MVTO

txn-id 字段可以作为这个元组的写锁来用。例如事务T想要更新这个tuple，需要查看字段 txn-id 是否等于 0（默认是0），如果是的话，再将自己 Tid 填到这个字段。其他事务再想如此操作，就会发现字段txn-id不是 0 也不是自己 Tid ，就会知道已有一个事务占据这个元组的写锁，自己不能再写了。

字段 begin-ts 和 end-ts 的时间戳代表这个元组版本的生命周期，他们的初始值会设置为 0。当这个元组被删除了，那么 begin-ts 会被设置上INF。

仍然是有时间戳排序扩展而来的，这里要强调一点，时间戳排序虽然名字里有排序，但是**主要目的是通过时间戳来管理和解决事务之间的冲突**，而不是传统意义上的排序，**更多是指事务按照时间戳的顺序来执行和提交，而不是对事务进行物理排序。**

可以说时间戳是用于验证和检测事务之间冲突的手段

Basic Timestamp Ordering(T/O) Protocol

数据库中的所有对象（一般就是指tuple这种对象）上面要附带两个时间戳，一个读时间戳（上一次读这个对象的事务的时间戳/事务号），一个写时间戳（上一次写这个对象的事务的时间戳/事务号）。

MVTO

涉及到版本相关的数据项，我们都会给予每个版本两个信：

- `begin_ts` 表示该版本从哪个时间戳开始可见。
- `end_ts` 表示该版本从哪个时间戳开始不可见。

当事务进行读操作时，会查找所有版本中满足 `begin_ts <= current_ts < end_ts` 的版本。如果找到符合条件版本，则操作成功，否则就等待。

当事务进行写操作时，会生成一个新的数据版本，并将该版本与事务的时间戳关联起来。此外还要更新现有版本的 `end_ts`，使其等于当前事务的时间戳，表示该版本从当前事务开始不再可见。

这里有一个点是，什么时候更新现有版本的 end_ts 呢？一般来说，是建议提交之后再更新的，**因为这样不会出现幻读或者说找不到对象的情况**（更新了一个版本之后没提交，另一个事务立马读）。

但是也可以创建版本的时候就立马更新 end_ts，这个时候就很麻烦了，因为找不到了，可能就需要单独处理，比如每个事务开始的时候，先创建一个快照，默认找不到的时候那么值就是快照里面的值。

例子：

当前有一个数据项 X，目前有两个版本，事务二（时间戳为2）进行 X 的写入，减少 200 余额：

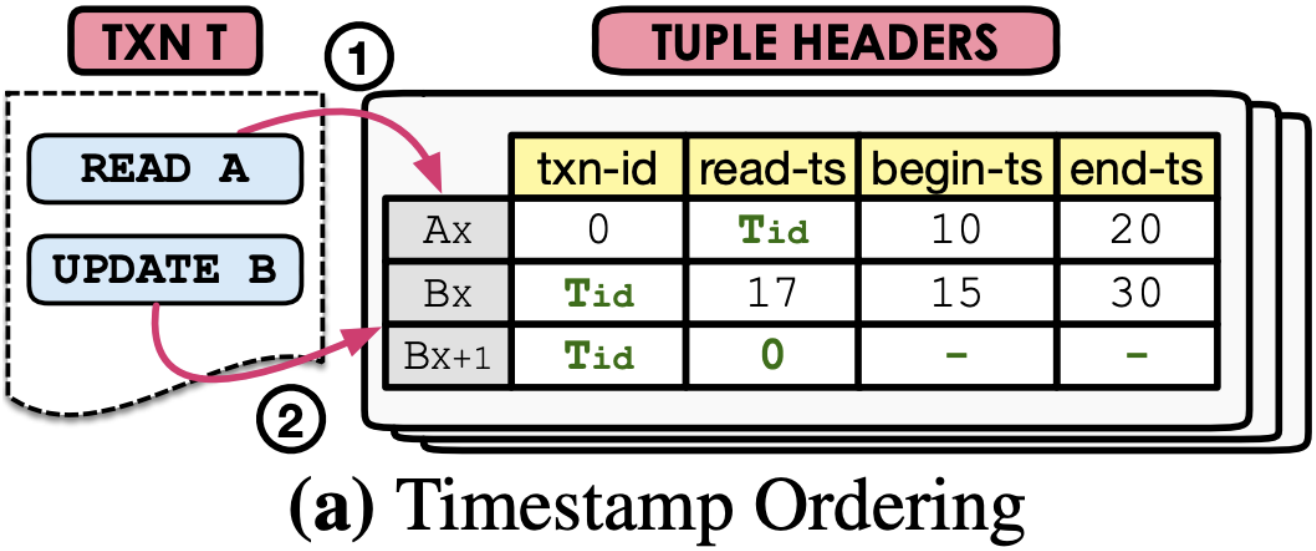
- 版本 1：{余额: 1000, `begin_ts`: 0, `end_ts`: 1}
- 版本 2：{余额: 1500, `begin_ts`: 1, `end_ts`: MAX}

