



Introduction to SQL and database objects

IBM Information Management – Cloud Computing Center of Competence
IBM Canada Labs

Agenda

- Overview
- Database objects
- SQL introduction
- The SELECT statement
- The UPDATE, INSERT and DELETE statements
- Working with JOINS

Supporting reading material & videos

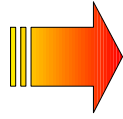
- **Reading materials**

- Getting started with DB2 Express-C eBook
 - Chapter 8: Working with database objects
- Database Fundamentals eBook
 - Chapter 5: Introduction to SQL
- Getting started with IBM Data Studio for DB2 eBook
 - Chapter 4: Creating SQL and XQuery scripts

- **Videos**

- db2university.com course AA001EN
 - Lesson 6: Working with database objects
 - Lesson 7: Introduction to SQL

Agenda



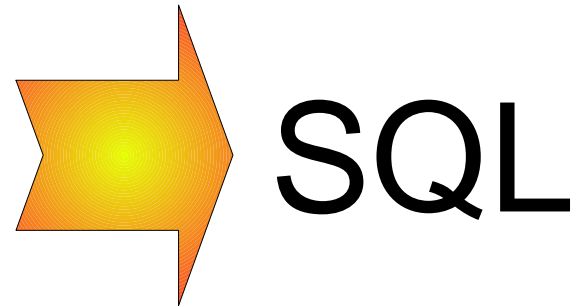
- **Overview**

- Database objects
- SQL introduction
- The SELECT statement
- The UPDATE, INSERT and DELETE statements
- Working with JOINS

Overview

Database objects

- Schema
- Table
- View
- Index
- Synonyms / Alias
- Database application objects
 - Sequences
 - Triggers
 - User Defined Functions (UDFs)
 - Stored Procedures



SQL

05/24/2011

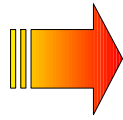
Template Documentation

5

© 2011 IBM Corporation

In this presentation we talk about database objects. To create and manipulate these objects, we use the SQL language.

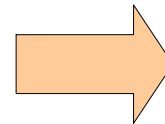
Agenda



- Overview
- **Database objects**
- SQL introduction
- The SELECT statement
- The UPDATE, INSERT and DELETE statements
- Working with JOINS

Database objects

- Schema
- Table
- View
- Index
- Synonyms / Alias
- Database application objects
 - Sequences
 - Triggers
 - User Defined Functions (UDFs)
 - Stored Procedures



These will be covered later in the course, in the topic about application development

Each of these objects are explained in more detail later in this presentation.

For Database Application objects, we will only cover “sequences” in this unit. The rest (highlighted in red) will be covered on a later presentation.

Schemas

- Schemas are *name spaces* for a collection of database objects
- All database objects (except public synonyms) are *qualified* by a two-part name:

```
<schema_name>.<object_name>
```

- A fully qualified object name must be unique
- To create a schema explicitly, use this statement:

```
CREATE SCHEMA <schema_name>
```

- Implicit schemas
 - Uses the connection ID as the implicit schema
 - Can be set with the SET SCHEMA command

In other products, or when talking about relational database concepts, “schema” normally means the entire structure of your database consisting of tables, views, etc. In DB2, a schema normally refers to the prefix or qualifier of an object. Every object in DB2 has two parts: **<schema_name>** and **<object_name>**.

A schema_name doesn’t need to be provided all the time when you work with database objects in DB2, because there will be an implicit schema being used.

Eg: **db2 connect to sample user db2admin using mypassword --> OK**

db2 create table abc.t1 (col1 int, col2 int) → OK

Note that “abc” doesn’t need to map to any user in the operating system. It is just an arbitrary name. This schema was implicitly created, assuming the user has the implicit_schema privilege.

db2 select * from t1 → error, table “db2admin.t1” not found.

Note DB2 uses the user ID of the connection as the implicit schema.

db2 select * from abc.t1 --> OK

db2 create table t1 (col1 int) --> OK, table db2admin.t1 is created

db2 select * from db2admin.t1 --> OK

db2 set schema raul --> OK, changing default schema in current window

db2 select * from t1 --> error, table raul.t1 not found

A typical error for new DB2 users is to create objects with one schema, and later on they cannot find those objects because they are connected with another user ID.

A schema can be used for grouping a bunch of tables together. Maybe they are related. One schema could be used for test, another for development, another one for production. Some utilities in DB2, such as db2move may use the schema as an option to determine what tables should be moved.

Scenario when you can use different schemas:

- Say you can develop an application (say in Java), and you embed SQL statements which refer to DB2 objects, but you don’t include the schema. - Say the connection userID/psw are passed to the program at run time. - Say you have two set of tables which are defined exactly the same, but one was created with the user ID of “TEST”, and the other one with the user ID of “DEV” (for development). - Since the program refers to DB2 objects without an schema, the implicit schema used is the connection ID. Therefore, if you connect as user TEST, you will point to the tables under the TEST schema, and if you use the user DEV, you’d be pointing to the tables under the DEV schema.

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			


- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

A relational database presents data as a collection of tables

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			


 Field
(Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

A table consists of data logically arranged in columns and rows

Each column contains values of the same data type

The storage representation of a column is called a field. So normally using the terms “**column**”, “**field**”, or “**attribute**” is the same.

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

↑
Field
(Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

↑
Field
(Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

↑
Field
(Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

Record (Row) →

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Each row contains a set of values for each column available

The storage representation of a row is called a record. So normally using the terms
“row”, “record”, or “tuple” is the same

Tables

Record (Row) →

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

Record (Row) →

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

Record (Row) →

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

↑
Field
(Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Each intersection of a row and column is called a value

Tables

EMPLOYEE Table

Record (Row) →	Lastname	Firstname	Age	Country
	Pan	Peter	15	Singapore
	Austin	Susan	32	Australia
	...			

↑
Field (Column)

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Tables

Value

Record (Row)

Field (Column)

EMPLOYEE Table

Lastname	Firstname	Age	Country
Pan	Peter	15	Singapore
Austin	Susan	32	Australia
...			

- **Tables store data**
- **A table consists of data logically arranged in columns and rows**
 - Each column contains values of the same data type
 - Each row contains a set of values for each column available

Can a column store data of different types? No

Can rows in a table have different number of columns? No

Creating a table

CREATE TABLE myTable (col1 integer)

myTable	
col1	
120	

```
CREATE TABLE artists
(artno          SMALLINT      not null,
 name           VARCHAR(50)  with default 'abc',
 classification  CHAR(1)      not null,
 bio            CLOB(100K)    logged,
 picture        BLOB(2M)      not logged compact
)
in mytbls1
```

Creating a table is done the same in pretty much all database products. Something to point out is that in DB2 we use the clause ***“in mytbls1”*** if we want to associate this table to the tablespace mytbls1. If we don't specify this clause, the table will normally be created in tablespace USERSPACE1.

Tips:

1) To list all tables/views for your current schema, issue:

db2 list tables

2) For another schema, say “arfchong”, issue:

db2 list tables for schema arfchong

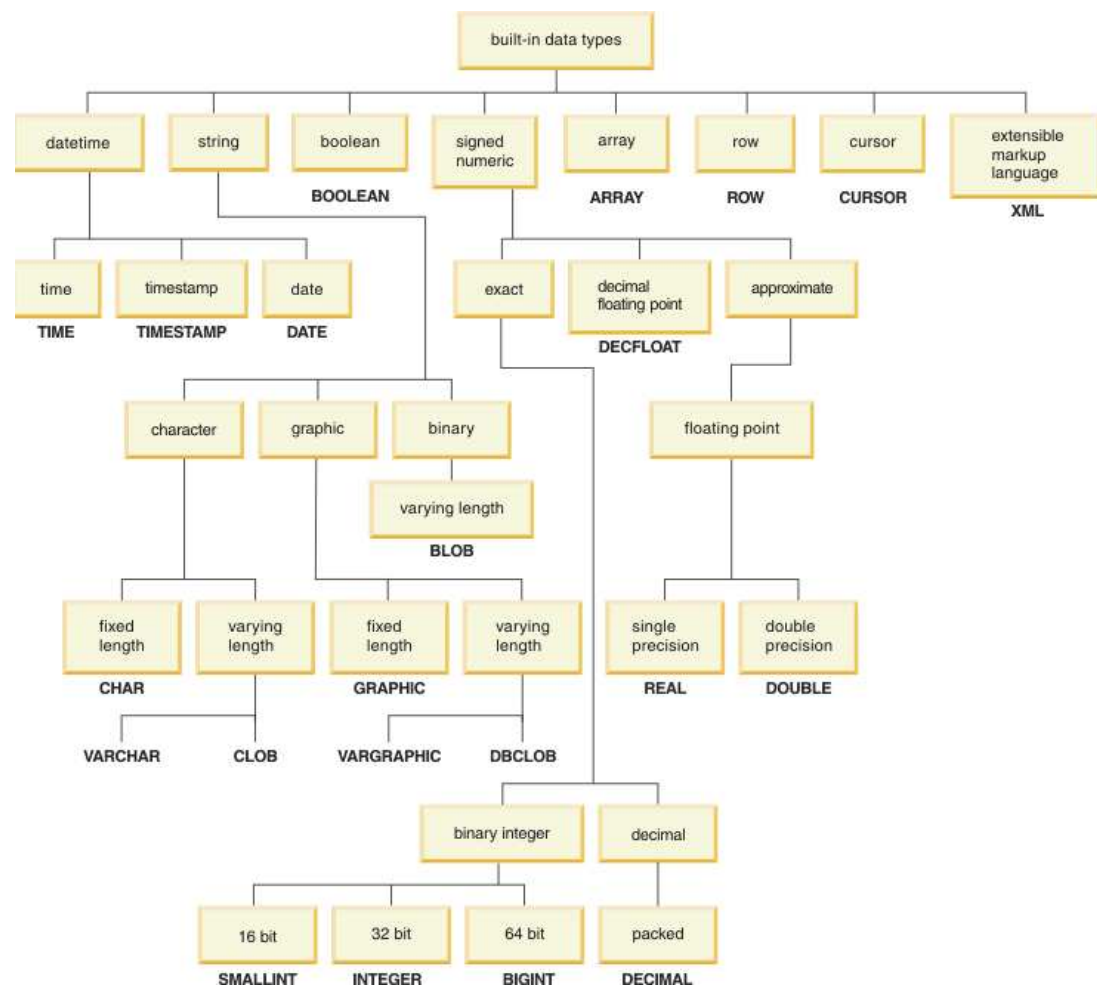
3) To describe a table or view (see the columns and data types) you can do:

db2 describe table employee

4) To describe the structure of a resulting table from a SELECT, issue:

db2 "describe select * from employee"

Data Types



This chart shows the built-in data types of DB2. The XML data type discussed in the pureXML presentation. The book “Getting started with DB2 Express-C explains in more detail these data types.

