一、索引

索引的本质

MySQL官方对索引的定义为：索引（Index）是帮助MySQL高效获取数据的数据结构。提取句子主干，就可以得到**索引的本质：索引是数据结构**。

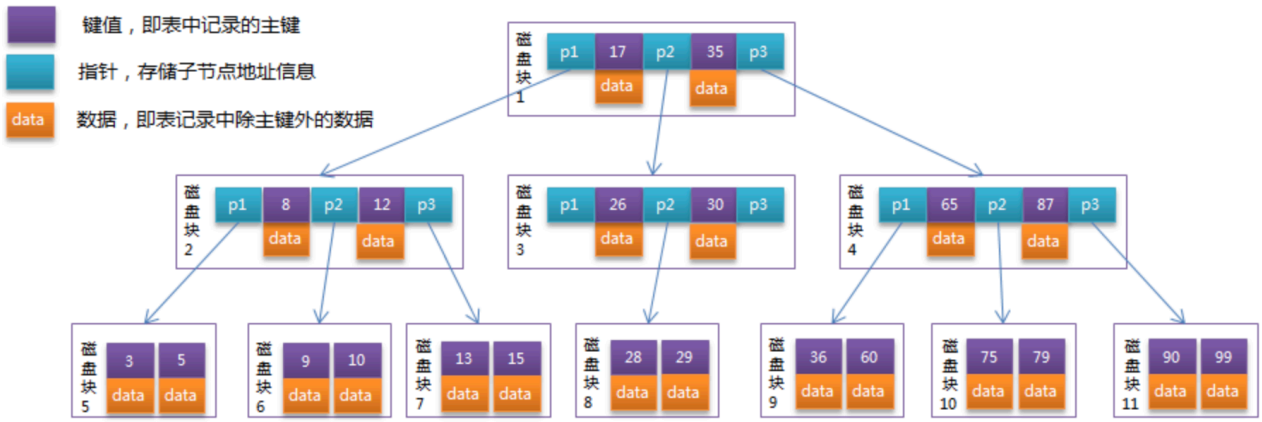
每种查找算法都只能应用于特定的数据结构之上，例如二分查找要求被检索数据有序，而二叉树查找只能应用于二叉查找树上，**但是数据本身的组织结构不可能完全满足各种数据结构**（例如，理论上不可能同时将两列都按顺序进行组织）。

所以，**在数据之外，数据库系统还维护着满足特定查找算法的数据结构**，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

索引本身也很大，不可能全部存储在内存中，因此**索引往往以索引文件的形式存储的磁盘上**。

MySQL的InnoDB存储引擎在设计时**是将索引根节点常驻内存的**，也就是说查找某一键值的行记录时最多只需要1~3次磁盘I/O操作。

B-Tree



B-Tree结构的数据可以让系统高效的找到数据所在的磁盘块。为了描述B-Tree，首先定义一条记录为一个二元组[key, data] ，**key为记录的键值，对应表中的主键值，data为一行记录中除主键外的数据**。对于不同的记录，key值互不相同。

一棵**m阶**的B-Tree有如下特性：

1. 每个节点**最多有m个孩子**。
2. **除了根节点和叶子节点外，其它每个节点至少有Ceil(m/2)个孩子**。
3. **若根节点不是叶子节点，则至少有2个孩子**
4. **所有叶子节点都在同一层**，且不包含其它关键字信息，具有相同的深度，等于树高h。
5. 每个非终端节点包含n个关键字信息（P0,P1,...Pn, k1,...kn）
6. 关键字的个数n满足：ceil(m/2)-1 <= n <= m-1
7. ki(i=1,...n)为关键字，且**关键字非递减排序**。
8. Pi(i=1,...n)为指向子树根节点的指针。P(i-1)指向的子树的所有节点关键字均小于等于ki，但都大于等于k(i-1)

B-Tree的查找

首先从根节点进行**二分查找**，如果找到则返回对应节点的data，否则对相应区间的指针指向的节点递归进行查找，直到找到节点或找到null指针，前者查找成功，后者查找失败。

```
BTree_Search(node, key) {
    if(node == null) return null;
    foreach(node.key)
    {
        if(node.key[i] == key) return node.data[i];
            if(node.key[i] > key) return BTree_Search(point[i]->node);
    }
    return BTree_Search(point[i+1]->node);
}
data = BTree_Search(root, my_key);
```

B+ 树原理

1. 数据结构

B Tree 指的是 Balance Tree，也就是**平衡树**。平衡树是一颗查找树，**并且所有叶子节点位于同一层**。

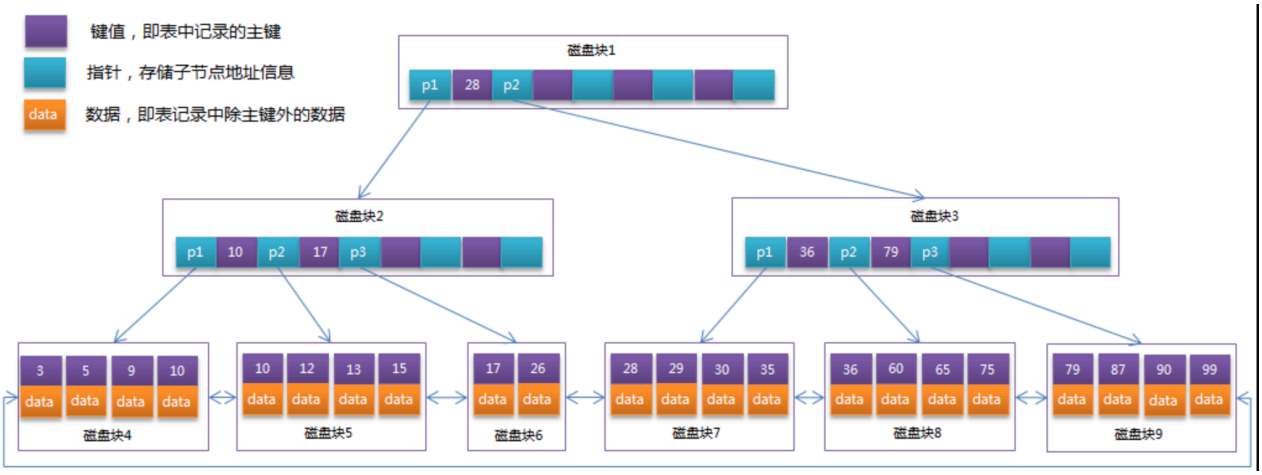
从上一节中的B-Tree结构图中可以看到每个节点中不仅包含数据的key值，还有data值。而每一个页的存储空间是有限的，如果**data数据较大时将会导致每个节点（即一个页）能存储的key的数量很小，当存储的数据量很大时同样会导致B-Tree的深度较大，增大查询时的磁盘I/O次数，进而影响查询效率。**在B+Tree中，所有数据记录节点都是按照键值大小顺序存放在同一层的叶子节点上，而非叶子节点上只存储key值信息，这样可以大大加大每个节点存储的key值数量，降低B+Tree的高度。

