# 概述

TiDB是PingCAP公司自主设计、研发的开源分布式关系型数据库，是一款同时支持在线事务处理与在线分析处理（Hybrid Transactional and Analytical Processing, HTAP）的融合型分布式数据库产品，具备水平扩容或者缩容、金融级高可用、实时HTAP、云原生的分布式数据库、兼容MySQL 5.7协议和MySQL生态等重要特性。目标是为用户提供一站式OLTP（Online Transactional Processing）、OLAP（Online Analytical Processing）、HTAP解决方案。TiDB适合高可用、强一致要求较高、数据规模较大等各种应用场景。

## 核心特性

**一键水平扩容或者缩容**

得益于TiDB存储计算分离的架构的设计，可按需对计算、存储分别进行在线扩容或者缩容，扩容或者缩容过程中对应用运维人员透明。

**金融级高可用**

数据采用多副本存储，数据副本通过Multi-Raft协议同步事务日志，多数派写入成功事务才能提交，确保数据强一致性且少数副本发生故障时不影响数据的可用性。可按需配置副本地理位置、副本数量等策略满足不同容灾级别的要求。

**实时HTAP**

提供行存储引擎TiKV、列存储引擎TiFlash两款存储引擎，TiFlash通过Multi-Raft Learner协议实时从TiKV复制数据，确保行存储引擎TiKV和列存储引擎TiFlash之间的数据强一致。TiKV、TiFlash可按需部署在不同的机器，解决HTAP资源隔离的问题。

**云原生的分布式数据库**

专为云而设计的分布式数据库，通过TiDB Operator可在公有云、私有云、混合云中实现部署工具化、自动化。

**兼容 MySQL 5.7协议和MySQL生态**

兼容MySQL 5.7协议、MySQL常用的功能、MySQL生态，应用无需或者修改少量代码即可从MySQL迁移到TiDB。提供丰富的数据迁移工具帮助应用便捷完成数据迁移。

