

Backups



- ❑ Backups are one of the most important parts of database maintenance.
- ❑ Unfortunately they are also sometimes the most neglected part of a larger software effort.
- ❑ No database design can protect against system crashes, power failures, and the user accidentally deleting critical information.
- ❑ Without good backups, you could lose significant chunks of data.

Two main kinds of backups that many databases support



- A full backup makes a copy of everything in the database.
 - ▣ Depending on the size of the database, this might take a lot of time and disk space.
 - ▣ For a reasonably small database that is only used during business hours, you might have the computer automatically back up the database at midnight.
 - ▣ Even if the backup takes several hours, the computer has little else to do in the middle of the night.

- An incremental backup only records data that has changed since an earlier date.
 - ▣ For example, you might back up all changes since the previous full backup.
 - ▣ To restore an incremental backup, you need to first restore a full backup and then reapply the changes saved in the incremental backup.

Incremental Backup

- Making an incremental backup is faster than making a full backup but restoring the data is harder.
 - ▣ Because they are faster, incremental backups are useful for really big databases where it would take too long to make a full backup.
- For example, suppose you have a really active database that records many thousands of transactions per day, such as a database that tracks keywords used in major news stories around the world.
 - ▣ Suppose that you need the database to be running at full speed 20 hours a day on weekdays but a full backup takes 12 hours.
 - ▣ Then on Saturday morning you might make a full backup and on other days you would make an incremental backup.
 - ▣ Now suppose the database crashes and burns on a Thursday.
 - ▣ To restore the database, you would restore the previous weekend's full backup and then apply the incremental backups for Monday, Tuesday, and Wednesday
 - ▣ That could take quite a while.

Making the backup process a bit faster

- To make the process a bit faster, you could make a larger incremental backup halfway through the week.
- On Monday, Tuesday, Thursday, and Friday, you would make an incremental backup recording changes since the previous day.
- On Wednesday you would make an incremental backup to record all changes made since the previous Saturday full backup.
- Now to recover from a crash on Thursday, you only need to restore Saturday's full backup and then Wednesday's incremental backup.
- It will still take a while but it will be a bit faster and easier.
- Wednesday's incremental backup will also take longer than the daily backups but it will be a lot faster than a full backup.

Some DBs allow you to perform backups while in use

- This is critical for databases that must be available most or all of the time.
- The backup will slow the database down so you still need to schedule backups for off-peak periods such as weekends or the middle of the night, but at least the database can keep running.
- For example, a local grocery store's cash registers perform downloads, uploads, and backups in the middle of the night.
- If you stop in around midnight, the self-checkout machines usually run much slower than they do during the day.

Backups are intended for unexpected damage to the DB

- That includes normal damage caused by logical disasters such as power glitches, viruses, spilled soda, and EBCAK (Error Between Chair and Keyboard) problems, but it also includes physical calamities such as fire, tornado, and volcanic eruption.
- Your backups do you no good if they're stored next to the database's computer and you are hit by one of these.
- A full backup won't do you any good if the flash drive or DVD that holds it is sitting on top of the computer and a meteor reduces the whole thing to a pile of steel and plastic splinters.
- To avoid this kind of problem, think about taking backups offsite.
- Of course, that creates a potential security issue if your data is sensitive (for example, credit card numbers, medical records, or salary information).

Make a backup plan

- Suppose you have a really large database.
- A full backup takes around 10 hours, whereas an incremental backup takes about 1 hour per day of changes that you want to include in the backup.
- You are not allowed to make backups during the peak usage hours of
 - ▣ 3:00am to 11:00pm weekdays, and
 - ▣ 6:00am to 8:00pm on weekends.
- Figure out what types of backups to perform on which days to make restoring the database as easy as possible.
 - ▣ 1. Figure out when you have time for a full backup.
 - ▣ 2. For each day after the full backup, make an incremental backup. Make the backup go back to the most complete previous backup it can reach given the time constraints.

1. Figure out when you have time for a full backup.

The following table shows the number of off-peak hours you have available for performing back-ups during each night of the week.

Night	Off-Peak Start	Off-Peak End	Off-Peak Hours
Monday	11:00pm	3:00am	4
Tuesday	11:00pm	3:00am	4
Wednesday	11:00pm	3:00am	4
Thursday	11:00pm	3:00am	4
Friday	11:00pm	6:00am	7
Saturday	8:00pm	6:00am	10
Sunday	8:00pm	3:00am	7

The only time when you have enough off-peak hours to perform a full backup is Saturday night.

2. For each day after the full backup, make an incremental backup. Make the backup go back to the most complete previous backup it can reach given the time constraints.

On Sunday, Monday, Tuesday, and Wednesday nights, the incremental backup has at least 4 hours so it can save changes for up to the previous 4 days. All of these backups should record the changes since the full backup on Saturday night. If you need to restore one of these backups, you only need to apply the previous full backup and then one incremental backup.

On Thursday and Friday nights, you don't have time to go all the way back to the previous full backup. There is time, however, to record all changes since the Wednesday night incremental backup so you should do so. If you need to restore one of these backups, you will have to restore the last full backup, then the Wednesday night backup, and then this backup.

The following table shows the complete backup schedule.

