<https://mp.weixin.qq.com/s/uW7NkJmQ2xNq-Dqe-bbibg>

# 你知道 SELECT COUNT(\*) 底层究竟干了啥吗？

* 一、InnoDB 全表 COUNT(\*)
* 二、数据结构
* 三、MyISAM 全表 COUNT(\*)
* 四、几个问题

“SELECT COUNT( \* ) FROM TABLE” 是个再常见不过的 SQL 需求了。在 MySQL 的使用规范中，我们一般使用事务引擎 InnoDB 作为(一般业务)表的存储引擎，在此前提下，COUNT( \* )操作的时间复杂度为 O(N)，其中 N 为表的行数。

而 MyISAM 表中可以快速取到表的行数。这些实践经验的背后是怎样的机制，以及为什么需要/可以是这样，就是此文想要探讨的。

先来看一下概况，MySQL COUNT( \* ) 在 2 种存储引擎中的部分问题：



下面就带着这些问题，以 InnoDB 存储引擎为主来进行讨论。

# 一、InnoDB 全表 COUNT(\*)

## ****主要问题：****

## 1、执行过程是怎样的？

2、如何计算 count？影响 count 结果的因素有哪些？

3、count 值存在哪里？涉及的数据结构是怎样的？

4、为什么 InnoDB 只能通过扫表来实现 count( \* )？(见本文最后的问题)

5、全表COUNT( \* )作为 table scan 类型操作的一个 case，有什么风险？

6、COUNT(\* )操作是否会像“SELECT \* ”一样可能读取大字段涉及的溢出页？

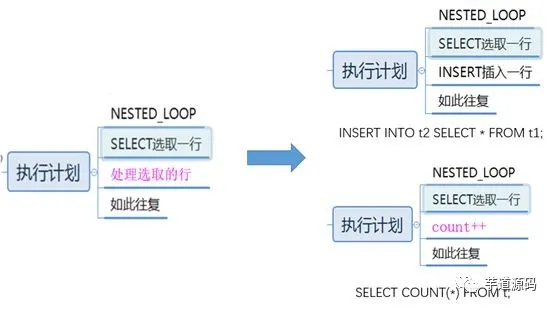
**1、执行框架 – 循环: 读取 + 计数？**

#### ****1.1、基本结论：****

* 全表扫描，一个循环解决问题。
* 循环内: 先读取一行，再决定该行是否计入 count。
* 循环内是一行一行进行计数处理的。

**1.2、说明：**

简单 SELELCT-SQL 的执行框架，类比 INSERT INTO … SELECT 是同样的过程。



下面会逐步细化如何读取与计数 ( count++ ) 。

**2、执行过程？**

执行过程部分，分为 4 个部分：

（1）COUNT( \* ) 前置流程: 从 Client 端发 SQL 语句，到 MySQL-Server 端执行 SELECT 之前，为后面的一些阐述做一铺垫。

（2）COUNT( \* ) 流程: 简要给出代码层面的流程框架及 2 个核心步骤的重点调用栈部分。

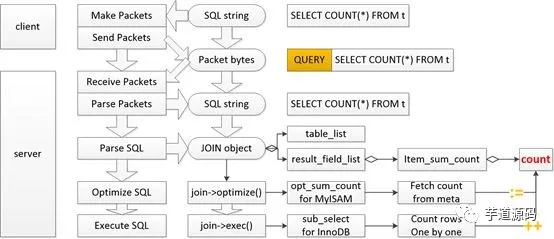
（3）读取一行: 可见性及 row\_search\_mvcc 函数，介绍可见性如何影响 COUNT( \* ) 结果。

（4）计数一行: Evaluate\_join\_record 与列是否为空，介绍计数过程如何影响 COUNT( \* ) 结果。

如果读者希望直接看如何进行 COUNT( \* )，那么也可以忽略 (1)，而直接跳到 (2) 开始看。

#### ****2.1、COUNT( \* ) 前置流程回忆 – 从 Client 端发 SQL 到 sub\_select 函数****

为了使看到的调用过程不太突兀，我们还是先回忆一下如何执行到 sub\_select 函数这来的：



（1）MySQL-Client 端发送 SQL 语句，根据 MySQL 通信协议封包发送。

（2）Mysql-Server 端接收数据包，由协议解析出 command 类型 ( QUERY ) 及 SQL 语句 ( 字符串 ) 。

（3）SQL 语句经过解析器解析输出为 JOIN 类的对象，用于结构化地表达该 SQL 语句。

PS: 这里的 JOIN 结构，不仅仅是纯语法结构，而是已经进行了语义处理，粗略地说，汇总了表的列表 ( table\_list )、目标列的列表 ( target\_list )、WHERE 条件、子查询等语法结构。