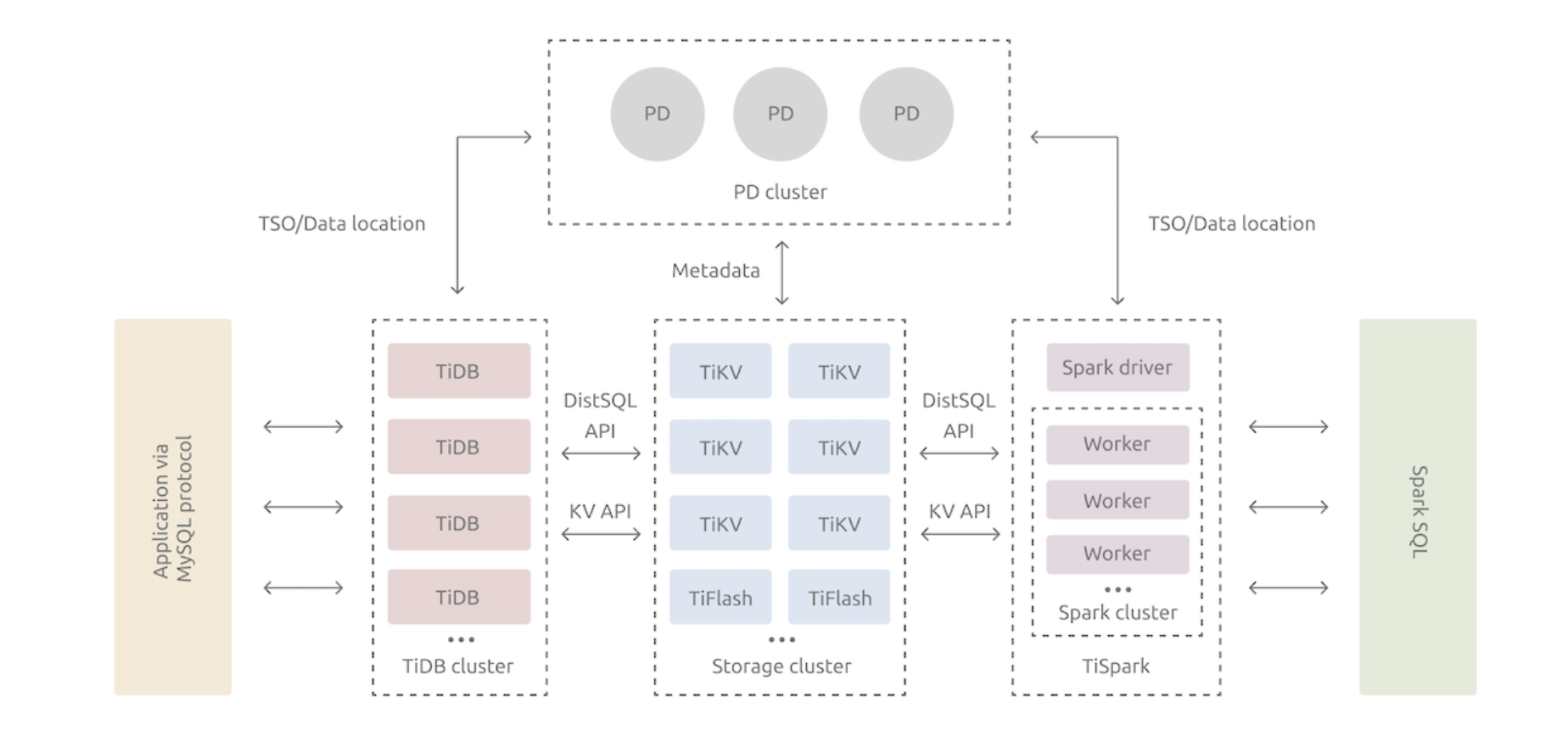
## 基本功能

## TiDB 6.1 Release Notes

## 与MySQL的兼容性

## 使用限制

# 架构



## 计算/SQL引擎

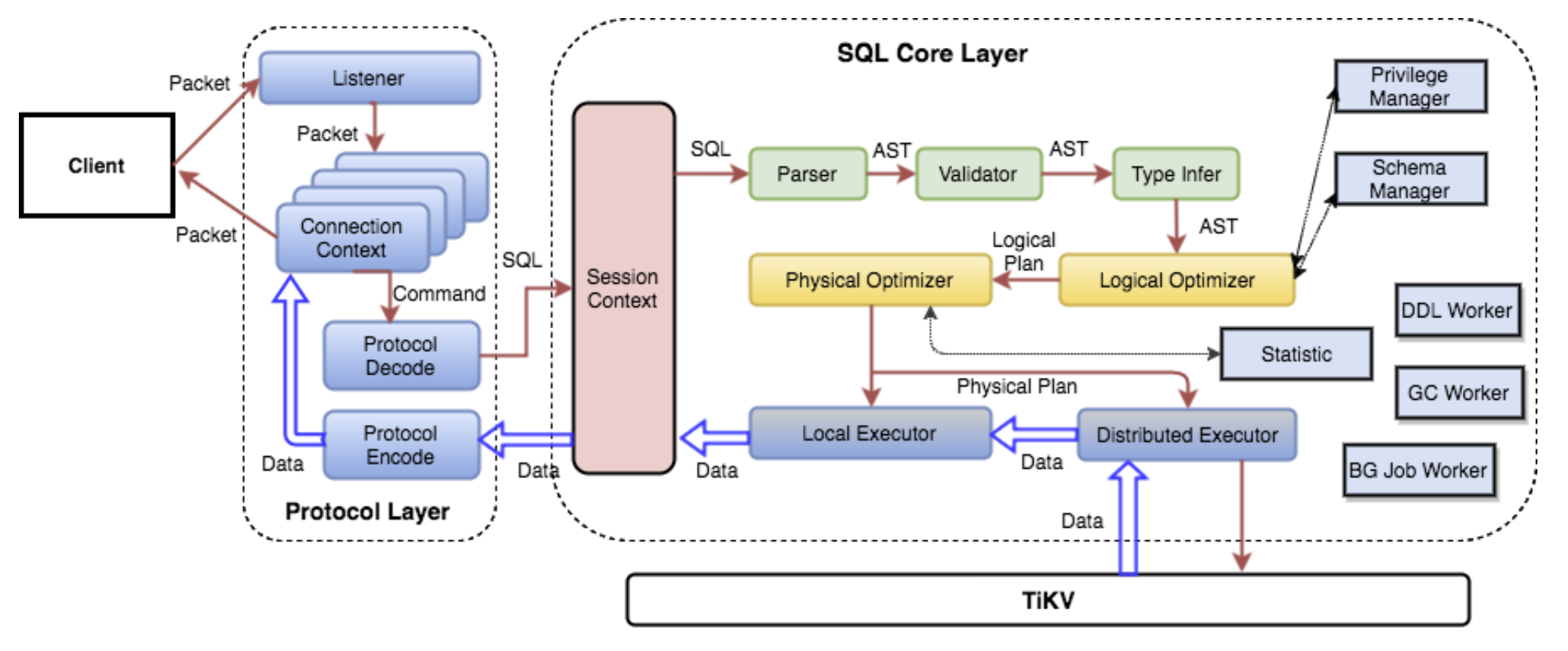
TiDB的SQL层，即TiDB Server，负责将SQL翻译成Key-Value操作，将其转发给共用的分布式Key-Value存储层TiKV，然后组装TiKV返回的结果，最终将查询结果返回给客户端。

这一层的节点都是无状态的，节点本身并不存储数据，节点之间完全对等。

### TiDB Server

TiDB Server：SQL层，对外暴露MySQL协议的连接endpoint，负责接受客户端的连接，执行SQL解析和优化，最终生成分布式执行计划。TiDB层本身是无状态的，实践中可以启动多个TiDB实例，通过负载均衡组件（如LVS、HAProxy或F5）对外提供统一的接入地址，客户端的连接可以均匀地分摊在多个TiDB实例上以达到负载均衡的效果。TiDB Server本身并不存储数据，只是解析SQL，将实际的数据读取请求转发给底层的存储节点TiKV（或TiFlash）。

#### 架构



用户的SQL请求会直接或者通过Load Balancer发送到TiDB Server，TiDB Server会解析MySQL Protocol Packet，获取请求内容，对SQL进行语法解析和语义分析，制定和优化查询计划，执行查询计划并获取和处理数据。数据全部存储在TiKV集群中，所以在这个过程中TiDB Server需要和TiKV交互，获取数据。最后TiDB Server需要将查询结果返回给用户。

#### 分布式运算

##### 表数据与Key-Value的映射关系

TiDB中数据到 (Key, Value)键值对的映射方案。这里的数据主要包括以下两个方面：

表中每一行的数据，以下简称表数据

表中所有索引的数据，以下简称索引数据

###### 表数据

在关系型数据库中，一个表可能有很多列。要将一行中各列数据映射成一个 (Key, Value)键值对，需要考虑如何构造Key。首先，OLTP场景下有大量针对单行或者多行的增、删、改、查等操作，要求数据库具备快速读取一行数据的能力。因此，对应的Key最好有一个唯一ID（显示或隐式的ID），以方便快速定位。其次，很多OLAP型查询需要进行全表扫描。如果能够将一个表中所有行的Key编码到一个区间内，就可以通过范围查询高效完成全表扫描的任务。

基于上述考虑，TiDB中的表数据与Key-Value的映射关系作了如下设计：

1、为了保证同一个表的数据放在一起，方便查找，TiDB会为每个表分配一个表ID，用TableID表示。表ID是一个整数，在整个集群内唯一。

2、TiDB会为表中每行数据分配一个行ID，用RowID表示。行ID也是一个整数，在表内唯一。对于行ID，TiDB做了一个小优化，如果某个表有整数型的主键，TiDB会使用主键的值当做这一行数据的行ID。

每行数据按照如下规则编码成 (Key, Value) 键值对：

Key: tablePrefix{TableID}\_recordPrefixSep{RowID}

Value: [col1, col2, col3, col4]

