链滴

浅谈 MySQL 索引的分类

作者：shuaibing90
原文链接：https://ld246.com/article/1607146186419
来源网站：链滴
许可协议：署名-相同方式共享 4.0 国际 (CC BY-SA 4.0)

```
<p><img src="https://ld246.com/images/img-loading.svg" alt="" data-src="https://b3logfile
com/bing/20191021.jpg?imageView2/1/w/960/h/540/interlace/1/q/100"></p>
<h2 id="什么是索引-">什么是索引？</h2>
<p>索引是辅助存储引擎高效获取数据的一种数据结构。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="MySQL 索引" data-src="http
://b3logfile.com/file/2020/12/solofetchupload4473542009164123409-4dc322ac.png?imageVi
w2/2/interlace/1/format/jpg"></p>
<p>很多人形象的说索引就是数据的目录，便于存储引擎快速的定位数据。</p>
<h2 id="索引的分类">索引的分类</h2>
<p>我们经常从以下几个方面对索引进行分类</p>
<p>从<strong>数据结构的角度</strong>对索引进行分类</p>
<ul>
<li>B+tree</li>
<li>Hash</li>
<li>Full-texts 索引</li>
</ul>
<p>从<strong>物理存储的角度</strong>对索引进行分类</p>
<ul>
<li>聚簇索引</li>
<li>二级索引(辅助索引)</li>
</ul>
<p>从<strong>索引字段特性角度</strong>分类</p>
<ul>
<li>主键索引</li>
<li>唯一索引</li>
<li>普通索引</li>
<li>前缀索引</li>
</ul>
<p>从<strong>组成索引的字段个数角度</strong>分类</p>
<ul>
<li>单列索引</li>
<li>联合索引（复合索引）</li>
</ul>
<h2 id="数据结构角度看索引">数据结构角度看索引</h2>
<p>下表是 MySQL 常见的存储引擎 InnoDB，MyISAM 和 Memory 分别支持的索引类型</p>
<table>
<thead>
<tr>
<th></th>
<th>InnoDB</th>
<th>MyISAM</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>B+tree 索引</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Hash 索引</td>
<td>No</td>
```

```
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Full-text 索引</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
<p>在实际使用中，InnoDB 作为 MySQL 建表时默认的存储引擎</p>
<p>对上表进行横向查看可以了解到，B+tree 是 MySQL 中被存储引擎采用最多的索引类型。</p>
<p>这里浅尝辄止的谈一下 B+tree 与 Hash 和红黑树的区别。</p>
<h3 id="B-tree和B-tree">B+tree 和 B-tree</h3>
<blockquote>
<p>1970 年，R.Bayer 和 E.Mccreight 提出了一种适用于外查找的平衡多叉树——B-树，磁盘管理
统中的目录管理，以及数据库系统中的索引组织多数采用 B-Tree 这种数据结构。 --数据结构 C 语言
第二版 严蔚敏</p>
</blockquote>
<p>B+tree 是 B-Tree 的一个变种。（哦，对了，B-tree 念 B 树，它不叫 B 减树。。。）</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="Btree1" data-src="https://b3
ogfile.com/file/2020/12/solofetchupload4486096145915991899-1e47f442.png?imageView2/2
interlace/1/format/jpg"></p>
<p>B+tree 只在叶子节点存储数据，而 B-tree 非叶子节点也存储数据，对此处有疑问的可以到下面
连接自己插入数据测试一番</p>
<p><a href="https://link.ld246.com/forward?goto=https%3A%2F%2Fwww.cs.usfca.edu%2F
7Egalles%2Fvisualization%2FBTree.html" target="_blank" rel="nofollow ugc">B-tree</a></p

<p><a href="https://link.ld246.com/forward?goto=https%3A%2F%2Fwww.cs.usfca.edu%2F
7Egalles%2Fvisualization%2FBPlusTree.html" target="_blank" rel="nofollow ugc">B+tree</a
</p>
<p>因此，B+tree 单个节点的数量更小，在相同的磁盘 IO 下能查询更多的节点。</p>
<p>另外 B+tree 叶子节点采用单链表链接适合 MySQL 中常见的基于范围的顺序检索场景，而 B-tre
 无法做到这一点。</p>
<h3 id="B-tree和红黑树">B+tree 和红黑树</h3>
<p><img src="https://ld246.com/images/img-loading.svg" alt="Btree1" data-src="https://b3
ogfile.com/file/2020/12/solofetchupload4486096145915991899-1e47f442.png?imageView2/2
interlace/1/format/jpg"></p>
<p>对于有 N 个叶子节点的 B+tree，搜索复杂度为 <strong>O(logdN)</strong> ,<strong>d
指 degree 是指 B+tree 的度</strong>，表示节点允许的最大子节点个数为 d 个，在实际的运用中 d
值是大于 100 的，即使数据达到千万级别时候 B+tree 的高度依然维持在 3-4 左右，保证了 3-4 次
盘 I/O 就能查到目标数据.</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="红黑树" data-src="https://b3
ogfile.com/file/2020/12/solofetchupload3013108488416317591-db56131d.png?imageView2/
/interlace/1/format/jpg"></p>
<p>从上图中可以看出红黑树是二叉树，节点的子节点个数最多为 2 个，意味着其搜索复杂度为 <str
ng>O(logN)</strong>,比 B+ 树高出不少，因此红黑树检索到目标数据所需经理的磁盘 I/O 次数更
。</p>
<h3 id="B-tree索引与Hash表">B+tree 索引与 Hash 表</h3>
<p>范围查询是 MySQL 数据库中常见的场景，而 Hash 表不适合做范围查询，Hash 表更适合做等
查询，另外 Hash 表还存在 Hash 函数选择和 Hash 值冲突等问题。</p>
<p>因为这些原因，B+tree 索引要比 Hash 表索引有更广的适用场景。</p>
<h2 id="物理存储角度看索引">物理存储角度看索引</h2>
```

