数据库范式

> - 目前，主要有六种范式：第一范式、第二范式、第三范式、BC范式、第四范式和第五范式。满足最低要求的叫第一范式，简称1NF。在第一范式基础上进一步满足一些要求的为第二范式，简称2NF。其余依此类推。

> - 范式可以避免数据冗余，减少数据库的空间，减轻维护数据完整性的麻烦，但是操作困难，因为需要联系多个表才能得到所需要数据，而且范式越高性能就会越差。要权衡是否使用更高范式是比较麻烦的，一般在项目中，用得最多的也就是第三范式，我认为使用到第三范式也就足够了，性能好而且方便管理数据。

> 第一范式：对于表中的每一行，必须且仅仅有唯一的行值.在一行中的每一列仅有唯一的值并且具有原子性.
> （第一范式是通过把重复的组放到每个独立的表中，把这些表通过一对多关联联系起来这种方式来消除重复组的）
> 第二范式：第二范式要求非主键列是主键的子集，非主键列活动必须完全依赖整个主键。主键必须有唯一性的元素,一个主键可以由一个或更多的组成唯一值的列组成。
> 一旦创建，主键无法改变，外键关联一个表的主键。主外键关联意味着一对多的关系.（第二范式处理冗余数据的删除问题。
> 当某张表中的信息依赖于该表中其它的不是主键部分的列的时候，通常会违反第二范式）
> 第三范式：第三范式要求非主键列互不依赖.（第三范式规则查找以消除没有直接依赖于第一范式和第二范式形成的表的主键的属性。
> 我们为没有与表的主键关联的所有信息建立了一张新表。每张新表保存了来自源表的信息和它们所依赖的主键）
> 第四范式：第四范式禁止主键列和非主键列一对多关系不受约束
> 第五范式：第五范式将表分割成尽可能小的块，为了排除在表中所有的冗余

- **主要三范式**
  - 消除歧义，消除冗余，消除依赖。

    > - 第一范式：列不可分，eg:【联系人】（姓名，性别，电话），一个联系人有家庭电话和公司电话，那么这种表结构设计就没有达到 1NF，达到第一个规范式即为关系型数据库

    > - 第二范式：有主键，保证完全依赖。eg:订单明细表【OrderDetail】（OrderID，ProductID，UnitPrice，Discount，Quantity，ProductName），Discount（折扣），Quantity（数量）完全依赖（取决）于主键（OderID，ProductID），而 UnitPrice，ProductName 只依赖于 ProductID，不符合2NF；

    > - 第三范式：无传递依赖(非主键列 A 依赖于非主键列 B，非主键列 B 依赖于主键的情况)，eg:订单表【Order】（OrderID，OrderDate，CustomerID，CustomerName，CustomerAddr，CustomerCity）主键是（OrderID），CustomerName，CustomerAddr，CustomerCity 直接依赖的是 CustomerID（非主键列），而不是直接依赖于主键，它是通过传递才依赖于主键，所以不符合 3NF。

索引（主键自动索引，其他字段均可添加索引）

> 索引是对数据库表中一个或多个列的值进行排序的一种特殊数据结构，以协助快速查询,可以用来查询数据库表中特定的记录，索引是提高数据库性能的重要方式，所有字段都可添加,一般添加到使用频率高的字段。

> 索引的实现通常使用B_TREE及其变种。索引加速了数据访问，因为存储引擎不会再去扫描整张表得到需要的数据；相反，它从根节点开始，根节点保存了子节点的指针，存储引擎会根据指针快速寻找数据。

