分布式 ID 的应用和考量

屈春河

创建日期：2020-06-07

1 基础知识

在关系型数据库中采用 INT 类型（32 位整数）或者 BIGINT 类型（64 位整数）作为表的主键是一种常见的做法，这种主键也常常被命名为 ID（***_ID）。最为简单的一种 ID 生成方式是自增长，即在定义表结构时采用关键字 AUTO_INCREMENT[MySQL5.7] 和 NOT NULL 修饰 ID，那么在插入一行数据时，如果 ID 为空，则数据库会自动采用 MAX(ID)+1[1]作为该行数据的主键。上述 ID 生成方式虽然简单，但是无法适用于如下的分布式环境：

- 分布式数据库集群，即多个数据库实例以 Shared-Nothing 方式组成数据库集群。这使得采用 AUTO_INCREMENT 方式生成的 ID 会在不同的数据库中产生重复。
- 分布方式插入数据，即存在多个应用相互独立地插入数据。为此，需要一种方案或者机制以确保在不同应用插入数据时每行数据的 ID 是唯一的。

针对于上述分布式环境的 ID 生成方案也被称为分布式 ID 生成方案，其实现非常多，归纳起来可以划分如下两大类：

- 集中协调方案，即通过集中式的服务来协调各个应用生成 ID 或者分配 ID，以实现 ID 的唯一性，例如使用 Spider 存储引擎，可以设置 spider_auto_increment_mode[Spider_Variables] 为 1，能够兼容 AUTO_INCREMEN 方式实现自增长 ID。
- 规则划分方案，即通过特定的规则对于可用 ID 进行划分，使得每个应用以排他方式占用其中一个划分。最为知名一个例子是 Twitter Snowflake 方案，其通过中间 10 个 bit 的工作机器 ID 划分 ID，使得 ID 并不会重复。

在本质上集中协调方案是一种针对分布式环境的集中式 ID 生成方案，其实现相对简单，但是性能往往不如基于规则划分的分布式方案。

虽然分布式 ID 生成方案很多，但是在实际应用时需要从如下几个方面评估和选择：

1) 有效性。满足 ID 的唯一性，不能存在重复的 ID。
2) 高效性。高效性主要涵盖如下几个指标：a）生成效率，即 ID 生成速度，尤其对于集中协调方案制而言，要避免生成 ID 成为插入数据的性能瓶颈；b）资源效率，ID 是 32 位或 64 位整数，要充分利用这些整数资源，避免大范围弃之不用而造成 ID 不够用（ID 溢出）的问题；c）使用效率，要结合存储引擎和查询模式，优化数据存储效率以及应用查询效率。
3) 简单性。在满足有效性和高效性的前提下，实现要尽可能的简单，以降低开发和集成的复杂性。

[1]在 MySQL 中根据系统变量 innodb_autoinc_lock_mode 配置的不同，生成的 ID 可能并不连续。

针对于上述的评估指标，在下文中将会举两个实际的例子，来说明如何根据业务需求和实际情况，选择和设计分布式 ID 方案。

2 例 1—日志型数据入库

日志型数据指的是那些一旦生成就不会被更改的数据，比如用户访问日志等。这些数据生成之后，会被实时地发送到 Kafak 集群。根据实际部署情况，Kafka 集群可能是一个或者多个，而 topic 也可能是一个或者多个。需要指出的是如果是一个 Kafka 集群并且是一个 topic，那么需要将 topic 配置为多个 Partition，而 Kafka Client 则需要采用相同的 group.id，从而实现多个 Kafka Client 以协同方式同时从一个 topic 获取消息。