在全表 COUNT( \* )-case 中，table\_list = [表“t”(别名也是“t”)]，target\_list = [目标列对象(列名为“COUNT( \* )”)]，当然这里没有 WHERE 条件、子查询等结构。

（4）JOIN 对象有 2 个重要的方法: JOIN::optimize(), JOIN::exec()，分别用于进行查询语句的优化 和 查询语句的执行。

* join->optimize()，优化阶段 (稍后 myisam 下全表 count( \* ) 操作会涉及这里的一点内容)。
* join->exec()，执行阶段 ( 重点 )，包含了 InnoDB 下全表count( \* ) 操作的执行流程。

（5）join->exec() 经过若干调用，将调用到 sub\_select 函数来执行简单 SQL，包括 COUNT( \* ) 。

（6）END of sub\_select 。

#### ****2.2、COUNT( \* ) 流程 ( 于 sub\_select 函数中 )****

上层的流程与代码是比较简单的，集中在 sub\_select 函数中，其中 2 类函数分别对应于前面”执行框架”部分所述的 2 个步骤 – 读取、计数。先给出结论如下：

（1）读取一行：从相对顶层的 sub\_select 函数经过一番调用，最终所有分支将调用到 row\_search\_mvcc 函数中，该函数就是用于从 InnoDB 存储引擎所存储的 B+-tree 结构中读取一行到内存中的一个 buf (uchar \* ) 中，待后续处理使用。

这里会涉及行锁的获取、MVCC 及行可见性的问题。当然对 于 SELECT COUNT( \* ) 这类快照读而言，只会涉及 MVCC 及其可见性，而不涉及行锁。详情可跳至“可见性与 row\_search\_mvcc 函数”部分。

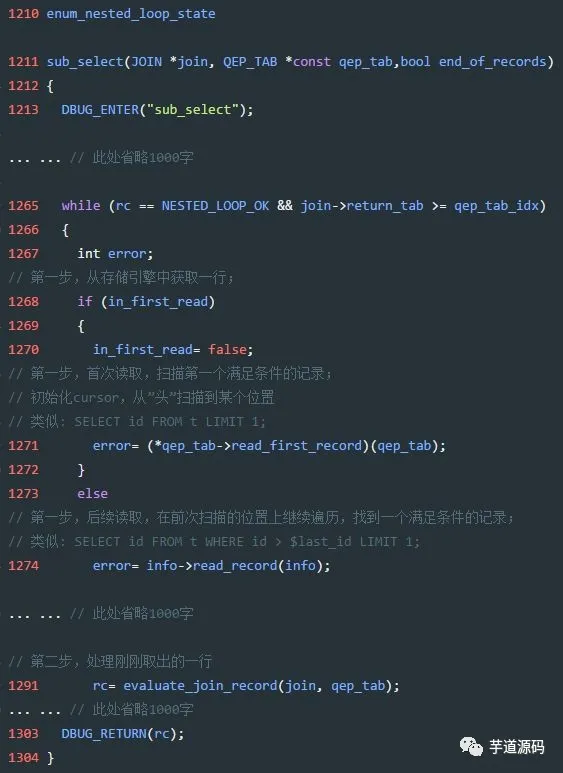
（2）计数一行: 代码层面，将会在 evaluate\_join\_record 函数中对所读取的行进行评估，看其是否应当计入 count 中 ( 即是否要 count++ )。

简单来说，COUNT(arg) 本身为 MySQL 的函数操作，对于一行来说，若括号内的参数 arg ( 某列或整行 ) 的值若不是 NULL，则 count++，否则对该行不予计数。详情可跳至“ Evaluate\_join\_record 与列是否为空”部分。

这两个阶段对 COUNT( \* )结果的影响如下: (两层过滤)



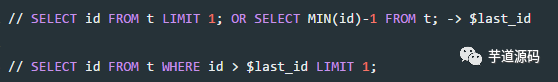
SQL 层流程框架相关代码摘要如下:



**Q：代码层面，第一步骤(读取一行)有 2 个分支，为什么？**

**A：** 从 InnoDB 接口层面考虑，分为 “读第一行” 和 “读下一行”，是 2 个不同的执行过程，读第一行需要找到一个 ( cursor ) 位置并做一些初始化工作让后续的过程可递归。

正如我们如果用脚本/程序来进行逐行的扫表操作，实现上就会涉及下面 2 个 SQL：



具体涉及到此例的代码，SQL 层到存储引擎层的调用关系，读取阶段的调用栈如下：(供参考)