Night	Backup Type
Monday	Incremental from last Saturday
Tuesday	Incremental from last Saturday
Wednesday	Incremental from last Saturday
Thursday	Incremental from last Wednesday
Friday	Incremental from last Wednesday
Saturday	Full
Sunday	Incremental from last Saturday

Don't forget to store copies of the backups offsite in a secure location.

Data Warehousing

- Many database applications have two components: an online part that is used in day-to-day business and an offline “data warehousing” part that is used to generate reports and perform more in-depth analysis of the data.
- The rules for a data warehouse are different than those for an online database.
- Often a data warehouse contains duplicated data, non-normalized tables, and special data structures that make building reports easier.
- Warehoused data is updated much less frequently than online data.
- In a data warehouse, flexibility in reporting is more important than speed.
- For the purposes of this class, it’s important that you be aware of your customers’ data warehousing needs so you can plan for appropriate database maintenance.
- In some cases, that may be as simple as passing a copy of the most recent full backup to a data analyst.
- In others, it may mean writing and executing special data extraction routines periodically.
- For example, as part of nightly maintenance (backups, cleaning up tables, and what have you), you might need to pull sales data into a separate table or database for later analysis.
- This class isn’t about data warehousing so we don’t say any more about it.

Repairing the Database



- Although databases provide lots of safeguards to protect your data, databases sometimes become corrupted.
- They are particularly prone to index corruption because it can take a while to update a database's index structures.
- If the computer crashes while the database is in the middle of updating its index trees, the trees may contain garbage, invalid keys, and pointers leading to nowhere.
- When an index becomes corrupted, the results you get from queries may become unpredictable.
- You may get the wrong records, records in the wrong order, or no records at all.
- The program using the database may even crash.
- To ensure that your DB works properly, you should periodically run its repair tools.
- That should clean up damaged records and indexes.

Compacting the Database

- When you delete a record, many databases don't actually release the space that the record occupied.
- Some databases may be able to undelete the record in case you decide you want it later but most databases do this so they can reuse the space later for new records.
- If you add and then remove a lot of records, however, the database can become full of unused space.
- The trees that databases typically use to store indexes are self-balancing.
- That ensures that they never grow too tall so searches are fast, but it also means that they contain extra unused space.
- They use that extra space to make adding new entries in the trees more efficient but, under some circumstances, the trees can contain a lot of unused space.
- Disk space is relatively cheap so the "wasted" space may not be much of an issue.
- Having some extra unused space in the database can even make adding and updating the database faster.
- In some cases, however, parts of the database may become fragmented so the database may take longer to load records that are logically adjacent but scattered around the disk.
- In that case, you may get better performance if you compact and defragment the DB..
- Look at your DB product's instructions to learn about good maintenance strategies.

Performance Tuning

- Normally you don't need to worry too much about how the database executes a query.
- In fact, if you start fiddling around with the way queries are executed, you take on all sorts of unnecessary responsibility.
- It's kind of like being an air traffic controller: when everything works, no one notices that you're doing your job, but when something goes wrong everyone knows it was your fault.
- Generally you shouldn't try to tell the DB how to do its job, but often you can help it help itself.
- Some databases use a statistical analysis of the values in an index to help decide how to perform queries using that index.
- If the distribution of the values changes, you can sometimes help the database realize that the situation has changed.
 - ▣ For example, the Transact-SQL statement `UPDATE STATISTICS` makes a SQL Server database update its statistics for a table or view, possibly leading to better performance in complex queries.
- Often you can make queries more efficient by building good indexes.
 - ▣ If you know that the users will be looking for records with specific values in a certain field, put an index on that field.
 - ▣ For example, if you know that you will need to search customer records by LastName, make LastName an index.

If you have a lot of experience with query optimization, you may even be able to give the database a hint about how it should perform a particular query. For example, you may know that a GROUP BY query will work best if the database uses a hashing algorithm. In Transact-SQL you could use the OPTION (HASH GROUP) clause to give the database that hint (technet.microsoft.com/en-us/library/ms181714.aspx). Only *serious* SQL nerds (with IQs exceeding their weights in pounds) should even consider this level of meddling.

Some databases provide tools such as query analyzers or execution plan viewers so you can see exactly how the database will perform an operation. That not only lets you learn more about how queries are performed so you can aspire to write your own query hints, but it also lets you look for problems in your database design. For example, an execution plan may point out troublesome WHERE clauses that require executing a function many times, searches on fields that are not indexed, and nested loops that you might be able to remove by rewriting a query.

More expensive database products may also be able to perform other optimizations at a more physical level. For example, database replication allows several databases to contain the same information and remain synchronized. This can be useful if you perform more queries than updates. If one database is the master and the others are used as read-only copies, the copies can take some of the query burden from the main database.

Another advanced optimization technique is partitioning. A partitioned table stores different records in different locations, possibly on different hard disks or even different computers. If your typical queries normally divide the data along partition boundaries, the separate partitions can operate more or less independently. You may even be able to back up different partitions separately, improving performance.

In a variation on partitioning, you use multiple databases to handle different parts of the data. For example, you might have different databases to handle customers in different states or time zones. Because the databases operate independently, they are smaller and faster, and you can back them up separately. You can extract data into a data warehouse to perform queries that involve more than one database.