<p>MySQL 中的两种常用存储引擎对索引的处理方式差别较大。</p>
<h3 id="InnoDB的索引">InnoDB 的索引</h3>
<p>首先看一下 InnoDB 存储引擎中的索引，InnoDB 表的索引按照叶子节点存储的是否
完整表数据分为聚簇索引和二级索引。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="MySQL 索引存储角度分类" da
a-src="https://b3logfile.com/file/2020/12/solofetchupload727395912023305426-2483ea2a.p
g?imageView2/2/interlace/1/format/jpg"></p>
<p>全表数据就是存储在聚簇索引中的。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="MySQL 聚簇索引和二级索引"
ata-src="https://b3logfile.com/file/2020/12/solofetchupload2826468312883612235-e7ef45ce
png?imageView2/2/interlace/1/format/jpg"></p>
<p>聚簇索引以外的其它索引叫做二级索引。</p>
<p>下面结合实际的例子介绍下这两类索引。</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
create table workers
 (
br>   id   int(11)   not nul
 auto_increment comment '员工工号',
    name
varchar(16) not null comm
nt '员工名字',
    sales int(11</s
an>) default null comment '员工销售业绩',
 <sp
n>   primary key (id)
) engine InnoDB
  AUTO_IN
REMENT = 10
  default charset = utf8;
 <s
an>
 insert into worker
(id, name, sales)
 values (1, '江南
, 12744);
 insert into</
pan> workers(id, name, sales)
 values (3
 '今何在', 14082);
 insert
into workers(id, name, sales)
 values (<
pan>7, '路明非', 14738);
 <spa
>insert into workers(id, name, sales)
 va
ues (8, '吕归尘', 7087);
insert into workers(id, name, sales)
 <s
an>values (11, '姬野', 8565)

