UNIT III
ADVANCED SQL

Advanced SQL: SQL Data Types and Schemas, Integrity Constraints, Authorization, Embedded SQL, Dynamic SQL, Functions and Procedural Constructs, Recursive Queries, Advanced SQL Features.

Relational Database Design: Features of Good Relational Design, Atomic Domains and First Normal Form, Functional-Dependency Theory, Decomposition using Functional Dependencies.

SQL Data Types and Schemas

Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
  - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
  - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
  - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
  - Example: interval '1' day
  - Subtracting a date/time/timestamp value from another gives an interval value
  - Interval values can be added to date/time/timestamp values

- Can extract values of individual fields from date/time/timestamp
  Example: **extract** (**year from** r.starttime)

- Can cast string types to date/time/timestamp
  Example: **cast** <string-valued-expression> **as date**
  Example: **cast** <string-valued-expression> **as time**

User-Defined Types

create type construct in SQL creates user-defined type

create type *Dollars* **as numeric (12,2) final**

Usage:

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

create domain construct in SQL-92 creates user-defined domain types

create domain *person_name* **char**(20) **not null**

Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.

- **create domain** *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Domain Constraints

- Domain constraints are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
  Example: create domain *Dollars* numeric (12, 2)
  create domain *Pounds* numeric (12,2)
- We cannot assign or compare a value of type Dollars to a value of type pounds.
  - However, we can convert type as below
    (cast *r.A* as *Pounds*)

Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*:
    - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
    - **clob**: character large object -- object is a large collection of character data
    - When a query returns a large object, a pointer is returned rather than the large object itself.

Index Creation

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))
- **create index** *studentID_index* **on** *student*(*ID*)
- Indices are data structures used to speed up access to records with specified values for index attributes
    - Ex: **select ***
      from *student*
      where *ID* = '12345'

can be executed by using the index to find the required record, without looking at all records of *student*

Create Table Extensions

Applications often require creation of tables that have the same schema as an existing table. SQL provides a **create table like** extension to support this task:

create table *temp instructor* **like** *instructor*;

The above statement creates a new table *temp instructor* that has the same schema as *instructor*

For example the following statement creates a table *t1* containing the results of a query.

create table *t1* **as**
(**select** *
from *instructor*
where *dept_name*= 'Music');

By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name.

Schemas, Catalogs, and Environments

Early file systems were flat; that is, all files were stored in a single directory. Current file systems, of course, have a directory (or, synonymously, folder) structure, with files stored within subdirectories. To name a file uniquely, we must specify the full path name of the file, for example, /users/avi/db-book/chapter3.tex.

Like early file systems, early database systems also had a single name space for all relations. Users had to coordinate to make sure they did not try to use the same name for different relations. Contemporary database systems provide a three-level hierarchy for naming relations. The top level of the hierarchy consists of **catalogs**, each of which can contain **schemas**. SQL objects such as relations and views are contained within a **schema**.

In order to perform any actions on a database, a user (or a program) must first *connect* to the database. The user must provide the user name and usually, a password for verifying the identity of the user. Each user has

a default catalog and schema, and the combination is unique to the user. When a user connects to a database system, the default catalog and schema are set up for the connection; this corresponds to the current directory being set to the user's home directory when the user logs into an operating system.

To identify a relation uniquely, a three-part name may be used, for example,
catalog5.univ schema.course

We may omit the catalog component, in which case the catalog part of the name is considered to be the default catalog for the connection. Thus if *catalog5* is the default catalog, we can use *univ schema.course* to identify the same relation uniquely. If a user wishes to access a relation that exists in a different schema than the default schema for that user, the name of the schema must be specified. However, if a relation is in the default schema for a particular user, then even the schema name may be omitted. Thus we can use just *course* if the default catalog is *catalog5* and the default schema is *univ schema*.

With multiple catalogs and schemas available, different applications and different users can work independently without worrying about name clashes. Moreover, multiple versions of an application—one a production version, other test versions—can run on the same database system. The default catalog and schema are part of an **SQL environment** that is set up for each connection. The environment additionally contains the user identifier (also referred to as the *authorization identifier*). All the usual SQL statements, including the DDL and DML statements, operate in the context of a schema.

We can create and drop schemas by means of **create schema** and **drop schema** statements. In most database systems, schemas are also created automatically when user accounts are created, with the schema name set to the user account name. The schema is created in either a default catalog, or a catalog specified in creating the user account. The newly created schema becomes the default schema for the user account. Creation and dropping of catalogs is implementation dependent and not part of the SQL standard.

Integrity Constraints

Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.

- A checking account must have a balance greater than $10,000.00
- A salary of a bank employee must be at least $4.00 an hour
- A customer must have a (non-null) phone number

Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (*P*), where *P* is a predicate

Not Null Constraint

Ex: -- Declare *name* and *budget* to be **not null**
name **varchar**(20) **not null**
budget **numeric**(12,2) **not null**

-- Declare the domain *Dollars* to be **not null**

create domain *Dollars* **numeric**(12,2) **not null**

The Unique Constraint

unique (*A*1, *A*2, …, *A*m)
The unique specification states that the attributes *A*1, *A*2, … *A*m form a candidate key.
Note: Candidate keys are permitted to be null (in contrast to primary keys).

primary key : both unique and not null
dept_name vachar(20) primary key;
or
primary key (dept_name);

The check clause

check (*P*), where *P* is a predicate
Example: ensure that semester is one of fall, winter, spring or summer:

create table *section* (
course_id **varchar** (8),
sec_id **varchar** (8),
semester **varchar** (6),
year **numeric** (4,0),
building **varchar** (15),
room_number **varchar** (7),
time slot id **varchar** (4),
primary key (*course_id*, *sec_id*, *semester*, *year*),
check (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'))
);

- The **check** clause in SQL-92 permits domains to be restricted:
  - Use **check** clause to ensure that an hourly_wage domain allows only values greater than a specified value.
  - **create domain** *hourly_wage* **numeric(5,2)**
    constraint *value_test* **check**(*value* > = 4.00)
  - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
  - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.

Default Values

SQL allows a default value to be specified for an attribute as illustrated by the following **create table** statement:

create table *student*
(*ID* **varchar** (5),
name **varchar** (20) **not null**,
dept_name **varchar** (20),
tot cred **numeric** (3,0) **default** 0,
primary key (*ID*));

The default value of the *tot cred* attribute is declared to be 0. As a result, when a tuple is inserted into the *student* relation, if no value is provided for the *tot cred* attribute, its value is set to 0. The following insert statement illustrates how an insertion can omit the value for the *tot cred* attribute.

insert into *student*(*ID*, *name*, *dept_name*) **values** ('12789', 'Newman', 'Comp. Sci.');

Referential Integrity

It Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
-- Example: If "Biology" is a department name appearing in one of the tuples in the *instructor* relation, then there exists a tuple in the *department* relation for "Biology".