B+Tree相对于B-Tree区别：

1. **非叶子节点只存储键值信息。**
2. **所有叶子节点之间都有一个链指针。**
3. **数据记录都存放在叶子节点中。**

B+ Tree 是基于 B Tree 和**叶子节点顺序访问指针进行实现**，它具有 B Tree 的平衡性，并且通过顺序访问指针来提高**区间查询**的性能。

在 B+ Tree 中，一个节点中的 key **从左到右非递减排列**，如果某个指针的左右相邻 key 分别是 keyi 和 keyi+1，且不为 null，则该指针指向节点的所有 key 大于等于 keyi 且小于等于 keyi+1。



2. B+Tree的查找

通常在B+Tree上有两个头指针，一个指向根节点，另一个指向关键字最小的叶子节点，而且所有叶子节点（即数据节点）之间是一种链式环结构。

因此可以对B+Tree进行两种查找运算：

- 对于**主键的范围查找和分页查找**，
- **从根节点开始，进行随机查找**。

3. 局部性原理与磁盘预读

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：

当一个数据被用到时，其附近的数据也通常会马上被使用。

程序运行期间所需要的数据通常比较集中。

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度**一般为页（page）的整倍数**。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，**每个存储块称为一页**（在许多操作系统中，**页的大小通常为4k**），主存和磁盘**以页为单位交换数据**。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

4. B+Tree索引的性能分析

InnoDB存**储引擎中页的大小为16KB**，而系统一个磁盘块的存储空间往往没有这么大，因此InnoDB每次申请磁盘空间时都会是**若干地址连续磁盘块**来达到页的大小16KB。一般表的**主键类型为INT**（占用4个字节）或**BIGINT**（占用8个字节），**指针类型也一般为4或8个字节，也就是说一个页（B+Tree中的一个节点）**中大概存储16KB/(8B+8B)=1K个键值（因为是估值，为方便计算，这里的K取值为〖10〗^3）。也就是说一个深度为3的B+Tree索引可以维护10^3 * 10^3 * 10^3 = 10亿 条记录。

5. B+Tree索引与其他数据结构索引的比较

1. 和红黑树索引的比较

红黑树等平衡树也可以用来实现索引，但是文件系统及数据库系统普遍采用 B+ Tree 作为索引结构，主要有以下两个原因：

（一）**更少的查找次数**

平衡树查找操作的时间复杂度和**树高 h 相关，O(h)=O(logdN)，其中 d 为每个节点的出度**。

红黑树的出度为 2，而 B+ Tree 的出度一般都非常大，所以红黑树的树高 h 很明显比 B+ Tree 大非常多，查找的次数也就更多。

（二）**利用磁盘预读特性**

为了减少磁盘 I/O 操作，磁盘往往不是严格按需读取，而是每次都会预读。预读过程中，磁盘进行顺序读取，顺序读取不需要进行磁盘寻道，并且只需要很短的旋转时间，速度会非常快。

2. 和哈希索引的比较

（一）**支持范围查询、排序和分组（而这些哈希索引都不行，哈希索引只能精准查询）**

（二）**对于I/O密集型应用，很好的利用了磁盘预读特性**（不可能把所有数据都在内存建立哈希，而一旦数据不在内存就会触发多次随机I/O）

6. InnoDB和MyISAM索引对比

「InnoDB引擎」中的B+Tree索引分类

聚簇索引和数据在磁盘上存在一起，找到索引就找到了数据

聚集索引（clustered index）：叶子节点 data 域记录着**完整的数据记录，因为无法把数据行存放在两个不同的地方，所以一个表只能有一个聚簇索引**

辅助索引（二级索引）（secondary index）：叶子节点的 data 域记录着**主键的值**，因此在使用辅助索引进行查找时，需要先查找到主键值，然后再到聚簇索引中进行查找。

为什么InnoDB中二级索引叶子节点不和MyISAM一样存记录的物理块指针?

这减小了移动数据或者数据页面分裂时维护二级索引的开销，因为InnoDB不需要更新索引的行指针

> **术语 "聚簇"**：表示数据行和相邻的键值紧凑地存储在一起
>
> 聚簇索引本身已经是**按照物理顺序放置的**，排序很快（即：只要索引是相邻的，那么对应的数据一定也是相邻地存放在磁盘上的）。非聚簇索引则没有按序存放，需要额外消耗资源来排序。
>
> **聚簇索引主键的插入速度要比非聚簇索引主键的插入速度慢很多（为了保证按索引排序顺序存放记录，可能会移动多条记录）**
>
> **范围查询时**，用聚簇索引也比用非聚簇索引好。

InnoDB聚集索引优点

- **可以把相关数据保存在一起。**例如实现电子邮箱时，可以**根据用户ID来聚集数据**，这样只需要从磁盘读取少数的数据页就能获取某个用户的全部邮件。**如果没有聚簇索引，则每封邮件都可能多一次磁盘IO**。
- **数据访问更快。**聚簇索引将索引和数据保存在同一个B+Tree中，因此从聚簇索引中获取数据通常比在非聚簇索引中查找要快。

- 使用覆盖索引扫描的查询可以直接使用页节点中的主键值。

InnoDB聚集索引缺点

- 聚簇数据最大限度地**提高了IO密集型应用的性能，但如果数据全部放在内存中，则访问的顺序就没那么重要了**，聚簇索引也就没什么优势了。
- **插入速度严重依赖于插入顺序**。
- **更新聚簇索引列的代价很高，**因为会强制InnoDB将每个被更新的行移动到新的位置。
- 基于聚簇索引的表插入新行，或者主键被更新导致需要移动行的时候，可能面临"**页分裂（page split)**"的问题。
- 聚簇索引**可能导致全表扫描变慢**，尤其是**行比较稀疏**，或者由于页分裂导致数据存储不连续的时候。
- 二级索引（非聚簇索引）可能比想象的要更大，因为在二给索引的叶子节点包含了引用行的主键列。
- 二级索引访问需要两次索引查找，而不是一次。