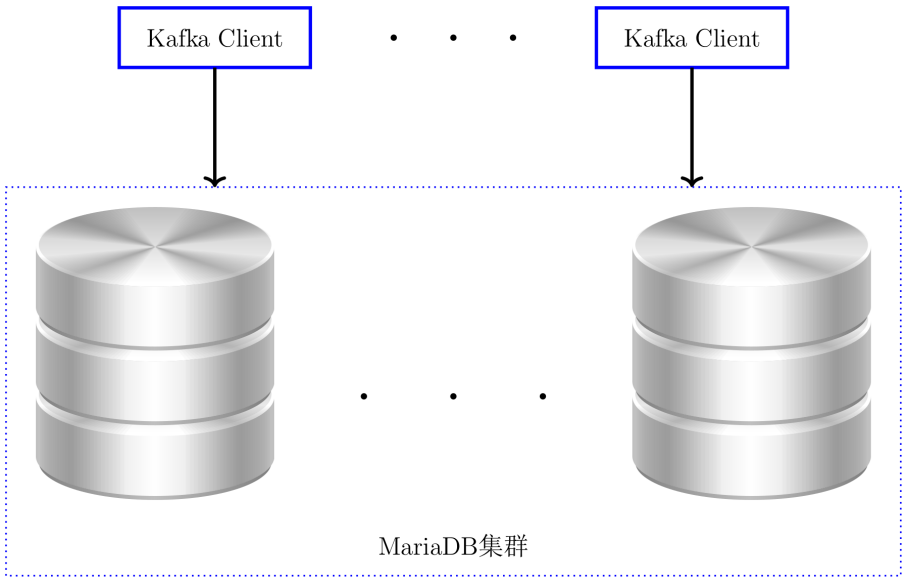

图 1: 日志型数据入库示意图

如图 1 所示，多个 Kafka Client 以分布方式入库数据，整个入库过程包含如下几个功能步骤：

1) 数据获取，从 Kafka 集群读取消息。
2) 数据整理。根据业务需求，对于读取的数据进行规范化。
3) 数据插入。将规范化后的数据插入一个或者多个日志表中。如果插入多个表，往往采用相同的 ID 以相互关联。
4) 数据汇总。插入/更新不同维度或者不同粒度的汇总表，用以支持相关的统计分析功能。

分布式 ID 主要针对于数据插入，即在插入原始日志表时需要生成唯一的 ID 作为表的主键。

针对此种需求，最为简单的方案是采用 Spider 引擎 [Spider_Overview] 和自增长 ID。作为一种集中式 ID 协调机制，此种方案的实现和使用都非常简单。然而，当数据规模非常庞大时，此种方案的查询效率非常低。一种常见的补充方案是定期地将数据转移到历史表中，例如以年为周期转移到历史表 ****_yyyy 或者以月为周期转移到历史表 ****_yyyy_mm。通过历史表，虽然能够在一定程度上改善查询效率，却增加了查询的复杂性，需要在查询语句中显示地指定查询哪个历史表，甚至于如果数据分布在多个历史表中，则不仅需要查询多个历史表，而且还要对查询结果进行 UNION 操作。为了优化数据查询，下文会介绍一种基于 Twitter Snowflake 的改良

方案，并结合分区 [Partition]，从而既可以获得远超自增长 ID 方案的查询性能，又无需在查询中显示地指定历史表。

如图-2 所示，64 位 ID 被划分为三个部分：第一部分，前 32 位为 Unix 时间戳，其为从格林威治时间 1970 年 1 月 1 日 00 点 00 分 00 秒到当前的总秒数；第二部分，中间 n 位代表服务 ID，可以根据需要调整 n 的大小；第三部分，后 32-n 位为自增长整数，当 32-n 位整数用尽时会自动归零并且从零开始增加。显然，为不同的应用分配不同的服务 ID，可以确保 ID 不会相同。

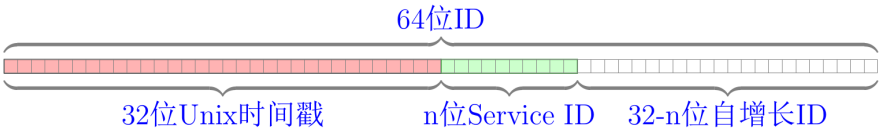

图 2: 图-2 ID 组成示意图

要根据实际情况，合理选择 n 的大小。如果 n 太大，一旦每秒产生的数据量超过 2^{32-n}，就会造成 ID 重复。如果 n 太小，会限制 Kafka Client 的数量，当入库操作比较耗时并且数据量比较庞大时，会导致数据积压，数据无法及时入库。具体而言，n 的取值跟如下几个因素有关：

- 应用的数量 a，即在图-1 中 Kafka Client 的数量，显然，$a <= 2^n$。
- 数据的峰值 m，即每秒最多产生多少个消息，也就是说，每秒需要多少不同的 ID。
- 数据的均值 q，即每秒平均产生多少个消息。
- 处理的耗时 t，即平均每个数据的入库时间，可以根据测试获得此值。
- 应用并发度 k，即每个应用使用多少线程并发处理数据入库操作。Kafka Client 在获取数据后采用线程池，以一个线程处理一个消息的方式完成步骤 2) 到 4) 的功能，其并发度 k 约等于线程池的最大可用线程数。