其中tablePrefix和recordPrefixSep都是特定的字符串常量，用于在Key空间内区分其他数据。

###### 索引数据

TiDB同时支持主键和二级索引（包括唯一索引和非唯一索引）。与表数据映射方案类似，TiDB为表中每个索引分配了一个索引ID，用IndexID表示。

对于主键和唯一索引，需要根据键值快速定位到对应的RowID，因此，按照如下规则编码成 (Key, Value) 键值对：

Key: tablePrefix{tableID}\_indexPrefixSep{indexID}\_indexedColumnsValue

Value: RowID

对于不需要满足唯一性约束的普通二级索引，一个键值可能对应多行，需要根据键值范围查询对应的RowID。因此，按照如下规则编码成 (Key, Value)键值对：

Key: tablePrefix{TableID}\_indexPrefixSep{IndexID}\_indexedColumnsValue\_{RowID}

Value: null

###### 总结

上述所有编码规则中的tablePrefix、recordPrefixSep和indexPrefixSep都是字符串常量，用于在Key空间内区分其他数据，定义如下：

tablePrefix = []byte{'t'}

recordPrefixSep = []byte{'r'}

indexPrefixSep = []byte{'i'}

另外请注意，上述方案中，无论是表数据还是索引数据的Key编码方案，一个表内所有的行都有相同的Key前缀，一个索引的所有数据也都有相同的前缀。这样具有相同的前缀的数据，在TiKV的Key空间内，是排列在一起的。因此只要小心地设计后缀部分的编码方案，保证编码前和编码后的比较关系不变，就可以将表数据或者索引数据有序地保存在TiKV中。采用这种编码后，一个表的所有行数据会按照RowID顺序地排列在TiKV的Key空间中，某一个索引的数据也会按照索引数据的具体的值（编码方案中的indexedColumnsValue）顺序地排列在Key空间内。

##### 元信息管理

TiDB中每个Database和Table都有元信息，也就是其定义以及各项属性。这些信息也需要持久化，TiDB将这些信息也存储在了TiKV中。

每个Database/Table都被分配了一个唯一的ID，这个ID作为唯一标识，并且在编码为Key-Value 时，这个ID都会编码到Key中，再加上m\_前缀。这样可以构造出一个Key，Value中存储的是序列化后的元信息。

除此之外，TiDB还用一个专门的 (Key, Value)键值对存储当前所有表结构信息的最新版本号。这个键值对是全局的，每次DDL操作的状态改变时其版本号都会加1。目前，TiDB把这个键值对持久化存储在PD Server中，其Key是 "/tidb/ddl/global\_schema\_version"，Value是类型为int64的版本号值。TiDB采用Online Schema变更算法，有一个后台线程在不断地检查PD Server中存储的表结构信息的版本号是否发生变化，并且保证在一定时间内一定能够获取版本的变化。

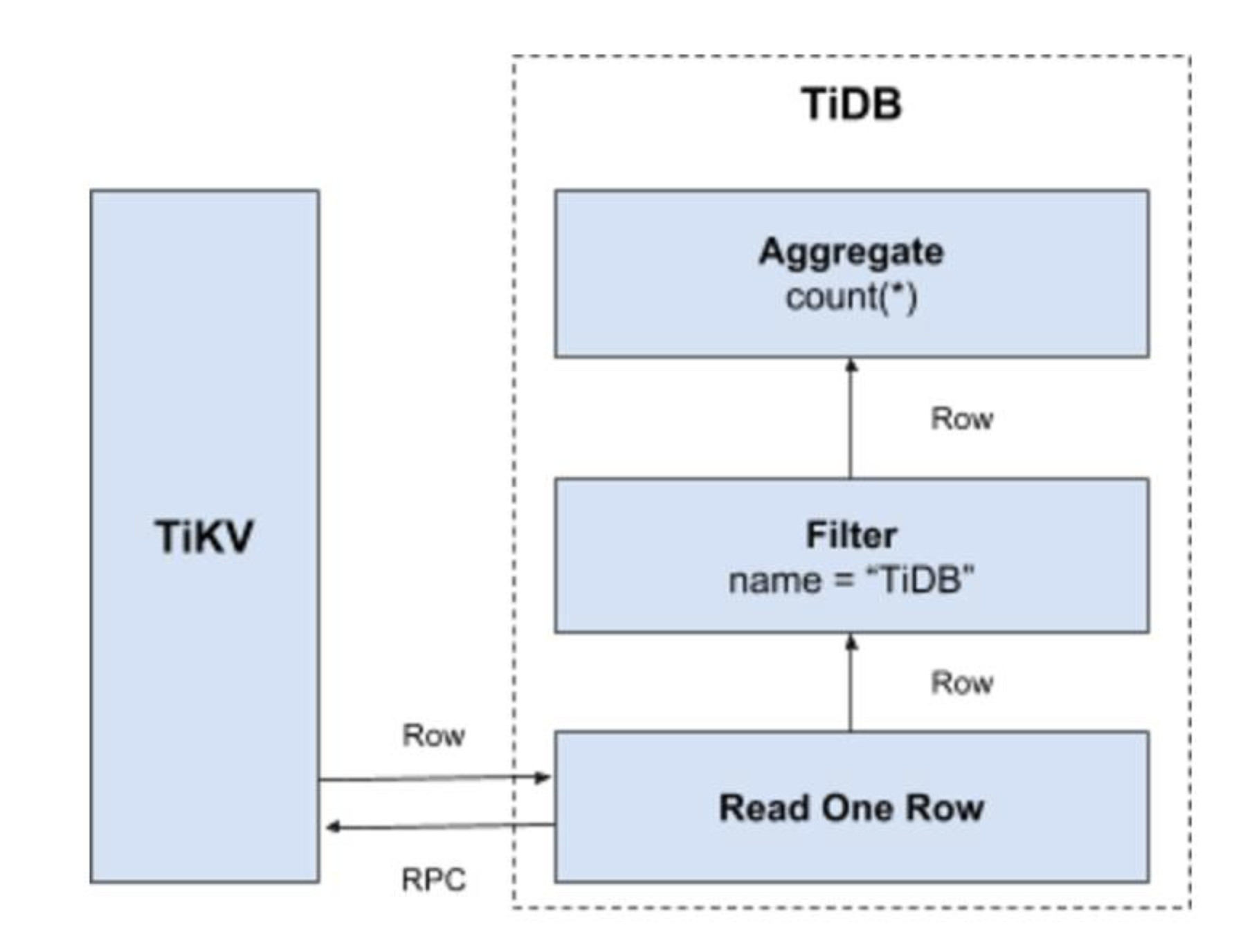
##### 分布式运算

最简单的方案就是通过表数据与Key-Value的映射关系方案，将SQL查询映射为对KV的查询，再通过KV接口获取对应的数据，最后执行各种计算。

比如select count(\*) from user where name = "TiDB"这样一个SQL语句，它需要读取表中所有的数据，然后检查name字段是否是TiDB，如果是的话，则返回这一行。具体流程如下：

