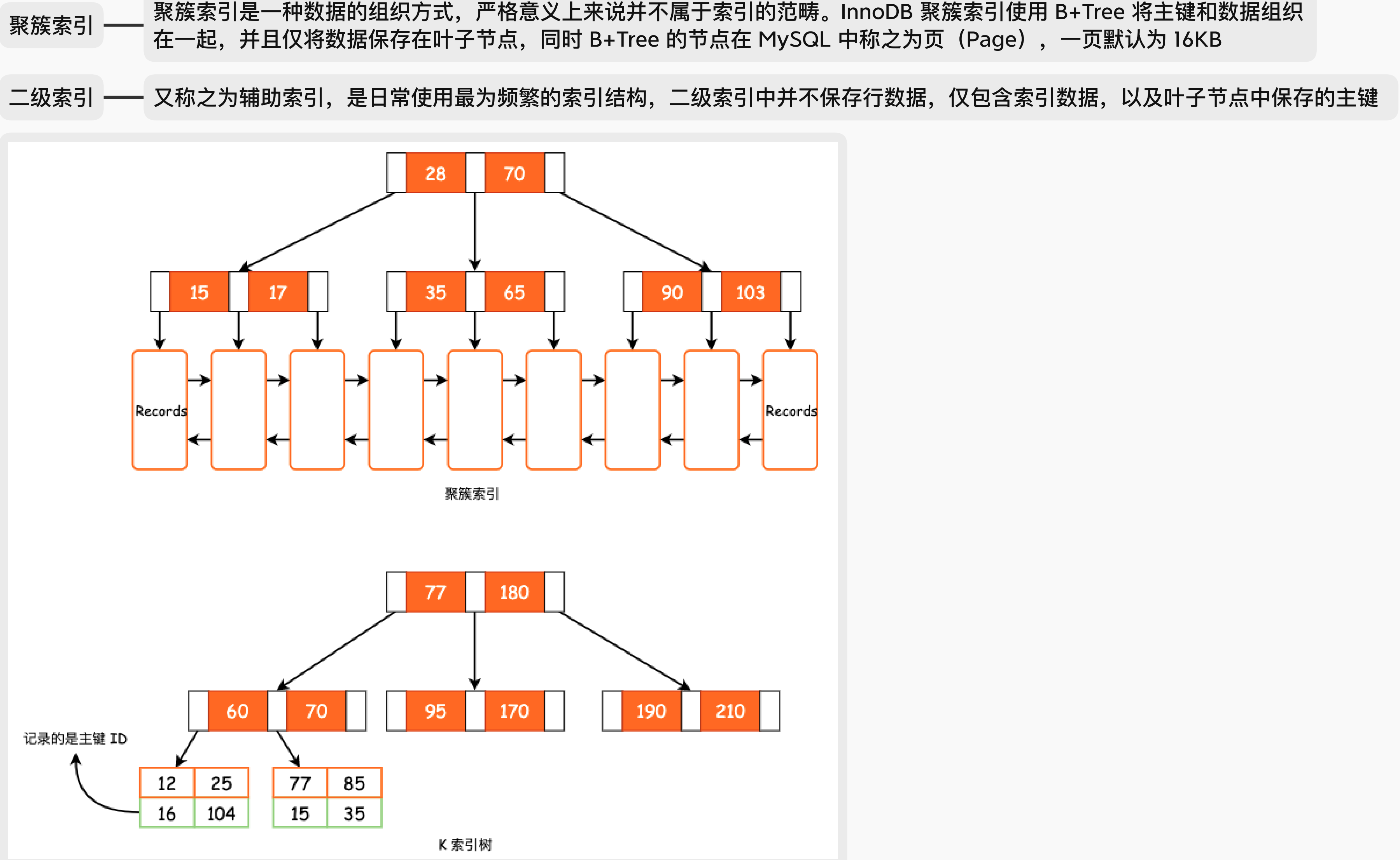
InnoDB Index

聚簇索引

基本索引模型

- 聚簇索引：聚簇索引是一种数据的组织方式，严格意义上来说并不属于索引的范畴。InnoDB 聚簇索引使用 B+Tree 将主键和数据组织在一起，并且仅将数据保存在叶子节点，同时 B+Tree 的节点在 MySQL 中称之为页（Page），一页默认为 16KB
- 二级索引：又称之为辅助索引，是日常使用最为频繁的索引结构，二级索引中并不保存行数据，仅包含索引数据，以及叶子节点中保存的主键