上述因素中，m 和 q 根据当前实际情况和未来业务规划进行估算，而 a 和 k 的取值则依赖于配置或者部署情况。受限于每秒生成的 ID 数量，每个应用每秒最多处理 2^{32-n} 个数据。由于应用数量可能少于 Kafka Partition 数量或者消息在各个 Kafka Partition 之间可能不平衡，因此需要满足 $m/a << 2^{32-n}$，以留出充足的余量。此外，每个应用每秒最多处理 k/t 个数据，需要满足 $a * k/t > q$，即 $2^n * k/t > q$，以确保数据能够及时入库。根据上述两个关系，可以大概估计出 n 的取值范围为 $\log_2(q * t/k) < n << 32 - \log_2(m/a)$。

合理使用数据库分区 (Partition)[Partition]，能够大大地减小查询时间。为了充分发挥分区的性能优势，需要满足如下两个条件：

- 查询条件中包含分区条件的约束，即根据查询条件就能确定数据所在分区。
- 查询条件所确定的分区数量不多，即所查询数据分布在不多的几个分区内。

针对于日志型数据的应用往往查询特定时间范围内的数据，而在图-2 中 ID 的前 32 位（ID»32）代表 Unix 时间戳。这意味着根据 ID 范围划分分区并根据 ID 范围进行查询，可以优化数据查询效率。依据数据规模，可以以年或月或周划分分区。如下 SQL 实例中，以自然月为周期定义分区，例如 6448965550394572800 对应于时间“2017-08-01 00:00:00”，而 6460469190800179200 对应于时间“2017-09-01 00:00:00”。

```
CREATE TABLE **** (
  id BIGINT UNSIGNED NOT NULL,
```

```
  ...
  PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=utf8
  PARTITION BY RANGE (id) (
  PARTITION pmin VALUES LESS THAN (6437461909988966400),
  PARTITION p201707 VALUES LESS THAN (6448965550394572800),
  PARTITION p201708 VALUES LESS THAN (6460469190800179200),
  ...
  PARTITION pmax VALUES LESS THAN MAXVALUE );
```

针对于上述的分区，在 WHERE 查询条件中必需添加 ID 范围约束，如下 SQL 实例所示。因为 InnoDB 对于主键采用聚簇索引，根据主键范围能够非常快速地读取所需数据。因此，通过本方案能够减小不必要的数据扫描，快速地定位到所需数据，从而大大减小了查询所需时间。

```
  SELECT ..
  FROM ...
  WHERE (id>=(UNIX_TIMESTAMP('2020-08-05 08:00:00')<<32)) AND
        (id< (UNIX_TIMESTAMP('2020-08-06 08:00:00')<<32)) AND
        ...
```

上述方案的一个潜在问题是 32 位 Unix 时间戳的溢出。如果系统需要持续运行数十年的时间，那么 ID 的前 32 位将会在格林威治时间 2038 年 01 月 19 日 03 时 14 分 07 秒溢出，即无法用 32 位无符号整数表示 Unix 时间戳。为了防止这种情况的发生，可以采用相对 Unix 时间戳，即 ID 的前 32 位保存从近期一个特定时间开始到当前时间的总秒数，例如从格林威治时间 2020 年 1 月 1 日 00 点 00 分 00 秒到当前的总秒数，其可以方便地通过当前时间的 UNIX 时间戳减去 UNIX_TIMESTAMP('2020-01-01 00:00:00') 得到。

3 例 2—跨数据中心数据同步

跨数据中心数据同步是针对同时满足如下约束的业务场景。

- 多个数据中心同时写入数据，即位于不同数据中心的应用都需要向相同的表写入数据。
- 不更改数据或者所更改的数据集互不相交，也就是说，位于不同数据中心的应用即使更改同一个表，但是所更改的数据不同。
- 每个数据中心的应用都需要读取全部数据。

