4
2
2
0

一文概述TiDB中的索引类型

数据源的TiDB学习之路 发表于 **2024-04-02**

原创 # 管理与运维

在关系型数据库中，索引是一种非常常用的数据结构，它通过缩小数据扫描的范围来提升查询时间和减少资源消耗。TiDB数据库中支持多种类型的索引，本文简要总结TiDB中的索引类型。

从大类上来划分，TiDB的索引可以分为**主键索引**和**二级索引**。主键索引就是在主键字段上建立的索引，二级索引是在非主键字段上建立的索引，下面通过示例简要描述二者。

主键索引

TiDB中的主键索引有两种，即**聚簇索引**和**非聚簇索引**。

聚簇索引

聚簇索引在有些数据库中也称为**索引组织表**，简单理解就是说表的数据在存储的时候就是按照主键字段进行排序存储的。聚簇索引是TiDB v5.0开始支持的特性，由于**表按主键排序**，因此不需要额外增加索引结构就可以实现按主键字段进行快速高效的查询和过滤。

较新的TiDB版本中默认创建的主键索引即为聚簇索引，这通过查看表结构可以看到，如下图所示中的**/*T![clustered_index] CLUSTERED */**代表创建出来的是聚簇索引。

```
mysql> create table t_cluster(a int not null primary key, b varchar(10));
Query OK, 0 rows affected (0.53 sec)

mysql> show create table t_cluster;
+-----------+-----------------------------------------------------------------------------------------------------------
------+
| Table     | Create Table
      |
+-----------+-----------------------------------------------------------------------------------------------------------
------+
| t_cluster | CREATE TABLE `t_cluster` (
  `a` int(11) NOT NULL,
  `b` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-----------+-----------------------------------------------------------------------------------------------------------
------+
1 row in set (0.01 sec)
```

非聚簇索引

非聚簇索引是通过**额外的索引条目**来实现数据的排序，表本身的数据存储并不是按主键字段排序的。在较新的TiDB版本中，如果想定义一个主键为非聚簇索引，需要显式在建表语句中定义**NONCLUSTERED**关键字，如下图所示。

```
mysql> create table t_noncluster(a int not null primary key nonclustered, b varchar(10));
Query OK, 0 rows affected (0.54 sec)

mysql> show create table t_noncluster;
+--------------+-----------------------------------------------------------------------------------------------------------
--------------+
| Table        | Create Table
               |
+--------------+-----------------------------------------------------------------------------------------------------------
--------------+
| t_noncluster | CREATE TABLE `t_noncluster` (
  `a` int(11) NOT NULL,
  `b` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] NONCLUSTERED */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--------------+-----------------------------------------------------------------------------------------------------------
--------------+
1 row in set (0.00 sec)
```

TiDB底层存储是KV结构，在聚簇索引中TiDB是把主键字段直接映射到Key中，但非聚簇索引中Key是由TiDB内部隐式分配的**_tidb_rowid**构成，主键本质是唯一索引。从下图我们也可以发现，非聚簇索引表中可以查询到_tidb_rowid这个隐含字段，而聚簇索引表中则不存在这个隐含字段。

```
mysql> select *,_tidb_rowid from t_noncluster limit 5;
+---+------+-------------+
| a | b    | _tidb_rowid |
+---+------+-------------+
| 1 | Z    |           1 |
| 2 | Y    |           2 |
| 3 | V    |           3 |
| 4 | G    |           4 |
| 5 | V    |           5 |
+---+------+-------------+
5 rows in set (0.00 sec)

mysql> select *,_tidb_rowid from t_cluster limit 5;
ERROR 1054 (42S22): Unknown column '_tidb_rowid' in 'field list'
```