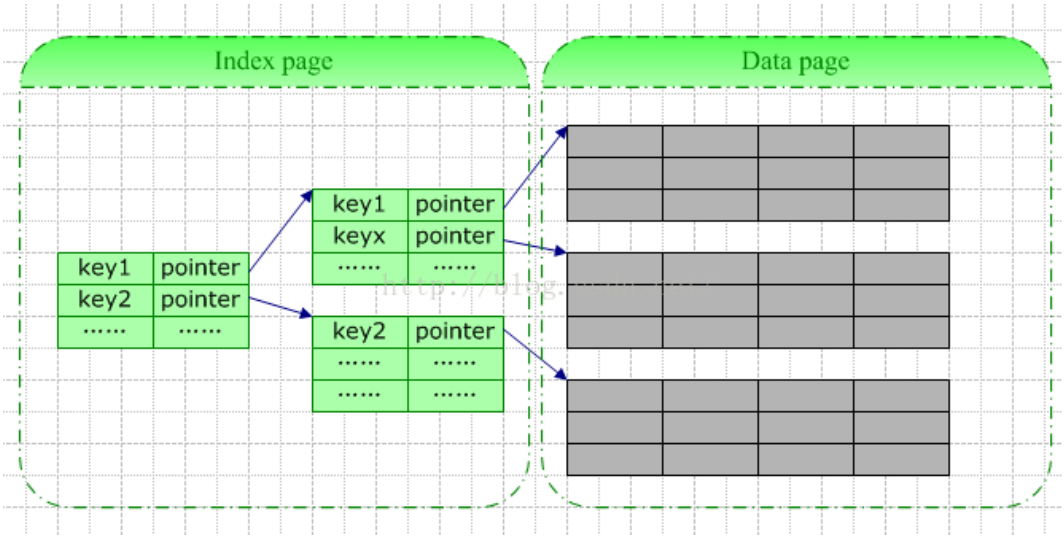
- **索引概念优缺点**

  - 优点：
    - 通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性；
    - 大大加快数据的检索速度，这也是创建索引的最主要的原因；
    - 加速表和表之间的连接；
    - 在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间；
    - 索引可以避免全表扫描。多数查询可以仅扫描少量索引页及数据页，而不是遍历所有数据页。
    - 对于非聚集索引，有些查询甚至可以不访问数据页
    - 聚集索引可以避免数据插入操作集中于表的最后一个数据页
    - 一些情况下，索引还可用于避免排序操作。
  - 缺点:
    - 创建索引和维护索引要耗费时间，具体地，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度；
    - 空间方面：索引需要占物理空间。,消耗资源。
    - 导致数据库系统更新数据的性能下降，因为大部分数据更新需要同时更新索引。

- **索引的存储**

  - 一条索引记录中包含的基本信息包括：键值（即你定义索引时指定的所有字段的值）+逻辑指针（指向数据页或者另一索引页）。
  - 当你为一张空表创建索引时，数据库系统将为你分配一个索引页，该索引页在你插入数据前一直是空的。此页此时既是根结点，也是叶结点。每当你往表中插入一行数据，数据库系统即向此根结点中插入一行索引记录。当根结点满时，数据库系统大抵按以下步骤进行分裂：
    - 创建两个儿子结点
    - 将原根结点中的数据近似地拆成两半，分别写入新的两个儿子结点
    - 根结点中加上指向两个儿子结点的指针

- 由于索引记录仅包含索引字段值（以及4-9字节的指针），索引实体比真实的数据行要小许多，索引页相较数据页来说要密集许多。一个索引页可以存储数量更多的索引记录，这意味着在索引中查找时在I/O上占很大的优势



- **索引使用规则**
  - 让限制条件更大的索引放在前面（根据“匹配索引扫描”效率更高）
  - 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。
  - 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有 必要。
- **什么情况下设置了索引但无法使用?**
  - 以“%(表示任意0个或多个字符)”开头的LIKE语句，模糊匹配；
  - OR语句前后没有同时使用索引；
  - 数据类型出现隐式转化（如varchar不加单引号的话可能会自动转换为int型）；
  - 对于多列索引，必须满足 最左匹配原则 (eg：多列索引col1、col2和col3，则 索引生效的情形包括 col1或col1，col2或col1，col2，col3)。
- **什么样的字段适合创建索引?**
  - 经常作查询选择的字段
  - 经常作表连接的字段
  - 经常出现在order by, group by, distinct 后面的字段
- **创建索引时需要注意什么?**
  - 非空字段：应该指定列为NOT NULL，除非你想存储NULL。在mysql中，含有空值的列很难进行查询优化，因为它们使得索引、索引的统计信息以及比较运算更加复杂。你应该用0、一个特殊的值或者一个空串代替空值；
  - 取值离散大的字段：（变量各个取值之间的差异程度）的列放到联合索引的前面，可以通过count()函数查看字段的差异值，返回值越大说明字段的唯一值越多字段的离散程度高；
  - 索引字段越小越好：数据库的数据存储以页为单位一页存储的数据越多一次IO操作获取

