分布与并行数据库 Pard 系统实验报告

金国栋　　韩涵　　黄文韬　　陈成

2018 年 1 月 17 日

目录

1 系统概述

1.1 任务描述

本实验主要设计和实现一个面向分析的分布式数据库系统，即通过网络将多个不同的局部数据库系统连接起来，使得用户可以通过分布式数据库管理系统达到透明性管理的目的，采用横向扩展的方式，扩展数据库系统的查询性能。

1.2 系统需求

1. 支持数据库的创建和删除。
2. 支持表的创建和删除。
3. 支持数据表的水平分片和垂直分片。
4. 支持元数据的存储和管理。
5. 支持数据从文件批量导入。
6. 支持插入记录。
7. 支持删除记录。
8. 支持 SQL 语句 *select ... from ... where ...*。

1.3 运行环境

Pard 系统采用 Java 语言开发，运行时需要 Java 8 以上的运行环境，目前仅支持 Linux 系统。Pard 采用 P2P 的设计思想，每个 Pard Server 都可

以部署在集群中的任意节点上，且每个节点都可以作为主节点被客户端访问。在我们的实验环境中，我们利用三台普通台式机搭建了一个集群，分别为 pard01、pard02 和 pard03。其中 pard01 和 pard02 分别部署了一个 Pard Server，pard03 部署了两个 Pard Server。

1.4 开发环境

系统的开发环境主要由 Git、Maven(v3.3.9+) 和 Java 8 构成。开发的 IDE 包括 Eclipse 和 Intellij IDEA。系统利用 Maven 实现了代码风格的管理和代码编译、打包、自动分发的功能，方便了项目的开发过程。

```
mvn clean compile
mvn clean package
```

1.5 使用说明

Pard 系统的使用说明如下：

1. 代码编译和打包：

   切换到 Pard 项目的根目录，执行 `mvn clean package`

   pard-assembly/target 目录下的*.zip* 或*.tar.gz* 文件即为部署文件。

   解压后目录结构如图 1

2. Pard Server 的启动：`./sbin/pard-server start/run` Pard Server 的启动方式分为两种：

   - start: 后台启动。Pard Server 进程将利用 nohup 后台启动。
   - run: 前台启动。

   对于后台启动的 Pard Server，可以调用 `./sbin/pard-server stop` 停止进程。

3. 配置文件：

   - *pard.name*: Pard Server 的名称，需要保证每个 Server 都不相同。

- *pard.host*: Pard Server 的网络 IP 地址。
- *pard.server.port*: Pard Server 的服务端口号。客户端连接该端口。
- *pard.web.port*: Pard Server 的网络端口号。
- *pard.rpc.port*: Pard Server 的 RPC 端口。供 Server 之间调用。
- *pard.exchange.port*: Pard Server 数据传输的端口。
- *pard.file.port*: Pard Server 文件传输的端口。
- *pard.connector.host*: Pard Server 连接的数据库的地址。
- *pard.connector.user*: Pard Server 连接的数据库的用户名。
- *pard.connector.password*: Pard Server 连接的数据库对应用户的密码。
- *pard.connector.driver*: Pard Server 使用的 JDBC 连接的 driver 类。

图 1: 发布版目录结构

1.6 项目管理

Pard 的开发利用 Github 进行协同，并且代码都开源在 Github 上。*https://github.com/dbiir/pard*

2 系统架构

2.1 整体架构

Pard 整体架构如图 2。用户通过命令行连接集群中的任意一个 Pard Node 进行交互，每个 Pard Node 都可以通过内置的 connector 连接多个 SlaveDB。通过 connector 的方式屏蔽底层 SlaveDB 的细节，这样 SlaveDB 可以为 PostgreSQL 或者 MySQL 等任意数据库，只需按照 connector 的接口开发对应的 connector 即可。

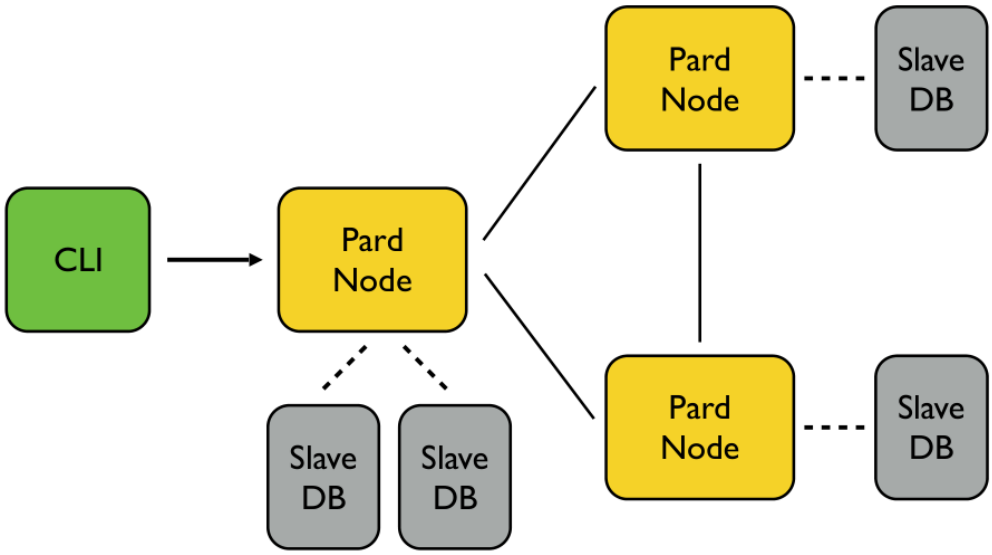

图 2: Pard 整体架构图

Pard 单个节点的架构如图 3。每个节点都有可能成为与用户直接交互的节点，应用层面诸如 Client、Web UI 是用户可见的抽象层级。下面来看底层实现。Pard 收到用户输入的 SQL 语句，转化为执行的 Job，先交给 SQL Parser 进行 SQL 的语法解析，得到抽象语法树 AST。再依据查询本身特性和数据划分的特点，在 SQL Optimizer 模块中进行优化。然后由 Job Planner 制定物理的查询执行计划，并将计划交给 Job scheduler，生成具体的查询执行任务，分发给各个节点去执行，并协调任务之间执行的顺序、同步异步等。

