从零开始的分布式数据库生活
(From Zero to Distributed Database)

Quhaha

目录

第一章 前言

1.1 为什么选择 ToyDB

1.2 ToyDB 的整体架构

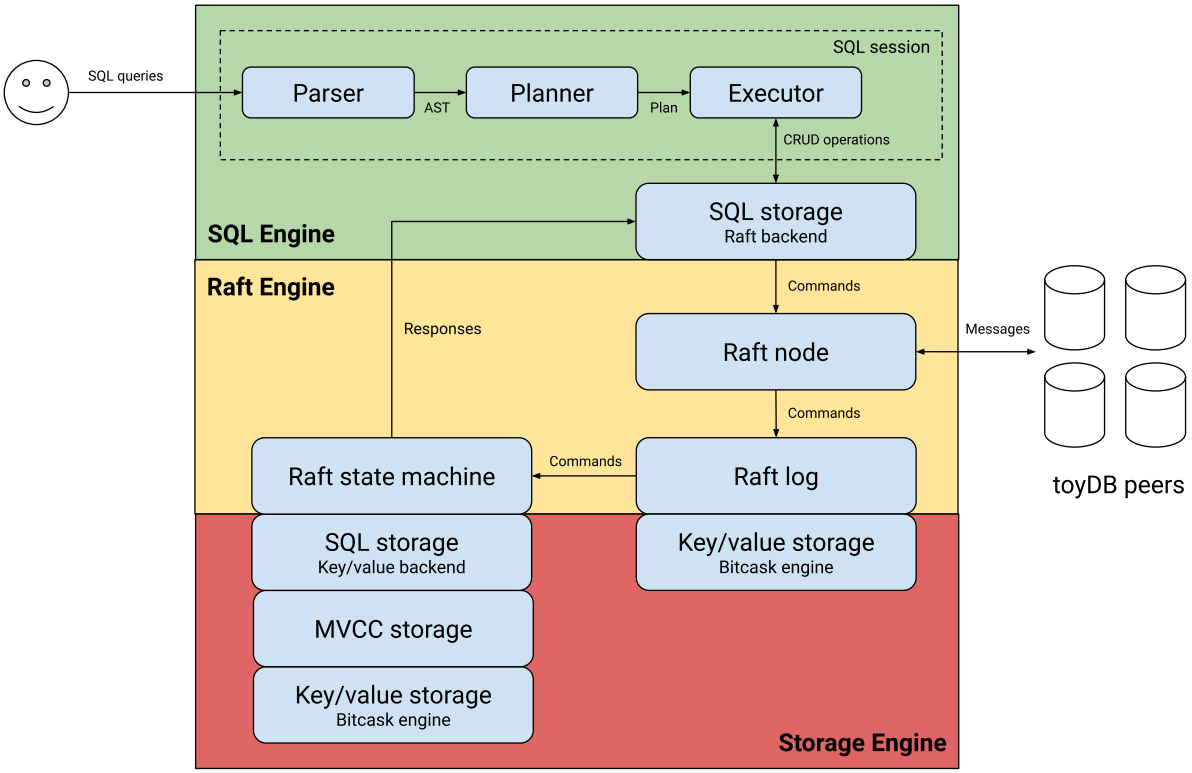

图 1-1　ToyDB 的整体架构

第二章 存储引擎

```
src/storage
├── bitcask.rs # 基于BitCask实现的存储引擎
├── engine.rs  # 存储引擎的trait
├── memory.rs  # 基于标准库中的BTree实现的存储引擎
├── mod.rs
├── mvcc.rs    # 在存储引擎之上实现MVCC事务
└── testscripts
    └── ....   # 测试文件
```

代码 2-1　存储引擎的代码结构

2.1 存储引擎介绍

存储引擎是干什么的。现在假设对面的同学是一个对数据库一无所知的同学，下面从两个方面来介绍一个存储引擎。

站在存储引擎的调用者来看，存储引擎就是一个随意使用的 HashMap。当我给定一个 key，存储引擎就会返回一个 value。或者我将 key 的值设置为 value。在稍后的 `storage::Engine` 我们将会看到。

