

10 | 走进黑盒：SQL是如何在数据库中执行的？

2020-03-19 李玥

后端存储实战课

[进入课程 >](#)



讲述：李玥

时长 16:06 大小 14.76M



你好，我是李玥。

上一节课我们讲了怎么来避免写出慢 SQL，课后我给你留了一道思考题：在下面这两个 SQL 中，为什么第一个 SQL 在执行的时候无法命中索引呢？

```
1 SELECT * FROM user WHERE left(department_code, 5) = '00028';  
2 SELECT * FROM user WHERE department_code LIKE '00028%';
```

复制代码



原因是，这个 SQL 的 WHERE 条件中对 department_code 这个列做了一个 left 截取的计算，对于表中的每一条数据，都得先做截取计算，然后判断截取后的值，所以不得不做全表

扫描。你在写 SQL 的时候，尽量不要在 WHERE 条件中，对列做任何计算。

到这里这个问题就结束了么？那我再给你提一个问题，这两个 SQL 中的 WHERE 条件，虽然写法不一样，但它俩的语义不就是一样的么？是不是都可以解释成：department_code 这一列前 5 个字符是 00028？从语义上来说，没有任何不同是吧？所以，它们的查询结果也是完全一样的。那凭什么第一条 SQL 就得全表扫描，第二条 SQL 就可以命中索引？

对于我们日常编写 SQL 的一些优化方法，比如说我刚刚讲的：“尽量不要在 WHERE 条件中，对列做计算”，很多同学只是知道这些方法，但是却不知道，为什么按照这些方法写出来的 SQL 就快？

要回答这些问题，需要了解一些数据库的实现原理。对很多开发者来说，数据库就是个黑盒子，你会写 SQL，会用数据库，但不知道盒子里面到底是怎么一回事儿，这样你只能机械地去记住别人告诉你的那些优化规则，却不知道为什么要遵循这些规则，也就谈不上灵活运用。

今天这节课，我带你一起打开盒子看一看，SQL 是如何在数据库中执行的。


数据库是一个非常非常复杂的软件系统，我会尽量忽略复杂的细节，用简单的方式把最主要的原理讲给你。即使这样，这节课的内容仍然会非常的硬核，你要有所准备。

数据库的服务端，可以划分为**执行器 (Execution Engine)** 和 **存储引擎 (Storage Engine)** 两部分。执行器负责解析 SQL 执行查询，存储引擎负责保存数据。

SQL 是如何在执行器中执行的？

我们通过一个例子来看一下，执行器是如何来解析执行一条 SQL 的。

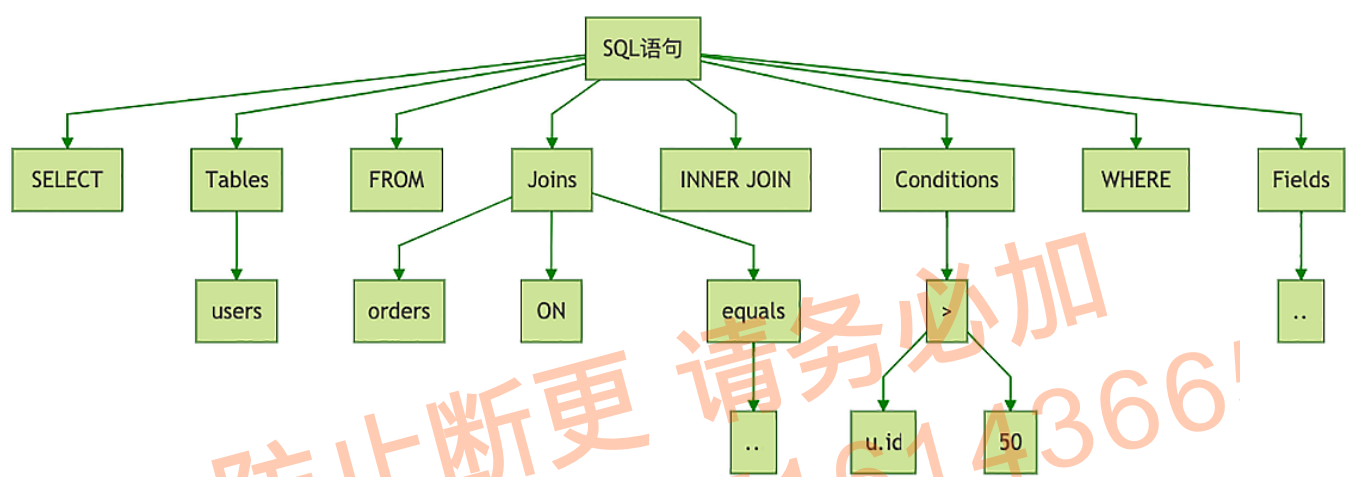
```
1 SELECT u.id AS user_id, u.name AS user_name, o.id AS order_id
2 FROM users u INNER JOIN orders o ON u.id = o.user_id
3 WHERE u.id > 50
```