聚簇与非聚簇索引的对比

聚簇索引的优势

既然TiDB在v5.0版本引入聚簇索引并设置为默认选项，那么聚簇索引肯定有其独特的优势，包括：

1. 插入数据时，减少一次索引数据写入
2. 等值查询时，减少一次索引数据读取
3. 范围查询时，减少多次索引数据读取

简单来说，使用聚簇索引，**在大部分场景下，写入和查询时都有一定的性能提升。**

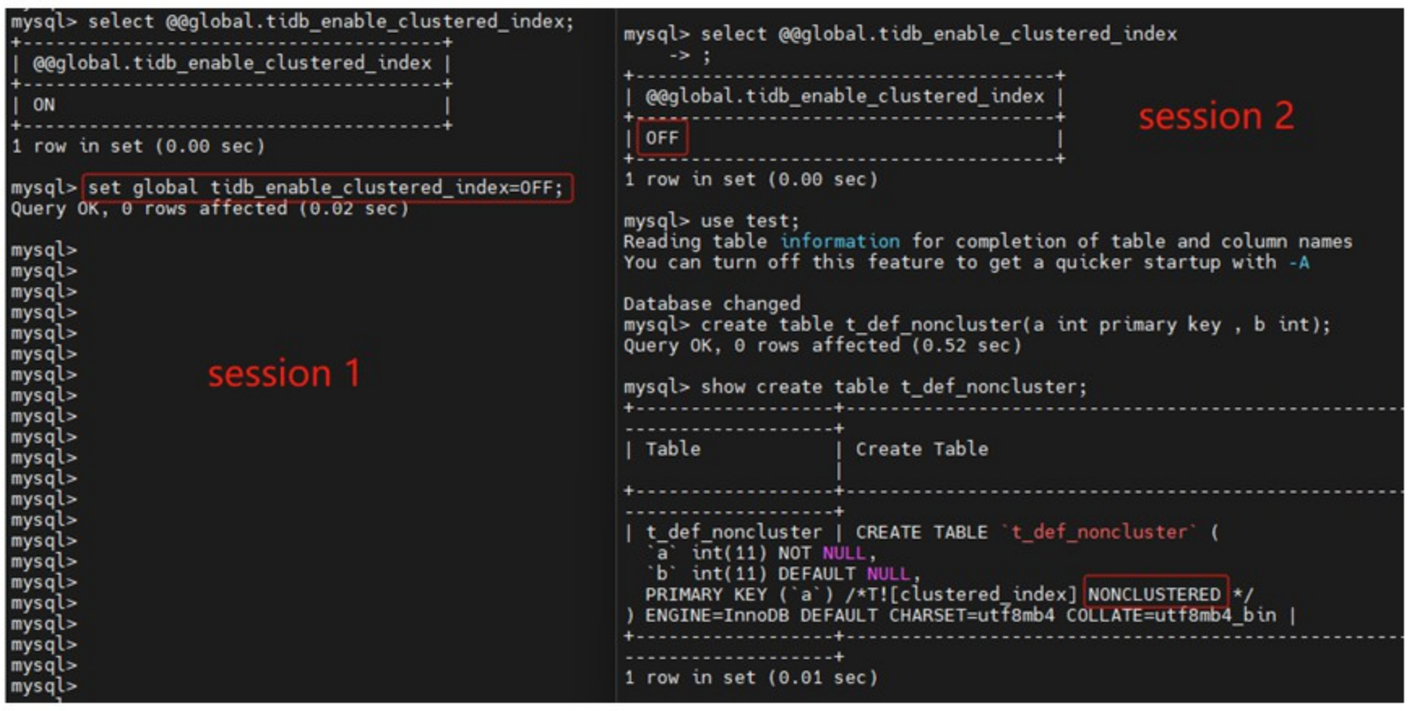
聚簇索引的劣势

1. 当批量插入大量取值相邻的主键时，可能产生表上较大的**写热点**问题
2. 当查询只需要获取少量字段时，由于要把整行数据取出可能导致性能反而不如非聚簇
3. 不支持在建表后添加或删除聚簇索引

聚簇与非聚簇的默认行为控制

前面提到在当前的TiDB版本中，主键默认创建为聚簇索引，这是由参数**tidb_enable_clustered_index**来控制的，ON表示所有主键默认创建为聚簇索引，OFF则表示默认创建为非聚簇索引。

因此，如果想针对单张表创建为非聚簇索引，我们使用上述所说的在表定义中添加NONCLUSTERED关键字。如果想全局控制这个行为，可以直接修改tidb_enable_clustered_index这个系统参数。比如说，我们设置这个变量的global范围为OFF，那后续创建的所有主键表都是非聚簇索引。



正如上面截图所示，会话1中将变量的global范围设置为OFF，在会话2中创建一个有主键的表，查看表结构发现默认创建为NONCLUSTERED属性。

二级索引

二级索引是在非主键字段上创建的索引，二级索引可以在建表语句中一起创建，也可以在事后单独创建二级索引。

建表时创建

在建表语句中，可以通过 **KEY index_name (column_name,...)** 或者 **INDEX index_name (column_name,...)** 来定义索引，两者含义相同。

单独事后创建

通过标准的CREATE INDEX语法创建，示例如下：

```
mysql> create table af_index(a int primary key, b int, c int);
Query OK, 0 rows affected (0.52 sec)

mysql> create index idx1 on af_index (b);
Query OK, 0 rows affected (1.01 sec)

mysql> create index idx2 on af_index (c);
Query OK, 0 rows affected (1.02 sec)

mysql> show create table af_index;
+----------+--------------------------------------------------------------------------------
------------------------------------------------------------+
| Table    | Create Table
                                                             |
+----------+--------------------------------------------------------------------------------
------------------------------------------------------------+
| af_index | CREATE TABLE `af_index` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  `c` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */,
  KEY `idx1` (`b`),
  KEY `idx2` (`c`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+----------+--------------------------------------------------------------------------------
------------------------------------------------------------+
1 row in set (0.00 sec)
```