其实上面有一个默认的假设是，存储引擎是一个线程安全的。但是在实际的系统中并不一定是线程安全的。那么如何解决呢？解决方式其实很简单，就是在存储引擎的外面加一个锁。这样就可以保证线程安全了。

那加锁是不是效率有点太低了呢？其实这个也与锁的粒度有关。如果锁的粒度太大，那么就会导致性能下降。如果锁的粒度太小，那么就会导致锁的争用。所以在实际的系统中，需要根据实际情况来选择锁的粒度。

在本书的存储引擎中，会提到一个 MVCC 的事务。MVCC 是一种多版本并发控制，它可以在不加锁的情况下实现事务的隔离。

这里又出现了一个陌生的名词，事务。事务是数据库中的一个重要概念，它是一组操作的集合，这组操作要么全部成功，要么全部失败。在数据库中，事务有四个特性，ACID。分别是原子性(Atomicity)，一致性(Consistency)，隔离性(Isolation)，持久性(Durability)。这四个特性是数据库中事务的基本要求。

当然存储引擎与编程语言中的 HashMap 还有一个不同是，它可以是持久化的，也就是说，当程序退出的时候，数据还是会保存在磁盘上。这样就可以保证数据不会丢失。

ToyDB 是一个分布式数据库，它实现的存储引擎不仅被 SQL 引擎使用，还被 Raft 引擎使用。在 Raft 引擎中，存储引擎是用来存储 Raft 的日志的。在 SQL 引擎中，存储引擎是用来存储数据的。

下面我们看一下这里存储引擎所需要实现的 trait。

2.2 Engine Trait

在存储引擎中，每一对 KV 都是以字母序来存储的字节序列（byte slice）。其中 Key 是有序的，这样就可以进行高效的范围查询。范围查询在一些场景下非常有用，比如在执行一个扫描表的 SQL 的时候（所有的行都是以相同的 key 前缀）。Key 应该使用 KeyCode 进行编码（接下来会讲到）。在写入以后，数据并没有持久化，还需要调用 flush()保证数据持久化。

在 ToyDB 中，即使是读操作，也只支持单线程，这是因为所有方法都包含一个存储引擎的可变引用。无论是 raft 的日志运用到状态机还是对文件的读取操作，都是只能顺序访问的。

在 ToyDB 中，实现了一个基于 BitCask 的，一个基于标准库 BTree 的存储引擎。

每一个可以被 ToyDB 使用的存储引擎都需要实现 `storage::Engine` 这个 trait。下面看一下这个 trait。

```
/// 带有 Self: Sized 是为了无法使用trait object(比如Box<dyn Engine>)
/// 但是也提供了一个scan_dyn()方法来返回一个trait object
pub trait Engine: Send {
    /// scan()返回的迭代器
    type ScanIterator<'a>: ScanIterator + 'a
    where
        Self: Sized + 'a;
    /// 删除一个key，如果不存在就什么都不发生
    fn delete(&mut self, key: &[u8]) -> Result<()>;
    /// 将缓冲区的内容写出到存储介质中
    fn flush(&mut self) -> Result<()>;
    fn get(&mut self, key: &[u8]) -> Result<Option<Vec<u8>>>;
    /// 遍历指定范围的key/value对
    fn scan(&mut self, range: impl std::ops::RangeBounds<Vec<u8>>) ->
Self::ScanIterator<'_>
    where
        Self: Sized;
    /// 与scan()类似，能被trait object使用. 由于使用了dynamic dispatch，所以性能会有
所下降
    fn scan_dyn(
        &mut self,
        range: (std::ops::Bound<Vec<u8>>, std::ops::Bound<Vec<u8>>),
    ) -> Box<dyn ScanIterator + '_>;
    /// 遍历指定前缀的key/value对
    fn scan_prefix(&mut self, prefix: &[u8]) -> Self::ScanIterator<'_>
    where
        Self: Sized,
    {
        self.scan(keycode::prefix_range(prefix))
```

```
    }
    /// 设置一个key/value对，如果存在则替换
    fn set(&mut self, key: &[u8], value: Vec<u8>) -> Result<()>;
    /// 获取存储引擎的状态
    fn status(&mut self) -> Result<Status>;
}
```

