Databases and information management

Lecture 13

Piotr Syga

2023-01-12

Limiting the communication overhead

Replication — a guide for MySQL Replication

- Master/Slave architecture – Slaves are mainly for read-only operations
- Master creates a binary CommitLog file from the created operations
- every Slave can load the database and replicate the Master's steps from the CommitLog

Limiting the communication overhead

- Master forks its process – a single thread for each Slave
- each Slave works in two threads – first receives the commit or change logs, the second for SQL queries performed by the user
- requires specific configuration and issue solving when we modify the database when connected to a Slave node (we can solve it by using multimaster/multi-source architecture, cf. manual for multi-source replication)
- data may be inconsistent (lazy synchronization) or we have significant communication overhead (eager synchronization)
- some problems include vertical (distance to the Slaves) and horizontal (the number of replicas) scalability

Limiting the communication overhead

Note: The password of the user used for replication is stored as **a plaintext** in the log files of the Master, hence it should be protected (file encryption, using a separate user, obfusating techniques . . .).
We can use:

```
mysql> CREATE USER 'repl'@'%.mydomain.com'
IDENTIFIED BY 'slavepass';
mysql> GRANT REPLICATION SLAVE ON *.* TO
'repl'@'%.mydomain.com';
```

Limiting the communication overhead

Duplicate nodes

- a particular type of a 'Master/Master replica'
- trade-off: consistency vs communication payload (deadlocks are possible in such communication)
- scalability is even worse than in the traditional replication
- no dedicated driver support

Basic approaches to Limiting Communication

There are several ways to limit the communication overhead:

- **Caching:** Storing frequently accessed data locally.
- **Batching:** Combining multiple requests into a single message.
- **Compression:** Compressing data before sending it.
- **Asynchronous communication:** Using asynchronous communication (such as message queues) can allow nodes to continue processing while waiting for a response, rather than blocking until a response is received.

Caching

Caching involves storing frequently accessed data locally, so that it can be accessed more quickly without having to communicate with other nodes. This can reduce the number of communication round trips between nodes, and can improve the performance of the distributed database. To implement caching in a distributed SQL database, each node can maintain a cache of frequently accessed data. When a node receives a request for data, it can first check its cache to see if the data is already stored locally. If the data is in the cache, it can be returned immediately, without having to communicate with other nodes. If the data is not in the cache, the node can fetch the data from the appropriate node and store it in the cache for future requests. One can use Bloom Filters (with deletion) or Cuckoo filters for quick validation of the cache.

Batching

Batching involves combining multiple requests into a single message, so that the overhead of sending multiple messages can be reduced. This can be particularly effective when the overhead of sending a message (such as the overhead of establishing a connection) is high compared to the size of the message itself.

To implement batching in a distributed SQL database, the client can send multiple requests in a single message, and the server can process all of the requests in a single pass. This can reduce the overhead of sending multiple messages, particularly when the client and server are located on different machines.

Asynchronous Communication

Asynchronous communication involves using a message queue or other mechanism to allow nodes to communicate without blocking. This can allow nodes to continue processing while waiting for a response, rather than blocking until a response is received. This can improve the performance of the distributed database, particularly when there are long delays between nodes.

To implement asynchronous communication in a distributed SQL database, nodes can use a message queue or other mechanism to send messages to each other. The sender can add a message to the queue and continue processing, while the receiver can retrieve the message from the queue and process it at a later time. This can reduce the overhead of waiting for a response, and can improve the overall performance of the distributed database.

Date's Rules

1. **Local autonomy** – all the operations performed within given node should be controlled by a local server, in particular security checks and access level control should be performed locally.
2. **Independence of central authority** – we do not require any 'trusted, central authority' that coordinates the work of the nodes, the capabilities and privileges of all the nodes should be similar to each other, in particular all the nodes should be able to establish locks on the database, start and finish the transactions.
3. **Distributed query processing** – each query that references distributed resources should be split into distributed queries, there should be no batch information transfer to a single node and processing the whole information locally.

Date's Rules

4. **Continuous operationability** – malfunction of a subset of the nodes (up to a given threshold) should not affect the accessibility not work of the whole database system, only the data stored solely on the corrupted nodes might not be available.
5. **Location transparency** – the end user (a guest client nor a front end programmer) should not need to know where the data is physically stored nor should they notice the difference in the accessibility or delay in the data stored in the remote nodes.
6. **Fragmentation transparency** – the end user should not be aware of the parts into which the database is divided, he does not need to know the addresses of individual parts or the comunication ports used for nodes communication.