 复制代码

这个 SQL 语义是，查询用户 ID 大于 50 的用户的所有订单，这是很简单的一个联查，需要查询 users 和 orders 两张表，WHERE 条件就是，用户 ID 大于 50。

数据库收到查询请求后，需要先解析 SQL 语句，把这一串文本解析成便于程序处理的结构化数据，这就是一个通用的语法解析过程。跟编程语言的编译器编译时，解析源代码的过程是完全一样的。如果是计算机专业的同学，你上过的《编译原理》这门课，其中很大的篇幅是在讲解这一块儿。没学过《编译原理》的同学也不用担心，你暂时先不用搞清楚，SQL 文本是怎么转换成结构化数据的，不妨碍你学习和理解这节课下面的内容。

转换后的结构化数据，就是一棵树，这个树的名字叫抽象语法树（AST，Abstract Syntax Tree）。上面这个 SQL，它的 AST 大概是这样的：



这个树太复杂，我只画了主要的部分，你大致看一下，能理解这个 SQL 的语法树长什么样就行了。执行器解析这个 AST 之后，会生成一个逻辑执行计划。所谓的执行计划，可以简单理解为如何一步一步地执行查询和计算，最终得到执行结果的一个分步骤的计划。这个逻辑执行计划是这样的：

复制代码

```
1 LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
2   LogicalFilter(condition=[$0 > 50])
3     LogicalJoin(condition=[$0 == $6], joinType=[inner])
4       LogicalTableScan(table=[users])
5       LogicalTableScan(table=[orders])
```

和 SQL、AST 不同的是，这个逻辑执行计划已经很像可以执行的程序代码了。你看上面这个执行计划，很像我们编程语言的函数调用栈，外层的方法调用内层的方法。所以，要理解这个执行计划，得从内往外看。

1. 最内层的 2 个 LogicalTableScan 的含义是，把 USERS 和 ORDERS 这两个表的数据都读出来。
2. 然后拿这两个表所有数据做一个 LogicalJoin，JOIN 的条件就是第 0 列 (u.id) 等于第 6 列 (o.user_id)。
3. 然后再执行一个 LogicalFilter 过滤器，过滤条件是第 0 列 (u.id) 大于 5。
4. 最后，做一个 LogicalProject 投影，只保留第 0(user_id)、1(user_name)、5(order_id) 三列。这里“投影 (Project)”的意思是，把不需要的列过滤掉。

把这个逻辑执行计划翻译成代码，然后按照顺序执行，就可以正确地查询出数据了。但是，按照上面那个执行计划，需要执行 2 个全表扫描，然后再把 2 个表的所有数据做一个 JOIN 操作，这个性能是非常非常差的。

我们可以简单算一下，如果，user 表有 1,000 条数据，订单表里面有 10,000 条数据，这个 JOIN 操作需要遍历的行数就是 $1,000 \times 10,000 = 10,000,000$ 行。可见，这种从 SQL 的 AST 直译过来的逻辑执行计划，一般性能都非常差，所以，需要对执行计划进行优化。

如何对执行计划进行优化，不同的数据库有不同的优化方法，这一块儿也是不同数据库性能有差距的主要原因之一。优化的总体思路是，在执行计划中，尽早地减少必须处理的数据量。也就是说，尽量在执行计划的最内层减少需要处理的数据量。看一下简单优化后的逻辑执行计划：