代码 2-2　strong::Engine

其中的 `get`， `set` 以及 `delete` 只是简单的读取以及写入 key/value 对，并且通过 `flush` 可以确保将缓冲区的内容写出到存储介质中(通过 `fsync` 系统调用等方式)。 `scan` 按照顺序迭代指定的 KV 对范围。这个对一些高级功能(SQL 表扫描等)至关重要。并且暗含了以下一些语义：

· 为了提高性能，存储的数据应该是有序的。

· key 应该保留字节编码，这样才能实现范围扫描。

对于存储引擎而已，并不关心 `key` 是什么，但是为了方便上层的调用，提供了一个称为 `KeyCode` 的 order-preserving 编码，具体可以在 第 5 章 看到。

此外在上面的代码中还需要注意两个东西，一个是 `ScanIterator`，一个是 `Status`。`ScanIterator` 是一个迭代器，用于遍历存储引擎中的 KV 对。`Status` 是用于获取存储引擎的状态，比如存储引擎的大小，垃圾量等。

现在简单看一下 `Status`。

```
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Status {
    /// The name of the storage engine.
    /// 存储引擎的名字
    pub name: String,
    /// The number of live keys in the engine.
    /// 存储引擎中的活跃key的数量
    pub keys: u64,
    /// The logical size of live key/value pairs.
    /// 存储引擎中的活跃key/value对的逻辑大小
    pub size: u64,
    /// The on-disk size of all data, live and garbage.
    /// 所有数据的磁盘大小，包括活跃和垃圾数据
    pub total_disk_size: u64,
    /// The on-disk size of live data.
    /// 活跃数据的磁盘大小
    pub live_disk_size: u64,
    /// The on-disk size of garbage data.
    /// 垃圾数据的磁盘大小
    pub garbage_disk_size: u64,
}

impl Status {
    pub fn garbage_percent(&self) -> f64 {
```

```
            if self.total_disk_size == 0 {
                return 0.0;
            }
            self.garbage_disk_size as f64 / self.total_disk_size as f64 * 100.0
        }
    }
```

代码 2-3　Status

简简单单几个属性，以及定义了一个计算垃圾比例的方法。这个比例可以用来判断是否需要进行压缩。

再看一下 `ScanIterator`。

```
/// A scan iterator, with a blanket implementation (in lieu of trait aliases).
pub trait ScanIterator: DoubleEndedIterator<Item = Result<(Vec<u8>, Vec<u8>)>> {}

impl<I: DoubleEndedIterator<Item = Result<(Vec<u8>, Vec<u8>)>>> ScanIterator for I {}
```

代码 2-4　ScanIterator

这里定义了一个 `trait`，用于遍历存储引擎中的 KV 对 (代码 2-2 中 `Scan*` 所用)。这个 `trait` 指定了 `Item` (`Item` 就是迭代项)类型为 `Result<(Vec<u8>, Vec<u8>)>`，其中 `Vec<u8>, Vec<u8>` 分别是 key，value。另外这个 `trait` 还需要组合 `DoubleEndedIterator` 这个 `trait`。

另外 `ScanIterator` 没有定义任何额外的方法，这个实现是空的，这种方式称为 blanket implementation（通用实现），这个允许我们为一大类类型提供一个统一的实现。

2.3 BitCask 存储引擎

ToyDB 使用的可持久化的存储引擎是 BitCask(参考第 6.1 节)的**变种**。简单来说，就是在写入的时候，先写入到只能追加写的 log 文件中，然后在内存中维护一个索引，索引内容为 (key -> 文件位置以及长度)。

