CSLR51 – Database Management Systems Laboratory
#Session: 09 || Date: 10/10/2024
Viva Due: Q.No. 1a (10/10/2024) Moodle Due: 16/10/2024 at 11 PM

Develop C code to simulate the following task:
1. Normalization of a Relational Schema Develop an implementation package that decomposes a given relational schema into the following Normal Forms. Your package may support interfaces for user input candidate key.
a. 2 nd Normal Form.
b. 3 rd Normal Form.
c. Boyce-Codd Normal Form.

a.

```
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <unordered_map>
#include <algorithm>

class FunctionalDependency {
public:
    std::set<std::string> determinant;
    std::set<std::string> dependent;

    FunctionalDependency(std::set<std::string> det, std::set<std::string> dep)
        : determinant(det), dependent(dep) {}
};

class Relation {
public:
    std::string name;
    std::set<std::string> attributes;
    std::vector<FunctionalDependency> fds;

    Relation(std::string n, std::set<std::string> attrs)
        : name(n), attributes(attrs) {}

    void addFunctionalDependency(const FunctionalDependency& fd) {
        fds.push_back(fd);
    }

    void display() const {
```

```
        std::cout << "Relation: " << name << "\nAttributes: ";
        for (const auto& attr : attributes) {
            std::cout << attr << " ";
        }
        std::cout << "\nFunctional Dependencies:\n";
        for (const auto& fd : fds) {
            std::cout << "  " << "{ ";
            for (const auto& attr : fd.determinant) {
                std::cout << attr << " ";
            }
            std::cout << "} -> { ";
            for (const auto& attr : fd.dependent) {
                std::cout << attr << " ";
            }
            std::cout << "}\n";
        }
    }
};

class Decomposer {
public:
    static std::vector<Relation> decomposeTo2NF(const Relation& relation, const
std::set<std::set<std::string>>& candidateKeys) {
        std::vector<Relation> decomposedRelations;

        // Step 1: Identify partial dependencies
        std::set<std::string> nonKeyAttributes = relation.attributes;
        for (const auto& key : candidateKeys) {
            for (const auto& attr : key) {
                nonKeyAttributes.erase(attr);
            }
        }

        // Step 2: Decompose based on functional dependencies
        for (const auto& fd : relation.fds) {
            // Check for partial dependency
            if (isPartialDependency(fd, candidateKeys)) {
                std::set<std::string> newRelationAttrs = fd.determinant;
                newRelationAttrs.insert(fd.dependent.begin(), fd.dependent.end());
                Relation newRelation("Decomposed_" + relation.name, newRelationAttrs);
                newRelation.addFunctionalDependency(fd);
                decomposedRelations.push_back(newRelation);
            }
        }
```

```
        // Add original relation if it has no partial dependencies
        if (decomposedRelations.empty()) {
            decomposedRelations.push_back(relation);
        }

        return decomposedRelations;
    }

private:
    static bool isPartialDependency(const FunctionalDependency& fd, const
std::set<std::set<std::string>>& candidateKeys) {
        // Check if fd is partially dependent on any candidate key
        for (const auto& key : candidateKeys) {
            if (key.size() > fd.determinant.size() &&
                std::all_of(fd.determinant.begin(), fd.determinant.end(),
                            [&key](const std::string& attr) {
                                return key.find(attr) != key.end();
                            })) {
                return true; // Partial dependency found
            }
        }
        return false;
    }

};

int main() {
    // Example usage
    Relation relation("StudentCourse", {"StudentID", "CourseID", "Instructor", "CourseName"});

    // Adding functional dependencies
    relation.addFunctionalDependency(FunctionalDependency({"StudentID"}, {"CourseID",
"Instructor"}));
    relation.addFunctionalDependency(FunctionalDependency({"CourseID"}, {"CourseName"}));

    // Input candidate keys
    std::set<std::set<std::string>> candidateKeys = { {"StudentID", "CourseID"} };

    // Display the original relation
    relation.display();

    // Decompose to 2NF
```

```
    std::vector<Relation> decomposedRelations = Decomposer::decomposeTo2NF(relation,
candidateKeys);

    // Display decomposed relations
    std::cout << "\nDecomposed Relations:\n";
    for (const auto& rel : decomposedRelations) {
        rel.display();
    }

    return 0;
}
```