 insert into workers(id, name, sales)</
pan>
 values (15, '凯撒', <span
8501);
 insert into workers(id,
ame, sales)
 values (20, '绘梨衣'
/span>, 7890);
 </pre>
<p>我们现在自己的测试数据库中创建一个包含销售员信息的测试表 workers</p>
<p>包含 id(主键),name,sales 三个字段，指定表的存储引擎为 InnoDB。</p>
<p>然后插入 8 条数据</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
94511" data-src="https://b3logfile.com/file/2020/12/solofetchupload5196619149879384178-
8b42135.png?imageView2/2/interlace/1/format/jpg"></p>
<p>这个例子当中，workers 表的聚簇索引建立在字段 id 上</p>
<p>为了准确模拟，我们先把主键 id 插入 b+tree 得到下图</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
94849" data-src="https://b3logfile.com/file/2020/12/solofetchupload8277729806873131044
147a10cf.png?imageView2/2/interlace/1/format/jpg"></p>
<p>然后在此图基础上，我画出了高清版。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="workers 表的聚簇索引" data-s
c="https://b3logfile.com/file/2020/12/solofetchupload1624619215807586838-895e621b.png?
mageView2/2/interlace/1/format/jpg"></p>
<p>从图中可以看到，聚簇索引的每个叶子节点存储了一行完整的表数据，叶子节点间采用单向链表

id 列递增连接，可以方便的进行顺序检索。</p>
<p>InnoDB 表要求必须有聚簇索引，默认在主键字段上建立聚簇索引，在没有主键字段的情况下，
的第一个 NOT NULL 的唯一索引将被建立为聚簇索引，在前两者都没有的情况下，InnoDB 将自动
成一个隐式自增 id 列并在此列上创建聚簇索引。</p>
<p>接着来看二级索引。</p>
<p>还以刚才的 workers 表为例</p>
<p>我们在 name 字段上添加二级索引 index_name</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
alter table workers add index index_name(name);</
pan></pre>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
01125" data-src="https://b3logfile.com/file/2020/12/solofetchupload694986917675646832-
108cd57.png?imageView2/2/interlace/1/format/jpg"></p>
<p>同样我们画出了二级索引 index_name 的 B+tree 示意图</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="workers 表的二级索引" data-s
c="https://b3logfile.com/file/2020/12/solofetchupload5342631828174988878-5d89c706.png?
mageView2/2/interlace/1/format/jpg"></p>
<p>图中可以看出二级索引的叶子节点并不存储一行完整的表数据，而是存储了聚簇索引所在列的值
也就是</p>
<p>workers 表中的 id 列的值。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="workers 表的聚簇索引" data-s
c="https://b3logfile.com/file/2020/12/solofetchupload1624619215807586838-895e621b.png?
mageView2/2/interlace/1/format/jpg"></p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="workers 表的二级索引" data-s
c="https://b3logfile.com/file/2020/12/solofetchupload5342631828174988878-5d89c706.png?
mageView2/2/interlace/1/format/jpg"></p>
<p>这两张示意图中 B+tree 的度设置为了 3 ，这也主要是为了方便演示。</p>
<p>实际的 B+tree 索引中，树的度通常会大于 100。</p>
<p>说了聚簇索引和二级索引 肯定要提到回表查询。</p>
<p>由于二级索引的叶子节点不存储完整的表数据，所以当通过二级索引查询到聚簇索引的列值后，
需要回到局促索引也就是表数据本身进一步获取数据。</p>
<p>比如说我们要在 workers 表中查询 名叫吕归尘的人</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
select * from workers where name=
span>'吕归尘';</pre>
<p>这条 sql 通过 name='吕归尘'的条件</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="workers 表的回表查询" data-s
c="https://b3logfile.com/file/2020/12/solofetchupload6371302269673599675-ab808554.png?
mageView2/2/interlace/1/format/jpg"></p>
<p>在二级索引 index_name 中查询到主键 id=8 ,接着带着 id=8 这个条件</p>
<p>进一步回到聚簇索引查询以后才能获取到完整的数据，很显然回表需要额外的 B+tree 搜索过程
必然增大查询耗时。</p>
<p>需要注意的是通过二级索引查询时，回表不是必须的过程,当 Query 的所有字段在二级索引中就
找到时，就不需要回表，MySQL 称此时的二级索引为覆盖索引或称触发了索引覆盖。</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
select id,name from workers where
ame='吕归尘';</pre>
<p>这句 sql 只查询了 id，和 name,二级索引就已经包含了 Query 所以需要的所有字段，就无需回
查询。</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
explain select id,name from workers where<
span> name='吕归尘';</pre>
<p>使用 explain 查看此条 sql 的执行计划</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205