- Let A be a set of attributes. Let R and S be two relations that contain attributes A and where A is the primary key of S. A is said to be a **foreign key** of R if for any values of A appearing in R these values also appear in S.

Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
- The primary key clause lists attributes that comprise the primary key.
- The unique key clause lists attributes that comprise a candidate key.
- The foreign key clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key. By default, a foreign key references the primary key attributes of the referenced table.

Example : *dept_name* **varchar**(20) **references** *department*
Or
create table *classroom* (*building* **varchar** (15), *room number* **varchar** (7), *capacity* **numeric** (4,0),
primary key (*building*, *room number*))

create table *department*
(*dept_name* **varchar** (20), *building* **varchar** (15), *budget* **numeric** (12,2) **check** (*budget* > 0),
primary key (*dept_name*))

create table *course*
(*course id* **varchar**(8),*title* **varchar**(50),*dept_name* **varchar**(20), *credits* **numeric** (2,0) **check** (*credits* > 0),
primary key (*course id*),
foreign key (*dept_name*) **references** *department*)

create table *instructor*
(*ID* **varchar** (5), *name* **varchar** (20), **not null** *dept_name* **varchar** (20),
salary **numeric** (8,2), **check** (*salary* > 29000),
primary key (*ID*),
foreign key (*dept_name*) **references** *department*)

create table *section*
(*course id* **varchar** (8), *sec id* **varchar** (8),
semester **varchar** (6), **check** (*semester* **in** ('Fall', 'Winter', 'Spring', 'Summer'),
year **numeric** (4,0), **check** (*year* > 1759 and *year* < 2100)
building **varchar** (15), *room number* **varchar** (7), *time slot id* **varchar** (4),
primary key (*course id*, *sec id*, *semester*, *year*),
foreign key (*course id*) **references** *course*,
foreign key (*building*, *room number*) **references** *classroom*)

When a referential-integrity constraint is violated, the normal procedure is to reject the action that caused the violation (that is, the transaction performing the update action is rolled back). However, a **foreign key** clause can specify that if a delete or update action on the referenced relation violates the constraint, then, instead of rejecting the action, the system must take steps to change the tuple in the referencing relation to restore the constraint. Consider this definition of an integrity constraint on the relation *course*:

Cascading Actions in Referential Integrity

create table *course*
(. . . **foreign key** (*dept_name*) **references** *department*
on delete cascade
on update cascade, . . .);

Because of the clause **on delete cascade** associated with the foreign-key declaration, if a delete of a tuple in *department* results in this referential-integrity constraint being violated, the system does not reject the delete. Instead, the delete "cascades" to the *course* relation, deleting the tuple that refers to department that was deleted. Similarly, the system does not reject an update to a field referenced by the constraint if it violates the constraint; instead, system updates the field *dept_name* in the referencing tuples in *course* to the new value as well.
SQL also allows the **foreign key** clause to specify actions other than **cascade**, if the constraint is violated: The referencing field (here, *dept_name*) can be set to *null* (by using **set null** in place of **cascade**), or to the default value for the domain (by using **set default**).

Integrity Constraint Violation During Transactions

Ex: **create table** *person* (
ID **char**(10),
name **char**(40),
mother **char**(10),
father **char**(10),
primary key *ID,*
foreign key *father* **references** *person,*
foreign key *mother* **references** *person*)

- How to insert a tuple without causing constraint violation?
  - insert father and mother of a person before inserting person
  - OR, set father and mother to null initially, update after inserting all persons (not possible if father and mother attributes declared to be **not null**)
  - OR defer constraint checking (next slide)

Complex Check Clauses

- **check** (*time_slot_id* **in** (**select** *time_slot_id* **from** *time_slot*))
  - why not use a foreign key here?
- Every section has at least one instructor teaching the section.
  - how to write this?
- Unfortunately: subquery in check clause not supported by pretty much any database
  - Alternative: triggers (later)
- **create assertion** <assertion-name> **check** <predicate>;
  - Also not supported by anyone

Assertions: An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.

- An assertion in SQL takes the form
  - **create assertion** <assertion-name> **check** <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
  - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting
  for all *X*, *P*(*X*)
  is achieved in a round-about fashion using
  not exists *X* such that not *P*(*X*)

--Ex: update credits if passed in course.

create assertion *credits earned constraint* **check**
(**not exists** (**select** ID
from *student*
where *tot cred* <> (**select sum**(*credits*)
from *takes* **natural join** *course*
where *student.ID= takes.ID*
and *grade* **is not null and** *grade*<> 'F')

Authorization

Forms of authorization on parts of the database:

- **Read** - allows reading, but not modification of data.
- **Insert** - allows insertion of new data, but not modification of existing data.
- **Update** - allows modification, but not deletion of data.
- **Delete** - allows deletion of data.

Forms of authorization to modify the database schema (covered in Chapter 8):

- **Index** - allows creation and deletion of indices.
- **Resources** - allows creation of new relations.
- **Alteration** - allows addition or deletion of attributes in a relation.
- **Drop** - allows deletion of relations.

Authorization Specification in SQL

The **grant** statement is used to confer authorization

grant <privilege list> **on** <relation name or view name> **to** <user list>

<user list> is:

- a user-id
- **public**, which allows all valid users the privilege granted
- A role

Granting a privilege on a view does not imply granting any privileges on the underlying relations. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

Privileges in SQL

- **select:** allows read access to relation,or the ability to query using the view
  - Example: grant users *U*1, *U*2, and *U*3 **select** authorization on the *branch* relation:
    grant select on department **to** *U*1*, U*2*, U*3
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation:

grant update (*budget*) **on** *department* **to** *Amit*, *Satoshi*;

The user name **public** refers to all current and future users of the system. Thus, privileges granted to **public** are implicitly granted to all current and future users.

Revoking Authorization in SQL: The **revoke** statement is used to revoke authorization.

revoke <privilege list> **on** <relation name or view name> **from** <user list>
Example: **revoke select on** *department* **from** *Amit*, *Satoshi*;
revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*;

- All privileges that depend on the privilege being revoked are also revoked.
- <privilege-list> may be **all** to revoke all privileges the revokee may hold.
- If the same privilege was granted twice to the same user by different grantees, the user may retain the privilege after the revocation.