The Keys to Success



- ❑ Not all indexes are created equal.
 - ❑ You need to tailor a table's indexes and keys to help the database perform the queries that you expect to actually perform.
 - ❑ Suppose you have a Customers table that contains the usual sorts of fields: CustomerId, FirstName, LastName, Street, City, State, and Zip.
 - ❑ It also includes some demographic information such as BirthDate, AnnualIncome, and Gender.
-
1. Decide which fields should be indexed to support normal database queries that must join Orders and OrderItems records to Customers records.
 2. Decide which fields should be indexed to support typical customer queries where a customer wants information about an order.
 3. Decide which fields should be indexed to support reporting queries such as “Find all orders placed by female customers between ages 15 and 25.”

Database Security




- Like database maintenance, database security is an important topic with details that vary from database to database.
- This lecture doesn't try to cover everything there is to know about database security.
- Instead it explains some of the general concepts that you should understand.
 - ▣ Pick a reasonable level of security for the database.
 - ▣ Choose good passwords.
 - ▣ Give users necessary privileges.
 - ▣ Promote a database's physical security.

The Right Level of Security

- Database security can range from nonexistent to tighter than Fort Knox.
- You can allow any user or application to connect to a database or you can use encryption to prevent even the DB itself from looking at data that it shouldn't see.
- Some databases can encrypt the data they contain so it's very hard for bad guys to peek at your data.
- Unfortunately it takes extra time to encrypt and decrypt data as you read and write it in the database, and that slows things down.

Passwords

- 
- Passwords are the most obvious form of security in most applications.
 - Different databases handle passwords differently and with different levels of safety.

Single-Password Databases

- At the weaker end of the spectrum, some databases provide only a single password for the entire DB.
- The single password provides access to the entire database.
- That means a bad guy who learns the password can get into the database.
- It also means anyone who should use the DB must share that password.
- One consequence of that is that you cannot easily tell which user makes which changes to the data.
- In practice that often means the program that provides a user interface to the database knows the password and then it may provide its own extra layer of password protection.

Example

- For example, the application might store user names and passwords (hopefully encrypted, not in their plain text form) in a table.
- When the user runs the program, it uses its hard-coded password to open the database and verifies the user's name and password in the table.
- It then decides whether to allow the user in (and decides what privileges the user deserves) or whether it should display a nasty message, shut itself down, send threatening email to the user's boss, and so forth.)
- There are a couple of reasons why this is a weak approach.
 - ▣ *First, the program must contain the password in some form so it can open the DB.*
 - Even if you encrypt the password within the code, a determined hacker will be able to get it back out.
 - At worst, a tenacious bit-monkey could examine the program's memory while it was executing and figure out what password the database used.
 - ▣ *A second reason why this approach can be risky is that it relies on the correctness of the UI.*
 - Every non-trivial program contains bugs so there's a chance that users will find some way to bypass the homemade security system and sneak in somewhere they shouldn't be

Individual Passwords

- More sophisticated databases give each user a separate password and that has several advantages over a single password database.
- If the database logs activity, you can tell who logged into the database when.
- If there are problems, the log may help you narrow down who caused the problem and when.
- If the database logs every interaction with the database (or if your application does), you can tell exactly who messed up.
- Another advantage to individual passwords is that the user interface program doesn't ever need to store a password.
- When the program starts, the user enters a user name and password and the program tries to use them to open the database. The database either opens or not and the program doesn't need to worry about why.
- Even a “seriously dope uberhacker with mad skillz” can't dig a password out of the application if the password isn't there.
- Because the database takes care of password validation, you can focus on what the program is supposed to help the users do instead of worrying about whether you made a mistake in the password validation code.
- If your database allows individual user passwords, use them.
- They provide a lot of benefits with relatively little extra work on your part.

Operating System Passwords



- ❑ Some databases don't manage passwords very well.
- ❑ They may use little or no encryption, may not enforce any password standards (allowing weak passwords such as "12345" and "password"), and may even write passwords into log files where a hacker can find them relatively easily.
- ❑ If your database can integrate its own security with the security provided by the operating system, make it do so.
- ❑ In any case, take advantage of the operating system's security.
- ❑ Make sure users pick good operating system passwords and don't share them
- ❑ A hacker won't get a chance to attack your database if he can't even log in to the operating system.

Good Passwords

- ❑ Picking good passwords is something of an art.
- ❑ You need to pick something obscure enough that an evil hacker (or your prankster coworkers) can't guess but that's also easy enough for you to remember.
- ❑ It's easy to become overloaded when you're expected to remember the database password in addition to your computer user name and password, bank PIN number, voice mail password, online banking password, PayPal password, eBay password, locker combination, anniversary, and children's names.
- ❑ And you don't want to use the same password for all of these because then if someone ever steals your eBay password, they know all of your passwords.
- ❑ Many companies have policies that require you to use certain characters in your password (must include letters, numbers, and a special character such as \$ or #, and you need to type every other character with your left hand and your eyes crossed).
- ❑ So what do users do when faced with dozens of passwords that must pass complex checks? They write their passwords down where they are easy to find.
- ❑ They pick sequential passwords such as Secret1, Secret2, and so forth.
- ❑ They use names and dates that are easy to remember and guess.
- ❑ By throwing names, dates, and common words at the DB, it would be easy to guess passwords.