Choosing the Proper Data Type

It is important to choose the proper data type because this affects performance and disk space. To choose the correct data type, you need to understand how your data will be used and its possible values:

Is your data variable in length, with a maximum length of fewer than 10 characters? Use CHAR

Is your data variable in length, with a minimum length of 10 characters? Use VARCHAR

Is your data fixed in length? Use CHAR

Is your data going to be used in sort operations? Use CHAR, VARCHAR, DECIMAL, INTEGER

Is your data going to be used in arithmetic operations? Use DECIMAL, REAL, DOUBLE, BIGINT, INTEGER, SMALLINT

Does your data require decimals? Use DECIMAL, REAL, DOUBLE, FLOAT

Do you need to store very small amounts of non-traditional data like audio or video? Use CHAR FOR BIT DATA, VARCHAR FOR BIT DATA, LONG VARCHAR FOR BIT DATA

Do you need to store non-traditional data like audio or video, or data larger than a character string can store? Use CLOB, BLOB, DBCLOB

Does the data contain timestamp information? Use TIMESTAMP

Does the data contain time information? TIME

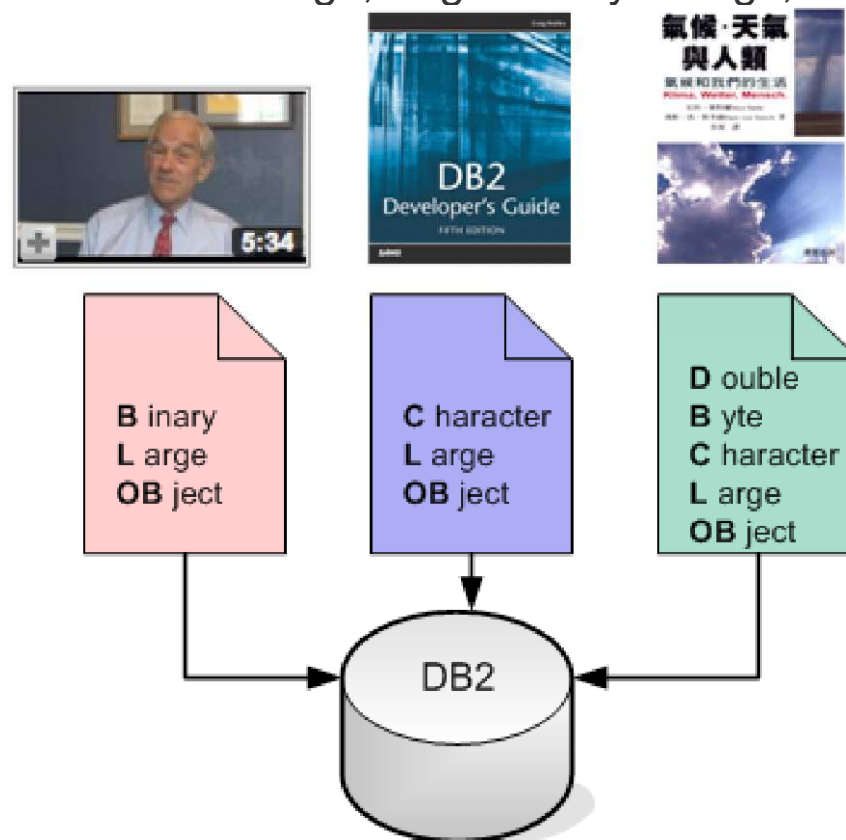
Does the data contain date information? DATE

Do you need a data type to enforce your business rules that has a specific meaning (beyond DB2 built-in data types)? Use a User-defined type

Will you work with XML documents in your database? Use XML

Large Objects

- Store large character strings, large binary strings, or files



We have data types for supporting large objects, such as large binary files (for video/audio). The data types are BLOB, CLOB and DBCLOB.

BLOBs store variable-length data in binary format and are ideal for storing video or audio information in your database. This data type has some restrictions; for example, you cannot sort by this type of column.

CLOBs store large amounts of variable-length single-byte character set (SBCS) or multi-byte character set (MBCS) character strings, for example, large amounts of text such as white papers or long documents.

DBCLOBs store large amounts of variable-length double-byte character set (DBCS) character strings, such as large amounts of text in Chinese.

Similar to LONG VARCHAR and LONG VARGRAPHIC data types, LOBs are accessed directly from disk without going through the buffer pool, so using LOBs is slower than using other data types. In addition, because changes to a database are logged in transaction log files, these files might get filled quickly when modifying a LOB column. To prevent this from happening, the **CREATE TABLE** statement has the **NOT LOGGED** option for LOB columns. For LOB columns defined as more than 1GB in size, **NOT LOGGED** is required.

The **CREATE TABLE** statement also has the **COMPACT** option for LOBs to allocate just the necessary disk space. However, if you perform an update to the LOB column that would increase the size of the LOB, DB2 would need to allocate more space at that time, which incurs a performance penalty. Note that this option does not compress the LOBs.

NOTE

Do not use LOBs to store data less than 32K in size. Instead, use VARCHAR or VARCHAR FOR BIT DATA, which can hold a maximum of 32,672 bytes. This will help with database performance.

Nulls

- A NULL represents an unknown state
 - Use NOT NULL on a column definition when that column must contain a known data value.
- Specifying a default value for NOT NULL columns may be useful

```
CREATE TABLE Staff (  
    ID          SMALLINT NOT NULL,  
    NAME        VARCHAR(9) ,  
    DEPT        SMALLINT NOT NULL with default 10,  
    JOB         CHAR(5) ,  
    YEARS       SMALLINT ,  
    SALARY      DECIMAL(7,2) ,  
    COMM        DECIMAL(7,2) with default 15  
)
```

Comparing to Oracle's behavior:

Say col1 has value of NULL

'xyz' || 'abc' || col1 ==> 'xyzabc'

In DB2

'xyz' || 'abc' || col1 ==> NULL (ANSI standard)

System Catalog Tables

- Each database has its own system catalog tables/views
- They store *metadata* about the *database* objects
- You can query these tables just like any other tables
- Reside in three schemas:
 - **SYSIBM** - base tables, optimized for DB2
 - **SYSCAT** - views based on SYSIBM tables, optimized for ease of use
 - **SYSSTAT** - database statistics

Examples:

- SYSCAT.TABLES, SYSCAT.INDEXES, SYSCAT.COLUMNS, SYSCAT.FUNCTIONS, SYSCAT.PROCEDURES

DB2 automatically creates system catalog tables when a database is created. They always reside in the SYSCATSPACE table space. System catalog tables contain information about all the objects in the database. For example, when you create a table space, its information will be stored into one or more system catalog tables. When this table space is referenced during a later operation, DB2 checks the corresponding system catalog tables to see whether the table space exists and whether the operation is allowed. Without the system catalog tables, DB2 will not be able to function.

Some of the information contained in system catalog tables includes the following:

- Definitions of all database objects
- Column data types of the tables and views
- Defined constraints
- Object privileges
- Object dependencies

System catalog tables or views use the SYSIBM, SYSCAT, or SYSSTAT schemas.

The **SYSIBM schema** is used for the base system catalog tables.

The **SYSCAT schema** is used for views defined on the system catalog tables. DB2 users should normally query the SYSCAT views rather than the SYSIBM tables for information.

The **SYSSTAT schema** is used for views containing information about database statistics and is also based on the system catalog tables.

Although you cannot update the tables and views residing under the SYSIBM and SYSCAT schemas, you can update the views under the SYSSTAT schema. Updating these views can sometimes influence the DB2 optimizer to choose a different access path.

User temporary tables

- Tables in memory created by an application
- Good for performance
- Require a USER temporary table space
- Two types:
 - Declared global temporary (DGTT)

```
DECLARE GLOBAL TEMPORARY TABLE mydgtt  
    (col1 int, col2 varchar(10))
```

- Create global temporary (CGTT)

```
CREATE GLOBAL TEMPORARY TABLE mycgtt  
    (col1 int, col2 varchar(10))
```

Temporary tables can be classified as system or user tables. DB2 manages system temporary tables in the system temporary table space. DB2 creates and drops these tables automatically. Since users don't have control over system temporary tables, we don't discuss them any further.

You create user temporary tables inside a user temporary table space. For example, the following statement creates a user temporary table space called *usrtmp4k*.

```
CREATE USER TEMPORARY TABLESPACE usrtmp4k  
    MANAGED BY SYSTEM USING ('C:\usrtmp')
```

User temporary tables, referred to as temporary tables from here on, store temporary data, that is, data that will be destroyed after a session or when a connection ends. Temporary tables are typically used in situations where you need to compute a large result set from an operation, and you need to store the result set temporarily to continue with further processing. Though transaction logging is allowed with temporary tables, most users don't need to log temporary data. In fact, not having transaction logging for this type of table improves performance. Temporary tables exist only for one connection; therefore, there are no concurrency or locking issues. To create a temporary table, use the **DECLARE** statement or the **CREATE GLOBAL TEMPORARY** statement. The difference between these two types is that in the first, the table DDL is also gone once the temp table is deleted; while for a CGTT, the table structure remains, only that the rows are deleted.

DB2 uses the schema *session* for all temporary tables regardless of the user ID connected to the database.

Views

- Virtual table derived from other tables or views
- Populated when invoked, based on a SELECT statement
- Some views can be updatable, others are read-only

```
CONNECT TO MYDB1
```

```
CREATE VIEW MYVIEW1
  AS SELECT ARTNO, NAME, CLASSIFICATION
  FROM ARTISTS
```

```
SELECT * FROM MYVIEW1
```

ARTNO	NAME	CLASSIFICATION
-----	-----	-----
10	HUMAN	A
20	MY PLANT	C
30	THE STORE	E
...		