Date's Rules

7. **Transaction transparency** – the end user should not be aware of the split of a transaction into a set of local transactions and the requirement of the end-state consensus (cf. 2PC), all the local transactions should be performed as a single 'atomic operation'.
8. **Replication with unrestricted writing** – the end user should not be aware of the number of replicas and the number of nodes that are involved into node-mirroring, the performance of the nodes should not be lower during the maintenance.
9. **Hardware independence** – the same script (set of queries) should be transferable on different nodes, in particular with non-homogenous servers in the same network.

Date's Rules

10. **Operating system independence** – within distributed databases, information should be seamlessly exchanged between different operating systems.
11. **Network independence** – the database system should not be sensitive to changes in the implementation of network protocols and should work in the same manner with different configurations.
12. ***Database management system and dialect independence*** – the database should be possible to distribute between various DBMS and use the same API, DBMS using various SQL dialects should cooperate without significant time overhead.

Database distribution types

- Horizontal distribution – each row is stored in one (or many in case of replicas) of the nodes, the rows are always complete
- Vertical distribution – we store complete columns in the selected nodes, the number of the columns stored are determined by a lossless decomposition of the relation (cf. the normalization algorithm)
- *Mixed* – the distribution allows us to separate columns by decomposition and store selected rows of such partial table $(\pi_{\mathrm{att}}(\sigma_{\mathrm{cond}}(R)))$

How to distribute the data

- Disjointness – the fragments of the stored rows do not overlap between the nodes (except replicas); we separate primary and non-primary attributes
- Completeness – during the data segmentation we do not lose any data (rows) nor inter-data dependencies (functional dependencies)
- Consistency – a single database (all the tables) should be divided in the same manner (vertical, horizontal, vertically–horizontal, horizontally–vertical)
- Ease of reconstruction – the reconstruction of the segmented data (unions, joins) should be quick and unambiguous, the segmentation should take into the accounts both conditional queries and aggregates

Sample horizontal distribution algorithms

- Round robin – balanced, quick distribution, reading requires checking all the nodes
- Ranged-based – selection of the rows based on their value (in the selected column), possibly (strongly) inbalanced, quick searches
- Compromise – Hash based selection: slower distribution, balanced, quick searches

Distributed queries

- Partitioning the query with respect to the target database nodes
- Local optimization and deadlock detection
- Gathering up the partial results (and error codes), synchronization
- Composition of the partial results

Useful commands: `EXPLAIN PLAN`, AUTOTRACE, SQLTRACE

Plan explanation

- We can use EXPLAIN for simple queries:
```
EXPLAIN SELECT T1.k1+T2.k2 FROM Tab1 T1 JOIN Tab2 T2
ON T1.id=T2.o_id
```
- Similarly we can get info on statements utilizing the non-local tables:
```
EXPLAIN PLAN SET statement_id = 1 FOR
SELECT T1.k1+T2.k2 FROM Tab1 T1 JOIN Tab2@DB2 T2 ON
T1.id=T2.o_id
```
- Reading the plan:
```
SELECT lpad (' ', 2*(level-1))||operation, options,
object_name, optimizer,object_node,other
FROM plan_table START WITH id=0 and statement_id = 1
CONNECT BY prior id=parent_id AND statement_id = 1
```

Examples – EXPLAIN

```
EXPLAIN SELECT * FROM genre;
EXPLAIN SELECT SUM(s.milliseconds) AS len,s.albumid FROM
 track s GROUP BY s.albumid HAVING len>100000;
EXPLAIN SELECT * FROM track t JOIN album a ON
 t.AlbumId=a.AlbumId WHERE a.AlbumId >7 AND title LIKE '%an%';

EXPLAIN WITH RECURSIVE cte AS
(
  SELECT 1 AS n, CAST('buffolo ' AS CHAR(300)) AS str
  UNION ALL
  SELECT n + 1, CONCAT(str, str) FROM cte WHERE n < 9
)
SELECT * FROM cte;
```

Examples – EXPLAIN PLAN

```
EXPLAIN PLAN FOR
SELECT * FROM t
WHERE c1 = 2
AND c2 <> 6
OR c3 BETWEEN 10 AND 20;

SELECT STATEMENT
   TABLE ACCESS T BY INDEX ROWID
      BITMAP CONVERSION TO ROWID
         BITMAP OR
            BITMAP MINUS
               BITMAP MINUS
                  BITMAP INDEX C1_IND SINGLE VALUE
                  BITMAP INDEX C2_IND SINGLE VALUE
               BITMAP INDEX C2_IND SINGLE VALUE
            BITMAP MERGE
               BITMAP INDEX C3_IND RANGE SCAN
```

Transactions

- during a single transaction we can use data from many nodes
- distributed transactions should be transparent for the user
- the distributed transaction is divided into local transaction based on the data allocation
- each local transaction should be treated as an atomic operation in a (larger) distributed transaction
- confirmation – 2PC

Two-phase commit (2PC)

How a simple transaction works?