各个节点还有存储管理模块，负责管理数据在内存中的组织形式。数据在内存中以 *Block* 的方式组织。节点中的通讯模块具体分两类：一类是任务的通信，使用 RPC 技术发送较小数据量的任务通知；另一类是 SQL 执行需要的大批量数据的通信，比如节点之间的数据 shuffle，我们使用 Netty 做大批量数据的异步传输。元数据由各节点的 Catalog 模块维护，数据分布在集群中的各个节点，由分布式的 KV 存储系统 etcd 负责存储和数据的同步，etcd 遵循的 raft 协议可以确保各个节点元数据的一致性。节点中的 Executor 是本地执行器，负责执行接收到的具体的查询任务，并调用 Connetor 与本地连接的数据库进行交互。NodeKeeper 模块负责集群中节点状态的维护，每个 Pard Server 启动的时候都需要向 NodeKeeper 注册，并在进程结束的时候通知 NodeKeeper。

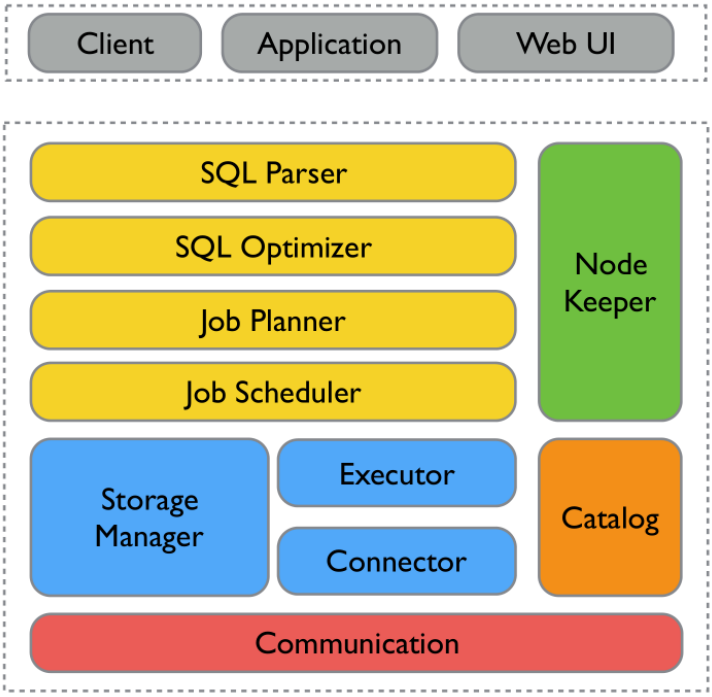

图 3: Pard 单个节点的架构

2.2 PardServer 启动/关闭流程

Pard Server 的启动流程：

1. 初始化配置。读取配置文件，并且检查配置项。
2. 加载 connector。加载配置的 connector，包括初始化数据库 JDBC 连接的连接池。
3. 加载 Catalog。初始化 Catalog 与 etcd 的连接，并启动 etcd 的 watch 线程。
4. 加载本地执行器 (Executor)。初始化本地执行器。
5. 加载 NodeKeeper。初始化本地的 NodeKeeper 模块。
6. 启动数据传输的 ExchangeServer。以线程方式初始化和启动 Netty。不阻塞主进程。
7. 启动文件传输的 FileExchangeServer。以线程方式初始化和启动 Netty。不阻塞主进程。
8. 启动 RPCServer。以线程方式初始化和启动 RPC 服务。不阻塞主进程。
9. 加载 JobScheduler。初始化 JobScheduler，并创建单例。
10. 加载 TaskScheduler。初始化 TaskScheduler，并创建单例。
11. Pard Server 注册。向 NodeKeeper 注册一个 Pard Server，包括名称、IP 地址、RPC 端口、数据传输端口、文件传输端口等。该信息会在 etcd 中进行同步，方便其他节点查询。
12. 启动 Pard Web Server。以线程方式启动内嵌的 Jetty 作为 web server，目前用于展示查询计划。
13. 启动 socket 监听。服务器和客户端之间采用 socket 进行通信，启动服务器端的 socket 监听线程。
14. 注册 shutdownHook。注册 JVM 进程停止的 shutdownHook，该 hook 中添加的方法将会按照顺序在 JVM 停止之前执行，进行清理和资源释放。

Pard Server 的停止流程：

1. 停止 web server。
2. 停止 socket 连接和监听。
3. 通知 NodeKeeper 更改 Pard Server 的状态为下线 (或从 NodeKeeper 中删除该节点)。
4. 停止 Catalog。中止 etcd 的 watch 线程，并释放与 etcd 的连接。
5. 停止本地执行器 (Executor)。停止正在执行的 task。
6. 停止数据传输的 ExchangeServer。关闭 Netty 线程。
7. 停止文件传输的 FileExchangeServer。关闭 Netty 线程。
8. 停止 RPCServer。关闭 RPC 线程。
9. 关闭 Connector。关闭与数据库的连接池。

