TiDB Schema**设计**

[TiDB Schema**设计**](#_bookmark0)[1**、**TiDB**数据库表结构**](#_bookmark1)

* 1. [TiDB数据库与MySQL的差异性对⽐](#_bookmark2)
  2. [表](#_bookmark3)
     1. [什么是表](#_bookmark4)
     2. [TiDB的主键](#_bookmark5)
     3. [表的分类](#_bookmark6)
  3. [schema的映射原理](#_bookmark7)
     1. [TiDB的region存储⽅式](#_bookmark8)
     2. [聚簇表的映射](#_bookmark9)
     3. [⾮聚簇表的映射](#_bookmark10)
     4. [聚簇表与⾮聚簇表映射规则](#_bookmark11)
     5. [聚簇表与⾮聚簇表设计](#_bookmark12)
     6. [⾮聚簇表之shard\_row\_id\_bits和PRE\_SPLIT\_REGIONS](#_bookmark13)
  4. [分区表](#_bookmark14)
     1. [分区表](#_bookmark15)
     2. [分区对于NULL值的处理](#_bookmark16)
     3. [分区管理](#_bookmark17)
  5. [数值类型](#_bookmark18)
  6. [⾃增ID](#_bookmark19)
     1. [AUTO\_INCREMENT基本概念](#_bookmark20)
     2. [AUTO\_INCREMENT实现原理](#_bookmark21)
     3. [AUTO\_INCREMENT使⽤限制](#_bookmark23)
     4. [AUTO\_RANDOM基础概念](#_bookmark22)
     5. [AUTO\_RANDOM实现原理](#_bookmark26)
     6. [AUTO\_RANDOM使⽤限制](#_bookmark25)
     7. [auto\_increment和auto\_random示例](#_bookmark24)
  7. [TiDB数据与对象使⽤限制](#_bookmark27)
  8. [Schema设计建议](#_bookmark28)
     1. [⾼兼容schema设计](#_bookmark29)
     2. [⾼性能schema设计](#_bookmark30)

[2**、数据库索引设计**](#_bookmark31)

* 1. [TiDB索引构成原理](#_bookmark32)
     1. [索引的KV映射原理](#_bookmark33)
  2. [索引设计](#_bookmark34)
     1. [索引的创建与映射关系](#_bookmark35)
     2. [表达式索引](#_bookmark36)
     3. [不可⻅索引](#_bookmark37)
  3. [索引最佳实践](#_bookmark38)

[3**、**TiDB**数据库库系统表使⽤**](#_bookmark39)

* 1. [TiDB系统表介绍](#_bookmark40)
     1. [mysql库](#_bookmark41)
     2. [Information\_schema](#_bookmark42)
  2. [系统表ops实际使⽤](#_bookmark43)
     1. [慢SQL应⽤-初步了解](#_bookmark45)

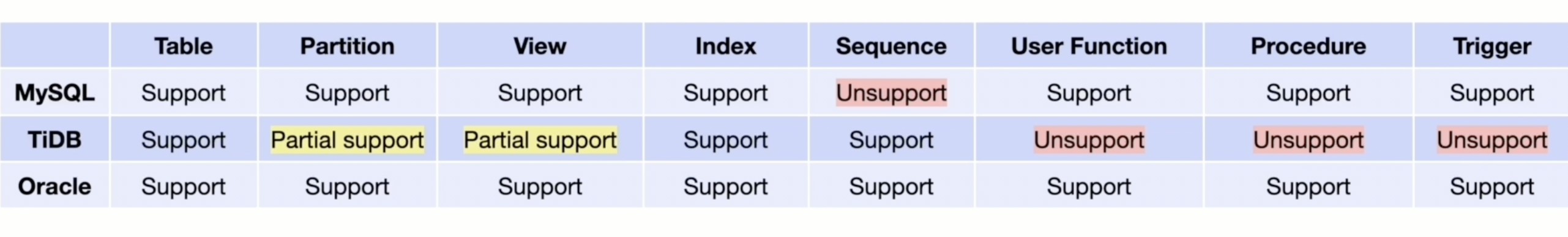
[3.3.2 热点表查询](#_bookmark44)

[3.3.3 SQL阻塞查询](#_bookmark46)

[3.2.4 其余mysql兼容查询](#_bookmark47)

1**、**TiDB**数据库表结****构**

* 1. TiDB**数据库与**MySQL**的差异性对⽐**



TiDB与mysql差异

1. 不⽀持:存储过程,触发器,⾃定义函数
2. 分区表性能⼀般
3. 视图部分⽀持

# 表

## 什么是表

* 表是集群中的⼀种逻辑对象，它从属于数据库，⽤于保存从SQL 中发送的数据。
* 表以⾏和列的形式组织数据记录,⼀张表⾄少有⼀列。若在表中定义了 n 个列，那么每⼀⾏数据都将拥有与这 n 个列中完全⼀致的字段。

建表语句：

CREATE TABLE {table\_name} ( {elements} );

* {table\_name}: 表名
* {elements}: 以逗号分隔的表元素列表，⽐如列定义，主键定义等。

CREATE TABLE `test`.`users` (

`id` bigint primary key,

`student\_name` varchar(100),

`birthday` varchar(100),

`phone` int(11),

`score` int(11)

);

table\_id的存储⽅式：information\_schema.tables

> select \* from information\_schema.tables where table\_schema='test' and table\_name='users' limit 1\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* TABLE\_CATALOG: def

TABLE\_SCHEMA: test

TABLE\_NAME: users

TABLE\_TYPE: BASE TABLE ENGINE: InnoDB

VERSION: 10

ROW\_FORMAT: Compact TABLE\_ROWS: 0

AVG\_ROW\_LENGTH: 0

DATA\_LENGTH: 0

MAX\_DATA\_LENGTH: 0

INDEX\_LENGTH: 0

DATA\_FREE: 0 AUTO\_INCREMENT: NULL

CREATE\_TIME: 2022-11-14 16:45:33 UPDATE\_TIME: NULL

CHECK\_TIME: NULL

TABLE\_COLLATION: utf8mb4\_bin CHECKSUM: NULL

CREATE\_OPTIONS: TABLE\_COMMENT:

TIDB\_TABLE\_ID: 2240 #TABLE\_ID TIDB\_ROW\_ID\_SHARDING\_INFO: NOT\_SHARDED(PK\_IS\_HANDLE)

TIDB\_PK\_TYPE: CLUSTERED

* + 1. TiDB**的主键**

1、定义：

* + - * 主键是⼀个或⼀组列，这个由所有主键列组合起来的值是数据⾏的唯⼀标识。
      * 主键在 CREATE TABLE 语句中定义。
      * 主键约束要求所有受约束的列仅包含⾮ NULL 值。

2、TiDB的主键与MySQL不⼀致的地⽅

* + - * InnoDB 中，Primary Key 的语义为：唯⼀，不为空，且为聚簇索引。
      * TiDB 中，Primary Key 的定义为：唯⼀，不为空。但主键不保证为聚簇索引。⽽是由另⼀组关键字 CLUSTERED、NONCLUSTERED 额外控制 Primary Key 是否为聚簇索引，若不指定，则由系统变量 @@global.tidb\_enable\_clustered\_index 影响。

3、使⽤说明

* + - * ⼀个表可以没有主键，主键也可以是⾮整数类型。/\*但此时 TiDB 就会创建⼀个 \_tidb\_rowid 作为隐式主键。隐式主键 \_tidb\_rowid 因为其单调递增的特性，可能在⼤批量写⼊场景下会导致写⼊热点，如果你写⼊量密集，可考虑通过 SHARD\_ROW\_ID\_BITS 和 PRE\_SPLIT\_REGIONS 两参数控制打散。但这可能导致读放⼤，可以⾃⾏取 舍。\*/

4、选择主键时应遵守的规则

* 在表内定义⼀个主键或唯⼀索引。
* 尽量选择有意义的列作为主键。
* 出于为性能考虑，尽量避免存储超宽表，表字段数不建议超过 60 个，建议单⾏的总数据⼤⼩不要超过 64K，数据

⻓度过⼤字段最好拆到另外的表。

* 不推荐使⽤复杂的数据类型。
* 需要 JOIN 的字段，数据类型保障绝对⼀致，避免隐式转换。

* 避免在单个单调数据列上定义主键。如果你使⽤单个单调数据列（例如：AUTO\_INCREMENT 的列）来定义主键，有可能会对写性能产⽣负⾯影响。可能的话，使⽤ AUTO\_RANDOM 替换 AUTO\_INCREMENT（这会失去主键的连续和递增特性）。
* 如果你 必须 在单个单调数据列上创建索引，且有⼤量写⼊的话。请不要将这个单调数据列定义为主键，⽽是使⽤

AUTO\_RANDOM 创建该表的主键，或使⽤ SHARD\_ROW\_ID\_BITS 和 PRE\_SPLIT\_REGIONS 打散 \_tidb\_rowid。

## 表的分类

#表的分类

* 聚簇表(Clustered Table) = 聚簇索引表(index-organized tables) = 索引组织表（index-organized tables）
  + 表示该表的主键为聚簇索引
  + 主键列数据（键） - ⾏数据（值）
* ⾮聚簇表(Non-Clustered Table)
  + 表示该表的主键为⾮聚簇索引
  + \_tidb\_rowid（键）- ⾏数据（值）
  + 主键列数据（键） - \_tidb\_rowid（值）
  1. schema**的映射原****理** 
     1. TiDB**的**region**存储⽅式**

TiDB中的数据是在RocksDB中以KV键值对的⽅式存储的数据存储管理的基本单元为Region

* 每⼀个region默认⼤⼩为96MB
* 每⼀个Region按照左闭右开的区间划分数据存储范围，如：[a ~ d),[d ~ g)
* 每⼀个schema会被分配⼀个唯⼀的TableID

## 聚簇表的映射

关系型数据⼀张表

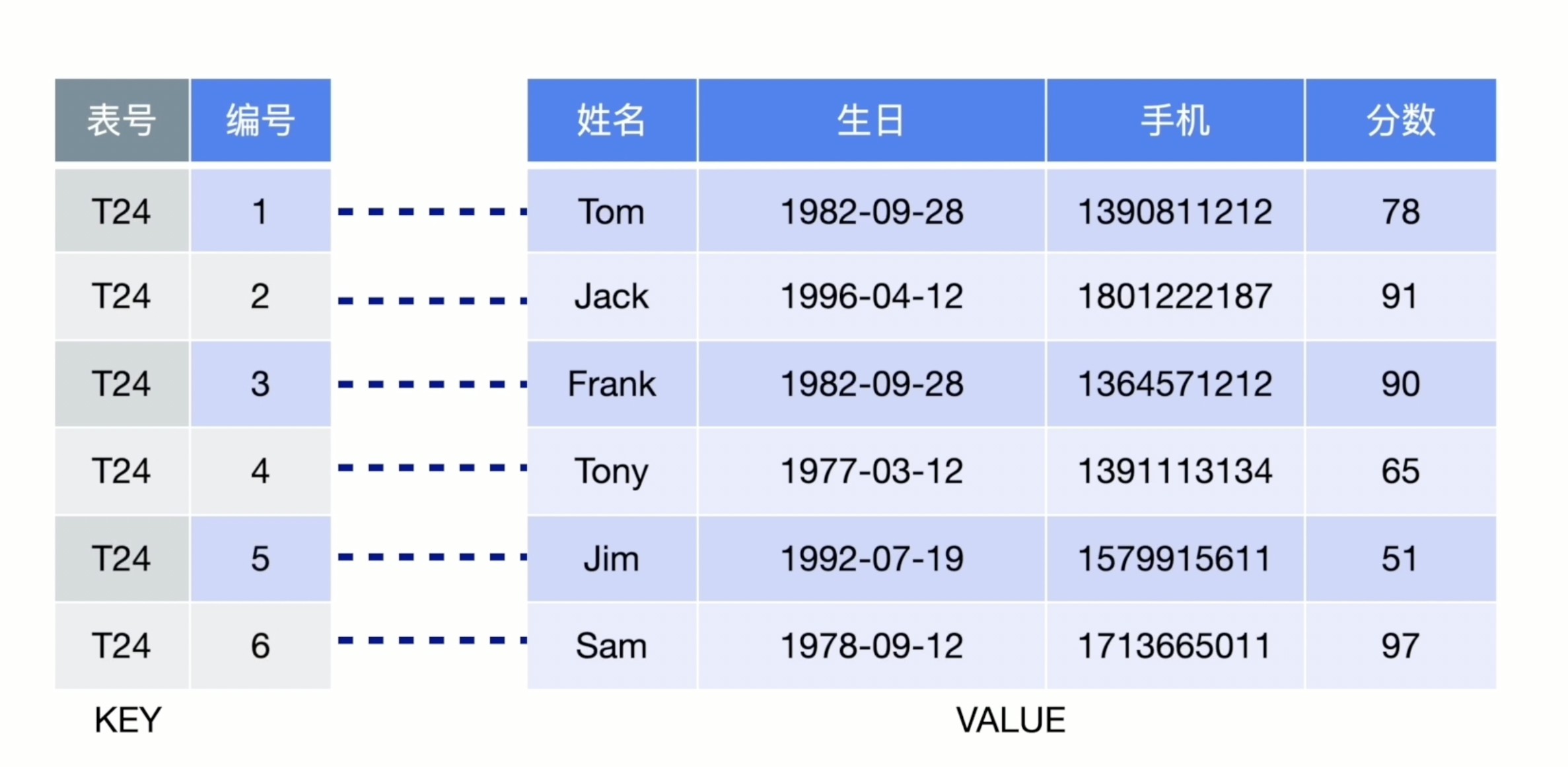


schema—>kv(聚簇表)

