Architecture of Enterprise Applications 13
MySQL Backup & Recovery

Haopeng Chen

RE*liable,* ***IN****telligent and* ***S****calable Systems Group (***REINS***)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Contents
  - Backup and Recovery Types
  - Database Backup Methods
  - Example Backup and Recovery Strategy
  - Using mysqldump for Backups
  - Point-in-Time (Incremental) Recovery
  - MyISAM Table Maintenance and Crash Recovery
  - From: https://dev.mysql.com/doc/refman/8.0/en/backup-and-recovery.html
- Objectives
  - 能够根据数据访问的具体场景，设计提高数据库访问性能和灾备能力的方案，包括集群部署和备份机制

Backup and Recovery

- It is important to back up your databases
  - so that you can recover your data and be up and running again in case problems occur, such as system crashes, hardware failures, or users deleting data by mistake.
  - Backups are also essential as a safeguard before upgrading a MySQL installation, and they can be used to transfer a MySQL installation to another system or to set up replica servers.
- Several backup and recovery topics with which you should be familiar:
  - Types of backups: Logical versus physical, full versus incremental, and so forth.
  - Methods for creating backups.
  - Recovery methods, including point-in-time recovery.
  - Backup scheduling, compression, and encryption.
  - Table maintenance, to enable recovery of corrupt tables.

Backup and Recovery Types

- Physical (Raw) Versus Logical Backups
  - Physical backups consist of raw copies of the directories and files that store database contents.
    - This type of backup is suitable for large, important databases that need to be recovered quickly when problems occur.
  - Logical backups save information represented as logical database structure (CREATE DATABASE, CREATE TABLE statements) and content (INSERT statements or delimited-text files).
    - This type of backup is suitable for smaller amounts of data where you might edit the data values or table structure, or recreate the data on a different machine architecture.

Backup and Recovery Types

- Physical backup methods have these characteristics:
  - The backup consists of exact copies of database directories and files. Typically this is a copy of all or part of the MySQL data directory.
  - Physical backup methods are faster than logical because they involve only file copying without conversion.
  - Output is more compact than for logical backup.
  - Because backup speed and compactness are important for busy, important databases, the MySQL Enterprise Backup product performs physical backups.
  - Backup and restore granularity ranges from the level of the entire data directory down to the level of individual files. This may or may not provide for table-level granularity, depending on storage engine.
    - For example, InnoDB tables can each be in a separate file, or share file storage with other InnoDB tables; each MyISAM table corresponds uniquely to a set of files.
  - In addition to databases, the backup can include any related files such as log or configuration files.

Backup and Recovery Types

- Physical backup methods have these characteristics:
  - Data from MEMORY tables is tricky to back up this way
    - because their contents are not stored on disk.
  - Backups are portable only to other machines that have identical or similar hardware characteristics.
  - Backups can be performed while the MySQL server is not running.
    - If the server is running, it is necessary to perform appropriate locking so that the server does not change database contents during the backup.
    - MySQL Enterprise Backup does this locking automatically for tables that require it.
  - Physical backup tools include the **mysqlbackup** of MySQL Enterprise Backup for InnoDB or any other tables, or file system-level commands (such as **cp**, **scp**, **tar**, **rsync**) for MyISAM tables.
  - For restore:
    - MySQL Enterprise Backup restores InnoDB and other tables that it backed up.
    - **ndb_restore** restores NDB tables.
    - Files copied at the file system level can be copied back to their original locations with file system commands.

Backup and Recovery Types

- Logical backup methods have these characteristics:
  - The backup is done by querying the MySQL server to obtain database structure and content information.
  - Backup is slower than physical methods because the server must access database information and convert it to logical format.
  - Output is larger than for physical backup, particularly when saved in text format.
  - Backup and restore granularity is available at the server level (all databases), database level (all tables in a particular database), or table level.
  - The backup does not include log or configuration files, or other database-related files that are not part of databases.
  - Backups stored in logical format are machine independent and highly portable.
  - Logical backups are performed with the MySQL server running.
  - Logical backup tools include the **mysqldump** program and the SELECT ... INTO OUTFILE statement. These work for any storage engine, even MEMORY.
  - To restore logical backups, SQL-format dump files can be processed using the **mysql** client. To load delimited-text files, use the LOAD DATA statement or the **mysqlimport** client.

Backup and Recovery Types

- Online Versus Offline Backups
  - Online backups take place while the MySQL server is running so that the database information can be obtained from the server.
  - Offline backups take place while the server is stopped.
  - This distinction can also be described as "hot" versus "cold" backups;
    - a "warm" backup is one where the server remains running but locked against modifying data while you access database files externally.
  - Online backup methods have these characteristics:
    - The backup is less intrusive to other clients, which can connect to the MySQL server during the backup and may be able to access data depending on what operations they need to perform.
    - Care must be taken to impose appropriate locking so that data modifications do not take place that would compromise backup integrity. The MySQL Enterprise Backup product does such locking automatically.

Backup and Recovery Types

- Online Versus Offline Backups
  - Offline backup methods have these characteristics:
    - Clients can be affected adversely because the server is unavailable during backup. For that reason, such backups are often taken from a replica that can be taken offline without harming availability.
    - The backup procedure is simpler because there is no possibility of interference from client activity.
  - A similar distinction between online and offline applies for recovery operations, and similar characteristics apply.
    - However, it is more likely for clients to be affected by online recovery than by online backup because recovery requires stronger locking.
    - During backup, clients might be able to read data while it is being backed up. Recovery modifies data and does not just read it, so clients must be prevented from accessing data while it is being restored.