1. 构造出Key Range：一个表中所有的RowID都在 [0, MaxInt64)这个范围内，使用0和MaxInt64根据行数据的Key编码规则，就能构造出一个 [StartKey, EndKey)的左闭右开区间。
2. 扫描Key Range：根据上面构造出的Key Range，读取TiKV中的数据。
3. 过滤数据：对于读到的每一行数据，计算name = "TiDB"这个表达式，如果为真，则向上返回这一行，否则丢弃这一行数据。
4. 计算count(\*)：对符合要求的每一行，累计到count(\*)的结果上面。

流程图如下：



这个方案是直观且可行的，但是在分布式数据库的场景下有一些显而易见的问题：

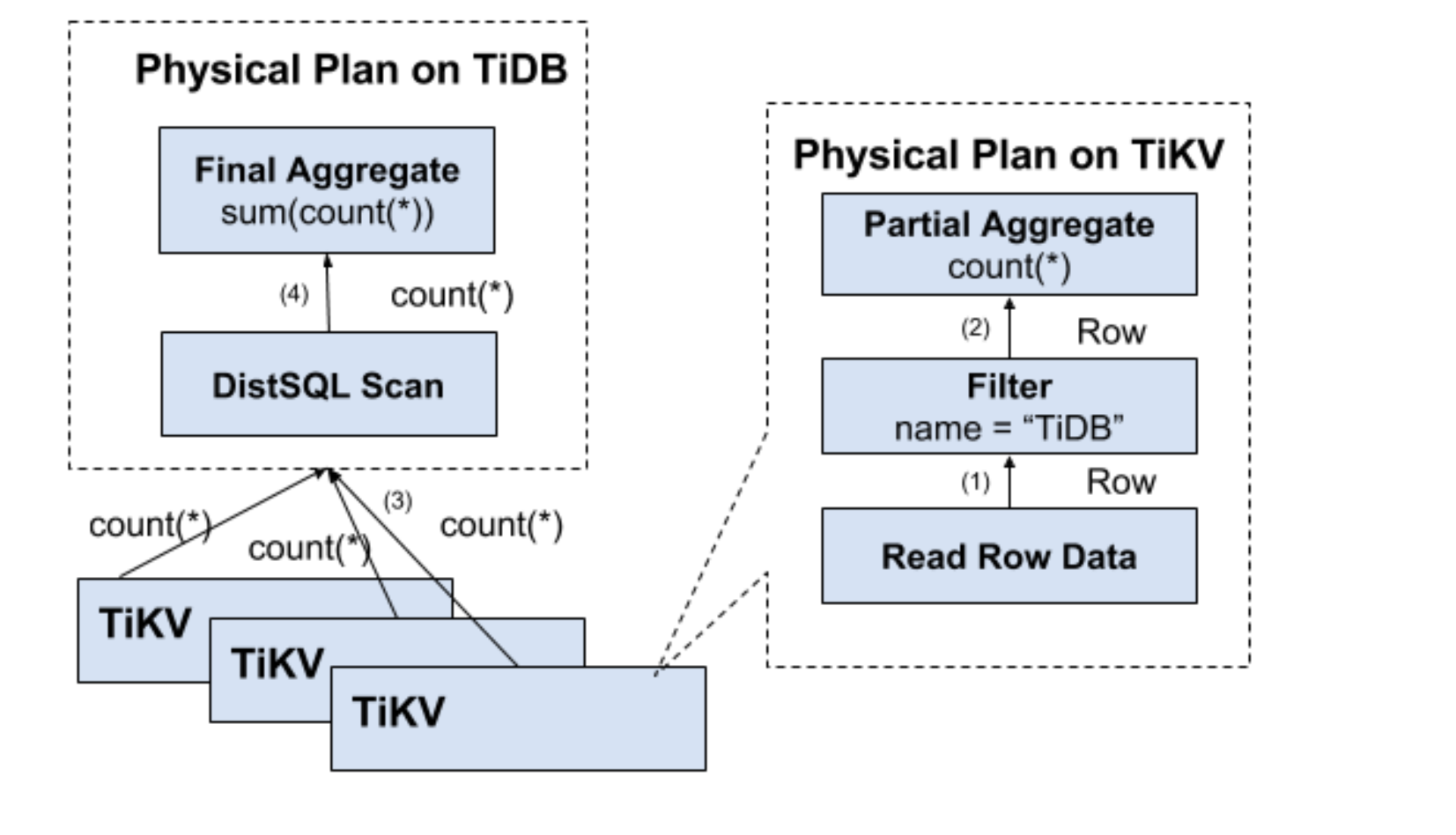
在扫描数据的时候，每一行都要通过KV操作从TiKV中读取出来，至少有一次RPC开销，如果需要扫描的数据很多，那么这个开销会非常大。

并不是所有的行都满足过滤条件name = "TiDB"，如果不满足条件，其实可以不读取出来。

此查询只要求返回符合要求行的数量，不要求返回这些行的值。

为了解决上述问题，计算应该需要尽量靠近存储节点，以避免大量的RPC调用。首先，SQL中的谓词条件name = "TiDB"应被下推到存储节点进行计算，这样只需要返回有效的行，避免无意义的网络传输。然后，聚合函数 Count(\*)也可以被下推到存储节点，进行预聚合，每个节点只需要返回一个Count(\*)的结果即可，再由SQL层将各个节点返回的Count(\*)的结果累加求和。

以下是数据逐层返回的示意图：



## 调度/路由

### PD

PD (Placement Driver) Server：整个TiDB集群的元信息管理模块，负责存储每个TiKV节点实时的数据分布情况和集群的整体拓扑结构，提供TiDB Dashboard管控界面，并为分布式事务分配事务ID。PD不仅存储元信息，同时还会根据TiKV节点实时上报的数据分布状态，下发数据调度命令给具体的TiKV节点，可以说是整个集群的“大脑”。此外，PD本身也是由至少3个节点构成，拥有高可用的能力。建议部署奇数个PD节点。

