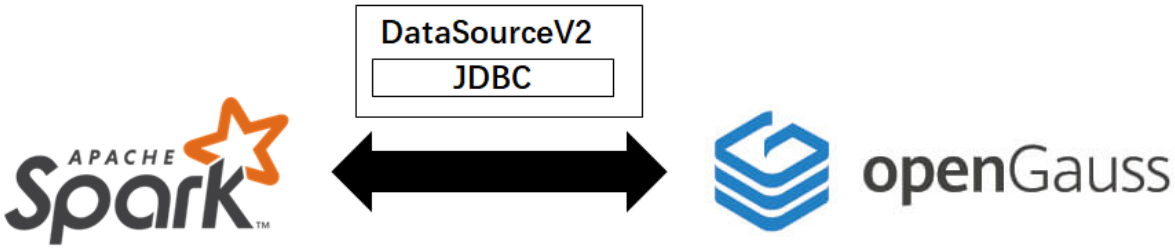
基于openGauss JDBC实现datasourceV2接口

大家好，我是一名参加openGauss社区Summer 2021 活动的学生，耳朵。我参与的题目是“openGauss支持Spark对接”。我把在参与活动中收获的部分操作经验写成了本文。在本文介绍的是通过spark datasource接口通过JDBC读取openGauss，让spark可以操作openGauss中的数据。



1. Spark DataSource接口简介

Spark 官网上对自己的评价是：*Lightning-fast unified analytics engine*，即闪电般迅速的统一分析引擎。Spark 是一个可以进行大规模数据处理的统一分析引擎。接触过大数据领域的同学应该了解 Spark 在业界的地位。

1.1 DataSource 接口的演进

让我们从 Data Source V1开始说起，这个数据源 API 从 Spark 1.3 的版本开始引入。通过这个 API 我们可以很方便的读取各种来源的数据，而且 Spark 可以使用优化器对数据源的读取进行优化，比如采用执行列裁剪、过滤下推等技术。Data Source V1存在着一些不足或设计上不合理的地方，比如**缺乏对列式存储读取的支持**、**不支持流处理** 和设计上理应算是底层的数据源API却依赖了上层API等问题。从 Spark 2.3 开始引入了Data Source V2 API 。

目前Spark 3.0是 Apache Spark 主要维护的版本。Spark 有改动 API 的习惯，以达到最新的标准。在这些 API 中有时也会有比较明显的变化。DataSource V2 API 就发生了较大改动，Spark 很少在两个版本之间频繁地对 API 进行变动。但是由于数据源对于Spark的重要性，意味着数据源相关的 API 将不断得到改进。同样在 spark 2.4中，这些 API 被标记了 **evolving**。这意味着它们注定会在未来被改变。

Data source 的使用并没有发生改变，也就是说，如果你采用的是第三方数据源，那么变动对你没有什么影响。这些改动主要面向数据源接口的开发者。

1.2 DataSource 接口具体介绍

1.2.1 读取数据相关

SupportsRead

此接口指示源支持读取。它有一个需要重写的抽象方法。

```
def newScanBuilder(options: CaseInsensitiveStringMap): ScanBuilder
```

ScanBuilder

用于构建 Scan 的接口。这个接口可以混合使用下推过滤器，以保留所需的信息，将过滤器推送给读者。公开的抽象方法是如下

```
def build(): Scan
```

Scan

数据源扫描的逻辑表示。此接口用于提供逻辑信息，例如实际的读模式是什么。这是一个通用的扫描所有不同的扫描，如批处理，微批处理等。需要重写的方法是

```
def readSchema(): StructType

def toBatch: Batch
```

readSchema -- 源的实际模式。这在 Table 接口中可能看起来像是重复的。之所以重复，是因为在列删除或其他优化之后，模式可能会发生变化，或者我们可能需要对模式进行推理。此方法返回数据的实际模式，其中表1返回初始模式。

toBatch -- 这个方法需要被重写来指示这个扫描配置应该用于批量读取。

Batch

数据源扫描用于批处理查询的物理表示形式。此接口用于提供物理信息，比如扫描数据有多少个分区，以及如何从分区中读取记录。需要重写的方法是

```
def planInputPartitions(): Array[InputPartition]

def createReaderFactory(): PartitionReaderFactory
```

planInputPartitions: 表的分区列表。这个数字决定数据集中分区的数量。

createReaderFactory: 创建Reader的工厂

PartitionReaderFactory

这是一个创建实际数据读取器的工厂类。数据读取器的创建发生在单个机器上。它暴露了一个方法。

```
def createReader(partition: InputPartition): PartitionReader[InternalRow]
```