Backup and Recovery Types

- Local Versus Remote Backups
  - A local backup is performed on the same host where the MySQL server runs,
    - whereas a remote backup is done from a different host. For some types of backups, the backup can be initiated from a remote host even if the output is written locally on the server host.
    - **mysqldump** can connect to local or remote servers.
      - For SQL output (CREATE and INSERT statements), local or remote dumps can be done and generate output on the client.
      - For delimited-text output (with the --tab option), data files are created on the server host.
    - SELECT ... INTO OUTFILE can be initiated from a local or remote client host, but the output file is created on the server host.
    - Physical backup methods typically are initiated locally on the MySQL server host so that the server can be taken offline, although the destination for copied files might be remote.

Backup and Recovery Types

- Snapshot Backups
  - Some file system implementations enable “snapshots” to be taken.
  - These provide logical copies of the file system at a given point in time, without requiring a physical copy of the entire file system.
    - (For example, the implementation may use copy-on-write techniques so that only parts of the file system modified after the snapshot time need be copied.)
  - MySQL itself does not provide the capability for taking file system snapshots.
    - It is available through third-party solutions such as Veritas, LVM, or ZFS.

Backup and Recovery Types

- Full Versus Incremental Backups
  - A full backup includes all data managed by a MySQL server at a given point in time.
  - An incremental backup consists of the changes made to the data during a given time span (from one point in time to another).
  - MySQL has different ways to perform full backups.
  - Incremental backups are made possible by enabling the server's binary log, which the server uses to record data changes.

Backup and Recovery Types

- Full Versus Point-in-Time (Incremental) Recovery
  - A full recovery restores all data from a full backup.
    - This restores the server instance to the state that it had when the backup was made.
    - If that state is not sufficiently current, a full recovery can be followed by recovery of incremental backups made since the full backup, to bring the server to a more up-to-date state.
  - Incremental recovery is recovery of changes made during a given time span.
    - This is also called point-in-time recovery because it makes a server's state current up to a given time. Point-in-time recovery is based on the binary log and typically follows a full recovery from the backup files that restores the server to its state when the backup was made.
    - Then the data changes written in the binary log files are applied as incremental recovery to redo data modifications and bring the server up to the desired point in time.

Backup and Recovery Types

- Backup Scheduling, Compression, and Encryption
  - Backup scheduling is valuable for automating backup procedures.
  - Compression of backup output reduces space requirements, and
  - encryption of the output provides better security against unauthorized access of backed-up data.
  - MySQL itself does not provide these capabilities.
    - The MySQL Enterprise Backup product can compress InnoDB backups, and compression or encryption of backup output can be achieved using file system utilities. Other third-party solutions may be available.

Database Backup Methods

- Making a Hot Backup with MySQL Enterprise Backup
  - Customers of MySQL Enterprise Edition can use the MySQL Enterprise Backup product to do physical backups of entire instances or selected databases, tables, or both.
  - This product includes features for incremental and compressed backups.
  - Backing up the physical database files makes restore much faster than logical techniques such as the mysqldump command.
  - InnoDB tables are copied using a hot backup mechanism.
    - (Ideally, the InnoDB tables should represent a substantial majority of the data.)
  - Tables from other storage engines are copied using a warm backup mechanism.

Database Backup Methods

- Making Backups with mysqldump
  - The **mysqldump** program can make backups. It can back up all kinds of tables.
  - For InnoDB tables, it is possible to perform an online backup that takes no locks on tables using the --single-transaction option to **mysqldump**.

- Making Backups by Copying Table Files
  - MyISAM tables can be backed up by copying table files (*.MYD, *.MYI files, and associated *.sdi files). To get a consistent backup, stop the server or lock and flush the relevant tables:
    FLUSH TABLES *tbl_list* WITH READ LOCK;
  - You need only a read lock; this enables other clients to continue to query the tables while you are making a copy of the files in the database directory.
  - The flush is needed to ensure that the all active index pages are written to disk before you start the backup.
  - You can also create a binary backup simply by copying the table files, as long as the server isn't updating anything.
  - (But note that table file copying methods do not work if your database contains InnoDB tables. Also, even if the server is not actively updating data, InnoDB may still have modified data cached in memory and not flushed to disk.)

Database Backup Methods

- Making Delimited-Text File Backups
  - To create a text file containing a table's data, you can use SELECT * INTO OUTFILE '*file_name*' FROM *tbl_name*.
  - The file is created on the MySQL server host, not the client host. For this statement, the output file cannot already exist because permitting files to be overwritten constitutes a security risk.
  - This method works for any kind of data file, but saves only table data, not the table structure.
  - Another way to create text data files (along with files containing CREATE TABLE statements for the backed up tables) is to use **mysqldump** with the --tab option.
  - To reload a delimited-text data file, use LOAD DATA or **mysqlimport**.

Database Backup Methods

- Making Incremental Backups by Enabling the Binary Log
  - MySQL supports incremental backups using the binary log.
  - The binary log files provide you with the information you need to replicate changes to the database that are made subsequent to the point at which you performed a backup.
  - At the moment you want to make an incremental backup (containing all changes that happened since the last full or incremental backup), you should rotate the binary log by using FLUSH LOGS.
  - The next time you do a full backup, you should also rotate the binary log using FLUSH LOGS or **mysqldump --flush-logs**.

Database Backup Methods

- Making Backups Using Replicas
  - If you have performance problems with a server while making backups, one strategy that can help is to set up replication and perform backups on the replica rather than on the source.
  - If you are backing up a replica, you should back up its connection metadata repository and applier metadata repository when you back up the replica's databases, regardless of the backup method you choose.
  - This information is always needed to resume replication after you restore the replica's data.
    - If your replica is replicating LOAD DATA statements, you should also back up any SQL_LOAD-* files that exist in the directory that the replica uses for this purpose.
    - The replica needs these files to resume replication of any interrupted LOAD DATA operations.