> 的数据越大效率越高。

- **主键、自增主键、主键索引与唯一索引概念区别**

  - 主键：指字段 唯一、不为空值 的列；
  - 主键索引：指的就是主键，主键是索引的一种，是唯一索引的特殊类型。创建主键的时候，数据库默认会为主键创建一个唯一索引；
  - 自增主键：字段类型为数字、自增、并且是主键；
  - 唯一索引：索引列的值必须唯一，但允许有空值。主键是唯一索引，这样说没错；但反过来说，唯一索引也是主键就错误了，因为唯一索引允许空值，主键不允许有空值，所以不能说唯一索引也是主键。

- **主键就是聚集索引吗？主键和索引有什么区别?**

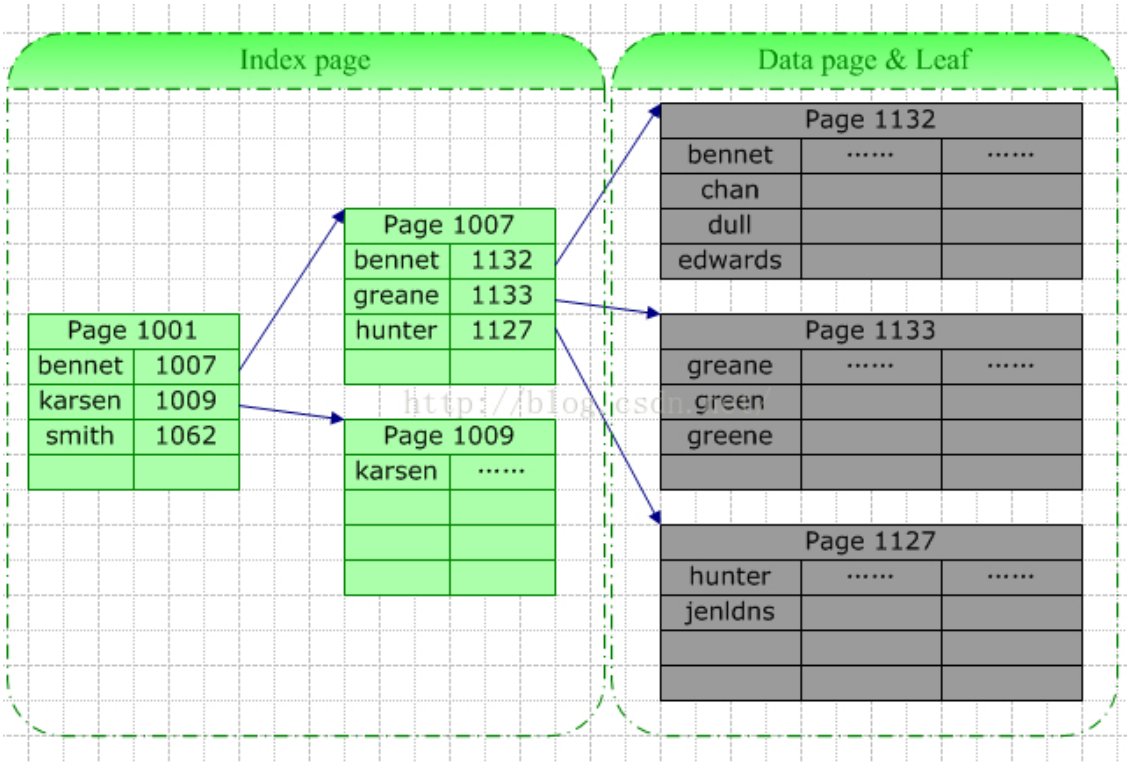
  - 主键是一种特殊的唯一性索引，其可以是聚集索引，也可以是非聚集索引。
  - 在SQLServer中，主键的创建必须依赖于索引，默认创建的是聚集索引，但也可以显式指定为非聚集索引。
  - InnoDB作为MySQL存储引擎时，默认按照主键进行聚集，如果没有定义主键，InnoDB会试着使用唯一的非空索引来代替。如果没有这种索引，InnoDB就会定义隐藏的主键然后在上面进行聚集。
  - 所以，对于聚集索引来说，你创建主键的时候，自动就创建了主键的聚集索引。