A **view** is a virtual table derived from one or more tables or other views. It is virtual because it does not contain any data, but a definition of a table based on the result of a **SELECT** statement.

A view does not need to contain all the columns of the base table. Its columns do not need to have the same names as the base table, either. You can use views to hide confidential information from users.

Views can be updateable, deleteable, insertable, or read-only. There are several rules that need to be followed when creating a view to consider it to fall into any of these classifications. For updateable, insertable and deleteable views, what occurs behind the scenes is that the base table is what is really being changed (updated/inserted/deleted)

Synonyms / Aliases

- Alternate name for a table or view
- A synonym is also known as an alias
- Synonyms can be private or public
 - Private synonyms have a schema_name as other db objects

```
CREATE SYNONYM empinfo FOR employees
```

- Public synonyms do not have a schema_name

```
CREATE PUBLIC SYNONYM empinfo FOR employees
```

An alias, also known as a synonym provides another name to a database object. In the case of **PUBLIC** SYNONYMS, you can create this synonym without any schema required (not even implicit). It's the only DB2 object where its name doesn't have to have two parts (schema name. Object name). It can just be "Object.name"

For example:

```
connect to sample user arfchong using mypsw
create public synonym raul for table arfchong.staff
select * from raul
select * from arfchong.raul ## Error
connect to sample user db2admin using psw
select * from raul
```

In the example, you first connect with user arfchong and create the public synonym raul which references the table arfchong.staff. The synonym itself doesn't use a schema. If you try using one, you will receive an error. Other users like db2admin in the example can also use synonym raul which is public.

If the keyword PUBLIC is not used, the synonym created would be a private synonym.

Indexes

- Ordered set of pointers that refer to rows in a base table.
- They are based upon one or more columns but stored as a separate entity.

DEPTID	ROW
A000	5
B001	2
C001	8
D001	11
E001	3
E002	6
E003	4
F001	1
F002	9
F003	7
G010	10

DEPTID	DEPTNAME	COSTCENTER
F001	ADMINISTRATION	10250
B001	PLANNING	10820
E001	ACCOUNTING	20450
E003	HUMAN RESOURCES	30200
A000	R & D	50120
E002	MANUFACTURING	50220
F003	OPERATIONS	50230
C001	MARKETING	42100
F002	SALES	42200
G010	CUSTOMER SUPPORT	42300
D001	LEGAL	60680

Row 1 →
Row 2 →
Row 3 →
Row 4 →
Row 5 →
Row 6 →
Row 7 →
Row 8 →
Row 9 →
Row 10 →
Row 11 →

Indexes are database objects that are built based on one or more columns of a table. They are used for two main reasons:

- To improve query performance. Indexes can be used to access the data faster using direct access to rows based on the index key values
- To guarantee uniqueness when they are defined as unique indexes

A **clustering index** is created so that the index pages physically map to the data pages. That is, all the records that have the same index key are physically close together.

Indexes

▪ Good for performance and to guarantee uniqueness

▪ Index Characteristics:

- ascending or descending
- unique or non-unique
- compound
- cluster
- bi-directional (default behavior)

▪ Examples:

```
create unique index artno_ix on artists (artno)
```

Indexes can be ascending or descending (Think of an index in a book where book titles are organized normally in ascending alphabetic order).

A unique index is used to uniquely identify a row in a table. A non-unique index will not uniquely identify a row in a table, but still helps with performance.

A compound index consists of several columns that form the index

A cluster index is where the index data is physically stored in a way that is close to its corresponding data. This can help in performance.

A bidirectional index is one that can be traversed in either direction (so it's ascending or descending)

Sequence objects

- **Generates a unique numeric value**
- **Used at the database level, independent of tables**
- **Example:**

```
CREATE SEQUENCE myseq  
  START WITH 1  
  INCREMENT BY 1  
  CACHE 5
```

```
INSERT INTO t1 VALUES (nextval for myseq, ...)
```

```
SELECT prevval for myseq FROM sysibm.sysdummy1
```

A **sequence** is a database object that allows automatic generation of value. Unlike identity columns, this object does not depend on any table—the same sequence object can be used across the database. To create a sequence, use the **CREATE SEQUENCE** statement as demonstrated here.

```
CREATE SEQUENCE myseq AS INTEGER  
  START WITH 1 INCREMENT BY 1  
  NO MAXVALUE  
  NO CYCLE  
  CACHE 5
```

This statement creates the sequence *myseq*, which is of type **INTEGER**. The sequence starts with a value of 1 and then increases by 1 each time it's invoked for the next value.

The **NO MAXVALUE** clause indicates there is no explicit maximum value in which the sequence will stop; therefore, it will be bound by the limit of the data type, in this case, **INTEGER**.

The **NO CYCLE** clause indicates the sequence will not start over from the beginning once the limit is reached.

CACHE 5 indicates five sequence numbers will be cached in memory, and the sixth number in the sequence would be stored in a catalog table. Sequence numbers are cached in memory for performance reasons; otherwise, DB2 needs to access the catalog tables constantly to retrieve the next value in line.

What would happen if your computer crashed and the following numbers were in the cache: 5, 6, 7, 8, and 9? These numbers would be lost, and the next time DB2 needed to retrieve a number, it would obtain the number from the catalog tables. In this example, 10 is the next number to be generated. If you are using the sequence number to generate unique identifiers, which must be in sequence with no gaps allowed, this would not work for you. The solution would be to use the **NO CACHE** clause to guarantee sequentially generated numbers with no gaps, but you will pay a performance cost.

For the sequence value, you can use any exact numeric data type with a scale of zero, including **SMALLINT**, **INTEGER**, **BIGINT**, and **DECIMAL**. In addition, any user-defined distinct type based on these data types can hold sequence values.

Constraints

- Constraints allow you to define rules for the data in your table.
- There are different types of constraints: (with suggested prefixes to use for the names):
 - PRIMARY KEY: `_pk`
 - UNIQUE: `_uq`
 - DEFAULT: `_df`
 - CHECK: `_ck`
 - FOREIGN KEY: `_fk`

Constraints allow you to define rules for the data in your table. There are different types of constraints:

A UNIQUE constraint prevents duplicate values in a table. This is implemented using unique indexes and is specified in the CREATE TABLE statement using the keyword UNIQUE.

A NULL is part of the UNIQUE data values domain. A PRIMARY KEY constraint is similar to a UNIQUE constraint, however it excludes NULL

as valid data. Primary keys always have an index associated with it.

A REFERENTIAL constraint is used to support referential integrity which allows you to manage relationships between tables.

A CHECK constraint ensures the values you enter into a column are within the rules specified in the constraint.

Constraints - Example

```
CREATE TABLE EMPLOYEE
(
  ID integer NOT NULL CONSTRAINT ID_pk PRIMARY KEY,
  NAME varchar(9),
  DEPT smallint CONSTRAINT dept_ck1
      CHECK (DEPT BETWEEN 10 AND 100),
  JOB char(5) CONSTRAINT dept_ck2
      CHECK (JOB IN ('Sales', 'Mgr', 'Clerk')),
  HIREDATE date,
  SALARY decimal(7,2),
  CONSTRAINT yearsal_ck
      CHECK (YEAR(HIREDATE) > 1986 OR SALARY > 40500)
)
```

The following example shows a table definition with several CHECK constraints and a PRIMARY KEY defined. For this table, four constraints should be satisfied before any data can be inserted into the table.

These constraints are:

- PRIMARY KEY constraint on the column ID This means that no duplicate values or nulls can be inserted.
- CHECK constraint on DEPT column. Only allows to insert data if the values are between 10 and 100.
- CHECK constraint on JOB column. Only allows to insert data if the values are 'Sales', 'Mgr' or 'Clerk'.
- CHECK constraint on the combination of the HIREDATE and SALARY columns. Only allows to insert data if the hire date year is greater than 1986 and the SALARY is greater than 40500.

Referential Integrity

DEPARTMENT table (Parent table)

DEPTNO (Primary key) or unique constraint	DEPTNAME	MGRNO
---	----------	-------

EMPLOYEE table (Dependent table)

EMPNO (Primary key)	FIRSTNAME	LASTNAME	WORKDEPT (Foreign key)	PHONENO
------------------------	-----------	----------	---------------------------	---------

```
create table employee (empno .....
    primary key (empno)
    foreign key (workdept)
    references department on delete restrict)
in DMS01
```

Referential integrity (RI) establishes relationships between tables. Using a combination of primary keys and foreign keys, it can enforce the validity of your data. Referential integrity reduces application code complexity by eliminating the need to place data level referential validation at the application level. A table whose column values depend on the values of other tables is called **dependant**, or **child table**; and a table that is being referenced is called the **base** or **parent** table. Only tables that have columns defined as UNIQUE or PRIMARY KEY can be referenced in other tables as foreign keys for referential integrity.

Parent Table:

Controlling data table in which the parent key exists

Dependent Table:

Dependent on the data in the parent table. It also contains a foreign key. For a row to exist in a dependent table, a matching row must first exist within a parent table

Primary Key:

Cannot contain NULL values and values must be unique. A primary key consists of one or more columns within a table

Foreign Key:

One or more columns in the dependant table used to reference columns in the parent table that are the primary key or unique constraint in the parent table. In the syntax, as shown in the example, you may not need to put the column names of the parent table. DB2 will automatically know.

Note in the syntax used that after “department” there is no need to specify the column (deptno) that is used to establish the RI. DB2 automatically can detect this. However, as we will see next, you can use other syntaxes that do indicate the name of the column in the parent table.

Referential Integrity – rules for deletion or updates

```
create table employee (empno ...
... references department on delete restrict)
```

- When a row is deleted or updated in a parent table, what happens to a row in the dependant tables?
- Use the clause “*on delete <behavior>*” or “*on update <behavior>*” when defining the RI to set the behavior to use:

Behavior	Description
NO ACTION	Do not allow the delete or update to happen on parent row (enforce <i>after</i> other referential constraints). This is the default.
RESTRICT	Do not allow the delete or update to happen on parent row (enforce <i>before</i> other referential constraints)
CASCADE	Cascade delete or update of parent row to dependant rows
SET NULL	Set all dependant rows to NULL

The use of NO ACTION or RESTRICT as delete or update rules for referential constraints determines when the constraint is enforced. A delete or update rule of RESTRICT is enforced before all other constraints, including those referential constraints with modifying rules such as CASCADE or SET NULL. A delete or update rule of NO ACTION is enforced after other referential constraints. One example where different behavior is evident involves the deletion of rows from a view that is defined as a UNION ALL of related tables.