Privileges



- ❑ Most relational databases allow you to restrict each user's access to specific tables and even columns within a table.
- ❑ Typically you would define groups such as Clerks or Managers and then grant permission for users in those groups to view certain data. You may also be able to grant exceptions for individual users.
- ❑ For example, suppose your Employees table contains columns holding three levels of data.
- ❑ Data available to anyone includes the employee's name, office number, phone number, and so forth.
- ❑ Data available only to managers includes the employee's salary and performance reviews.
- ❑ Data available to human resources includes the employee's next of kin, insurance information, school grades, and beneficiary name.
- ❑ You could get into serious trouble if some of that data were to slip out.

GRANT and REVOKE



- ❑ The SQL GRANT and REVOKE statements let you give and withdraw privileges.
- ❑ It is generally safest to give users the fewest privileges possible to do their jobs
- ❑ Then if an application contains a bug and accidentally tries to do something stupid, such as dropping a table or showing the user sensitive information, the database won't allow it.
- ❑ Rather than remembering to remove every extraneous privilege from a new user, many database administrators revoke all privileges and then grant those that are needed.
- ❑ That way the administrator cannot forget to remove some critical privilege.

CREATE USER Syntax

```
CREATE USER user_specification
    [, user_specification] ...
```

user_specification:

```
user [IDENTIFIED BY [PASSWORD] 'password']
```

The [CREATE USER](#) statement creates new MySQL accounts. To use it, you must have the global [CREATE USER](#) privilege or the [INSERT](#) privilege for the `mysql` database. For each account, [CREATE USER](#) creates a new row in the `mysql.user` table and assigns the account no privileges. An error occurs if the account already exists.

Each account name uses the format described in [Section 6.2.3, “Specifying Account Names”](#). For example:

```
CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'mypass';
```

If you specify only the user name part of the account name, a host name part of `'%'` is used.

The user specification may indicate how the user should authenticate when connecting to the server:

- To enable the user to connect with no password (which is *insecure*), include no [IDENTIFIED BY](#) clause:

```
CREATE USER 'jeffrey'@'localhost';
```

- To assign a password, use [IDENTIFIED BY](#) with the literal plaintext password value:

```
CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'mypass';
```

- To avoid specifying the plaintext password if you know its hash value (the value that [PASSWORD\(\)](#) would return for the password), specify the hash value preceded by the keyword [PASSWORD](#):

```
CREATE USER 'jeffrey'@'localhost'
IDENTIFIED BY PASSWORD '*90E462C37378CED12064BB3388827D2BA3A9B689';
```

GRANT Syntax

GRANT

```
    priv_type [(column_list)]  
    [, priv_type [(column_list)] ...  
ON [object_type] priv_level  
TO user_specification [, user_specification] ...  
[REQUIRE {NONE | ssl_option [[AND] ssl_option] ...}]  
[WITH with_option ...]
```

object_type:

```
TABLE  
| FUNCTION  
| PROCEDURE
```

priv_level:

```
*  
| *.*  
| db_name.*  
| db_name.tbl_name  
| tbl_name  
| db_name.routine_name
```

user_specification:

```
    user [IDENTIFIED BY [PASSWORD] 'password']
```

ssl_option:

```
SSL  
| X509  
| CIPHER 'cipher'  
| ISSUER 'issuer'  
| SUBJECT 'subject'
```

with_option:

```
GRANT OPTION  
| MAX_QUERIES_PER_HOUR count  
| MAX_UPDATES_PER_HOUR count  
| MAX_CONNECTIONS_PER_HOUR count  
| MAX_USER_CONNECTIONS count
```

The [GRANT](#) statement grants privileges to MySQL user accounts. [GRANT](#) also serves to specify other account characteristics such as use of secure connections and limits on access to server resources. To use [GRANT](#), you must have the [GRANT OPTION](#) privilege, and you must have the privileges that you are granting.

Normally, a database administrator first uses [CREATE USER](#) to create an account, then [GRANT](#) to define its privileges and characteristics. For example:

```
CREATE USER 'jeffrey'@'localhost' IDENTIFIED BY 'mypass';
GRANT ALL ON db1.* TO 'jeffrey'@'localhost';
GRANT SELECT ON db2.invoice TO 'jeffrey'@'localhost';
GRANT USAGE ON *.* TO 'jeffrey'@'localhost' WITH MAX_QUERIES_PER_HOUR 90;
```

However, if an account named in a [GRANT](#) statement does not already exist, [GRANT](#) may create it under the conditions described later in the discussion of the [NO AUTO CREATE USER](#) SQL mode.

The [REVOKE](#) statement is related to [GRANT](#) and enables administrators to remove account privileges. See [Section 13.7.1.5, “REVOKE Syntax”](#).