主键索引和二级索引是从选取哪个字段的维度来区分的，TiDB也支持一些特殊场景使用的索引，如唯一索引、表达式索引、多值索引等。

唯一索引

唯一索引在功能上等同于**普通索引+唯一性约束**，使用**CREATE UNIQUE INDEX**创建。如果一个字段被定义为唯一索引，那么这个字段中的每个值必须要唯一才行，对于有冲突的数据是无法被写入的。

```
mysql> create unique index t_uniq_idx on t_uniq(b);
Query OK, 0 rows affected (1.02 sec)

mysql> show create table t_uniq;
+--------+--------------------------------------------------------------------------------
-----------------------------+
| Table  | Create Table
                             |
+--------+--------------------------------------------------------------------------------
-----------------------------+
| t_uniq | CREATE TABLE `t_uniq` (
  `a` int(11) NOT NULL,
  `b` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */,
  UNIQUE KEY `t_uniq_idx` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+--------+--------------------------------------------------------------------------------
-----------------------------+
1 row in set (0.00 sec)

mysql> insert into t_uniq values(1,'ABC');
Query OK, 1 row affected (0.00 sec)

mysql> insert into t_uniq values(2,'ABC');
ERROR 1062 (23000): Duplicate entry 'ABC' for key 't_uniq.t_uniq_idx'
mysql>
```

上述示例中，当b字段添加了唯一索引之后，插入两条相同的值时第2条插入语句报唯一性错误。

表达式索引

在数据库查询过滤时，有些时候我们不是直接基于值本身来过滤，而是基于值转换后的过滤，比如说要查询所有值为'abc'的数据，但并不区别大小写，'ABC'、'Abc'等都满足条件。查询语句大概会写成SELECT ... FROM .. WHERE lower(col1)='abc'，但如果我们在col1上建立一个普通的索引，这种情况下执行计划是无法选择走索引扫描的。

因此，我们需要创建一个表达式索引，有了表达式索引，上述语句便可以走索引扫描。

```
mysql> create index idx_lower on t_functbl((lower(b)));
Query OK, 0 rows affected (3.02 sec)

mysql> explain select * from t_functbl where lower(b)='abc';
+-------------------------------+---------+-----------+--------------------------------------------------------+----------------------------------------------+
| id                            | estRows | task      | access object                                          | operator info                                |
+-------------------------------+---------+-----------+--------------------------------------------------------+----------------------------------------------+
| Projection_4                  | 0.00    | root      |                                                        | test.t_functbl.a, test.t_functbl.b           |
| └─IndexLookUp_10              | 0.00    | root      |                                                        |                                              |
|   ├─IndexRangeScan_8(Build)   | 0.00    | cop[tikv] | table:t_functbl, index:idx_lower(lower(`b`))           | range:["abc","abc"], keep order:false        |
|   └─TableRowIDScan_9(Probe)   | 0.00    | cop[tikv] | table:t_functbl                                        | keep order:false                             |
+-------------------------------+---------+-----------+--------------------------------------------------------+----------------------------------------------+
4 rows in set (0.01 sec)

mysql> show create table t_functbl;
+-----------+--------------------------------------------------------------------------------
-----------------------------------------+
| Table     | Create Table
                                         |
+-----------+--------------------------------------------------------------------------------
-----------------------------------------+
| t_functbl | CREATE TABLE `t_functbl` (
  `a` int(11) NOT NULL,
  `b` varchar(10) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */,
  KEY `idx_lower` ((lower(`b`)))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-----------+--------------------------------------------------------------------------------
-----------------------------------------+
1 row in set (0.00 sec)
```

*注意：在创建表达式索引时，在表达式外面需要多写一层括号，如上图所示，create index idx_lower on t_functbl(**(lower(b))**);如果这里少一层括号会报语法错误。*

组合索引

所谓组合索引，就是说索引里面由多个字段组合而成，因为有些时候我们需要根据多个维度来筛选我们想要的数据，比如会按照姓名和年龄两个条件来找到某个人的信息。

```
mysql> create table t_mix(id int primary key, name varchar(20), age int);
Query OK, 0 rows affected (0.52 sec)

mysql> create index idx_mix on t_mix(name, age);
Query OK, 0 rows affected (1.01 sec)

mysql> show create table t_mix;
+-------+------------------------------------------------------------------------------------------
---------------------------------------------------------------------+
| Table | Create Table
                                                                     |
+-------+------------------------------------------------------------------------------------------
---------------------------------------------------------------------+
| t_mix | CREATE TABLE `t_mix` (
  `id` int(11) NOT NULL,
  `name` varchar(20) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */,
  KEY `idx_mix` (`name`,`age`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-------+------------------------------------------------------------------------------------------
---------------------------------------------------------------------+
1 row in set (0.00 sec)
```