OUTPUT:
2 NF:

```
reethi@DESKTOP-8744EFO:~/dir1/dbms$ ./a.out
Relation: StudentCourse
Attributes: CourseID CourseName Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
  { CourseID } -> { CourseName }

Decomposed Relations:
Relation: Decomposed_StudentCourse
Attributes: CourseID Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
Relation: Decomposed_StudentCourse
Attributes: CourseID CourseName
Functional Dependencies:
  { CourseID } -> { CourseName }
```

b.

```
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <unordered_map>
#include <algorithm>

class FunctionalDependency {
public:
    std::set<std::string> determinant;
```

```
    std::set<std::string> dependent;

    FunctionalDependency(std::set<std::string> det, std::set<std::string> dep)
        : determinant(det), dependent(dep) {}
};

class Relation {
public:
    std::string name;
    std::set<std::string> attributes;
    std::vector<FunctionalDependency> fds;

    Relation(std::string n, std::set<std::string> attrs)
        : name(n), attributes(attrs) {}

    void addFunctionalDependency(const FunctionalDependency& fd) {
        fds.push_back(fd);
    }

    void display() const {
        std::cout << "Relation: " << name << "\nAttributes: ";
        for (const auto& attr : attributes) {
            std::cout << attr << " ";
        }
        std::cout << "\nFunctional Dependencies:\n";
        for (const auto& fd : fds) {
            std::cout << "  { ";
            for (const auto& attr : fd.determinant) {
                std::cout << attr << " ";
            }
            std::cout << "} -> { ";
            for (const auto& attr : fd.dependent) {
                std::cout << attr << " ";
            }
            std::cout << "}\n";
        }
    }
};

class Decomposer {
public:
    static std::vector<Relation> decomposeTo3NF(const Relation& relation, const
std::set<std::set<std::string>>& candidateKeys) {
        std::vector<Relation> decomposedRelations;
```

```
        std::set<std::string> processedAttributes;

        // Step 1: For each functional dependency, create a new relation
        for (const auto& fd : relation.fds) {
            std::set<std::string> relationAttrs = fd.determinant;
            relationAttrs.insert(fd.dependent.begin(), fd.dependent.end());
            Relation newRelation("Decomposed_" + relation.name, relationAttrs);
            newRelation.addFunctionalDependency(fd);
            decomposedRelations.push_back(newRelation);
            processedAttributes.insert(relationAttrs.begin(), relationAttrs.end());
        }

        // Step 2: Identify any attributes not included in the decomposed relations
        std::set<std::string> remainingAttributes;
        for (const auto& attr : relation.attributes) {
            if (processedAttributes.find(attr) == processedAttributes.end()) {
                remainingAttributes.insert(attr);
            }
        }

        // Step 3: If there are any remaining attributes, create a new relation
        if (!remainingAttributes.empty()) {
            Relation remainingRelation("Decomposed_" + relation.name + "_Remaining",
remainingAttributes);
            decomposedRelations.push_back(remainingRelation);
        }

        return decomposedRelations;
    }
};

int main() {
    std::string relationName;
    std::set<std::string> attributes;
    int numAttributes;

    // Input relation name and attributes
    std::cout << "Enter relation name: ";
    std::cin >> relationName;
    std::cout << "Enter number of attributes: ";
    std::cin >> numAttributes;

    std::cout << "Enter attributes (space-separated): ";
    for (int i = 0; i < numAttributes; ++i) {
```

```
    std::string attr;
    std::cin >> attr;
    attributes.insert(attr);
}

Relation relation(relationName, attributes);

// Input functional dependencies
int numFDs;
std::cout << "Enter number of functional dependencies: ";
std::cin >> numFDs;

for (int i = 0; i < numFDs; ++i) {
    std::set<std::string> determinant, dependent;
    int numDeterminants, numDependents;

    std::cout << "Enter number of determinants for FD " << (i + 1) << ": ";
    std::cin >> numDeterminants;

    std::cout << "Enter determinants (space-separated): ";
    for (int j = 0; j < numDeterminants; ++j) {
        std::string attr;
        std::cin >> attr;
        determinant.insert(attr);
    }

    std::cout << "Enter number of dependents for FD " << (i + 1) << ": ";
    std::cin >> numDependents;

    std::cout << "Enter dependents (space-separated): ";
    for (int j = 0; j < numDependents; ++j) {
        std::string attr;
        std::cin >> attr;
        dependent.insert(attr);
    }

    relation.addFunctionalDependency(FunctionalDependency(determinant, dependent));
}

// Input candidate keys
int numKeys;
std::cout << "Enter number of candidate keys: ";
std::cin >> numKeys;
```

```
    std::set<std::set<std::string>> candidateKeys;
    for (int i = 0; i < numKeys; ++i) {
        std::set<std::string> key;
        int numKeyAttributes;

        std::cout << "Enter number of attributes in candidate key " << (i + 1) << ": ";
        std::cin >> numKeyAttributes;

        std::cout << "Enter attributes for candidate key " << (i + 1) << " (space-separated): ";
        for (int j = 0; j < numKeyAttributes; ++j) {
            std::string attr;
            std::cin >> attr;
            key.insert(attr);
        }

        candidateKeys.insert(key);
    }

    // Display the original relation
    relation.display();

    // Decompose to 3NF
    std::vector<Relation> decomposedRelations = Decomposer::decomposeTo3NF(relation,
candidateKeys);

    // Display decomposed relations
    std::cout << "\nDecomposed Relations:\n";
    for (const auto& rel : decomposedRelations) {
        rel.display();
    }

    return 0;
}
```