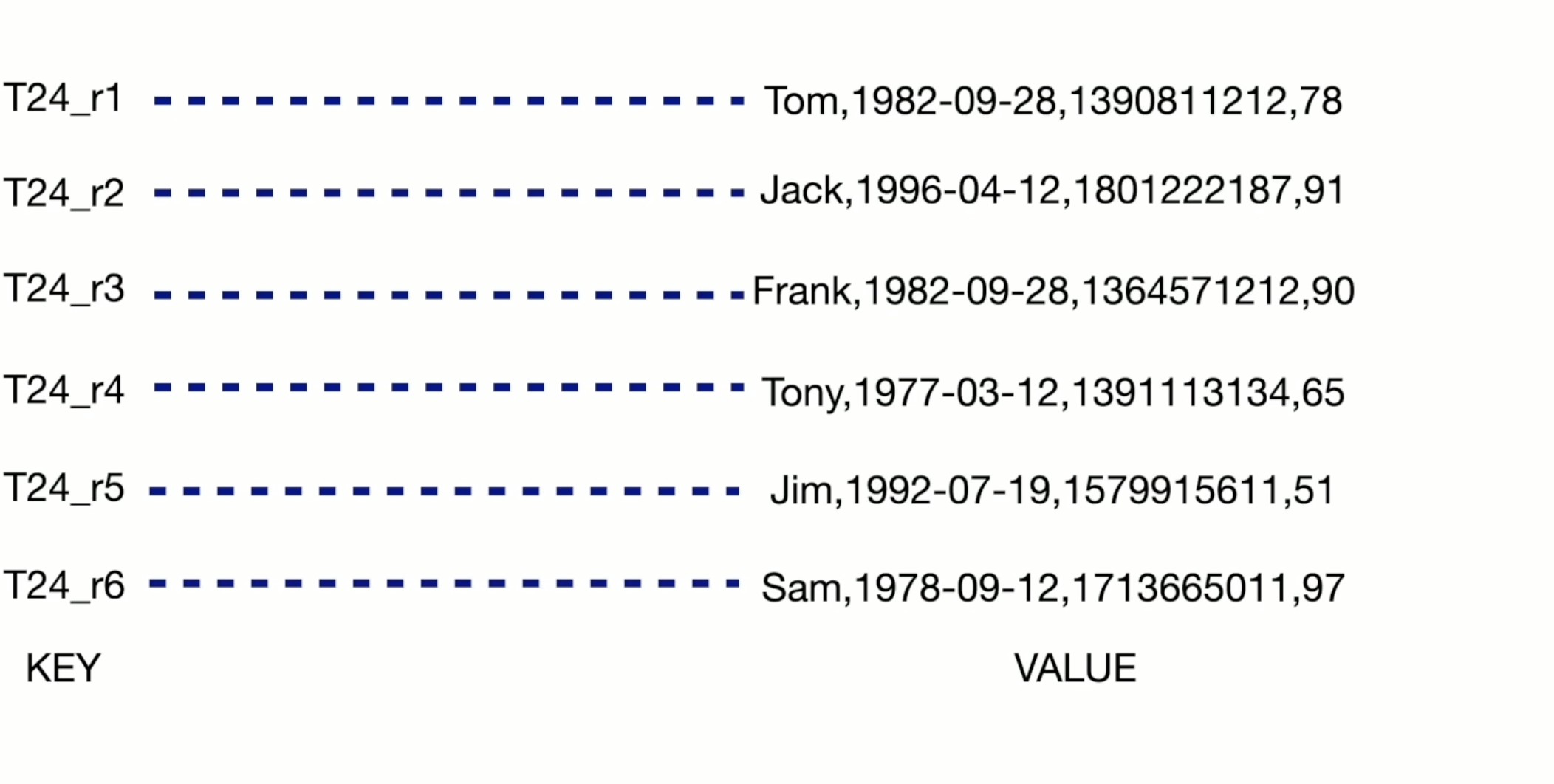


保证key唯⼀

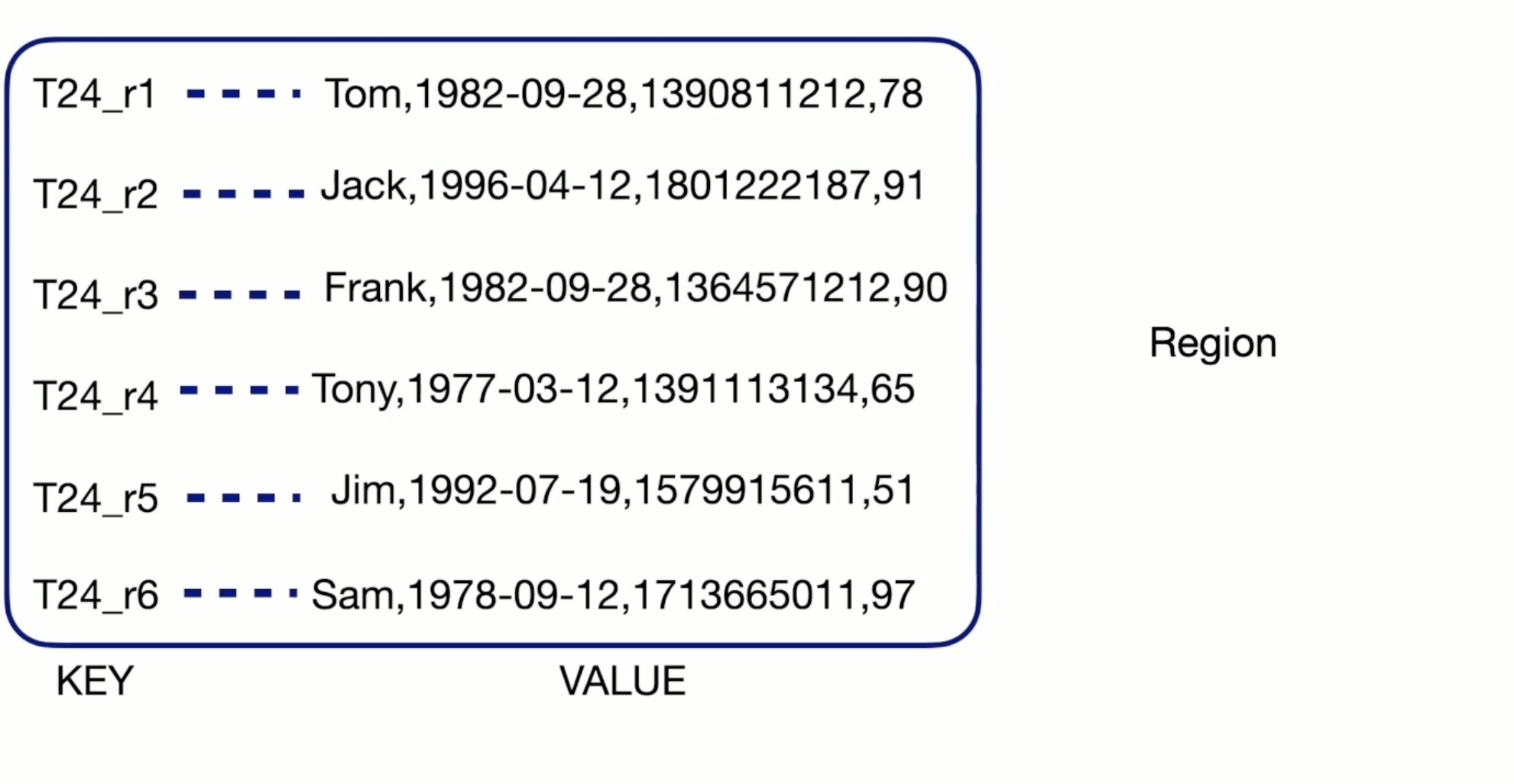
key=表编号+主键



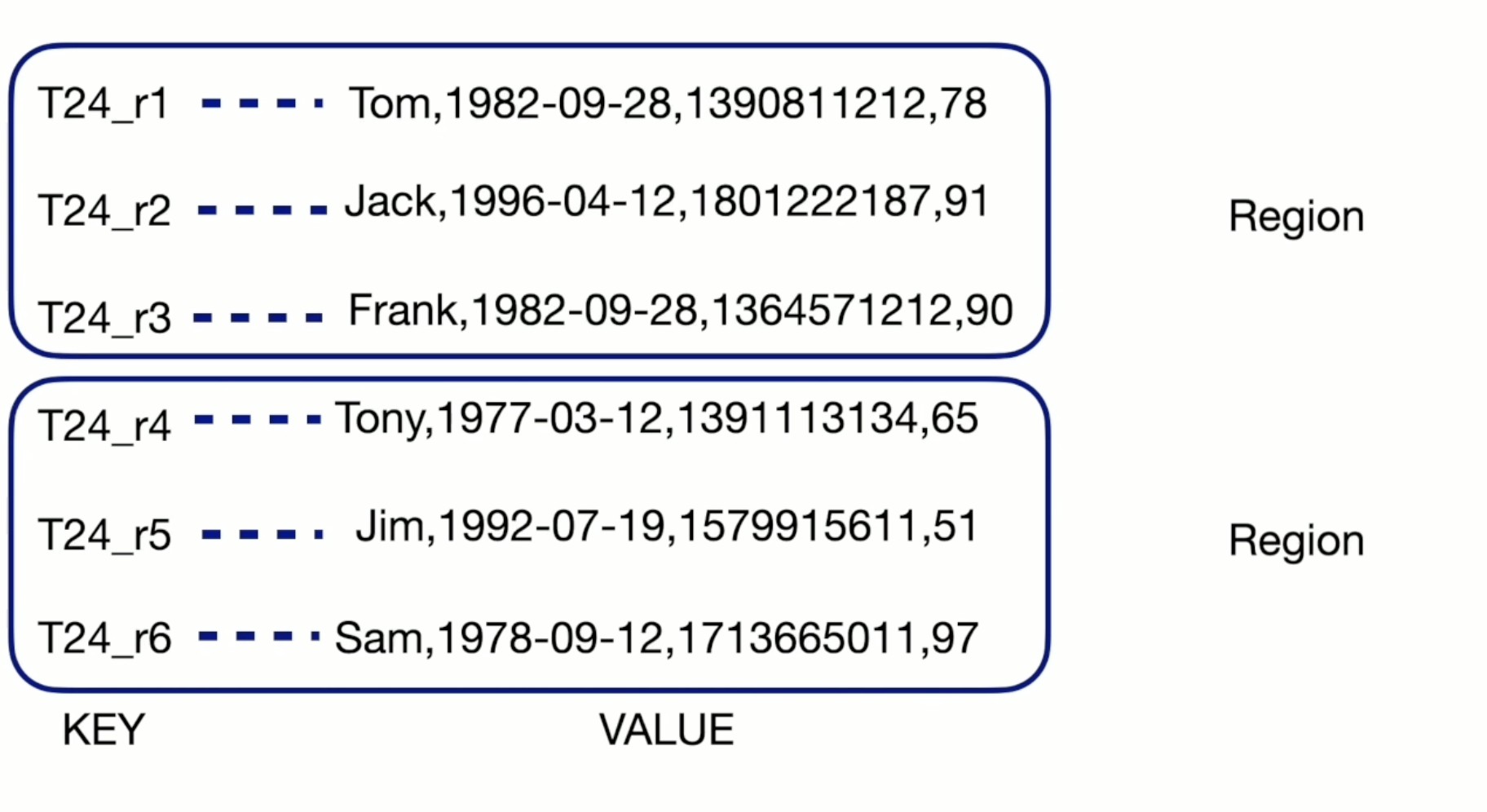
最终变成k-v形态



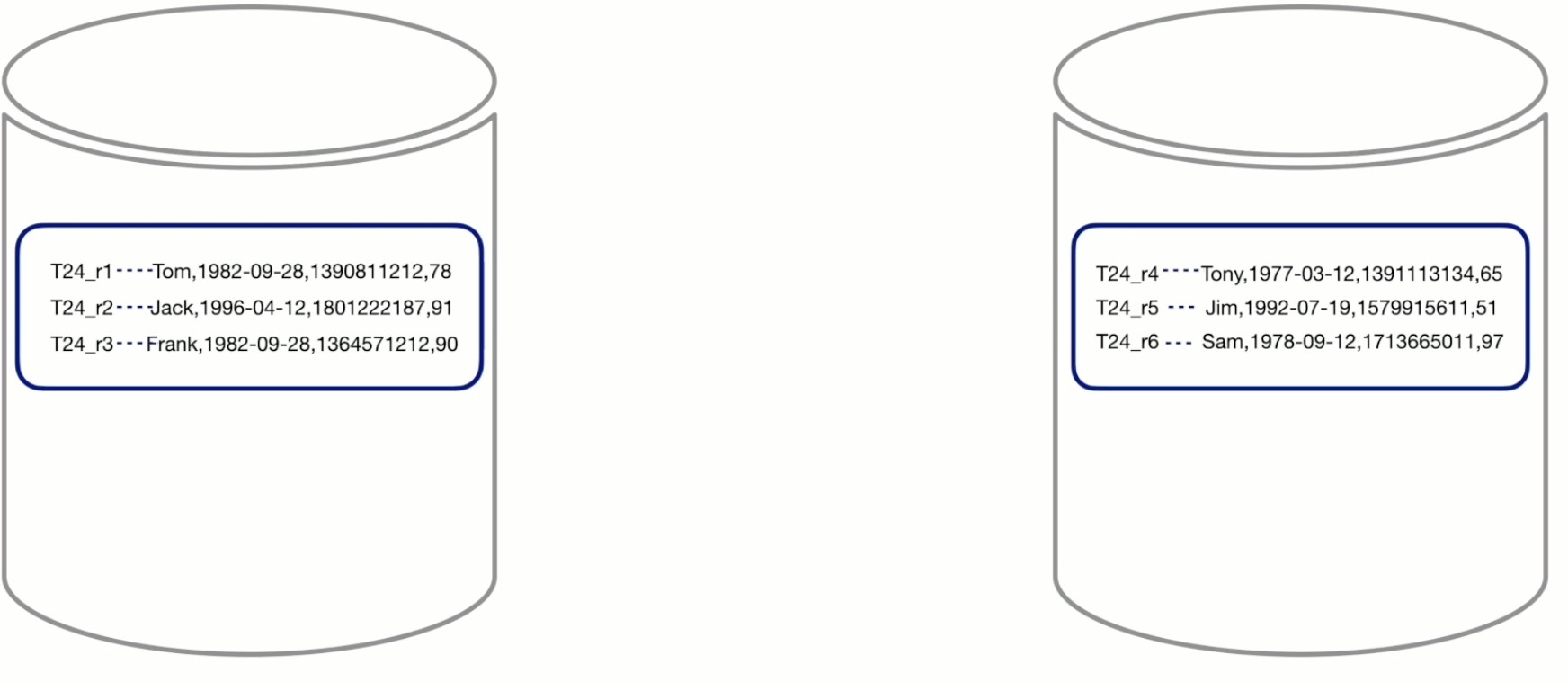
多个键值对放在⼀起变成⼀个region,默认96MB

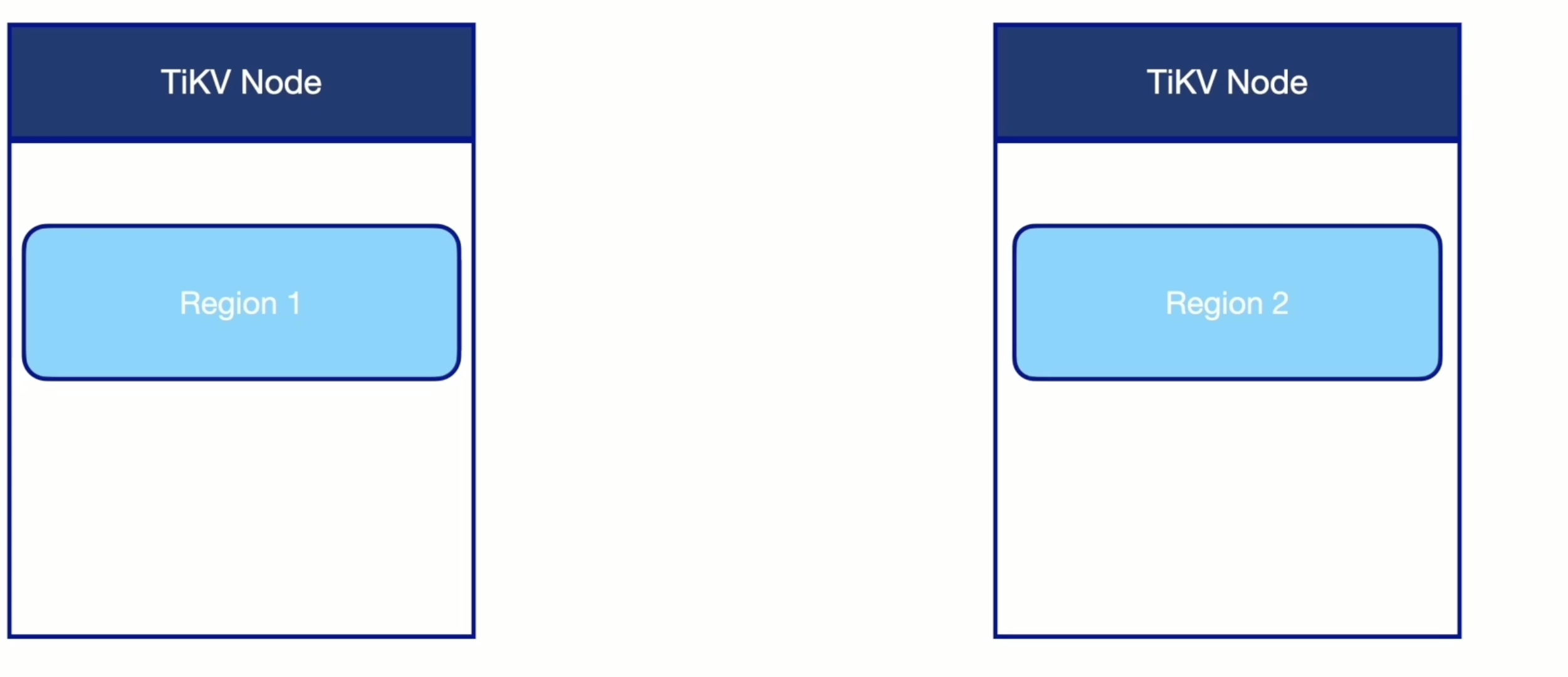


达到96MB后⼀个region分裂成两个region



PD调度将不同的region分布在不同的tikv节点中





## ⾮聚簇表的映射

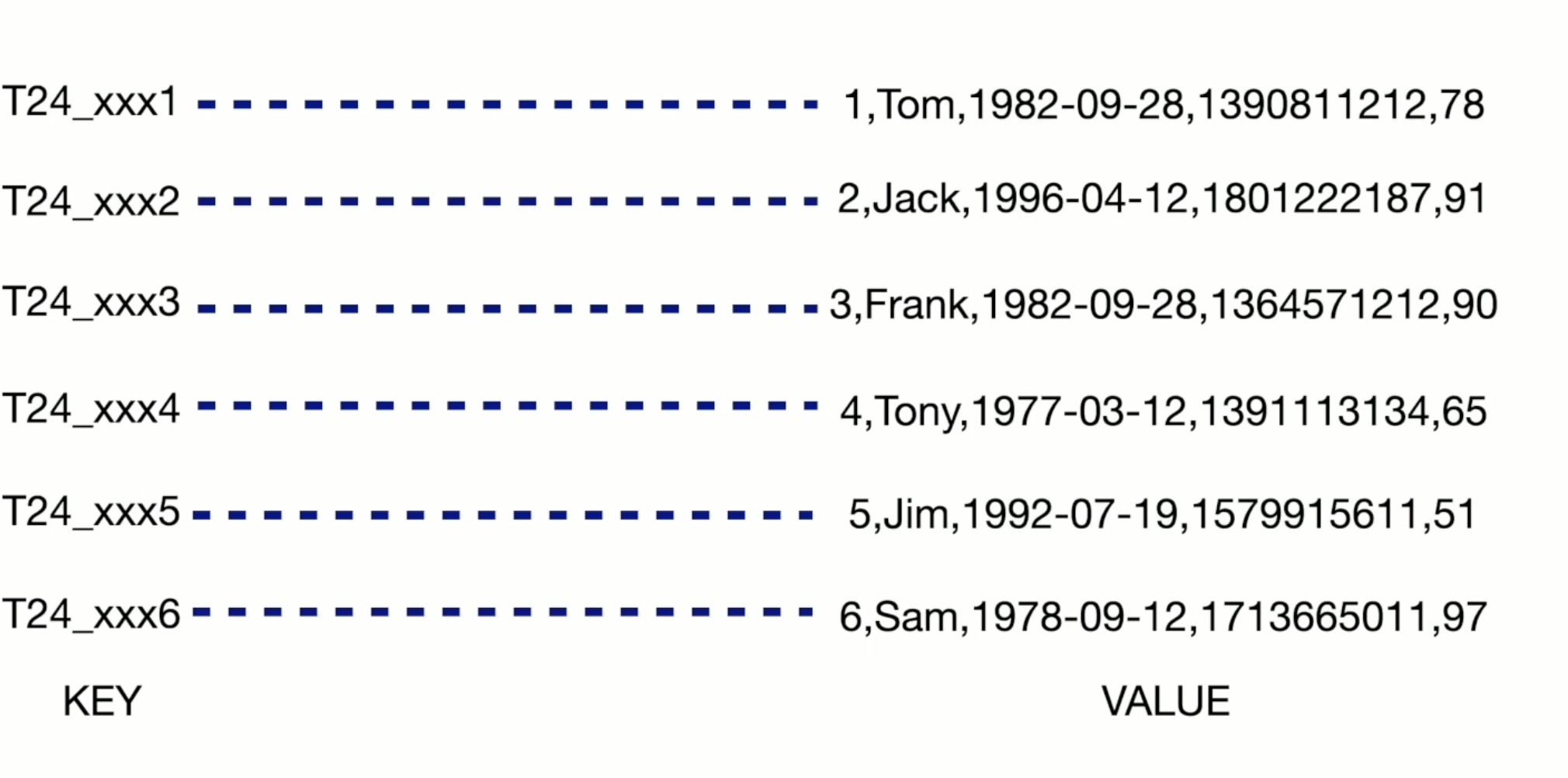
key：系统⾃动或隐式的为每⼀⾏⽣成rowid



table\_id+rowid 组成key



最终形成KV键值对



多个region分裂



## 聚簇表与⾮聚簇表映射规则

\*\* 1、⾮聚簇表KV的映射规则\*\*

Key: tablePrefix{TableID}\_recordPrefixSep{\_Tidb\_RowID} Value: [col1,col2,col3,col4] #KV存储中Value存储真实的⾏数据

\*\* 2、聚簇表的KV映射规则 \*\*

Key: tablePrefix{TableID}\_recordPrefixSep{Col1} #col1为cluster index

Value: [col2,col3,col4]

