### 对索引Include子句的仔细研究

翻译：陈雁飞 校对：

### 作者简介

Markus Winand专注于传授高效的SQL，有[面授](https://winand.at/sql-training/inhouse)和[网络](https://winand.at/sql-training/online)课程两种。他使用[现代SQL](https://modern-sql.com/)方法大大减少了开发时间，并且通过[智能索引](https://use-the-index-luke.com/sql/sorting-grouping/indexed-order-by)优化SQL运行时间。编写的《SQL Performance Explained》已经成为标准读物。

### 译者简介

**陈雁飞**，开源PostgreSQL爱好者，一直从事PostgreSQL数据库运维工作

像微软的SQL Server，IBM的DB2，同时包括从11发行版本开始的PostgreSQL数据库，在创建索引的语句中都支持include子句。介绍PostgreSQL中的这个特性是促使我写这篇长文介绍include子句的动机。

在详细介绍之前，让我们先简单回顾下（非聚集）B树索引的工作原理以及强大的仅索引扫描（index-only scan）。

内容如下:

1. 回顾：B-tree索引
2. 回顾：仅索引扫描（Index-Only scan）
3. Include子句
4. 对Include列进行过滤
5. Include子句上的唯一索引
6. 兼容性
7. PostgreSQL：可见性检查前不会过滤

### 回顾：B-tree索引

要理解include子句，首先必须了解使用索引最多会影响三层数据结构：

* B-Tree
* B-tree叶子节点级别上的双向链表
* 表

前面两个结构在一起形成一个索引，因此它们可以组合成一个单独的项，比

如：”B-tree 索引”。我更喜欢将它们独立分开，因为它们满足不同的需求并且对性能产生不同的影响。此外，解释include子句需要对它们进行区分。

|  |
| --- |
|  |

在一般情况下，数据库软件开始遍历B-tree直到在叶子节点层级（1）上找到第一个匹配的元组。然后沿着双向链表，直到找到所有匹配的元组（2），并且最终从表（3）中获得这些匹配的元组。实际上，最后两步可以交错进行，但是这与理解一般的概念无关。

|  |
| --- |
|  |

下面的公式可以让你大致了解每个步骤需要多少次读取操作。这三个步骤的总和是索引访问的总工作量0。

* B-Tree：log100(<表中行数>)，经常小于5
* 双向链表：<索引读取的行数> / 100
* 表：<表中读取的行数>1

当加载几行时，B-tree在所有步骤中的贡献是最多的。当你只需要从表中获取少量行时，首先就会进行这一步。不论哪种情况下-少数或者多行-双向链表通常是次要因素，因为它将具有相似值的行彼此相邻的存储，这使得的那个读操作可以获取100行甚至更多。公式中通过单独的除数反映这一点2。

**注意：**

如果你正在考虑“这就是我们拥有聚簇索引的原因”，请阅读我的文章“[Unreasonable Defaults: Primary Key as Clustering Key](https://use-the-index-luke.com/blog/2014-01/unreasonable-defaults-primary-key-clustering-key)”。

优化的通用思路是，做更少的工作获得相同的目标。当涉及索引访问的时候，如果数据库软件不需要任何数据结构中的数据，那么它将忽略对数据结构的访问3。

你可以阅读更多关于B-Tree索引内部工作原理：《[*SQL Performance Explained*](https://sql-performance-explained.com/?utm_source=use-the-index-luke.com&utm_campaign=blog-2019-04-include&utm_medium=web&utm_content=btree)》中《[*Chapter 1,* Anatomy of an SQL Index](https://use-the-index-luke.com/sql/anatomy)》章节。

### 回顾：仅索引扫描

仅索引扫描的工作原理如下：如果需要的数据可以在索引的双向链表中获得，则会省略对表的访问。

考虑下面从“[Index-Only Scan:Avoding Table Access](https://use-the-index-luke.com/sql/clustering/index-only-scan-covering-index)”中借过来的索引和查询例子。

|  |
| --- |
| CREATE INDEX idx  ON sales  ( subsidiary\_id, eur\_value )  SELECT SUM(eur\_value)  FROM sales  WHERE subsidiary\_id = ? |

乍一看，你可能会对列eur\_value出现在索引定义中产生疑惑，因为它没有出现在where子句中。

**B-tree索引对很多子句有帮助**

一个常见的误解是认为索引只对where子句有帮助。

B-tree索引同样可以对order by、group by、select和其他语句产生帮助。

索引的B-Tree部分作用很多，而双向链表不能被其他子句使用。

这个例子中的关键点是B-tree索引恰好具有所必须的列，这样数据库软件不需要访问表本身。这就是我们说的仅索引扫描。

|  |
| --- |
|  |

应用上面的公式，如果只有几行满足where子句，那么性能优势非常小。另一方面，如果满足where子句的行数非常多，比如数百万，那么读操作的数量基本上减少了100倍。