InnoDB聚集索引使用注意

最好避免随机的聚簇索引，特别对于I/O密集型的应用。例如，从性能的角度考虑，使用UUID作为聚簇索引会很糟糕：它使得聚簇索引的插入变得完全随机，这是最坏的情况，使得数据没有任何聚集特性。

如果正在使用InnoDB表并且没有什么数据需要聚集，那么可以定义一个代理键作为主键，这种主键的数据应该和应用无关，最简单的方法是**使用auto_increment自增列**。这样可以**保证数据行是按照顺序写入**，对于根据主键做关联操作的性能也会更好。

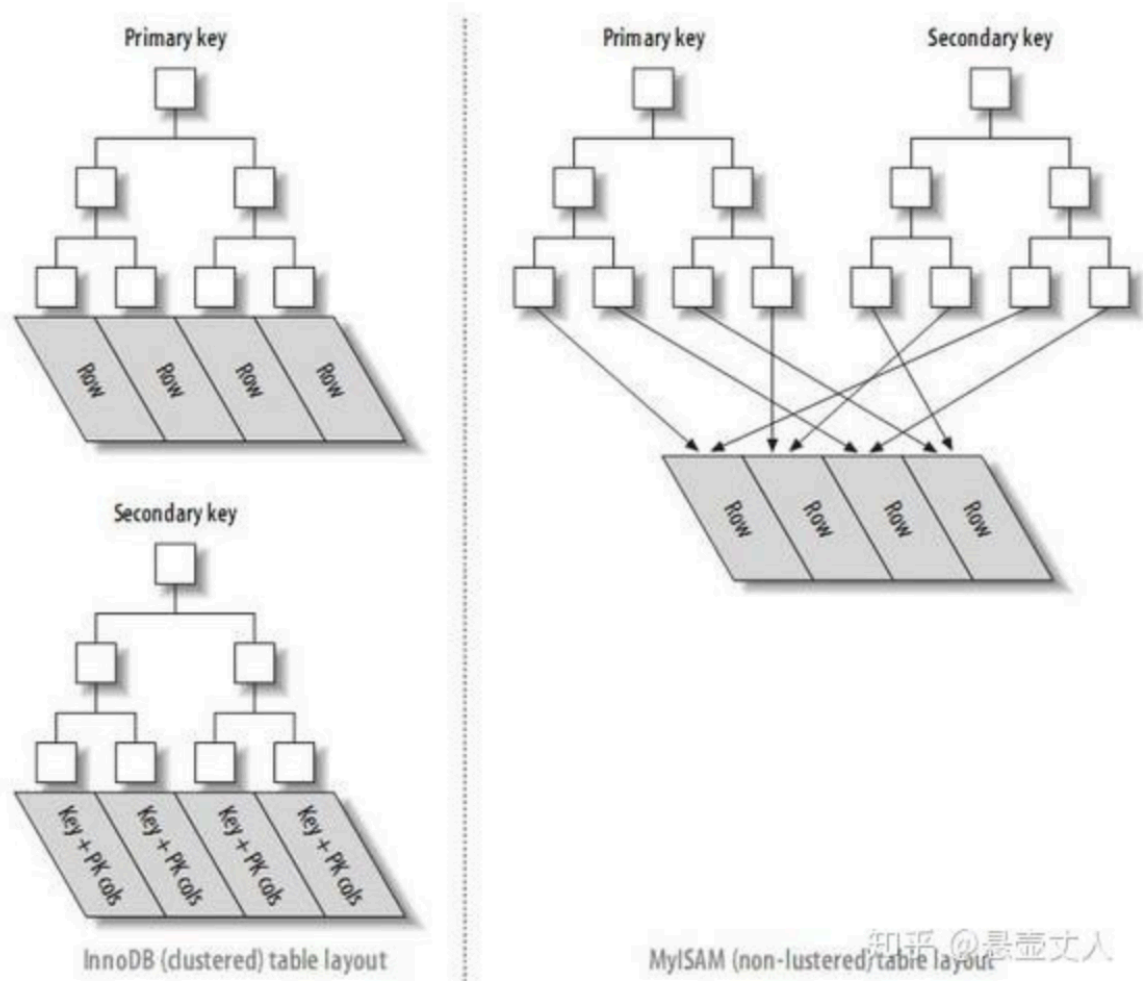
MyISAM中二级索引（非聚簇索引）

MyISAM**按照数据插入的顺序存储在磁盘上**，索引和数据是分开存储存储的

通过key_buffer把索引先缓存到内存中，当需要访问数据时（通过索引访问数据），在内存中直接搜索索引，然后通过索引找到磁盘相应数据，**这也就是为什么索引不在key buffer命中时，速度慢的原因**。

叶子节点中保存的实际上是**指向存放数据的物理块的指针（记录的行号，注意和InnoDB辅助索引的不同！）**

> **MYISAM的主键索引和二级索引没有任何区别**，主键索引仅仅只是一个叫做PRIMARY的唯一、非空的索引，且**MYISAM引擎中可以不设主键**。



MySQL索引的分类（根据数据结构）

索引是在**存储引擎层实现的**，而不是在服务器层实现的，所以**不同存储引擎具有不同的索引类型和实现**。

1. B+Tree索引（存储到磁盘）

是大多数 MySQL 存储引擎的默认索引类型。

因为不再需要进行全表扫描，只需要对树进行搜索即可，所以查找速度快很多。

除了用于查找，**还可以用于排序和分组**。

可以指定多个列作为索引列，多个索引列共同组成键。

适用于全键值、键值范围和键前缀查找，其中键前缀查找只适用于**最左前缀查找。如果不是按照索引列的顺序进行查找，则无法使用索引**。

2. 哈希索引（HASH INDEX，内存索引，不存储到磁盘）

哈希索引能以 O(1) 时间进行查找，但是**失去了有序性**：

- **无法用于排序与分组**；
- 只支持精确查找，**无法用于部分查找和范围查找**。

InnoDB 存储引擎有一个特殊的功能叫“**自适应哈希索引**”，当某个索引值被使用的非常频繁时，会在 B+Tree 索引之上再创建一个哈希索引，这样就让 B+Tree 索引具有哈希索引的一些优点，**比如快速的哈希查找**。但是要注意，**B+索引比Hash索引的优势在于，「范围查询」，以及「进行磁盘I/O操作」（因为不可能把所有数据都在内存中建立Hash）**