针对上述业务场景，下文将会介绍一种基于 AUTO_INCREMENT 自增长 ID 和 Kafka 消息队列的方案。需要说明的是，采用例 1 所示的 ID 生成方式也是一种可选方案，其中不同的服务 ID 对应于不同的数据中心，但是例 1 中的方案将 ID 生成推给应用，不仅增加了应用的复杂性，而且有些情况下还难以实现，比如对于 PHP 应用，难于协同多个 PHP 进程（请求）生成 ID。此外，如果多个数据中心需要更改相同的数据，在一些情况下也能够通过补充方案进行支持，但是无法支持分布式事务。

图-3所示为在两个数据中心之间同步数据的功能示意图。对于多个数据中心的情况，类似于图-3需要每个数据中心对应一个 topic。对于一个数据中心而言，其一方面将本地数据中心的数据发送到对应的 topic，另一个方面从其他 topic 获取消息并插入到本地数据库。

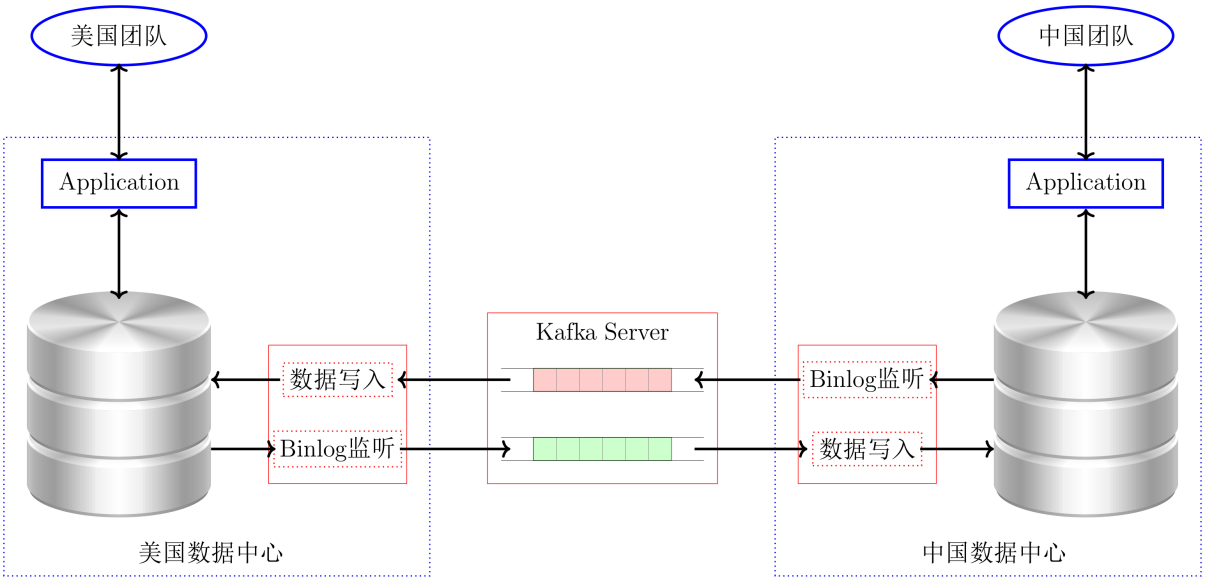

图 3: 跨数据中心数据同步示意图

对于 AUTO_INCREMENT 修饰的自增长主键，MySQL 提供了两个系统变量用于支持源和源之间 (source-to-source) 的复制：auto_increment_increment 和 auto_increment_offse[auto_increment]。上述两个系统变量分别定义了自增长主键的初始值和增加步长。如果在 N 个数据中心之间同步数据，那么配置 auto_increment_offset=N 并且针对不同数据中心分别配置 auto_increment_increment 为 1, 2, ..., N。通过上述配置，N 个数据库实例中的自增长主键将不会重复。此外，还需要将 MySQL 系统变量 binlog_format 配置为 row。

在图-3中，Binlog 监听功能采用 Binlog Connector[Binlog_Connector] 连接 MySQL Server，其在本质上充当了 MySQL Server 的 Slave，能够实时地从 MySQL Server 获取 Binlog 的插入 (INSERT) 和更新 (UPDATE) 日志，并进一步将日志解析和转化为消息，然后将消息插入到特定的 Kafka topic。数据写入功能实时地获取 Kafka 消息，然后将消息转化为对应的 SQL 语句并依次逐个逐个地执行，从而将数据写入到数据库中。