When successfully executed from the [mysql](#) program, [GRANT](#) responds with `Query OK, 0 rows affected`. To determine what privileges result from the operation, use [SHOW GRANTS](#). See [Section 13.7.5.17, “SHOW GRANTS](#)

REVOKE Syntax

REVOKE

```
priv_type [(column_list)]  
[, priv_type [(column_list)]] ...  
ON [object_type] priv_level  
FROM user [, user] ...
```

REVOKE ALL PRIVILEGES, GRANT OPTION

```
FROM user [, user] ...
```

The [REVOKE](#) statement enables system administrators to revoke privileges from MySQL accounts. Each account name uses the format described in [Section 6.2.3, "Specifying Account Names"](#). For example:

```
REVOKE INSERT ON *.* FROM 'jeffrey'@'localhost';
```

If you specify only the user name part of the account name, a host name part of `'%'` is used.

For details on the levels at which privileges exist, the permissible `priv_type` and `priv_level` values, and the syntax for specifying users and passwords, see [Section 13.7.1.3, "GRANT Syntax"](#)

To use the first [REVOKE](#) syntax, you must have the [GRANT OPTION](#) privilege, and you must have the privileges that you are revoking.

To revoke all privileges, use the second syntax, which drops all global, database, table, column, and routine privileges for the named user or users:

```
REVOKE ALL PRIVILEGES, GRANT OPTION FROM user [, user] ...
```

To use this [REVOKE](#) syntax, you must have the global [CREATE USER](#) privilege or the [UPDATE](#) privilege for the `mysql` database.

[REVOKE](#) removes privileges, but does not drop `mysql.user` table entries. To remove a user account entirely, use [DROP USER](#) (see [Section 13.7.1.2, "DROP USER Syntax"](#)) or [DELETE](#).

Table 13.1. Permissible Privileges for [GRANT](#) and [REVOKE](#)

Privilege	Meaning
ALL [PRIVILEGES]	Grant all privileges at specified access level except GRANT OPTION
ALTER	Enable use of ALTER TABLE
ALTER ROUTINE	Enable stored routines to be altered or dropped
CREATE	Enable database and table creation
CREATE ROUTINE	Enable stored routine creation
CREATE TEMPORARY TABLES	Enable use of CREATE TEMPORARY TABLE
CREATE USER	Enable use of CREATE USER , DROP USER , RENAME USER , and REVOKE ALL PRIVILEGES
CREATE VIEW	Enable views to be created or altered
DELETE	Enable use of DELETE
DROP	Enable databases, tables, and views to be dropped
EXECUTE	Enable the user to execute stored routines
FILE	Enable the user to cause the server to read or write files
GRANT OPTION	Enable privileges to be granted to or removed from other accounts
INDEX	Enable indexes to be created or dropped
INSERT	Enable use of INSERT
LOCK TABLES	Enable use of LOCK TABLES on tables for which you have the SELECT privilege
PROCESS	Enable the user to see all processes with SHOW PROCESSLIST
REFERENCES	Not implemented
RELOAD	Enable use of FLUSH operations
REPLICATION CLIENT	Enable the user to ask where master or slave servers are
REPLICATION SLAVE	Enable replication slaves to read binary log events from the master
SELECT	Enable use of SELECT
SHOW DATABASES	Enable SHOW DATABASES to show all databases
SHOW VIEW	Enable use of SHOW CREATE VIEW
SHUTDOWN	Enable use of mysqladmin shutdown
SUPER	Enable use of other administrative operations such as CHANGE MASTER TO , KILL , PURGE BINARY LOGS , SET GLOBAL , and mysqladmin debug command
UPDATE	Enable use of UPDATE
USAGE	Synonym for "no privileges"

The following three MySQL scripts demonstrate user privileges. You can execute the first and third scripts in the MySQL Command Line Client. You need to start the Command Line Client in a special way (described shortly) to use the second script properly.

The following script prepares a test database for use:

```
CREATE DATABASE UserDb;
USE UserDb;

-- Create a table.
CREATE TABLE People (
  FirstName      VARCHAR(5)      NOT NULL,
  LastName       VARCHAR(40)     NOT NULL,
  Salary         DECIMAL(10,2)   NULL,
  PRIMARY KEY (LastName, FirstName)
);

-- Create a new user with an initial password.
-- Note that this password may appear in the logs.
CREATE USER Rod IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM Rod;

-- Grant privileges that the user really needs.
--GRANT INSERT ON UserDb.People TO Rod;
GRANT INSERT (FirstName, LastName, Salary) ON UserDb.People TO Rod;
GRANT SELECT (FirstName, LastName) ON UserDb.People TO Rod;
GRANT DELETE ON UserDb.People TO Rod;
```


Note that the user name is case-sensitive, so type Rod not rod or ROD. When prompted, enter the password "secret." The Command Line Client should run in the operating system command window and you should see the mysql prompt.

Now you can execute the following script to test the user's privileges:

```
USE UserDB;

-- Make some records.
INSERT INTO People VALUES('Annie', 'Lennox', 50000);
INSERT INTO People VALUES('Where', 'Waldo', 60000);
INSERT INTO People VALUES('Frank', 'Stein', 70000);

-- Select the records.
-- This fails because we don't have SELECT privilege on the Salary column.
SELECT * FROM People ORDER BY FirstName, LastName;

-- Select the records.
-- This works because we have SELECT privilege on FirstName and LastName.
SELECT FirstName, LastName FROM People ORDER BY FirstName, LastName;

-- Create a new table.
-- This fails because we don't have CREATE TABLE privileges.
CREATE TABLE MorePeople (
    FirstName          VARCHAR(5)      NOT NULL,
    LastName            VARCHAR(40)     NOT NULL,
    PRIMARY KEY (LastName, FirstName)
);

-- Delete the records.
DELETE FROM People;
```

This script sets UserDB as the default database and inserts some records into the People table. This works because the user Rod has privileges to insert FirstName, LastName, and Salary values into this table.