Roles

- **create role** instructor;
- **grant** *instructor* **to Amit;**
- Privileges can be granted to roles:
  - **grant select on** *takes* **to** *instructor*;
- Roles can be granted to users, as well as to other roles
  - **create role** *teaching_assistant*
  - **grant** *teaching_assistant* **to** *instructor*;
    - *Instructor* inherits all privileges of *teaching_assistant*
- Chain of roles
  - **create role** *dean*;
  - **grant** *instructor* **to** *dean*;
  - **grant** *dean* **to** Satoshi;

Authorization on Views

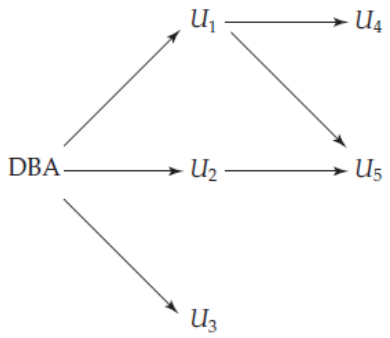
- **create view** *geo_instructor* **as**
  (**select** *
  from *instructor*
  where *dept_name* = 'Geology');
- **grant select on** *geo_instructor* **to** *geo_staff*
- Suppose that a *geo_staff* member issues
  - **select** *
    from *geo_instructor*;
- What if
  - *geo_staff* does not have permissions on *instructor?*
  - creator of view did not have some permissions on *instructor?*

Other Authorization Features

- **references** privilege to create foreign key
  - **grant reference** (*dept_name*) **on** *department* **to** Mariano;

- transfer of privileges
  - **grant select on** *department* **to** Amit **with grant option**;

The passing of a specific authorization from one user to another can be represented by an **authorization graph**. The nodes of this graph are the users. Consider the graph for update authorization on *teaches*. The graph includes an edge $U_i \rightarrow U_j$ if user U_i grants update authorization on *teaches* to U_j. The root of the graph is the database administrator.



Authorization-grant graph ($U_1, U_2, \ldots, U_5$ are users and DBA refers to the database administrator).

Revoking of Privileges

Revocation of a privilege from a user/role may cause other users/roles also to lose that privilege. This behavior is called *cascading revocation*. In most database systems, cascading is the default behavior. However, the **revoke** statement may specify **restrict** in order to prevent cascading revocation:

- **revoke select on** *department* **from** Amit, Satoshi **cascade**;
- **revoke select on** *department* **from** Amit, Satoshi **restrict**;

The following **revoke** statement revokes only the grant option, rather thanthe actual **select** privilege:

- **revoke grant option for select on** *department* **from** Amit;

Cascading revocation is inappropriate in many situations. Suppose Satoshi has the role of *dean*, grants *instructor* to Amit, and later the role *dean* is revoked from Satoshi (perhaps because Satoshi leaves the university); Amit continues to be employed on the faculty, and should retain the *instructor* role.

To deal with the above situation, SQL permits a privilege to be granted by a role rather than by a user. SQL has a notion of the current role associated with a session. By default, the current role associated with a session is null (except in some special cases). The current role associated with a session can be set by executing **set role** *role name*. The specified role must have been granted to the user; else the **set role** statement fails. To grant a privilege with the grantor set to the current role associated with a session, we can add the clause:

granted by current role to the grant statement, provided the current role is not null.

Accessing SQL from a Programming Language

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a database programmer must have access to a general-purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, there must be a means to combine SQL with a general-purpose programming language.

There are two approaches to accessing SQL from a general-purpose programming language:

1. **Dynamic SQL:** A general-purpose program can connect to and communicate with a database server using a collection of functions (for procedural languages) or methods (for object-oriented languages). Dynamic SQL allows the program to construct an SQL query as a character string at runtime, submit the query, and then retrieve the result into program variables a tuple at a time. The *dynamic SQL* component of SQL allows programs to construct and submit SQL queries at runtime.
2. **Embedded SQL:** Like dynamic SQL, embedded SQL provides a means by which a program can interact with a database server. However, under embedded SQL, the SQL statements are identified at compile time using a preprocessor. The preprocessor submits SQL statements to the database system for pre compilation and optimization; then it replaces the SQL statements in the application program with appropriate code and function calls before invoking the programming-language compiler.

Embedded SQL

SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding same queries in a general-purpose programming language. However, a programmer must have access to a database from a general purpose programming language for two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.
SQL is designed so that queries written in it can be optimized automatically and executed efficiently—and providing the full power of a programming language makes automatic optimization exceedingly difficult.

2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and Cobol. A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise *embedded* SQL. The basic form of these languages follows that of the System R embedding of SQL into PL/I.

EXEC SQL statement is used to identify embedded SQL request to the preprocessor

EXEC SQL <embedded SQL statement > END_EXEC

Note: this varies by language (for example, the Java embedding uses

SQL { <embedded SQL statement > };)

Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

Before executing any sql statement, the program must first connect to the database as:

EXEC SQL **connect to** server_name **user** user_name END-EXEC

To write a relational query, we use the **declare cursor** statement. The result of the query is not yet computed. Rather, the program must use the **open** and **fetch** commands to obtain the result tuples.

Example: From within a host language, find the ID and name of students who have completed more than the number of credits stored in variable credit_amount.

- Specify the query in SQL and declare a *cursor* for it

```
EXEC SQL
  declare c cursor for
select ID, name
```

from *student*
where tot_cred > *:credit_amount*
END_EXEC

The **open** statement causes the query to be evaluated
EXEC SQL **open** *c* END_EXEC
The **fetch** statement causes the values of one tuple in the query result to be placed on host language variables.
EXEC SQL **fetch** *c* **into** :si*, :sn* END_EXEC

Repeated calls to **fetch** get successive tuples in the query result
A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available

The **close** statement causes the database system to delete the temporary relation that holds the result of the query.
EXEC SQL **close** *c* END_EXEC

These details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.

SQLJ, the Java embedding of SQL, provides a variation, where Java iterators are used in place of cursors. SQLJ associates the results of a query with an iterator, and the next() method of the Java iterator interface can be used to step through the result tuples, just as the preceding examples use **fetch** on the cursor. Embedded SQL expressions for database modification (**update**, **insert**, and **delete**) do not return a result. Thus, they are somewhat simpler to express. A database modification request takes the form

EXEC SQL < any valid **update, insert,** or **delete**> END-EXEC

Embedded SQL allows a host-language program to access the database, but it provides no assistance in presenting results to the user or in generating reports.

Dynamic SQL

Allows programs to construct and submit SQL queries at run time.
-- Example of the use of dynamic SQL from within a C program.

```
char *  sqlprog = "update instrucotor
                   set sal = sal * 1.05
                 where ID = ?"
EXEC SQL prepare dynprog  from :sqlprog;
char ins [10] = "201101";
EXEC SQL execute dynprog using :instructor;
```

The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.

ODBC and JDBC

API (application-program interface) for a program to interact with a database server
Application makes calls to
- Connect with the database server
- Send SQL commands to the database server
- Fetch tuples of result one-by-one into program variables

ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
JDBC (Java Database Connectivity) works with Java

ODBC: Open DataBase Connectivity(ODBC) is a standard for application program to communicate with a database server. It is an Application Program Interface (API) to
- open a connection with a database,
- send queries and updates,
- get back results.

Applications such as GUI, spreadsheets, etc. can use ODBC

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
  - connection handle,
  - the server to which to connect
  - the user identifier,
  - password
- Must also specify types of arguments:
  - SQL_NTS denotes previous argument is a null-terminated string.
- Good programming requires checking results of every function call for errors;

JDBC: JDBC is a Java API for communicating with database systems supporting SQL. It supports a variety of features for querying and updating data, and for retrieving query results

- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
  - Open a connection
  - Create a "statement" object
  - Execute queries using the Statement object to send queries and fetch results
  - Exception mechanism to handle errors

Procedural Extensions and Stored Procedures

SQL provides a **module** language that permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.

