Assuming we have a database schema called ‘interswitch’, below are scripts that I will use execute multiple tasks;

Monitoring performance:

This script includes a few different queries to monitor the performance of the interswitch database. The first query shows the total size of the database, the second query shows the size of each table in the database, and the third query shows the top 10 queries by execution time.

The last few queries show the current status of the database, including the number of threads connected, the size of the InnoDB buffer pool, the number of read requests and reads from the buffer pool, and the number of pages flushed from the buffer pool. These metrics can help you identify potential performance issues and adjust your server configuration as needed.

```
USE interswitch;

-- Show the total size of the database

SELECT

  CONCAT(ROUND(SUM(data_length + index_length) / 1024 / 1024, 2), ' MB') AS `database_size`

FROM

  information_schema.tables

WHERE

  table_schema = 'interswitch';

-- Show the size of each table in the database

SELECT

  table_name AS `Table`,

  CONCAT(ROUND(data_length / 1024 / 1024, 2), ' MB') AS `Data Size`,

  CONCAT(ROUND(index_length / 1024 / 1024, 2), ' MB') AS `Index Size`

FROM

  information_schema.tables

WHERE

  table_schema = 'interswitch';
```

```
-- Show the top 10 queries by execution time

SELECT

  query,

  execution_count,

  ROUND(total_latency / 1000000000000,   6) AS `Total Latency (s)`,

  ROUND(mean_latency / 1000000000000,   6) AS `Mean Latency (s)`

FROM

  performance_schema.events_statements_summary_by_digest

WHERE

  schema_name = 'interswitch'

ORDER BY

  total_latency DESC

LIMIT 10;

-- Show the current status of the database

SHOW STATUS LIKE 'Threads_connected';

SHOW STATUS LIKE 'Innodb_buffer_pool_size';

SHOW STATUS LIKE 'Innodb_buffer_pool_read_requests';

SHOW STATUS LIKE 'Innodb_buffer_pool_reads';

SHOW STATUS LIKE 'Innodb_buffer_pool_pages_flushed';
```

Manage Database size:

This script creates a new table called "archived_transactions" to store transactions that are older than 36 months. It then moves any transactions that meet this criteria from the "transactions" table to the "archived_transactions" table using an INSERT INTO...SELECT statement.

Next, the script deletes the archived transactions from the "transactions" table using a DELETE statement. It then optimizes the "transactions" table to reclaim disk space using the OPTIMIZE TABLE statement.

Finally, the script shows the total size of the database using a SELECT statement to confirm that the size has been reduced after archiving.

```
USE interswitch;

-- Create a new table to store archived transactions

CREATE TABLE IF NOT EXISTS archived_transactions (

  transaction_id INT PRIMARY KEY,

  user_id INT,

  amount DECIMAL(10,2),

  transaction_date TIMESTAMP,

  description VARCHAR(255)

);

-- Move transactions older than 36 months to the archive table

INSERT INTO archived_transactions (transaction_id, user_id, amount, transaction_date, description)

SELECT transaction_id, user_id, amount, transaction_date, description

FROM transactions

WHERE transaction_date < DATE_SUB(NOW(), INTERVAL 36 MONTH);

-- Delete archived transactions from the transactions table

DELETE FROM transactions

WHERE transaction_date < DATE_SUB(NOW(), INTERVAL 36 MONTH);

-- Optimize the table to reclaim disk space

OPTIMIZE TABLE transactions;

-- Show the total size of the database after archiving

SELECT

  CONCAT(ROUND(SUM(data_length + index_length) / 1024 / 1024,  2), ' MB')  AS `database_size`

FROM

  information_schema.tables

WHERE

  table_schema = 'interswitch';
```

Implementing Always On Availability Group in our Interswitch DB.:

This script creates an Always On Availability Group called "interswitch_ag" with two synchronous-commit replicas named "Node1" and "Node2". The script adds the "interswitch" database to the availability group and joins the secondary replica to the availability group.

The script then creates a full backup of the primary database and restores it on the secondary replica with the NORECOVERY option to ensure that it can receive transactions.

Finally, the script includes commands to manually fail over to the secondary replica for testing purposes, and then manually fail back to the primary replica after testing is complete.