05734" data-src="https://b3logfile.com/file/2020/12/solofetchupload8475403450367018873dfe6e728.png?imageView2/2/interlace/1/format/jpg"></p>
<p>执行计划的 Extra 字段中出现了 Using where;Using index 表明查询触发了索引 index_name 索引覆盖,且对索引做了 where 筛选，这里不需要回表。</p>
<p>下面做对比，查询一下没有索引的</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">explain select id,name,sales from workers where name='吕归尘';</pre>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_2020120513015" data-src="https://b3logfile.com/file/2020/12/solofetchupload15068504694095206022871b30b.png?imageView2/2/interlace/1/format/jpg"></p>
<p>Extra 为 Using Index Condition 表示会先条件过滤索引，过滤完索引后到所有符合索引条件的数据行，随后用 WHERE 子句中的其他条件去过滤这些数据行。Index Condition Pushdown (ICP)是 MySQL 5.6 以上版本中的新特性,是一种在存储引擎层使索引过滤数据的一种优化方式。ICP 开启时的执行计划含有 Using index condition 标示 ，表示优化使用了 ICP 对数据访问进行优化。</p>
<p>如果你对此感兴趣去查阅对应的官方文档和技术博客。</p>
<p>这次我们简化来理解，不考虑 ICP 对数据访问的优化，</p>
<p>当关闭 ICP 时,Index 仅仅是 data access 的一种访问方式，存储引擎通过索引回表获取的数据传递到 MySQL Server 层进行 WHERE 条件过滤。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_2020120520615" data-src="https://b3logfile.com/file/2020/12/solofetchupload61438830782290787890172a07b.png?imageView2/2/interlace/1/format/jpg"></p>
<p>Extra 为 Using where 只是提醒我们 MySQL 将用 where 子句来过滤结集。这个一般发生在 MySQL 服务器，而不是存储引擎层。一般发生在不能走索引扫描的情况下或者索引扫描，但是有些查询条件不在索引当中的情况下。</p>
<p>这里表明没有触发索引覆盖，进行回表查询。</p>
<h3 id="MyISAM的索引">MyISAM 的索引</h3>
<p>说完了 InnoDB 的索引，接下来我们来看 MyISAM 的索引</p>
<p>以 MyISAM 存储引擎存储的表不存在聚簇索引。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="MyISAM 存储引擎下的 workers 表的索引" data-src="https://b3logfile.com/file/2020/12/solofetchupload67770566243243460-2e01d993.png?imageView2/2/interlace/1/format/jpg"></p>
<p>MyISAM 索引 B+tree 示意图</p>
<p>MyISAM 表中的主键索引和非主键索引的结构是一样的，从上图中我们可以看到</p>
<p>他们的叶子节点是不存储表数据的，节点中存放的是表数据的地址，所以 MyISAM 表可以没有键。</p>
<p>MyISAM 表的数据和索引是分开的，是单独存放的。</p>
<p>MyISAM 表中的主键索引和非主键索引的区别仅在于主键索引 B+tree 上的 key 必须符合主键限制，</p>
<p>非主键索引 B+tree 上的 key 只要符合相应字段的特性就可以了。</p>
<h2 id="索引字段特性角度看索引">索引字段特性角度看索引</h2>
<h4 id="主键索引">主键索引</h4>