Stored Procedures

- Can store procedures in the database
- then execute them using the **call** statement
- permit external applications to operate on the database without knowing about internal details

Functions and Procedures: SQL:1999 supports functions and procedures. Functions/procedures can be written in SQL itself, or in an external programming language

--Functions are particularly useful with specialized data types such as images and geometric objects

- Example: functions to check if polygons overlap, or to compare images for similarity

-- Some database systems support **table-valued functions**, which can return a relation as a result

SQL:1999 also supports a rich set of imperative constructs, including Loops, if-then-else, assignment constructs. Many databases have proprietary procedural extensions to SQL that differ from SQL:1999

SQL Functions: *Define a function that, given the name of a department, returns the count of the number of instructors in that department.*

```
create function dept_count (dept_name varchar(20))
    returns integer
   begin
      declare d_count integer;
      select count (* ) into d_count
      from instructor
      where instructor.dept_name = dept_name
      return d_count;
    end
```

- *Find the department name and budget of all departments with more than 12 instructors.*

```
        select dept_name, budget
      from department
      where dept_count (dept_name ) > 1
```

SQL Procedures

- *The dept_count function could instead be written as procedure:*

```
create procedure dept_count_proc (in dept_name varchar(20), out d_count integer)
begin
        select count(*) into d_count
        from instructor
       where instructor.dept_name = dept_count_proc.dept_name
end
```

- *Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.*

```
        declare d_count integer;
    call dept_count_proc( 'Physics', d_count);
```

- *Procedures and functions can be invoked also from dynamic SQL*
- *SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ*

Procedural Constructs

1) *Compound statement: **begin ... end**,*
   - *May contain multiple SQL statements between **begin** and **end**.*
   - *Local variables can be declared within a compound statements*

2) ***While** and **loop** statements:*

```
        declare n integer default 0;
        while n < 10 do
           set n = n + 1
        end loop
        loop
    set n = n  – 1
        until n = 0
        end loop
```

3) ***For** loop*
   - *Permits iteration over all results of a query*
   - *Example:*

```
    declare n  integer default 0;
    for r  as
        select budget from department
         where dept_name = 'Music'
     do
            set n = n - r.budget
     end loop
```

4) *Conditional statements (**if-then-else**)*

*SQL:1999 also supports a **case** statement similar to C case statement*

- *Example procedure: registers student after ensuring classroom capacity is not exceeded*
  - *Returns 0 on success and -1 if capacity is exceeded*
  - *See book for details*
- *Signaling of exception conditions, and declaring handlers for exceptions*

```
        declare out_of_classroom_seats condition
    declare exit handler for out_of_classroom_seats
    begin
    …
       ..  signal out_of_classroom_seats
    end
```

- *The handler here is **exit** -- causes enclosing **begin..end** to be exited*

Recursion in SQL: *SQL:1999 permits recursive view definition*
Example: find which courses are a prerequisite, whether directly or indirectly, for a specific course

```
with recursive rec_prereq(course_id, prereq_id) as (
        select course_id, prereq_id
        from prereq
    union
        select rec_prereq.course_id, prereq.prereq_id,
        from rec_rereq, prereq
        where rec_prereq.prereq_id = prereq.course_id
    )

select ∗from rec_prereq;
```

This example view, rec_prereq, is called the transitive closure of the prereq relation

The Power of Recursion

- *Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.*

Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of prereq with itself

- *This can give only a fixed number of levels of prereq*
- *Given a fixed non-recursive query, we can construct a database with a greater number of levels of prerequisites on which the query will not work*
- *Alternative: write a procedure to iterate as many times as required*

- *Computing transitive closure using iteration, adding successive tuples to rec_prereq*
  - *The next we show a prereq relation*
  - *Each step of the iterative process constructs an extended version of rec_prereq from its recursive definition.*
  - *The final result is called the fixed point of the recursive view definition.*
- *Recursive views are required to be **monotonic**. That is, if we add tuples to prereq the view rec_prereq contains all of the tuples it contained before, plus possibly more*

Example of Fixed-Point Computation

course_id	*prereq_id*
BIO-301	BIO-101
BIO-399	BIO-101
CS-190	CS-101
CS-315	CS-101
CS-319	CS-101
CS-347	CS-101
EE-181	PHY-101

Iteration Number	Tuples in cl
0	
1	(CS-301)
2	(CS-301), (CS-201)
3	(CS-301), (CS-201)
4	(CS-301), (CS-201), (CS-101)
5	(CS-301), (CS-201), (CS-101)

Recursive queries should not use any of the following constructs as they make the query nonmonotonic.

1) *Aggregation on the recursive view*
2) *Not exists on a subquery that uses the recursive view*
3) *Set difference (except) whose right hand side uses the recursive view.*

Advanced SQL Features

Create table extensions: *In SQL 1999 to Create a table with the same schema as an existing table: create table temp_ins like instructor*
it requires 2 steps 1) creation and 2) store data.

In SQL 2003 we have a provision that combines both the steps into one: Create table ti as < sql query> ***with data***

More on Sub Queries:

--SQL:2003 allows subqueries to occur anywhere a value is required provided the subquery returns only one value. This applies to updates as well
--SQL:2003 allows subqueries in the from clause to access attributes of other relations in the from clause using the lateral construct:

```
select name, salary, avg_salary
from instructor I1,
lateral (select avg(salary) as avg_salary
         from instructor I2
         where I2.dept_name= I1.dept_name);
```

When sub queries are written in either select or where clause we can access the attributes of outer query but when the sub query is written in from clause we cannot access. So to facilitate this we use lateral clause.

RELATIONAL DATABASE DESIGN

Features of Good Relational Design

Schema for the university database.