写入后：

- 版本 1：{余额: 1000, `begin_ts`: 0, `end_ts`: 1}
- 版本 2：{余额: 1500, `begin_ts`: 1, `end_ts`: 2}
- 版本 3：{余额: 1300, `begin_ts`: 2, `end_ts`: MAX}

miniob 里面的 MVCC 的写法就是 MVTO 模式的

最常见的 MVTO 应该是这样的：



(a) Timestamp Ordering

可以看到这里多了两点，也就是 read-ts 和 txn-id：

1. txn-id: 0表示没有被锁，非 0 表示该锁被 ID=txn-id 的事务锁持有
2. read-ts: 读取该条记录的事务中 txn-id 的最大值

txn-id 存在的情况就是为了防止写写冲突，对A=100，事务1要+50，事务2要-30，不加的话事务二写 A 的时候就不会等待，这样就造成了写写冲突。

read-ts 的作用，比如：如果一条数据已经被一个 txn_id 更大的事务A读过了，那么事务B（小于A）的更新就得回滚。

miniob 是读已提交级别的，写写冲突很粗暴是直接回滚后面的，同样的也是在 commit 的时候更新 end_xid

存储结构

Double write buffer

会存储在磁盘中，结构为开头一个 double write buffer header 作为一些元信息，之后才是一个个的页面。

```
+------------------+------------------+------------------+------------
-+
| DoubleWriteBuffer Header | Page 0 | Page 1 | Page 2 |
| +------------------+------------------+------------------+----------
-+
(DoubleWriteBufferHeader::SIZE) (DoubleWritePage::SIZE)
```

buffer pool 来说，一个文件一个 buffer pool，但是所有 buffer pool 共用一个 BPFrameManager

SQL

对于命令，需要在 stmt.cpp 中添加处理对应解析出来的 sql 语句

整个的 sql 都是以 net 模块下 sql_task_handler.cpp 来执行的

对于基础的 command，是通过之前知道的大 case 去执行的

而对于像是 insert，select 之类的操作，我们是会在：

我们会在 write 的过程中，打开 sqlresult 的 open 函数，并在其中执行对应的操作

每个操作对应一个 operator，存放在 `/sql/operator` 下面

对于 update 的实现，有一个最粗暴的办法就是，直接先全部 delete 掉，最后全部 insert 回去

简单的说，我们需要在 operator 下面新建对应的操作，然后通过 update 解析成对应的模块

对于 select 他所用的 operator 是 TableScanPhysicalOperator

而对于条件的过滤是 storage 模块下面 common 里面的 confition_filter.h

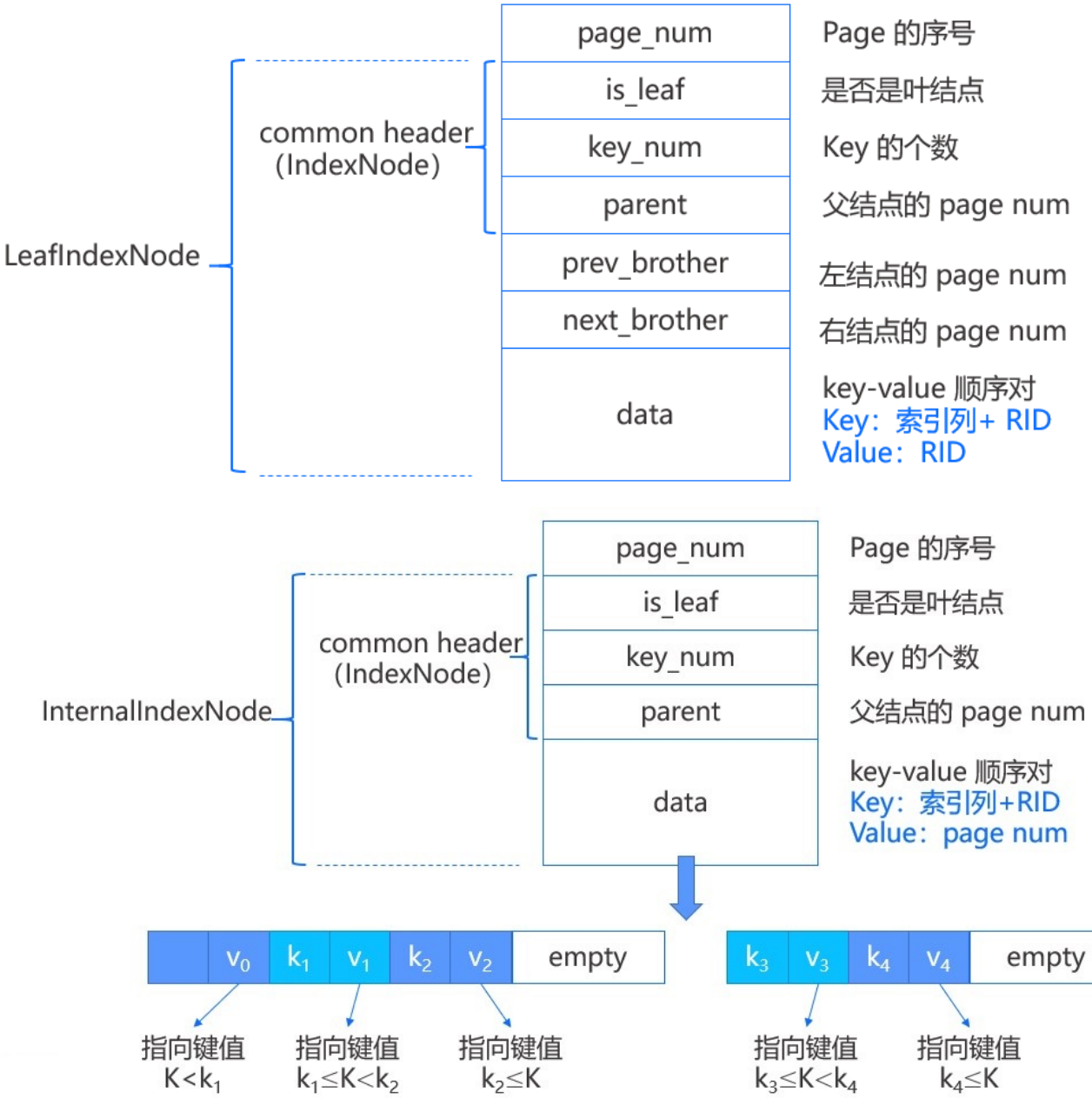
对于属性的类型的增加，比如增加 date 之类的，需要在 observer 的 common 下面的 type 文件夹里面去增加