Consider the database design shown in Figure 20-1.

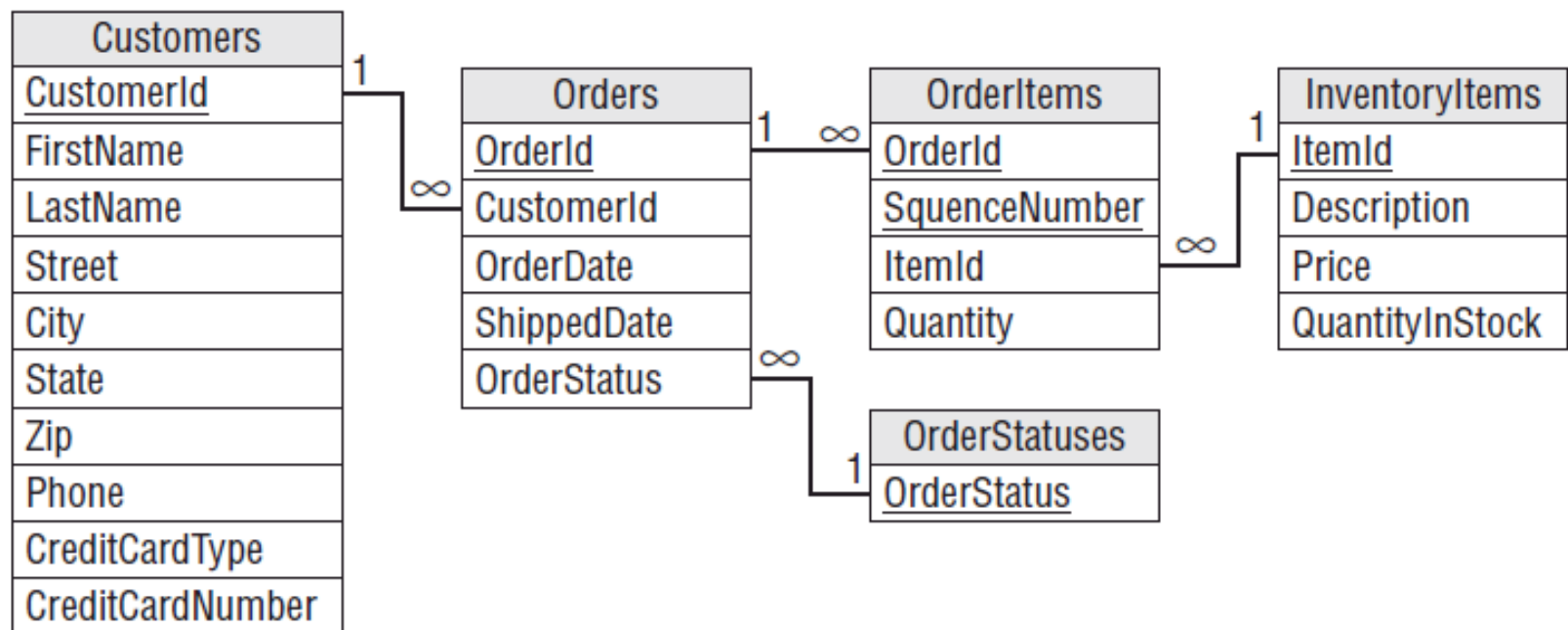


Figure 20-1

Many users need relatively few privileges to do their jobs. Write a SQL script that gives a shipping clerk enough privileges to fulfill orders in the database shown in Figure 20-1 by following these steps:

1. Make a permission table showing the CRUD (Create, Read, Update, and Delete) privileges that the user needs for the tables and their fields.
2. Deny all privileges.
3. Grant `SELECT` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly address a shipment.
4. Grant `UPDATE` privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly record a shipment.

1. Make a table showing the CRUD (Create, Read, Update, and Delete) privileges that the user needs for the tables and their fields.

The following table lists the privileges that the user needs. The user might need Create or Delete privileges for tables and Read or Update privileges for fields.

Table or Field	Privileges
Customers	–
CustomerId	R
FirstName	R
LastName	R
Street	R
City	R
State	R
Zip	R
Phone	R
CreditCardType	–
CreditCardNumber	–
Orders	–
OrderId	R
CustomerId	R
OrderDate	R
ShippedDate	RU
OrderStatus	RU
OrderItems	–
OrderId	R
SequenceNumber	R
ItemId	R
Quantity	R
InventoryItems	–
ItemId	R
Description	R
Price	–
QuantityInStock	RU
OrderStatuses	–
OrderStatus	R

2. Deny all privileges.

The following MySQL code creates a ShippingClerk user and revokes all privileges. (Again this is a terrible password. Don't use it.)

```
CREATE USER ShippingClerk IDENTIFIED BY 'secret';

-- Revoke all privileges for the user.
REVOKE ALL PRIVILEGES, GRANT OPTION FROM ShippingClerk;
```

3. Grant SELECT privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly address a shipment.

To prepare and ship orders, the user must see all fields in the Orders, OrderItems, and InventoryItems tables. The clerk must also see the name and address information in the Customers table.