1. *classroom*(*building*, *room number*, *capacity*)
2. *department*(*dept_name*, *building*, *budget*)
3. *course*(*course id*, *title*, *dept_name*, *credits*)
4. *instructor*(*ID*, *name*, *dept_name*, *salary*)
5. *section*(*course id*, *sec id*, *semester*, *year*, *building*, *room number*, *time slot id*)
6. *teaches*(*ID*, *course id*, *sec id*, *semester*, *year*)
7. *student*(*ID*, *name*, *dept_name*, *tot cred*)
8. *takes*(*ID*, *course id*, *sec id*, *semester*, *year*, *grade*)
9. *advisor*(*s ID*, *i ID*)
10. *time slot*(*time slot id*, *day*, *start time*, *end time*)
11. *prereq*(*course id*, *prereq id*)

Design Alternatives: Larger-Schema

- Suppose we combine *instructor* and *department* into *inst_dept*
  - *(No connection to relationship set inst_dept)*

ID	*name*	*salary*	*dept_name*	*building*	*budget*
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
83821	Brandt	92000	Comp. Sci.	Taylor	100000
15151	Mozart	40000	Music	Packard	80000
33456	Gold	87000	Physics	Watson	70000
76543	Singh	80000	Finance	Painter	120000

- Result is possible repetition of information

A Combined Schema without Repetition

- Consider combining relations
  - *sec_class(sec_id, building, room_number)* and
  - *section(course_id, sec_id, semester, year)*
- into one relation
  - *section(course_id, sec_id, semester, year, building, room_number)*
- No repetition in this case

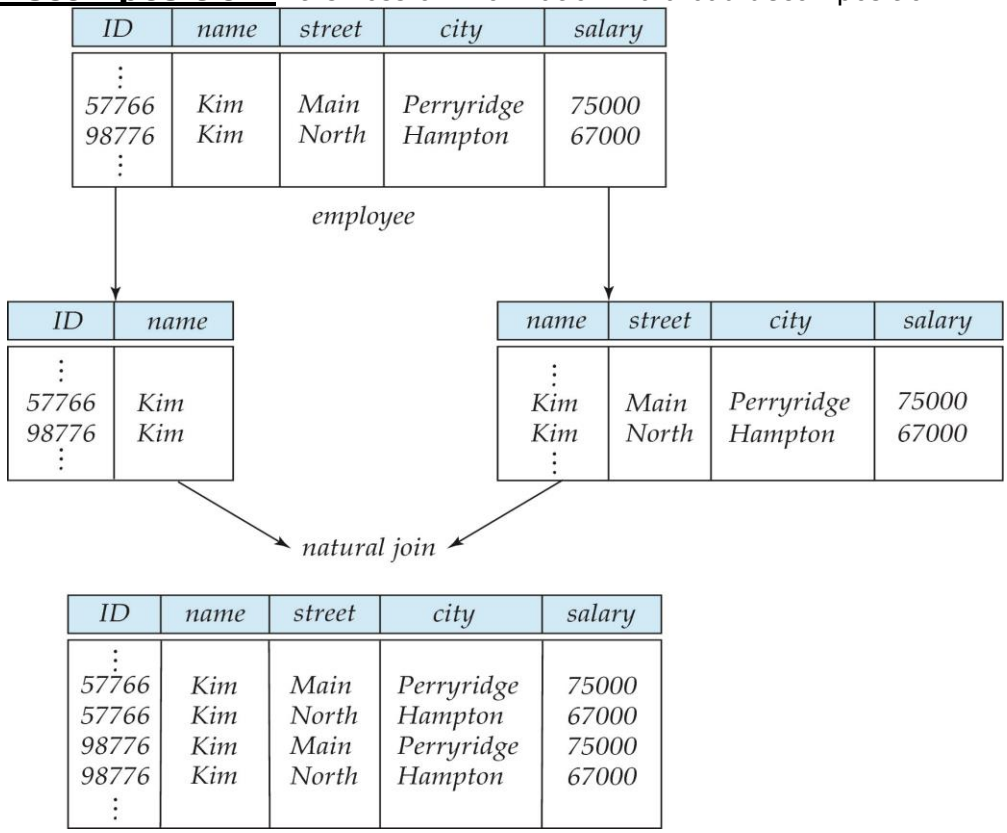
Design Alternatives: Smaller-Schema

- Suppose we had started with *inst_dept.* How would we know to split up (**decompose**) it into *instructor* and *department*?
- Write a rule "if there were a schema (*dept_name, building, budget*), then *dept_name* would be a candidate key"
- Denote as a **functional dependency**:
  - *dept_name* $\rightarrow$ *building, budget*
- In *inst_dept*, because *dept_name* is not a candidate key, the building and budget of a department may have to be repeated.
  - This indicates the need to decompose *inst_dept*
- Not all decompositions are good. Suppose we decompose
  employee(ID, name, street, city, salary) into

- *employee1* (*ID*, *name*)
- *employee2* (*name*, *street, city, salary*)

➢ The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a **lossy decomposition**.

A Lossy Decomposition: It is Loss of information via a bad decomposition.



Our decomposition is unable to represent certain important facts of University employees. So we need to avoid this, and it is said to be **lossy decomposition** and on the other hand if we are able to get by original data then it is called as a **lossless decomposition**.

Example of **Lossless join decomposition**

- Decomposition of *R = (A, B, C) into* $R_1 = (A, B)$ $R_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B	C
α	1	A
β	2	B

Functional Dependencies

➢ Constraints on the set of legal relations.
➢ Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
➢ A functional dependency is a generalization of the notion of a *key.*

Definition: Functional dependency is a relationship that exists when one attribute uniquely determines another attribute. It is a relationship between attributes of a table dependent on each other. It helps in preventing data redundancy and gets to know about bad designs**.** Functional dependencies are constraints on the set of legal relations.
Syntax: A -> B
Ex: Account no -> Balance for account table.

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$, then the functional dependency $\alpha \rightarrow \beta$ holds on R if and only if for any legal relations r(R), whenever any two tuples t1 and t2 of r agree on the attributes of α, they also agree on the attributes of β. That is,

$$t1[\alpha] = t2[\alpha] \Rightarrow t1[\beta] = t2[\beta]$$

Example: Consider *r*(A,*B*) with the following instance of *r.*

A	B
1	4
1	5
3	7

On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.

- *K* is a superkey for relation schema *R* if and only if $K \rightarrow R$
- *K* is a candidate key for *R* if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

  inst_dept (ID, name, salary, dept_name, building, budget).

  We expect these functional dependencies to hold:

  dept_name→ building and

  ID → building

  but would not expect the following to hold: dept_name → salary

Use of Functional Dependencies: We use functional dependencies to:

- test relations to see if they are legal under a given set of functional dependencies.
  -- If a relation *r* is legal under a set *F* of functional dependencies, we say that *r* **satisfies** *F.*
- specify constraints on the set of legal relations
  -- We say that *F* **holds on** *R* if all legal relations on *R* satisfy the set of functional dependencies *F.*

Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.

For example, a specific instance of *instructor* may, by chance, satisfy *name* $\rightarrow$ *ID.*

Closure of a Set of Functional Dependencies

Given a set *F* of functional dependencies, there are certain other functional dependencies that are logically implied by *F*.

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$

The set of **all** functional dependencies logically implied by *F* is the ***closure*** of *F*.

- We denote the *closure* of *F* by F+.
- F+ is a superset of *F*.

Partial Dependency: If proper subset of candidate key determines non-prime attribute, it is called partial dependency.

Transitive Dependency: When an indirect relationship causes functional dependency it is called Transitive Dependency.

If P -> Q and Q -> R is true, then P-> R is a transitive dependency.

Trivial Dependency: If a functional dependency (FD) $X \rightarrow Y$ holds, where Y is a subset of X, then it is called a trivial FD. Trivial FDs always hold.

Non-Trivial Dependency: If an FD $X \rightarrow Y$ holds, where Y is not a subset of X, then it is called a non-trivial FD.