- 查询过程
  - 主键查询：直接在聚簇索引中查找即可，因为聚簇索引本身就是使用主键进行组织的，并且这棵 B+Tree 的叶子节点保存着数据
  - 普通索引查询：首先需要通过 K 索引树找到该索引对应的主键，然后再去聚簇索引中进行查找，这个过程称之为回表

关于主键

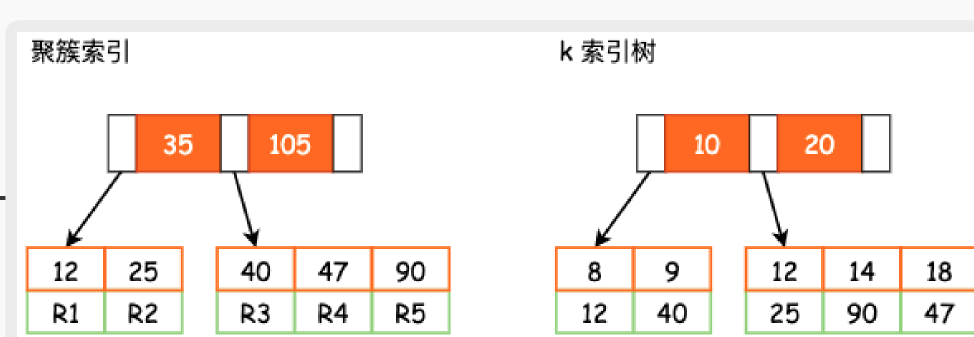
- 首先，对于 InnoDB 存储引擎的一张表来说，我们可以在创建 table 的时候不指定 primary key，但是这并不代表生成的表结构中没有聚簇索引
- 聚簇索引组织方式的选择
  1. 若 table 中指定了主键，那么主键将作为聚簇索引的组织方式
  2. 若 table 中未指定主键，那么 InnoDB 会选择一个 UNIQUE NOT NULL 的非空唯一索引作为聚簇索引的组织方式
  3. 若 table 中既未指定主键，同时也没有非空唯一索引的话，InnoDB 会生成一个隐藏的主键
- 总之，InnoDB 一定会找一个东西来组织聚簇索引，这是 InnoDB 运行的根本

B+Tree 这棵 N 叉树的 N 和什么有关系?

N 和 InnoDB Page Size 以及索引的长度有关，Page Size 越小，N 越小；索引长度越长，也会导致 N 越小

普通索引与唯一索引

查询



select * from T where k = 12;

- 普通索引：顺着 k 索引树查找主键 ID，当找到 k = 12 这一条数据时，还需要向后查找，判断是否有其他数据也为 12
- 唯一索引：顺着 k 索引树查找主键 ID，当找到 k = 12 这一条数据时直接返回，因为 k 是 unique 的，不可能出现重复

两者效率几乎相同，因为 B+Tree 叶子节点是 page，所以会有相邻的特性，即使向后多找几条数据，大概率也是纯内存操作

更新

- Change Buffer
  - 当我们更新一条数据时，如果该数据的 page 在内存中则直接进行更新。若该数据页不在内存中，且不会影响数据一致性的前提下，InnoDB 会将这个修改写入至 change buffer 中（change buffer 随 redo log 持久化），后续再读取该行数据时，将数据读入内存，然后执行 page 和 change buffer 的 merge 操作
  - 使用 change buffer 明显可以加快 update 语句的执行，不需要将 page 从硬盘中读取至内存中
- 普通索引：在更新数据时，如果更新的页不在内存中的话，由于不需要进行唯一性检查，所以直接写入到 change buffer 中，持久化交给 redo log 来做
- 唯一索引：在更新数据时，如果更新的页不在内存中的话，由于需要进行唯一性检查，所以必须把 page 读到内存中，判断后进行操作