**注意：**

仅索引扫描将性能提升一个或两个数量级并不少见。

上面的例子使用B-tree叶子节点的双向链表中包含eur\_value列的情况。尽管B-Tree中其它节点也包含该列，但是这个查询不会用到这些节点中的信息。

### Include子句

Include子句允许我们可以对如下两种类型列进行区分，一个是我们希望的完整索引（关键列），另外一种是只在叶子节点上需要的（包含列）。这意味着我们可以从非叶子节点上删除该列，如果不需要的话。

使用include子句，我们可以优化查询语句的索引：

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id )  **INCLUDE ( eur\_value )** |

查询仍然可以使用该索引进行仅索引扫描，因此性能基本是一样的。

|  |
| --- |
|  |

除了图片上中的明显差异外，还有一个更微妙的区别：叶节点条目的顺序不考虑include中的列。索引仅仅按照其关键列排序4。这样做有两个后果：不能使用inclde列来阻止排序，也不能将它们视为唯一的（见下文）。

**覆盖索引（Converting index）**

术语”覆盖索引”有时用于仅索引扫描或者include子句的上下文中。由于这个术语通常用于不同的含义，所以我一般避免使用它。

重要的是，对于给定的查询语句是否可以通过给定的索引使用仅索引扫描方式。与该索引是否具有include子句或包含表中所有列是不相关的。

与原始索引定义相比，新定义的带有include子句的索引有以下优势：

* 索引树可能具有较少的层级 (<~40%)

由于双向列表上的树节点不包含include列信息，因此数据库可以在每个块存储更多分支，从而索引树可以具有更少的层级。

* 索引会略小（<~3%）

由于树的非叶节点不包含include列信息，因此该索引的总体大小略小。但是，在任何情况下，索引的叶子层级节点需要的空间是最多的，因此其他节点潜在的空间节省相对而言就比较少。

* 记录了索引了目的

这绝对是include子句中最被低估的好处：索引自身的定义表明了为什么列会在索引中。

让我详细说明最后一项。

当扩展一个现有索引的时候，知道索引当前定义方式的原因是非常重要的。在不会破坏任何其他查询的情况下，可以直接应用这些知识自由更改索引的定义。

查询语句如下：

|  |
| --- |
| SELECT \*  FROM sales  WHERE subsidiary\_id = ?  **ORDER BY ts DESC**  **FETCH FIRST 1 ROW ONLY** |

和以前一样，对于给定的subsidiary，这个查询将获取最新的sale信息（ts表示时间戳）。

优化这个语句，一般建立以关键列（subsidiary\_id，ts）开头的索引。通过这个索引，数据库可以直接定位到subsidiary最近的元组上。而不需要读取所有的元组，因为双向链表根据索引键值排序，即任何给定的subsidiary最后一个元组，必然拥有该subsidiary中最大的ts值。使用此方法，查询基本上与主键查找一样快。参考“[Indexing Order By](https://use-the-index-luke.com/sql/sorting-grouping/indexed-order-by)”和“[Querying Top-N Rows](https://use-the-index-luke.com/sql/partial-results/top-n-queries)”阅读关于这项技术的更加详细信息。

**为我自己代言**

我从事培训、其他SQL相关服务以及卖书谋生。可以阅读 <https://winand.at/>了解更多内容。

在为这个查询语句添加索引之前，我们应该检查是否存在可以更改（扩展）现有索引的方式来支持这个查询技巧。这通常是一个很好的做法，因为扩展现有索引对维护的开销影响小于增加一个新的索引。但是在更改现有索引的时候，我们需要确保不会对其他查询语句造成影响。

如果我们查询原始索引定义，会遇到一个问题：

|  |
| --- |
| CREATE INDEX idx  ON sales  ( subsidiary\_id, eur\_value ) |

要使这个索引支持上述查询的order by子句，我们需要在两个已经存在的索引列之间插入ts列，如下所示：

|  |
| --- |
| CREATE INDEX idx  ON sales  ( subsidiary\_id**, ts**, eur\_value ) |

但是，这可能会使得该索引对于需要eur\_value在第二列的查询不起作用，比如它在where或者order by子句中。更改这个索引涉及相当大的风险：破坏了其他查询，除非我们知道没有此类查询。如果我们不知道，通常最好保持索引不变，并为新查询新建一个索引。