\*\* 3、示例：\*\*

CREATE TABLE `ratings` (

`id` bigint PRIMARY KEY CLUSTERED,

`name` varchar(20),

`role` varchar(20)

);

#插⼊3⾏数据

insert into ratings values (1,'tidb','sql layer'); insert into ratings values (2,'tikv','KV engine'); insert into ratings values (3,'PD','manager');

#数据的KV映射关系

t10\_r1 --> ["tidb","sql layer"]

t10\_r2 --> ["tikv","KV engine"]

t10\_r3 --> ["PD","manager"]

* + 1. **聚簇表与⾮聚簇表设计**

聚簇表设计

1、定义：

* 表中⾏存储的顺序与主键存储的顺序⼀致即为聚簇表
* 表的主键即为KV映射中Key的⼀部分
* 通过主键访问⾏记录时，可以直接获取⾏记录

2、优劣势：优势：

* 插⼊数据时会减少⼀次从⽹络写⼊索引数据。
* 等值条件查询仅涉及主键时会减少⼀次从⽹络读取数据。
* 范围条件查询仅涉及主键时会减少多次从⽹络读取数据。
* 等值或范围条件查询仅涉及主键的前缀时会减少多次从⽹络读取数据。劣势：
* 批量插⼊⼤量取值相邻的主键时，可能会产⽣较⼤的写热点问题。
* 当使⽤⼤于 64 位的数据类型作为主键时，可能导致表数据需要占⽤更多的存储空间。该现象在存在多个⼆级索引时尤为明显。

3、建表语句：

CREATE TABLE t (a BIGINT PRIMARY KEY CLUSTERED, b VARCHAR(255));

CREATE TABLE t1 (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) CLUSTERED);

使⽤可执⾏的注释语法指定⾮聚簇索引属性：

CREATE TABLE t (a BIGINT PRIMARY KEY /\*T![clustered\_index] CLUSTERED \*/, b VARCHAR(255));

CREATE TABLE t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /\*T![clustered\_index] CLUSTERED \*/);

4、未显式指定聚簇关键字的语句，默认⾏为受系统变量 @@global.tidb\_enable\_clustered\_index 影响。该变量有三个取值：

- OFF 表示所有主键默认使⽤⾮聚簇索引。

* ON 表示所有主键默认使⽤聚簇索引。
* INT\_ONLY，默认值，⾏为受alter-primary-key参数控制（true:主键⾮聚簇，false:单整主键默认聚簇）

5、添加、删除聚簇索引

* 不⽀持在建表之后添加或删除聚簇索引
* 不⽀持聚簇索引和⾮聚簇索引的互相转换

ALTER TABLE t ADD PRIMARY KEY(b, a) CLUSTERED; -- 暂不⽀持

ALTER TABLE t DROP PRIMARY KEY; -- 如果主键为聚簇索引，则不⽀持

ALTER TABLE t DROP INDEX `PRIMARY`; -- 如果主键为聚簇索引，则不⽀持

ERROR 8200 (HY000): Unsupported drop primary key when the table's pkIsHandle is true

⾮聚簇表设计

1、定义：

* + - * 表中⾏数据存储顺序与主键存储的顺序不⼀定⼀致的表即为⾮聚簇表
      * ⾏数据的键由tidb内部隐式分配的tidb\_rowid构成，⽽主键本质上是唯⼀索引
      * 通过主键访问⾏记录时，不可以直接获取⾏记录，需要从额外存储的主键获取⾏的"\_tidb\_rowid",再通过此⾏的

\_tidb\_rowid获取⾏记录。⽐聚簇表多⼀次回表操作。

2、使⽤说明：

* + - * 业务尽量不要使⽤⾮聚簇表,多⼀次回表操作.

3、建表语句：

CREATE TABLE non\_t (a BIGINT PRIMARY KEY NONCLUSTERED, b VARCHAR(255));

CREATE TABLE non\_t1 (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) NONCLUSTERED);

使⽤可执⾏的注释语法指定⾮聚簇索引属性：

CREATE TABLE non\_t (a BIGINT PRIMARY KEY /\*T![clustered\_index] NONCLUSTERED \*/, b VARCHAR(255));

CREATE TABLE non\_t (a BIGINT, b VARCHAR(255), PRIMARY KEY(a, b) /\*T![clustered\_index] NONCLUSTERED \*/);

4、添加、删除⾮聚簇索引

* + - * ⽀持在建表之后添加或删除⾮聚簇索引

ALTER TABLE non\_t ADD PRIMARY KEY(b, a) NONCLUSTERED;

ALTER TABLE non\_t1 ADD PRIMARY KEY(b, a); -- 不指定关键字，则为⾮聚簇索引

ALTER TABLE non\_t DROP PRIMARY KEY; ALTER TABLE non\_t1 DROP INDEX `PRIMARY`;

判断是否为聚簇索引表

* 执⾏语句 SHOW CREATE TABLE。
* 执⾏语句 SHOW INDEX FROM。
* 查询系统表 information\_schema.tables 中的 TIDB\_PK\_TYPE 列。

mysql> show create table t\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Table: t

Create Table: CREATE TABLE `t` (

`a` bigint(20) NOT NULL,

`b` varchar(255) DEFAULT NULL,

PRIMARY KEY (`a`) /\*T![clustered\_index] CLUSTERED \*/

) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4\_bin

1 row in set (0.00 sec)

-- 通过 SHOW INDEX FROM 查看，Clustered ⼀列可能的结果为 Yes 或 No mysql> SHOW INDEX FROM t\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Table: t

Non\_unique: 0 Key\_name: PRIMARY

Seq\_in\_index: 1 Column\_name: a Collation: A Cardinality: 0

Sub\_part: NULL Packed: NULL

Null: Index\_type: BTREE

Comment: Index\_comment:

Visible: YES Expression: NULL

Clustered: YES

1 row in set (0.00 sec)

-- 查询 information\_schema.tables 系统表

mysql> SELECT TIDB\_PK\_TYPE FROM information\_schema.tables WHERE table\_schema = 'test' AND table\_name = 't';

+ +

| TIDB\_PK\_TYPE |

+ +

| CLUSTERED |

+ +

1 row in set (0.03 sec)

限制

1. 明确不⽀持且没有⽀持计划的使⽤限制：
   * 不⽀持与 SHARD\_ROW\_ID\_BITS ⼀起使⽤；PRE\_SPLIT\_REGIONS 在聚簇索引表上不⽣效。
   * 不⽀持对聚簇索引表进⾏降级。如需降级，请使⽤逻辑备份⼯具迁移数据。

2. 尚未⽀持，但未来有计划⽀持的使⽤限制：

- 尚未⽀持通过 ALTER TABLE 语句增加、删除、修改聚簇索引。

3. 特定版本的限制：

- 在 v5.0 版本中，聚簇索引不⽀持与 TiDB Binlog ⼀起使⽤。

### **⾮聚簇表之**shard\_row\_id\_bits**和**PRE\_SPLIT\_REGIONS

功能⽤途

#1、SHARD\_ROW\_ID\_BITS

（1）功能：

* + - * 对于NONCLUSTERED，tidb会使⽤⼀个隐式的rowid，造成写⼊热点
      * SHARD\_ROW\_ID\_BITS，通过调整隐式⽣成的\_tidb\_rowid的⾼位，从⽽把 rowid 打散写⼊多个不同的 Region

打散热点

* + - * + SHARD\_ROW\_ID\_BITS = 4 表示 16 个分⽚
        + SHARD\_ROW\_ID\_BITS = 6 表示 64 个分⽚
        + SHARD\_ROW\_ID\_BITS = 0 表示默认值 1 个分⽚

#（2）说明

⾼位:顺序主键变成不顺序主键插⼊,分散到多个region

1001-->0001

2002-->0002

3003-->0003

#（3）语句示例：

* + - * CREATE TABLE: CREATE TABLE t (c int) SHARD\_ROW\_ID\_BITS = 4;
      * ALTER TABLE: ALTER TABLE t SHARD\_ROW\_ID\_BITS = 4;

#2、pre\_split\_regions #（1）定义

* + - * 建表后会将表拆分成2^(PRE\_SPLIT\_REGIONS)个region
      * PRE\_SPLIT\_REGIONS 必须⼩于等于 SHARD\_ROW\_ID\_BITS

#（2）作⽤:

* + - * 减少region⽣成的成本

#（3）语句示例：

create table t2 (a int auto\_increment PRIMARY KEY NONCLUSTERED, b int) shard\_row\_id\_bits = 4 pre\_split\_regions=2;

查看表t2⽣成的region

mysql> show table t2 regions\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |
| --- | --- |
| REGION\_ID: | 1831624969 |
| START\_KEY: | t\_2281\_i\_1\_ |
| END\_KEY: | t\_2281\_r\_2305843009213693952 |
| LEADER\_ID: | 1831624970 |
| LEADER\_STORE\_ID: | 1831605622 |
| PEERS: | 1831624970, 1831624971, 1831624972 |
| SCATTERING: | 0 |
| WRITTEN\_BYTES: | 1623 |
| READ\_BYTES: | 0 |
| APPROXIMATE\_SIZE(MB): | 1 |

APPROXIMATE\_KEYS: 0

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |
| --- | --- |
| REGION\_ID: | 1831624973 |
| START\_KEY: | t\_2281\_r\_2305843009213693952 |
| END\_KEY: | t\_2281\_r\_4611686018427387904 |
| LEADER\_ID: | 1831624974 |
| LEADER\_STORE\_ID: | 1831605622 |
| PEERS: | 1831624974, 1831624975, 1831624976 |
| SCATTERING: | 0 |
| WRITTEN\_BYTES: | 0 |
| READ\_BYTES: | 0 |
| APPROXIMATE\_SIZE(MB): | 1 |
| APPROXIMATE\_KEYS: | 0 |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 3. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |  |
| --- | --- |
| REGION\_ID: | 1831624977 |
| START\_KEY: | t\_2281\_r\_4611686018427387904 |
| END\_KEY: | t\_2281\_r\_6917529027641081856 |
| LEADER\_ID: | 1831624978 |
| LEADER\_STORE\_ID: | 1831605622 |
| PEERS: | 1831624978, 1831624979, 1831624980 |
| SCATTERING: | 0 |
| WRITTEN\_BYTES: | 0 |
| READ\_BYTES: | 0 |
| APPROXIMATE\_SIZE(MB): | 1 |
| APPROXIMATE\_KEYS: | 0 |

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 4. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

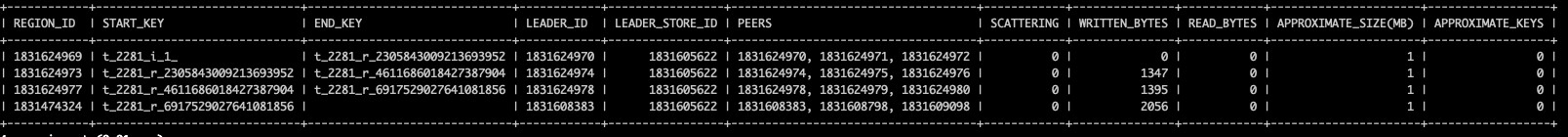
|  |  |  |
| --- | --- | --- |
| REGION\_ID:  START\_KEY: END\_KEY: | 1831474324  t\_2281\_r\_6917529027641081856 | |
| LEADER\_ID: | 1831608383 |  |
| LEADER\_STORE\_ID: | 1831605622 |  |
| PEERS: | 1831608383, | 1831608798, 1831609098 |
| SCATTERING: | 0 |  |
| WRITTEN\_BYTES: | 2056 |  |
| READ\_BYTES: | 0 |  |
| APPROXIMATE\_SIZE(MB): | 1 |  |
| APPROXIMATE\_KEYS: | 0 |  |

4 rows in set (0.01 sec)

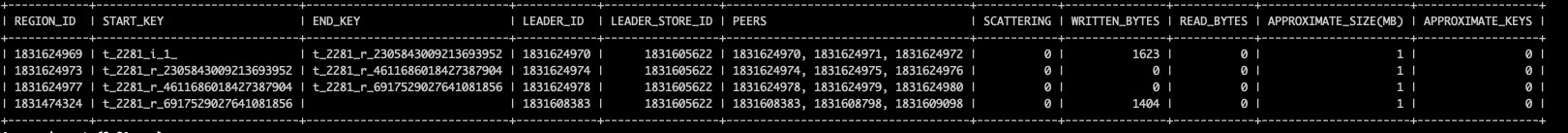
插⼊数据：

insert into t2(b) values(100); insert into t2(b) values(200); insert into t2(b) values(300); insert into t2(b) values(400); insert into t2(b) values(500); insert into t2(b) values(600); insert into t2(b) values(700); insert into t2(b) values(800); insert into t2(b) values(900);

insert into t2(b) values(1000);