为了实现数据同步的正确性，还需要解决如下两个问题

- 确保操作的时序性。在执行 SQL 写入数据时，写入操作要按照 Binlog 中的顺序依次执行。例如，在 Binlog 中如果操作 op_1 位于操作 op_2 的前面，那么在写入数据库时要确保操作 op_2 开始执行的时间一定不能早于操作 op_1 执行完毕的时间，即只有一个写入操作执行完毕后，才能开始执行后续写入操作。
- 避免操作循环同步。根据写入操作的来源，可以将写入操作划分为两类，一类是来自本地应用的本地操作，另一类是来自于 Kafka 消息的异地操作。MySQL Server 无法区分上述两类操作，因此这两类都会被写入 Binlog。如果 Binlog 监听功能不加区分，这些异地操就会再次被同步到其他数据库中心，造成写入操作消息在数据中心之间来回往返的传递，甚至形成操作消息风暴。

从两个方面来解决操作时序性问题。第一，确保消息在 Kafka Server 中的存储顺序（offset 顺序）与对应操作在 Binlog 中的顺序相同。为此，采用单线程执行 Binlog 监听功能，并且增加如下配置 [Producer_Configst]，以保证 Kafka Producer 依照顺序发送消息。第二，确保依照消息

的存储顺序执行对应的 SQL 语句。为此，topic 的 Partition 要设置为 1，并且数据写入功能采用单线程，即使用一个线程顺序执行如下操作：读取消息，依照消息顺序逐个逐个地解析消息并执行写入操作。

```
acks=all
max.in.flight.requests.per.connection=1
```

对操作循环同步问题，则采用基于 Guava Cache 的过滤功能，过滤掉异地操作。在执行 SQL 语句插入数据之前，需要将操作缓存到 Cache 中。如果为 INSERT 操作，则 Cache key 为依照字典排序的主键，例如主键分别为 k1,k2,...,kn，对应的 key 则为 k1=v1&k2=v2&...kn=vn，而对应的 Cache value 则为 AtomicInteger(1)。如果为 UPDATE 操作，则 Cache key 可以分为两个部分，前一部分是依照字典排序的主键，后一部分是依照字典排序的更新列，例如主键分别为 k1,k2,...,kn，更新的列分别为 c1,c2,...,cm，对应的 key 则为 k1=v1&k2=v2&...&kn=vn&c1=w1&c2=w2&...cm=wm，而对应的 Cahce value 取值，还需要判断 Cache 中是否已经存在此 key：如果 key 不存在，则 value 直接设置为 AtomicInteger(1)；否则将 Cache 中已经缓存的 value 加 1。Binlog 监听功能在获得写入操作（MySQL 写入事件）后，需要根据上述规则获得对应的 Cache key，并且判断 key 在 Cache 中是否存在：如果不存在，则对应的操作为本地操作，需要发送到 Kafka 消息队列；如果存在，则将对应的 value 减 1，然后判断 value 是否为 0，如果为 0，则将此 key/value 对从 Cache 中删除。

```
public void initialize() {
    Cache<String, AtomicInteger>
            factory = CacheBuilder.newBuilder()
                                  .softValues()
                                  .expireAfterWrite(
                                      expireSecondsAfterWrite,TimeUnit.
                                      SECONDS)
                                  .build();
    cache = factory.asMap();
}


private String toKey(MysqlWriteRow event) {
    List<Column> primaryKeys = event.primaryKeys();
    Column[] sortedPrimaryKeys = primaryKeys.toArray(new Column[primaryKeys
        .size()]);
    Arrays.sort(sortedPrimaryKeys, COLUMN_COMPARATOR);
    StringBuilder builder = new StringBuilder(128);
    builder.append(event.table())
           .append(":")
           .append(event.type())
           .append(":");
    for(Column c : sortedPrimaryKeys) {
        builder.append(c.name())
               .append("=")
               .append(c.value())
               .append("&");
```

```
        }
        if (MysqlEvent.Type.UPDATE.toString().equals(event.type())) {
            List<Column> row = event.row();
            Column[] sortedColumns = row.toArray(new Column[row.size()]);
            Arrays.sort(sortedColumns, COLUMN_COMPARATOR);
            for(Column c : sortedColumns) {
                builder.append(c.name())
                        .append("=")
                        .append(c.value())
                        .append("&");
            }
        }
        return builder.substring(0, builder.length()-1);
    }

    private final ColumnComparator COLUMN_COMPARATOR = new ColumnComparator();

    private ConcurrentMap<String, AtomicInteger> cache;
    private long expireSecondsAfterWrite = 600;
```

