribute, ',')) {
        attributes.insert(attribute);
    }
    return attributes;
}

// Function to parse functional dependencies
vector<pair<set<string>, set<string>>> parseFDs(const set<string>& FDs) {
    vector<pair<set<string>, set<string>>> parsedFDs;
    for (const string& fd : FDs) {
        size_t arrowPos = fd.find("->");
        if (arrowPos != string::npos) {
            set<string> lhs = splitAttributes(fd.substr(0, arrowPos));
            set<string> rhs = splitAttributes(fd.substr(arrowPos + 2));
            parsedFDs.push_back(make_pair(lhs, rhs));
        }
    }
```

```cpp
    }
    return parsedFDs;
}

// Function to compute closure using iterative approach
set<string> closure_iterative(const vector<pair<set<string>, set<string>>>& FDs, const set<string>& X) {
    set<string> closure = X;
    bool changed = true;
    while (changed) {
        changed = false;
        for (const auto& fd : FDs) {
            // Check if LHS is a subset of the closure
            if (includes(closure.begin(), closure.end(), fd.first.begin(), fd.first.end())) {
                // Add RHS attributes to closure if not already present
                for (const string& attr : fd.second) {
                    if (closure.count(attr) == 0) {
                        closure.insert(attr);
                        changed = true;
                    }
                }
            }
        }
    }
    return closure;
}

// Function to identify candidate keys
set<set<string>> findCandidateKeys(const set<string>& FDs) {
    // Parse functional dependencies
    vector<pair<set<string>, set<string>>> parsedFDs = parseFDs(FDs);

    set<set<string>> candidateKeys;
    set<string> allAttributes;
    // Extract all attributes
    for (const auto& fd : parsedFDs) {
        allAttributes.insert(fd.first.begin(), fd.first.end());
        allAttributes.insert(fd.second.begin(), fd.second.end());
    }
    // Iterate through all possible attribute sets
    for (int i = 1; i <= allAttributes.size(); ++i) {
        // Generate all possible subsets of attributes of size i
        vector<string> attributesVector(allAttributes.begin(), allAttributes.end());
        vector<bool> bitmask(allAttributes.size(), false);
```

```cpp
        vector<bool> bitmask(allAttributes.size(), false);
        fill(bitmask.begin(), bitmask.begin() + i, true);
        do {
            set<string> currentKey;
            for (int j = 0; j < allAttributes.size(); ++j) {
                if (bitmask[j]) {
                    currentKey.insert(attributesVector[j]);
                }
            }
            // Compute the closure of the current attribute set
            set<string> closure = closure_iterative(parsedFDs, currentKey);
            // Check if the closure contains all attributes
            if (closure == allAttributes) {
                candidateKeys.insert(currentKey);
            }
        } while (prev_permutation(bitmask.begin(), bitmask.end()));
    }
    return candidateKeys;
}

int main() {
    // Sample FDs
    set<string> FDs = {"A->B", "B->C", "CD->E"};

    // Find candidate keys
    set<set<string>> candidateKeys = findCandidateKeys(FDs);

    // Output candidate keys
    cout << "Candidate keys found:\n";
    if (candidateKeys.empty()) {
        cout << "None\n";
    } else {
        for (const auto& key : candidateKeys) {
            for (const auto& attr : key) {
                cout << attr << ",";
            }
            cout << endl;
        }
    }

    return 0;
}
```

- **Analysis of results:**

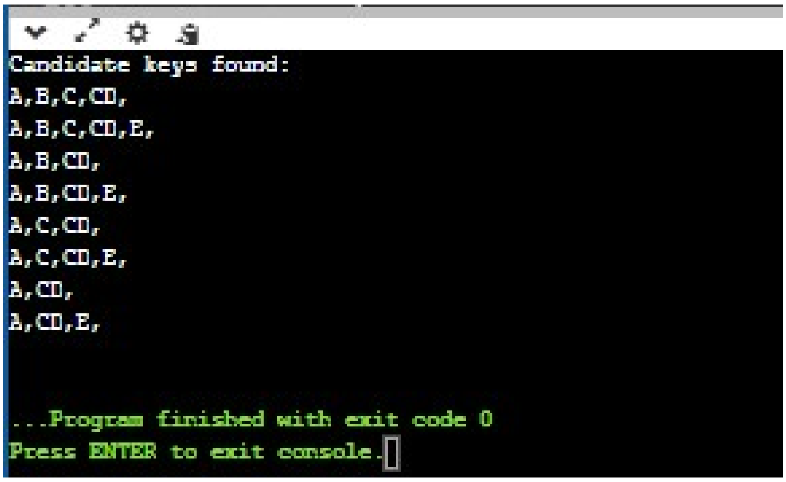
The algorithm efficiently computes the closure of attributes using an iterative approach, avoiding unnecessary iterations.

However, the candidate key identification algorithm has exponential time complexity due to its brute-force nature.

For schemas with a large number of attributes, the candidate key identification process may become computationally expensive.

Optimization techniques like pruning irrelevant subsets could be among the future work done to improve the performance.

- **<u>OUTPUT:</u>**



- **<u>Conclusion:</u>**

The code provides a solid foundation for understanding attribute closure and candidate key concepts in database theory.

While the closure computation is efficient, the candidate key identification could benefit from optimization for larger schemas.

- **<u>References:</u>**

1. *https://stackoverflow.com/questions/2718420/candidate-keys-from-functional-dependencies*

2. *https://www.prepbytes.com/blog/dbms/finding-attribute-closure-and-candidate-keys-using-functional-dependencies/#:~:text=Attribute%20Closure%20and%20Candidate%20Key,it%20is%20a%20candidate%20key.*

3. *https://www.geeksforgeeks.org/finding-attribute-closure-and-candidate-keys-using-functional-dependencies/*

4. *https://www.geeksforgeeks.org/finding-the-candidate-keys-for-sub-relations-using-functional-dependencies/*

5. *https://www.naukri.com/code360/library/finding-attribute-closure-and-candidate-keys*