The following MySQL statements grant privileges to select those fields:

```
GRANT SELECT ON ShippingDb.Orders TO ShippingClerk;
GRANT SELECT ON ShippingDb.OrderItems TO ShippingClerk;
GRANT SELECT ON ShippingDb.InventoryItems TO ShippingClerk;
GRANT SELECT (CustomerId, FirstName, LastName, Street, City, State,
             Zip, Phone) ON ShippingDb.Customers TO ShippingClerk;
```

Notice that the last statement grants privileges to select specific fields and doesn't let the clerk view the customer table's other fields such as CreditCardNumber.

4. Grant UPDATE privileges for the fields in the Customers, Orders, OrderItems, and InventoryItems tables that the clerk needs to properly record a shipment.

When shipping an order, the clerk must update the InventoryItems table's QuantityInStock field. The clerk must also update the Orders table's OrderStatus and ShippedDate fields. The following statements grant the necessary privileges:

```
GRANT UPDATE (QuantityInStock) ON ShippingDb.InventoryItems TO ShippingClerk;  
GRANT UPDATE (OrderStatus, ShippedDate) ON ShippingDb.Orders TO ShippingClerk;
```

You can download scripts that demonstrate these privileges from the book's Web site. The script `MakeShippingClerk.sql` builds a test database and uses the previous code snippets to create the ShippingClerk with the correct privileges. The `UseShippingClerk.sql` script performs the tasks that the shipping clerk would perform while shipping an order. The `DropShippingClerk.sql` script deletes the ShippingClerk account and the test database.

Transactions



In almost all applications that access MySQL databases, multiple users concurrently attempt to view and modify data.

The simultaneous operations may result in data that is inconsistent and inaccurate.

Using transactions avoid these problems by isolating each operation.

In the context of SQL and relational databases,

a transaction is a set of one or more SQL statements that perform a set of related actions.

The statements are grouped together and treated as a single unit whose success or failure depends on the successful execution of each statement in the transaction.

In addition to ensuring the proper execution of a set of SQL statements,

a transaction locks the tables involved in the transaction so that other users cannot modify any rows that are involved in the transaction.

ACID – generally, every transaction is ACID



(Atomicity-Consistency-Isolation-Durability) compliant in accordance with concurrency control and transparency in the DS. It is NOT required, but the best dbase transactions are 'ACID friendly'.

ATOMICITY – ALL or NO operations in a transaction are performed.

Transaction Recovery System – ensures atomicity and enables the all or nothing output.

CONSISTENCY – consistent state is maintained before a transaction starts and after it concludes.

ISOLATION – concurrent transactions DO NOT interfere with each other.

Partial results of incomplete transactions are not visible to others before the transactions are committed.

DURABILITY – transactions results are locked/ permanent after being committed.

System guarantees that results of a committed transaction will be permanent even if a failure occurs after the commit.

Starting a Transaction



MySQL provides the

- ❑ `START TRANSACTION`
to create a transaction, and
- ❑ `COMMIT`, `ROLLBACK` statements
to end it.
- ❑ The `COMMIT` statement
saves changes to the database
- ❑ The `ROLLBACK` statement
will undo any changes made during the transaction and database is
reverted to the pre-transaction state.

The START TRANSACTION Statement



The START TRANSACTION statement requires no clauses or options:

START TRANSACTION

START TRANSACTION statement notifies MySQL that the statements that follow should be treated as a unit, until the transaction ends, successfully or otherwise.

Note: A BEGIN statement can also be used to start a transaction.

Committing a Transaction




The COMMIT Statement

The COMMIT statement is used to terminate a transaction and to save all changes made by the transaction to the database. There are no additional clauses or options:

COMMIT

The following transaction is made of two INSERT statements, followed by a COMMIT statement:



```
START TRANSACTION;  
INSERT INTO Studio VALUES (101, 'MGM Studios');  
INSERT INTO Studio VALUES (102, 'Wannabe  
    Studios');  
COMMIT;  
SELECT * FROM Studio;
```

Ensure the table type is as of InnoDB before this trying this example.



```
START TRANSACTION;
```

```
UPDATE Studio SET title = 'Temporary Studios'  
WHERE id = 101;
```

```
UPDATE Studio SET title = 'Studio with no buildings'  
WHERE id = 102;
```

```
SELECT * FROM Studio;
```

```
ROLLBACK;
```

```
SELECT * FROM Studio;
```

Adding Savepoints to Your Transaction



The **SAVEPOINT** and **ROLLBACK TO SAVEPOINT** statements isolate portions of a transaction. The **SAVEPOINT** statement defines a marker in a transaction, and the **ROLLBACK TO SAVEPOINT** statement allows you to roll back a transaction to a predetermined marker (savepoint).



Start Transactions;

SAVEPOINT savepoint1

INSERT INTO Studio VALUES (105, 'Noncomformant Studios');

INSERT INTO Studio VALUES (106, 'Studio Cartel');

SELECT * FROM Studio;

ROLLBACK TO SAVEPOINT savepoint1;

INSERT INTO Studio VALUES (105, 'Moneymaking Studios');

INSERT INTO Studio VALUES (106, 'Studio Mob');

SELECT * FROM Studio;

COMMIT;



A *dirty read* can take place when:


Transaction A modifies data in a table.