但是如果我们从include子句出发看索引，情况会完全发生变化。

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id )  INCLUDE ( eur\_value ) |

由于eur\_value列在include子句中，没有存在于非叶子节点上，因此它既不能用于树的查找，也不能用于排序。将新列添加到键值末尾是相对安全地。

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id**, ts** )  INCLUDE ( eur\_value ) |

尽管对于其他查询仍然存在一定的负面影响，但是通常情况下值得冒险5。

从索引发展角度看，如果这是你需要的，将列放入include子句是非常有帮助的。刚刚添加的索引将成为仅索引扫描的主要候选。

### 对Include列进行过滤

到目前为止，我们一直关注include子句如何满足仅索引扫描。让我们看另外一个案例，通过在索引中增加一个额外列获得的收益。

|  |
| --- |
| SELECT \*  FROM sales  WHERE subsidiary\_id = ?  AND notes LIKE '%search term%' |

这里我将search term设置为文本值，以展示前缀和后缀匹配的情况。当然，你在代码中可以使用绑定参数的方式。

现在，我们为这个查询考虑正确的索引。显然，subsidiary\_id  需要放置在第一个位置上。如果我们采用上面的索引，它已经满足了需求：

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id**, ts** )  INCLUDE ( eur\_value ) |

如开始描述的那样，数据库软件将按照下面三个步骤使用该索引：

1. 对于给定的subsidiary，它将使用B-tree来查找第一个索引元组
2. 它将沿着双向链表查询该subsidiary的所有sales值

（3） 它将从表中获取所有相关的sales值，删除那些鱼notes列上like模式不匹配的元组并返回剩余的行。

问题是上面过程中最后一步：访问表并加载行，但是不知道它们是否能满足最终的结果。通常，访问表是运行查询总工作量的最大贡献者。扫描最终不会选择的数据对于性能来说是无法接受的。

**重要：**

避免加载对最终查询结果没有影响的数据。

这个特定查询的挑战在于它使用了in-fix like 模式。普通的B-tree索引不支持搜索此类模式。但是，B-tree索引仍然支持对此类索引进行过滤。请注意这里的重点：索引和过滤。

换句话说，如果notes列存在于双向链表中，那么数据库软件可以在从表中获取数据前应用like模式（PostgreSQL不是这样，见下文）。如果like模式没有匹配上，那么将阻止对表的访问。如果表中更多其他列，则仍然需要对表进行访问，以获取这些满足where子句的列-由于使用的是select \*。

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id, ts )  INCLUDE ( eur\_value**, notes** ) |

如果表中有更多其他列，则不会使用仅索引扫描。尽管如此，如果与like模式配置部分行非常少的情况下，使用该索引的性能与仅索引扫描接近。相反的情况，如果所有行都匹配该模式，由于索引大小的增加，性能会稍微下降一些。但是，平衡点很容易实现：为了提高整体性能，通常情况下，类似的过滤器会删除一小部分行。获得收益取决于包含列的大小。

### Include子句上的唯一索引

最后但是同样重要的一点是关于include子句一个完全不同的方面：include子句的唯一索引仅仅考虑了关键列的唯一性。

这允许我们在创建唯一索引的时候，在叶子节点上可以增加额外的列，例如用于快速索引扫描。

|  |
| --- |
| CREATE UNIQUE INDEX …  ON … ( id )  INCLUDE ( payload ) |

这个索引不仅能够保证在id列上值不会重复6，并且下面的查询可以使用该索引进行快速索引扫描。

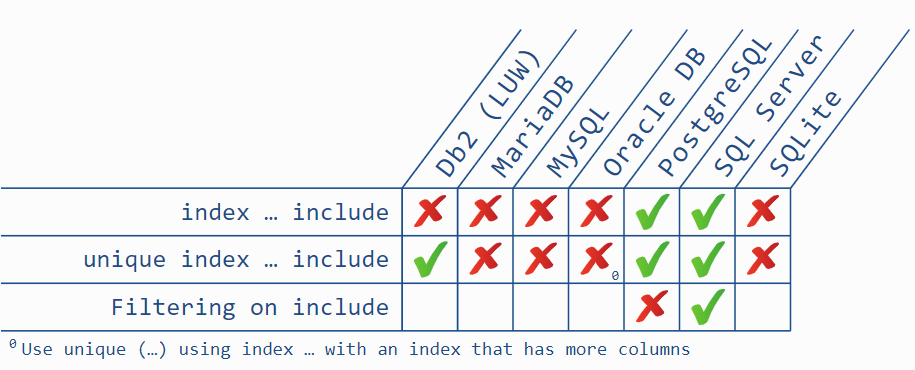
|  |
| --- |
| SELECT payload  FROM …  WHERE id = ? |