插⼊10条数据后：



# 分区表

## 分区表

1、类型：

* + - * Range 分区
      * List 分区
      * list columns 分区
      * Hash 分区

2、作⽤：

* + - * range分区，list分区和List COLUMNS 分区可以⽤于解决业务员中⼤量删除带来的性能问题
      * hash分区则可以⽤于⼤量希尔场景下的数据打散
      * 分务表的每个唯⼀键，必须包含分区表达式中⽤到的所有列。

3、Range 分区

* + - * range分区范围取值,⽐如⽉份时间管理。
      * 可以基于 timestamp 列的值分区，并使⽤ unix\_timestamp() 函数

示例：

CREATE TABLE employees ( id INT NOT NULL, fname VARCHAR(30), lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01', separated DATE DEFAULT '9999-12-31',

job\_code INT,

store\_id INT NOT NULL) PARTITION BY RANGE (store\_id) (

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | PARTITION | p0 | VALUES | LESS | THAN | (6), |
| PARTITION | p1 | VALUES | LESS | THAN | (11), |
| PARTITION | p2 | VALUES | LESS | THAN | (16), |
| PARTITION | p3 | VALUES | LESS | THAN | (21), |
| PARTITION | p4 | VALUES | LESS | THAN | MAXVALUE |
| ); |  |  |  |  |  |  |

4、List 分区

* + - * 将 session 变量 tidb\_enable\_list\_partition 的值设置为 ON
      * 可通过 PARTITION ... VALUES IN (...) ⼦句对值进⾏定义

示例：

CREATE TABLE employees ( id INT NOT NULL,

hired DATE NOT NULL DEFAULT '1970-01-01',

store\_id INT)

PARTITION BY LIST (store\_id) (

PARTITION pNorth VALUES IN (1, 2, 3, 4, 5),

PARTITION pEast VALUES IN (6, 7, 8, 9, 10),

PARTITION pWest VALUES IN (11, 12, 13, 14, 15),

PARTITION pCentral VALUES IN (16, 17, 18, 19, 20)

);

5、list columns 分区

* + - * 是 List 分区的⼀种变体
      * 可以将多个列⽤作分区键
      * 可以将整数类型以外的数据类型的列⽤作分区列
      * 可以使⽤字符串类型、DATE 和 DATETIME 类型的列

示例：

CREATE TABLE employees\_1 ( id INT NOT NULL,

fname VARCHAR(30), lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01', separated DATE DEFAULT '9999-12-31',

job\_code INT, store\_id INT, city VARCHAR(15))

PARTITION BY LIST COLUMNS(city) (

PARTITION pRegion\_1 VALUES IN('LosAngeles', 'Seattle', 'Houston'), PARTITION pRegion\_2 VALUES IN('Chicago', 'Columbus', 'Boston'), PARTITION pRegion\_3 VALUES IN('NewYork', 'LongIsland', 'Baltimore'), PARTITION pRegion\_4 VALUES IN('Atlanta', 'Raleigh', 'Cincinnati')

);

6、Hash分区

* + - * 保证数据均匀地分散到⼀定数量的分区⾥⾯
      * 打散数据,避免热点

示例：

CREATE TABLE employees ( id INT NOT NULL, fname VARCHAR(30), lname VARCHAR(30),

hired DATE NOT NULL DEFAULT '1970-01-01', separated DATE DEFAULT '9999-12-31',

job\_code INT, store\_id INT)

PARTITION BY HASH(store\_id) PARTITIONS 4;

* + 1. **分区对于**NULL**值的处理**

TiDB 允许计算结果为 NULL 的分区表达式。注意，NULL 不是⼀个整数类型，NULL ⼩于所有的整数类型值，正如

ORDER BY 的规则⼀样。

1、Range 分区对 NULL 的处理

* + - * 分区列的计算结果是 NULL，则这⼀⾏会被插⼊到最⼩的那个分区

mysql> CREATE TABLE par\_t1 ( c1 INT,

c2 VARCHAR(20)) PARTITION BY RANGE(c1) (

PARTITION p0 VALUES LESS THAN (0), PARTITION p1 VALUES LESS THAN (10), PARTITION p2 VALUES LESS THAN MAXVALUE

);

mysql> insert into par\_t1 values(NULL,'aa'); mysql> select \* from par\_t1 partition(p0);

+------| +

| c1 | c2 |

+------| +

| NULL | aa |

+------| +

1 row in set (0.00 sec)

mysql> select \* from par\_t1 partition(p1); Empty set (0.00 sec)

mysql> select \* from par\_t1 partition(p2); Empty set (0.00 sec)

-- 删除 p0 后验证：

mysql> alter table par\_t1 drop partition p0; Query OK, 0 rows affected (0.08 sec)

mysql> select \* from t1; Empty set (0.00 sec)

2、Hash 分区对 NULL 的处理

* + - * 分区表达式的计算结果为 NULL，会被当作 0 值处理

mysql> CREATE TABLE th ( c1 INT,

c2 VARCHAR(20))

PARTITION BY HASH(c1) PARTITIONS 2;

mysql> INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan'); mysql> select \* from th partition (p0);

+------| +

| c1 | c2 |

+------| +

| NULL | mothra |

| 0 | gigan |

+------| +

2 rows in set (0.00 sec)

mysql> select \* from th partition (p1); Empty set (0.00 sec)

* + 1. **分区管理**
* 对于 LIST 和 RANGE 分区表,可以通过alter执⾏添加和删除分区的操作，hash分区不可以；
* Range 分区中，ADD PARTITION 只能在分区列表的最后⾯添加

ALTER TABLE members DROP PARTITION p2;

ALTER TABLE members truncate PARTITION p2;

ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));

## 分区裁剪：

* 不需要扫描那些匹配不上的分区
* 优化器可以通过 where 条件裁剪的两个场景：
  + partition\_column = constant
  + partition\_column IN (constant1, constant2, ..., constantN)
* 分区裁剪暂不⽀持 LIKE 语句
* 对于 Hash 分区类型，只有等值⽐较的查询条件能够⽀持分区裁剪。

示例：

CREATE TABLE t1 (

fname VARCHAR(50) NOT NULL, lname VARCHAR(50) NOT NULL,

region\_code TINYINT UNSIGNED NOT NULL, dob DATE NOT NULL)

PARTITION BY RANGE( region\_code ) ( PARTITION p0 VALUES LESS THAN (64), PARTITION p1 VALUES LESS THAN (128), PARTITION p2 VALUES LESS THAN (192), PARTITION p3 VALUES LESS THAN MAXVALUE

);

SELECT \* FROM t1 WHERE region\_code > 125 AND region\_code < 130;

# 数值类型

TiDB ⽀持除空间类型 (SPATIAL) 之外的所有 MySQL 数据类型，包括：

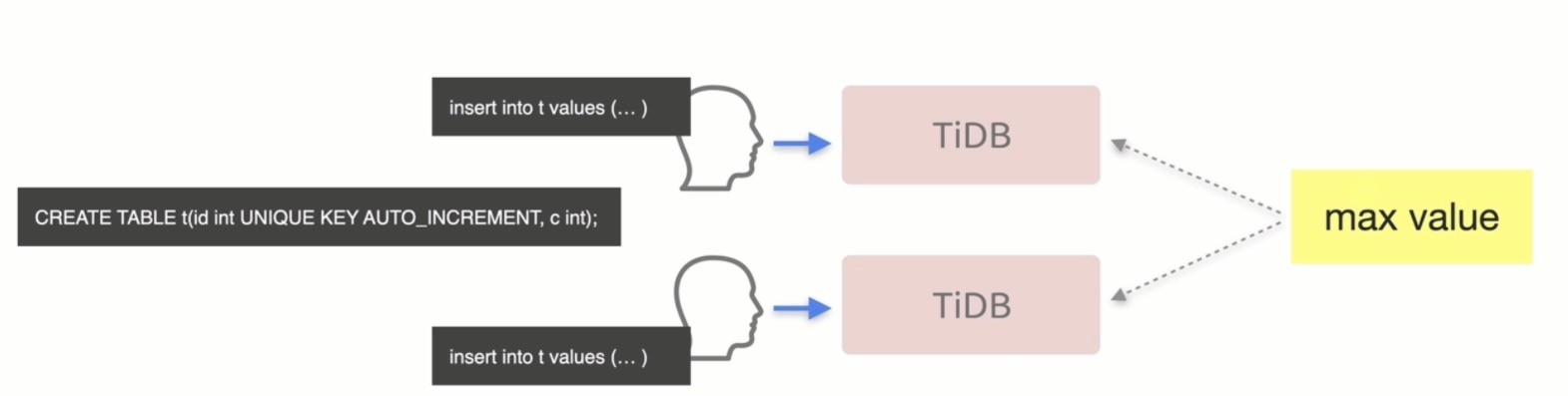
* 数值型类型
* 字符串类型
* 时间和⽇期类型
* JSON 类型

TiDB列的默认值：

* 数值型类型、绝⼤部分字符串类型列的默认值必须是常量
* 时间和⽇期类型，列的默认值可以使⽤函数作为默认值
* blob/text/json不可以设置默认值
  1. **⾃增**ID
     1. AUTO\_INCREMENT**基本概念**

1、基本概念：

* ⽤于⾃动填充缺省列值的列属性
* ⽀持显式指定列值的插⼊语句
  + 1. AUTO\_INCREMENT**实现原理**



* 对于每⼀个⾃增列，都使⽤⼀个全局可⻅的键值对⽤于记录当前已分配的最⼤ ID
* 为了避免写请求放⼤的问题，每个 TiDB 节点在分配 ID 时，都申请⼀段 ID 作为缓存
* 当前预分配的ID段使⽤完毕后，或TiDB重启，都会重新再次申请新的ID段。

两个⽤户插⼊两个tidb节点怎么分配⾃增ID? ⾼并发设计?

每个表存储⼀个最⼤值,分给TiDB server节点⼀段ID。

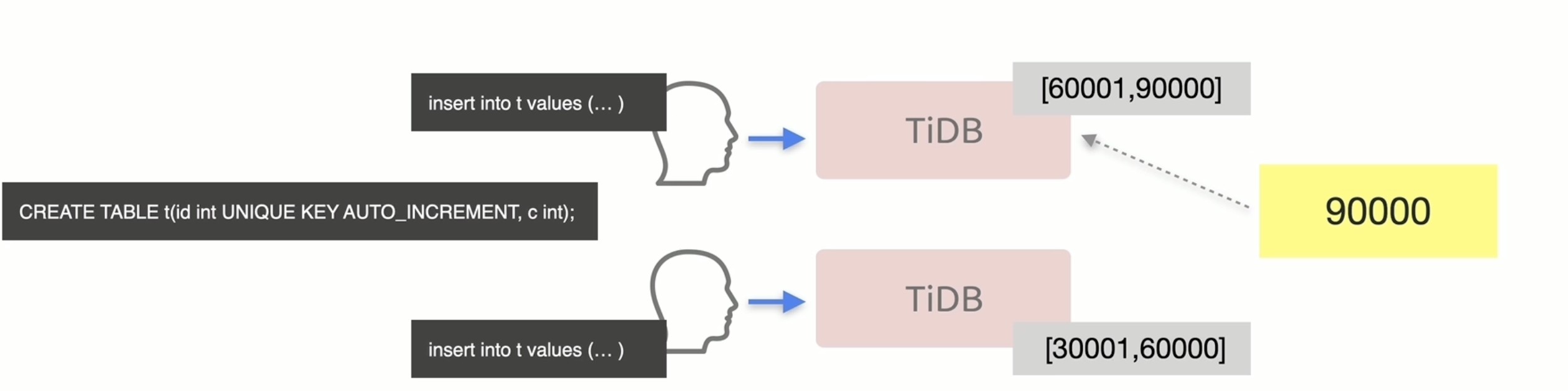
- 第⼀个⽤户[1,30000]

- 第⼆个分配[30001,60000]