2.3 时间安排

如图 4。

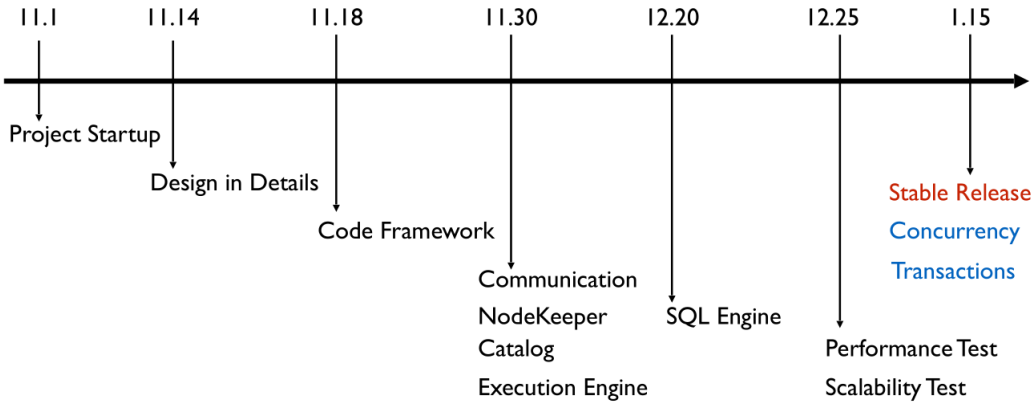

图 4: 时间安排 timeline

3 各模块详细设计

3.1 执行流程

总体执行流程如图 5。

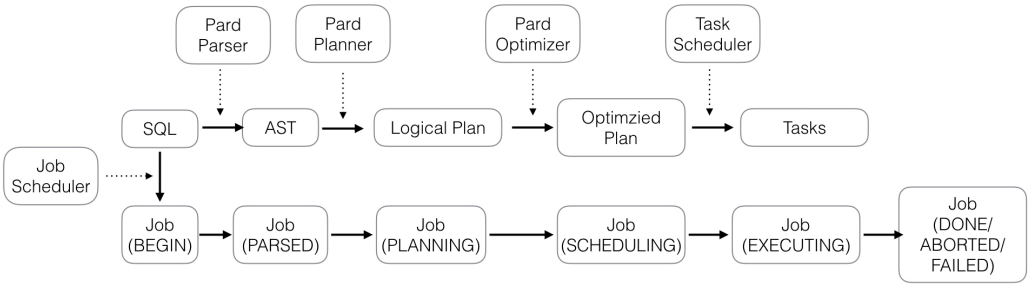

图 5: 查询执行流程

在 PardServer 中, 每个客户端连接由一个单独的线程负责, 通过 socket 方式连接。用户输入的 SQL 语句会提交到 *JobScheduler* 中, 创建一个新的 *Job*。JobSchdueler 会维护和更新该 Job 的状态信息。具体流程如下：

- SQL 语句先由 *Pard Parser* 进行语义解析, 得到一个抽象语法树 (AST), 同时 Job 的状态更新为 *PARSED*。
- 抽象语法树由 *Pard Planner* 制定逻辑查询计划, 同时 Job 的状态更新为 *PLANNING*。
- 生成的逻辑查询计划由 *Pard Optimizer* 负责进行优化, 同时 Job 的状态更新为 *SCHEDULING*。
- 优化后的查询计划交给 *Task Scheduler* 生成执行的任务, 并进行任务

的调度、分发和监控，各节点负责接收任务并执行。同时 Job 的状态更新为 *EXECUTING*。

- 任务执行完毕以后，Job 的最终状态为三种，*DONE*、*ABORTED* 和 *FAILED*。这三种状态分别表示执行顺利完成、执行被放弃和执行失败。

在查询执行过程中，每次任务执行时都需要通过 NodeKeeper 同步当前集群的节点状态，如果有节点状态为非在线，则该查询转为 *ABORTED* 状态，并刷出日志提醒用户。

在 JobScheduler 中记录了该 Server 所有正在执行的 job 的情况，并且已执行完的 Job 根据三种状态，分别维护了一个列表。这些信息可以提供给 web 端进行状态显示。

3.1.1 CREATE/DROP/INSERT 执行流程

CREATE/DROP/INSERT 生成执行任务后，任务将通过 RPC 的方式发送给对应节点，并返回执行结果。

3.1.2 LOAD 执行流程

LOAD 生成执行任务后，共有两类 task。一类在本地读取文件，并按照数据划分的规则将文件内容划分后存储在内存中；另一类将内存中划分后的数据发送到对应节点，并调用节点的 Executor 执行本地数据库的 LOAD 操作。最后收集各节点的执行状态，返回最终状态给用户。

3.1.3 SELECT 执行流程

单表的 *SELECT* 生成包含查询树的执行任务后，分发给对应节点执行。执行任务时，节点的 Executor 调用本地的 Connector。本地 Connector 根据查询树生成对应的 SQL 语句，如适应于 PostgreSQL 语法规则的 SQL 语句。Connector 执行的结果以 Block 的形式流式地返回给 Executor，即 Executor 每次读取一个 Block，而不是一次读取整个结果集，这种方式有效地节省了内存的使用，防止数据量大时内存不够。

错误编号	错误代码	错误描述
-10000	ParseError	Syntax error
-10001	SchemaExsits	Schema already exists
-10002	SchemaNotExists	Schema does not exist
-10003	SchemaNotSpecified	Schema is not specified
-10004	TableExists	Table already exists
-10005	NotaColumnDefinition	Definition is not a column definition
-10006	ColumnDataTypeNotExists	Data type is not specified
-10007	UnknownHorizontalPartition	Specified horizontal partition is unknown
...	...	...