Normalization of Database

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables. Normalization is used for mainly two purposes.

- Eliminating redundant (useless) data.
- Ensuring data dependencies make sense i.e data is logically stored to maintain data consistency.

Problem without Normalization: Without Normalization, it becomes difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anamolies are very frequent if Database is not normalized.

- **Updation Anamoly:** We may want to update data in record, but it may exist in different tables or at different places due to redundancy.
- **Insertion Anamoly:** We may try to insert a new record but data may not be fully available or that record itself doesn't exist. Eg: to insert student details without knowing his department leads to Insertion Anamoly.
- **Deletion Anamoly:** When we try to delete a record it might have been saved or exists in some other place of database due to redundancy.

Normalization helps to remove anomalies and ensure that database is in consistent state.

Normalization Types:

Normalization types are divided into following normal forms.

1. First Normal Form
2. Second Normal Form
3. Third Normal Form
4. BCNF
5. Fourth Normal Form
6. Fifth Normal Form

First normal form (1NF): A relation is in first normal form if every attribute in every row can contain an atomic (only one single) value. An attribute (column) of a table cannot hold multiple values. It should hold only atomic values.

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

Domain is atomic if its elements are considered to be indivisible units

Examples of non-atomic domains:

- Set of names, composite attributes
- Identification numbers like CS101 that can be broken up into parts

A relational schema R is in first normal form if the domains of all attributes of R are atomic. Non-atomic values complicate storage and encourage redundant storage of data.

- ***We assume all relations are in first normal form***

Second normal form (2NF): A database is in second normal form if it satisfies the following conditions:

- It is in first normal form
- All non-prime attributes are fully functional dependent on the primary key

An attribute that is not part of any candidate key is known as non-prime attribute.

A table that is in 1st normal form and contains only a single key as the primary key is automatically in 2nd normal form.

Example: Suppose a school wants to store the data of teachers and the subjects they teach. They create a table that looks like this: Since a teacher can teach more than one subjects, the table can have multiple rows for a same teacher.

teacher_id	subject	teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	Chemistry	40

Candidate Keys: {teacher_id, subject}

Non prime attribute: teacher_age

The table is in 1 NF because each attribute has atomic values. However, it is not in 2NF because non prime attribute teacher_age is dependent on teacher_id alone which is a proper subset of candidate key. This violates the rule for 2NF as the rule says "**no** non-prime attribute is dependent on the proper subset of any candidate key of the table".

To make the table complies with 2NF we can break it in two tables like this:

teacher_details table:

teacher_id	teacher_age
111	38
222	38
333	40

teacher_subject table:

teacher_id	subject
111	Maths
111	Physics
222	Biology
333	Physics
333	Chemistry

Now the tables comply with Second normal form (2NF).

Third Normal Form(3NF): A relation schema *R* is in **third normal form** (3NF) if for all: $\alpha \rightarrow \beta$ in *F*+ with at least one of the following holds:

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
2. α is a superkey for *R*
3. Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R.*

(**NOTE***:* each attribute may be in a different candidate key)

If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold). And, Third condition is a minimal relaxation of BCNF to ensure dependency preservation.

Boyce-Codd Normal Form(BCNF):

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form

$\alpha \rightarrow \beta$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

1. $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
2. α is a superkey for *R*

Example schema *not* in BCNF: *instr_dept* (*ID, name, salary, dept_name, building, budget*)

because *dept_name*$\rightarrow$ *building, budget* holds on *instr_dept,* but *dept_name* is not a super key

Decomposing a Schema into BCNF

Suppose we have a schema *R* and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose *R* into:

1. (α U β)
2. (*R* - (β - α))

In our example,

α = *dept_name*

β = *building, budget*

and *inst_dept* is replaced by

- (α U β) = (*dept_name, building, budget*)
- (*R* - (β - α)) = (*ID, name, salary, dept_name*)

BCNF and Dependency Preservation

Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation. If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving.* Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form.*

How good is BCNF?

There are database schemas in BCNF that do not seem to be sufficiently normalized

Ex:1 Consider a relation

inst_info (ID, child_name, phone)

where an instructor may have more than one phone and can have multiple children

ID	*child_name*	*phone*
99999 99999 99999 99999	David David William Willian	512-555-1234 512-555-4321 512-555-1234 512-555-4321

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if we add a phone 981-992-3443 to 99999, we need to add two tuples

  (99999, David, 981-992-3443)

  (99999, William, 981-992-3443)

- Therefore, it is better to decompose *inst_info* into:

inst_child

ID	*child_name*
99999 99999 99999 99999	David David William Willian

inst_phone

ID	*phone*
99999 99999 99999 99999	512-555-1234 512-555-4321 512-555-1234 512-555-4321

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF)

Ex: Consider a database: *classes* (*course, teacher, book*) such that (*c, t, b*) $\in$ *classes* means that *t* is qualified to teach *c,*and *b* is a required textbook for *c*
The database is supposed to list for each course the set of teachers any one of which can be the course's instructor, and the set of books, all of which are required for the course (no matter who teaches it).

course	Teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	pete	Stallings

Classes

There are no non-trivial functional dependencies and therefore the relation is in BCNF
Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted
(database, Marilyn, DB Concepts)
(database, Marilyn, Ullman)

Therefore, it is better to decompose *classes* into:
Teaches:

Course	teacher
database	Avi
database	Hunk
database	Sudarshan
operating systems	Avi
operating systems	Jim

Text:

Course	Book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF),

Multivalued Dependencies (MVDs): Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **multivalued dependency**

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[\beta] = t_2[\beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

Example: Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets. *Y, Z, W*

- We say that $Y \twoheadrightarrow Z$ (Y **multidetermines** Z) if and only if for all possible relations r (R)
  $< y_1, z_1, w_1 > \in r$ and $< y_1, z_2, w_2 > \in r$
  Then $< y_1, z_1, w_2 > \in r$ and $< y_1, z_2, w_1 > \in r$
- Note that since the behavior of Z and W are identical it follows that
  $Y \twoheadrightarrow Z$ if $Y \twoheadrightarrow W$
- In our example:
  ID $\twoheadrightarrow$ *child_name*
  ID $\twoheadrightarrow$ *phone_number*
- The above formal definition is supposed to formalize the notion that given a particular value of Y (*ID*) it has associated with it a set of values of Z *(child_name)* and a set of values of W *(phone_number)*, and these two sets are in some sense independent of each other.
  Note: If $Y \rightarrow Z$ then $Y \twoheadrightarrow Z$

Fourth Normal Form (4NF): A relation schema R is in **4NF** with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:

- $\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
- α is a superkey for schema R

- If a relation is in 4NF it is in BCNF

Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used

Goals of Normalization

Let R be a relation scheme with a set F of functional dependencies.