3. 全文索引（FULL TEXT INDEX）

1. MySQL 5.6 以前的版本，只有 MyISAM 存储引擎支持全文索引；
2. MySQL 5.6 及以后的版本，MyISAM 和 InnoDB 存储引擎均支持全文索引;
3. 只有字段的数据类型为 **char**、**varchar**、**text 及其系列**才可以建全文索引。

用于查找**文本中的关键词，而不是直接比较是否相等**。

查找条件使用 MATCH AGAINST，而不是普通的 WHERE。

```
select * from fulltext_test
    where match(content,tag) against('xxx xxx');
```

match() 函数中**指定的列必须和全文索引中指定的列完全相同**，否则就会报错，无法使用全文索引，这是因为全文索引不会记录关键字来自哪一列。如果想要对某一列使用全文索引，请单独为该列创建全文索引。

全文索引使用**倒排索引**实现，它记录着**关键词到其所在文档的映射**。

4. 空间数据索引（R-Tree）

MyISAM 存储引擎支持**空间数据索引（R-Tree）**，可以用于**地理数据存储。和B-Tree索引不同，这类索引「无须前缀查询」**。空间数据索引会从**所有维度**来索引数据，可以有效地使用任意维度来进行组合查询。

必须使用 GIS 相关的函数来维护数据。

MySQL索引分类（根据逻辑）

1. PRIMARY KEY 主键索引

主键是一种唯一性索引，每个表只能有一个主键，**在单表查询中**，PRIMARY主键索引与UNIQUE唯一索引的检索效率并没有多大的区别，**但在关联查询中，PRIMARY主键索引的检索速度要高于UNIQUE唯一索引。**

2. INDEX 普通索引

这是最基本的索引类型，而且它没有唯一性之类的限制。s

3. UNIQUE 唯一索引

这种索引和前面的“普通索引”基本相同，但有一个区别：**索引列的所有值都只能出现一次，即必须唯一**。

4. 多列索引（组合、复合索引）

复合索引指多个字段上创建的索引，只有在查询条件中使用了创建索引时的第一个字段，索引才会被使用。使用复合索引时遵循**「最左前缀原则」**

最左前缀原则

顾名思义是最左优先，以最左边的为起点**任何连续的索引都能匹配上**， 注：**如果第一个字段是范围查询需要单独建一个索引** 注：在创建联合索引时，要根据业务需求，where子句中使用最频繁的一列放在最左边。这样的话扩展性较好，比如 userid 经常需要作为查询条件，而 mobile 不常常用，则需要把 userid 放在联合索引的第一位置，即最左边

当创建(a,b,c)联合索引时，相当于创建了(a)单列索引，(a,b)联合索引以及(a,b,c)联合索引 想要索引生效的话,只能使用 a和a,b和a,b,c三种组合；当然，我们上面测试过，**a,c组合也可以，但实际上只用到了a的索引，c并没有用到！**

索引优化

1. 独立的列

在进行查询时，**索引列不能是表达式的一部分，也不能是函数的参数**，否则无法使用索引。

例如下面的查询不能使用 actor_id 列的索引：

```
SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;
```

2. 多列索引

在需要使用**多个列作为条件进行查询时**，使用多列索引比使用多个单列索引性能更好。例如下面的语句中，最好把 actor_id 和 film_id 设置为多列索引。

```
SELECT film_id, actor_ id FROM sakila.film_actor
WHERE actor_id = 1 AND film_id = 1;
```

3. 索引列的顺序

让「选择性最强」的索引列放在前面。

索引的选择性是指：**不重复的索引值和记录总数的比值**。最大值为 1，此时每个记录都有唯一的索引与其对应。选择性越高，查询效率也越高。

例如下面显示的结果中 customer_id 的选择性比 staff_id 更高，因此最好把 customer_id 列**放在多列索引的前面**。

```
SELECT COUNT(DISTINCT staff_id)/COUNT(*) AS staff_id_selectivity,
COUNT(DISTINCT customer_id)/COUNT(*) AS customer_id_selectivity,
COUNT(*)
FROM payment;
```

```
   staff_id_selectivity: 0.0001
customer_id_selectivity: 0.0373
               COUNT(*): 16049
```

4. 前缀索引

对于 BLOB、TEXT 和 VARCHAR 类型的列，必须使用**前缀索引**，只索引开始的部分字符。

对于前缀**长度的选取需要根据索引选择性来确定**。

5. 覆盖索引

索引包含所有需要查询的字段的值。

具有以下优点：

- 索引通常远小于数据行的大小，只读取索引能大大减少数据访问量。
- **一些存储引擎（例如 MyISAM）在内存中只缓存索引**，而数据依赖于操作系统来缓存。因此，只访问索引可以不使用系统调用（通常比较费时）。
- 对于 InnoDB 引擎，**若辅助索引能够覆盖查询**，则无需访问主索引。

索引的优点

- 大大减少了服务器需要扫描的数据行数。
- **帮助服务器避免进行排序和分组，以及避免创建临时表（B+Tree 索引是有序的，可以用于 ORDER BY 和 GROUP BY 操作**。临时表主要是在排序和分组过程中创建，因为不需要排序和分组，也就不需要创建临时表）。
- **将随机 I/O 变为顺序 I/O**（B+Tree 索引是有序的，会将相邻的数据都存储在一起）。

索引的使用条件

- 对于**非常小的表、「大部分情况下简单的全表扫描比建立索引更高效」**；
- 对于中到大型的表，索引就非常有效；
- 但是对于特大型的表，建立和维护索引的代价将会随之增长。这种情况下，**需要用到一种技术可以直接区分出需要查询的一组数据**，而不是一条记录一条记录地匹配，**例如可以使用分区技术**。