Database Backup Methods

- Recovering Corrupt Tables
  - If you have to restore MyISAM tables that have become corrupt, try to recover them using REPAIR TABLE or **myisamchk -r** first. That should work in 99.9% of all cases.
- Making Backups Using a File System Snapshot
  - If you are using a Veritas file system, you can make a backup like this:
    - From a client program, execute FLUSH TABLES WITH READ LOCK.
    - From another shell, execute mount vxfs snapshot.
    - From the first client, execute UNLOCK TABLES.
    - Copy files from the snapshot.
    - Unmount the snapshot.
  - Similar snapshot capabilities may be available in other file systems, such as LVM or ZFS.

Example Backup and Recovery Strategy

- Let's discusse a procedure for performing backups that enables you to recover data after several types of crashes:
  - Operating system crash
  - Power failure
  - File system crash
  - Hardware problem (hard drive, motherboard, and so forth)

- Assume that data is stored in the InnoDB storage engine, which has support for transactions and automatic crash recovery.
- Assume also that the MySQL server is under load at the time of the crash. If it were not, no recovery would ever be needed.

Example Backup and Recovery Strategy

- For cases of operating system crashes or power failures, we can assume that MySQL's disk data is available after a restart.
  - The InnoDB data files might not contain consistent data due to the crash, but InnoDB reads its logs and finds in them the list of pending committed and noncommitted transactions that have not been flushed to the data files.
  - InnoDB automatically rolls back those transactions that were not committed, and flushes to its data files those that were committed.
- For the cases of file system crashes or hardware problems, we can assume that the MySQL disk data is *not* available after a restart.
  - This means that MySQL fails to start successfully because some blocks of disk data are no longer readable.
  - In this case, it is necessary to reformat the disk, install a new one, or otherwise correct the underlying problem.
  - Then it is necessary to recover our MySQL data from backups, which means that backups must already have been made. To make sure that is the case, design and implement a backup policy.

Example Backup and Recovery Strategy

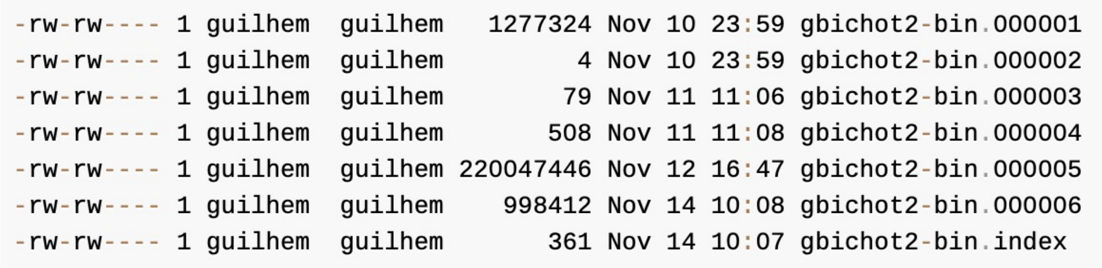
- Establishing a Backup Policy
  - To be useful, backups must be scheduled regularly.
  - Assume that we make a full backup of all our InnoDB tables in all databases using the following command on Sunday at 1 p.m., when load is low:

    ```
    shell> mysqldump --all-databases --master-data --single-transaction > backup_sunday_1_PM.sql
    ```

  - The resulting .sql file produced by **mysqldump** contains a set of SQL INSERT statements that can be used to reload the dumped tables at a later time.
  - This backup operation acquires a global read lock on all tables at the beginning of the dump.
  - Full backups are necessary, but it is not always convenient to create them.

Example Backup and Recovery Strategy

- Establishing a Backup Policy
  - To make incremental backups, we need to save the incremental changes.
  - In MySQL, these changes are represented in the binary log,
    - so the MySQL server should always be started with the --log-bin option to enable that log. With binary logging enabled, the server writes each data change into a file while it updates data.



```
-rw-rw---- 1 guilhem  guilhem   1277324 Nov 10 23:59 gbichot2-bin.000001
-rw-rw---- 1 guilhem  guilhem         4 Nov 10 23:59 gbichot2-bin.000002
-rw-rw---- 1 guilhem  guilhem        79 Nov 11 11:06 gbichot2-bin.000003
-rw-rw---- 1 guilhem  guilhem       508 Nov 11 11:08 gbichot2-bin.000004
-rw-rw---- 1 guilhem  guilhem 220047446 Nov 12 16:47 gbichot2-bin.000005
-rw-rw---- 1 guilhem  guilhem    998412 Nov 14 10:08 gbichot2-bin.000006
-rw-rw---- 1 guilhem  guilhem       361 Nov 14 10:07 gbichot2-bin.index
```

  - The MySQL binary logs take up disk space. To free up space, purge them from time to time:

```
shell> mysqldump --single-transaction --flush-logs --master-data=2 \ -
-all-databases --delete-master-logs > backup_sunday_1_PM.sql
```

Example Backup and Recovery Strategy

- Using Backups for Recovery
  - Now, suppose that we have a catastrophic unexpected exit on Wednesday at 8 a.m. that requires recovery from backups.
  - To recover, first we restore the last full backup we have (the one from Sunday 1 p.m.):

```
shell> mysql < backup_sunday_1_PM.sql
```

  - At this point, the data is restored to its state as of Sunday 1 p.m.
  - To restore the changes made since then, we must use the incremental backups; that is, the gbichot2-bin.000007 and gbichot2-bin.000008 binary log files.
  - Fetch the files if necessary from where they were backed up, and then process their contents like this:

```
shell> mysqlbinlog gbichot2-bin.000007 gbichot2-bin.000008 | mysql
```