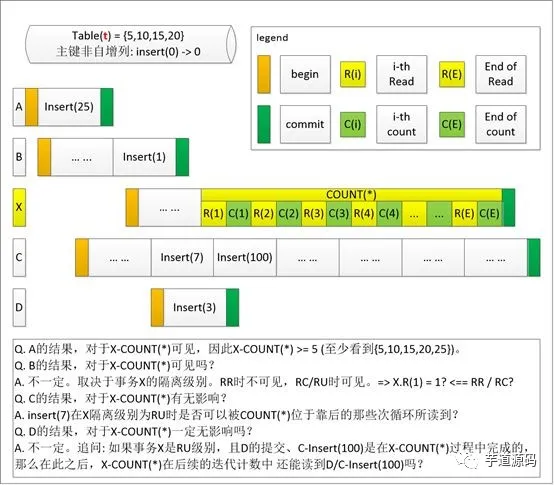


我们可以看到，无论是哪一个分支的读取，最终都殊途同归于 row\_search\_mvcc 函数。

以上是对 LOOP 中的代码做一些简要的说明，下面来看 row\_search\_mvcc 与 evaluate\_join\_record 如何输出最终的 count 结果。

#### ****2.3、行可见性及 row\_search\_mvcc 函数****

这里我们主要通过一组 case 和几个问题来看行可见性对 COUNT( \* ) 的影响。



**Q：对于“SELECT COUNT( \* ) FROM t”或者“SELECT MIN(id) FROM t”操作，第一次的读行操作读到的是表 t 中 ( B+ 树最左叶节点 page 内 ) 的最小记录吗？( ha\_index\_first 为何也调用 row\_search\_mvcc 来获取最小 key 值？)**

**A：** 不一定。即使是 MIN ( id ) 也不一定就读取的是 id 最小的那一行，因为也同样有行可见性的问题，实际上 index\_read 取到的是 当前事务内语句可见的最小 index 记录。这也反映了前面提到的 join\_read\_first 与 join\_read\_next “殊途同归”到 row\_search\_mvcc 是理所应当的。

**Q：针对图中最后一问，如果事务 X 是 RU ( Read-Uncommitted ) 隔离级别，且 C-Insert ( 100 ) 的完成是在 X-count( \* ) 执行过程中 ( 仅扫描到 5 或 10 这条记录 ) 完成的，那么 X-count( \* ) 在事务 C-Insert ( 100 ) 完成后，能否在之后的读取过程中看到 100 这条记录呢？**

**A：** MySQL 采取”读到什么就是什么”的策略，即 X-count( \* ) 在后面可以读到 100 这条记录。

#### ****2.4、evaluate\_join\_record 与列是否为空****

**Q：某一行如何计入 count？**

**A：** 两种情况会将所读的行计入 count:

（1）如果 COUNT 函数中的参数是某列，则会判断所读行中该列定义是否 Nullable 以及该列的值是否为 NULL；若两者均为是，则不会计入 count，否则将计入 count。

* e.g. SELECT COUNT(col\_name) FROM t
* col\_name 可以是主键、唯一键、非唯一键、非索引字段

（2）如果 COUNT 中带有 \* ，则会判断这部分的整行是否为 NULL，如果判断参数为 NULL，则忽略该行，否则 count++。

* e.g-1. SELECT COUNT(\*) FROM t
* e.g-2. SELECT COUNT(B.\*) FROM A LEFT JOIN B ON A.id = B.id

**Q：特别地，对于 SELECT COUNT(id) FROM t，其中 id 字段是表 t 的主键，则如何？**

**A：** 效果上等价于 COUNT( \* )。因为无论是 COUNT( \* )，还是 COUNT ( pk\_col ) 都是因为有主键从而充分断定索取数据不为 NULL，这类 COUNT 表达式可以用于获取当前可见的表行数。

**Q：用户层面对 InnoDB COUNT( \* ) 的优化操作问题**

**A：** 这个问题是业界熟悉的一个问题，扫描非空唯一键可得到表行数，但所涉及的字节数可能会少很多(在表的行长与主键、唯一键的长度相差较多时)，相对的 IO 代价小很多。

相关调用栈参考如下：

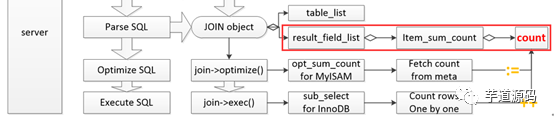


# 二、数据结构

**Q：count 值存储在哪个内存变量里？**

**A：** SQL 解析后，存储于表达 COUNT( \* ) 这一项中，((Item\_sum\_count\*)item\_sum)->count

如下图所示回顾我们之前“COUNT( \* )前置流程”部分提到的 JOIN 结构。



即 SQL 解析器为每个 SQL 语句进行结构化，将其放在一个 JOIN 对象 ( join ) 中来表达。在该对象中创建并填充了一个列表 result\_field\_list 用于存放结果列，列表中每个元素则是一个结果列的 ( Item\_result\_field\* ) 对象 ( 指针 ) 。