表 1: Pard 错误信息

3.1.4 JOIN 执行流程

Pard 中两表 JOIN 的执行分为两个阶段, 第一个阶段为数据 shuffle, 第二个阶段为本地 join 和主节点 merge 查询结果。shuffle 阶段节点读取本地待 shuffle 的数据表的内容，按照数据划分规则，发送到对应的节点，对应节点收到数据后存储在本地的临时表中（我们采用了 PostgreSQL 的内存临时表机制来优化）。在所有节点完成 shuffle 后，进入第二阶段，节点在本地执行 join 查询（数据表和临时表之间），然后将查询结果发回主节点。主节点收集查询结果并合并后返回给客户端。

3.1.5 错误信息

查询执行过程可能产生各种错误, 对此, 系统设计了一套错误信息。例如表 1。

3.2 节点通信

通讯任务依据数据传输量的大小，和任务本身性质，可以自然的分为两类：一类是描述节点要负责的任务本身的通知，数据量很小，我们选用 RPC 技术来实现；另一类是因数据 partition 产生的大批量数据传输，数据

量通常很大，我们选用 Netty 框架来实现。Insert、Create、Drop 等语句返回值比较少，不需要太多交互通信的也都是用 RPC 来做的，其余大批量通信比较多的涉及到 Netty。Pard 有对 Byte array 做可选压缩的功能，

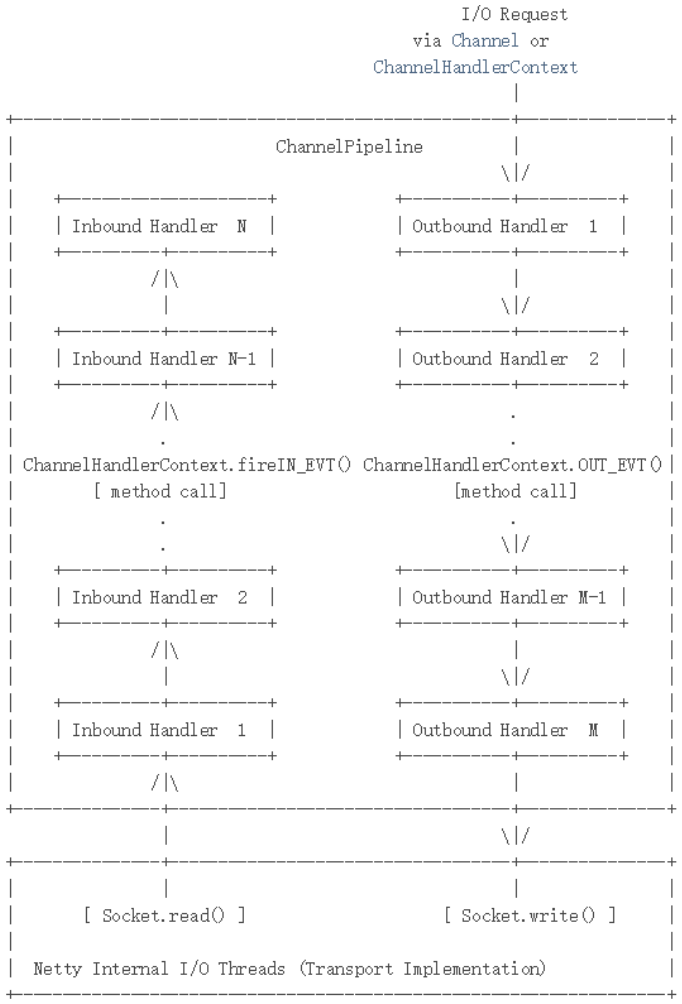

图 6: Pard inbound outbound

Pard 中 Netty 的事件可以分为 Inbound 和 Outbound 事件. 从图 6可以看出, inbound 事件和 outbound 事件的流向是不一样的, inbound 事件的流行是从下至上, 而 outbound 刚好相反, 是从上到下。并且 inbound 的传递方式是通过调用相应的 `ChannelHandlerContext.fireIN_EVT()` 方法, 而 outbound 方法的的传递方式是通过调用 `ChannelHandlerContext.OUT_EVT()`

方法。例如 `ChannelHandlerContext.fireChannelRegistered()` 调用会发送一个 *ChannelRegistered* 的 inbound 给下一个 ChannelHandlerContext, 而 `ChannelHandlerContext.bind` 调用会发送一个 *bind* 的 outbound 事件给下一个 ChannelHandlerContext。`PardFileExchangeClient` 的 `run` 方法的 pileline 就加入了新的 `ExchangeFileSendHandler` (`extends ChannelInboundHandlerAdapter`) 的实例。

网络传输是只能传输 byte array 的, 拿到后要做语义解析。我们开始自己做了一个 Object Encoder 和 Decoder, 还有使用 gzip 算法做数据压缩的可选功能。Netty 自己也实现了解析 byte array 的这类需求, 叫编解码技术。服务器编码数据后发送到客户端, 客户端需要对数据进行解码。编解码器由两部分组成：编码器、解码器。解码器：负责将消息从字节或其他序列形式转成指定的消息对象；编码器：将消息对象转成字节或其他序列形式在网络上传输。我们的编解码其都是 ChannelHandler 的实现。入站“ByteBuf”读取 bytes 后由 ToIntegerDecoder 进行解码, 然后将解码后的消息存入 List 集合中, 然后传递到 ChannelPipeline 中的下一个 ChannelInboundHandler。

3.3 元数据

Pard 使用 etcd, 以 Key-Value 形式存储 GDD。etcd 的官方定义是：

> A highly-available key value store for shared configuration and service discovery.