Example Backup and Recovery Strategy

- Using Backups for Recovery
  - We now have recovered the data to its state as of Tuesday 1 p.m., but still are missing the changes from that date to the date of the crash.
  - To not lose them, we would have needed to have the MySQL server store its MySQL binary logs into a safe location (RAID disks, SAN, ...) different from the place where it stores its data files, so that these logs were not on the destroyed disk.
  - (That is, we can start the server with a --log-bin option that specifies a location on a different physical device from the one on which the data directory resides.)
  - If we had done this, we would have the gbichot2-bin.000009 file (and any subsequent files) at hand, and we could apply them using **mysqlbinlog** and **mysql** to restore the most recent data changes with no loss up to the moment of the crash:

```
shell> mysqlbinlog gbichot2-bin.000009 ... | mysql
```

Example Backup and Recovery Strategy

- Backup Strategy Summary
  - In case of an operating system crash or power failure, InnoDB itself does all the job of recovering data. But to make sure that you can sleep well, observe the following guidelines:
  - Always tun the MySQL server with binary logging enabled (that is the default setting for MySQL 8.0).
    - If you have such safe media, this technique can also be good for disk load balancing (which results in a performance improvement).
  - Make periodic full backups, using the **mysqldump** command that makes an online, nonblocking backup.
  - Make periodic incremental backups by flushing the logs with FLUSH LOGS or **mysqladmin flush-logs**.

Using mysqldump for Backups

- Consider using the MySQL Shell dump utilities, which provide
  - parallel dumping with multiple threads, file compression, and progress information display, as well as cloud features such as Oracle Cloud Infrastructure Object Storage streaming, and MySQL Database Service compatibility checks and modifications.
  - Dumps can be easily imported into a MySQL Server instance or a MySQL Database Service DB System using the MySQL Shell load dump utilities.
- A dump file can be used in several ways:
  - As a backup to enable data recovery in case of data loss.
  - As a source of data for setting up replicas.
  - As a source of data for experimentation:
    - To make a copy of a database that you can use without changing the original data.
    - To test potential upgrade incompatibilities.

Using mysqldump for Backups

- **mysqldump** produces two types of output, depending on whether the --tab option is given:
  - Without --tab, **mysqldump** writes SQL statements to the standard output.
    - This output consists of CREATE statements to create dumped objects (databases, tables, stored routines, and so forth), and INSERT statements to load data into tables.
    - The output can be saved in a file and reloaded later using **mysql** to recreate the dumped objects. Options are available to modify the format of the SQL statements, and to control which objects are dumped.
  - With --tab, **mysqldump** produces two output files for each dumped table.
    - The server writes one file as tab-delimited text, one line per table row. This file is named *tbl_name*.txt in the output directory.
    - The server also sends a CREATE TABLE statement for the table to **mysqldump**, which writes it as a file named *tbl_name*.sql in the output directory.

Using mysqldump for Backups

- Dumping Data in SQL Format with mysqldump
  - By default, mysqldump writes information as SQL statements to the standard output. You can save the output in a file:

```
shell> mysqldump [arguments] > file_name
```

  - To dump all databases, invoke mysqldump with the --all-databases option:

```
shell> mysqldump --all-databases > dump.sql
```

  - To dump only specific databases, name them on the command line and use the --databases option:

```
shell> mysqldump --databases db1 db2 db3 > dump.sql
```

  - To dump a single database, name it on the command line:

```
shell> mysqldump --databases test > dump.sql
```

  - In the single-database case, it is permissible to omit the --databases option:

```
shell> mysqldump test > dump.sql
```

  - To dump only specific tables from a database, name them on the command line following the database name:

```
shell> mysqldump test t1 t3 t7 > dump.sql
```

Using mysqldump for Backups

- Reloading SQL-Format Backups
  - If the dump file was created by **mysqldump** with the --all-databases or --databases option, it contains CREATE DATABASE and USE statements and it is not necessary to specify a default database into which to load the data:

    ```
    shell> mysql < dump.sql
    ```

  - Alternatively, from within **mysql**, use a source command:

    ```
    mysql> source dump.sql
    ```

  - If the file is a single-database dump not containing CREATE DATABASE and USE statements, create the database first (if necessary):

    ```
    shell> mysqladmin create db1
    ```

  - Then specify the database name when you load the dump file:

    ```
    shell> mysql db1 < dump.sql
    ```

  - Alternatively, from within **mysql**, create the database, select it as the default database, and load the dump file:

    ```
    mysql> CREATE DATABASE IF NOT EXISTS db1; mysql> USE db1; mysql> source
    dump.sql
    ```

Using mysqldump for Backups

- Dumping Data in Delimited-Text Format with mysqldump
  - If you invoke **mysqldump** with the --tab=*dir_name* option, it uses *dir_name* as the output directory and dumps tables individually in that directory using two files for each table. The table name is the base name for these files.
    - For a table named t1, the files are named t1.sql and t1.txt. The .sql file contains a CREATE TABLE statement for the table. The .txt file contains the table data, one line per table row.
  - The following command dumps the contents of the db1 database to files in the /tmp database:

```
shell> mysqldump --tab=/tmp db1
```

Using mysqldump for Backups

- Reloading Delimited-Text Format Backups
  - To reload a table, first change location into the output directory. Then process the .sql file with **mysql** to create an empty table and process the .txt file to load the data into the table:

    ```
    shell> mysql db1 < t1.sql
    shell> mysqlimport db1 t1.txt
    ```

  - An alternative to using **mysqlimport** to load the data file is to use the LOAD DATA statement from within the **mysql** client:

    ```
    mysql> USE db1;
    mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1;
    ```

  - If you used any data-formatting options with **mysqldump** when you initially dumped the table, you must use the same options with **mysqlimport** or LOAD DATA to ensure proper interpretation of the data file contents:

    ```
    shell> mysqlimport --fields-terminated-by=, --fields-enclosed-by='"' --lines-terminated-
    by=0x0d0a db1 t1.txt
    ```

  - Or:

    ```
    mysql> USE db1;
    mysql> LOAD DATA INFILE 't1.txt' INTO TABLE t1 FIELDS TERMINATED BY ',' FIELDS ENCLOSED
    BY '"' LINES TERMINATED BY '\r\n';
    ```

mysqldump Tips

- How to make a copy a database
- How to copy a database from one server to another
- How to dump stored programs (stored procedures and functions, triggers, and events)
- How to dump definitions and data separately

mysqldump Tips

- Making a Copy of a Database
  - `shell> mysqldump db1 > dump.sql`
  - `shell> mysqladmin create db2`
  - `shell> mysql db2 < dump.sql`

  - Do not use --databases on the **mysqldump** command line because that causes USE db1 to be included in the dump file, which overrides the effect of naming db2 on the **mysql** command line.

mysqldump Tips

- Copy a Database from one Server to Another
  - On Server 1:

```
shell> mysqldump --databases db1 > dump.sql
```

  - Copy the dump file from Server 1 to Server 2.
  - On Server 2:

```
shell> mysql < dump.sql
```

  - Alternatively, you can omit --databases from the **mysqldump** command.
  - On Server 1:

```
shell> mysqldump db1 > dump.sql
```

  - On Server 2:

```
shell> mysqladmin create db1
shell> mysql db1 < dump.sql
```

mysqldump Tips

- Dumping Stored Programs
  - Several options control how **mysqldump** handles stored programs (stored procedures and functions, triggers, and events):
    - --events: Dump Event Scheduler events
    - --routines: Dump stored procedures and functions
    - --triggers: Dump triggers for tables
  - The --triggers option is enabled by default so that when tables are dumped, they are accompanied by any triggers they have.
    - The other options are disabled by default and must be specified explicitly to dump the corresponding objects.
    - To disable any of these options explicitly, use its skip form: --skip-events, --skip-routines, or --skip-triggers.

mysqldump Tips

- Dumping Table Definitions and Content Separately
  - The --no-data option tells **mysqldump** not to dump table data, resulting in the dump file containing only statements to create the tables.
    - Conversely, the --no-create-info option tells **mysqldump** to suppress CREATE statements from the output, so that the dump file contains only table data.
  - For example, to dump table definitions and data separately for the test database, use these commands:

```
shell> mysqldump --no-data test > dump-defs.sql
shell> mysqldump --no-create-info test > dump-data.sql
```

  - For a definition-only dump, add the --routines and --events options to also include stored routine and event definitions:

```
shell> mysqldump --no-data --routines --events test > dump-defs.sql
```

mysqldump Tips

- Using mysqldump to Test for Upgrade Incompatibilities
  - When contemplating a MySQL upgrade, it is prudent to install the newer version separately from your current production version.
    - Then you can dump the database and database object definitions from the production server and load them into the new server to verify that they are handled properly. (This is also useful for testing downgrades.)
  - On the production server:

```
shell> mysqldump --all-databases --no-data --routines --events > dump-defs.sql
```

  - On the upgraded server:

```
shell> mysql < dump-defs.sql
```

  - Because the dump file does not contain table data, it can be processed quickly.

mysqldump Tips

- Using mysqldump to Test for Upgrade Incompatibilities
  - This enables you to spot potential incompatibilities without waiting for lengthy data-loading operations. Look for warnings or errors while the dump file is being processed.
  - After you have verified that the definitions are handled properly, dump the data and try to load it into the upgraded server.
  - On the production server:

```
shell> mysqldump --all-databases --no-create-info > dump-data.sql
```

  - On the upgraded server:

```
shell> mysql < dump-data.sql
```

  - Now check the table contents and run some test queries.

Point-in-Time (Incremental) Recovery

- Point-in-time recovery refers to
  - recovery of data changes up to a given point in time.
  - Typically, this type of recovery is performed after restoring a full backup that brings the server to its state as of the time the backup was made.
  - Point-in-time recovery then brings the server up to date incrementally from the time of the full backup to a more recent time.

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Binary Log
  - To restore data from the binary log, you must know the name and location of the current binary log files.

    ```
    mysql> SHOW BINARY LOGS;
    ```

  - To determine the name of the current binary log file, issue the following statement:

    ```
    mysql> SHOW MASTER STATUS;
    ```

  - To apply events from the binary log, process **mysqlbinlog** output using the **mysql** client:

    ```
    shell> mysqlbinlog binlog_files | mysql -u root -p
    ```

  - If binary log files have been encrypted, which can be done from MySQL 8.0.14 onwards, **mysqlbinlog** cannot read them directly as in the above example, but can read them from the server using the --read-from-remote-server (-R) option. For example:

    ```
    shell> mysqlbinlog --read-from-remote-server --host=host_name --
    port=3306 --user=root --password --ssl-mode=required binlog_files |
    mysql -u root -p
    ```

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Binary Log
  - To view events from the log, send mysqlbinlog output into a paging program:

    ```
    shell> mysqlbinlog binlog_files | more
    ```

  - Alternatively, save the output in a file and view the file in a text editor:

    ```
    shell> mysqlbinlog binlog_files > tmpfile
    shell> ... edit tmpfile ...
    ```

  - Saving the output in a file is useful as a preliminary to executing the log contents with certain events removed, such as an accidental DROP TABLE.
  - You can delete from the file any statements not to be executed before executing its contents.
  - After editing the file, apply the contents as follows:

    ```
    shell> mysql -u root -p < tmpfile
    ```

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Binary Log
  - If you have more than one binary log to apply on the MySQL server, the safe method is to process them all using a single connection to the server.
  - Here is an example that demonstrates what may be *unsafe*:

```
shell> mysqlbinlog binlog.000001 | mysql -u root -p # DANGER!!
shell> mysqlbinlog binlog.000002 | mysql -u root -p # DANGER!!
```

  - Processing binary logs this way using different connections to the server causes problems if the first log file contains a CREATE TEMPORARY TABLE statement and the second log contains a statement that uses the temporary table.
    - When the first **mysql** process terminates, the server drops the temporary table.
    - When the second **mysql** process attempts to use the table, the server reports “unknown table.”