在 COUNT( \* )-case 中，结果列列表只包含一个元素，( Item\_sum\_count: public Item\_result\_field ) 类型对象 ( name = “COUNT( \* )”)，其中该类所特有的成员变量 count即为所求。

# 三、MyISAM 全表 COUNT(\*)

由于 MyISAM 引擎并不常用于实际业务中，仅做简要描述如下：

1、MyISAM-COUNT( \* ) 操作是 O(1) 时间复杂度的操作。

2、每张 MyISAM 表中存放了一个 meta 信息-count 值，在内存中与文件中各有一份，内存中的 count 变量值通过读取文件中的 count 值来进行初始化。

3、SELECT COUNT( \* ) FROM t 会直接读取内存中的表 t 对应的 count 变量值。

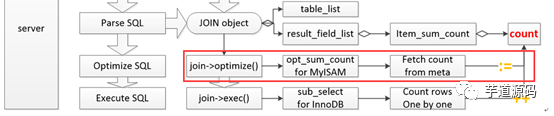
4、内存中的 count 值与文件中的 count 值由写操作来进行更新，其一致性由表级锁来保证。

5、表级锁保证的写入串行化使得，同一时刻所有用户线程的读操作要么被锁，要么只会看到一种数据状态。

# 四、几个问题

**Q：MyISAM 与 InnoDB 在 COUNT( \* ) 操作的执行过程在哪里开始分道扬镳？**

* 共性：共性存在于 SQL 层，即 SQL 解析之后的数据结构是一致的，count 变量都是存在于作为结果列的 Item\_sum\_count 类型对象中；返回给客户端的过程也类似 – 对该 count 变量进行赋值并经由 MySQL 通信协议返回给客户端。
* 区别：InnoDB 的 count 值计算是在 SQL 执行阶段进行的；而 MyISAM 表本身在内存中有一份包含了表 row\_count 值的 meta 信息，在 SQL 优化阶段通过存储引擎的标记给优化器一个 hint，表明该表所用的存储引擎保存了精确行数，可以直接获取到，无需再进入执行器。



**Q：InnoDB 中为何无法向 MyISAM 一样维护住一个 row\_count 变量？**

**A：** 从 MVCC 机制与行可见性问题中可得到原因，每个事务所看到的行可能是不一样的，其 count( \* ) 结果也可能是不同的；反过来看，则是 MySQL-Server 端无法在同一时刻对所有用户线程提供一个统一的读视图，也就无法提供一个统一的 count 值。

PS: 对于多个访问 MySQL 的用户线程 ( COUNT( \* ) ) 而言，决定它们各自的结果的因素有几个:

（1）一组事务执行前的数据状态(初始数据状态)。

（2）有时间重叠的事务们的执行序列 (操作时序，事务理论表明 并发事务操作的可串行化是正确性的必要条件)。

（3）事务们各自的隔离级别(每个操作的输入)。

其中 1、2 对于 Server 而言都是全局或者说可控的，只有 3 是每个用户线程中事务所独有的属性，这是 Server 端不可控的因素，因此 Server 端也就对每个 COUNT( \* ) 结果不可控了。

**Q：InnoDB-COUNT( \* ) 属 table scan 操作，是否会将现有 Buffer Pool 中其它用户线程所需热点页从 LRU-list 中挤占掉，从而其它用户线程还需从磁盘 load 一次，突然加重 IO 消耗，可能对现有请求造成阻塞？**

**A：** MySQL 有这样的优化策略，将扫表操作所 load 的 page 放在 LRU-list 的 oung/old 的交界处 ( LRU 尾部约 3/8 处 )。这样用户线程所需的热点页仍然在 LRU-list-young 区域，而扫表操作不断 load 的页则会不断冲刷 old 区域的页，这部分的页本身就是被认为非热点的页，因此也相对符合逻辑。

PS: 个人认为还有一种类似的优化思路，是限定扫描操作所使用的 Buffer Pool 的大小为 O(1) 级别，但这样做需要付出额外的内存管理成本。

**Q：InnoDB-COUNT( \* ) 是否会像 SELECT \* FROM t 那样读取存储大字段的溢出页(如果存在)？**

**A：否。** 因为 InnoDB-COUNT( \* ) 只需要数行数，而每一行的主键肯定不是 NULL，因此只需要读主键索引页内的行数据，而无需读取额外的溢出页。