- We start the transaction.
- We perform operations one-by-one without committing.
- If any of the operations fail we rollback to the state at the start of the transaction and abandon the query.
- We commit the operations.

What is the keypoint? Knowing that some operation failed.

Distributed transactions

- We treat the local transactions as a 'simple' operations, yet we don't want to do them one by one, since the data is disjoint, we want to do them in parallel.
- Every local transaction may include different number of operations, hence they last various time.
- If a single local transaction fails, every local transaction should be rolled back, even those that managed to end already.
- How do we know that a node failed? What do we do when a node fails after their local transaction is done?

Two-phase commit (2PC)

Phase 1: Voting

1. The node to which the user is connected is chosen as a Coordinator and initiates the transaction.
2. The Coordinator sends a (local) query to each node and awaits the answers
3. Each participant performs the local transaction – if it succeeds, it sends `ack` message to the Coordinator, if the transaction fails it rolls back and sends `abort` message, if the node disconnects no message is sent

Two-phase commit (2PC)

Phase 2: Finalization

1. If any of the users sends `abort`, the Coordinator sends `ROLLBACK` message to all the nodes, every node that got the `ROLLBACK` message, rolls back their local transaction and sends `ack` to the coordinator. The Coordinator rolls back the distributed transaction only after receiving all the roll back acknowledgments.
2. We introduce a time limit after which the Coordinator decides that the node failed and if no messages are received in this time period, we proceed as with an `abort`. – *How to treat the timed out nodes?*
3. If every node acknowledged the execution of their transactions, the Coordinator sends `COMMIT` request to the nodes. All nodes should confirm it by sending `ack` to the Coorinator. After getting all the acknowledgments, the distributed transaction is committed. If some `acks` do not arrive, the transaction is rolled back just as in case of the `abort`.

Deadlock Recovery in 2PC

In a deadlock two or more transactions are waiting for each other to complete, resulting in a standstill. This occurs when two transactions have conflicting locks on the same data, and both await the other to release the lock, yet the release condition (eg., `ack` after commit or rollback) cannot be met.

To recover from a deadlock in a 2PC protocol, the coordinator node can use a deadlock detection and resolution algorithm to identify and break it. This may involve rolling back one of the transactions to release the lock, or it may involve using a priority system to determine which transaction should be allowed to proceed.

The automatic recovery is challenging as it requires coordinating the actions of multiple nodes and ensuring that the data remains consistent and accurate.

In-doubt transactions

The transactions that have not been neither rolled back nor commited by the Coordinator are called *in-doubt* transactions.
To break the deadlock we may try several approaches:

- automatic (RECO) – transaction is delayed (cf. `sleep` command in OS), we try to repeat the operations after the node that did not respond with a message restablishes connection
- manual recovery:
  - we have to check which of the local transactions blocks the protocol (`transID`)
  - user with privileges to the transaction state perspective `DBA_2PC_PENDING`, and evocation grph `DBA_2PC_NEIGHBORS` runs: `SELECT * FROM SYS.DBA_2PC_PENDING WHERE local_tran_id = transID`
  - we have to check the `STATE` in the perspective (collecting, prepared, committed, *forced commit*, *forced abort*) and manually commit or abort the transaction:
    `COMMIT (ROLLBACK) FORCE transID`

RECO Process for Resolving In-Doubt Transactions

RECO (Recovery Coordinator) process is used to resolve in-doubt transactions. An in-doubt transaction is a transaction that is in an uncertain state due to a failure or other issue, and it can occur when a coordinator node initiates the commit phase of 2PC cannot reach all of the participant nodes to complete the commit.
The RECO process is responsible for determining the status of in-doubt transactions and taking the appropriate action to resolve them. This may involve committing the transaction, rolling it back, or taking some other action depending on the circumstances.

RECO Process for Resolving In-Doubt Transactions

To resolve in-doubt transactions, the RECO process may use various techniques, such as:

- Checking the status of the participant nodes and any error messages that may have been returned
- Using a priority system to determine which transactions should be given priority
- Using logs and other records to determine the status of in-doubt transactions and to ensure the integrity and consistency of the data

RECO process checks the status of the participant nodes and any errors that may have been returned. If the participant nodes are unavailable or errors have been returned, the transaction is rolled back. If the participant nodes are available and no errors have been returned, the RECO process commits the transaction.