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Binary Log
  - To avoid problems like this, use a *single* connection to apply the contents of all binary log files that you want to process.
  - Here is one way to do so:

    ```
    shell> mysqlbinlog binlog.000001 binlog.000002 | mysql -u root -p
    ```

  - Another approach is to write the whole log to a single file and then process the file:

    ```
    shell> mysqlbinlog binlog.000001 > /tmp/statements.sql
    shell> mysqlbinlog binlog.000002 >> /tmp/statements.sql
    shell> mysql -u root -p -e "source /tmp/statements.sql"
    ```

  - When writing to a dump file while reading back from a binary log containing GTIDs (“Replication with Global Transaction Identifiers”), use the --skip-gtids option with **mysqlbinlog**, like this:

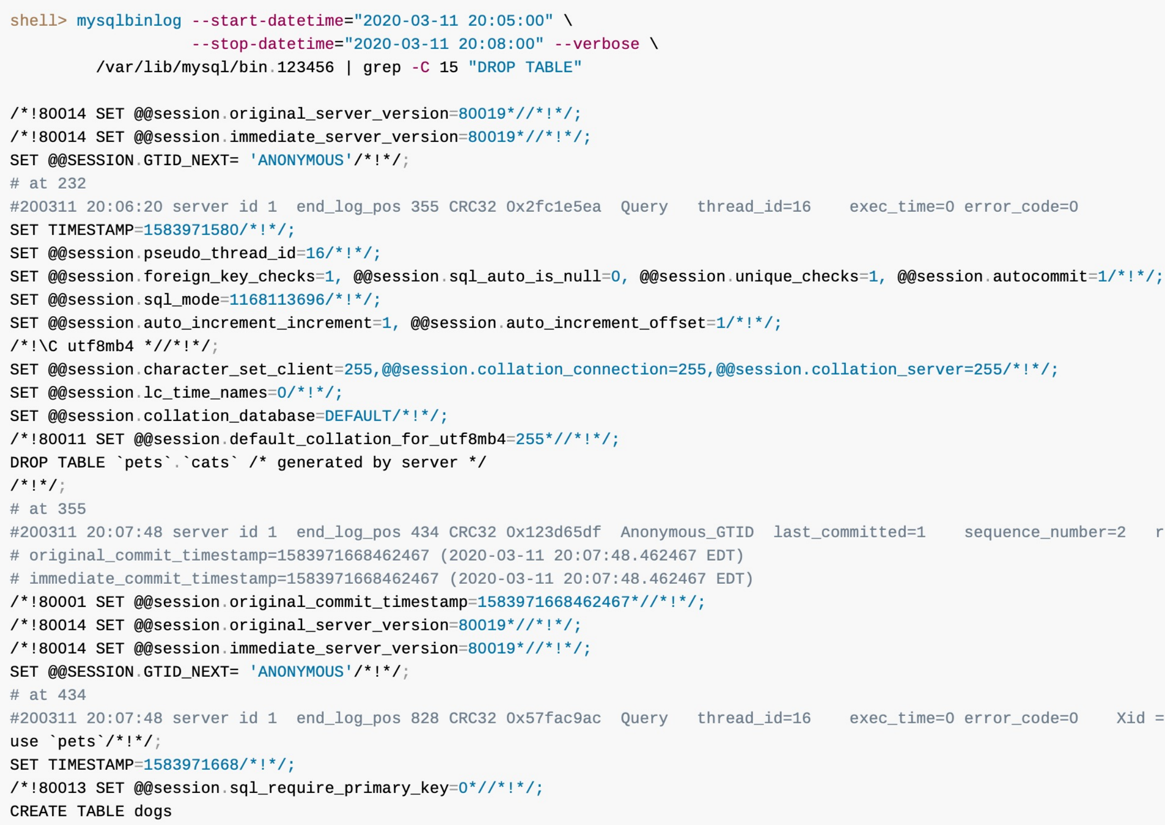
    ```
    shell> mysqlbinlog --skip-gtids binlog.000001 > /tmp/dump.sql
    shell> mysqlbinlog --skip-gtids binlog.000002 >> /tmp/dump.sql
    shell> mysql -u root -p -e "source /tmp/dump.sql"
    ```

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Event Positions
  - As an example, suppose that around 20:06:00 on March 11, 2020, an SQL statement was executed that deleted a table.
    - You can perform a point-in-time recovery to restore the server up to its state right before the table deletion.
  - These are some sample steps to achieve that:
    - Restore the last full backup created before the point-in-time of interest (call it t_p, which is 20:06:00 on March 11, 2020 in our example). When finished, note the binary log position up to which you have restored the server for later use, and restart the server.
    - Find the precise binary log event position corresponding to the point in time up to which you want to restore your database.
      - In our example, given that we know the rough time where the table deletion took place (t_p), we can find the log position by checking the log contents around that time using the **mysqlbinlog** utility.
      - Use the --start-datetime and --stop-datetime options to specify a short time period around t_p, and then look for the event in the output.

