**代价模型**

mysql 5.7代价计算相对之前的版本有较大的改进。例如

* 代价模型参数可以动态配置，可以适应不同的硬件
* 区分考虑数据在内存和在磁盘中的代价
* 代价精度提升为浮点型
* jion计算时不仅要考虑condition,还要考虑condition上的filter，具体参见参数condition\_fanout\_filter

5.7 在代价类型上分为io,cpu和memory, 5.7的代价模型还在完善中，memory的代价虽然已经收集了，但还没有没有计算在最终的代价中。  
5.7 在源码上对代价模型进行了大量重构，代价分为server层和engine层。server层主要是cpu的代价，而engine层主要是io的代价。  
5.7 引入了两个系统表mysql.server\_cost和mysql.engine\_cost来分别配置这两个层的代价。  
以下分析均基于mysql5.7.10

**server\_cost**

* row\_evaluate\_cost (default 0.2)  
  计算符合条件的行的代价，行数越多，此项代价越大
* memory\_temptable\_create\_cost (default 2.0)  
  内存临时表的创建代价
* memory\_temptable\_row\_cost (default 0.2)  
  内存临时表的行代价
* key\_compare\_cost (default 0.1)  
  键比较的代价，例如排序
* disk\_temptable\_create\_cost (default 40.0)  
  内部myisam或innodb临时表的创建代价
* disk\_temptable\_row\_cost (default 1.0)  
  内部myisam或innodb临时表的行代价

由上可以看出创建临时表的代价是很高的，尤其是内部的myisam或innodb临时表。

**engine\_cost**

* io\_block\_read\_cost (default 1.0)  
  从磁盘读数据的代价，对innodb来说，表示从磁盘读一个page的代价
* memory\_block\_read\_cost (default 1.0)  
  从内存读数据的代价，对innodb来说，表示从buffer pool读一个page的代价

目前io\_block\_read\_cost和memory\_block\_read\_cost默认值均为1，实际生产中建议酌情调大memory\_block\_read\_cost，特别是对普通硬盘的场景。

**代价配置**

cost参数可以通过修改mysql.server\_cost和mysql.engine\_cost来实现。初始这两个表中的记录cost\_value项均为NULL, 代价值都取上两节介绍的初始值。  
当修改cost\_value为非NULL时，代价值按设定的值计算。修改方法如下：

## 修改io\_block\_read\_cost值为2

UPDATE mysql.engine\_cost

SET cost\_value = 2.0

WHERE cost\_name = 'io\_block\_read\_cost';

# FLUSH OPTIMIZER\_COSTS 生效，只对新连接有效，老连接无效。

FLUSH OPTIMIZER\_COSTS;

另外，在主备环境下，修改cost参数时主备都要修改。因为mysql.server\_cost和mysql.engine\_cost的更新不会参与复制。

**代价分析示例**

初始化数据

create table t1(c1 int primary key, c2 int unique,c3 int) engine=innodb;

let $loop=100;

while($loop)

{

eval insert into t1(c1,c2,c3) values($loop, $loop+1, $loop+2);

dec $loop;

}

set optimizer\_trace = "enabled=on";

cost参数都取默认值，以下示例中会用到row\_evaluate\_cost(0.2),io\_block\_read\_cost(1.0),io\_block\_read\_cost(1.0),memory\_block\_read\_cost(1.0)

**示例1**

以下语句选择覆盖索引c2

explain select c1,c2 from t1 where c2 > 10;

id select\_type table partitions type possible\_keys key key\_len ref rows filtered Extra

1 SIMPLE t1 NULL range c2 c2 5 NULL 91 100.00 Using where; Using index

查看optimizer\_trace, 可以看出全表扫描代价为23.1，通过c2上的索引扫描代价为19.309， 最后选择c2上的索引扫描。

"rows\_estimation": [

{

"table": "`t1`",

"range\_analysis": {

"table\_scan": {

"rows": 100,

"cost": 23.1

},

"potential\_range\_indexes": [

{

"index": "PRIMARY",

"usable": false,

"cause": "not\_applicable"

},

{

"index": "c2",

"usable": true,

"key\_parts": [

"c2"

]

}

],

"best\_covering\_index\_scan": {

"index": "c2",

"cost": 21.109,

"chosen": true

},

"setup\_range\_conditions": [

],

"group\_index\_range": {

"chosen": false,

"cause": "not\_group\_by\_or\_distinct"

},

"analyzing\_range\_alternatives": {

"range\_scan\_alternatives": [

{

"index": "c2",

"ranges": [

"10 < c2"

],

"index\_dives\_for\_eq\_ranges": true,

"rowid\_ordered": false,

"using\_mrr": false,

"index\_only": true,

"rows": 91,

"cost": 19.309,

"chosen": true

}

],

"analyzing\_roworder\_intersect": {

"usable": false,

"cause": "too\_few\_roworder\_scans"

}

},

"chosen\_range\_access\_summary": {

"range\_access\_plan": {

"type": "range\_scan",

"index": "c2",

"rows": 91,

"ranges": [

"10 < c2"

]

},

"rows\_for\_plan": 91,

"cost\_for\_plan": 19.309,

"chosen": true

}

}

}

]

},

{

"considered\_execution\_plans": [

{

"plan\_prefix": [

],

"table": "`t1`",

"best\_access\_path": {

"considered\_access\_paths": [

{

"rows\_to\_scan": 91,

"access\_type": "range",

"range\_details": {

"used\_index": "c2"

},

"resulting\_rows": 91,

"cost": 37.509,

"chosen": true

}

]

},

"condition\_filtering\_pct": 100,

"rows\_for\_plan": 91,

"cost\_for\_plan": 37.509,

"chosen": true

}

]

