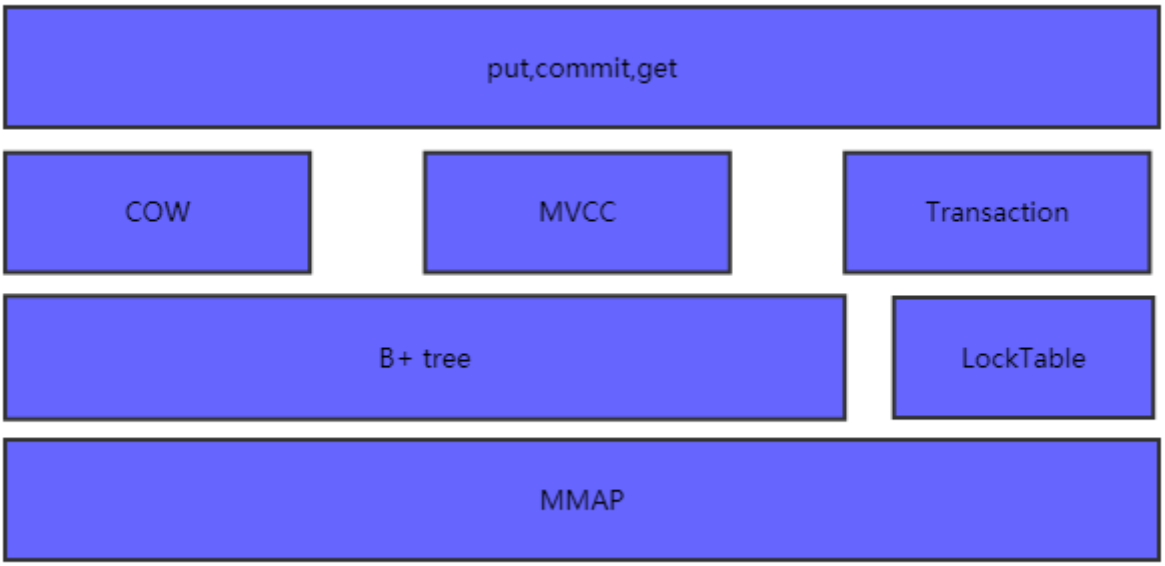
lmdb个人总结

- LMDB（Lightning Memory-Mapped Database）是一个小型数据库，具有一些强大的功能：
    - 有序映射接口（键始终排序，支持范围查找）
    - 带有MVCC（多版本并发控制）的完全事务性，完全ACID（原子性，一致性，隔离性，耐久性）语义。
    - 读者/写者事务：读者不会阻止写者，写者也不会阻止读者。
        - 写入程序已完全序列化，因此写入始终无死锁。
    - 读事务非常低开销，可以不使用malloc或任何其他阻塞调用来执行。
    - 支持多线程和多进程并发，环境可以由同一主机上的多个进程打开。
    - 可以创建多个子数据库，其中事务覆盖所有子数据库。
    - 内存映射，允许零拷贝查找和迭代。
    - 免维护，无需外部过程或后台清理/压缩。
    - 防崩溃，无需日志或崩溃恢复过程。
    - 没有应用程序级缓存。
        - LMDB充分利用了操作系统的缓冲区高速缓存。
    - 32KB的目标代码和C的6KLOC。
- LMDB的基本结构



- 内存映射（Memory Map）
    - 内存映射就是把物理内存映射到进程的地址空间之内，这些应用程序就可以直接使用输入输出的地址空间。
        - 使用内存映射文件处理存储于磁盘上的文件时，将不需要由应用程序对文件执行I/O操作，这意味着在对文件进行处理时将不必再为文件申请并分配缓存，所有的文件缓存操作均由系统直接管理，由于取消了将文件数据加载到内存、数据从内存到文件的回写以及释放内存块等步骤，使得内存映射文件在处理大数据量的文件时能起到相当重要的作用。
    - 一般文件的io过程（两次数据拷贝）