Table T1 is a parent of table T3; delete rule as noted below.

Table T2 is a parent of table T3; delete rule CASCADE.

```
CREATE VIEW V1 AS SELECT * FROM T1 UNION ALL SELECT * FROM T2
DELETE FROM V1
```

If table T1 is a parent of table T3 with a delete rule of RESTRICT, a restrict violation will be raised (SQLSTATE 23001) if there are any child rows for parent keys of T1 in T3.

If table T1 is a parent of table T3 with a delete rule of NO ACTION, the child rows may be deleted by the delete rule of CASCADE when deleting rows from T2 before the NO ACTION delete rule is enforced for the deletes from T1. If deletes from T2 did not result in deleting all child rows for parent keys of T1 in T3, then a constraint violation will be raised (SQLSTATE 23504).

Note that the SQLSTATE returned is different depending on whether the delete or update rule is RESTRICT or NO ACTION.

Referential Integrity – Syntax 1

```
CREATE TABLE DEPENDANT_TABLE  
(  
    ID integer CONSTRAINT id_fk  
        REFERENCES BASE_TABLE(UNIQUE_OR_PRIMARY_KEY),  
    NAME varchar(9),  
    ...  
);
```

Using this syntax, the constraint is defined with the column used for the constraint, in this case the column is “ID”

Referential Integrity – Syntax 2

```
CREATE TABLE DEPENDANT_TABLE
(
    ID integer,
    NAME varchar(9),
    ...,
    CONSTRAINT constraint_name FOREIGN KEY (ID)
    REFERENCES BASE_TABLE(UNIQUE_OR_PRIMARY_KEY)
);
```

Using this syntax, the constraint is defined after all columns are defined.

Referential Integrity – Syntax 3

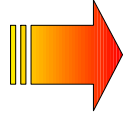
```
CREATE TABLE DEPENDANT_TABLE  
(  
    ID INTEGER,  
    NAME VARCHAR(9),  
    ...  
);
```

```
ALTER TABLE DEPENDANT_TABLE  
    ADD CONSTRAINT constraint_name FOREIGN KEY (ID)  
    REFERENCES BASE_TABLE(UNIQUE_OR_PRIMARY_KEY);
```

Using this syntax, the constraint is defined after the table has been created.

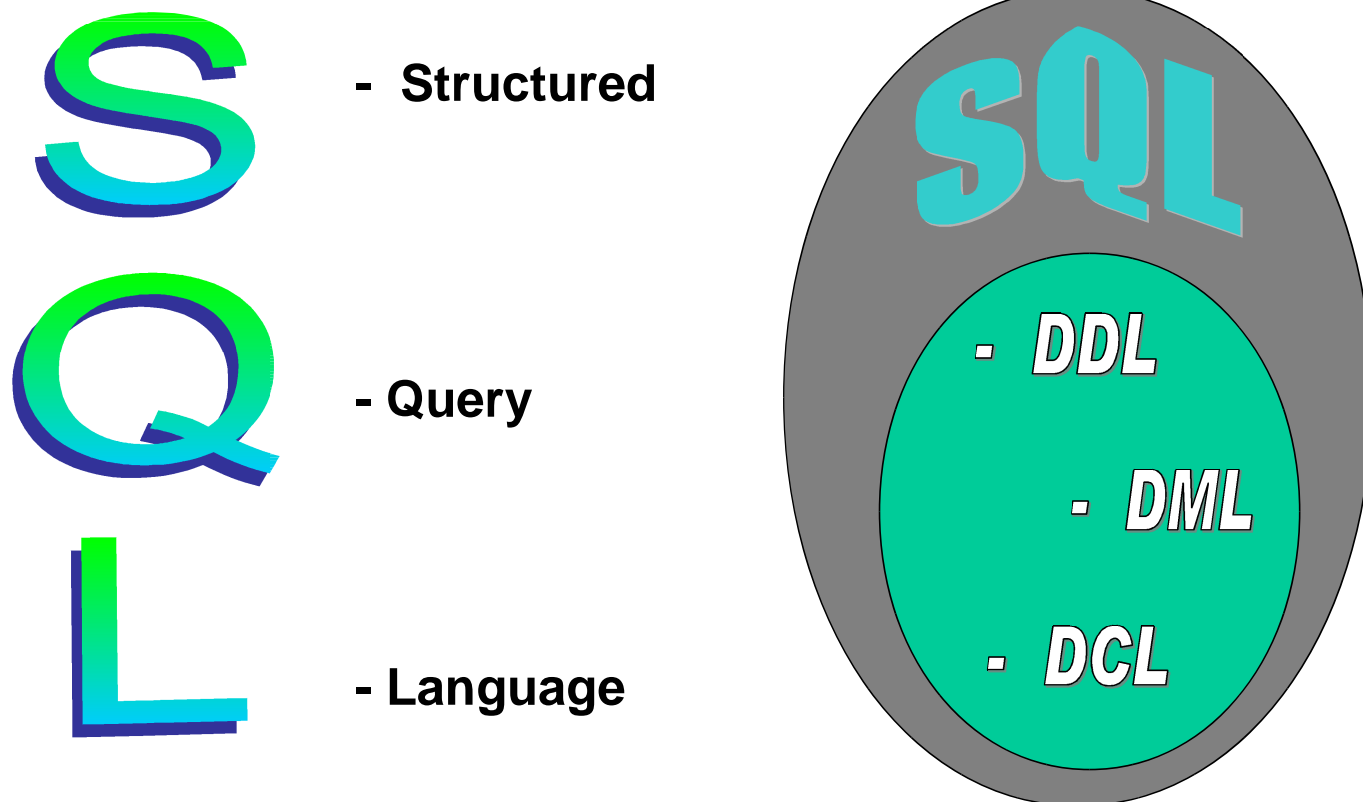
In the above sample code, if the constraint **name** is not specified, the DB2 system will generate the name automatically. This generated string is 15 characters long, for example 'CC1288717696656'.

Agenda



- Overview
- Database objects
- **SQL introduction**
- The SELECT statement
- The UPDATE, INSERT and DELETE statements
- Working with JOINS

Structured Query Language - SQL



SQL - Structured Query Language - is the standard relational database language. Structured Query Language is a high-level language that allows users to manipulate relational data. One of the strengths of SQL is that users need only specify the information they need without having to know how to retrieve it. The database management system is responsible for developing the access path needed to retrieve the data. SQL works at a set level, meaning that it is designed to retrieve rows of one or more tables.

SQL has three categories based on the functionality involved:

DDL – Data definition language used to define, change, or drop database objects

DML – Data manipulation language used to read and modify data

DCL – Data control language used to grant and revoke authorizations

Some people also talk of TCL (Transaction Control language) with statements such as COMMIT, ROLLBACK, SAVEPOINT which group DML statements together to be executed as a unit

Data Definition Language(DDL)

- Creates, modifies, deletes database objects
 - Statements: **CREATE**, **ALTER**, **DROP**, **DECLARE**
- **Examples:**

```
CREATE TABLE <table name> ...  
CREATE SCHEMA <schema name>...  
ALTER TABLE <table name>...  
ALTER INDEX <index name>...  
DROP TABLE <table name>...  
DROP INDEX <index name>...  
DECLARE GLOBAL TEMPORARY TABLE <table name>...
```

Data Definition Language(DDL)

- **Other examples:**

```
ALTER TABLE myTable  
    ALTER COLUMN col1 set not null
```

```
RENAME <object type> <object name> to <new name>
```

```
ALTER TABLE <table name>  
    RENAME COLUMN <column name> TO <new name>
```

Once a database object is created, it may be necessary to change its properties to suit changing business requirements. Dropping and recreating the object is one way to achieve this modification; however, dropping the object has severe side effects. A better way to modify database objects is to use the ALTER SQL statement. For example, assuming you would like to change a table definition so that NULLs are not allowed for a given column, you can try this SQL statement:

ALTER TABLE myTable ALTER COLUMN col1 SET NOT NULL

Similarly, other modifications to the table like adding or dropping a column, defining or dropping a PRIMARY KEY, and so on, can be achieved using the appropriate alter table syntax. The ALTER statement can also be used with other database objects.

Data Manipulation Language (DML)

- Retrieves, inserts, updates, and deletes database objects
- Statements: **SELECT**, **INSERT**, **UPDATE**, **DELETE**
- Examples:

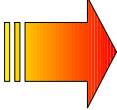
```
SELECT * FROM employee
```

```
INSERT INTO EMPLOYEE (name) values ('Peter')
```

```
UPDATE EMPLOYEE set name = 'Paul'
```

```
DELETE FROM employee
```

Agenda

- Overview
- Database objects
- SQL introduction
-  ▪ **The SELECT statement**
- The UPDATE, INSERT and DELETE statements
- Working with JOINS

The SELECT statement

select * **from** <tablename>

Select statement: Query

Result from the query: Result set/table

The main purpose of a database management system is not just to store data, but also facilitate retrieval of the same.

So now, how do we see the data that we have stored in a table?

That's when select statements come into picture.

A select statement in its simplest form is

`select * from <tablename>`

Here the select statement is termed as the query and the output we get from executing this query is called a result set or a result table.

Example: select * from book

```
db2=> select * from book
```

BOOK_ID	TITLE	EDITION	YEAR	PRICE	ISBN	PAGES	AISLE	DESCRIPTION
B1	Getting st	3	2009	24.99	978-0-	280	DB-A01	Teaches you the esse
B2	Database F	1	2010	24.99	978-0-	300	DB-A02	Teaches you the fund
B3	Getting st	1	2010	24.99	978-0-	298	DB-A01	Teaches you the esse
B4	Getting st	1	2010	24.99	978-0-	278	DB-A01	Teaches you the esse

4 record(s) selected.

Let us take table book for our example here.

select * from book”

when executed from the command prompt would give out the result set like this.

You can also use IBM Data Studio for executing these select statements.

For better viewing, we have used the command prompt outputs.

Make a note that this is not the exact output for this particular query since we have truncated the result set for display purpose.

“*” here refers to all the columns that are there in the table.