- Decide whether a relation scheme R is in "good" form.
- In the case that a relation scheme R is not in "good" form, decompose it into a set of relation scheme {*R1, R2, ..., Rn*} such that
  - each relation scheme is in good form
  - the decomposition is a lossless-join decomposition
  - Preferably, the decomposition should be dependency preserving.

Functional-Dependency Theory

We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies. We then develop algorithms to generate lossless decompositions into BCNF and 3NF And we then develop algorithms to test if a decomposition is dependency-preserving.

Closure of a Set of Functional Dependencies: Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.

For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow$ C

The set of all functional dependencies logically implied by *F* is the ***closure*** of *F*.
We denote the *closure* of *F* by *F+.*

We can find all of F+ by applying **Armstrong's Axioms**:

1. if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
2. if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
3. if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

These rules are
- **sound** (generate only functional dependencies that actually hold) and
- **complete** (generate all functional dependencies that hold).

Example:

Let R = (A, B, C, G, H, I) and fd are given by
$F = \{ A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$
some members of *F+*
- $A \rightarrow H$
  - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
- $AG \rightarrow I$
  - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
- $CG \rightarrow HI$
  - by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$,

and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity.

We can further simplify manual computation of *F+* by using the following additional rules.
1. If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
2. If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
3. If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity)**

The above rules can be inferred from Armstrong's axioms.

Algorithm for Computing F+ :

To compute the closure of a set of functional dependencies F:

```
F + = F
repeat
    for each functional dependency f in F+
        apply reflexivity and augmentation rules on f
        add the resulting functional dependencies to F +
    for each pair of functional dependencies f1and f2 in F +
        if f1 and f2 can be combined using transitivity
            then add the resulting functional dependency to F +
until F + does not change any further
```

Closure of Attribute Sets: Given a set of attributes a, define the *closure* of a under *F* (denoted by a+) as the set of attributes that are functionally determined by a under *F*

Algorithm to compute a+, the closure of a under *F*

```
result := a;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ
        end
```

Example of Attribute Set Closure

Let R = (A, B, C, G, H, I) and fd's are given by

$F = \{A \rightarrow B$
$A \rightarrow C$
$CG \rightarrow H$
$CG \rightarrow I$
$B \rightarrow H\}$

Now to compute *(AG)+* we trace above algorithm to get :

1. *result = AG*
2. *result = ABCG* *($A \rightarrow C$ and $A \rightarrow B$)*
3. *result = ABCGH* *($CG \rightarrow H$ and $CG \subseteq AGBC$)*
4. *result = ABCGHI* *($CG \rightarrow I$ and $CG \subseteq AGBCH$)*

Is *AG* a candidate key?

1. Is AG a super key?
    1. Does $AG \rightarrow R$? == Is (AG)+ $\supseteq$ R
2. Is any subset of AG a superkey?
    1. Does $A \rightarrow R$? == Is (A)+ $\supseteq$ R
    2. Does $G \rightarrow R$? == Is (G)+ $\supseteq$ R

Uses of Attribute Closure:

There are several uses of the attribute closure algorithm:

1. Testing for super key:
    a. To test if α is a superkey, we compute α+, and check if α+ contains all attributes of *R*.
2. Testing functional dependencies
    a. To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in *F*+), just check if $\beta \subseteq \alpha$+.
    b. That is, we compute α+ by using attribute closure, and then check if it contains β.
    c. Is a simple and cheap test, and very useful
3. Computing closure of F
    a. For each $\gamma \subseteq R$, we find the closure γ+, and for each $S \subseteq \gamma$+, we output a functional dependency $\gamma \rightarrow S$.

Canonical Cover: Sets of functional dependencies may have redundant dependencies that can be inferred from the others.

For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, \quad B \rightarrow C\}$

Parts of a functional dependency may be redundant

Ex:: on RHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow CD\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Ex:: on LHS: $\{A \rightarrow B, \quad B \rightarrow C, \quad AC \rightarrow D\}$ can be simplified to
$\{A \rightarrow B, \quad B \rightarrow C, \quad A \rightarrow D\}$

Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies.

Canonical Cover Definition: A *canonical cover* for *F* is a set of dependencies *Fc* such that

1. *F* logically implies all dependencies in *Fc,* and
2. *Fc* logically implies all dependencies in *F,* and
3. No functional dependency in *Fc* contains an extraneous attribute, and
4. Each left side of functional dependency in *Fc* is unique.

Extraneous Attributes: Consider a set *F* of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in *F*.

1. Attribute A is **extraneous** in α if $A \in \alpha$
   and *F* logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
2. Attribute *A* is **extraneous** in β if $A \in \beta$
   and the set of functional dependencies
   $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies *F.*

Note: implication in the opposite direction is trivial in each of the cases above, since a "stronger" functional dependency always implies a weaker one

Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$

B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (I.e. the result of dropping *B* from $AB \rightarrow C$).

Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$

C is extraneous in $AB \rightarrow CD$ since AB $\rightarrow C$ can be inferred even after deleting *C.*

Testing if an Attribute is Extraneous: *Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.*

1) *To test if attribute $A \in \alpha$ is extraneous in α*
   a) *compute $(\{\alpha\} - A)+$ using the dependencies in F*
   b) *check that $(\{\alpha\} - A)+$ contains β; if it does, A is extraneous in α*
2) *To test if attribute $A \in \beta$ is extraneous in β*
   a) *compute $\alpha+$ using only the dependencies in*
      $F' = (F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$,
   b) *check that $\alpha+$ contains A; if it does, A is extraneous in β*

Computing a Canonical Cover:

To compute a canonical cover for *F*:

repeat
  Use the union rule to replace any dependencies in *F*
    $\alpha 1 \rightarrow \beta 1$ and $\alpha 1 \rightarrow \beta 2$ with $\alpha 1 \rightarrow \beta 1\ \beta 2$
  Find a functional dependency $\alpha \rightarrow \beta$ with an
    extraneous attribute either in α or in β
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$
until *F* does not change

Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied

Let R = (A, B, C) and
F = {A → BC
B → C
A → B
AB → C}

Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$

Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$

1. *A* is extraneous in $AB \rightarrow C$
   a. Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
      i. Yes: in fact, $B \rightarrow C$ is already present!
   b. Set is now $\{A \rightarrow BC, B \rightarrow C\}$
2. *C* is extraneous in $A \rightarrow BC$
   a. Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
      i. Yes*:* using transitivity on $A \rightarrow B$ *and* $B \rightarrow$ C.
         1. Can use attribute closure of *A* in more complex cases
3. The canonical cover is: $\{ A \rightarrow B , B \rightarrow C \}$

Decomposition using functional dependencies

Lossless-join Decomposition: For the case of $R = (R1, R2)$, we require that for all possible relations *r* on schema *R*

$$r = \prod R1\ (r)\ \bowtie\ \prod R2\ (r)$$

A decomposition of *R* into *R*1 and *R*2 is lossless join if and only if at least one of the following dependencies is in *F*+:

1. $R1 \cap R2 \rightarrow R1$
2. $R1 \cap R2 \rightarrow R2$

The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies

Example: *Let R = (A, B, C) and F = {A → B, B → C)*
It Can be decomposed in two different ways

1. *R*1 = *(A, B), R*2 = *(B, C)*
   a. Lossless-join decomposition:
      i. *R*1 ∩ *R*2 = {*B*} and *B* → *BC*
   b. Dependency preserving
2. *R1 = (A, B), R*2 = *(A, C)*
   a. Lossless-join decomposition:
      i. *R*1 ∩ *R*2 = {*A*} and *A* → A*B*
   b. Not dependency preserving
      (cannot check *B* → *C* without computing *R1* *R*2)

Dependency Preservation: Let *Fi* be the set of dependencies *F* + that include only attributes in *Ri.* Decomposition is dependency preserving, if
(*F*1 ∪ *F*2 ∪ … ∪ *F*n)+ = *F* +

If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.