1. 首先调用read()（系统调用）先将文件内容从硬盘拷贝到内核空间的一个缓冲区（一次数据拷贝）。
2. 然后再将这些数据拷贝到用户空间。（一次数据拷贝）

- 内存映射的文件io过程（一次数据拷贝）
  1. 调用mmap()（系统调用）分配逻辑地址。
  2. 逻辑地址转换成物理地址。
  3. 进程第一次访问ptr指向的内存地址，发生缺页中断。由中断处理函数将数据拷贝到相应的内存地址上。（一次数据拷贝）
  4. 虚拟地址置换。
- lmdb使用mmap文件映射,不管这个文件存储实在内存上还是在持久存储上。
- lmdb的所有读取操作都是通过mmap将要访问的文件只读的映射到虚拟内存中，直接访问相应的地址。
- 因为使用了read-only的mmap，同样避免了程序错误将存储结构写坏的风险。
- 写操作，则是通过write系统调用进行的，这主要是为了利用操作系统的文件系统一致性，避免在被访问的地址上进行同步。
- lmdb把整个虚拟存储组织成B+Tree存储,索引和值读存储在B+Tree的页面上（聚集索引）。
- LMDB中使用append-only B+树，其更新操作最终都会转换为B+树的物理存储的append操作，文件不支持内部的修改，只支持append。
  - append增加了存储开销，因为旧的数据依然存在。带来一个额外的好处就是，旧的链路依然存在，依然可以正常的访问，例如过去有个人持有了过去的root的指针，那么过去的整棵树都完整的存在。

- COW(Copy-on-write)写时复制
  - 如果有多个调用者（callers）同时要求相同资源（如内存或磁盘上的数据存储），他们会共同获取相同的指针指向相同的资源，直到某个调用者试图修改资源的内容时，系统才会真正复制一份专用副本（private copy）给该调用者，而其他调用者所见到的最初的资源仍然保持不变。
  - 因此多个调用者只是读取操作时可以共享同一份资源。
  - 优点
    - 如果调用者没有修改该资源，就不会有副本（private copy）被创建。

- MVCC(Multiversion concurrency control)多版并发本控制
  - 当MVCC数据库需要更新数据项时，它不会用新数据覆盖旧数据，而是将旧数据标记为已过时并在其他位置添加新版本。
    - 因此存储了多个版本，但只有一个是最新版本。
  - MVCC通过保存数据的历史版本，根据比较版本号来处理数据的是否显示，从而达到读取数据的时候不需要加锁就可以保证事务隔离性的效果。
  - 数据库系统维护当前活跃的事务ID列表m_ids，其中最小值up_limit_id和最大值low_limit_id，被访问的事务ID：trx_id。
    - 如果trx_id< up_limit_id，说明trx_id对应的事务在生成可读视图前已经被提交了，可以被当前事务访问
    - 如果trx_id> low_limit_id，说明事务trx_id生成可读视图后才生成的，所以不可以被当前事务访问到。
    - 如果up_limit_id <=trx_id<= low_limit_id，判断trx_id是否在m_ids中，若在，则说明trix_id在生成可读视图时还处于活跃，不可以被访问到；弱国不在m_ids中，说明在生成可读视图时该事务已经被提交了，故可以被访问到。

- MVCC/COW在LMDB中的实现
    - LMDB对MVCC加了一个限制，即只允许一个写线程存在，从根源上避免了写写冲突，当然代价就是写入的并发性能下降。
        - 因为只有一个写线程，所以不会不需要wait日志、读写依赖队列、锁队列等一系列控制并发、事务回滚、数据恢复的基础工具。
    - MVCC的基础就是COW，对于不同的用户来说，若其在整个操作过程中不进行任何的数据改变，其就使用同一份数据即可。若需要进行改变，比如增加、删除、修改等，就需要在私有数据版本上进行，修改完成提交之后才给其他事务可见。