- **索引设计原则**

  - 添加索引的字段应该出现在where 语句中，不是select后面要查询的字段。
  - 索引的值，尽量唯一，效率更高。
  - 不要添加过多的索引
  - 主键自带索引，其他字段均可添加索引。 主键 id 最适合添加索引字段。
  - 索引分类
  - 聚集索引，表数据按照索引的顺序来存储的。对于聚集索引，叶子结点即存储了真实的数据行，不再有另外单独的数据页。
  - 非聚集索引，表数据存储顺序与索引顺序无关。对于非聚集索引，叶结点包含索引字段值及指向数据页数据行的逻辑指针，该层紧邻数据页，其行数量与数据表行数据量一致。
    - 在一张表上只能创建一个聚集索引，因为真实数据的物理顺序只可能是一种。如果一张表没有聚集索引，那么它被称为“堆集”（Heap）。这样的表中的数据行没有特定的顺序，所有的新行将被添加的表的末尾位置。
  - 普通索引：没有任何限制条件，任何数据类型的字段都可以添加
  - 唯一索引：索引值必须唯一，比如主键
  - 全文索引：只能添加在 char, varchar, text类型的字段，查询字段较大的字符串类型的字段时可以提高速率。（InnDB不支持）
  - 单列索引(聚集索引)：只对应一个字段的索引,(一个表只能建立一个聚集索引)。
  - 多列索引(非聚集索引)；在一张表多个字段创建一个索引,对每一行索引的列值并用一个指针指向数据所在的页面
  - SQLserver默认情况下建立的是非聚集索引，不需要重新组织表中的数据，对数据不排序，不需要全表扫描。

- 空间索引：只能建立在空间数据类型上(gis地理信息系统)。（经纬度数据不支持InnDB）

- **聚集索引**



  - 聚集索引与查询操作: 我们在名字字段上建立聚集索引，当需要在根据此字段查找特定的记录时，数据库系统会根据特定的系统表查找的此索引的根，然后根据指针查找下一个，直到找到
  - 例如我们要查询“Green”，由于它介于[Bennet,Karsen]，据此我们找到了索引页1007，在该页中“Green”介于[Greane, Hunter]间，据此我们找到叶结点1133（也即数据结点），并最终在此页中找以了目标数据行。
  - 这里的查找可能是从磁盘读取(Physical Read)或是从缓存中读取(Logical Read)，如果此表访问频率较高，那么索引树中较高层的索引很可能在缓存中被找到。所以真正的IO可能小于上面的情况。