实际上, etcd 作为一个受到 Zookeeper 与 doozer 启发而催生的项目, 除了拥有与之类似的功能外, 更具有以下 4 个特点:

1. 简单：基于 HTTP+JSON 的 API 让你用 curl 命令就可以轻松使用;
2. 安全：可选 SSL 客户认证机制;
3. 快速：每个实例每秒支持一千次写操作;
4. 可信：使用 Raft 算法充分实现了分布式。

所以我们选用 etcd 存储 GDD。Pard 的一个亮点是基于 etcd 的 watch 机制, 减少了 IO 的操作。只有当 GDD 发生改变时, 才去取数据, 更新。我们

使用 etcd 的持久监听 (stream), 当有事件时, 会连续触发, 不需要客户端重新发起监听。因为 GDD 本来就很小, 所我们在内存中维护 GDD。Pard watch 的有 site、schema、user, 每个站点开启三个 Thread 来 watch。调用 `watcher.listen()` 方法持续的 watch。

元数据模型关键结构如下 :

```
public class GDD {
HashMap<String,Site>siteList;
HashMap<String,Schema>schemaList;
HashMap<String,User>userList;
}

public class Site {
int Id;
String name;
String ip;
int port;
boolean isLeader;
}

public class Schema {
String name;
int id;
List<Table>tableList;
List<Privilege>userList;
//Statics statics;
}
public class Privilege {
int use;//1,read, 3, write,5create 7,delete;
int uid;
String username;
}
```

```
public class Table {
String tablename;
int id;
HashMap<String,Column>columns;
List<Fragment>fragment;
List<Privilege>userList;
int isFragment;
Statics statics;
}

public class Column {
int id;
int dataType;
String columnName;
int len;
int index;//0:none; 1:hashindex; 2:btreeindex; 3:others
boolean isKey;
}
public class Fragment {
int fragmentType;//0: horizontal;1:vertical
List<Condition>condition;
Table subTable
int siteId;
Statics statics;
}
public class Condition {
String columnName;
int CompareType;//define less,great,equal..
String value;
int dataType;//the datatype of value
}
```

```
public class Statics {
String columnName;
String min;
String max;
String mean;
String mode;
String median;
HashMap<String,Integer>staticList;
}
public class User {
int uid;
String username;
HashMap<String,Privilege>tableList;
HashMap<String,Privilege>schemaList;
}
```

3.4 SQL 解析

语义语法解析使用 Antlrv4。ANTLR—Another Tool for Language Recognition，可以根据输入的 SQL 命令和我们自定义的规则，根据自动生成语法树，

Pard 支持的语法如下：

```
statement
: query
#statementDefault
| USE schema=identifier
#use
| CREATE SCHEMA (IF NOT EXISTS)? qualifiedName
#createSchema
| DROP SCHEMA (IF EXISTS)? qualifiedName (CASCADE | RESTRICT)?
#dropSchema
```

```
| ALTER SCHEMA qualifiedName RENAME TO identifier
#renameSchema
| CREATE TABLE (IF NOT EXISTS)? qualifiedName
tableElementPart
(',' tableElementPart)*
partitionOps?
#createTable
| CREATE INDEX indexName=identifier ON
indexTbl=qualifiedName '(' identifier (',' identifier)*')'
#createIndex
| DROP INDEX indexName=identifier
#dropIndex
| DROP TABLE (IF EXISTS)? qualifiedName
#dropTable
| INSERT INTO qualifiedName columnAliases? query
#insertInto
| DELETE FROM qualifiedName (WHERE booleanExpression)?
#delete
| ALTER TABLE from=qualifiedName RENAME TO to=qualifiedName
#renameTable
| ALTER TABLE tableName=qualifiedName
RENAME COLUMN from=identifier TO to=identifier
#renameColumn
| ALTER TABLE tableName=qualifiedName
DROP COLUMN column=qualifiedName
#dropColumn
| ALTER TABLE tableName=qualifiedName
ADD COLUMN column=columnDefinition
#addColumn
| GRANT
(privilege (',' privilege)* | ALL PRIVILEGES)
```

```
ON TABLE? qualifiedName TO grantee=identifier
(WITH GRANT OPTION)?
#grant
| REVOKE
(GRANT OPTION FOR)?
(privilege (',' privilege)* | ALL PRIVILEGES)
ON TABLE? qualifiedName FROM grantee=identifier
#revoke
| SHOW GRANTS
(ON TABLE? qualifiedName)?
#showGrants
| EXPLAIN ANALYZE? VERBOSE?
('(' explainOption (',' explainOption)* ')')? statement
#explain
| SHOW STATS (FOR | ON) qualifiedName
#showStats
| SHOW STATS FOR '(' querySpecification ')'
#showStatsForQuery
| DESCRIBE qualifiedName
#showColumns
| DESC qualifiedName
#showColumns
| START TRANSACTION (transactionMode (',' transactionMode)*)?
#startTransaction
| COMMIT WORK?
#commit
| ROLLBACK WORK?
#rollback
| SHOW PARTITIONS (FROM | IN) qualifiedName
#showPartitions
| SHOW SCHEMAS
```

```
#showSchemas
| SHOW TABLES (FROM schemaName=identifier)?
#showTables
| LOAD path=identifier INTO table=qualifiedName
#load
;
```