当垃圾量(包含替换以及删除的 key)大于一定阈值的时候，将在内存中的 key 写入到新的 log 文件中，然后替换老的 log 文件。替换的过程被称为压缩，会导致写放大问题，但是可以通过控制阈值来减小影响。

通过上面的描述，可以分析出几个语义：

- BitCask 要求所有的 Key 必须能存放在内存中。
- BitCask 在启动的时候需要扫描 Log 文件来构建索引。
- 删除文件的时候并不是真正的删除。
    - 实际上是写入一个墓碑值(tombstone value)，读取到墓碑值就认为是删除了。

下面先看一下 ToyDB 的宏观架构，再自底向上的看一下实现过程。

```
struct Log {
    /// Path to the log file.
    /// 日志文件的路径
```

```
    path: PathBuf,
    /// The opened file containing the log.
    /// 包含日志的打开文件
    file: std::fs::File,
}

/// Maps keys to a value position and length in the log file.
/// 将key映射到日志文件中的value位置和长度
type KeyDir = std::collections::BTreeMap<Vec<u8>, (u64, u32)>;

pub struct BitCask {
    /// The active append-only log file.
    /// 当前的只追加写的日志文件
    log: Log,
    /// Maps keys to a value position and length in the log file.
    /// 将key映射到日志文件中的value位置和长度
    keydir: KeyDir,
}
```

代码 2-5 bitcask

在 ToyDB 中，`BitCask` 中包含一个管理内存中索引文件的数据结构 `keydir` 以及一个用来写 Log 的文件 `log` 。

其中 `keydir` 是一个 BTree，key 是一个 Vec，value 是一个元组(u64, u32)，其中 u64 是 value 在 Log 文件中的位置，u32 是 value 的长度。这个结构是有序的，这样就可以进行范围查询。

再来看 `log` ，它包含了一个文件路径 `path` 以及一个文件句柄 `file` 。这个文件是只追加写的，这样就可以保证写入的顺序是正确的。

下面先看 `log` 实现部分，再来回看 `BitCask` 的部分。

2.3.1 Log 的实现

每一个 log entry 包含四个部分:

- Key 的长度，大端 u32
- Value 的长度，大端 i32，-1 表示墓碑值
- Key 的字节序列(最大 2GB)
- Value 的字节序列(最大 2GB)

在 Log 中， `new` 比较简单，是打开一个 log 文件，当不存在的时候就创建这个文件。并且在使用过程中一直使用的排它锁，这样就可以保证只有一个线程在写入。

```
fn new(path: PathBuf) -> Result<Self> {
    if let Some(dir) = path.parent() {
        std::fs::create_dir_all(dir)?
    }
```

```
                let file = std::fs::OpenOptions::new()
                    .read(true)
                    .write(true)
                    .create(true)
                    .truncate(false)
                    .open(&path)?;
                file.try_lock_exclusive()?;
                Ok(Self { path, file })
            }
```

代码 2-6 new

`read_value`, `write_value` 这两个函数也比较简单，用于读取 value 以及写入 KV 对。

```
/// 从file的value_pos位置读取value_len长度的数据
fn read_value(&mut self, value_pos: u64, value_len: u32) -> Result<Vec<u8>> {
    let mut value = vec![0; value_len as usize];
    self.file.seek(SeekFrom::Start(value_pos))?;
    self.file.read_exact(&mut value)?;
    Ok(value)
}
```

代码 2-7 read_value

```
/// 写入key/value对，返回写入的位置和长度
/// 墓碑值使用 None Value
fn write_entry(&mut self, key: &[u8], value: Option<&[u8]>) -> Result<(u64, u32)> {
    let key_len = key.len() as u32;
    // map_or 是 Option类型的方法，用于在 Option 为 Some 以及 None 时执行不同的操作
    let value_len = value.map_or(0, |v| v.len() as u32);
    let value_len_or_tombstone = value.map_or(-1, |v| v.len() as i32);
    // 这里 4 + 4 就是 key_len(u32) 和 value_len_or_tombstone(u32) 的长度
    let len = 4 + 4 + key_len + value_len;

    let pos = self.file.seek(SeekFrom::End(0))?;
    // BufWriter 是一个带有缓冲的写操作，可以减少实际IO操作的次数
    let mut w = BufWriter::with_capacity(len as usize, &mut self.file);
    w.write_all(&key_len.to_be_bytes())?;
    w.write_all(&value_len_or_tombstone.to_be_bytes())?;
    w.write_all(key)?;
    if let Some(value) = value {
        w.write_all(value)?;
    }
    w.flush()?;

    Ok((pos, len))
}
```