**全表扫描的代价23.1**

包括io和cpu的代价

test\_quick\_select：

double scan\_time=

cost\_model->row\_evaluate\_cost(static\_cast<double>(records)) + 1;

Cost\_estimate cost\_est= head->file->table\_scan\_cost();

cost\_est.add\_io(1.1);//这里加1.1应该是个调节值

cost\_est.add\_cpu(scan\_time);

其中io代价table\_scan\_cost会根据buffer pool大小和索引大小来估算page in memory和in disk的比例，分别算出代价。

handler::table\_scan\_cost()

ha\_innobase::scan\_time()\*table->cost\_model()->page\_read\_cost(1.0);//1\*1=1

//其中scan\_time计算数据所占page数，

page\_read\_cost计算读取单个page的代价

buffer\_block\_read\_cost(pages\_in\_mem) + io\_block\_read\_cost(pages\_on\_disk);

io代价为1+1.1=2.1

cpu代价为row\_evaluate\_cost

double row\_evaluate\_cost(double rows) const

{

DBUG\_ASSERT(m\_initialized);

DBUG\_ASSERT(rows >= 0.0);

return rows \* m\_server\_cost\_constants->row\_evaluate\_cost(); // 100 \* 0.2(row\_evaluate\_cost)=20;

}

cpu代价为20+1=21；

最终代价为2.1+21=23.1

**c2索引扫描代价19.309**

同样也分为io和cpu代价

multi\_range\_read\_info\_const：

\*cost= index\_scan\_cost(keyno, static\_cast<double>(n\_ranges),

static\_cast<double>(total\_rows));

cost->add\_cpu(cost\_model->row\_evaluate\_cost(static\_cast<double>(total\_rows)) + 0.01);

io代价 1.0987925356750823\*1=1.0987925356750823

index\_scan\_cost:

const double io\_cost= index\_only\_read\_time(index, rows) \* //估算index占page个数 = 1.0987925356750823

table->cost\_model()->page\_read\_cost\_index(index, 1.0); //根据buffer pool大小和索引大小来估算page in memory和in disk的比例，计算读一个page的代价。 = 1

cpu代价91\*0.2+0.01=18.21

cost->add\_cpu(cost\_model->row\_evaluate\_cost(

static\_cast<double>(total\_rows)) + 0.01); //这里根据过滤条件算出的total\_rows为91

最终代价1.0987925356750823+18.21=19.309

**示例2**

以下语句选择了全表扫描

explain select \* from t1 where c2 > 10;

id select\_type table partitions type possible\_keys key key\_len ref rows filtered Extra

1 SIMPLE t1 NULL ALL c2 NULL NULL NULL 100 91.00 Using where

查看optimizer\_trace, 可以看出全表扫描代价为23.1，通过c2上的索引扫描代价为110.21， 最后选择全表扫描。

"rows\_estimation": [

{

"table": "`t1`",

"range\_analysis": {

"table\_scan": {

"rows": 100,

"cost": 23.1

},

"potential\_range\_indexes": [

{

"index": "PRIMARY",

"usable": false,

"cause": "not\_applicable"

},

{

"index": "c2",

"usable": true,

"key\_parts": [

"c2"

]

}

],

"setup\_range\_conditions": [

],

"group\_index\_range": {

"chosen": false,

"cause": "not\_group\_by\_or\_distinct"

},

"analyzing\_range\_alternatives": {

"range\_scan\_alternatives": [

{

"index": "c2",

"ranges": [

"10 < c2"

],

"index\_dives\_for\_eq\_ranges": true,

"rowid\_ordered": false,

"using\_mrr": false,

"index\_only": false,

"rows": 91,

"cost": 110.21,

"chosen": false,

"cause": "cost"

}

],

"analyzing\_roworder\_intersect": {

"usable": false,

"cause": "too\_few\_roworder\_scans"

}

}

}

}

]

},

{

"considered\_execution\_plans": [

{

"plan\_prefix": [

],

"table": "`t1`",

"best\_access\_path": {

"considered\_access\_paths": [

{

"rows\_to\_scan": 100,

"access\_type": "scan",

"resulting\_rows": 91,

"cost": 21,

"chosen": true

}

]

},

"condition\_filtering\_pct": 100,

"rows\_for\_plan": 91,

"cost\_for\_plan": 21,

"chosen": true

}

]

},

**全表扫描代价23.1**

同上一节分析

**c2索引扫描代价为110.21**

上一节通过c2索引扫描代价为19.309，因为是覆盖索引不需要回表，所以代价较少。而此例是需要回表的。

multi\_range\_read\_info\_const：

\*cost= read\_cost(keyno, static\_cast<double>(n\_ranges),

static\_cast<double>(total\_rows));

cost->add\_cpu(cost\_model->row\_evaluate\_cost(

static\_cast<double>(total\_rows)) + 0.01);

io代价需回表

read\_cost: //92\*1=92

const double io\_cost= read\_time(index, static\_cast<uint>(ranges)

static\_cast<ha\_rows>(rows)) \*

table->cost\_model()->page\_read\_cost(1.0);

read\_time: //91+1=92

virtual double read\_time(uint index, uint ranges, ha\_rows rows)

{ return rows2double(ranges+rows); }

这里回表时计算代价为每行代价为1，默认认为回表时每行都对于聚集索引的一个page.

io代价为92

cpu代价为91\*0.2+0.01=18.21  
cost->add\_cpu(cost\_model->row\_evaluate\_cost( static\_cast<double>(total\_rows)) + 0.01);

最后代价为92+18.21=110.21

**总结**

5.7 代价模型优化还在持续改进中，相信后续的版本会越来越好。代价的参数的配置需谨慎，需要大量的测试和验证。