## 存储引擎

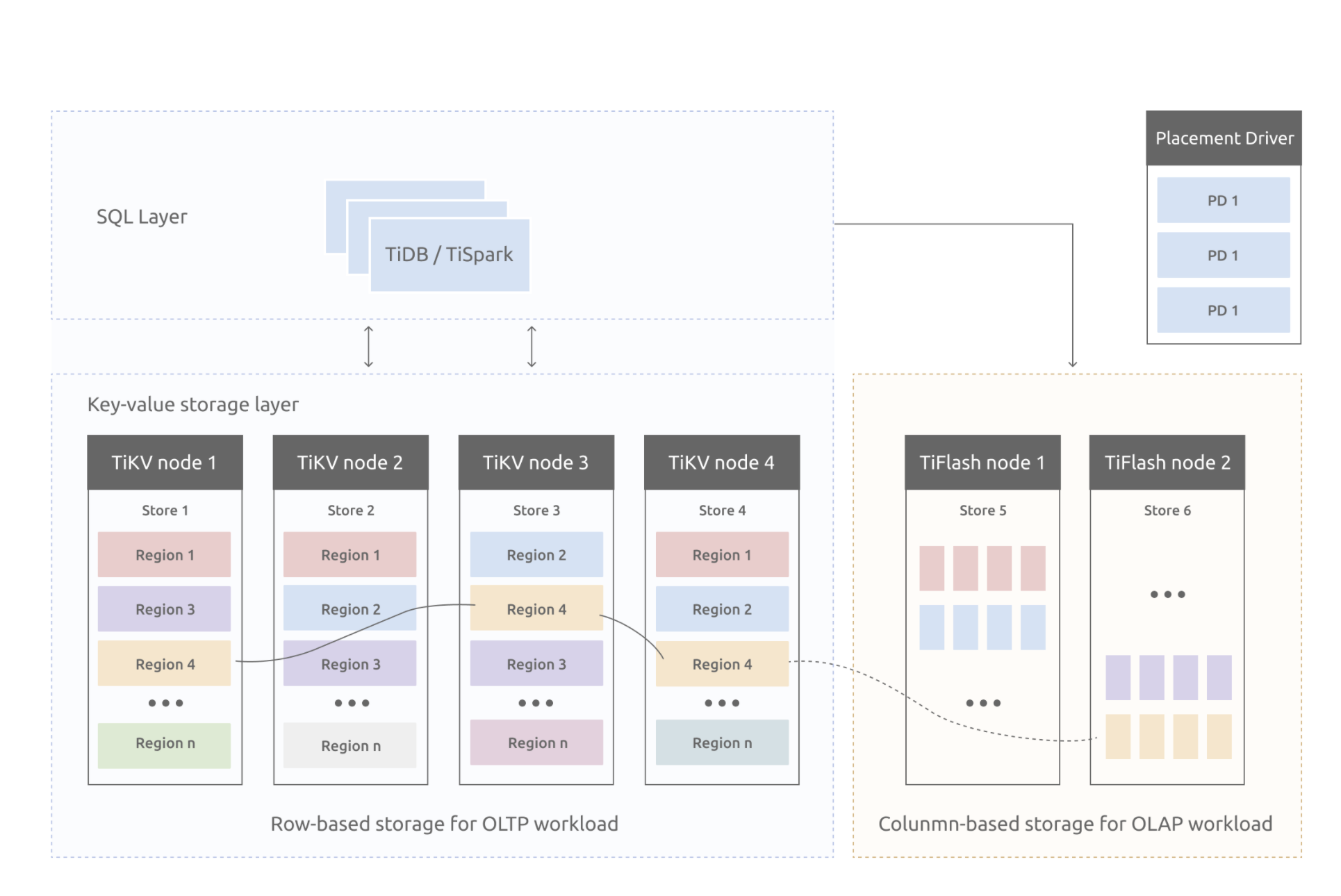
TiDB是一款分布式HTAP数据库，它目前有两种存储节点，分别是TiKV和TiFlash。TiKV采用了行式存储，更适合TP类型的业务；而TiFlash采用列式存储，擅长AP类型的业务。TiFlash通过raft协议从TiKV节点实时同步数据，拥有毫秒级别的延迟，以及非常优秀的数据分析性能。它支持实时同步TiKV的数据更新，以及支持在线DDL。我们把TiFlash作为Raft Learner融合进TiDB的raft体系，将两种节点整合在一个数据库集群中，上层统一通过TiDB节点查询，使得TiDB成为一款真正的HTAP数据库。

### TiKV

参考：<https://docs.pingcap.com/zh/tidb/stable/tikv-overview>

TiKV Server：负责存储数据，从外部看TiKV是一个分布式的提供事务的Key-Value存储引擎。存储数据的基本单位是Region，每个Region负责存储一个 Key Range（从StartKey到EndKey的左闭右开区间）的数据，每个TiKV节点会负责多个Region。TiKV的API在KV键值对层面提供对分布式事务的原生支持，默认提供了SI (Snapshot Isolation) 的隔离级别，这也是TiDB在SQL层面支持分布式事务的核心。TiDB的SQL层做完SQL解析后，会将SQL的执行计划转换为对TiKV API的实际调用。所以，数据都存储在TiKV中。另外，TiKV中的数据都会自动维护多副本（默认为三副本），天然支持高可用和自动故障转移。

注：这里之所以摒弃了传统的行存储采用键值，主要是因为这样可以使用LSM这种存储引擎结构，这样可以提升写性能。



Key-Value Pairs（键值对）

作为保存数据的系统，首先要决定的是数据的存储模型，也就是数据以什么样的形式保存下来。TiKV的选择是Key-Value模型，并且提供有序遍历方法。

TiKV 数据存储的两个关键点：

1、这是一个巨大的Map（可以类比一下C++的std::map），也就是存储的是 Key-Value Pairs（键值对）

2、这个Map中的Key-Value pair按照Key的二进制顺序有序，也就是可以Seek到某一个Key的位置，然后不断地调用Next方法以递增的顺序获取比这个Key大的Key-Value。

注意，本文所说的TiKV的KV存储模型和SQL中的Table无关。不讨论 SQL 中的任何概念，专注于讨论如何实现TiKV这样一个高性能、高可靠性、分布式的Key-Value存储。

#### 本地存储 (RocksDB)

任何持久化的存储引擎，数据终归要保存在磁盘上，TiKV也不例外。但是TiKV没有选择直接向磁盘上写数据，而是把数据保存在RocksDB中，具体的数据落地由RocksDB负责。这个选择的原因是开发一个单机存储引擎工作量很大，特别是要做一个高性能的单机引擎，需要做各种细致的优化，而RocksDB是由Facebook开源的一个非常优秀的单机KV存储引擎，可以满足TiKV对单机引擎的各种要求。这里可以简单的认为RocksDB是一个单机的持久化Key-Value Map。