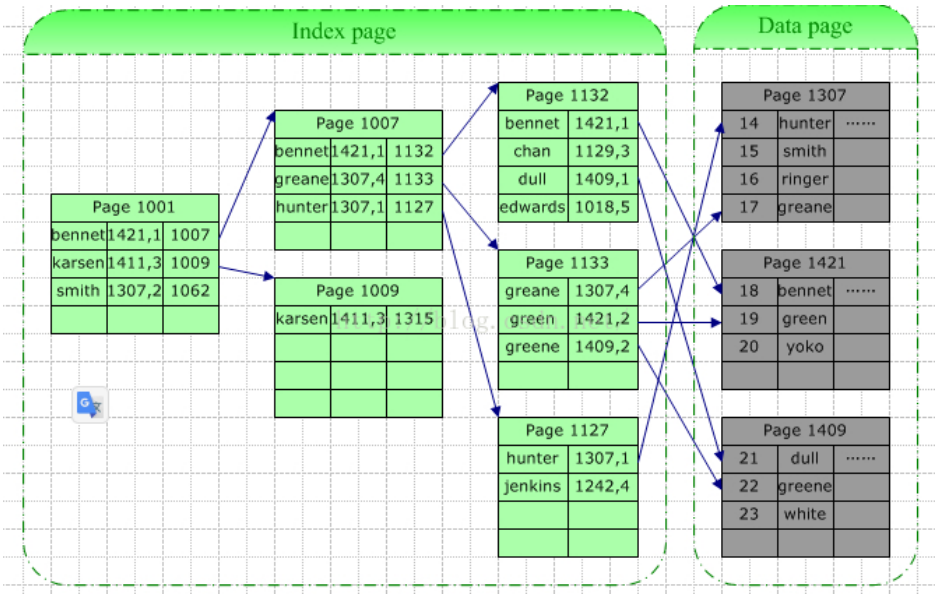
- **非聚集索引**

  - 非聚集索引与聚集索引相比：
    - 叶子结点并非数据结点
    - 叶子结点为每一真正的数据行存储一个“键-指针”对
    - 叶子结点中还存储了一个指针偏移量，根据页指针及指针偏移量可以定位到具体的数据行。
    - 类似的，在除叶结点外的其它索引结点，存储的也是类似的内容，只不过它是指向下一级的索引页的。
  - 聚集索引是一种稀疏索引，数据页上一级的索引页存储的是页指针，而不是行指针。而对于非聚集索引，则是密集索引，在数据页的上一级索引页它为每一个数据行存储一条索引记录。
  - 对于根与中间级的索引记录，它的结构包括：
    - 索引字段值

        - RowId（即对应数据页的页指针+指针偏移量）。在高层的索引页中包含RowId是为了当索引允许重复值时，当更改数据时精确定位数据行。
        - 下一级索引页的指针
    - 对于叶子层的索引对象，它的结构包括：
        - 索引字段值
        - RowId

- **索引覆盖**: 索引覆盖是这样一种索引策略：当某一查询中包含的所需字段皆包含于一个索引中，此时索引将大大提高查询性能。

    - 包含多个字段的索引，称为复合索引。索引最多可以包含31个字段，索引记录最大长度为600B。如果你在若干个字段上创建了一个复合的非聚集索引，且你的查询中所需Select字段及Where,Order By,Group By,Having子句中所涉及的字段都包含在索引中，则只搜索索引页即可满足查询，而不需要访问数据页。由于非聚集索引的叶结点包含所有数据行中的索引列值，使用这些结点即可返回真正的数据，这种情况称之为“索引覆盖”。
    - 在索引覆盖的情况下，包含两种索引扫描：
        - 匹配索引扫描
            - 此类索引扫描可以让我们省去访问数据页的步骤，当查询仅返回一行数据时，性能提高是有限的，但在范围查询的情况下，性能提高将随结果集数量的增长而增长。
            - 针对此类扫描，索引必须包含查询中涉及的的所有字段，另外，还需要满足：Where子句中包含索引中的“引导列”（Leading Column），例如一个复合索引包含A,B,C,D四列，则A为“引导列”。如果Where子句中所包含列是BCD或者BD等情况，则只能使用非匹配索引扫描。



        - 非匹配索引扫描: 如果Where子句中不包含索引的导引列，那么将使用非配置索引扫描。这最终导致扫描索引树上的所有叶子结点，当然，它的性能通常仍强于扫描所有的数据页。

- 创建索引：创建（唯一）[聚集|非聚集] 索引 索引*名* *在表*名{列_名}

```
    CREATE[UNIQUE] [CLUSTERED|NONCLUSTERED] INDEX index_name  ON
table_name{column_name...}[WITH FILLFACTOR=x],
    FILLFACTOR表示填充因子，定义该索引每页上的空间量，0--100.

    ------------------------------------
    alter   table  表名  add    index   索引名（要添加索引字段名）
    create  index  索引名  on    表名（要添加的字段名）
```