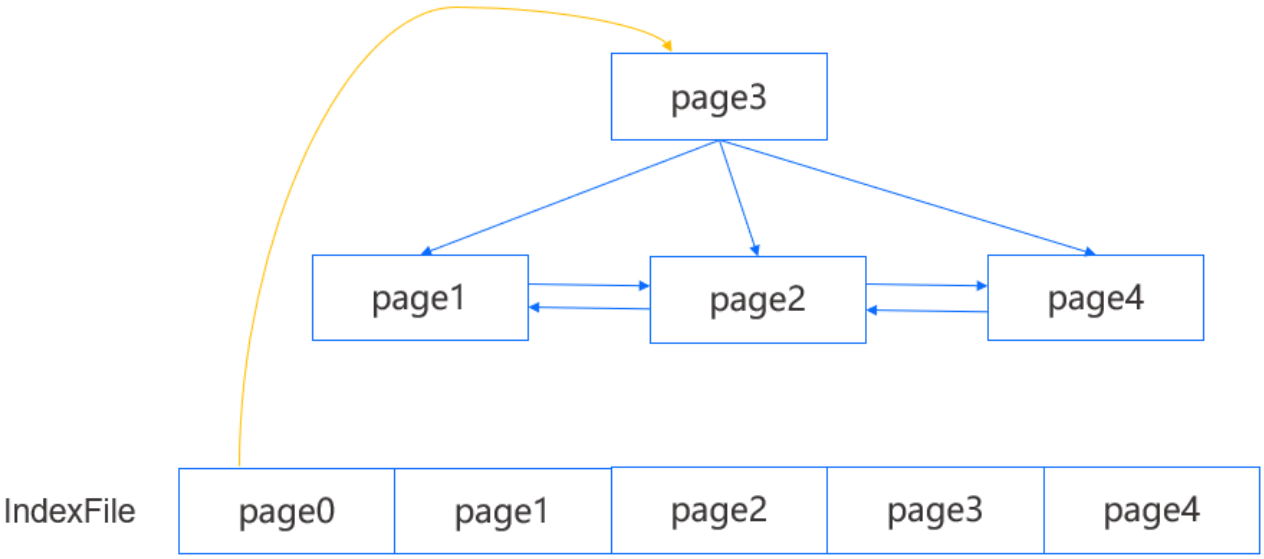
有一个问题是在 storage 里面的 condition_filter.cpp 下面的 init 模块有一个 todo，比如我们需要去实现整数与浮点数之间的对比。

```
// if (!field_type_compare_compatible_table[type_left][type_right]) {
//     // 不能比较的两个字段， 要把信息传给客户端
//     return RC::SCHEMA_FIELD_TYPE_MISMATCH;
//   }
```

```
// NOTE: 这里没有实现不同类型的数据比较，比如整数跟浮点数之间的对比
// 但是选手们还是要实现。这个功能在预选赛中会出现
if (type_left != type_right) {
  return RC::SCHEMA_FIELD_TYPE_MISMATCH;
}
```

B+ tree




page3
page1
page2
page4
IndexFile
page0
page1
page2
page3
page4