PartitionReader

最后，我们有了实际读取数据的接口。下面是方法

```
def next : Boolean

def get : T

def close() : Unit
```

正如你从接口上看到的，它看起来像一个简单的基于迭代器的阅读器。

1.2.2 写入数据相关

值得一提的是，Datasource V2 在 API 级别具有支持事务，这也是与上一版本的数据源 API 相比的明显进步。

SupportsWrite

此接口指示源支持写操作。它有一个需要重写的抽象方法。

```
def newWriteBuilder(logicalWriteInfo: LogicalWriteInfo): WriteBuilder
```

上面的方法构建了新的 write builder。

WriteBuilder

是一个为写操作构建配置的接口，我们需要覆盖一个用于批量写操作的接口。

```
def buildForBatch(): BatchWrite
```

BatchWrite

为批量写操作创建工厂的接口。

```
def createBatchWriterFactory(physicalWriteInfo:
PhysicalWriteInfo):DataWriterFactory

def commit(writerCommitMessages: Array[WriterCommitMessage]):Unit

def abort(writerCommitMessages: Array[WriterCommitMessage]): Unit
```

createBatchWriterFactory: 创建数据 writer 工厂的方法

commit: 正如我们前面所讨论的，上面的方法是作者事务支持的一部分。当所有的写操作完成后，调用这个方法来提交。这就是全面提交。单独的分区提交将出现在 DataWriter 接口中，我们将在下面讨论它。WriterCommitMessage 是一个消息接口，数据源应该使用它来定义自己的消息，以指示每个分区写的状态。

abort: 这是一个镜像接口。每当工作完全失败时，就调用这个函数。这用于清除部分写入的数据。

DataWriterFactory

这是一个工厂类，用于创建实际的数据编写器。

```
def createDataWriter(partitionId:Int, taskId:Long): DataWriter[T]
```

partitionId: 分区的 id。这有助于编写器理解它正在写入的分区。taskId: task的id。方法返回 DataWriter。

DataWriter

最后我们有一个实际写入数据的接口

```
def write(record: T): Unit

def commit(): WriterCommitMessage

def abort(): Unit
```

write 方法是负责实际写入的方法。其他两个方法与 BatchWriter 相同，但是现在它们在分区级别上工作。这些方法负责在分区级别提交或处理写入故障。

这个接口中由 commit 方法发送的 WriterCommitMessage 是发送给 BatchWrite 的消息。这有助于数据源理解每个分区的状态。

2. openGauss JDBC Driver

这一部分我们主要讲如何获取 openGauss JDBC Driver与获取之后在IDEA中如何使用。

2.1 获取openGauss JDBC Driver

获取方式可以分为两种，分别是直接获取jar包与从源码进行构建jar包。本节只介绍直接获取的方式，从源码进行构建可以参考社区仓库openGauss-connector-jdbc的README。

2.1.1 直接获取

在使用openGauss JDBC 驱动之前，请确保您的服务器已经可以正常运行 openGauss 数据库（参考openGauss快速入门）。

从maven中央仓库获取

Java开发者可从maven中央仓库中直接获取jar包，坐标如下：

```
<groupId>org.opengauss</groupId>
<artifactId>opengauss-jdbc</artifactId>
```

从社区官网下载安装包

1. 在官网下载安装包。

   点击链接，在openGauss Connectors部分下，根据您部署数据库的服务器的对应系统选择JDBC_version的下载按钮。version的下载按钮。{version}即您需要的版本号。