 复制代码

```
1 LogicalProject(user_id=[$0], user_name=[$1], order_id=[$5])
2     LogicalJoin(condition=[$0 == $6], joinType=[inner])
3         LogicalProject(id=[$0], name=[$1])                // 尽早执行投影
4             LogicalFilter(condition=[$0 > 50])           // 尽早执行过滤
5                 LogicalTableScan(table=[users])
6         LogicalProject(id=[$0], user_id=[$1])             // 尽早执行投影
7             LogicalTableScan(table=[orders])
```

对比原始的逻辑执行计划，这里我们做了两点简单的优化：

1. 尽早地执行投影，去除不需要的列；
2. 尽早地执行数据过滤，去除不需要的行。

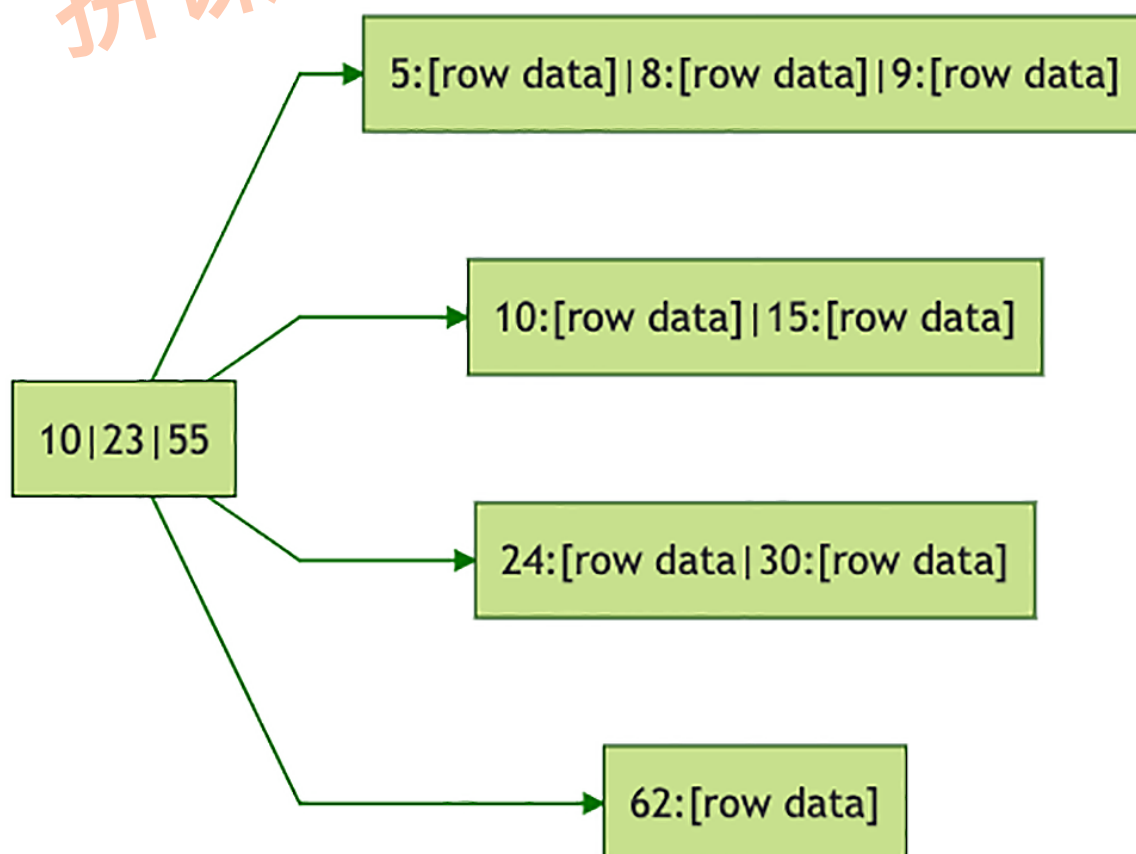
这样，就可以在做 JOIN 之前，把需要 JOIN 的数据尽量减少。这个优化后的执行计划，显然会比原始的执行计划快很多。