As you can see, all the data rows for all the columns that are in the table book are displayed.

Example:

`select <column 1>, ... , <column n> from <tablename>`

```
db2=> select book_id, title, edition, year,
price, isbn, pages, aisle, description from
book
```

BOOK_ID	TITLE	EDITION	YEAR	PRICE	ISBN	PAGES	AISLE	DESCRIPTION
B1	Getting st	3	2009	24.99	978-0-	280	DB-A01	Teaches you the esse
B2	Database F	1	2010	24.99	978-0-	300	DB-A02	Teaches you the fund
B3	Getting st	1	2010	24.99	978-0-	298	DB-A01	Teaches you the esse
B4	Getting st	1	2010	24.99	978-0-	278	DB-A01	Teaches you the esse

4 record(s) selected.

We can retrieve data for all the columns by specifying the column names individually in our select statement too.

Here is an example of that.

The result set is same as our previous example but we are using the column names here.

Projecting columns from a table

select <column 1>, <column 2> from book

```
db2 => select book_id, title from book
```

```
BOOK_ID  TITLE
```

```
-----  
B1       Getting started with DB2 Express-C  
B2       Database Fundamentals  
B3       Getting started with DB2 App Dev  
B4       Getting started with WAS CE
```

```
4 record(s) selected.
```

Relational databases allow retrieving a subset of columns from a table.

In this example, we are retrieving data from just two columns `book_id` and `title` from the table `book`.

```
select book_id, title from book
```


Changing the order of the columns

select <column 2>, <column 1> from book

```
db2 => select title, book_id from book
```

TITLE	BOOK_ID
Getting started with DB2 Express-C	B1
Database Fundamentals	B2
Getting started with DB2 App Dev	B3
Getting started with WAS CE	B4

4 record(s) selected.

Let's see what happens if the order of the columns are interchanged as in this example.

Select title, book_id from book.

As you can see in the result set, order of the columns will always match the order that is passed to the select statement.

Order as defined in the create table statement is ignored.

Restricting rows from a table

select book_id, title from book *WHERE predicate*

```
db2 => select book_id, title from book
        WHERE book_id='B1'

BOOK_ID  TITLE
-----
B1       Getting started with DB2 Express-C

1 record(s) selected.
```

We are now quite familiar with this query, right?

```
select book_id, title from book
```

Let's assume we have a need to restrict the result set.

For example, we need to know the title of the book whose book_id is B1.

Relational operation helps us in restricting the result set by allowing us to use a clause "where".

Where" always requires a predicate.

A predicate is a condition that evaluates to true, false or unknown.

So for our need, we shall use the where clause with the predicate book_id='B1'.

Take a look at the query that we have used.

The result set is restricted to just one row whose condition evaluates to true.

Restricting rows from a table (Continued)

- **Comparison operators**
 - Equals to =
 - Greater than >
 - Lesser than <
 - Greater than or equal to >=
 - Less than or equal to <=
 - Not equal to <>

We saw an example which used the comparison operator “equals to”

We have other comparison operators that are supported by the relational database management system.

Here is a list of them. Equals to, greater than, lesser than

Greater than or equal to, Less than or equal to, not equal to

You can try out these different operators yourself to know how they work.

Limiting Result Set Size

select book_id, title from book

FETCH FIRST <n> ROWS ONLY

```
db2 => select book_id, title from book
        FETCH FIRST 2 ROWS ONLY
```

```
BOOK_ID  TITLE
-----
```

```
B1       Getting started with DB2 Express-C
```

```
B2       Database Fundamentals
```

```
2 record(s) selected.
```

Relational databases also allow us to limit the number of rows we want to view although the result set may have many records to return.

We use “fetch first <n> rows only”

This is a clause that restricts the result set.

The example,

```
select book_id, title from book FETCH FIRST 2 ROWS ONLY
```

will restrict the result set to just 2 rows.

Restricting rows using multiple conditions

select book_id, title from book where

predicate_1 AND predicate_2

```
db2 => select book_id, title from book
where book_id like 'B%' AND title like
'Getting%'
```

```
BOOK_ID  TITLE
-----  -
```

```
B1      Getting started with DB2 Express-C
B3      Getting started with DB2 App Dev
B4      Getting started with WAS CE
```

```
3 record(s) selected.
```

We can use Boolean operator AND to further restrict the rows by supplying multiple conditions.

This AND operator needs two predicates and rows satisfying both predicates are called qualifying rows.

Note that the order of the predicate does not affect the result set.

```
select book_id, title from book where book_id like
'B%' and title like 'Getting%'.
```

You might have noticed the character % here in this example

We shall revisit this particular character in the next few slides.

Selecting from multiple tables

`select <column 1>,<column 2> from <table 1>, <table 2>`

```
db2 => select author_id,title from
        book, author_list
```

```
AUTHOR_ID  TITLE
```

```
-----
A1         Getting started with DB2 Express-C
A2         Getting started with DB2 Express-C
A3         Getting started with DB2 Express-C
A4         Getting started with DB2 Express-C
...
A17        Getting started with WAS CE
A19        Getting started with WAS CE
A20        Getting started with WAS CE
```

```
80 record(s) selected.
```

Suppose we have a need to get the author's lastname and firstname along with the title of the book. Can you think of a straight forward query to retrieve the data we want? There isn't a simple select query which can accomplish this requirement. The reason behind this is, table book contains details only about the book and table author contains details only about the authors. In order to get the title of the book along with the authors who authored particular book, we would need to call both tables in one single query.

Let us see a simple example of retrieving data using two tables. Let us take table author_list which contains 20 rows and book table which contains 4 rows. Notice the end result from the result set.

```
select author_id, title from book, author_list
```

It displays 80 records selected.

Wondering how did so many rows get projected?

It is the direct product of two sets. 20 rows from author_list multiplied by 4 rows from book which turns out to be 80 rows in total.

This behaviour is called the Cartesian Product. Again notice that this output is

truncated for this display purpose. But this is not we intended to see when we thought of selecting from multiple tables right? We can use the join conditions

conditions to narrow down our search and get meaningful results.

Selecting from multiple tables (Continued)

```
db2 => select title, lastname, firstname from book,
author_list, author where
        book.book_id=author_list.book_id and
author_list.author_id=author.author_id and
        book.book_id='B1'
```

TITLE	LASTNAME	FIRSTNAME
Getting started with	CHONG	RAUL
Getting started with	AHUJA	RAV
Getting started with	HAKES	IAN

3 record(s) selected.

In this example, the result set is very much meaningful which displays the title of the book and who all have authored it. But take a look at the query.

It looks a little complicated right?

Not when you actually know when and how to use multiple tables.

While using multiple tables to retrieve data, join conditions play a major role.

As you can see in the query used in this example, we are telling the database to search rows whose book_id in book table matches the book_id in author_list table and author_id in author_list table matches that of author table.

We have also passed one more condition that is book_id should be B1 only.

For display purpose, the result set data values are truncated here

Was it a little confusing to say book_id in book table, author_id in author table etc?

It can be confusing while composing a query as well. Hence for that reason, we have correlation names. They are just aliases for a table that is being referred to in a SQL statement.

These correlation names immediately follow the table name in the FROM clause.

Correlation Names

```
db2 => select title,lastname,firstname from book
B, author_list AL, author A where
B.book_id=AL.book_id and AL.author_id=A.author_id
and B.book_id ='B1'
```

TITLE	LASTNAME	FIRSTNAME
-----	-----	-----
Getting started with	CHONG	RAUL
Getting started with	AHUJA	RAV
Getting started with	HAKES	IAN

3 record(s) selected.

We shall re-write our previous query using correlation names.

Just a letter B is being given for table book, AL for author_list table and A for author table.

For the join condition, the columns are qualified with the table name using the correlation names that are provided.

Here again, for the display purpose, the data values in the result set are truncated.

Sorting the result-set

```
db2 => select title from book
```

```
TITLE
```

```
-----  
Getting started with DB2 Express-C  
Database Fundamentals  
Getting started with DB2 App Dev  
Getting started with WAS CE
```

```
4 record(s) selected.
```

Here is a simple result set from the query

“select title from book”.

The order doesn't seem to be proper right? Relational databases provide an option to sort the output. Order by clause is used in a query to sort the result-set by a specified column.

Order By clause

```
db2 => select title from book
        order by title

TITLE
-----
Database Fundamentals
Getting started with DB2 App Dev
Getting started with DB2 Express-C
Getting started with WAS CE

  4 record(s) selected.
```

Take a look at this one: select title from book order by title

We have used order by on the column title to sort it

By default, it sorts in ascending order.

To sort in descending order, we just use the keyword desc.

The result-set would be now sorted according to the column specified and in descending order.

Order By clause (Continued)

```
db2 => select title from book  
        order by title desc
```

TITLE

Getting started with WAS CE
Getting started with DB2 Express-C
Getting started with DB2 App Dev
Database Fundamentals

4 record(s) selected.

Here in this example, title column is sorted in descending order.

Notice the order of the first three records.

First 3 words of the title remain the same.

The sorting starts from the point where the characters are different.

There is another way of specifying which columns need to be sorted

Order By clause (Continued)

```
db2 => select title, pages from book  
        order by 2
```

TITLE	PAGES
-----	-----
Getting started with WAS CE	278
Getting started with DB2 Express-C	280
Getting started with DB2 App Dev	298
Database Fundamentals	300

4 record(s) selected.

This example: select title, pages from book order by 2 indicates the column sequence number in the query for the sorting order.

Instead of specifying the column name pages, # 2, is used.

Column “pages” take the second place in the column list in the query and hence # 2 for the sort orders.

Eliminating Duplicates

```
db2 => select country from author order by 1
COUNTRY
-----
AU
BR
...
CN
CN
...
IN
IN
IN
...
RO
RO

20 record(s) selected.
```

Here is a query which lists down the country the authors belong to.

```
select country from author order by 1
```

There are about 20 authors from different countries and some of them come from the same country as others.

When we list down, there are duplicates. At the moment all we need to know is the country from where the authors come from.

Having duplicate values really does not make any sense.

In such cases, eliminating duplicates is the best possible approach.

To eliminate duplicates, the keyword “distinct” is used.

Distinct clause

```
db2 => select distinct(country) from author

COUNTRY
-----
AU
BR
CA
CN
IN
RO

6 record(s) selected.
```

Take a look at this example which reduces the result set to just 6 rows as compared to 20 in our previous example.

```
select distinct(country) from author
```

We now want to know the number of authors from the same country.