2. 解压压缩包。

   ```
   tar -zxvf openGauss-${version}-JDBC.tar.gz
   ```

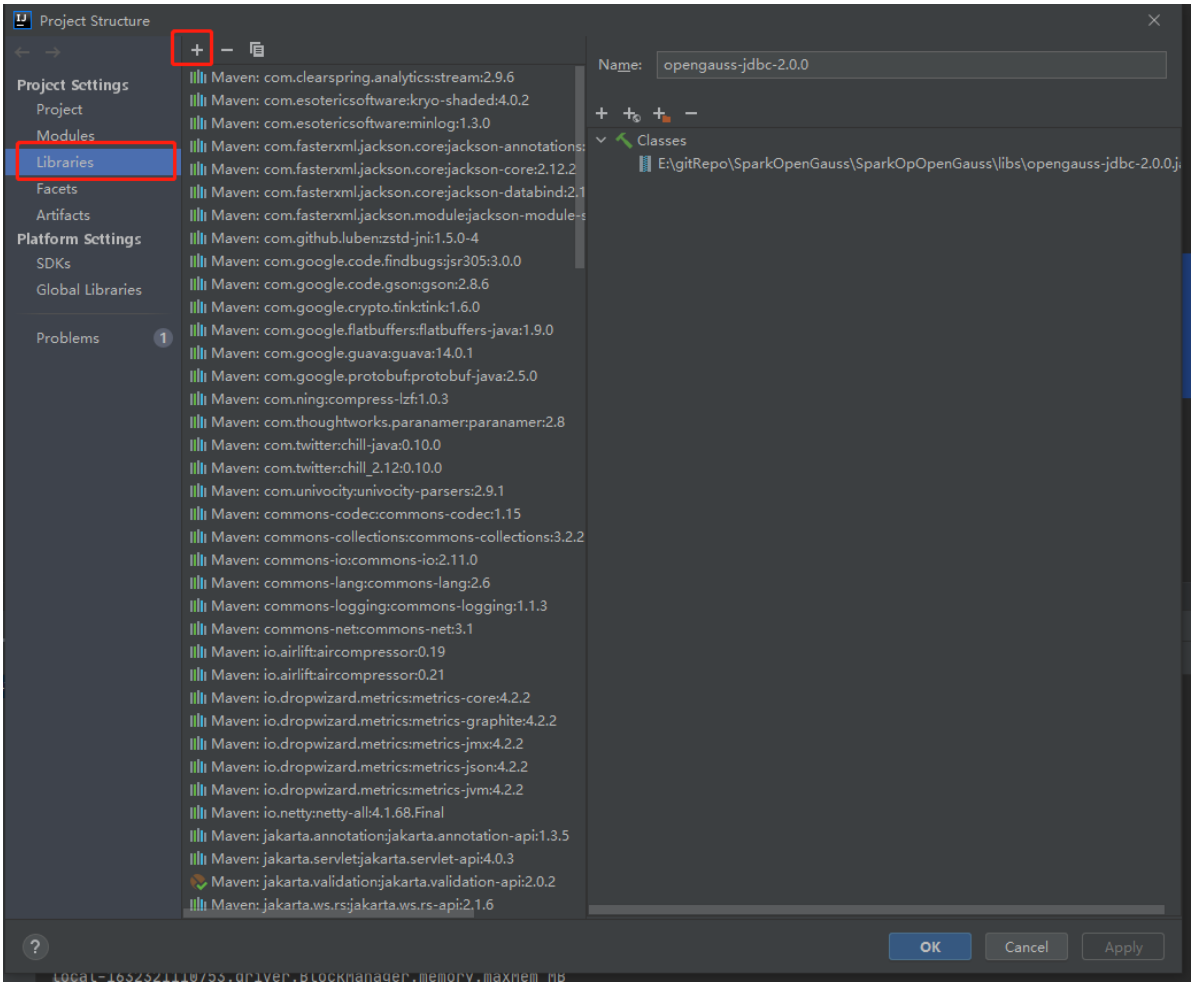
3. 解压后可以看到同级目录下出现了两个jar包，分别是opengauss-jdbc-version.jar和postgresql.jar。opengauss-jdbc-version.jar和postgresql.jar。opengauss-jdbc-{version}.jar是可以与PG-JDBC共存的包, 包名自2.0.1之后的版本全部从org.postgresql变更为org.opengauss, 并且驱动名称从jdbc:postgresql://替换为jdbc:opengauss://。目前从maven中央仓库中获取的也是这个包。