OUTPUT:
3 NF:

```
reethi@DESKTOP-8744EFO:~/dir1/dbms$ ./a.out
Enter relation name: StudentCourse
Enter number of attributes: 4
Enter attributes (space-separated): StudentID CourseID Instructor CourseName
Enter number of functional dependencies: 2
Enter number of determinants for FD 1: 1
Enter determinants (space-separated): StudentID
```

```
Enter number of dependents for FD 1: 2
Enter dependents (space-separated): CourseID Instructor
Enter number of determinants for FD 2: 1
Enter determinants (space-separated): CourseID
Enter number of dependents for FD 2: 1
Enter dependents (space-separated): CourseName
Enter number of candidate keys: 1
Enter number of attributes in candidate key 1: 2
Enter attributes for candidate key 1 (space-separated): StudentID CourseID
Relation: StudentCourse
Attributes: CourseID CourseName Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
  { CourseID } -> { CourseName }

Decomposed Relations:
Relation: Decomposed_StudentCourse
Attributes: CourseID Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
Relation: Decomposed_StudentCourse
Attributes: CourseID CourseName
Functional Dependencies:
  { CourseID } -> { CourseName }
```

c.

```
#include <iostream>
#include <vector>
#include <set>
#include <string>
#include <unordered_set>
#include <unordered_map>
#include <algorithm>

class FunctionalDependency {
public:
    std::set<std::string> determinant;
    std::set<std::string> dependent;

    FunctionalDependency(std::set<std::string> det, std::set<std::string> dep)
        : determinant(det), dependent(dep) {}
};

class Relation {
```

```
public:
    std::string name;
    std::set<std::string> attributes;
    std::vector<FunctionalDependency> fds;

    Relation(std::string n, std::set<std::string> attrs)
        : name(n), attributes(attrs) {}

    void addFunctionalDependency(const FunctionalDependency& fd) {
        fds.push_back(fd);
    }

    void display() const {
        std::cout << "Relation: " << name << "\nAttributes: ";
        for (const auto& attr : attributes) {
            std::cout << attr << " ";
        }
        std::cout << "\nFunctional Dependencies:\n";
        for (const auto& fd : fds) {
            std::cout << "  { ";
            for (const auto& attr : fd.determinant) {
                std::cout << attr << " ";
            }
            std::cout << "} -> { ";
            for (const auto& attr : fd.dependent) {
                std::cout << attr << " ";
            }
            std::cout << "}\n";
        }
    }
};

class Decomposer {
public:
    static std::vector<Relation> decomposeToBCNF(const Relation& relation, const
std::set<std::set<std::string>>& candidateKeys) {
        std::vector<Relation> decomposedRelations;
        std::vector<Relation> workList = { relation };

        while (!workList.empty()) {
            Relation currentRelation = workList.back();
            workList.pop_back();

            // Check if the current relation is in BCNF
```

```
        bool isBCNF = true;
        for (const auto& fd : currentRelation.fds) {
            if (!isSuperKey(fd.determinant, currentRelation, candidateKeys)) {
                isBCNF = false;
                break;
            }
        }

        if (isBCNF) {
            decomposedRelations.push_back(currentRelation);
        } else {
            // Decompose the relation
            for (const auto& fd : currentRelation.fds) {
                if (!isSuperKey(fd.determinant, currentRelation, candidateKeys)) {
                    // Create new relations based on the functional dependency
                    std::set<std::string> newRelationAttrs = fd.determinant;
                    newRelationAttrs.insert(fd.dependent.begin(), fd.dependent.end());
                    Relation newRelation("Decomposed_" + currentRelation.name,
newRelationAttrs);
                    newRelation.addFunctionalDependency(fd);
                    decomposedRelations.push_back(newRelation);

                    // Create relation for the remaining attributes
                    std::set<std::string> remainingAttrs;
                    for (const auto& attr : currentRelation.attributes) {
                        if (newRelationAttrs.find(attr) == newRelationAttrs.end()) {
                            remainingAttrs.insert(attr);
                        }
                    }
                    if (!remainingAttrs.empty()) {
                        Relation remainingRelation("Remaining_" + currentRelation.name,
remainingAttrs);
                        // Transfer non-redundant functional dependencies
                        for (const auto& existingFD : currentRelation.fds) {
                            if (newRelationAttrs.find(*existingFD.determinant.begin()) ==
newRelationAttrs.end()) {
                                remainingRelation.addFunctionalDependency(existingFD);
                            }
                        }
                        workList.push_back(remainingRelation);