目前并没有实现全部的完整 SQL 功能,只支持单表和两表的基本 Select 语句。

3.5 查询计划及优化

3.5.1 语义检查与逻辑计划

当一条 SQL 语句经过 Parser 转为抽象语法树后，我们根据 SQL 语句类型的不同，将语法树转为逻辑计划，同时进行语义检查。逻辑计划分为这样几类。

- 与数据库 Schema、Table 相关的逻辑计划。主要针对的是 DDL，即 Schema 和 Table 的 Create、Drop 和 Show 语句。这类计划的特点是除了向各个站点发送请求外（Show 不需要），还需要修改全局数据字典（必需），且结果返回值数据量较小。
- 记录的增删改计划，主要针对的是各类 DML 语句。这类计划不需要修改全局数据字典（当前没有考虑数据分布），只需要向各个站点分发数据，结果以一个或几个值的形式返回。
- 对记录的查询，主要针对的是 DQL，即 select 语句。这类计划的 SQL 语句复杂，需要将抽象语法树转化为逻辑查询计划，即一棵逻辑查询计划树，并做查询优化和查询本地化。

对于所有输入的抽象语法树，我们都在生成逻辑计划前进行语义检查，对 Create Table 语句，主要从 GDD 中检查表是否存在，各列的类型数据库是否支持，长度定义是否合理；对 Insert 语句，主要检查插入的表是否

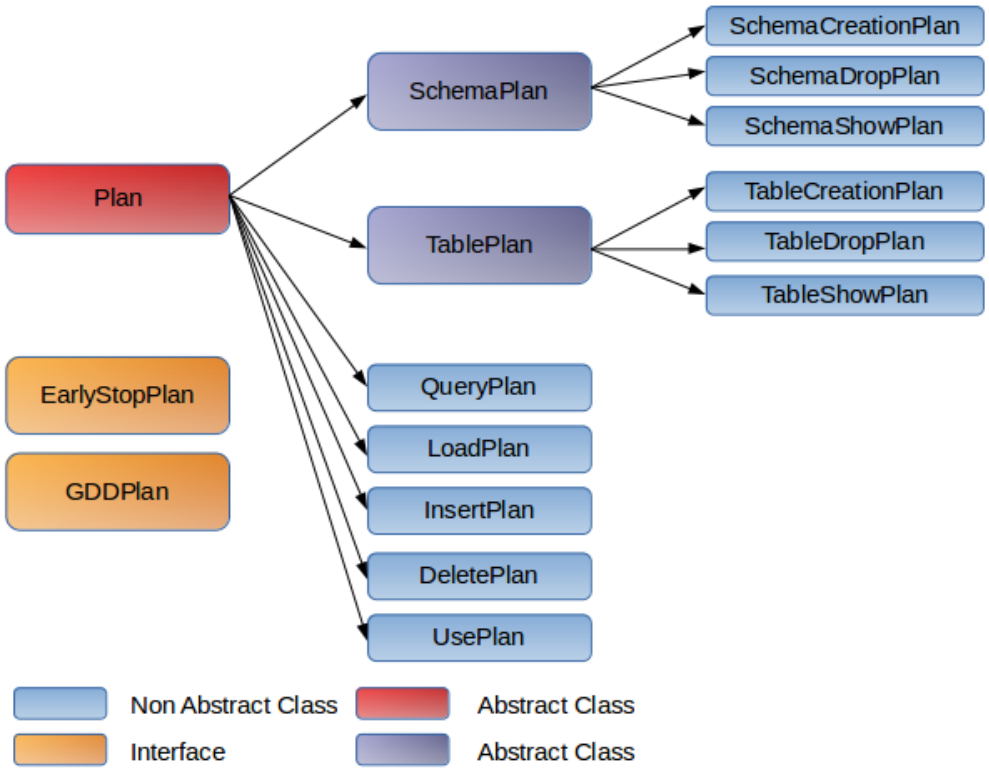

图 7: Plan 类继承层次

存在，Insert table 后面跟的列的数目和 values 中每个 Row 的列的数目是否一致，数据类型是否匹配等。

此外，还有 Use 语句的 Plan，针对一个 Client 在服务器端使用一个线程处理的特点，我们使用 Java 的 ThreadLocal 对象来缓存每个 Client 当前的 Schema，这样毋需建立会话 (Session) 机制就可以获得会话机制带来的好处。

同时，我们还为各类计划设计可选的两个接口供实现，分别是 GDDPlan 和 EarlyStopPlan。GDDPlan 用于需要修改全局数据字典的计划，提供 beforeExecution 和 afterExcution 两个方法，分别在计划执行前和计划执行后对 GDD 进行操作；EarlyStopPlan 为语义检查结束后可以提前终止的计划设计，在 Plan 中通知 QueryHandler 该语句已经执行完成。比如 Create Table if not exsits 时表已经存在，或者生成 Select 语句计划时，发现 where 条件对所有站点都为永假式，或进行优化后发现只剩一个没有孩子的 UnionNode，此时调用 EarlyStopPlan 的 isAlreadyDone 方法，都会返回 True。