2.2 如何使用openGauss JDBC Driver

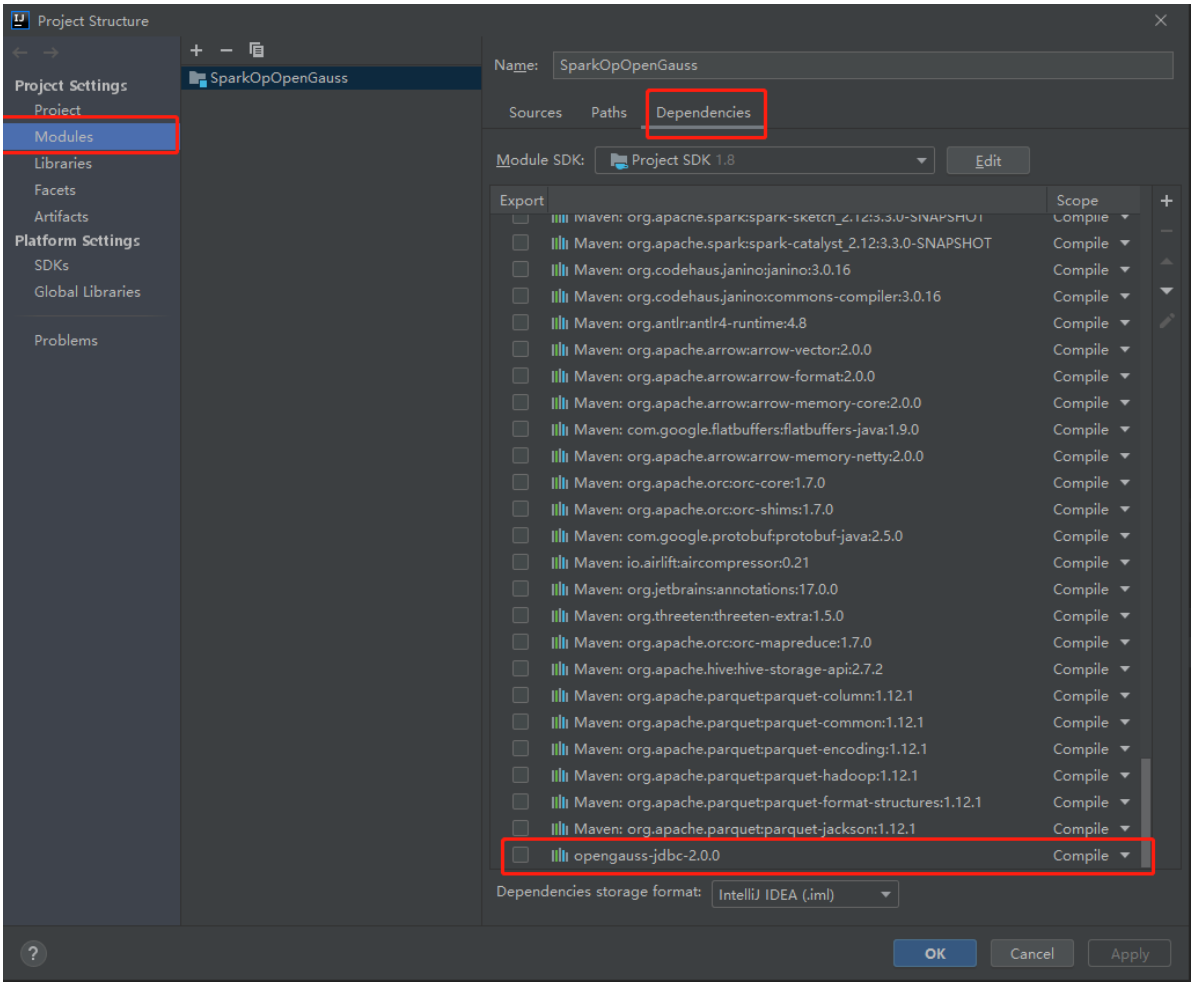
本节只介绍在maven项目中如何使用openGauss JDBC 驱动。使用方式包括两种，即使用下载好的jar包与从maven中央获取jar包使用。

2.2.1 本地包的使用

首先，为方便使用，在项目目录下创建一个lib目录，记为 GaussPro/lib。将已经准备好的jar包放入 GaussPro/lib 目录下。依次选择菜单栏中的"FIle" -> "Project Structure...", 进入如下界面。



接着选择左侧中的 "Libraries", 选择中间栏的 "+"号，选择 "Java"，定位到 GaussPro/lib/opengauss-jdbc-2.0.0.jar，选择 OK 即可正常使用。如果jar包是从官网下载的，JDBC url 请使用 `jdbc:postgresql://x.x.x.x:port/dbname` 。如果jar包是从社区 master 分支自行构建得到的，url 请使用 `jdbc:opengauss://x.x.x.x:port/dbname` 。



添加完毕后，可以在“Project Structure”中选择“Modules”，之后选择“Dependencies”查看是否成功添加了依赖。

2.2.2 使用Maven中央仓库的包

目前 openGauss 的 JDBC 驱动包已经可以在 maven 中央仓库获得，坐标如下。

```
<!-- https://mvnrepository.com/artifact/org.opengauss/opengauss-jdbc -->
<dependency>
    <groupId>org.opengauss</groupId>
    <artifactId>opengauss-jdbc</artifactId>
    <version>2.0.1-compatibility</version>
</dependency>
```