在 page 不在内存中的情况，普通索引的更新效率要高于唯一索引的更新效率

- 新增：在新增数据时，由于主键是唯一的，因此对聚簇索引的写入必然需要将页面读入内存中。但是如果还存在其它索引写入的话，那么非唯一索引也可以使用 change buffer 来优化

综上，如果业务端能够保证数据的唯一性，并且是写多读少，那么使用普通索引比唯一索引会有更高的性能

覆盖索引

前面我们提到了，当我们使用二级索引查询某一行数据时，首先到该二级索引树上查询主键 ID，然后拿着主键 ID 再去聚簇索引中查找完整的一行数据。这个过程需要遍历两棵 B+Tree，所以也称之为回表

覆盖索引就是指我们可以直接在一棵 B+Tree 上找到数据，而不需要进行回表的操作

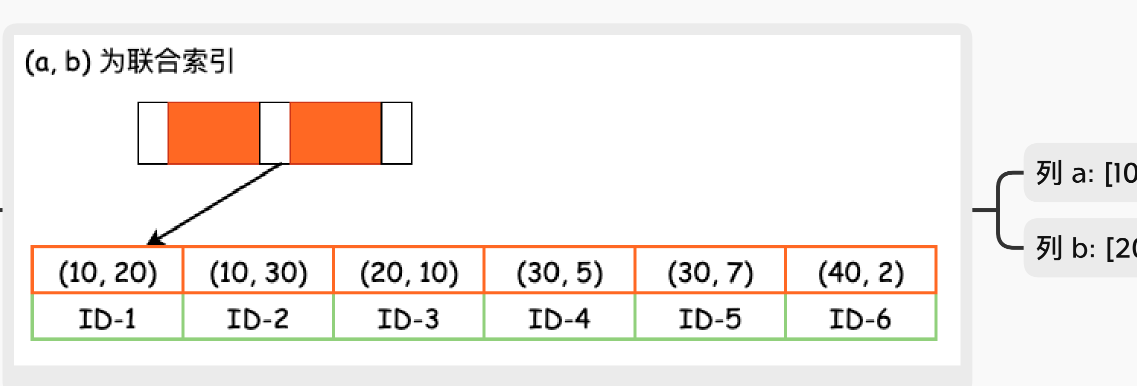
1. 如 select * from T where id = 1024，可直接在聚簇索引中找到数据
2. 如 select count(*) from T，当表 T 中存在二级索引时，MySQL 会直接统计二级索引的条目，不需要去聚簇索引中遍历。count(*) 会统计 NULL，count(Field) 不会统计 NULL
3. 又如业务需要经常使用 mobile 查询 name，那么我们就可以将 (mobile, name) 定义成联合索引，只需要遍历 (mobile, name) 组成的 B+Tree 即可得到结果

总之，覆盖索引的本质就是只查询一棵 B+Tree 即可找到我们所需要的数据

最左前缀原则

一言以蔽之，最左匹配原则就是匹配联合索引的前 N 个字段，或者是字符串索引的前 M 个字符

联合索引



- 列 a: [10, 10, 20, 30, 30, 40]
- 列 b: [20, 30, 10, 5, 7, 2]

联合索引既然也是索引，那么也会单独的占有一棵 B+Tree，此时每一条索引数据中都会有两个索引，即 a, b。其组织方式也是有序的，其实就是按照索引定义的顺序进行排序，首先用 a 从小到大排序，若两条数据的 a 相同，则使用 b 从小到大的方式进行排序。这样一来，a 一定是有序的，但是 b 却不一定

如果用 C++ 代码来表示的话，就相当于对一个 vector<vector<T>> 进行排序

```
auto f = [](vector<int>& a, vector<int>& b){
    if (a[0] == b[0]) return a[1] < b[1];
    return a[0] < b[0];
};
```

此外，如果我们建立了 (a, b ,c) 这三个字段的联合索引以后，其实就相当于有了 (a)、(a, b) 以及 (a, b, c) 这三个索引，不需要再去为字段 a 额外的添加索引了

联合索引有时候会和覆盖索引一起出现，比如业务有一个高频操作，通过 mobile 查询 name，那么我们就可以将 (mobile, name) 建立联合索引，查询时既能够使用最左匹配原则快速定位到 mobile，并且只需要在一棵 B+Tree 上找到所有数据，无需回表