#### Raft协议

接下来TiKV的实现面临一件更难的事情：如何保证单机失效的情况下，数据不丢失，不出错？

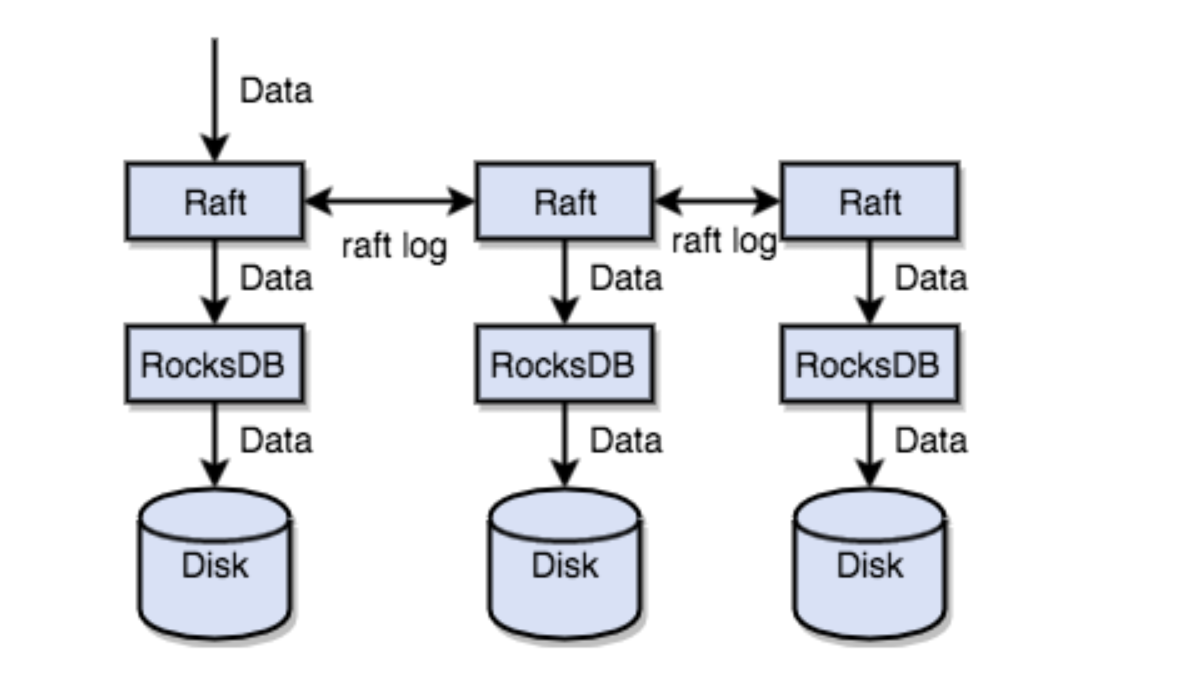
简单来说，需要想办法把数据复制到多台机器上，这样一台机器无法服务了，其他的机器上的副本还能提供服务；复杂来说，还需要这个数据复制方案是可靠和高效的，并且能处理副本失效的情况。TiKV选择了Raft算法。Raft 提供几个重要的功能：

1、Leader（主副本）选举

2、成员变更（如添加副本、删除副本、转移Leader等操作）

3、日志复制

TiKV利用Raft来做数据复制，每个数据变更都会落地为一条Raft日志，通过Raft的日志复制功能，将数据安全可靠地同步到复制组的每一个节点中。不过在实际写入中，根据Raft的协议，只需要同步复制到多数节点，即可安全地认为数据写入成功。



总结一下，通过单机的RocksDB，TiKV可以将数据快速地存储在磁盘上；通过Raft，将数据复制到多台机器上，以防单机失效。数据的写入是通过Raft这一层的接口写入，而不是直接写RocksDB。通过实现Raft，TiKV变成了一个分布式的Key-Value存储，少数几台机器宕机也能通过原生的Raft协议自动把副本补全，可以做到对业务无感知。

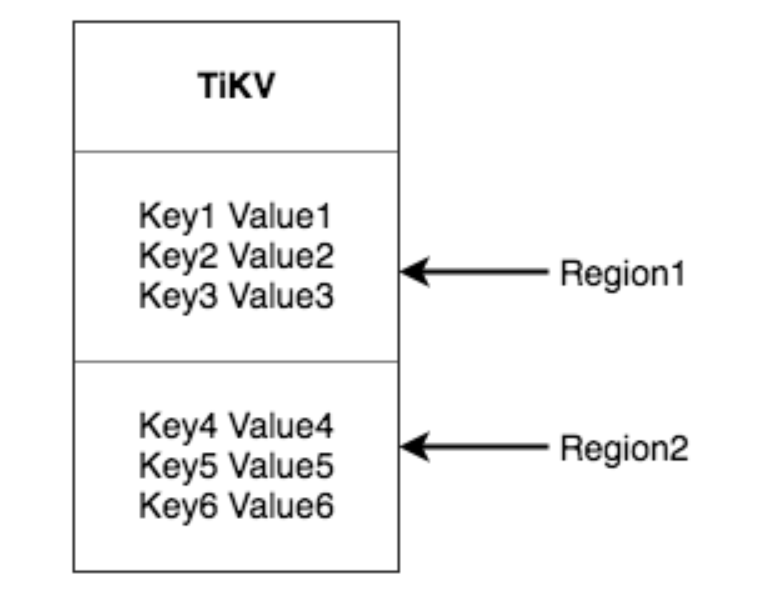
#### Region

首先，为了便于理解，假设所有的数据都只有一个副本。前面提到，TiKV可以看做是一个巨大的有序的KV Map，那么为了实现存储的水平扩展，数据将被分散在多台机器上。对于一个KV系统，将数据分散在多台机器上有两种比较典型的方案：

Hash：按照Key做Hash，根据Hash值选择对应的存储节点。

Range：按照Key分Range，某一段连续的Key都保存在一个存储节点上。

TiKV选择了第二种方式，将整个Key-Value空间分成很多段，每一段是一系列连续的Key，将每一段叫做一个Region，并且会尽量保持每个Region中保存的数据不超过一定的大小，目前在TiKV中默认是96MB。每一个Region都可以用 [StartKey，EndKey)这样一个左闭右开区间来描述。