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Event Positions



```
shell> mysqlbinlog --start-datetime="2020-03-11 20:05:00" \
                   --stop-datetime="2020-03-11 20:08:00" --verbose \
         /var/lib/mysql/bin.123456 | grep -C 15 "DROP TABLE"

/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 232
#200311 20:06:20 server id 1  end_log_pos 355 CRC32 0x2fc1e5ea  Query   thread_id=16    exec_time=0 error_code=0
SET TIMESTAMP=1583971580/*!*/;
SET @@session.pseudo_thread_id=16/*!*/;
SET @@session.foreign_key_checks=1, @@session.sql_auto_is_null=0, @@session.unique_checks=1, @@session.autocommit=1/*!*/;
SET @@session.sql_mode=1168113696/*!*/;
SET @@session.auto_increment_increment=1, @@session.auto_increment_offset=1/*!*/;
/*!\C utf8mb4 *//*!*/;
SET @@session.character_set_client=255,@@session.collation_connection=255,@@session.collation_server=255/*!*/;
SET @@session.lc_time_names=0/*!*/;
SET @@session.collation_database=DEFAULT/*!*/;
/*!80011 SET @@session.default_collation_for_utf8mb4=255*//*!*/;
DROP TABLE `pets`.`cats` /* generated by server */
/*!*/;
# at 355
#200311 20:07:48 server id 1  end_log_pos 434 CRC32 0x123d65df  Anonymous_GTID  last_committed=1    sequence_number=2   r
# original_commit_timestamp=1583971668462467 (2020-03-11 20:07:48.462467 EDT)
# immediate_commit_timestamp=1583971668462467 (2020-03-11 20:07:48.462467 EDT)
/*!80001 SET @@session.original_commit_timestamp=1583971668462467*//*!*/;
/*!80014 SET @@session.original_server_version=80019*//*!*/;
/*!80014 SET @@session.immediate_server_version=80019*//*!*/;
SET @@SESSION.GTID_NEXT= 'ANONYMOUS'/*!*/;
# at 434
#200311 20:07:48 server id 1  end_log_pos 828 CRC32 0x57fac9ac  Query   thread_id=16    exec_time=0 error_code=0    Xid =
use `pets`/*!*/;
SET TIMESTAMP=1583971668/*!*/;
/*!80013 SET @@session.sql_require_primary_key=0*//*!*/;
CREATE TABLE dogs
```

Point-in-Time (Incremental) Recovery

- Point-in-Time Recovery Using Event Positions
  - Apply the events in binary log file to the server, starting with the log position your found in step 1 (assume it is 155) and ending at the position you have found in step 2 that is *before* your point-in-time of interest (which is 232):

    ```
    shell> mysqlbinlog --start-position=155 --stop-position=232
    /var/lib/mysql/bin.123456 \ | mysql -u root -p
    ```

  - Your database has now been restored to the point-in-time of interest, t_p, right before the table pets.cats was dropped.
  - Beyond the point-in-time recovery that has been finished, if you also want to reexecute all the statements *after* your point-in-time of interest, use **mysqlbinlog** again to apply all the events after t_p to the server.
  - We noted in step 2 that after the statement we wanted to skip, the log is at position 355; we can use it for the --start-position option, so that any statements after the position are included:

    ```
    shell> mysqlbinlog --start-position=355 /var/lib/mysql/bin.123456 \ | mysql
    -u root -p
    ```

Point-in-Time (Incremental) Recovery

- MyISAM Table Maintenance and Crash Recovery
  - You can use **myisamchk** to check, repair, or optimize database tables.
  - MyISAM table maintenance can also be done using the SQL statements that perform operations similar to what **myisamchk** can do:
    - To check MyISAM tables, use CHECK TABLE.
    - To repair MyISAM tables, use REPAIR TABLE.
    - To optimize MyISAM tables, use OPTIMIZE TABLE.
    - To analyze MyISAM tables, use ANALYZE TABLE.
  - One advantage of these statements over **myisamchk** is that the server does all the work.
    - With **myisamchk**, you must make sure that the server does not use the tables at the same time so that there is no unwanted interaction between **myisamchk** and the server.

Point-in-Time (Incremental) Recovery

- Using myisamchk for Crash Recovery
  - If you run mysqld with external locking disabled (which is the default), you cannot reliably use myisamchk to check a table when mysqld is using the same table.
    - If you can be certain that no one can access the tables using mysqld while you run myisamchk, you only have to execute mysqladmin flush-tables before you start checking the tables.
    - If you cannot guarantee this, you must stop mysqld while you check the tables.
    - If you run myisamchk to check tables that mysqld is updating at the same time, you may get a warning that a table is corrupt even when it is not.
  - If the server is run with external locking enabled, you can use myisamchk to check tables at any time.
    - In this case, if the server tries to update a table thatmyisamchk is using, the server waits for myisamchk to finish before it continues.

Point-in-Time (Incremental) Recovery

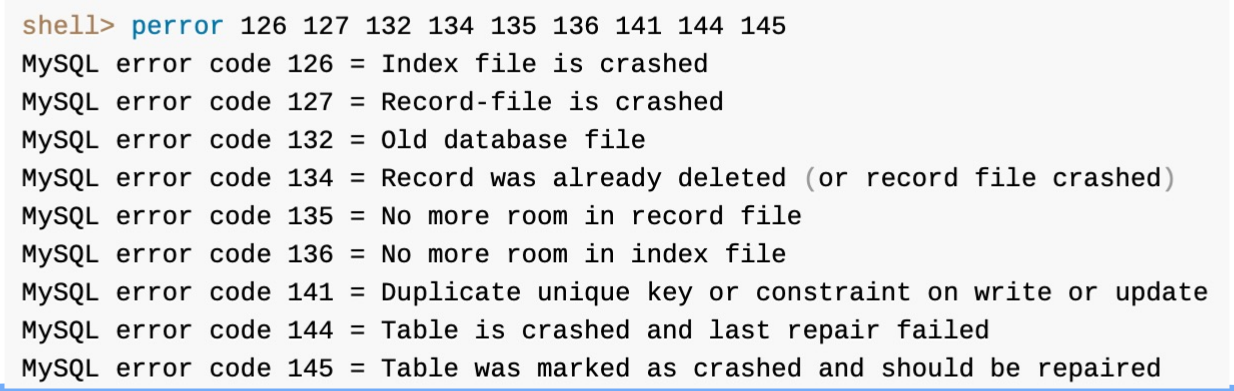
- Using myisamchk for Crash Recovery
  - If you use **myisamchk** to repair or optimize tables, you *must* always ensure that the **mysqld** server is not using the table (this also applies if external locking is disabled).
    - If you do not stop **mysqld**, you should at least do a **mysqladmin flush-tables** before you run **myisamchk**. Your tables *may become corrupted* if the server and **myisamchk** access the tables simultaneously.
  - **myisamchk** works by creating a copy of the .MYD data file row by row.
    - It ends the repair stage by removing the old .MYD file and renaming the new file to the original file name.
    - If you use --quick, **myisamchk** does not create a temporary .MYD file, but instead assumes that the .MYD file is correct and generates only a new index file without touching the .MYD file.
    - This is safe, because **myisamchk** automatically detects whether the .MYD file is corrupt and aborts the repair if it is.