3.5.2 逻辑查询计划以及优化

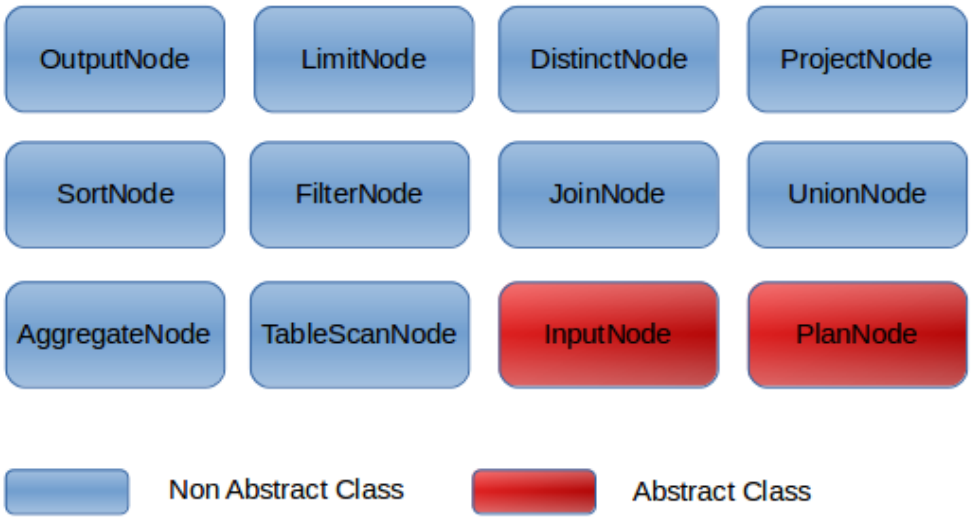

图 8: PlanNode

如 8所示，我们目前提供 10 种逻辑查询计划的符号，对于一个 Select 语句，最终会被转换为一棵由这些符号组成的逻辑查询树，并对各个 TableScanNode 进行本地化，对本地化后的树进行优化。我们的优化主要实现了以下规则：

- 表的本地化，即把 GDD 中的表的概念，根据 GDD 中分片信息，转化为本地数据库中的表。
- 投影和选择下推。这里需要注意的一点是，投影下推时不仅要考虑上层投影的条件，还要考虑上层的 Filter 和 Join 条件，也就是说，投影下推时除了把某个表上的列投影到本地化后的结点上时，需要增加上层 filter 和 join 所包含的列属性。
- 表达式剪枝，Pard 实现了一套表达式优化的组件，可以将多个表达式 and 或 or 起来，并进行逻辑运算。对于求得表达式为 False 的结点，会在逻辑查询后减掉，通过此方法可以消去不必要的 Join 和 Union。
- Join 下推。对两个 UnionNode 进行 Join, 下推 Join 后变为几个 JoinNode 的 Union

3.5.3 表达式计算和剪枝

Pard 实现了一套表达式组件 (如图 9), 可以将多个表达式 and 或 or 起来，并进行逻辑运算。这套表达式组件目前用于逻辑查询计划的优化，插入数据时确定待插入数据在结点的分布，以及 Join 时 Shuffle 数据。

目前这套表达式组件提供以下规则：

- 幂等律，即多个相同的表达式进行 and 和 or 运算，得到表达式本身
- 分配律，例如 a*(b+c)=a*b+a*c
- 吸收律，例如 a*(a+b)=a,a+a*b=a*b (部分实现)
- 交换律，我们的 and/or 支持多个条件同时 and/or，不考虑顺序

基于这套规则，我们实现了以下功能：

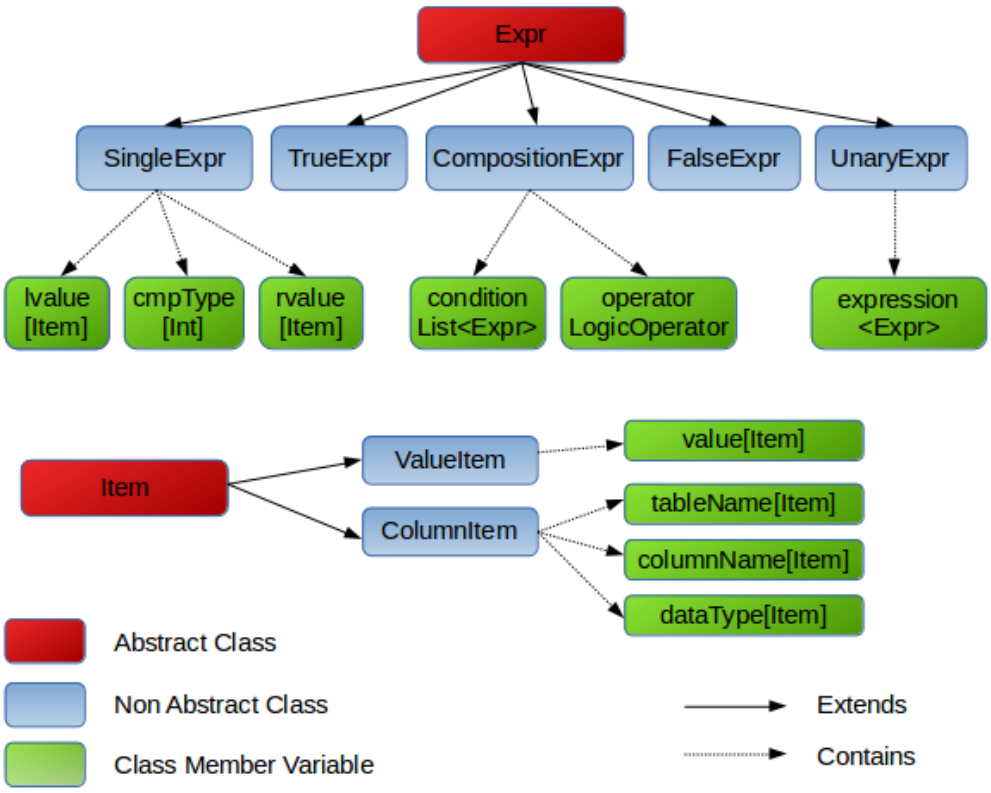

图 9: Expression

- 表达式化简,例如 (a>5) and (a>3), 化简后为 (a>5), 再比如 (a>10)and(a<3), 化简后为 false
- 表达式连接，即可以把多个表达式用 and 或 or 连接起来
- 表达式提取，即从多表的列参与的表达式中，根据表名提取列名，以及根据表名提取该表在表达式中的条件
- 表达式替换，将表达式的列名替换为值，并计算表达式的真假；或者将某表名替换为另一表名，用于 join 的本地化