第⼀tidb节点宕机

1. 重新分配id,[60001,90000]

保证了插⼊的绝对⽐上⼀次分配⼤,即单调递增



* + 1. AUTO\_INCREMENT**使⽤限制**

1. 定义的列必须为主键或者索引的⾸列。
2. 只能定义在类型为整数、FLOAT 或 DOUBLE 的列上。
3. 不⽀持与列的默认值 DEFAULT 同时指定在同⼀列上。
4. 不⽀持alter\_table加⼊auto\_increment
5. 删除需要设置变量 @@tidb\_allow\_remove\_auto\_inc移除
6. ⾃增ID不能作为TiDB最新值依据
   * 1. AUTO\_RANDOM**基础概念**

* AUTO\_RANDOM 是应⽤在 BIGINT 类型列的属性，⽤于列值的⾃动分配
* ⽤于解决⼤批量写⼊tidb 聚簇表 造成的 ⾃增主键列 的表热点问题

CREATE TABLE t (a BIGINT AUTO\_RANDOM, b VARCHAR(255), PRIMARY KEY (a)); CREATE TABLE t (a BIGINT PRIMARY KEY AUTO\_RANDOM, b VARCHAR(255));

CREATE TABLE t (a BIGINT AUTO\_RANDOM(6), b VARCHAR(255), PRIMARY KEY (a));

1. auto\_random实验
   * insert into t values(),();
   * 每次batch insert,两个batch之间是随机打散到不同region中,⼀个batch⾥⾯是连续的

在⽤户执⾏ INSERT 语句时：

* + - 如果语句中显式指定了 AUTO\_RANDOM 列的值，则该值会被正常插⼊到表中。
    - 如果语句中没有显式指定 AUTO\_RANDOM 列的值，TiDB 会⾃动⽣成⼀个随机的值插⼊到表中。

tidb> CREATE TABLE t (a BIGINT PRIMARY KEY AUTO\_RANDOM, b VARCHAR(255));

Query OK, 0 rows affected, 1 warning (0.01 sec)

tidb> INSERT INTO t(a, b) VALUES (1, 'string'); Query OK, 1 row affected (0.00 sec)

tidb> SELECT \* FROM t;

+---+ +

| a | b |

+---+ +

| 1 | string |

+---+ +

1 row in set (0.01 sec)

tidb> INSERT INTO t(b) VALUES ('string2');

Query OK, 1 row affected (0.00 sec)

tidb> INSERT INTO t(b) VALUES ('string3');

Query OK, 1 row affected (0.00 sec)

tidb> SELECT \* FROM t;

+---------------------+ +

| a | b |

+---------------------+ +

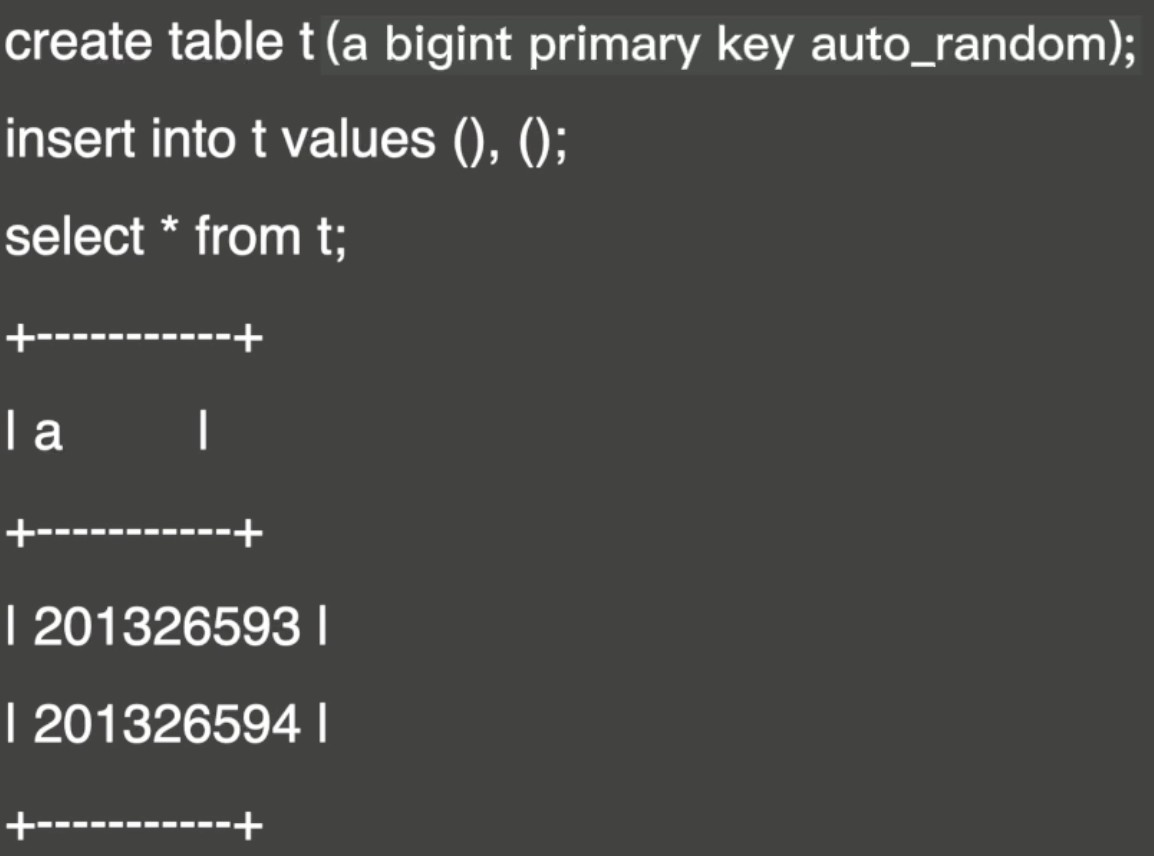
| 1 | string |

| 1152921504606846978 | string2 |

| 4899916394579099651 | string3 |

+---------------------+ +

3 rows in set (0.00 sec)



* + 1. AUTO\_RANDOM**实现原理**



- TiDB ⾃动分配的 AUTO\_RANDOM(S) 列值共有 64 位。

-> S 表示分⽚位的数量，取值范围是 1 到 15，默认为 5。若要使⽤不同⻓度的随机位，可以调整 AUTO\_RANDOM括号后⾯的数字

-> 符号位的⻓度由该列是否存在 UNSIGNED 属性决定：存在则为 0，否则为 1.

-> ⾃增位的值保存在存储引擎中，按顺序分配，每次分配完值后会⾃增 1。

* + 1. AUTO\_RANDOM**使⽤限制**
* AUTO\_RANDOM列类型只能是bigint类型
* 主键属性是nonclustered，即使是整型主键列，也不⽀持使⽤AUTO\_RANDOM
* 不建议⾃⾏显式指定含有 AUTO\_RANDOM 列的值。不恰当地显式赋值，可能会导致该表提前耗尽⽤于⾃动分配的数值。
* 不⽀持使⽤ ALTER TABLE 来修改 AUTO\_RANDOM 属性，包括添加或移除该属性。
* 不⽀持修改含有 AUTO\_RANDOM 属性的主键列的列类型。
* 不⽀持与 AUTO\_INCREMENT 同时指定在同⼀列上。
* 不⽀持与列的默认值 DEFAULT 同时指定在同⼀列上。
* AUTO\_RANDOM 列的数据很难迁移到 AUTO\_INCREMENT 列上，因为 AUTO\_RANDOM 列⾃动分配的值通常都很

⼤。

### auto\_increment**和**auto\_random**示例**

#1、创建⼀个auto\_increment的⾮聚簇表，并查看region分布

CREATE TABLE t\_11 (a BIGINT auto\_increment PRIMARY KEY NONCLUSTERED, b VARCHAR(255))

shard\_row\_id\_bits = 4 pre\_split\_regions=2;

>show table t\_11 regions\G

insert into t\_11(b) values('aa'); insert into t\_11(b) values('bb'); insert into t\_11(b) values('cc'); insert into t\_11(b) values('dd'); show table t\_11 regions;

#2、创建⼀个auto\_random的聚簇表，并查看region分布

|  |  |
| --- | --- |
| REGION\_ID:  START\_KEY: END\_KEY: | 1831474324  t\_2288\_ |
| LEADER\_ID: | 1831608383 |
| LEADER\_STORE\_ID: | 1831605622 |
| PEERS: | 1831608383, 1831608798, 1831609098 |
| SCATTERING: | 0 |
| WRITTEN\_BYTES: | 1585 |
| READ\_BYTES: | 0 |
| APPROXIMATE\_SIZE(MB): | 1 |
| APPROXIMATE\_KEYS: | 0 |

* 1. TiDB**数据与对象使⽤限****制**

CREATE TABLE t\_12 (a BIGINT PRIMARY KEY AUTO\_RANDOM, b VARCHAR(255));

>show table t\_12 regions\G

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\**\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*****\**

1 row in set (0.01 sec)

#将它的rergion进⾏⼿动split

> split table t\_12 between(0) and (9223372036854775808) regions 4;

+--------------------+ +

| TOTAL\_SPLIT\_REGION | SCATTER\_FINISH\_RATIO |

+--------------------+ +

| 3 | 1 |

+--------------------+ +

>show table t\_12 regions;

insert into t\_12(b) values ('a'),('b'),('c'),('d'),('e');

insert into t\_12(b) values ('aa'),('bb'),('cc'),('dd'),('ee'); show table t\_12 regions;

数据类型使⽤限制



对象限制



* 1. Schema**设计建****议** 
     1. **⾼兼容**schema**设计**



⾮聚簇表+shard\_row\_id\_bits

* + 1. **⾼性能**schema**设计**

聚簇表 + auto\_random



2**、数据库索引设****计**

索引有两种常⻅的类型，分别为：

* Primary Key: 即主键索引，即标识在主键列上的索引。
* Secondary Index: 即⼆级索引，即在⾮主键上标识的索引。
  1. TiDB**索引构成原****理** 
     1. **索引的**KV**映射原理**

\*\* 1、聚簇表的KV映射规则 \*\*

Key: tablePrefix{TableID}\_recordPrefixSep{Col1} #col1为cluster index

Value: [col2,col3,col4]

\*\* 2、⾮聚簇表KV的映射规则\*\*

Key: tablePrefix{TableID}\_recordPrefixSep{\_Tidb\_RowID}

Value: [col1,col2,col3,col4] #KV存储中Value存储真实的⾏数据

3、唯⼀索引&⾮聚簇表的主键 映射⽅式

Key: tablePrefix{TableID}\_indexPrefixSep{indexID}\_indexedColumnsValue Value: RowID

-> ⾮聚簇表,RowID是 \_TiDB\_RowID

-> 聚簇表,RowID是主键

4、⼆级⾮唯⼀索引 映射⽅式

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

Value: null

-> column value 不唯⼀

-> 加上{row\_id}作为唯⼀区分

-> value为null, tableID+indexID+rowid= 表+索引+⾏数直接找到数据

-> 如果⼆级索引是唯⼀索引,value存储主键索引

5、索引示例

* + - 1. 创建⾮聚簇表user

CREATE TABLE User (

ID int not null primary key NONClustered, Name varchar(20),

Role varchar(20), Age int,

KEY idxAge (Age)

);

* + - 1. 往表⾥插⼊ 3 ⾏数据：

insert into user values (100, "TiDB", "SQL Layer", 10); insert into user values (200, "TiKV", "KV Engine", 20); insert into user values (300, "PD", "Manager", 30);

(3)数据的KV映射关系

t10\_r1 --> [100,"TiDB", "SQL Layer", 10]

t10\_r2 --> [200,"TiKV", "KV Engine", 20]

t10\_r3 --> [300,"PD", "Manager", 30]

1. 主键的KV映射关系

t10\_i1\_100 --> 1

t10\_i1\_200 --> 2

t10\_i1\_300 --> 3

1. ⼆级索引的KV映射关系 t10\_i2\_10\_1 --> null t10\_i2\_20\_2 --> null t10\_i2\_30\_3 --> null

# 索引设计

## 索引的创建与映射关系

借鉴Percolate模型,⼀个事务写⼊同时写⼊primary+多个secondarykey索引创建

1、TiDB的索引创建语法与MySQL的索引创建语法保持兼容

（1）索引可以在新建表的同时创建

KEY `{index\_name}` (`{column\_names}`)

eg：

create table index\_t1(

id int not null primary key auto\_increment, c1 int not null,

c2 int not null, key idx\_c1(c1));