What do you think we need to do now?

Yes, counting them would give us the exact number of authors from each country.

For this purpose, we use the clause – “group by”.

Group by clause

```
db2 => select country, count(country) from  
author group by country
```

```
COUNTRY 2
```

COUNTRY	2
AU	1
BR	1
CA	3
CN	6
IN	6
RO	3

```
6 record(s) selected.
```

A “group by” clause groups a result into subsets

that have matching values for one or more columns.

In our example, countries are grouped and then the count is given.

Observe the column heading for the 2nd column in the result set displayed.

It shows the numeric value 2 since it is the 2nd

column in the result set and it is a derived column

column and not that is something directly available in the table.

Group by clause (Continued)

```
db2 => select country, count(country)
        as count from author group by country
```

```
COUNTRY COUNT
```

```
-----
AU          1
BR          1
CA          3
CN          6
IN          6
RO          3
```

```
6 record(s) selected.
```

We can pass a meaningful name to the query to display in the result set for these kind of derived columns.

Now that we have the count of authors from different countries, we can further restrict the number of rows by passing some conditions.

For instance, we can check if there are more than 4 authors from the same country

To pass a condition to a group by clause, we use the keyword HAVING.

Having clause

```
db2 => select country, count(country) as  
count from author group by country  
having count(country) > 4
```

```
COUNTRY COUNT
```

```
-----
```

```
CN 6
```

```
IN 6
```

```
2 record(s) selected.
```

Having clause is used in combination with the Group By clause.

It is very important to note that “where” clause is for the entire result set whereas “having” clause works only with the “group by” clause.

We can use the following statement to restrict our result set even further.

```
select country, count(country) as count from author  
group by country having count(country) > 4
```

String patterns

```
db2 => select firstname from author  
where firstname like 'R%'
```

```
FIRSTNAME
```

```
-----
```

```
RAUL
```

```
RAV
```

```
2 record(s) selected.
```

How do we retrieve data when we remember only some letters in a data row and not the entire value?

Do you think the search gets difficult?

Not in a relational database.

It offers the use of string patterns which can be used while searching data rows which match such conditions.

Let us take a look at some examples.

```
select firstname from author where firstname like 'R%'
```

The usage of % after the letter R implies that the data begins with an R and followed by any letter or number.

Also, % means there is no limit to the number of characters that follow R.

String patterns (Continued)

```
db2 => select firstname, lastname from  
author where lastname like '_ua%'
```

FIRSTNAME	LASTNAME
Jiang Lin	Quan
Dai	Xuan
Chi Run	Hua

3 record(s) selected.

Another usage of this string pattern is through the character '_'.

This replaces exactly one character as you can see in this example.

```
select firstname, lastname from author
```

```
where lastname like '_ua%'
```

We are searching for the lastname of the authors

which begins with any character

followed by letters ua and anything after that.

We get to see 3 authors coming out of this query.

Example:

```
db2 => select title, pages from book where  
pages >= 290 AND pages <=300
```

TITLE	PAGES
Database Fundamentals	300
Getting started with DB2 App Dev	298

2 record(s) selected.

So far we have searched data records based on
on mathematical operators and string patterns.
Let's assume we have a need to look for books
whose # of pages is more than 290 but less than 300.

Do you agree we can write a query like this?

```
select title, pages from book
```

```
where pages >= 290 AND pages <= 300
```

Of course yes. We are able to retrieve the data that we want.

Searching in Ranges

select <column 1>, <column 2> from <tablename>
where <column n> **BETWEEN** <starting range> **AND**
<ending range>

```
db2 => select title, pages from book where  
pages between 290 and 300
```

TITLE	PAGES
Database Fundamentals	300
Getting started with DB2 App Dev	298

2 record(s) selected.

Let us re-write this query using a new clause between-and.

```
select title, pages from book where pages between 290 and 300
```

The result set is same as our previous query.

Just that this is a better approach of searching in ranges
as the values in the range are inclusive.

Example:

```
db2 => insert into book values ('B5','For  
Example',5,NULL,24.99,'978-0-9866283-1-4',400,'DB-  
A12','Dummy Entry')
```

```
DB20000I The SQL command completed successfully.
```

For introducing the next two operators IS and IS NOT,
I am inserting a single row with the following values into the Book table.
As you have noticed, we are inserting a NULL value into the column year.

Searching for NULL

select <column 1>, <column 2> from <tablename>
where <column n> **IS NULL**

```
db2 => select book_id,title,year from  
book where year IS NULL
```

BOOK_ID	TITLE	YEAR
B5	For Example	-

1 record(s) selected.

We shall now see how to retrieve the data when a particular value is NULL.

Is operator is used for fetching NULL values.

```
select book_id, title year from book where year IS NULL
```

Searching for NOT NULL

select <column 1>, <column 2> from <tablename>
where <column n> **IS NOT NULL**

```
db2 => select book_id,title,year from  
book where year IS NOT NULL
```

BOOK_ID	TITLE	YEAR
B1	Getting started with DB2 E	2009
B2	Database Fundamentals	2010
B3	Getting started with DB2 A	2010
B4	Getting started with WAS C	2010

4 record(s) selected.

Likewise, IS NOT operator is used

for fetching values which are NOT NULL.

```
select book_id, title, year from book where year is NOT NULL
```


Example:

```
db2 => select firstname, lastname, country
from author where country='AU' OR
country='BR'
```

FIRSTNAME	LASTNAME	COUNTRY
-----	-----	-----
Xiqiang	Ji	AU
Juliano	Martins	BR

2 record(s) selected.

There are instances where some data values cannot be grouped under ranges such as country.

If we need to look for authors from countries AU and BR, we will have to use a query like this.

```
select firstname, lastname, country from author where
country='AU' OR country='BR'
```

This was a simple statement which could be used.

Continued...

```
db2 => select firstname, lastname,  
country from author where  
country='CA' or country='IN' or  
country='RO' or country='CN'
```

What if we want to list the authors from CA, IN,
RO and CN countries?

```
select firstname, lastname, country from author where  
country='CA' or country='IN' or country='RO' or country='CN'
```

One needs to be very careful while using this long list of conditions.
Instead, how about using the IN operator?

Searching for a set of values

select <column 1>, <column 2> from <tablename>
where <column n> **IN (a set of values)**

```
db2 => select firstname, lastname,  
country from author where country  
IN ( 'AU' , 'BR' )
```

FIRSTNAME	LASTNAME	COUNTRY
Xiqliang	Ji	AU
Juliano	Martins	BR

2 record(s) selected.

The IN operator allows us to specify multiple values in a where clause.

This operator will take a list of expressions to compare against.

Following statement would return information

on authors who belong to AU and BR.

```
select firstname, lastname, country from author where country IN ('AU','BR')
```

Agenda

- Overview
- Database objects
- SQL introduction
- The SELECT statement
-  ▪ **The UPDATE, INSERT and DELETE statements**
- Working with JOINS

INSERT statements

- Adds new rows to a table or updatable views
 - Inserting row on to updatable views, inserts data into underlying table
- Syntax:

```
INSERT INTO [TableName | Viewname]
<([ColumnName],...)>
VALUES ([Value],...)
```

Once a table has been created, it needs to be populated with data.

By executing INSERT SQL statement, which can work directly with the table or with an updatable view that references the table to be populated we can add rows one at a time.

Syntax of INSERT statement looks like this:

```
INSERT INTO [TableName | ViewName]
```

```
ColumnName, VALUES
```

where INSERT and INTO are the key words

ColumnName identifies one or more columns in a table or a view.

Values can specify one or more data values to be added to the columns in the table or updatable view specified.

The number of values provided in the *values* clause must be equal to the number of column names specified in the column name list.

INSERT statements (continued)

- Example:

```
INSERT INTO AUTHOR  
  (AUTHOR_ID, LASTNAME, FIRSTNAME, EMAIL, CITY, COUNTRY)  
VALUES  
  ( 'A1' , 'CHONG' , 'RAUL' , 'RFC@IBM.COM' , 'TORONTO' , 'CA' )
```

- **VALUES** clause specifies one or more data values to be added to the column(s) in the table or updatable view specified.

Suppose you want to add a record to a base table named AUTHOR which has columns like AUTHOR_ID LASTNAME,FIRSTNAME,EMAIL,CITY,COUNTRY as column names.

An INSERT statement looks like this:

INSERT INTO AUTHOR followed by columns names
and values that need to be inserted into the table.

INSERT statements (continued)

- Syntax:

```
INSERT INTO [TableName | ViewName]
    <([ColumnName],...)>
    [SELECT statement]
```

- Example:

```
INSERT INTO AUTHOR_1
    SELECT AUTHOR_ID, LASTNAME, FIRSTNAME, EMAIL, CITY, COUNTRY
    FROM AUTHOR
    WHERE FIRSTNAME = 'RAUL'
```

This is another syntax for INSERT SQL statement where a SELECT statement is provided in place of the VALUES clause and result of this subselect are assigned to the appropriate columns.

Here is an example. AUTHOR_1 table is populated with the values that are being returned by executing the SELECT statement on table AUTHOR.

INSERT statement (continued)

- Example – Simple INSERT

```
INSERT INTO AUTHOR VALUES  
    ( 'A1' , 'CHONG' , 'RAUL' , 'RFC@IBM.COM.' , 'TORONTO' , 'CA' )
```

- Example – Inserting multiple rows

```
INSERT INTO AUTHOR  
    (AUTHOR_ID, LASTNAME, FIRSTNAME, EMAIL, CITY, COUNTRY)  
VALUES  
    ( 'A1' , 'CHONG' , 'RAUL' , 'RFC@IBM.COM.' , 'TORONTO' , 'CA' ) ,  
    ( 'A2' , 'AHUJA' , 'RAV' , 'RA@IBM.COM.' , 'TORONTO' , 'CA' )
```

Here are a few example of INSERT statements.

Multiple rows can be inserted at a time by specifying each row in values clauses separated by commas.

UPDATE statement

- Modifies the data in a table or a view
- Syntax:

```
UPDATE [TableName | ViewName]
    SET [[ColumnName]=[Value] | NULL | DEFAULT, ...]
<WHERE [Condition]>
```

- Types
 - Searched update
 - Updates one or more rows in a table
 - Positioned update
 - Always embedded in a program – uses cursors

Specific data values present in a table can be altered by executing the UPDATE SQL statements.