Testing for Dependency Preservation: To check if a dependency $\alpha \rightarrow \beta$ is preserved in a decomposition of *R* into *R*1, *R*2, …, *R*n we apply the following test (with attribute closure done with respect to *F*)

```
result = α
while (changes to result) do
    for each Ri in the decomposition
        t = (result ∩ Ri)+ ∩ Ri
        result = result ∪ t
```

If *result* contains all attributes in β, then the functional dependency $\alpha \rightarrow \beta$ is preserved.

- We apply the test on all dependencies in *F* to check if a decomposition is dependency preserving
- This procedure takes polynomial time, instead of the exponential time required to compute *F*+ and (*F*1 ∪ *F*2 ∪ … ∪ *F*n)+

Example:
Let R = (*A, B, C*) and *F* = {*A* → *B* , *B* → *C*}
Key = {*A*}
R is not in BCNF
Decomposition *R*1 = (*A, B), R*2 = *(B, C)*

a. *R*1 and *R*2 in BCNF
b. Lossless-join decomposition
c. Dependency preserving

Testing for BCNF

To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF

1. Compute α+ (the attribute closure of α), and
2. Verify that it includes all attributes of *R*, that is, it is a super key of *R*.

Simplified test: To check if a relation schema *R* is in BCNF, it suffices to check only the dependencies in the given set *F* for violation of BCNF, rather than checking all dependencies in *F*+.

- If none of the dependencies in *F* causes a violation of BCNF, then none of the dependencies in *F*+ will cause a violation of BCNF either.

However, using only *F* is incorrect when testing a relation in a decomposition of R
Consider *R* = (*A, B, C, D, E*), with *F* = { *A* → *B, BC* → *D*}

- Decompose *R* into *R*1 = (*A,B*) and *R*2 = (*A,C,D, E*)
- Neither of the dependencies in *F* contain only attributes from (*A,C,D,E*) so we might be mislead into thinking *R*2 satisfies BCNF.
- In fact, dependency *AC* → *D* in *F*+ shows *R*2 is not in BCNF.

Testing Decomposition for BCNF: To check if a relation *Ri* in a decomposition of *R* is in BCNF,

a. Either test Ri for BCNF with respect to the restriction of F to Ri (that is, all FDs in F+ that contain only attributes from Ri)
b. or use the original set of dependencies *F* that hold on *R*, but with the following test:
    1. for every set of attributes $\alpha \subseteq Ri$, check that α+ (the attribute closure of α) either includes no attribute of *Ri*- α, or includes all attributes of *Ri*.
    ii. If the condition is violated by some $\alpha \rightarrow \beta$ in *F*, the dependency
    $\alpha \rightarrow (\alpha+ - \alpha) \cap Ri$
    can be shown to hold on *Ri*, and *Ri* violates BCNF.
    iii. We use above dependency to decompose *Ri*

BCNF and Dependency Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving
Let R = (J, K, L) and *F = {JK → L, L → K }*
Two candidate keys = *JK* and *JL*

1. *R* is not in BCNF
2. Any decomposition of *R* will fail to preserve
    1. JK → L

This implies that testing for *JK → L* requires a join

Third Normal Form: There are some situations where

- BCNF is not dependency preserving, and
- efficient checking for FD violation on updates is important

Solution: define a weaker normal form, called Third Normal Form (3NF)

- Allows some redundancy (with resultant problems; we will see examples later)
- But functional dependencies can be checked on individual relations without computing a join.
- There is always a lossless-join, dependency-preserving decomposition into 3NF.

3NF Example

Consider Relation R: let *R = (J, K, L)* and *F = {JK → L, L → K }*

- Two candidate keys: *JK* and *JL*
- *R* is in 3NF
    - *JK → L JK* is a superkey
        L → K K is contained in a candidate key

- Relation *dept_advisor*:
    - *dept_advisor* (*s_ID, i_ID, dept_name)*
      F = {s_ID, dept_name → i_ID, i_ID → dept_name}
    - Two candidate keys: *s_ID, dept_name,* and *i_ID, s_ID*
    - *R* is in 3NF
        - *s_ID, dept_name → i_ID s_ID*
            - *dept_name* is a superkey
        - *i_ID → dept_name*
            - *dept_name* is contained in a candidate key

Redundancy in 3NF: There is some redundancy in this schema

Example of problems due to redundancy in 3NF

Let R = (J, K, L) and F = {JK → L, L → K }

J	L	K
J1	L1	K1
J2	L1	K1
J3	L1	K1
null	L2	K2

- repetition of information (Ex:, the relationship *l*1, *k*1)
    - (*i_ID, dept_name)*

- need to use null values (Ex:, to represent the relationship
  *l*2, *k*2 where there is no corresponding value for *J*).
  - (*i_ID, dept_nameI)* if there is no separate relation mapping instructors to departments

Testing for 3NF: Optimization: Need to check only FDs in *F*, need not check all FDs in *F*+.

1. Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a super key.
2. If α is not a super key, we have to verify if each attribute in β is contained in a candidate key of *R*
   a. this test is rather more expensive, since it involve finding candidate keys
   b. testing for 3NF has been shown to be NP-hard
   c. Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time

Comparison of BCNF and 3NF: It is always possible to decompose a relation into a set of relations that are in 3NF such that:

1. the decomposition is lossless
2. the dependencies are preserved

It is always possible to decompose a relation into a set of relations that are in BCNF such that:

- the decomposition is lossless
- it may not be possible to preserve dependencies.

Design Goals: Goal for a relational database design is:

1. BCNF.
2. Lossless join.
3. Dependency preservation.

If we cannot achieve this, we accept one of

- Lack of dependency preservation
- Redundancy due to use of 3NF

Interestingly, SQL does not provide a direct way of specifying functional dependencies other than super keys. Can specify FDs using assertions, but they are expensive to test that Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.

Overall Database Design Process

We have assumed schema *R* is given

- *R* could have been generated when converting E-R diagram to a set of tables.
- *R* could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
- Normalization breaks *R* into smaller relations.
- *R* could have been the result of some ad hoc design of relations, which we then test/convert to normal form.