在 IDEA 中可以看到 jar包内的包名已经发生了改变。在代码中的url也应该使用 `jdbc:opengauss://x.x.x.x:port/dbname` 。

3. 基于openGauss JDBC 实现DataSource接口

我们在第 1 部分已经介绍了 Spark Data Source 的关于数据读取与写入接口。下面是针对 openGauss 自带的数据库例子 school.sql文件中生成的表 class 的一些具体实现。

3.1 基础接口的实现

实现 `TableProvider` 接口

TableProvider 接口可以对现有表应用数据操作，比如读取、追加、删除和覆盖。

```
class DefaultSource extends TableProvider {
  override def inferSchema(options: CaseInsensitiveStringMap): StructType =
OpenGaussTable.schema

  override def getTable(
                         schema: StructType,
                         partitioning: Array[Transform],
                         properties: util.Map[String, String]
                       ): Table = new
OpenGaussTable(properties.get("tableName"))
}
```

实现 `SupportsRead`、`SupportsWrite` 接口

```
class OpenGaussTable(val name: String) extends SupportsRead with SupportsWrite {
  override def schema(): StructType = OpenGaussTable.schema

  override def capabilities(): util.Set[TableCapability] = Set(
    TableCapability.BATCH_READ,
    TableCapability.BATCH_WRITE
  ).asJava

  override def newScanBuilder(options: CaseInsensitiveStringMap): ScanBuilder =
new OpenGaussScanBuilder(options)

  override def newWriteBuilder(info: LogicalWriteInfo): WriteBuilder = new
OpenGaussWriteBuilder(info.options)
}
```

具体的表结构

```
object OpenGaussTable {
  /*Table products*/
  /*Database school, table course*/
  val schema: StructType = new StructType().add("cor_id",
IntegerType).add("cor_name", StringType).add("cor_type",
StringType).add("credit", DoubleType)
}
```

数据库连接的属性

```
case class ConnectionProperties(url: String, user: String, password: String,
tableName: String, partitionColumn: String, partitionSize: Int)
```

3.2 读相关接口的实现

```

class OpenGaussScanBuilder(options: CaseInsensitiveStringMap) extends ScanBuilder
{
  override def build(): Scan = new OpenGaussScan(ConnectionProperties(
```

```
    options.get("url"), options.get("user"), options.get("password"),
options.get("tableName"), options.get("partitionColumn"),
options.get("partitionSize").toInt
  ))
}

class OpenGaussPartition extends InputPartition

class OpenGaussScan(connectionProperties: ConnectionProperties) extends Scan with
Batch {
  override def readSchema(): StructType = OpenGaussTable.schema

  override def toBatch: Batch = this

  override def planInputPartitions(): Array[InputPartition] = Array(new
OpenGaussPartition)

  override def createReaderFactory(): PartitionReaderFactory = new
OpenGaussPartitionReaderFactory(connectionProperties)
}

class OpenGaussPartitionReaderFactory(connectionProperties:
ConnectionProperties)
  extends PartitionReaderFactory {
  override def createReader(partition: InputPartition):
PartitionReader[InternalRow] = new
OpenGaussPartitionReader(connectionProperties)
}


class OpenGaussPartitionReader(connectionProperties: ConnectionProperties)
extends PartitionReader[InternalRow] {
  private val connection = DriverManager.getConnection(
    connectionProperties.url, connectionProperties.user,
connectionProperties.password
  )
  private val statement = connection.createStatement()
  private val resultSet = statement.executeQuery(s"select * from
${connectionProperties.tableName}")

  override def next(): Boolean = resultSet.next()

  override def get(): InternalRow = InternalRow(
    resultSet.getInt(1),
    UTF8String.fromString(resultSet.getString(2)),
    UTF8String.fromString(resultSet.getString(3)),
    resultSet.getDouble(4))

  override def close(): Unit = connection.close()

}
```