多值索引

多值索引跟组合索引是两个概念，多值索引是一种定义在数组列上的二级索引，它主要用于**索引JSON数组**，以下示例来自TiDB官网。

```
mysql> CREATE TABLE customers (
    ->     id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->     name CHAR(10),
    ->     custinfo JSON,
    ->     INDEX zips((CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)))
    -> );
Query OK, 0 rows affected (0.52 sec)

mysql> show create table customers;
+-----------+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
| Table     | Create Table

+-----------+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
| customers | CREATE TABLE `customers` (
  `id` bigint(20) NOT NULL AUTO_INCREMENT,
  `name` char(10) DEFAULT NULL,
  `custinfo` json DEFAULT NULL,
  PRIMARY KEY (`id`) /*T![clustered_index] CLUSTERED */,
  KEY `zips` ((cast(json_extract(`custinfo`, _utf8mb4'$.zipcode') as unsigned array)))
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+-----------+------------------------------------------------------------------------------------------
------------------------------------------------------------------------------------------------------
1 row in set (0.00 sec)
```

覆盖索引

覆盖索引通常是指为了索引回表导致的额外开销，把查询所需要的所有字段都添加到索引中。对于kv存储的数据库，通常的覆盖索引会把查询条件（即"谓词"）涉及的字段放在索引的Key中，而把谓词以外的字段存放在键值对的Value中。

笔者之前使用过的一款数据库中使用覆盖索引的语法为：

CREATE INDEX index-name ON tablename(col1,col2..) **WITH COLUMNS(col3,col4...)**

通过上述语法把col1、col2..映射到Key，而把col3、col4..映射到Value中。

TiDB中的覆盖索引并没有上述的实现方式，而是较为简单的采用了组合索引的方式代替，即把查询所有字段都映射到Key值中，

CREATE INDEX index-name ON tablename(col1,col2,col3,col4...)

那么假如查询语句如下：

SELECT col3,col4 FROM .. WHERE col1=.. AND col2=..

由于索引中包含语句中所有需要的字段信息，因为只需要查询索引即可，而不需要再回表，这在TiDB中称为**覆盖索引优化(covering index optimization)**，参考 用 EXPLAIN 查看索引查询的执行计划 | PingCAP 文档中心

不可见索引

不可见索引其实不能算是一种索引类型，应该算是索引的一种属性。TiDB默认创建的索引都是可见的，有时候我们发现表上创建了过多的索引而且SQL又使用了错误的索引，这时候我们就想**临时把索引设置为不可见**。因为如果直接删除索引的话再创建时可能会非常耗时，我们仅仅是想让优化器看不到这个索引而已，索引的顺序还是会跟着表的数据变动而变化。在这种场景下，我们可以将索引设置为不可见索引，如下图所示。

```
mysql> show create table t_invis;
+---------+-------------------------------------------------------------------------------------------------------------------------
------------------------+
| Table   | Create Table
                       |
+---------+-------------------------------------------------------------------------------------------------------------------------
------------------------+
| t_invis | CREATE TABLE `t_invis` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */,
  KEY `t_idx_invis` (`b`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+---------+-------------------------------------------------------------------------------------------------------------------------
------------------------+
1 row in set (0.00 sec)

mysql> alter table t_invis alter index t_idx_invis invisible;
Query OK, 0 rows affected (0.52 sec)

mysql> show create table t_invis;
+---------+-------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
| Table   | Create Table
                                               |
+---------+-------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
| t_invis | CREATE TABLE `t_invis` (
  `a` int(11) NOT NULL,
  `b` int(11) DEFAULT NULL,
  PRIMARY KEY (`a`) /*T![clustered_index] CLUSTERED */,
  KEY `t_idx_invis` (`b`) /*!80000 INVISIBLE */
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin |
+---------+-------------------------------------------------------------------------------------------------------------------------
-----------------------------------------------+
1 row in set (0.00 sec)
```


评论

添加评论

评 论

wanwan 2024-12-31 00:07
我的理解主键更多的用在数据库内部机制上，唯一索引主要是服务于业务数据查询。

回复

wanwan 2024-12-31 00:06
我们where条件后面，正常人都不会去写主键=xxx，而是唯一索引条件=xxx

回复

wanwan 2024-12-31 00:05
但是我们在实际使用的时候，不会将主键作为查询条件，我们更多都是是用业务唯一数据去作为条件查询，这样就有了唯一索引和主键的区别。能否阐述清楚?

回复

vincentLi 2024-08-01 10:08
很详细，学习啦

回复

< 1 >