- LMDB的事务实现
    - Atom（A）原子性：LMDB中通过txn数据结构和cursor数据结构的控制，通过将脏页列表放入dirtylist中，当txn进行提交时再一次性统一刷新到磁盘中或者abort时都不提交保证事务要不全成功、要不全失败。对于长事务，若页面spill到磁盘，因为COW技术，这些页面未与整棵B-Tree的rootpage产生关联，因此后续的事务还是不能访问到这些页面，同样保证了事务的原子性。
    - Consistency(C)一致性： 有如上的操作,保证其数据就是一致的，不存在因为多线程同时写数据导致数据产生错误的情况。
    - Isolation（I）隔离性：事务隔离通过锁控制（MUTEX），LMDB支持的锁互斥是进程级别/线程级别，支持的隔离方式为锁表支持，读读之间不锁，写等待读完成之后开始，读等待写完成后开始。
    - Duration（D）持久性：，没有使用WAL、undo/redo log等技术来保证系统崩溃时数据库的可用性，其保证数据持续可用的技术是**COW技术和只有一线程写技术。**
        - 假如LMDB或者系统崩溃时，只有读操作，那么数据本来就没有发生变化，因此数据将不可能遭到破坏。假如崩溃时，有一个线程在进行写操作，则只需要判断最后的页面号与成功提交到数据库中的页面号是否一致，若不一致则说明写操作没有完成，则最后一个事务写失败，数据在最后一个成功的页面前的是正确的，后续的属于崩溃事务的，不能用，这样就保证了数据只要序列化到磁盘则一定可用，要不其就是还没有遵循ACI原则序列化到磁盘。

```
import lmdb
# map_size定义最大储存容量，单位是kb，以下定义1TB容量
env = lmdb.open("./train", map_size=1099511627776)

# 参数write设置为True才可以写入
txn = env.begin(write=True) # 开启事务

# 通过cursor()遍历所有数据和键值
for key, value in txn.cursor():
    print (key, value)

# 添加数据和键值
txn.put(key = '1', value = 'aaa')
txn.put(key = '2', value = 'bbb')
txn.put(key = '3', value = 'ccc')

# 通过键值删除数据
txn.delete(key = '1')
```

```
# 修改数据
txn.put(key = '3', value = 'ddd')

# 通过commit()函数提交更改
txn.commit()            # 提交事务

env.close()     # 关闭事务
```

HDF5

HDF5 (Hierarchical Data Format) 是一种常见的跨平台数据储存文件，可以存储不同类型的图像和数码数据，并且可以在不同类型的机器上传输，同时还有统一处理这种文件格式的函数库。

- HDF5 文件结构中有 2 primary objects:
    - Groups
        - 类似于文件夹，每个 HDF5 文件其实就是根目录 (root)为 group'/'
    - Datasets
        - 类似于 NumPy 中的数组 array
        - 每个 dataset 可以分成两部分
            - 原始数据 (raw) data values
            - 元数据 metadata
                - 描述并提供有关其他数据的信息的数据
- 示例

```
+-- Dataset
|   +-- (Raw) Data Values (eg: a 4 x 5 x 6 matrix)
|   +-- Metadata
|   |   +-- Dataspace (eg: Rank = 3, Dimensions = {4, 5, 6})
|   |   +-- Datatype (eg: Integer)
|   |   +-- Properties (eg: Chuncked, Compressed)
|   |   +-- Attributes (eg: attr1 = 32.4, attr2 = "hello", ...)
|
```

    - Dataspace 给出原始数据的秩 (Rank) 和维度 (dimension)
    - Datatype 给出数据类型
    - Properties 说明该 dataset 的分块储存以及压缩情况
        - Chunked: 子集的访问时间更短； 可扩展的
        - Chunked & Compressed: 提高存储效率，传输速度
    - Attributes 为该 dataset 的其他自定义属性
- 整个HDF5结构

```
+-- /
|   +-- group_1
```

```
|   |   +-- dataset_1_1
|   |   |   +-- attribute_1_1_1
|   |   |   +-- attribute_1_1_2
|   |   |   +-- ...
|   |   |
|   |   +-- dataset_1_2
|   |   |   +-- attribute_1_2_1
|   |   |   +-- attribute_1_2_2
|   |   |   +-- ...
|   |   |
|   |   +-- ...
|   |
|   +-- group_2
|   |   +-- dataset_2_1
|   |   |   +-- attribute_2_1_1
|   |   |   +-- attribute_2_1_2
|   |   |   +-- ...
|   |   |
|   |   +-- dataset_2_2
|   |   |   +-- attribute_2_2_1
|   |   |   +-- attribute_2_2_2
|   |   |   +-- ...
|   |   |
|   |   +-- ...
|   |
|   +-- ...
|
```

- HDF5的特性
    - 自述性
        - 对于一个HDF 文件里的每一个数据对象，有关于该数据的综合信息（元数据）
            - 在没有任何外部信息的情况下，HDF 允许应用程序解释HDF文件的结构和内容。
    - 通用性
        - 许多数据类型都可以被嵌入在一个HDF文件里
            - 例如，通过使用合适的HDF 数据结构，符号、数字和图形数据可以同时存储在一个HDF 文件里。
    - 灵活性
        - HDF允许用户把相关的数据对象组合在一起，放到一个分层结构中，向数据对象添加描述和标签。
        - 它还允许用户把科学数据放到多个HDF 文件里。
    - 扩展性
        - HDF极易容纳将来新增加的数据模式，容易与其他标准格式兼容。
    - 跨平台性
        - HDF 是一个与平台无关的文件格式。HDF 文件无需任何转换就可以在不同平台上使用。