到这里，执行器只是在逻辑层面分析 SQL，优化查询的执行逻辑，我们执行计划中操作的数据，仍然是表、行和列。在数据库中，表、行、列都是逻辑概念，所以，这个执行计划叫“逻辑执行计划”。执行查询接下来的部分，就需要涉及到数据库的物理存储结构了。

SQL 是如何在存储引擎中执行的？

数据真正存储的时候，无论在磁盘里，还是在内存中，都没法直接存储这种带有行列的二维表。数据库中的二维表，实际上是怎么存储的呢？这就是存储引擎负责解决的问题，存储引擎主要功能就是把逻辑的表行列，用合适的物理存储结构保存到文件中。不同的数据库，它们的物理存储结构是完全不一样的，这也是各种数据库之间巨大性能差距的根本原因。

我们还是以 MySQL 为例来说一下它的物理存储结构。MySQL 非常牛的一点是，它在设计层面对存储引擎做了抽象，它的存储引擎是可以替换的。它默认的存储引擎是 InnoDB，在 InnoDB 中，数据表的物理存储结构是以主键为关键字的 B+ 树，每一行数据直接就保存在 B+ 树的叶子节点上。比如，上面的订单表组织成 B+ 树，是这个样的：



这个树以订单表的主键 `orders.id` 为关键字组织，其中 “62:[row data]” ，表示的是订单号为 62 的一行订单数据。在 InnoDB 中，表的索引也是以 B+ 树的方式来存储的，和存储数据的 B+ 树的区别是，在索引树中，叶子节点保存的不是行数据，而是行的主键值。


如果通过索引来检索一条记录，需要先后查询索引树和数据树这两棵树：先在索引树中检索到行记录的主键值，然后再用主键值去数据树中去查找这一行数据。

简单了解了存储引擎的物理存储结构之后，我们回过头来继续看 SQL 是怎么在存储引擎中继续执行的。优化后的逻辑执行计划将会被转换成物理执行计划，物理执行计划是和数据的物理存储结构相关的。还是用 InnoDB 来举例，直接将逻辑执行计划转换为物理执行计划：

 复制代码

```
1 InnodbProject(user_id=[0], user_name=[1], order_id=[5])
2   InnodbJoin(condition=[0 == 6], joinType=[inner])
3     InnodbTreeNodesProject(id=[key], name=[data[1]])
4       InnodbFilter(condition=[key > 50])
5         InnodbTreeScanAll(tree=[users])
6       InnodbTreeNodesProject(id=[key], user_id=[data[1]])
7         InnodbTreeScanAll(tree=[orders])
```

物理执行计划同样可以根据数据的物理存储结构、是否存在索引以及数据多少等各种因素进行优化。这一块儿的优化规则同样是非常复杂的，比如，我们可以把对用户树的全树扫描再按照主键过滤这两个步骤，优化为对树的范围查找。

 复制代码

```
1 PhysicalProject(user_id=[0], user_name=[1], order_id=[5])
2   PhysicalJoin(condition=[0 == 6], joinType=[inner])
3     InnodbTreeNodesProject(id=[key], name=[data[1]])
4       InnodbTreeRangeScan(tree=[users], range=[key > 50]) // 全树扫描再按
5     InnodbTreeNodesProject(id=[key], user_id=[data[1]])
6       InnodbTreeScanAll(tree=[orders])
```

最终，按照优化后的物理执行计划，一步一步地去执行查找和计算，就可以得到 SQL 的查询结果了。

理解数据库执行 SQL 的过程，以及不同存储引擎中的数据和索引的物理存储结构，对于正确使用和优化 SQL 非常有帮助。

比如，我们知道了 InnoDB 的索引实现后，就很容易明白为什么主键不能太长，因为表的每个索引保存的都是主键的值，过长的主键会导致每一个索引都很大。再比如，我们了解了执行计划的优化过程后，就很容易理解，有的时候明明有索引却不能命中的原因是，数据库在对物理执行计划优化的时候，评估发现不走索引，直接全表扫描是更优的选择。

回头再来看一下这节课开头的那两条 SQL，为什么一个不能命中索引，一个能命中？原因是 InnoDB 对物理执行计划进行优化的时候，能识别 LIKE 这种过滤条件，转换为对索引树的范围查找。而对第一条 SQL 这种写法，优化规则就没那么“智能”了。