二、查询性能优化

使用Explain进行分析

Explain 用来分析 SELECT 查询语句，**开发人员可以通过分析 Explain 结果来优化查询语句**。

比较重要的字段有：

- select_type : 查询类型，有简单查询、联合查询、子查询等
- key : 使用的索引
- rows : 扫描的行数

优化数据访问

1. 减少请求的数据量

- 只返回必要的列：最好不要使用 SELECT * 语句。
- 只返回必要的行：使用 LIMIT 语句来限制返回的数据。
- **缓存重复查询的数据**：使用缓存可以避免在数据库中进行查询，特别在要查询的数据经常被重复查询时，缓存带来的查询性能提升将会是非常明显的。

2. 减少服务端扫描的行数

最有效的方式是使用索引来**覆盖查询**。

重构查询方式

1. 切分大查询

一个大查询如果一次性执行的话，可能一次锁住很多数据、占满整个事务日志、耗尽系统资源、阻塞很多小的但重要的查询。

```
DELETE FROM messages WHERE create < DATE_SUB(NOW(), INTERVAL 3 MONTH);
```

```
rows_affected = 0
do {
    rows_affected = do_query(
    "DELETE FROM messages WHERE create  < DATE_SUB(NOW(), INTERVAL 3 MONTH)
LIMIT 10000")
} while rows_affected > 0
```

2. 分解大连接查询

将一个大连接查询分解成对每一个表进行一次单表查询，然后在应用程序中进行关联，这样做的好处有：

- **让缓存更高效**。对于连接查询，如果其中一个表发生变化，那么整个查询缓存就无法使用。而分解后的多个查询，即使其中一个表发生变化，对其它表的查询缓存依然可以使用。
- **分解成多个单表查询**，这些单表查询的缓存结果更可能被其它查询使用到，从而减少冗余记录的查询。
- **减少锁竞争**；
- 在应用层进行连接，可以更容易对数据库进行拆分，从而更容易做到高性能和可伸缩。
- 查询本身效率也可能会有所提升。例如下面的例子中，使用 IN() 代替连接查询，可以让 MySQL 按照 ID 顺序进行查询，这可能比随机的连接要更高效。

```
SELECT * FROM tab
JOIN tag_post ON tag_post.tag_id=tag.id
JOIN post ON tag_post.post_id=post.id
WHERE tag.tag='mysql';
```

```
SELECT * FROM tag WHERE tag='mysql';
SELECT * FROM tag_post WHERE tag_id=1234;
SELECT * FROM post WHERE post.id IN (123,456,567,9098,8904);
```

三、存储引擎

InnoDB

是 MySQL **默认的事务型（ACID）存储引擎**，只有在需要它不支持的特性时，才考虑使用其它存储引擎。

实现了**四个标准的隔离级别**，默认级别是**可重复读**（REPEATABLE READ）。在可重复读隔离级别下，通**过多版本并发控制（MVCC）+ 间隙锁（Next-Key Locking）防止幻影读**。

主索引是聚簇索引，在索引中保存了数据，从而避免直接读取磁盘，因此对查询性能有很大的提升。

内部做了很多优化，包括**从磁盘读取数据时采用的可预测性读**、能够加快读操作并且**自动创建的自适应哈希索引**、能够加速插入操作的插入缓冲区等。

支持真正的在线热备份。其它存储引擎不支持在线热备份，要获取一致性视图需要停止对所有表的写入，而在读写混合场景中，停止写入可能也意味着停止读取。

使用 **InnoDB存储引擎** MySQL将在数据目录下创建一个名为 `ibdata1` 的10MB大小的自动扩展数据文件，以及两个名为 `ib_logfile0` 和 `ib_logfile1` 的5MB大小的日志文件（**InnoDB用日志来保证持久性，崩溃恢复**）

使用场景

- **更新密集的表**。InnoDB存储引擎特别适合处理多重并发的更新请求。
- **事务**。InnoDB存储引擎是支持事务的标准MySQL存储引擎。
- **自动灾难恢复**。与其它存储引擎不同，InnoDB表能够自动从灾难中恢复。
- **外键约束**。MySQL支持外键的存储引擎只有InnoDB。
- **支持自动增加列AUTO_INCREMENT属性**。 一般来说，如果需要事务支持，并且**有较高的并发读取频率**，InnoDB是不错的选择。

MyISAM（已经被官方放弃）

设计简单，数据以紧密格式存储。**对于只读数据，或者表比较小、可以容忍修复操作，则依然可以使用它**。

提供了大量的特性，包括压缩表、空间数据索引等。

不支持事务。

不支持行级锁，只能对整张表加锁，读取时会对需要读到的所有表加**共享锁，写入时**则对表加**排它锁**。但在表**有读取操作的同时，也可以往表中插入新的记录**，这被称为**并发插入（CONCURRENT INSERT）**。

可以手工或者自动执行检查和修复操作，但是和事务恢复以及崩溃恢复不同，可能导致一些数据丢失，而且**修复操作是非常慢的**。

如果指定了 **「DELAY_KEY_WRITE」** 选项，**在每次修改执行完成时，不会立即将修改的索引数据写入磁盘，而是会写到内存中的键缓冲区**，只有在**清理缓冲区或者关闭表的时候才会将对应的索引块写入磁盘**。这种方式可以极大的提升写入性能，**但是在数据库或者主机崩溃时会造成索引损坏，需要执行修复操作**。

使用场景

Web、数据仓储和其他应用环境下最常使用的存储引擎之一

拥有较高的插入、查询速度

不支持事务

- **选择密集型的表**。MyISAM存储引擎在筛选大量数据时非常迅速，这是它最突出的优点。
- **插入密集型的表**。MyISAM的并发插入特性允许**同时选择和插入数据（并发插入）**。例如：MyISAM存储引擎很适合**管理邮件或Web服务器日志数据**。

为什么MyISAM会比Innodb的查询速度快?