（2）索引创建⽀持alter table...add index语句在已有表中添加⼀个索引

-- 创建普通索引

alter table index\_t1 add index idx\_c2(c2);

-- 创建唯⼀索引

alter table index\_t1 add unique index uidx\_c2(id);

1. 添加索引是底层异步添加,不影响读写
   * 对业务⽆感,对你也⽆感。
   * 如果是⼤表 百亿 那么添加索引是个很痛苦的操作，要⼏天的时间.
2. 每次仅能⽀持⼀个DDL操作
3. 联合索引: 在搜索使⽤时候与mysql使⽤规则⼀致，遵循最左定值匹配原则
   * 映射：tableid\_rowid\_c1,c2,c3\_indexid
   * 联合索引(a,b,c)相当于创建了a,(a,b),(a,b,c)三个索引

-- 可以有效使⽤索引的场景

select \* from index\_t2 where a between 1 and 20;

select \* from index\_t2 where a=3 and b between 1 and 20; select \* from index\_t2 where a=3 and b=4 and between 1 and 20;

-- 部分使⽤索引的场景

select \* from index\_t2 where a between 1 and 20 and b=4; -- (a)

select \* from index\_t2 where a=3 and b between 1 and 20 and c=4; -- (a,b)

-- 不能使⽤索引的场景

select \* from index\_t2 where b=3 and c=2; select \* from index\_t2 where c=4;

1. 覆盖索引，提升性能减少回表

select c1, c2 from t where c1 > 10;，-- 要优化这个查询可以创建组合索引 Index c12 (c1, c2)。

6、示例：

-- 创建表t13

CREATE TABLE t13 (a BIGINT PRIMARY KEY CLUSTERED AUTO\_RANDOM, b VARCHAR(255));

-- 写⼊数据

insert into t13(b) values ('a'),('b'),('c'),('d'),('e');

insert into t13(b) values ('aa'),('bb'),('cc'),('dd'),('ee');

insert into t13(b) values ('s'),('d'),('f'),('g'),('h');

insert into t13(b) values ('z'),('x'),('c'),('v'),('b');

insert into t13(b) values ('eee'),('rrq'),('ttt'),('ggg'),('yj');

-- 将表t13的region⼿⼯打散

> split table t13 between(0) and (9223372036854775807) regions 16;

-- 查看表的region分布

show table t13 regions;

-- 创建普通⼆级索引idx\_b

alter table t13 add index idx\_d(b);

-- 查看表t13索引idx\_d的region分布

show table t13 index idx\_d regions;

-- 将索引idx\_d的region⼿⼯进⾏split

split table t13 index idx\_d between('a') and ('g') regions 4; show table t13 index idx\_d regions;

## 表达式索引

1、表达式索引

* 表达式索引是⼀种特殊的索引，能将索引建⽴于表达式上。
* 表达式索引不能为主键
* 表达式索引中的表达式不能包含以下内容：易变函数，例如 rand() 和 now()

相关的创建&删除语句：

CREATE INDEX idx1 ON t1 ((lower(col1)));

或者等价的语句：

ALTER TABLE t1 ADD INDEX idx1((lower(col1)));

还可以在建表的同时指定表达式索引：

CREATE TABLE t1(col1 char(10), col2 char(10), index((lower(col1))));

-- 删除表达式索引与删除普通索引的⽅法⼀致：

DROP INDEX idx1 ON t1;

2、查看表达式索引中包含的函数：

mysql> select @@tidb\_allow\_function\_for\_expression\_index;

+ +

| @@tidb\_allow\_function\_for\_expression\_index |

+ +

| lower, md5, reverse, upper, vitess\_hash |

+ +

1 row in set (0.00 sec)

3、哪些语句可以使⽤到表达式索引？ SELECT lower(col1) FROM t;

当过滤的条件中有相同的表达式时，可以使⽤表达式索引。例如：

SELECT \* FROM t WHERE lower(col1) = "a"; SELECT \* FROM t WHERE lower(col1) > "a";

SELECT \* FROM t WHERE lower(col1) BETWEEN "a" AND "b";

SELECT \* FROM t WHERE lower(col1) in ("a", "b");

SELECT \* FROM t WHERE lower(col1) > "a" AND lower(col1) < "b";

当查询按照相同的表达式进⾏排序时，可以使⽤表达式索引。例如：

SELECT \* FROM t ORDER BY lower(col1);

当聚合函数或者 GROUP BY 中包含相同的表达式时，可以使⽤表达式索引

SELECT max(lower(col1)) FROM t；

SELECT min(col1) FROM t GROUP BY lower(col1);

4、查看表达式索引对应的表达式

show index 或查看系统表

mysql> show indexes from t1\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* Table: t1

Non\_unique: 1

Key\_name: expression\_index Seq\_in\_index: 1

Column\_name: NULL Collation: A

Cardinality: 0 Sub\_part: NULL

Packed: NULL Null: YES

Index\_type: BTREE Comment:

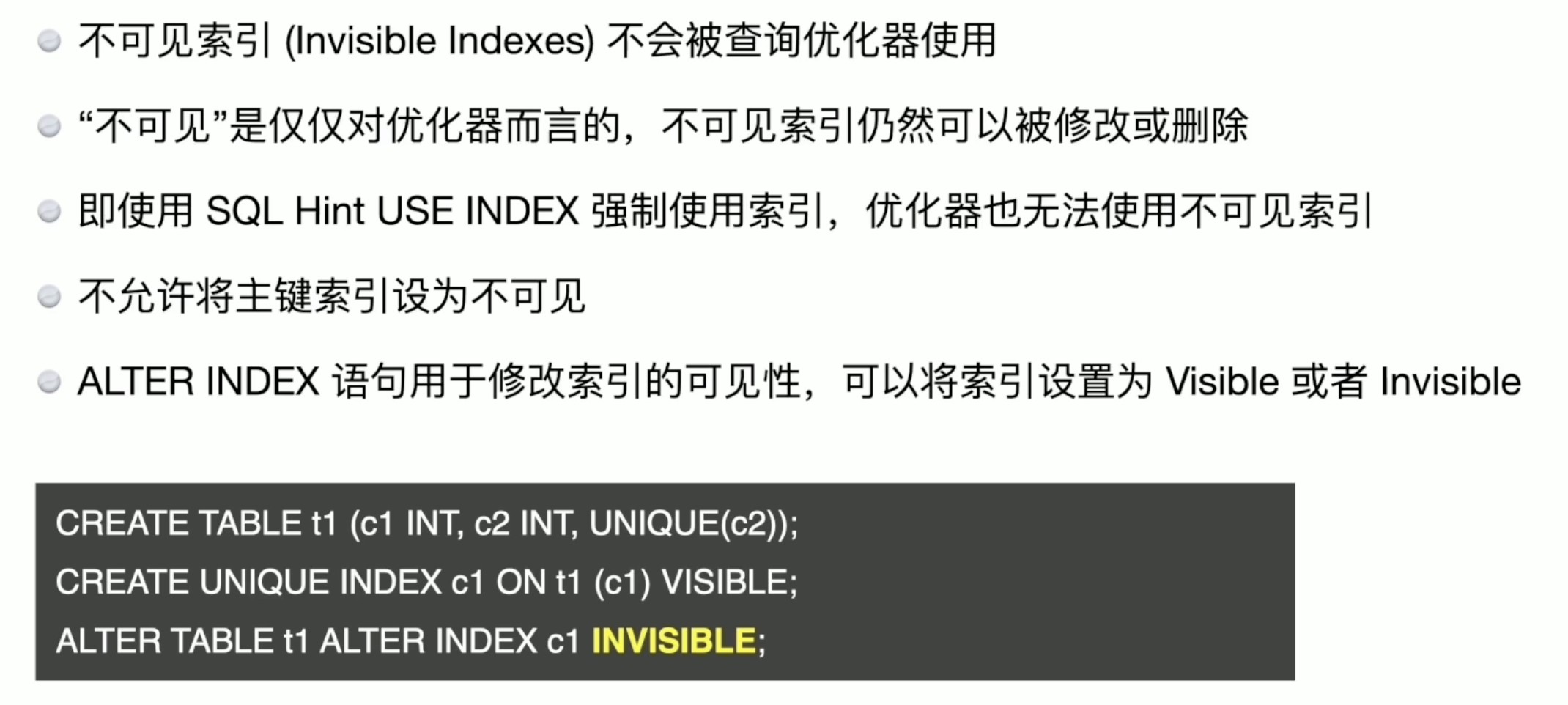
Index\_comment:

Visible: YES Expression: lower(`col1`)

Clustered: NO

1 row in set (0.00 sec)

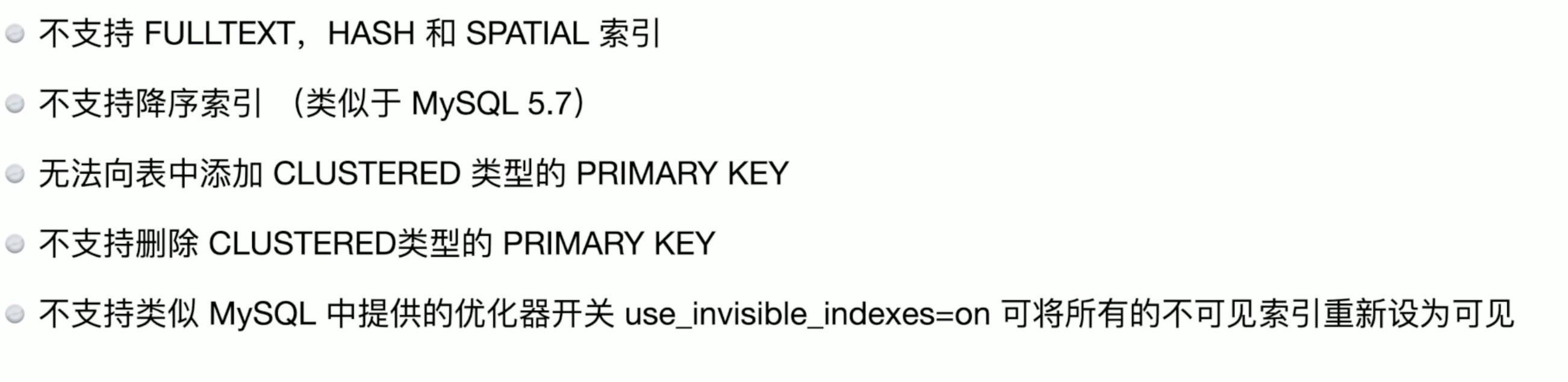
* + 1. **不可⻅索引**



与mysql8.0⼀样,⽤来监测是否索引有⽤,先隐藏-->确认业务⽆⽤-->drop index

# 索引最佳实践

与mysql兼容性



索引的最佳实践

1、创建索引的最佳实践

1. 建⽴你需要使⽤的数据的所有列的组合索引，利⽤覆盖索引优化
2. 避免创建不需要的⼆级索引
3. 根据具体的业务特点创建合适的索引。
   * 区分度⽐较⼤的列，通过索引能显著地减少过滤后的⾏数。
   * 有多个查询条件时，可以选择组合索引，注意需要把等值条件的列放在组合索引的前⾯。

（4）使⽤有意义的⼆级索引名

2、使⽤索引的最佳实践

1. 删除⽆⽤索引
2. 使⽤组合索引时，需要满⾜最左前缀原则。
3. 在查询条件中使⽤索引列作为条件时，不要在索引列上做计算，函数，或者类型转换的操作，会导致优化器⽆法使

⽤该索引。

1. 尽量使⽤覆盖索引，即索引列包含查询列，避免总是 SELECT \* 查询所有列的语句。
2. 查询条件使⽤ !=，NOT IN 时，⽆法使⽤索引。
3. 使⽤ LIKE 时如果条件是以通配符 % 开头，也⽆法使⽤索引。
4. 有多个索引可供选择时，推荐使⽤Hint
5. 查询条件使⽤ IN 表达式时，后⾯匹配的条件数量建议不要超过 300 个，否则执⾏效率会较差。