Point-in-Time (Incremental) Recovery

- How to Check MyISAM Tables for Errors
  - **myisamchk *tbl_name***
    - This finds 99.99% of all errors. What it cannot find is corruption that involves *only* the data file (which is very unusual). If you want to check a table, you should normally run **myisamchk** without options or with the -s (silent) option.
  - **myisamchk -m *tbl_name***
    - This finds 99.999% of all errors. It first checks all index entries for errors and then reads through all rows. It calculates a checksum for all key values in the rows and verifies that the checksum matches the checksum for the keys in the index tree.
  - **myisamchk -e *tbl_name***
    - This does a complete and thorough check of all data (-e means "extended check"). It does a check-read of every key for each row to verify that they indeed point to the correct row. This may take a long time for a large table that has many indexes. Normally, **myisamchk** stops after the first error it finds. If you want to obtain more information, you can add the -v (verbose) option. This causes **myisamchk** to keep going, up through a maximum of 20 errors.
  - **myisamchk -e -i *tbl_name***
    - This is like the previous command, but the -i option tells **myisamchk** to print additional statistical information.
  - In most cases, a simple **myisamchk** command with no arguments other than the table name is sufficient to check a table.

Point-in-Time (Incremental) Recovery

- How to Repair MyISAM Tables
  - Symptoms of corrupted tables include queries that abort unexpectedly and observable errors such as these:
    - Can't find file *tbl_name*.MYI (Errcode: *nnn*)
    - Unexpected end of file
    - Record file is crashed
    - Got error *nnn* from table handler
  - To get more information about the error, run **perror** *nnn*, where *nnn* is the error number.:



```
shell> perror 126 127 132 134 135 136 141 144 145
MySQL error code 126 = Index file is crashed
MySQL error code 127 = Record-file is crashed
MySQL error code 132 = Old database file
MySQL error code 134 = Record was already deleted (or record file crashed)
MySQL error code 135 = No more room in record file
MySQL error code 136 = No more room in index file
MySQL error code 141 = Duplicate unique key or constraint on write or update
MySQL error code 144 = Table is crashed and last repair failed
MySQL error code 145 = Table was marked as crashed and should be repaired
```

Point-in-Time (Incremental) Recovery

- How to Repair MyISAM Tables
- ***Stage 1: Checking your tables***
  - Run **myisamchk *.MYI** or **myisamchk -e *.MYI** if you have more time. Use the -s (silent) option to suppress unnecessary information.
  - If the **mysqld** server is stopped, you should use the --update-state option to tell **myisamchk** to mark the table as "checked."
  - You have to repair only those tables for which **myisamchk** announces an error. For such tables, proceed to Stage 2.
  - If you get unexpected errors when checking (such as out of memory errors), or if **myisamchk** crashes, go to Stage 3.

Point-in-Time (Incremental) Recovery

- How to Repair MyISAM Tables
- ***Stage 2: Easy safe repair***
  - First, try **myisamchk -r -q *tbl_name*** (-r -q means “quick recovery mode”). This attempts to repair the index file without touching the data file. If the data file contains everything that it should and the delete links point at the correct locations within the data file, this should work, and the table is fixed. Start repairing the next table. Otherwise, use the following procedure:
    - Make a backup of the data file before continuing.
    - Use **myisamchk -r *tbl_name*** (-r means “recovery mode”). This removes incorrect rows and deleted rows from the data file and reconstructs the index file.
    - If the preceding step fails, use **myisamchk --safe-recover *tbl_name***. Safe recovery mode uses an old recovery method that handles a few cases that regular recovery mode does not (but is slower).
  - If you get unexpected errors when repairing (such as out of memory errors), or if **myisamchk** crashes, go to Stage 3.

Point-in-Time (Incremental) Recovery

- How to Repair MyISAM Tables
- ***Stage 3: Difficult repair***
  - You should reach this stage only if the first 16KB block in the index file is destroyed or contains incorrect information, or if the index file is missing. In this case, it is necessary to create a new index file. Do so as follows:
    - Move the data file to a safe place.
    - Use the table description file to create new (empty) data and index files:

```
shell> mysql db_name
mysql> SET autocommit=1;
mysql> TRUNCATE TABLE tbl_name;
mysql> quit
```

    - Copy the old data file back onto the newly created data file. (Do not just move the old file back onto the new file. You want to retain a copy in case something goes wrong.)
  - Go back to Stage 2. **myisamchk -r -q** should work. (This should not be an endless loop.)

作业七(第一部分)

- 请你根据上课内容，针对你在E-BookStore项目中的数据库设计，详细回答下列问题：
  1. 请你详细描述如何通过全量备份和增量备份来实现系统状态恢复。(2分)
  2. 请你根据MySQL缓存的工作原理，描述预取机制的优点。(1分)

  – 请提交包含上述问题答案的文档，答案应该结合你的E-BookStore的具体设计来阐述，不要过于泛化。

- 评分标准：
  – 分值如问题描述，答案不唯一，只要你的说理合理即可视为正确。

Thank You!