InnoDB在做SELECT的时候，要维护的东西比MyISAM引擎多很多:

1）**数据块，InnoDB要缓存，MyISAM只缓存索引块**，这中间还有换进换出的减少；

2）InnoDB**寻址要映射到块，再到行**，MyISAM记录的直接是文件的OFFSET，定位比InnoDB要快

3）InnoDB还**需要维护MVCC一致**；虽然你的场景没有，但他还是需要去检查和维护MVCC (Multi-Version Concurrency Control)多版本并发控制

MEMORY

MEMORY存储引擎将**表中的数据存储到内存中**，为查询和引用其他表数**据提供快速访问**。

比较

- 事务：InnoDB **是事务型的**，可以使用 Commit 和 Rollback 语句。
- 并发：MyISAM **只支持表级锁**，而 InnoDB **还支持行级锁**。
- 外键：InnoDB **支持外键**。
- 备份：InnoDB **支持在线热备份**。
- 崩溃恢复：MyISAM **崩溃后发生损坏的概率比 InnoDB 高很多，而且恢复的速度也更慢**。
- 其它特性：MyISAM **支持压缩表和空间数据索引**。

存储引擎的选择

在实际工作中，选择一个合适的存储引擎是一个比较复杂的问题。每种存储引擎都有自己的优缺点，不能笼统地说谁比谁好。但建议选择使用**InnoDB**

存储引擎的对比

特　　性	InnoDB	MyISAM	MEMORY
事务安全	支持	无	无
存储限制	64TB	有	有
空间使用	高	低	低
内存使用	高	低	高
插入数据的速度	低	高	高
对外键的支持	支持	无	无

InnoDB： 支持**事务处理**，支持**外键**，支持**崩溃修复**能力和**并发控制**。如果需要对事务的**完整性要求比较高**（比如银行），要求实现并发控制（比如售票），那选择InnoDB有很大的优势。如果需要**频繁的更新、删除操作的数据库**，也可以选择InnoDB，**因为支持事务的提交（commit）和回滚（rollback）**。

MyISAM： 插入数据快，空间和内存使用比较低。如果表**主要是用于插入新记录和读出记录**，那么选择MyISAM能实现处理高效率。如果应用的完整性、并发性要求比较低，也可以使用。

MEMORY： 所有的数据都在内存中，数据的处理速度快，但是**安全性不高**。如果需要**很快的读写速度**，对数据的**安全性要求较低**，可以选择MEMOEY。它对表的大小有要求，**不能建立太大的表**。所以，这类数据库只使用在相对较小的数据库表。

注意：同一个数据库也可以使用多种存储引擎的表。如果一个表要求**比较高的事务处理**，可以选择InnoDB。这个数据库中可以将**查询要求比较高**的表选择MyISAM存储。如果该数据库需要一个**用于查询的临时表**，可以选择MEMORY存储引擎。

四、数据类型

整型

TINYINT, SMALLINT, MEDIUMINT, INT, BIGINT 分别使用 8, 16, 24, 32, 64 位存储空间，一般情况下越小的列越好。

INT(11) 中的**数字只是规定了交互工具显示字符的个数，对于存储和计算来说是没有意义的**。

浮点数

FLOAT 和 DOUBLE 为浮点类型，**DECIMAL 为高精度小数类型**。CPU 原生支持浮点运算，但是不支持 DECIMAl 类型的计算，**因此 DECIMAL 的计算比浮点类型需要更高的代价**。

FLOAT、DOUBLE 和 DECIMAL **都可以指定列宽**，例如 DECIMAL(18, 9) 表示总共 18 位，取 9 位存储小数部分，剩下 9 位存储整数部分。

字符串

主要有 CHAR 和 VARCHAR 两种类型，一种是定长的，一种是变长的。

VARCHAR 这种变长类型能够节省空间，因为只需要存储必要的内容。**但是在执行 UPDATE 时可能会使行变得比原来**长，当超出一个页所能容纳的大小时，就要执行额外的操作。MyISAM 会将行拆成不同的片段存储，而 InnoDB 则需要分裂页来使行放进页内。

在进行存储和检索时，会保留 VARCHAR 末尾的空格，而会删除 CHAR 末尾的空格。

时间和日期

MySQL 提供了两种相似的日期时间类型：DATETIME 和 TIMESTAMP。

1. DATATIME

能够保存从 1001 年到 9999 年的日期和时间，精度为秒，使用 **8 字节**的存储空间。

它与时区无关。

默认情况下，MySQL 以一种可排序的、无歧义的格式显示 DATETIME 值，例如“2008-01-16 22:37:08”，这是 ANSI 标准定义的日期和时间表示方法。

2. TIMESTAMP

和 UNIX 时间戳相同，保存从 1970 年 1 月 1 日午夜（格林威治时间）以来的秒数，使用 **4 个字节**，只能表示从 1970 年到 2038 年。

它和时区有关，也就是说一个时间戳在不同的时区所代表的具体时间是不同的。

MySQL 提供了 FROM_UNIXTIME() 函数把 UNIX 时间戳转换为日期，并提供了 UNIX_TIMESTAMP() 函数把日期转换为 UNIX 时间戳。

默认情况下，如果插入时没有指定 TIMESTAMP 列的值，会将这个值设置为当前时间。

应该尽量使用 TIMESTAMP，因为它比 DATETIME 空间效率更高。

五、切分

水平切分

水平切分主要解决“数据库数据量大”问题

- 线性降低单库数据容量
- 线性提升数据库**写性能**

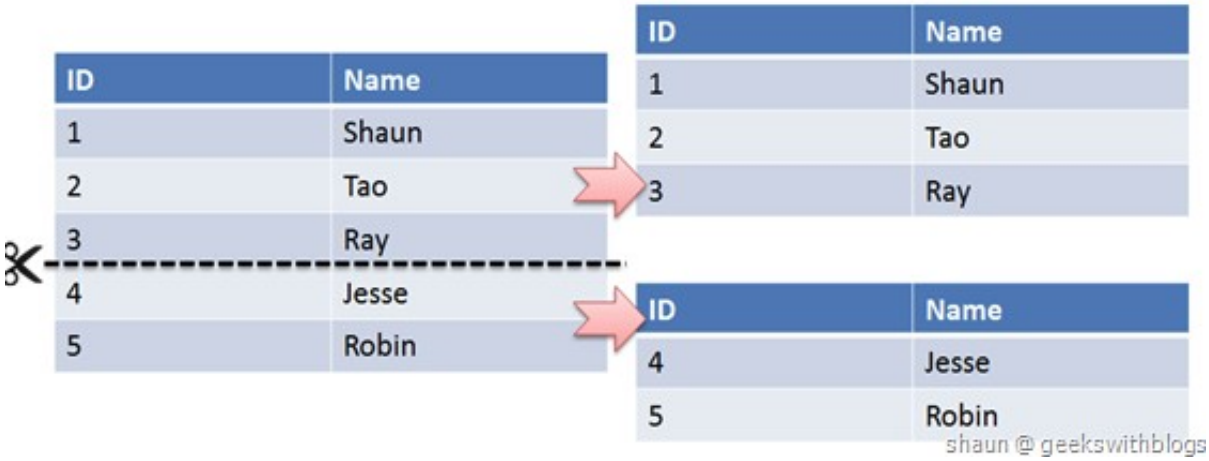
水平切分，也是一种常见的数据库架构，一般来说：

- **每个数据库之间没有数据重合**，没有类似binlog同步的关联
- 所有数据**并集**，**组成全部数据**
- 会用算法，来完成数据分割，**例如“取模”**