4 任务分工及小结

4.1 任务分工

组员分工如下：

1. 金国栋：系统设计，SQL 语法解析，节点通信，任务执行框架。
2. 韩涵：查询计划和查询优化。
3. 黄文韬：Executor 和 PostgreSQL Connector 实现、客户端实现。
4. 陈成：实验报告撰写。

4.2 金国栋的小结

这学期分布式数据库的课程中学习到了分布式事务的许多知识。在此之前，我对事务的理论涉及较少，理解得也不深入，经过本学期的学习，对分布式事务的执行有了大致的了解，同时也了解了分布式一致性的协议，收获颇多。

这学期的课程实验是设计和实现面向查询的分布式数据库系统，我之前对面向 OLAP 的分布式系统有过一些了解，但没有仔细思考过分布式查询的优化，范老师的讲解正好补上了我的这点缺憾。因此，在课程实验的设计上，我非常想把积累到的知识和经验应用进来，实现一个实际可用的系统，在系统设计时做得比较复杂，没有完全参照实验要求。另外，在开发过

程中，过于理想地预估了组员的完成度和完成效率，没有合理地安排好时间。这两点导致在课程结束时无法按时完成所有功能。不过在开发的过程中，我们也验证了系统设计的合理性，并对一些设计做了调整，在后续的开发中希望总结经验，凝练想法，实现一个完整的分布式数据库系统。

4.3 韩涵的小结

本学期的《分布式与并行式数据库》，对我而言是非常有意义的课。在课堂上，老师用简单易懂的语言向我们介绍了分布式数据库的基本原理与模型；在课下，在师兄的带领下我们进行了深入的实践。这门课与张孝老师的《数据库系统原理与实现》以及卢卫老师的《分布式系统与云计算》互为补充，帮助我们构建起了更完善的知识体系结构。

在写逻辑计划的生成以及 Join 的任务调度时我常常想起两句话，一句话是“世事洞明皆学问”，一句话是“纸上得来终觉浅”，老师课上讲完某一节后常常问我们听过课后有没有想出代码如何实现，我总是一脸茫然地摇摇头。某个知识点的实现该用什么结构，各个组件之间如何交互、需要其他组件提供什么样的服务，如何实现才能高效……当时是考虑不到的。比如表达式的化简在思维上很简单的东西，实现起来也花了不少时间。语义检查四个字读起来可能花不了 2 秒，但是对 AST 做语义检查却需要对 SQL 语句的每一部分进行细致的处理。我们的大作业，实现了一个还不是太完善的分布式数据库，如果此时再去翻一遍书，或者听一次课，相信想的东西会和之前不一样吧，虽然不敢说听课后就能马上写代码，但是在大脑中关于分布式数据库的理论和实现会变得更近一些。

4.4 黄文韬的小结

在这门课上，我主要负责 Pard 分布式数据库的 Client 客户端以及 Executor 和 Connector 部分的工作，在编码的过程中遇到了许许多多的问题，在自己摸索和大家的帮助下不断克服困难，终于完成了自己的工作。编码之后，我对分布式数据库的整体架构有了更加清楚的认识，也明白了 Connector 和 Executor 在整个数据库的重要地位和作用，深感写好一个系统的不易，尤其是这种分布式系统，随着开发的进行，代码量的增加，许多

不经意间写的代码都隐藏了很多 bug，而分布式系统在调试方面也不如单机的方便。所以，以后要多注重代码的单元测试，养成良好的编码习惯。另外，在编程语言的应用上，我也学习到了很多细节上的优化，这些优化乍看对性能的提升很小，但当这段代码被多次反复调用以后，性能瓶颈就体现了出来。

4.5 陈成的小结

本门课程中，学习了分布式数据库的一些非常基础的知识，相比于另一门数据库管理系统实现，这门课就多在了分布式的部分，数据如何 partition，不同的 partition 方式下查询等操作如何优化。大作业确实非常 challenging，比如 Netty 框架设计复杂精妙，我学习花了很大时间，一个个 example 的改、试，后来在师兄的帮助下才正确的运用在了项目中。前几次课与卢卫老师的大数据管理冲突，我都没来，后来觉得还是应该去一下明确本学期在课程上的投入。课程消耗精力太多的话，没时间做自己该负责的科研项目了。我认为，我们没有在预期时间完成任务的原因是当初系统设计的导向不是任务导向，当初可能要求太高了，各种复杂的结构、情况都考虑的话，过早优化是万恶之源，可能基本的任务都没完成就得不偿失了。通过这次大作业感受到搭建一个能够 work 的系统并不是一件容易的事情，总是会有莫名其妙的 bug 困扰着你并消磨你的热情；另一个体会是，永远不要把工作往后推，因为一到期末 deadline 很多，你并不会像自己预期那样高效，连续的熬夜也不能迅速打造完美的系统。

4.6 致谢

最后，特别感谢范老师对我们的理解和支持。感谢邵明锐和纪昀红同学的参与，他们分别帮助我们实现了元数据管理模块以及 Delete 语句的执行计划生成。