它并没有识别出来，这个条件同样可以转换为对索引树的范围查找，而走了全表扫描。并不是说第一个 SQL 写的不好，而是数据库还不够智能。那现实如此，我们能做的就是尽量了解数据库的脾气秉性，按照它现有能力，尽量写出它能优化好的 SQL。

小结

一条 SQL 在数据库中执行，首先 SQL 经过语法解析成 AST，然后 AST 转换为逻辑执行计划，逻辑执行计划经过优化后，转换为物理执行计划，再经过物理执行计划优化后，按照优化后的物理执行计划执行完成数据的查询。几乎所有的数据库，都是由**执行器**和**存储引擎**两部分组成，执行器负责执行计算，存储引擎负责保存数据。

掌握了查询的执行过程和数据库内部的组成，你才能理解那些优化 SQL 的规则，这些都有助于你更好理解数据库行为，更高效地去使用数据库。

最后需要说明的一点是，今天这节课所讲的内容，不只是适用于我们用来举例的 MySQL，几乎所有支持 SQL 的数据库，无论是传统的关系型数据库、还是 NoSQL、NewSQL 这些新兴的数据库，无论是单机数据库还是分布式数据库，比如 HBase、Elasticsearch 和 SparkSQL 等等这些数据库，它们的实现原理也都符合我们今天这节课所讲的内容。

思考题

课后请你选一种你熟悉的**非关系型数据库**，最好是支持 SQL 的，当然，不支持 SQL 有自己的查询语言也可以。比如说 HBase、Redis 或者 MongoDB 等等都可以，尝试分析一下查

询的执行过程，对比一下它的执行器和存储引擎与 MySQL 有什么不同。

欢迎你在留言区与我讨论，如果你觉得今天的内容对你有帮助，也欢迎把它分享给你的朋友。

后端存储实战课

类电商平台存储技术应用指南

李玥

京东零售计算存储平台部资深架构师



新版升级：点击「👤请朋友读」，20位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 09 | 怎么能避免写出慢SQL?

下一篇 11 | MySQL如何应对高并发（一）：使用缓存保护MySQL

精选留言 (9)

写留言



李鑫

2020-03-19

老师好,看了你的课程 感觉有点浅，比如这一篇只是简单介绍了下索引和数据的底层存储结构。像页分裂这些更加底层的没有讲到，建议老师后续的课程可以由浅入深。



5



Simon

2020-03-19

文中说: 每一行数据直接就保存在 B+ 树的叶子节点上
这句话可能会有误会,
实际上B+树的节点存的是"页", 而具体的数据在页里面
展开



2



一步

2020-03-19

MySQL 的执行计划是如何进行查看的?

展开

1

1



大叶枫

2020-03-19

建议配合实际工作场景得问题, 逐步深入的来解执行计划的实战用途。



1



刘楠

2020-03-20

看了下mysql还是有点蒙, 慢慢理解了

展开



小袁

2020-03-19

如何结合文章理解小表驱动还是大表驱动呢? 我还是想不清楚。

1



峰

2020-03-19

基本上就是一个提供了命令式的语言, 用户告诉告诉数据库做什么东西就行。没有sql 就一般在存储层简单封装了一层对外的接口, 而这层接口就和存储模型有很大关系, 比如hbase, redis 都是键值存储, 所以对外主要操作就是get put 等等。

展开



一步

2020-03-19

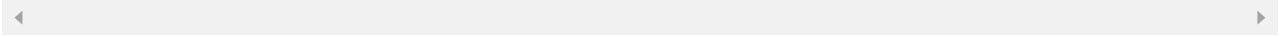
存储引擎再执行执行计划的的时候, 是把整个执行计划执行完成后把数据返给执行器, 还

是每执行一条执行计划获取数据就返给执行器，然后执行器在做运算的？

个人认为是整个执行计划执行完成后获得最终的数据在返给执行器，但是这个有没有办法去验证的？

展开 ∨

作者回复：一般都是在存储引擎直接执行的。



南山

2020-03-19

很赞同为什么要了解原理的原因，只有知道原理以及内部执行逻辑，遇到问题才能不会像无头苍蝇一样靠运气和蒙。

展开 ∨