Around the same time, another Transaction B reads the table, *before* those modifications are committed to the database.

Transaction A rolls back (cancels) the changes, returning the database to its original state.

Transaction B now has data inconsistent with the database.

Worse, Transaction B may modify the data based on its initial read, which is incorrect or *dirty read*.




The transaction isolation level determine the degree to which other transactions can “see” details inside an in-progress transaction and are arranged in hierarchical order.

Serializable

Repeatable Read

Read Committed


Read Uncommitted



SERIALIZABLE — In the **SERIALIZABLE** isolation level of MySQL, data reads are implicitly run with a read lock

REPEATABLE READ

provides more isolation from a transaction, ensuring that data reads are the same throughout the transaction even if the data has been changed and committed by a different transaction. The **SERIALIZABLE** isolation level provides the slowest performance but also the most isolation



READ COMMITTED provides some isolation and slightly slower performance, because only committed data changes are seen by other transactions. However, READ COMMITTED does not address the issue of data changing in the middle of a transaction.

READ UNCOMMITTED is the easiest isolation level to implement and provides the fastest performance. The problem with READ UNCOMMITTED is that it provides no isolation between transactions.

Locking Nontransactional Tables

- ❑ MySQL supports the use of transactions only for InnoDB and BDB tables.
- ❑ There might be times, though, when you want to lock other types of tables that are included in your database.
- ❑ By locking nontransactional tables manually, you can group SQL statements together and set up a transaction-like operation in order to prevent anyone from changing the tables that participate in your operation.
- ❑ To lock a nontransactional table, you must use the LOCK TABLES statement.
- ❑ Once you've completed updating the tables, you should use the UNLOCK TABLES statement to release them.

The **LOCK TABLES** Statement

To lock a table in a MySQL database, you should use the `LOCK TABLES` statement, as shown in the following syntax:

```
LOCK {TABLE | TABLES}
<table name> [AS <alias>] {READ [LOCAL] | [LOW_PRIORITY] WRITE}
[[, <table name> [AS <alias>] {READ [LOCAL] | [LOW_PRIORITY] WRITE}}...]
```

To use the statement, you must specify the `LOCK` keyword, the `TABLE` or `TABLES` keyword, one or more tables, and the type of lock for each table. The `TABLE` and `TABLES` keywords are synonymous, and you can use either one, whether you're locking one table or multiple tables. Generally, `TABLE` is used for one table, and `TABLES` is used for multiple tables, but either can be used in either situation.

For each table that you specify, you have the option of providing an alias for the table. In addition, you must specify a `READ` lock type or a `WRITE` lock type. If you specify `READ`, any connection can read from the table, but no connection can write to the table. If you specify `READ LOCAL`, nonconflicting `INSERT` statements can be executed by any connection. If you specify `WRITE`, the current connection can read or write to the table, but no other connections can access the table until the lock has been removed. If you specify `LOW_PRIORITY WRITE`, other connections can obtain `READ` locks while the current session is waiting for the `WRITE` lock.

Once you lock a table, it remains locked until you explicitly unlock the table with the `UNLOCK TABLES` statement (described in the text that follows) or end your current session.

Now take a look at an example of a `LOCK TABLE` statement. The following statement places a lock on the `Books` table:

```
LOCK TABLE Books READ;
```

As you can see, a `READ` lock has been placed on the table. Now only read access is available to all connections. You are not limited, however, to placing a lock on only one table. For example, the following `LOCK TABLES` statement places locks on the `Books` and `BookOrders` tables:

```
LOCK TABLES Books READ, BookOrders WRITE;
```

In this case, a `READ` lock is placed on the `Books` table, and a `WRITE` lock is placed on the `BookOrders` table. As a result, other connections can read from the `Books` table, but they cannot access the `BookOrders` table.

The **UNLOCK TABLES** Statement

Once you've completed accessing a locked table, you should explicitly unlock the table or end your current session. To unlock one or more locked tables, you must use the `UNLOCK TABLES` statement, shown in the following syntax:

```
UNLOCK {TABLE | TABLES}
```

As you can see, you must specify the `UNLOCK` keyword along with the `TABLE` or `TABLES` keyword. As with the `LOCK TABLES` statement, the `TABLE` and `TABLES` keywords are synonymous, which means that you can use either one, regardless of the number of tables that you're unlocking.

One thing to notice about the `UNLOCK TABLES` statement is that no table names are specified. When you use this statement, all tables that have been locked from within the current session are unlocked. For example, to unlock the `Books` and `BookOrders` tables, you would use a statement similar to the following:

```
UNLOCK TABLES;
```

If only one table is locked, you can still use this statement, although you can also use the `TABLE` keyword, rather than `TABLES`. Either way, any locked tables are now unlocked.

In general, you should try to use InnoDB tables when setting up your system to support transactions. In the case of the `DVDRentals` database, all tables are defined as InnoDB. Should you ever run into a situation in which you want to lock another type of table manually, you can use the `LOCK TABLES` and `UNLOCK TABLES` statements.