Syntax for UPDATE Statement is

UPDATE [TableName | ViewName]

SET [[ColumnName]=[Value]

WHERE [Condition],

where update is a key word, Column identifies the name of one or more columns that contain data values to be modified.

Value identifies one or more data values that are to be used to replace existing value found in the column specified.

Like the INSERT statements the update statements can also use the results of query or subselect.

There are two types of UPDATE statements:

- 1) Searched updates: which is used to update one or more rows in a table.
- 2) Positioned updates: these are always embedded in a program and requires a cursor to be created, opened and positioned on the row to be updated. Here in this presentation we will concentrate on Searched Update.

UPDATE statement (Continued)

- Example

```
UPDATE AUTHOR  
  
    SET  LASTNAME  = 'KATTA'  
  
        FIRSTNAME = 'LAKSHMI'  
  
WHERE AUTHOR_ID = 'A2'
```

- Or

```
UPDATE AUTHOR  
  
    SET  (LASTNAME, FIRSTNAME) = ('KATTA', 'LAKSHMI')  
  
WHERE  AUTHOR_ID = 'A2'
```

Warning: If no WHERE clause is used, all rows will be updated!

In this example you can see, the lastname and the firstname is updated to 'KATTA' and 'LAKSHMI' where AUTHOR_ID is A2.

This is another approach for updating the values.

If you don't specify the WHERE clause then all the rows will be updated to KATTA and LAKSHMI as their last and first names.

UPDATE statement (Continued)

- Example to remove values

```
UPDATE AUTHOR  
    SET COUNTRY = NULL
```

The update statement can also be used to remove values from columns.

This is done by changing the column's current value to NULL.

Note that this is possible only for nullable columns. Here in this example COUNTRY in all rows are removed by setting it to NULL

DELETE statement

- Used to remove ROWS from a table or a view
- Syntax

```
DELETE FROM [TABLEName | ViewName]  
    <WHERE [Condition]>
```

- Types
 - Searched Delete
 - Used to delete one or more rows in a table
 - Positioned Delete
 - Always embedded in a program – uses cursors

DELETE statement is used when one or more complete rows of data needs to be removed.

DELETE statement is also of two types that is

Searched Delete and Positioned Delete

Here we can see the syntax of delete statement

DELETE FROM [TABLEname | ViewName]

followed by WHERE condition.

DELETE statement (Continued)

- Example

```
DELETE FROM AUTHOR  
WHERE AUTHOR_ID IN ( 'A2' , 'A3' )
```

Warning:

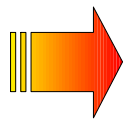
If no WHERE clause is used, all rows will be deleted!

Here you can see simple example where rows having AUTHOR_ID A2 and A3 are removed from the table.

If you don't specify the WHERE clause all the rows in the table will be removed.

Agenda

- Overview
- Database objects
- SQL introduction
- The SELECT statement
- The UPDATE, INSERT and DELETE statements



- **Working with JOINS**

Types of Joins

- **INNER JOIN**
- **LEFT OUTER JOIN (or LEFT JOIN for short)**
- **RIGHT OUTER JOIN (or RIGHT JOIN for short)**
- **FULL OUTER JOIN (or FULL JOIN for short)**
- **CROSS JOIN**

The JOIN operator allows you to combine data from two tables. The JOIN operator is very important when it comes to manipulating data. And it is easy to understand the reason why. When you look at the relational model, you notice that information is “split” into different tables. This is a natural consequence of normalization, a very important concept associated to relational modeling. So, when you try to put information together to create a report, for instance, you usually need to gather data from several tables. And here is where you will use the JOIN operator.

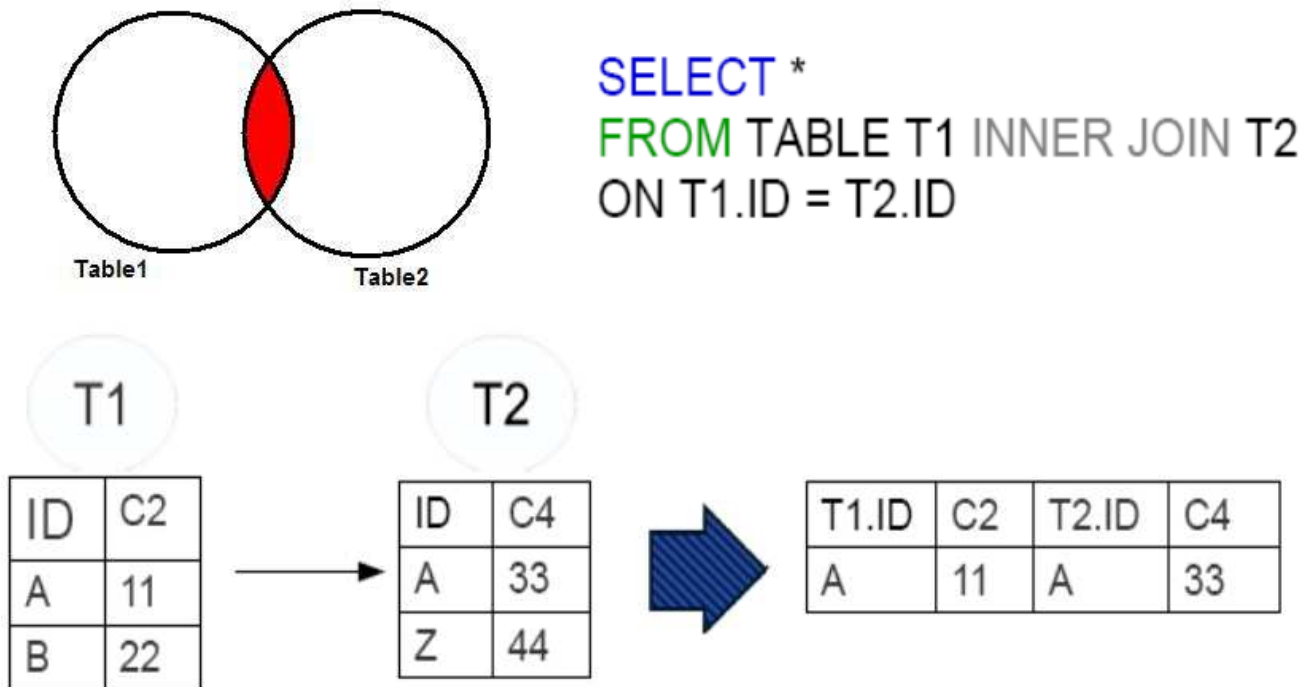
The JOIN operator is part of standard SQL, that is: it works roughly the same way in DB2 or any other RDBMS you find. When you use the JOIN operator you are always combining data from two tables. And the first thing you need to do is identify the relationship between those two tables. Remember we are talking about relational databases, right? In other words, you need to identify the column or columns on each table that should be used as the “link” between those tables.

Next, identify the type of join you want to use. We cover each of the different types of JOINS listed in this slide in more detail soon.

SQL offers you several different types of joins. It all depends on what you are looking for. For instance, you can extract a data set corresponding to the intersection of the two tables involved. Or you can choose a bigger data set. You can go up to the point of selecting the combination of all the data from those two tables.

INNER JOIN

- Result is the intersection of the two tables

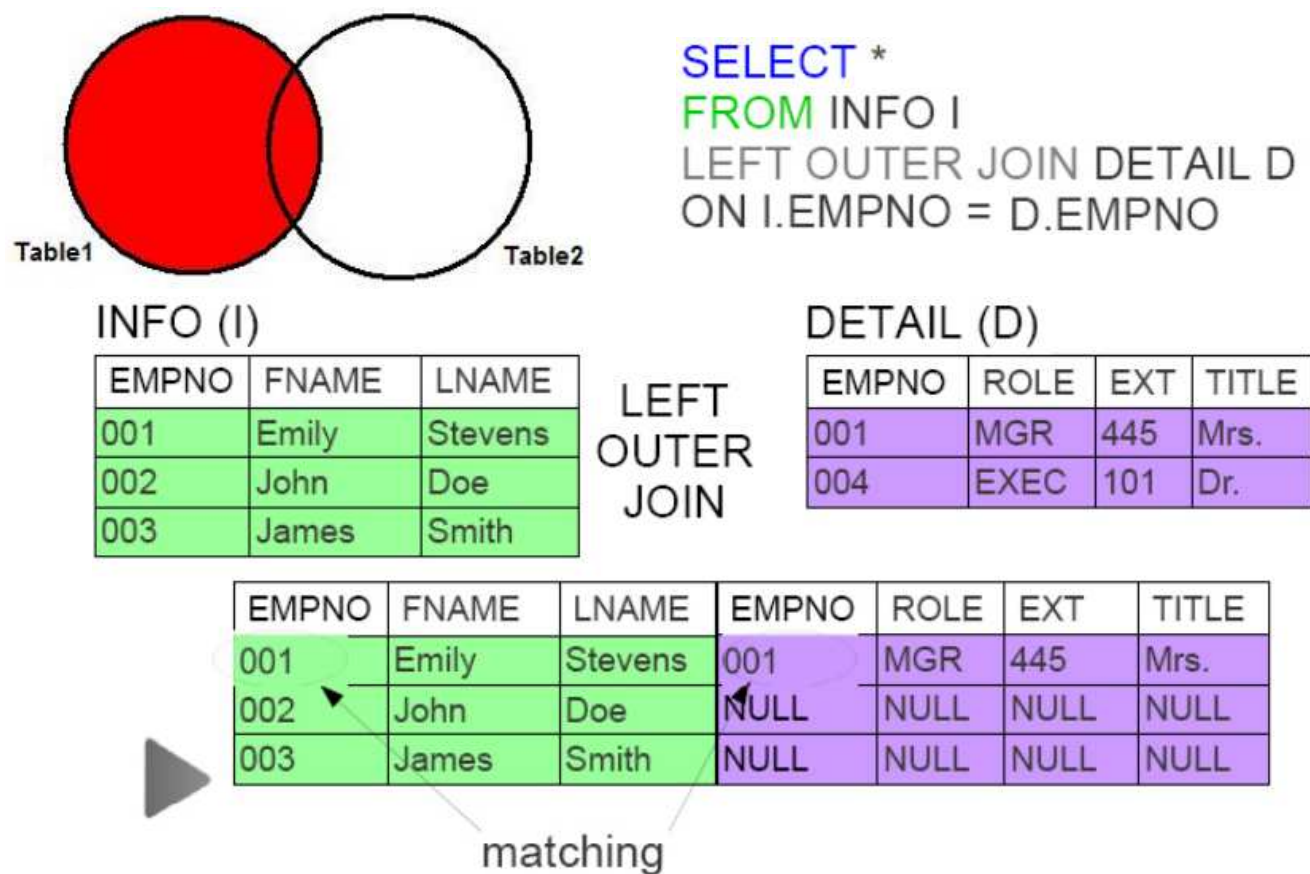


The most common type of join is the inner join. which returns a set of data that represents the intersection of those two tables.