> - **查看索引**：show index from table_name
> - 删除索引

```
(1) alter  table  表名  drop index   索引名 。
(2) drop  index 索引名  on  表名。
```

索引的底层实现原理和优化

> 在数据结构中，我们最为常见的搜索结构就是二叉搜索树和AVL树(高度平衡的二叉搜索树，为了提高二叉搜索树的效率，减少树的平均搜索长度)了。然而，无论二叉搜索树还是AVL树，当数据量比较大时，都会由于树的深度过大而造成I/O读写过于频繁，进而导致查询效率低下，因此对于索引而言，多叉树结构成为不二选择。特别地，B-Tree的各种操作能使B树保持较低的高度，从而保证高效的查找效率。

- B-Tree(平衡多路查找树) :B_TREE是一种平衡多路查找树，是一种动态查找效率很高的树形结构。B_TREE中所有结点的孩子结点的最大值称为B_TREE的阶，B_TREE的阶通常用m表示，简称为m叉树。一般来说，应该是m>=3。一颗m阶的B_TREE或是一颗空树，或者是满足下列条件的m叉树：
  - 树中每个结点最多有m个孩子结点
  - 若根结点不是叶子节点，则根结点至少有2个孩子结点；
  - 除根结点外，其它结点至少有(m/2的上界)个孩子结点；
  - 结点的结构如下图所示，其中，n为结点中关键字个数，(m/2的上界)-1 <= n <= m-1；di(1<=i<=n)为该结点的n个关键字值的第i个，且di< d(i+1)；ci(0<=i<=n)为该结点孩子结点的指针，且ci所指向的节点的关键字均大于或等于di且小于d(i+1)；
  - 所有的叶结点都在同一层上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。
- B+Tree： InnoDB存储引擎的索引实现
  - B+Tree是应文件系统所需而产生的一种B_TREE树的变形树。一棵m阶的B+树和m阶的B_TREE的差异在于以下三点：
  - n 棵子树的结点中含有n个关键码
  - 所有的叶子结点中包含了全部关键码的信息，及指向含有这些关键码记录的指针，且叶子结点本身依关键码的大小自小而大的顺序链接；
  - 非终端结点可以看成是索引部分，结点中仅含有其子树根结点中最大（或最小）关键码
  - 对于B+树，不管查找成功与否，每次查找都是走了一条从根到叶子结点的路径。
- 为什么说B+-tree比B 树更适合实际应用中操作系统的文件索引和数据库索引?
  - B+tree的磁盘读写代价更低：B+tree的内部结点并没有指向关键字具体信息的指针(红色部分)，因此其内部结点相对B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关

> 键字也就越多，相对来说IO读写次数也就降低了；
> - B+tree的查询效率更加稳定：由于内部结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引，所以，任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当；
> - 数据库索引采用B+树而不是B树的主要原因：B+树只要遍历叶子节点就可以实现整棵树的遍历，而且在数据库中基于范围的查询是非常频繁的，而B树只能中序遍历所有节点，效率太低。

- 文件索引和数据库索引为什么使用B+树?

> - 文件与数据库都是需要较大的存储，也就是说，它们都不可能全部存储在内存中，故需要存储到磁盘上
> - 而所谓索引，则为了数据的快速定位与查找，那么索引的结构组织要尽量减少查找过程中磁盘I/O的存取次数，因此B+树相比B树更为合适。
> - 数据库系统巧妙利用了局部性原理与磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入，而红黑树这种结构，高度明显要深的多，并且由于逻辑上很近的节点(父子)物理上可能很远，无法利用局部性。
> - 最重要的是，B+树还有一个最大的好处：方便扫库。B树必须用中序遍历的方法按序扫库，而B+树直接从叶子结点挨个扫一遍就完了，B+树支持range-query非常方便，而B树不支持，这是数据库选用B+树的最主要原因。