建立在主键字段上的索引
一张表最多只有一个主键索引
索引列值不允许为 null
通常在创建表的时候一起创建

<h4 id="唯一索引">唯一索引</h4>

建立在 UNIQUE 字段上的索引就是唯一索引
一张表可以有多个唯一索引，索引列值允许为 null


```
<p>我们演示创建索引</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
<span><span>create</span> <span>table</span> persons</span><br> <span>(</span>
br> <span>    id    <span>int</span>(<span>11</span>) <span>not</span> <span>null</
pan> auto_increment comment <span>'主键id'</span>,</span><br> <span>     eno   <span>
nt</span>(<span>11</span>) comment <span>'工号'</span>,</span><br> <span>     eid
<span>int</span>(<span>11</span>) comment <span>'身份证号'</span>,</span><br> <
pan>     veid <span>int</span>(<span>11</span>) comment <span>'虚拟身份证号'</span>
</span><br> <span>     name <span>varchar</span>(<span>16</span>) comment <span>
名字'</span>,</span><br> <span>     primary key (id) comment <span>'主键索引'</span>,<
span><br> <span>     UNIQUE key (eno) comment <span>'eno唯一索引'</span>,</span><b
> <span>     UNIQUE key (eid) comment <span>'eid唯一索引'</span></span><br> <span>)
ngine = InnoDB</span><br> <span>  auto_increment = <span>1000</span></span><br>
<span>  default charset = utf8;</span><br> <span><span></span></span><br> <span>
span>alter</span> <span>table</span> persons</span><br> <span>     add unique index
ndex_veid (veid) comment <span>'veid唯一索引'</span>;</span></pre>
<p>通过 show index from persons;命令我们看到已经成功创建了三个唯一索引。</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
25114" data-src="https://b3logfile.com/file/2020/12/solofetchupload1566419295529986l4-e
d9bda9.png?imageView2/2/interlace/1/format/jpg"></p>
<h4 id="普通索引">普通索引</h4>
<p>主键索引和唯一索引对字段的要求是要求字段为主键或 unique 字段，</p>
<p>而那些建立在普通字段上的索引叫做普通索引，既不要求字段为主键也不要求字段为 unique。<
p>
<h4 id="前缀索引">前缀索引</h4>
<p>前缀索引是指对字符类型字段的前几个字符或对二进制类型字段的前几个 bytes 建立的索引，而
是在整个字段上建索引。</p>
<p>例如，可以对 persons 表中的 name(varchar(16))字段 中 name 的前 5 个字符建立索引。</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
<span><span>create</span> index index_name <span>on</span> persons (name(<span>
</span>)) comment <span>'前缀索引'</span>;</span><br> <span>show index <span>fro
</span> persons;</span></pre>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
25836" data-src="https://b3logfile.com/file/2020/12/solofetchupload3506859226087613750-
cfc2540.png?imageView2/2/interlace/1/format/jpg"></p>
<p>前缀索引可以建立在类型为</p>
<ul>
<li>char</li>
<li>varchar</li>
<li>binary</li>
<li>varbinary</li>
</ul>
<p>的列上，可以大大减少索引占用的存储空间，也能提升索引的查询效率。</p>
<h2 id="索引列的个数角度看索引"><strong>索引列的个数角度</strong>看索引</h2>
<ul>
<li>建立在单个列上的索引为单列索引
<ul>
<li>上文演示的都是单列索引</li>
</ul>
</li>
<li>建立在多列上的称为联合索引（复合索引）</li>
</ul>
<p>演示一下联合索引</p>
<pre class="md-fences md-end-block md-fences-with-lineno ty-contain-cm modeLoaded">
```

create index index_id_name on workers(id,name) co
ment '组合索引';</pre>
<p>这条语句在我们演示表 workers 中建立 id，name 这两个字段的联合索引。</p>
<p>借助 show index 命令查看索引的详细信息 操作后结果如下：</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="截图录屏_选择区域_20201205
30730" data-src="https://b3logfile.com/file/2020/12/solofetchupload3032431177145337696
d861ff6a.png?imageView2/2/interlace/1/format/jpg"></p>
<p>虽然详细信息当中列出了两条关于联合索引的条目，但并不表示联合索引是建立了多个索引，联
索引是一个索引结构，这两个条目表示的是组合索引中字段的具体信息，按建立索引时的书写顺序排
。</p>
<p>同样我们来看下联合索引的 B+tree 示意图</p>
<p><img src="https://ld246.com/images/img-loading.svg" alt="联合索引的 Btree 加" data-sr
="https://b3logfile.com/file/2020/12/solofetchupload6130596889042124173-3f0e4671.png?
mageView2/2/interlace/1/format/jpg"></p>
<p>从图中看到</p>
<p>组合索引的非叶子节点保存了两个字段的值作为 B+tree 的 key 值，当 B+tree 上插入数据时，
按字段 id 比较，在 id 相同的情况下按 name 字段比较。</p>

原文链接：浅谈 MySQL 索引的分类