一个水平切分集群中的每一个数据库，通常称为一个**“分片”**。

水平切分又称为 Sharding，它是将同一个表中的记录拆分到多个结构相同的表中。

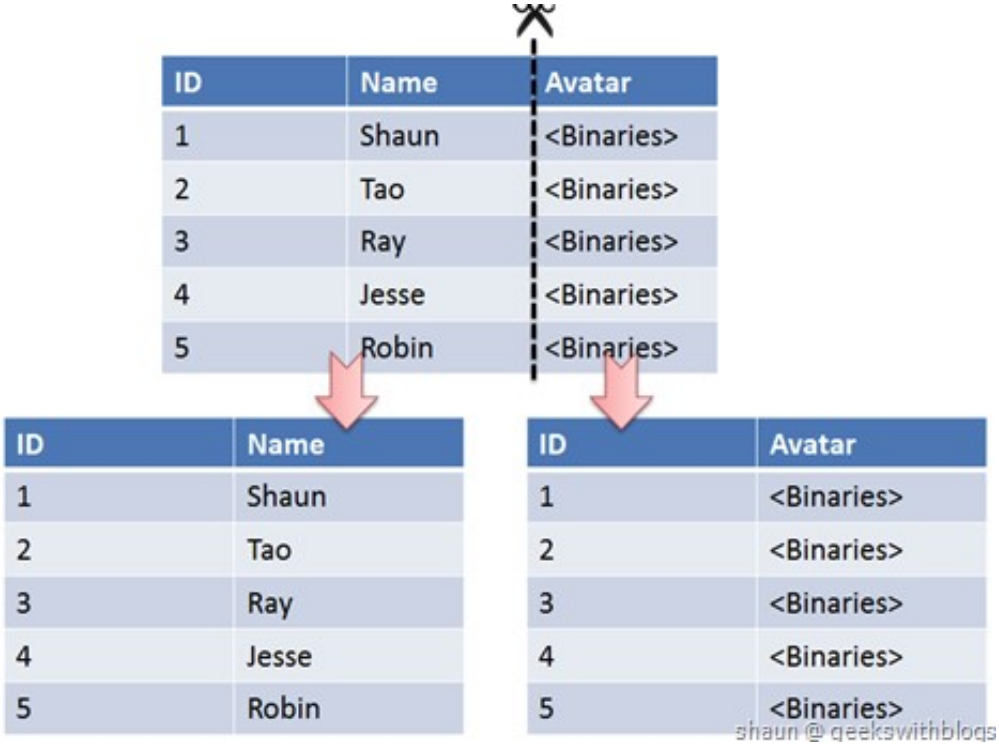
当一个表的数据不断增多时，Sharding 是必然的选择，它可以将数据**分布到集群的不同节点上，从而缓存单个数据库的压力。**



垂直切分

垂直切分是将一张表**按列切分成多个表**，通常是**按照列的关系密集程度进行切分**，也可以利用垂直切分将经常被使用的列和不经常被使用的列切分到不同的表中。

在数据库的层面使用**垂直切分将按数据库中表的密集程度部署到不同的库中**，例如将原来的**电商数据库垂直切分成商品数据库、用户数据库等**。



Sharding策略

- 哈希取模：hash(key) % N；
- 范围：可以是 ID 范围也可以是时间范围；
- 映射表：**使用单独的一个数据库来存储映射关系**。

Sharding存在的问题

1. 事务问题

使用分布式事务来解决，比如 XA 接口。

2. 连接

可以将原来的连接分解成**多个单表查询**，然后在用户程序中进行连接。

3. ID唯一性

- **使用全局唯一 ID（GUID）**
- **为每个分片指定一个 ID 范围**
- **分布式 ID 生成器 (如 Twitter 的 Snowflake 算法)**

六、复制

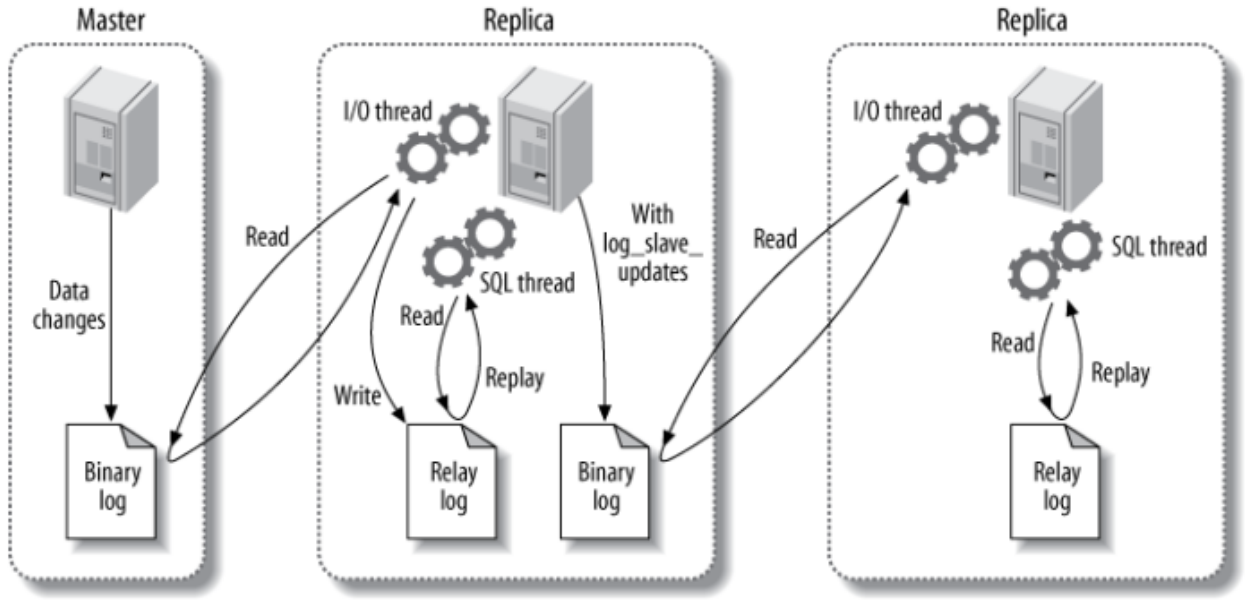
主从复制

主要涉及三个线程：binlog 线程、I/O 线程和 SQL 线程。

- **binlog 线程**： 负责将**主服务器上的数据更改写入二进制日志（Binary log）**中。
- **I/O 线程**： 负责**从主服务器上读取二进制日志**，并写入**从服务器的重放日志（Replay log）**中。
- **SQL 线程**： 负责读取**重放日志并重放其中的 SQL 语句**。