这里需要注意的是这个行为不会严格约束include子句：数据库只需要通过一个带有唯一键值列的作为最左边的列—并不关心其他额外列，就可以很好地区分唯一约束和唯一索引了。

对于Oracle数据库，对应的语法格式如下：

|  |
| --- |
| CREATE INDEX …  ON … ( id, payload )  ALTER TABLE … ADD UNIQUE ( id )  USING INDEX … |

### 兼容性



（使用unique (…) using index .. 创建的索引可以包含有多个列）

译者注：参考博客原文，可以清晰看到每款数据库对各语法支持的详细信息。

### PostgreSQL: 可见性检查前不会过滤

PostgreSQL数据库在对索引应用过滤器的时候存在一定的限制。简单的说，除了少数情况，一般情况下数据库不会对索引列进行过滤。更糟糕的情况是，其中一些过滤仅仅对存储在索引中关键部分对应数据才生效，对于include子句是不生效的。这就意味着，移动列到include子句可能会对性能产生负面的影响，即使满足上面描述的逻辑。

长话短说，由于PostgreSQL在表中保留旧行版本，直到对所有的事务不可见之后， vacuum进程在后面的某个时间点会清除它们。为了确定行版本是否可见（对于给定的事务），每张表有两个属性列用于记录改行创建和删除的时间：xmin和xmax。只有当前事务处于xmin/xmax范围内的时候，才能看到该行的信息7。

遗憾的是，索引并不保存有xmin/xmax信息8。

这也就是说当PostgreSQL查看一个索引项的时候，并不知道这个索引对应的元组对当前事务是否可见。它可能是一个已经删除的或者修改未提交的元组。正确做法应该是从表中找到对应的元组，并且检查xmin/xmax的值。

结果就是在PostgreSQL中其实没有仅索引扫描。无论在索引中放入多少列，PostgreSQL总是需要检查可见性，这些信息没有包含在索引中。

当然，PostgreSQL支持仅索引操作，但是它仍然需要通过扫描索引外的数据来检查每个行版本的可见性。不同于检查表，仅索引扫描首先检查可见性地图（visibility map）。这个可见性地图非常密集，因此读操作的次数（希望）小于从表中获取xmin/xmax的情况。然而，这个可见性视图有时并不总能给出明确的答案：可见性视图只能给出该行一定可见或者不确定是否可见。后面一种情况下，快速索引扫描仍然需要从表中获取xmin/xmax（参考explain analyze中的“Heap Fetches”）。

在短暂介绍完可见性话题后，我们将转移到对索引级别的过滤问题上来。

SQL允许在where子句中使用任意复杂的表达式。这些表达式有可能触发运行时错误，比如“除零”。如果PostgreSQL在确认相应元组的可见性之前评估这类表达式，即使不可见的行也可能导致此类错误。为了防止出现这种情况，PostgreSQL通常会在评估此类表达式之前检查元组的可见性。

这个一般规则有一个例外的情况。由于在搜索索引的时候无法检查可见性，因此能用于搜索的操作符必须始终能够安全地使用。这些运算符定义在相应的运算符类中。对于一个从这类运算符中获取的简单比较过滤器，PostgreSQL可以在检查可见性之前使用该过滤器，因为它知道这些操作符是安全地。关键是只有键值列拥有一个与之对应的运算符。Include子句中的列不能应用于过滤器，除非已经确认了它们的可见性。这是我从[PostgreSQL极客邮件列表](https://www.postgresql.org/message-id/flat/7DF52167-4379-4A1E-A957-90D774EBDF21%40winand.at)中得到的信息。

使用前面的索引和查询语句对上面的内容进行演示：

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id, ts )  INCLUDE ( eur\_value**, notes** )  SELECT \*  FROM sales  WHERE subsidiary\_id = ?  AND notes LIKE '%search term%' |

为了简洁，编辑之后的执行计划如下：

|  |
| --- |
| QUERY PLAN  ----------------------------------------------  Index Scan using idx on sales (actual rows=16)  Index Cond: (subsidiary\_id = 1)  **Filter: (notes ~~ '%search term%')**  Rows Removed by Filter: 240  Buffers: shared hit=54 |

查询语句中的like过滤显示在Filter中，而不是在Index Cond。这就意味着它是应用于整个表级别的。另外，对于获取16行元组，命中的共享缓存的数量也相当高。