                    }
                    break; // Process one FD at a time
                }
            }
```

```
            }
        }

        return decomposedRelations;
    }

private:
    static bool isSuperKey(const std::set<std::string>& determinant, const Relation& relation,
const std::set<std::set<std::string>>& candidateKeys) {
        std::set<std::string> closure = computeClosure(determinant, relation);
        return closure == relation.attributes;
    }

    static std::set<std::string> computeClosure(const std::set<std::string>& attrs, const Relation&
relation) {
        std::set<std::string> closure = attrs;
        bool added;
        do {
            added = false;
            for (const auto& fd : relation.fds) {
                if (std::includes(closure.begin(), closure.end(), fd.determinant.begin(),
fd.determinant.end())) {
                    for (const auto& attr : fd.dependent) {
                        if (closure.insert(attr).second) {
                            added = true;
                        }
                    }
                }
            }
        } while (added);
        return closure;
    }
};

int main() {
    std::string relationName;
    std::set<std::string> attributes;
    int numAttributes;

    // Input relation name and attributes
    std::cout << "Enter relation name: ";
    std::cin >> relationName;
    std::cout << "Enter number of attributes: ";
    std::cin >> numAttributes;
```

```
std::cout << "Enter attributes (space-separated): ";
for (int i = 0; i < numAttributes; ++i) {
    std::string attr;
    std::cin >> attr;
    attributes.insert(attr);
}

Relation relation(relationName, attributes);

// Input functional dependencies
int numFDs;
std::cout << "Enter number of functional dependencies: ";
std::cin >> numFDs;

for (int i = 0; i < numFDs; ++i) {
    std::set<std::string> determinant, dependent;
    int numDeterminants, numDependents;

    std::cout << "Enter number of determinants for FD " << (i + 1) << ": ";
    std::cin >> numDeterminants;

    std::cout << "Enter determinants (space-separated): ";
    for (int j = 0; j < numDeterminants; ++j) {
        std::string attr;
        std::cin >> attr;
        determinant.insert(attr);
    }

    std::cout << "Enter number of dependents for FD " << (i + 1) << ": ";
    std::cin >> numDependents;

    std::cout << "Enter dependents (space-separated): ";
    for (int j = 0; j < numDependents; ++j) {
        std::string attr;
        std::cin >> attr;
        dependent.insert(attr);
    }

    relation.addFunctionalDependency(FunctionalDependency(determinant, dependent));
}

// Input candidate keys
int numKeys;
```

```
    std::cout << "Enter number of candidate keys: ";
    std::cin >> numKeys;

    std::set<std::set<std::string>> candidateKeys;
    for (int i = 0; i < numKeys; ++i) {
        std::set<std::string> key;
        int numKeyAttributes;

        std::cout << "Enter number of attributes in candidate key " << (i + 1) << ": ";
        std::cin >> numKeyAttributes;

        std::cout << "Enter attributes for candidate key " << (i + 1) << " (space-separated): ";
        for (int j = 0; j < numKeyAttributes; ++j) {
            std::string attr;
            std::cin >> attr;
            key.insert(attr);
        }

        candidateKeys.insert(key);
    }

    // Display the original relation
    relation.display();

    // Decompose to BCNF
    std::vector<Relation> decomposedRelations = Decomposer::decomposeToBCNF(relation,
candidateKeys);

    // Display decomposed relations
    std::cout << "\nDecomposed Relations:\n";
    for (const auto& rel : decomposedRelations) {
        rel.display();
    }

    return 0;
}
```

OUTPUT:
BCNF:

```
reethi@DESKTOP-8744EFO:~/dir1/dbms$ g++ l9_q3.cpp
reethi@DESKTOP-8744EFO:~/dir1/dbms$ ./a.out
Enter relation name: StudentCourse
Enter number of attributes: 4
Enter attributes (space-separated): StudentID CourseID Instructor CourseName
```

```
Enter number of functional dependencies: 2
Enter number of determinants for FD 1: 1
Enter determinants (space-separated): StudentID
Enter number of dependents for FD 1: 2
Enter dependents (space-separated): CourseID Instructor
Enter number of determinants for FD 2: 1
Enter determinants (space-separated): CourseID
Enter number of dependents for FD 2: 1
Enter dependents (space-separated): CourseName
Enter number of candidate keys: 1
Enter number of attributes in candidate key 1: 2
Enter attributes for candidate key 1 (space-separated): StudentID CourseID
Relation: StudentCourse
Attributes: CourseID CourseName Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
  { CourseID } -> { CourseName }

Decomposed Relations:
Relation: Decomposed_StudentCourse
Attributes: CourseID CourseName
Functional Dependencies:
  { CourseID } -> { CourseName }
Relation: Decomposed_Remaining_StudentCourse
Attributes: CourseID Instructor StudentID
Functional Dependencies:
  { StudentID } -> { CourseID Instructor }
```