注意，这里的Region还是和SQL中的表没什么关系。这里的讨论依然不涉及SQL，只和KV有关。

将数据划分成Region后，TiKV将会做两件重要的事情：

1、以 Region为单位，将数据分散在集群中所有的节点上，并且尽量保证每个节点上服务的Region数量差不多。

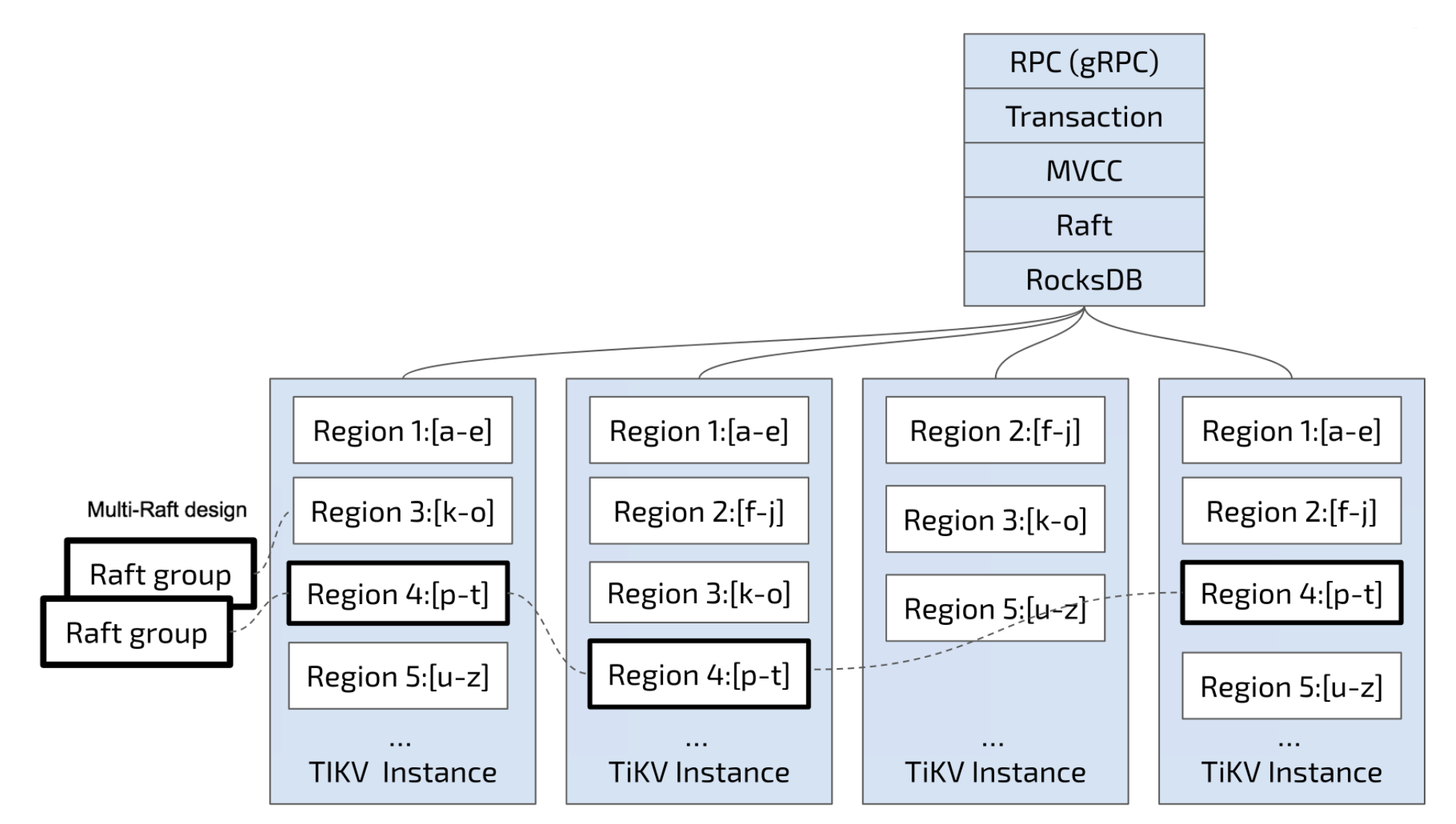
2、以Region为单位做Raft的复制和成员管理。

这两点非常重要：

先看第一点，数据按照Key切分成很多Region，每个Region的数据只会保存在一个节点上面（暂不考虑多副本）。TiDB系统会有一个组件 (PD)来负责将Region尽可能均匀的散布在集群中所有的节点上，这样一方面实现了存储容量的水平扩展（增加新的节点后，会自动将其他节点上的Region调度过来），另一方面也实现了负载均衡（不会出现某个节点有很多数据，其他节点上没什么数据的情况）。同时为了保证上层客户端能够访问所需要的数据，系统中也会有一个组件(PD)记录Region在节点上面的分布情况，也就是通过任意一个Key就能查询到这个 Key在哪个Region中，以及这个Region目前在哪个节点上（即Key的位置路由信息）。

对于第二点，TiKV是以Region为单位做数据的复制，也就是一个Region的数据会保存多个副本，TiKV将每一个副本叫做一个Replica。Replica之间是通过Raft来保持数据的一致，一个Region的多个Replica会保存在不同的节点上，构成一个Raft Group。其中一个Replica会作为这个Group的Leader，其他的Replica作为Follower。默认情况下，所有的读和写都是通过Leader进行，读操作在Leader上即可完成，而写操作再由Leader复制给Follower。

理解了Region之后，应该可以理解下面这张图：



以Region为单位做数据的分散和复制，TiKV就成为了一个分布式的具备一定容灾能力的KeyValue系统，不用再担心数据存不下，或者是磁盘故障丢失数据的问题。

注：将数据按照region划分后，复制的粒度变小了，这样在速度和成本上都有提升。

#### MVCC

很多数据库都会实现多版本并发控制 (MVCC)，TiKV也不例外。设想这样的场景：两个客户端同时去修改一个Key的Value，如果没有数据的多版本控制，就需要对数据上锁，在分布式场景下，可能会带来性能以及死锁问题。TiKV的MVCC实现是通过在Key后面添加版本号来实现，简单来说，没有MVCC之前，可以把TiKV看做这样的：

Key1 -> Value

Key2 -> Value

……

KeyN -> Value

有了MVCC之后，TiKV的Key排列是这样的：

Key1\_Version3 -> Value

Key1\_Version2 -> Value

Key1\_Version1 -> Value

……

Key2\_Version4 -> Value

Key2\_Version3 -> Value

Key2\_Version2 -> Value

Key2\_Version1 -> Value

……

KeyN\_Version2 -> Value

KeyN\_Version1 -> Value

……

注意，对于同一个Key的多个版本，版本号较大的会被放在前面，版本号小的会被放在后面（Key是有序的排列），这样当用户通过一个Key + Version来获取Value的时候，可以通过Key和Version构造出MVCC的Key，也就是Key\_Version。然后可以直接通过RocksDB的 SeekPrefix(Key\_Version) API，定位到第一个大于等于这个Key\_Version的位置。