Note that you will need to customize this script to reflect the specifics of your environment, such as server names and file paths.

```
-- Create the availability group

CREATE AVAILABILITY GROUP interswitch_ag

WITH

  (DB_FAILOVER = ON, CLUSTER_TYPE = NONE)

FOR

  REPLICA ON

    ('Node1' WITH

      (

        ENDPOINT_URL = 'TCP://Node1:5022',

        AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,

        FAILOVER_MODE = AUTOMATIC

      ),

     'Node2' WITH

      (

        ENDPOINT_URL = 'TCP://Node2:5022',

        AVAILABILITY_MODE = SYNCHRONOUS_COMMIT,

        FAILOVER_MODE = AUTOMATIC

      )

    );
```

```
-- Add the interswitch database to the availability group

ALTER AVAILABILITY GROUP interswitch_ag

ADD DATABASE interswitch;


-- Join the secondary replica to the availability group

ALTER AVAILABILITY GROUP interswitch_ag

JOIN WITH (CLUSTER_TYPE = NONE);


-- Create a full backup of the primary database

BACKUP DATABASE interswitch

TO DISK = 'C:\SQLBackup\interswitch.bak';


-- Restore the backup on the secondary replica with NORECOVERY option

RESTORE DATABASE interswitch

FROM DISK = 'C:\SQLBackup\interswitch.bak'

WITH NORECOVERY;


-- Manually fail over to the secondary replica for testing purposes

ALTER AVAILABILITY GROUP interswitch_ag

FAILOVER;


-- Manually fail back to the primary replica after testing

ALTER AVAILABILITY GROUP interswitch_ag

FAILOVER;
```

Database Mirroring:

This script sets up database mirroring for the "interswitch" database by creating endpoints for the principal and mirror servers and backing up the principal database. The script then restores the database on the mirror server with the NORECOVERY option, which prepares it to receive transactions from the principal server.

Next, the script sets up the mirroring session between the principal and mirror servers using the ALTER DATABASE statement. The script also sets the safety option to OFF to ensure that the mirroring session can fail over quickly.

Finally, the script includes commands to manually fail over to the mirror database for testing purposes and manually fail back to the principal database after testing is complete. Note that the FORCE_SERVICE_ALLOW_DATA_LOSS option is used to allow data loss in case the mirroring session cannot be restored to its original state. This option should only be used in extreme circumstances.

You will need to customize this script to reflect the specifics of your environment, such as server names and file paths. Additionally, you may want to implement automatic failover or other configurations to ensure high availability and disaster recovery.

```
-- Create the endpoints for the principal and mirror servers

CREATE ENDPOINT Mirroring

STATE = STARTED

AS TCP (LISTENER_PORT = 5022)

FOR DATABASE_MIRRORING

(AUTHENTICATION = WINDOWS NEGOTIATE, ENCRYPTION = REQUIRED ALGORITHM AES);


-- Backup the principal database and restore it on the mirror server with NORECOVERY option

BACKUP DATABASE interswitch TO DISK = 'C:\SQLBackup\interswitch.bak';

RESTORE DATABASE interswitch WITH NORECOVERY

FROM DISK = 'C:\SQLBackup\interswitch.bak'

WITH

  MOVE 'interswitch' TO 'C:\SQLData\interswitch.mdf',

  MOVE 'interswitch_log' TO 'C:\SQLLog\interswitch_log.ldf';
```

```
-- Set up the mirroring session between the principal and mirror servers

ALTER DATABASE interswitch SET PARTNER = 'TCP://MirrorServer:5022';

ALTER DATABASE interswitch SET SAFETY OFF;

ALTER DATABASE interswitch SET PARTNER SAFETY OFF;


-- Manually fail over to the mirror database for testing purposes

ALTER DATABASE interswitch SET PARTNER FAILOVER;


-- Manually fail back to the principal database after testing

ALTER DATABASE interswitch SET PARTNER FORCE_SERVICE_ALLOW_DATA_LOSS;`
```

Table Partitioning Implementation:

This script partitions the transaction table by date range using a partition function that maps each year to a specific partition. The script also creates a partition scheme that stores each partition in a separate filegroup, and a partitioned table with a clustered primary key on the ID and TransactionDate columns.

```
-- Create the partition function
CREATE PARTITION FUNCTION pf_TransactionDateRange (DATE)
AS RANGE LEFT FOR VALUES
('2019-01-01', '2020-01-01', '2021-01-01');

-- Create the partition scheme
CREATE PARTITION SCHEME ps_TransactionDateRange
AS PARTITION pf_TransactionDateRange
TO
(
   [PRIMARY],
   [fg_Transaction_2019],
   [fg_Transaction_2020],
   [fg_Transaction_2021]
);

-- Create the partitioned table
CREATE TABLE [dbo].[Transaction_Partitioned](
   [ID] [int] NOT NULL,
   [TransactionDate] [date] NOT NULL,
   [Amount] [decimal](18, 2) NOT NULL,
   CONSTRAINT [PK_Transaction_Partitioned] PRIMARY KEY CLUSTERED
   (
      [ID] ASC,
        [TransactionDate] ASC
    ) ON ps_TransactionDateRange(TransactionDate)
) ON ps_TransactionDateRange(TransactionDate);

-- Switch the data to the partitioned table
ALTER TABLE [dbo].[Transaction] SWITCH TO [dbo].[Transaction_Partitioned] PARTITION 1;
```