How to Determine All Candidate Keys for a Given Relation and a Set of FDs (V)

Same example but different presentation: Let $R(A, B, C, D, E, F)$ be a relation schema, and let $S = \{ DF \rightarrow C, BC \rightarrow F, E \rightarrow A, ABC \rightarrow E \}$ be a set of FDs. Determine all candidate keys.

Draw table with three columns (this corresponds to steps 1, 2, 3, 4, 6): L = Attributes on left sides only, R = Attributes on right sides only, B = Attributes on both sides

<table>
<thead>
<tr>
<th>L</th>
<th>B</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B, D$</td>
<td>$A, C, E, F$</td>
<td>-</td>
</tr>
</tbody>
</table>

Perform steps 5 and 7.
7.6 Boyce-Codd Normal Form

- A relation schema $R$ with FDs $F$ is in **Boyce-Codd normal form (BCNF)**, if, and only if, it is in 3NF and for each FD $A \rightarrow B \in F$ at least one of the following conditions holds:
  - $B \subseteq A$, i.e., the FD $A \rightarrow B$ is trivial.
  - $A$ is superkey of $R$.

- conclusion: The BCNF eliminates dependencies among attributes that are part of a candidate key.

- example:
  - $CarIndex(\text{manufacturer}, \text{manufacturer-id}, \text{model-id})$
  - consider FDs:
    - $FD1: \{\text{model-id, manufacturer}\} \rightarrow \{\text{manufacturer-id}\}$
    - $FD2: \{\text{manufacturer-id}\} \rightarrow \{\text{manufacturer}\}$

- example is in 3NF, but not in BCNF
The following anomalies can arise:

- Insertion of the same manufacturer with different manufacturer ids (and different model ids) is possible.
- 1:1-relationship between manufacturer and manufacturer-id is connected to model-id.

Properties of a schema in BCNF

- A relation schema $R$ with associated FDs $F$ can be decomposed into relation schemas $R_1, ..., R_n$ so that holds:
  
  + The decomposition is lossless.
  + The schemas $R_i (1 \leq i \leq n)$ are all in BCNF.

- But: We cannot always find a BCNF decomposition which is also dependency-preserving. This case is seldom in practice.

Procedure

- Decomposition of a schema from 3NF to BCNF
- Check if this decomposition is dependency-preserving. If this is the case, take this schema. Otherwise use the original schema in 3NF.
- Example: Producer(manufacturer-id, manufacturer), CarIndexNew(manufacturer-id, model-id), decomposition maintains FD2 but loses FD1
  
  lossless join decomposition since CarIndex = Producer $\bowtie$ CarIndexNew
8. Application Programming

8.1 Introduction

Database and programming languages

- SQL is a powerful declarative query language. The formulation of queries in SQL is usually simpler than the coding of the same queries in an all-purpose programming language. There are at least two reasons for accessing a database with a programming language from a user perspective:
  - Not all queries can be expressed in SQL (little functionality for “everyday” programming) since SQL does not have the full expressive power of a programming language. In order to be able to express such queries, SQL can be embedded into a more powerful language. Applications are usually developed in imperative and object-oriented languages (C, Cobol, Fortran, Java, C++, ...).
  - Non-declarative actions like printing, interaction with the user, or transmission of query results to a graphical user interface are outside of SQL. Task sharing: query processing and updates with SQL, all other tasks with the aid of a programming language.
SQL queries can be automatically optimized and efficiently executed. The use of a programming language only makes an automatic optimization extraordinarily difficult.

Ad hoc queries are posed mostly by experts and more seldom. Frequently non-interactive batch applications are needed. Often the possibilities of the DBMS for representing data are limited and unsuitable for user requirements.

Special integration problem (impedance mismatch):
- programming language supports the processing of single data records (tuple-oriented approach).
- SQL supports the processing of data records, i.e., of relations (set-oriented approach).

consequently the question: How can the programming of database tasks be combined with the “usual” tasks without abandoning the benefits of SQL?
Alternatives for coupling

- **loose coupling**: Constructs of the database language are embedded into a program of a programming language and specially marked. Particular methods are employed to migrate from the set-oriented processing of SQL to the processing of single variables/records of the programming language.
  - CALL interface
    + provision of libraries
  - embedding with preprocessor (**embedded SQL**)
    + static: structure of the SQL commands is predefined
    + dynamic: arbitrary SQL commands are allowed

- **integration**: A special database language is developed which incorporates the usual programming language concepts and the set-oriented operations of the relational DBMS in a most possible unified way.
  - language extensions
    + of SQL
    + of an imperative or object-oriented programming language
  - script languages
    + languages similar to BASIC without type concept
    + simple connection to windows- and graphic-oriented interfaces
8.2 Procedural CALL Interface

Use of a library by employing the Oracle Call Interface (OCI)
Components of the CALL interface

- data structures shared by the AP (application program) and the database server
  - for establishing the communication
  - for processing a query
- cursor concept: data structure used in the AP for accessing the relations of the DB
- in AP storage of SQL queries in a string
- type checking only possible in the AP
- binding of the variables of the AP to the data structures of the DBMS server

running an AP:

1. establishing the connection to the DBMS server
2. initialising a cursor
3. parsing an SQL statement
4. binding input variables to an SQL statement
5. closing the cursor
6. decoupling from the server
7. executing an update or a query
- executing a query
  - requesting the output parameters
  - binding the output to variables of the AP
  - positioning the cursor
  - abort of the query

- drawbacks of the CALL interface
  - complicated programming
  - error-prone

- advantage: high flexibility
JDBC - a CALL interface in Java

- database programming in Java together with a use of SQL
  - **JDBC** (*Java Data Base Connectivity*) protocol allows Java applications to access relational databases, independently of a particular DBMS.
  - Queries are transmitted as uninterpreted strings to the DBMS.
  - Results are sent through objects of a class *ResultSet* from the DBMS to the AP.

- client-server concept
  - DBMS runs as server for several clients on another computer than the AP.

- uniform interface for different DBMS

- use of strong typing (whenever possible)

- support of important concepts
  - static queries
  - SQL queries that can be parameterized
  - support of very large objects
  - dynamic queries and metadata
Establishing a connection

- creation of a connection object
- `Connection con = DriverManager.getConnection("jdbc:oracle:thin:@venus.mathematik.uni-marburg.de:1521:Init_DB", "scott", "tiger");`
  - First string corresponds to an URL to the database.
  - Second string is the user name.
  - Third string is the password.
- Before creating the connection object the corresponding driver class has to selected.
  - `Class.forName("oracle.jdbc.driver.OracleDriver");`

Interpreted queries

- SQL query is interpreted (translated and at the same time executed). The result of the query is transferred to an object of class `ResultSet`. A repeated execution of the query requires a new interpretation. The query itself cannot be parameterized.
example:

// Creation of a new object of class Statement
Statement stmt = con.createStatement();

// Translation of the query and creation of a new object of class ResultSet
ResultSet rs = stmt.executeQuery("select count(*) as number from user_tables");

// Operation next provides the functionality of an iterator.
rs.next();

// Access to the attribute values with get functions
System.out.println("Number of tables: "+ rs.getInt(1));
System.out.println("Number of tables: "+ rs.getString("number")); // alternatively

Precompiled queries

- In these queries the translation of an SQL query is separated from the execution of the query.

example:

// An SQL query is translated with two parameters.
PreparedStatement stmt =
   con.prepareStatement("select x, y from Points where x < ? and x > ?");