查看索引的region分布情况

#功能

1.通过知道 index region分布,知道tidb写⼊读取的写⼊是否有热点

#查看表table\_a所有索引分布情况

show table schema\_a.table\_a regions \G

#查看表table\_a某个索引分布式

show table schema\_a.table\_a index idx\_1 regions \G mysql> show table t2 index idx1 regions\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* REGION\_ID: 1159

START\_KEY:

END\_KEY: t\_74\_r\_6917529027641081856 LEADER\_ID: 1161

LEADER\_STORE\_ID: 7

|  |  |
| --- | --- |
| PEERS: | 1160, 1161, 1162 |
| SCATTERING: | 0 |
| WRITTEN\_BYTES: | 1968 |
| READ\_BYTES: | 133220 |
| APPROXIMATE\_SIZE(MB): | 1 |
| APPROXIMATE\_KEYS: | 1677 |
| SCHEDULING\_CONSTRAINTS: |  |
| SCHEDULING\_STATE: |  |
| 1 row in set (0.01 sec) |  |

3**、**TiDB**数据库库系统表使****⽤**

* 1. TiDB**系统表介绍**

1.mysql存储tidb系统表

- 系统表 mysql.users等

1. information\_schema提供了⼀种查看系统元数据的ANSI标准⽅法
   * 与mysql兼容的表: TABLES、processlist、columns等
   * TiDB⾃定义的表: CLUSTER\_CONFIG、cluster\_hardware、tiflash\_replica等

3. Metrics\_schema

- promethus监控指标视图

4.PERFORMANCE\_SCHEMA

- 兼容mysql,保留⼤部分视图

### mysql**库**

1. mysql.user：账户权限
2. mysql.db：库权限
3. mysql.tables\_priv
4. mysql.GLOBAL\_VARIABLES(⾮常重要)
   * TiDB全局系统变量
   * ⽀持对系统表GLOBAL\_VARIABLES 修改
5. mysql.tidb(⾮常重要)
   * 可直接修改表达到修改参数
   * 15个值,存放tidb特定相关，

- tikv\_gc\_life\_time ⾮常重要,gc时间,默认10min

* + 1. Information\_schema
       1. cluster\_info
          - 集群节点节本信息,启动时间运⾏时间,版本等

+----------------+-------------+------+------+---------+ +

| Field | Type | Null | Key | Default | Extra |

+----------------+-------------+------+------+---------+ +

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | TYPE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | INSTANCE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | STATUS\_ADDRESS | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | VERSION | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | GIT\_HASH | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | START\_TIME | | | varchar(32) | | | YES | | | | | NULL | | | | |
| | | UPTIME | | | varchar(32) | | | YES | | | | | NULL | | | | |

+----------------+-------------+------+------+---------+ +

7 rows in set (0.00 sec)

* + - 1. cluster\_config(⾮常重要)
         * 所有组件配置
         * 组成

mysql> DESC cluster\_config;

+----------+--------------+------+------+---------+ +

| Field | Type | Null | Key | Default | Extra |

+----------+--------------+------+------+---------+ +

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | TYPE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | INSTANCE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | KEY | | | varchar(256) | | | YES | | | | | NULL | | | | |
| | | VALUE | | | varchar(128) | | | YES | | | | | NULL | | | | |

+----------+--------------+------+------+---------+ +

* + - 1. DDL\_JOBS(⾮常重要)
         * 监控ddl
         * admin ddl jobs 语法查询此表

mysql> desc DDL\_JOBS;

+--------------+-------------+------+------+---------+ +

| Field | Type | Null | Key | Default | Extra |

+--------------+-------------+------+------+---------+ +

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | JOB\_ID | | | bigint(21) | | | YES | | | | | NULL | | | | |
| | | DB\_NAME | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | TABLE\_NAME | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | JOB\_TYPE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | SCHEMA\_STATE | | | varchar(64) | | | YES | | | | | NULL | | | | |
| | | SCHEMA\_ID | | | bigint(21) | | | YES | | | | | NULL | | | | |
| | | TABLE\_ID | | | bigint(21) | | | YES | | | | | NULL | | | | |
| | | ROW\_COUNT | | | bigint(21) | | | YES | | | | | NULL | | | | |
| | | CREATE\_TIME | | | datetime | | | YES | | | | | NULL | | | | |
| | | START\_TIME | | | datetime | | | YES | | | | | NULL | | | | |
| | | END\_TIME | | | datetime | | | YES | | | | | NULL | | | | |

| STATE | varchar(64) | YES | | NULL | |

| QUERY | varchar(64) | YES | | NULL | |

+--------------+-------------+------+------+---------+ +

13 rows in set (0.01 sec)

mysql> SELECT \* FROM ddl\_jobs LIMIT 3\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* JOB\_ID: 44

DB\_NAME: mysql

TABLE\_NAME: opt\_rule\_blacklist JOB\_TYPE: create table

SCHEMA\_STATE: public SCHEMA\_ID: 3

TABLE\_ID: 43

ROW\_COUNT: 0

START\_TIME: 2020-07-06 15:24:27

END\_TIME: 2020-07-06 15:24:27

STATE: synced

QUERY: CREATE TABLE IF NOT EXISTS mysql.opt\_rule\_blacklist ( name char(100) NOT NULL

);

* + - 1. CLUSTER\_SLOW\_QUERY
         * 集群慢⽇志
      2. SLOW\_QUERY
         * tidb节点慢⽇志
      3. CLUSTER\_PROCESSLIST
         * 集群连接统计
      4. TABLES
         * 与mysql⼀样表信息
      5. TIDB\_HOT\_REGIONS
         * 读写热点信息

#Tiflash

* + - 1. TIFLASH\_REPLICA
         * 提供了有关可⽤的 TiFlash 副本的信息。
      2. TIFLASH\_TABLES
  1. **系统表**ops**实际使****⽤** 
     1. **慢**SQL**应⽤**-**初步了解**
        1. 搜索某个⽤户的topN慢SQL

select query\_time, query, user from information\_schema.slow\_query

where is\_internal = false -- 排除 TiDB 内部的慢查询 SQL

and user = "test" -- 查找的⽤户名

order by query\_time desc limit 2;

* + - 1. 根据SQL指纹搜索同类慢查询

（1）先根据最近慢sql list查找特定的慢sql语句

select query\_time, query, digest from information\_schema.slow\_query where is\_internal = false

and time between '2022-10-01' and '2022-11-01' order by query\_time desc

limit 1;

(2)再根据digest确定同类型的慢查询 select query\_time, query

from information\_schema.slow\_query where digest='xxxxxx';

* + - 1. 执⾏计划变化,同类型sql适⽤

select count(distinct plan\_digest) as count, digest,

min(query)

from cluster\_slow\_query group by digest

having count > 1 limit 3;

- 找到慢SQL

select min(plan),

plan\_digest

from cluster\_slow\_query

where digest='043d1b737d6b73270458af551d8932327b2c4c3683bbe2d7659ef7243eae460a' group by plan\_digest\G

3.3.2 **热点表查询**

#统计当前读写热点表

select db\_name,table\_name,index\_name,

type, -- 读写热点分类

sum(FLOW\_BYTES), -- 每分钟流量

count(1), group\_concat(h.region\_id), count(DISTINCT p.store\_id),

group\_concat(p.store\_id) from

TIDB\_HOT\_REGIONS h JOIN

TIKV\_REGION\_PEERS p on h.region\_id = p.region\_id and p.IS\_LEADER=1

group by db\_name,table\_name,index\_name,type;

#统计当前读写热点store

select

sum(FLOW\_BYTES), -- 每分钟流量

count(1) -- region数量

from

TIDB\_HOT\_REGIONS h JOIN

TIKV\_REGION\_PEERS p on h.region\_id = p.region\_id and p.IS\_LEADER=1

group by p.store\_id order by count(1) desc;

3.3.3 SQL**阻塞查询**

v5.2.0.可⽤,简化阻塞查询

* DATA\_LOCK\_WAITS
  + 展示了集群中所有 TiKV 节点上当前正在发⽣的悲观锁等锁的情况。
  + 仅拥有process权限的⽤户可以查询
  + 从所有tikv节点实时获取
  + 如果集群规模很⼤、负载很⾼，查询该表有造成性能抖动的潜在⻛险

mysql> DESC data\_lock\_waits;

+------------------------+---------------------+------+------+---------+ +

| Field | Type | Null | Key | Default | Extra |

+------------------------+---------------------+------+------+---------+ +

+------------------------+---------------------+------+------+---------+ +

* DEADLOCKS
  + 提供当前 TiDB 节点上最近发⽣的若⼲次死锁错误的信息。
  + 默认容纳最近10次死锁错误的信息
  + 仅拥有 PROCESS 权限的⽤户可以查询该表。
* TIDB\_TRX
  + 提供了当前 TiDB 节点上正在执⾏的事务的信息。
  + 仅拥有 PROCESS 权限的⽤户可以获取该表中的完整信息。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | | KEY | | | text |  | | | NO | | | | | NULL | | | | |
| | | KEY\_INFO | | | text |  | | | YES | | | | | NULL | | | | |
| | | TRX\_ID | | | bigint(21) | unsigned | | | NO | | | | | NULL | | | | |
| | | CURRENT\_HOLDING\_TRX\_ID | | | bigint(21) | unsigned | | | NO | | | | | NULL | | | | |
| | | SQL\_DIGEST | | | varchar(64) |  | | | YES | | | | | NULL | | | | |
| | | SQL\_DIGEST\_TEXT | | | text |  | | | YES | | | | | NULL | | | | |

死锁问题排查

|  |  |  |
| --- | --- | --- |
| **事务**1 | **事务**2 | **说明** |
| Begin; | begin; |  |
| update deadlock\_t set b=11 where a=1; |  |  |
|  | update deadlock\_t set b=44 where a=2; |  |
| update deadlock\_t set b=33 where a=2; |  | 事务1阻塞 |
|  | update deadlock\_t set b=55 where a=1; | 事务2报出死锁错误: ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction |

#死锁示例

create table deadlock\_t (a int,b int); insert into deadlock\_t values(1,1),(2,2); session A:

begin

update deadlock\_t set b=11 where a=1; update deadlock\_t set b=33 where a=2;

session B:

begin

update deadlock\_t set b=44 where a=2; update deadlock\_t set b=55 where a=1;

ERROR 1213 (40001): Deadlock found when trying to get lock; try restarting transaction

mysql> select \* from information\_schema.DEADLOCKS\G

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 1. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DEADLOCK\_ID: 1

OCCUR\_TIME: 2022-11-13 23:16:45.384561

RETRYABLE: 0

TRY\_LOCK\_TRX\_ID: 437348606740529154 CURRENT\_SQL\_DIGEST:

c5c3eb2e6cd596f15a5d27546e46ac362d0941d642bc28ea2b4577d38b81b9e7 CURRENT\_SQL\_DIGEST\_TEXT: update `deadlock\_t` set `b` = ? where `a` = ?

KEY: 7480000000000000535F728000000000000002

KEY\_INFO:

{"db\_name":"test","table\_name":"deadlock\_t","handle\_type":"int","handle\_value":"2","db\_ id":2,"table\_id":83}

TRX\_HOLDING\_LOCK: 437348610201092097

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* 2. row \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* DEADLOCK\_ID: 1

OCCUR\_TIME: 2022-11-13 23:16:45.384561

RETRYABLE: 0

TRY\_LOCK\_TRX\_ID: 437348610201092097 CURRENT\_SQL\_DIGEST:

c5c3eb2e6cd596f15a5d27546e46ac362d0941d642bc28ea2b4577d38b81b9e7 CURRENT\_SQL\_DIGEST\_TEXT: update `deadlock\_t` set `b` = ? where `a` = ?

KEY: 7480000000000000535F728000000000000001 KEY\_INFO:

{"db\_name":"test","table\_name":"deadlock\_t","handle\_type":"int","handle\_value":"1","db\_ id":2,"table\_id":83}

TRX\_HOLDING\_LOCK: 437348606740529154

2 rows in set (0.00 sec)

3.2.4 **其余**mysql**兼容查询**

1. 库表⼤⼩

SELECT TABLE\_SCHEMA,table\_name, table\_rows, (data\_length + index\_length) as num, concat( round( ( data\_length + index\_length ) / 1024 / 1024/1024, 2 ), 'GB' ) DATA

FROM information\_schema.TABLES order by num desc limit 50;

1. 连接数

...