Birla Institute of Technology & Science, Pilani, K. K. BIRLA Goa campus
Database Systems and Applications (IS F243)
Second Semester 2012-2013
Lab-3
To study integrity constraints

Constraints are used for enforcing rules that the columns in a table have to confirm with.

Following are the various types of integrity constraints:

- Domain integrity constraints
- Entity integrity constraints
- Referential integrity constraints

Domain integrity constraints: These constraints set a range, and any violations that take place will prevent the user from performing the manipulations that caused the breach.

Two types are:

1. **NOT NULL CONSTRAINT:** By default table can contain null values. This constraint ensures that the table contains value.
2. **CHECK CONSTRAINT:** This can be defined to allow only a particular range of values.

Entity integrity constraints: There are two types of entity integrity constraints.

1. **UNIQUE CONSTRAINT:** The unique constraint designates a column or a group of columns as a unique key. This constraint allows only unique values to be stored in column and thus it rejects duplication of records.
2. **PRIMARY KEY CONSTRAINT:** This constraint is same as unique constraint but in addition to preventing duplication it also does not allow null values. This constraint cannot be put on the column having 'long' data type.

Referential integrity constraints: It enforces relationship between tables. It designates a column or combination of columns as a foreign key. The foreign key establishes a relationship with a specified primary or unique key in another table called the referenced key. In this relationship, the table containing the foreign key is called the child table, and the table containing the referenced key is called the parent table.

The InnoDB Storage Engine

InnoDB is a high-reliability and high-performance storage engine for MySQL. Starting with MySQL 5.5, it is the default MySQL storage engine. Key advantages of InnoDB include:

- Its design follows the ACID model, with transactions featuring commit, rollback, and crash-recovery capabilities to protect user data.
- Row-level locking and Oracle-style consistent reads increase multi-user concurrency and performance.
- **InnoDB** tables arrange your data on disk to optimize common queries based on primary keys. Each **InnoDB** table has a primary key index called the clustered index that organizes the data to minimize I/O for primary key lookups.
- **To maintain data integrity, InnoDB also supports FOREIGN KEY referential-integrity constraints.**
- You can freely mix **InnoDB** tables with tables from other MySQL storage engines, even within the same statement. For example, you can use a join operation to combine data from **InnoDB** and **MEMORY** tables in a single query.

To determine whether your server supports **InnoDB** use the **SHOW ENGINES** statement.

Creating and Using InnoDB Tables

To create an **InnoDB** table, specify an **ENGINE=InnoDB** option in the **CREATE TABLE** statement:

```
CREATE TABLE customers (a INT, b CHAR (20), INDEX (a)) ENGINE=InnoDB;
```

FOREIGN KEY Constraints

InnoDB supports foreign keys, which let you cross-reference related data across tables, and foreign key constraints, which help keep this spread-out data consistent. The syntax for an **InnoDB** foreign key constraint definition in the **CREATE TABLE** or **ALTER TABLE** statement looks like this:

```
[CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

reference_option:
  RESTRICT | CASCADE | SET NULL | NO ACTION

index_name represents a foreign key ID. If given, this is ignored if an index for the foreign key is defined explicitly. Otherwise, if **InnoDB** creates an index for the foreign key, it uses ***index_name*** for the index name.

Foreign keys definitions are subject to the following conditions:

- Foreign key relationships involve a parent table that holds the central data values, and a child table with identical values pointing back to its parent. The **FOREIGN KEY** clause is specified in the child table. The parent and child tables must both be **InnoDB** tables. They must not be **TEMPORARY** tables.
- Corresponding columns in the foreign key and the referenced key must have similar internal data types inside**InnoDB** so that they can be compared without a type conversion. *The size and sign of integer types must be the same*. The length of string types need not be the same. For nonbinary (character) string columns, the character set and collation must be the same.
- **InnoDB** requires indexes on foreign keys and referenced keys so that foreign key checks can be fast and not require a table scan. In the referencing table, there must be an index where the foreign key columns are listed as the *first* columns in the same order. Such an index is created on the referencing table automatically if it does not exist. This index might be silently dropped later, if you create another index that can be used to enforce the foreign key constraint. ***index_name***, if given, is used as described previously.
- **InnoDB** permits a foreign key to reference any index column or group of columns. However, in the referenced table, there must be an index where the referenced columns are listed as the *first* columns in the same order.
- Index prefixes on foreign key columns are not supported. One consequence of this is that **BLOB** and **TEXT** columns cannot be included in a foreign key because indexes on those columns must always include a prefix length.
- If the **CONSTRAINT** ***symbol*** clause is given, the ***symbol*** value must be unique in the database. If the clause is not given, **InnoDB** creates the name automatically.

Referential Actions

This section describes how foreign keys play an important part in safeguarding referential integrity.

InnoDB rejects any **INSERT** or **UPDATE** operation that attempts to create a foreign key value in a child table if there is no a matching candidate key value in the parent table.

When an **UPDATE** or **DELETE** operation affects a key value in the parent table that has matching rows in the child table, the result depends on the *referential action* specified using **ON UPDATE** and **ON DELETE** subclauses of the**FOREIGN KEY** clause. **InnoDB** supports five options regarding the action to be taken. If **ON DELETE** or **ON UPDATE**are not specified, the default action is **RESTRICT**.

- **CASCADE**: Delete or update the row from the parent table, and automatically delete or update the matching rows in the child table. Both **ON DELETE CASCADE** and **ON UPDATE CASCADE** are supported. Between two tables, do not define several **ON UPDATE CASCADE** clauses that act on the same column in the parent table or in the child table.
  Note

  Currently, cascaded foreign key actions do not activate triggers.

- **SET NULL**: Delete or update the row from the parent table, and set the foreign key column or columns in the child table to **NULL**. Both **ON DELETE SET NULL** and **ON UPDATE SET NULL** clauses are supported.
  If you specify a **SET NULL** action, *make sure that you have not declared the columns in the child table as **NOT NULL***.
- **RESTRICT**: Rejects the delete or update operation for the parent table.
  Specifying **RESTRICT** (or **NO ACTION**) is the same as omitting the **ON DELETE** or **ON UPDATE** clause.
- **NO ACTION**: A keyword from standard SQL. In MySQL, equivalent to **RESTRICT**. **InnoDB** rejects the delete or update operation for the parent table if there is a related foreign key value in the referenced table. Some database systems have deferred checks, and **NO ACTION** is a deferred check. In MySQL, foreign key constraints are checked immediately, so **NO ACTION** is the same as **RESTRICT**.
- **SET DEFAULT**: This action is recognized by the parser, but **InnoDB** rejects table definitions containing **ON DELETE SET DEFAULT** or **ON UPDATE SET DEFAULT** clauses.

InnoDB supports foreign key references between one column and another within a table. (A column cannot have a foreign key reference to itself.) In these cases, “child table records” really refers to dependent records within the same table.

Examples of Foreign Key Clauses

Here is a simple example that relates **parent** and **child** tables through a single-column foreign key:

```
CREATE TABLE parent (id INT NOT NULL,
                     PRIMARY KEY (id)
) ENGINE=INNODB;
CREATE TABLE child (id INT, parent_id INT,
                    INDEX par_ind (parent_id),
                    FOREIGN KEY (parent_id) REFERENCES parent(id)
                     ON DELETE CASCADE
) ENGINE=INNODB;
```

A more complex example in which a **product_order** table has foreign keys for two other tables. One foreign key references a two-column index in the **product** table. The other references a single-column index in the **customer**table:

```
CREATE TABLE product (category INT NOT NULL, id INT NOT NULL,
                      price DECIMAL,
                      PRIMARY KEY(category, id)) ENGINE=INNODB;
CREATE TABLE customer (id INT NOT NULL,
                       PRIMARY KEY (id)) ENGINE=INNODB;
CREATE TABLE product_order (no INT NOT NULL AUTO_INCREMENT,
                            product_category INT NOT NULL,
                            product_id INT NOT NULL,
                            customer_id INT NOT NULL,
                            PRIMARY KEY(no),
                            INDEX (product_category, product_id),
                            FOREIGN KEY (product_category, product_id)
                              REFERENCES product(category, id)
                              ON UPDATE CASCADE ON DELETE RESTRICT,
                            INDEX (customer_id),
                            FOREIGN KEY (customer_id)
                              REFERENCES customer(id)) ENGINE=INNODB;
```

InnoDB enables you to add a new foreign key constraint to a table by using ALTER TABLE:

```
ALTER TABLE tbl_name
    ADD [CONSTRAINT [symbol]] FOREIGN KEY
    [index_name] (index_col_name, ...)
    REFERENCES tbl_name (index_col_name,...)
    [ON DELETE reference_option]
    [ON UPDATE reference_option]
```

The foreign key can be self referential (referring to the same table). When you add a foreign key constraint to a table using **ALTER TABLE**, *remember to create the required indexes first.*

Foreign Keys and ALTER TABLE

InnoDB supports the use of **ALTER TABLE** to drop foreign keys:

```
ALTER TABLE tbl_name DROP FOREIGN KEY fk_symbol;
```

If the **FOREIGN KEY** clause included a **CONSTRAINT** name when you created the foreign key, you can refer to that name to drop the foreign key. Otherwise, the ***fk_symbol*** value is internally generated by **InnoDB** when the foreign key is created. To find out the symbol value when you want to drop a foreign key, use the **SHOW CREATE TABLE** statement. For example:

```
mysql> SHOW CREATE TABLE ibtest11c\G
*************************** 1. row ***************************
       Table: ibtest11c
Create Table: CREATE TABLE `ibtest11c` (
  `A` int(11) NOT NULL auto_increment,
  `D` int(11) NOT NULL default '0',
  `B` varchar(200) NOT NULL default '',
  `C` varchar(175) default NULL,
  PRIMARY KEY  (`A`,`D`,`B`),
  KEY `B` (`B`,`C`),
  KEY `C` (`C`),
  CONSTRAINT `0_38775` FOREIGN KEY (`A`, `D`)
REFERENCES `ibtest11a` (`A`, `D`)
ON DELETE CASCADE ON UPDATE CASCADE,
  CONSTRAINT `0_38776` FOREIGN KEY (`B`, `C`)
REFERENCES `ibtest11a` (`B`, `C`)
ON DELETE CASCADE ON UPDATE CASCADE
) ENGINE=INNODB CHARSET=latin1
1 row in set (0.01 sec)
```

```
mysql> ALTER TABLE ibtest11c DROP FOREIGN KEY `0_38775`;
```

You cannot add a foreign key and drop a foreign key in separate clauses of a single **ALTER TABLE** statement. Separate statements are required.

If **ALTER TABLE** for an **InnoDB** table results in changes to column values (for example, because a column is truncated), **InnoDB**'s **FOREIGN KEY** constraint checks do not notice possible violations caused by changing the values.