在位图索引/堆扫描中，这种现象变得更加明显。

|  |
| --- |
| QUERY PLAN  -----------------------------------------------  Bitmap Heap Scan on sales (actual rows=16)  Recheck Cond: (idsubsidiary\_id= 1)  **Filter: (notes ~~ '%search term%')**  Rows Removed by Filter: 240  Heap Blocks: exact=52  Buffers: shared hit=54  -> Bitmap Index Scan on idx (actual **rows=256**)  Index Cond: (subsidiary\_id = 1)  Buffers: shared hit=2 |

在位图索引扫描中不会涉及like过滤器。相反，它返回256行数据，比通过where子句过滤的数据多了16行。

注意，在这种特定场景下，这个不是include列的特性。将include列移动到索引键中仍然会得到同样的结果。

|  |
| --- |
| CREATE INDEX idx  ON sales ( subsidiary\_id, ts**, eur\_value, notes**)  QUERY PLAN  -----------------------------------------------  Bitmap Heap Scan on sales (actual rows=16)  Recheck Cond: (subsidiary\_id = 1)  **Filter: (notes ~~ '%search term%')**  Rows Removed by Filter: 240  Heap Blocks: exact=52  Buffers: shared hit=54  -> Bitmap Index Scan on idx (actual **rows=256**)  Index Cond: (subsidiary\_id = 1)  Buffers: shared hit=2 |

这是因为like运算符不在运算符类中，因此它被认为是不安全的。

如果你使用运算符类中的操作，比如等于，那么执行计划将发生改变。

|  |
| --- |
| SELECT \*  FROM sales  WHERE subsidiary\_id = ?  AND **notes = 'search term'** |

现在位图索引扫描使用where子句中所有条件，只将其余的16行记录传给位图堆扫描。

|  |
| --- |
| QUERY PLAN  ----------------------------------------------  Bitmap Heap Scan on sales (actual rows=16)  Recheck Cond: (subsidiary\_id = 1  AND notes = 'search term')  Heap Blocks: exact=16  Buffers: shared hit=18  -> Bitmap Index Scan on idx (actual **rows=16**)  Index Cond: (subsidiary\_id = 1  AND **notes = 'search term'**)  Buffers: shared hit=2 |

需要注意的是，这要求相应的列在键值列中。如果你将notes列移动到include子句中，那么由于它没有关联的运算类，导致数据库不再认为比较运算是安全的。因此，PostgreSQL将此过滤器应用于表访问，直到检查了可见性。

|  |
| --- |
| QUERY PLAN  -----------------------------------------------  Bitmap Heap Scan on sales (actual rows=16)  Recheck Cond: (id = 1)  Filter: (notes = 'search term')  Rows Removed by Filter: 240  Heap Blocks: exact=52  Buffers: shared hit=54  -> Bitmap Index Scan on idx (actual **rows=256**)  Index Cond: (id = 1)  Buffers: shared hit=2 |

### 脚注

1. 假设每个页面/块有100个索引项
2. 最坏情况：所有感兴趣的行在不同块中，比如：最坏的可能情况是聚类因子。
3. 双向链表可最终可能成为限制因素，例如：在仅索引扫描或者当从表中获取的行数远小于从双向链表中读取的行这种更加通用的场景。
4. Oracle数据库中索引快速完全扫描是一种极端场景，其中只使用了一个结构的一部分：B树的叶节点，即没有双向链表的链接。附带说明：我仍然想知道为什么索引快速全扫描只被Oracle数据库用作快速索引扫描。
5. 原因很简单：插入新行时，树只支持搜索关键列。这意味着即使根据所有列队叶节点排序，也无法直接定位到新行所在位置。数据库需要检查具有相同键值的所有行
6. 主要风险是集群因子变得更坏-特别是在Oracle数据库中。参考“[Automatically Optimized Clustering Factor](https://use-the-index-luke.com/sql/sorting-grouping/indexed-order-by#sb-order-by-clustering-factor)”。另一个风险是双向链表的长度。
7. 除去重复的NULL值，它被唯一约束所接受。参考 [modern-sql.com](https://modern-sql.com/)上的[Null in Unique Constraints](https://modern-sql.com/concept/null" \l "unique)。
8. 这当然是简化的。阅读“[How Postgres Makes Transactions Atomic](https://brandur.org/postgres-atomicity)”了解PostgreSQL中的快照是如何工作的。
9. 通过include增加xmin/xmax的想法是很吸引人的，但是“不支持在系统列上创建索引”。维护索引的xmax值需要每次更新/删除操作的时候需要更新所有的索引信息。

### 原文地址

<https://use-the-index-luke.com/blog/2019-04/include-columns-in-btree-indexes>