#### 分布式ACID事务

TiKV的事务采用的是Google在BigTable中使用的事务模型：Percolator，TiKV根据这篇论文实现，并做了大量的优化。

#### Titan

### TiFlash

参考：<https://docs.pingcap.com/zh/tidb/stable/tiflash-overview>

TiFlash：TiFlash是一类特殊的存储节点。和普通TiKV节点不一样的是，在TiFlash内部，数据是以列式的形式进行存储，主要的功能是为分析型的场景加速。

# 分布式事务

# 数据分布

# 复制/一致性

# 备份恢复

## 备份TiDB集群快照

## 使用BR恢复集群

# 容错/故障切换

# 兼容性

# 扩展性/扩容

## 使用TiUP扩容缩容TiDB集群

## 使用TiDB Operator

# 高并发

# 高可用

# 数据安全

## 安全加固

### 为TiDB客户端服务端间通信开启加密传输

### 为TiDB组件间通信开启加密传输

### 生成自签名证书

### 静态加密

### 为TiDB落盘文件开启加密

### 日志脱敏

## 权限

### 与MySQL安全特性差异

### 权限管理

### TiDB用户账户管理

### 基于角色的访问控制

### TiDB证书鉴权使用指南

# 数据压缩

# 数据迁移

## 概述

数据迁移方案如下：

全量数据迁移。

数据导入：使用TiDB Lightning将Aurora Snapshot，CSV文件或Mydumper SQL文件的数据全量导入到TiDB集群。

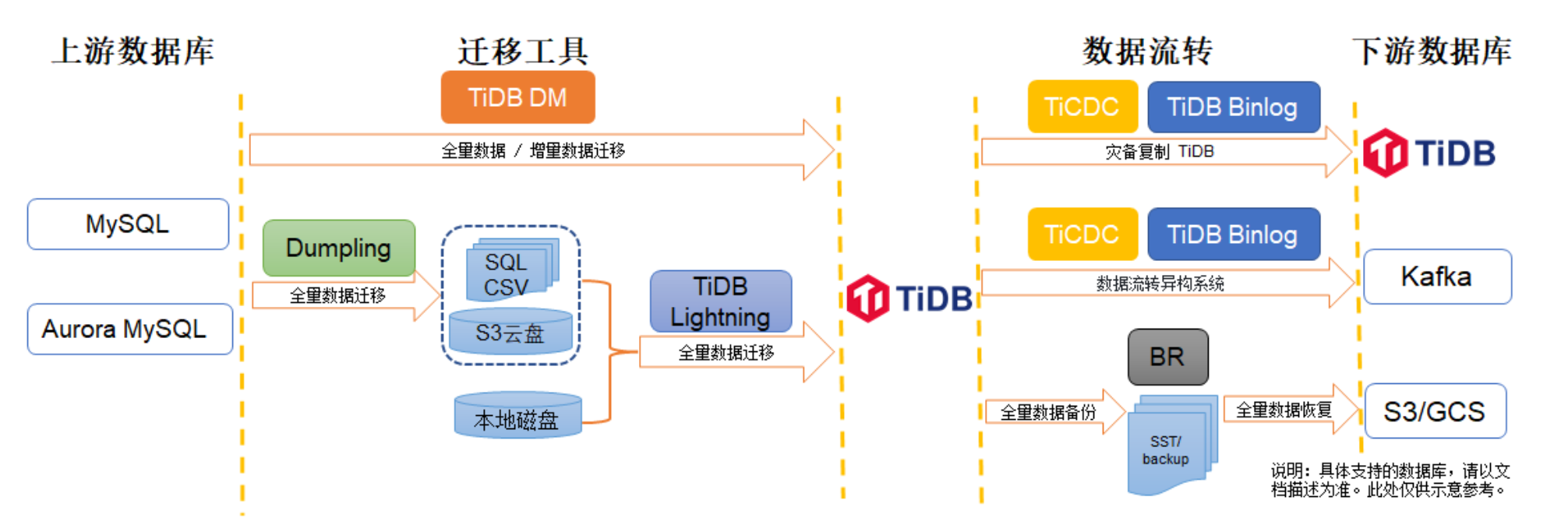
数据导出：使用Dumpling将TiDB集群的数据全量导出为CSV文件或Mydumper SQL文件，从而更好地配合从MySQL数据库或MariaDB数据库进行数据迁移。

TiDB DM（Data migration）也提供了适合小规模数据量数据库（例如小于1 TiB）的全量数据迁移功能。

## 迁移工具

TiDB提供了丰富的数据迁移相关的工具，用于全量迁移、增量迁移、备份恢复、数据同步等多种场景。

下图显示了各迁移工具的使用场景：



### TiDB DM

### Dumpling

### TiDB Lightning

### Backup & Restore (BR)

### TiCDC

### TiDB Binlog

### sync-diff-inspector

# 数据集成

数据集成一般是指数据在各个独立的数据源之间流动、转换和汇集。随着数据量的爆炸式增长和数据价值被深度挖掘，对数据集成的需求越来越普遍和迫切。为了避免TiDB成为数据孤岛，顺利与各个数据系统进行集成，TiCDC提供将TiDB增量数据变更日志实时同步到其他数据系统的能力。

## 与Confluent Cloud进行数据集成

你可以使用TiCDC将TiDB的增量数据同步到Confluent Cloud，并借助Confluent Cloud的能力最终将数据分别同步到ksqlDB、Snowflake、SQL Server。

## 与Apache Kafka和Apache Flink进行数据集成

可以使用TiCDC将TiDB的增量数据同步到Apache Kafka，并使用Apache Flink消费Kafka中的数据。

# 性能调优

## 系统调优

## 软件调优

### 配置

#### TiDB内存调优

#### TiKV线程调优

#### TiKV内存调优

#### TiKV Follower Reader

#### TiFlash调优

### 下推计算结果缓存

## SQL性能调优

# 运维/监控告警

## 监控指标

### Overview面板

### TiDB面板

### PD面板

### TiKV面板

### TiFlash监控指标

### TiCDC监控指标

# 故障诊断