3.3 写相关接口的实现

```
class OpenGaussWriteBuilder(options: CaseInsensitiveStringMap) extends
WriteBuilder {
```

```
  override def buildForBatch(): BatchWrite = new
OpenGaussBatchWrite(ConnectionProperties(
    options.get("url"), options.get("user"), options.get("password"),
options.get("tableName"), options.get("partitionColumn"),
options.get("partitionSize").toInt
  ))
}

class OpenGaussBatchWrite(connectionProperties: ConnectionProperties) extends
BatchWrite {
  override def createBatchWriterFactory(physicalWriteInfo: PhysicalWriteInfo):
DataWriterFactory =
    new OpenGaussDataWriterFactory(connectionProperties)

  override def commit(writerCommitMessages: Array[WriterCommitMessage]): Unit =
{}

  override def abort(writerCommitMessages: Array[WriterCommitMessage]): Unit =
{}
}

class OpenGaussDataWriterFactory(connectionProperties: ConnectionProperties)
extends DataWriterFactory {
  override def createWriter(partitionId: Int, taskId:Long):
DataWriter[InternalRow] =
    new OpenGaussWriter(connectionProperties)
}

object WriteSucceeded extends WriterCommitMessage

class OpenGaussWriter(connectionProperties: ConnectionProperties) extends
DataWriter[InternalRow] {

  val connection = DriverManager.getConnection(
    connectionProperties.url,
    connectionProperties.user,
    connectionProperties.password
  )

  val statement = "insert into ${connectionProperties.tableName} (cor_name,
cor_type, credit) values (?,?,?)"
  val preparedStatement = connection.prepareStatement(statement)

  override def write(record: InternalRow): Unit = {
    val cor_name = record.getString(0)
    val cor_type = record.getString(1)
    val credit = record.getDouble(2)

    preparedStatement.setString(0, cor_name)
    preparedStatement.setString(1, cor_type)
    preparedStatement.setDouble(2, credit)
    preparedStatement.executeUpdate()
  }

  override def commit(): WriterCommitMessage = WriteSucceeded

  override def abort(): Unit = {}
```

```
  override def close(): Unit = connection.close()
}
```

3.4 自定义 Data Source 接口的使用

与正常Data Source 接口的使用的唯一区别就是区别输入源的 format：

```
.format("org.opengauss.spark.sources.opengauss") #这是我们实现接口的文件所在的包名
```

直接使用 JDBC 接口则是（这样使用则不是V2）：

```
.format("jdbc")
```

4. 暑期 2021 的一些感想

作为学生，也需要平衡时间来参加暑期2021。在这个过程中，我对 Spark 对接 openGauss 的流程做了验证。在整个流程的验证中发现了一些问题，也在社区提交了 Issue 和 PR 进行解决，例如：

- openGauss-connector-jdbc jar构建与使用指引(中文版已合入，英文版正在合入)：https://gitee.com/opengauss/openGauss-connector-jdbc/pulls/48
  通过完成openGauss-connector-jdbc的文档，指引用户获取、编译、使用openGauss JDBC。
- Spark将openGauss作为数据源的完整示例（正在合入）：https://gitee.com/opengauss/examples/pulls/18
  提到的所有demo、测试及更详细的代码实现都在这个PR中，手把手教你如何通过spark datasource 连接openGauss，该PR还在review中，欢迎大家在PR中留言提出建议。

其实我在暑假期间同时参加了 openGauss 社区的任务打榜赛，即根据个人的issue和 PR 数量排名。在参加这两个项目的过程中，社区的工作人员都是非常和善的，我碰到的问题也能及时的得到回应和解决，提交的PR也会及时被Review。在参加打榜赛的时候，我发现了一些规则上不完善的问题，反馈之后工作人员也及时更新了规则。

现在openGauss-server仓库是非常活跃的，反映的问题也能及时得到解决。在开发的同时，openGauss 的功能也在逐渐完善，比如之前 openGauss 是不能直接使用 LOAD 命令加载共享库，需要修改源码重新编译，到现在官方博客已经放出了一些如何移植插件、增加算子的教程。可以看到社区新的仓库在逐渐增加，同时周边仓库的活跃度也在上升。现在期望openGauss能及时跟上PG单个查询中并行执行的特性（个人催更）。

参与暑期2021也让我对开源技术有了更深的理解。Git 推动了开源文化的发展并改善了开发者的工作模式，但是除了技术之外，社区参与者的积极性与负责程度决定了开源社区的氛围。导师与社区中帮助我的人让我认识到开源文化中，人是最重要的组成部分。作为一名个人开发者，热情和积极是很重要的动力，在开源社区中所付出的东西都有额外收获。比如，参与开源可以收获很多东西，包括但不限于：提升专业技能、遇见对你有帮助的大牛、提高语言和组织能力清晰的描述问题与解决方案、积累个人声望提高存在感。在这个过程中，我确实在这几个方面得到了提高。感谢社区给机会：）