Take a look at the venn diagram representing this join to understand it.

Then you can check the SQL syntax you need to use to do the join.

LEFT JOIN



89

© 2011 IBM Corporation

Now let's talk about the left join operator. The Venn diagram for the left join operator is like this. You bring everything from the left table and just the information from the second table that fits to the other. When we say left table we mean the table to the left of the join operator, in this example, the borrower table. As you can see, the syntax is quite similar to the previous case. We just changed the operator itself.

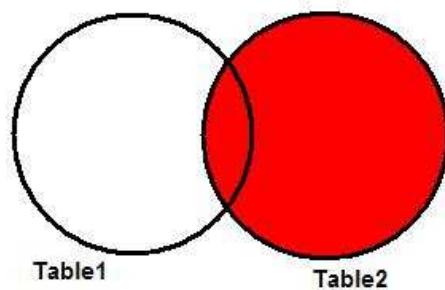
Notice when we run the left join we are picking columns from both tables but some records show values with NULL from the right table. This happens when the right table doesn't have the correspondent key.

When you work with a right join operator you'll see the very opposite effect. You might have no value also but this time the records from the left table will be affected.

To "process" the result set of this join, keep it simple by following these steps:

- 1) Take the entire left table
- 2) Take one row from the right table, and see if there is a matching row based on the column used for the JOIN (in this case, EMPNO)
- 3) if there is a match, put it besides the corresponding column of the left table
- 4) If there is no match, put NULLs, and complete all rows with nulls
- 5) If the question is how many rows you will get at the end, you know you'll be picking all the rows from the left table, so if there is no WHERE clause, then you'd get at least the same number of rows as the left table

RIGHT JOIN



```
SELECT *
FROM INFO I
RIGHT OUTER JOIN DETAIL D
ON I.EMPNO = D.EMPNO
```

INFO (I)

EMPNO	FNAME	LNAME
001	Emily	Stevens
002	John	Doe
003	James	Smith

RIGHT
OUTER
JOIN

DETAIL (D)

EMPNO	ROLE	EXT	TITLE
001	MGR	445	Mrs.
004	EXEC	101	Dr.

EMPNO	FNAME	LNAME	EMPNO	ROLE	EXT	TITLE
001	Emily	Stevens	001	MGR	445	Mrs.
NULL	NULL	NULL	004	EXEC	101	Dr.

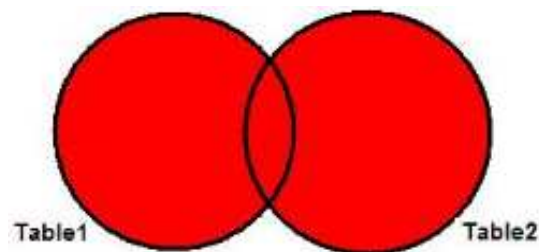
matching

The RIGHT JOIN works as a mirror of the LEFT JOIN: It selects the data set corresponding to the intersection of the two tables plus the remaining records from the table positioned to the right of the JOIN operator. Again, take a look at the corresponding Venn Diagram, and the SQL syntax. As you can see, the syntax is exactly the same, you just need to change the operator from “left join” to “right join”

To “process” the result set of this join, keep it simple by following these steps:

- 1) Take the entire right table
- 2) Take one row from the left table, and see if there is a matching row based on the column used for the JOIN (in this case, EMPNO)
- 3) if there is a match, put it (on the left) besides the corresponding column of the right table
- 4) If there is no match, put NULLs, and complete all rows with nulls
- 5) If the question is how many rows you will get at the end, you know you'll be picking all the rows from the right table, so if there is no WHERE clause, then you'd get at least the same number of rows as the right table

FULL JOIN



```
SELECT *
FROM INFO I
FULL OUTER JOIN DETAIL D
ON I.EMPNO = D.EMPNO
```

INFO (I)

EMPNO	FNAME	LNAME
001	Emily	Stevens
002	John	Doe
003	James	Smith

FULL
OUTER
JOIN

DETAIL (D)

EMPNO	ROLE	EXT	TITLE
001	MGR	445	Mrs.
004	EXEC	101	Dr.

EMPNO	FNAME	LNAME	EMPNO	ROLE	EXT	TITLE
001	Emily	Stevens	001	MGR	445	Mrs.
002	John	Doe	NULL	NULL	NULL	NULL
003	James	Smith	NULL	NULL	NULL	NULL
NULL	NULL	NULL	004	EXEC	101	Dr.

matching

The FULL JOIN brings all the data from both tables. It is more like extracting the union of two datasets. The NULL values might appear on the columns either the left or right tables, or both depending on the correspondence of the keys on both tables.

The syntax is almost the same: We just need to change the operator. Here, we're going to change from operator RIGHT JOIN to FULL JOIN.

To “process” the result set of this join, keep it simple by following these steps:

- 1) Take the table with most number of rows, say it's the left table in this example (so these steps assume that)
- 2) Take one row from the right table, and see if there is a matching row based on the column used for the JOIN (in this case, EMPNO)
- 3) if there is a match, put it (on the right) besides the corresponding column of the left table
- 4) If there is no match, create one more row, and put NULLs on the rest of the columns for this row
- 5) If the question is how many rows you will get at the end, you know you'll be picking all the rows from the left table, and from the right table, and then deduct the rows that have the same EMPNO. So in this example, the left table has 3 rows, the right table has 2 rows. There is one row that has the same EMPNO, so the final result would be $3+2-1=4$ rows

CROSS JOIN

- Cartesian product
- Simpler syntax

SELECT * FROM INFO I, DETAIL D

INFO (I)

EMPNO	FNAME	LNAME
001	Emily	Stevens
002	John	Doe
003	James	Smith

DETAIL (D)

EMPNO	ROLE	EXT	TITLE
001	MGR	445	Mrs.
004	EXEC	101	Dr.

CROSS
JOIN



EMPNO	FNAME	LNAME	EMPNO	ROLE	EXT	TITLE
001	Emily	Stevens	001	MGR	445	Mrs.
002	John	Doe	004	EXEC	101	Dr.
003	James	Smith	001	MGR	445	Mrs.
001	Emily	Stevens	004	EXEC	101	Dr.
002	John	Doe	001	MGR	445	Mrs.
003	James	Smith	004	EXEC	101	Dr.

Finally, let's talk about the CROSS JOIN operator. The CROSS JOIN is something very different from the previous examples. It calculates a Cartesian product of the two tables, that is: combine each record from the first table with every record from the second table.

This time we will have a slightly different syntax. When we talk about CROSS JOINS, there's no combining key. CROSS JOINS are seldom used but they are very helpful when you need to generate test data.

To “process” the result set of this join, keep it simple by following these steps:

- 1) Take the left table
- 2) There is no JOIN condition at all, so for each row on the left table, put each row of the right table.

Eg:

001 Emily Stevens 001 MGR 445 Mrs

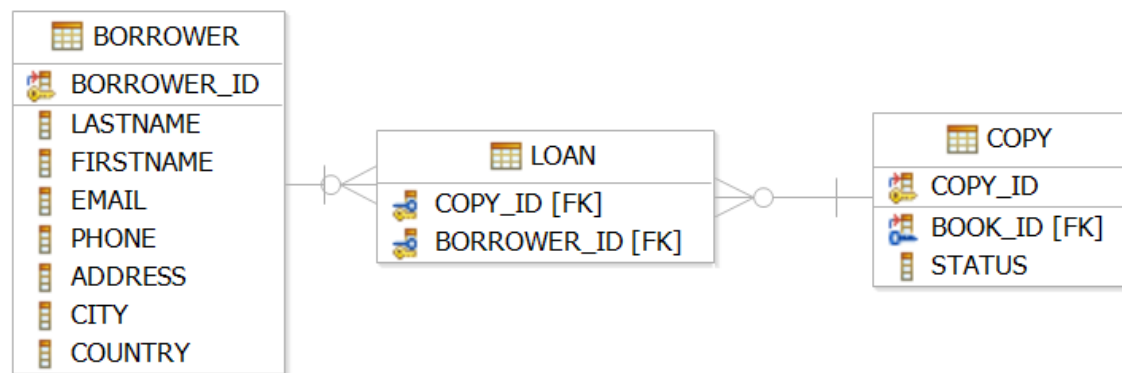
001 Emily Stevens 004 EXEC 101 Dr.

Repeat this process for each row on the left table.

- 3) If the question is how many rows you will get at the end, just multiply the rows from the left table times the rows from the right table. In this example: $3 \times 2 = 6$

Joining Multiple Tables

- You have to combine tables 2 at a time.



```
SELECT B.LASTNAME, L.COPY_ID, C.STATUS
FROM BORROWER B
INNER JOIN LOAN L ON B.BORROWER_ID = L.BORROWER_ID
INNER JOIN COPY C ON L.COPY_ID = C.COPY_ID
```

Until now, you only saw examples of combining two tables. But what if you need to combine data from three or more different tables?

You simply need to add new joins to the SQL statement. In fact, you can also use different types of joins to combine those tables.

Take a look at this simple diagram.

We have here 3 tables, BORROWER, LOAN, and COPY

If I need to write a SQL statement to get information from all of them

I need to do simple joins combining them two-by-two

First I will join information from BORROWER and LOAN

then I will join the information from LOAN and COPY, it's quite simple.



Thank you!

Use the forum in the db2university.com course AA001EN if you have technical questions about the materials covered in this course. Fellow students, faculty and IBMers can help you!