双机热备特指基于高可用系统中的两台服务器的热备（或高可用），双机高可用按工作中的切换方式分为：**主-备方式（Active-Standby方式）**和**双主机方式（Active-Active方式）**，主-备方式即指的是一台服务器处于某种业务的激活状态（即Active状态），另一台服务器处于该业务的备用状态（即Standby状态)。而双主机方式即指**两种不同业务分别在两台服务器上互为主备状态**（即Active-Standby和Standby-Active状态）。

双机热备就是使用MySQL提供的一种主从备份机制实现。所谓双机热备**其实是一个复制的过程**，复制过程中一个服务器充当主服务器，一个或多个服务器充当从服务。这个复制的过程实质上是从服务器复制主服务器上MySQL的二进制日志（bin-log），并在从服务器上还原主服务器上的sql语句操作，这样**只要两个数据库的初态是一样的，就能一直同步**。



读写分离

分组主要解决"数据库读性能瓶颈"问题

- 线性提升数据库**读性能**
- 通过**消除读写锁冲突提升数据库写性能**

一主多从，读写分离，主动同步，是一种常见的数据库架构，一般来说：

- 主库，提供数据库写服务
- 从库，提供数据库读服务
- 主从之间，通过某种机制同步数据，例如mysql的binlog

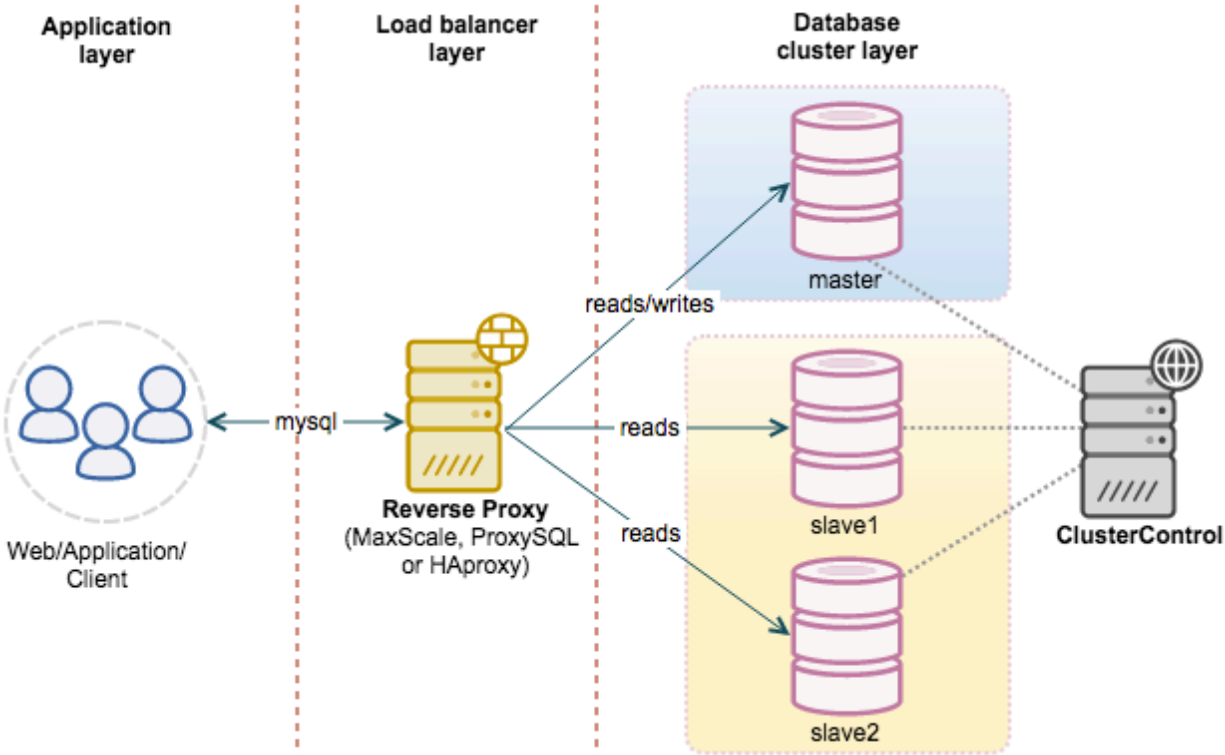
一个**组从同步集群**通常称为一个**“分组”**。

主服务器处理「写操作」以及实时性要求比较高的读操作，而从服务器处理「读操作」。

读写分离能提高性能的原因在于：

- 主从服务器负责各自的读和写，**极大程度缓解了锁的争用**；
- **从服务器可以使用 MyISAM**，提升查询性能以及节约系统开销；
- **增加冗余，提高可用性。**

读写分离常用「反向代理」方式来实现，代理服务器接收应用层传来的读写请求，然后决定转发到哪个服务器。



为什么不喜欢读写分离?

对于互联网**大数据量**，**高并发量**，**高可用要求高**，**一致性要求高**，前端面向用户的业务场景，如果数据库读写分离：

- 数据库连接池需要区分：读连接池，写连接池
- 如果要保证读高可用，**读连接池要实现故障自动转移**
- **有潜在的主库从库一致性问题**

使用缓存的解决方案

- 如果面临的是“**读性能瓶颈**”问题，**增加缓存**可能来得更直接，更容易一点
- 关于成本，**从库的成本比缓存高不少**
- 对于云上的架构，以阿里云为例，**主库提供高可用服务，从库不提供高可用服务**

当然，使用缓存架构的**潜在问题**：**如果缓存挂了**，流量全部压到数据库上，**数据库会雪崩**。不过幸好，**云上缓存**一般都提供**高可用的服务**。