如果各个数据中心所更改的数据集有重合，可以通过补充方案支持一些特殊的情况。补充方案 1：将有重合的更改操作集中到一个数据中心，并以服务的形式向外提供更改功能；其他数据中心的应用或者通过基于 Web Service 的同步调用或者通过基于消息队列的异步调用来请求此服务。补充方案 1 的问题是时延非常大，如果应用需要及时地获得更新数据，以执行后续操作，那么补充方案 1 就无法满足。我们的业务场景更加特殊，仅仅有一个表（为了方便起见，表名称为 resource）的数据需要同时更改，即需要更新这个表的对应列，关联和去关联其他表的，实现类似于资源分配和回收的功能。为此，我们设计了补充方案 2，其采用预先分配方式，支持上述功能：由 manager 角色从可用的资源中提前分配一定数量的资源给业务人员，即在表 resource 中从 member_id 列为 0 的行中选择一定数量的行并将 member_id 列设置为给定的 member_id；在各个数据中心中应用根据业务人员的 member_id 从表 resource 中选择已经分配给此业务人员的资源，关联或去关联特定表，从而实现了操作数据集的不重合。

上述两个例子仅仅是抛砖引玉，常言道，没有最好的方案，只有最适合的方案。因此，要根据实际情况和业务需求设计方案。

4 附录—64 位 ID 生成代码

```
package qch.concurrent;

import java.math.BigInteger;
import java.util.concurrent.ThreadLocalRandom;
import java.util.concurrent.atomic.AtomicInteger;

/**
```

```
 * Created by Qu Chunhe on 2020-06-15.
 */
public class Id {

    public Id() {
        this(22);
    }

    public Id(int bits) {
        BITS = (bits>31) || (bits<1) ? 22 : bits;
        ID_BOUND = (1 << BITS) - 1;
        SERVICE_ID_MASK = (1 << (32 - BITS)) - 1;
        autoIncrementId = new AtomicInteger(ThreadLocalRandom.current().nextInt
            (ID_BOUND));
    }

    public BigInteger next(long unixTime, int serviceId) {
        BigInteger highPart = new BigInteger(Long.toUnsignedString(unixTime));
        highPart = highPart.shiftLeft(32);
        long lowPart = ((serviceId & SERVICE_ID_MASK) << BITS) | next();

        return highPart.or(new BigInteger(Long.toUnsignedString(lowPart)));
    }

    private int next() {
        int currentId;
        do {
            currentId = autoIncrementId.getAndIncrement();
            if (currentId > ID_BOUND) {
                synchronized (lock) {
                    if (autoIncrementId.get() > ID_BOUND) {
                        autoIncrementId.set(0);
                    }
                }
            }
        } while (currentId > ID_BOUND);

        return currentId;
    }

    private final int BITS;
    private final int ID_BOUND;
    private final long SERVICE_ID_MASK;

    private final AtomicInteger autoIncrementId;
    private final Object lock = new Object();
}
```

参考文献

AUTO_INCREMENT. Auto increment variables[EB/OL]. https://dev.mysql.com/doc/refman/5.7/en/replication-options-master.html#sysvar_auto_increment_increment.

BINLOG_CONNECTOR. Mysql binlog connector java[EB/OL]. https://github.com/shyiko/mysql-binlog-connector-java.

MYSQL5.7. Auto_increment handling in innodb[EB/OL]. https://dev.mysql.com/doc/refman/5.7/en/innodb-auto-increment-handling.html.

PARTITION. Partitioning overview[EB/OL]. https://mariadb.com/kb/en/partitioning-overview/.

PRODUCER_CONFIGST. Kafka producer configs[EB/OL]. http://kafka.apache.org/documentation/#producerconfigs.

SPIDER_OVERVIEW. Spider storage engine overview[EB/OL]. https://mariadb.com/kb/en/spider-storage-engine-overview/.

SPIDER_VARIABLES. Spider server system variables[EB/OL]. https://mariadb.com/kb/ko/spider-server-system-variables/.