代码 2-8 write_value

`build_keydir` 就比较复杂了，用来构建索引(ToyDB 只有在重启的时候才会构建)。

```rust
/// Builds a keydir by scanning the log file. If an incomplete entry is
/// encountered, it is assumed to be caused by an incomplete write operation
/// and the remainder of the file is truncated.
/// 通过扫描log文件来构建一个keydir. 如果遇到不完整的条目, 就会假设是因为不完整的写操
作
/// 并且截断文件.
fn build_keydir(&mut self) -> Result<KeyDir> {
    let mut len_buf = [0u8; 4];
    let mut keydir = KeyDir::new();
    let file_len = self.file.metadata()?.len();
    let mut r = BufReader::new(&mut self.file);
    let mut pos = r.seek(SeekFrom::Start(0))?;

    while pos < file_len {
        // Read the next entry from the file, returning the key, value
        // position, and value length or None for tombstones.
        // 读取一条新的条目, 返回key, value位置, 以及value长度或者墓碑值(None)
        let result = || -> std::result::Result<(Vec<u8>, u64, Option<u32>),
std::io::Error> {
            // r 在当前文件指针位置读取数据到 len_buf 中
            // 读取完成以后文件指针会自动向后移动 len_buf.len() 的大小
            r.read_exact(&mut len_buf)?;
            let key_len = u32::from_be_bytes(len_buf);
            r.read_exact(&mut len_buf)?;
            let value_len_or_tombstone = match i32::from_be_bytes(len_buf) {
                l if l >= 0 => Some(l as u32),
                _ => None, // -1 for tombstones
            };
            let value_pos = pos + 4 + 4 + key_len as u64;

            let mut key = vec![0; key_len as usize];
            r.read_exact(&mut key)?;

            if let Some(value_len) = value_len_or_tombstone {
                if value_pos + value_len as u64 > file_len {
                    // 这里就是遇到了不完整的条目
                    return Err(std::io::Error::new(
                        std::io::ErrorKind::UnexpectedEof,
                        "value extends beyond end of file",
                    ));
                }
                // 在当前文件指针位置移动 value_len 的大小
                // 使用 seek_relative 而不是 seek 是为了避免丢弃缓冲区
                //
                // seek 是把文件指针立刻移动到某个位置, 旧的缓冲区的数据可能和新的位
置不匹配
```

```
                // 所以缓冲失效会被丢弃
                r.seek_relative(value_len as i64)?; // avoids discarding buffer
            }

            Ok((key, value_pos, value_len_or_tombstone))
        }();

        match result {
            // Populate the keydir with the entry, or remove it on tombstones.
            // 填充 keydir，或者在墓碑值的时候删除
            Ok((key, value_pos, Some(value_len))) => {
                keydir.insert(key, (value_pos, value_len));
                pos = value_pos + value_len as u64;
            }
            Ok((key, value_pos, None)) => {
                keydir.remove(&key);
                pos = value_pos;
            }
            // If an incomplete entry was found at the end of the file, assume an
            // incomplete write and truncate the file.
            // 这里就是遇到了不完整的条目
            Err(err) if err.kind() == std::io::ErrorKind::UnexpectedEof => {
                log::error!("Found incomplete entry at offset {}, truncating
file", pos);
                self.file.set_len(pos)?;
                break;
            }
            Err(err) => return Err(err.into()),
        }
    }
    Ok(keydir)
}
```

代码 2-9　build_keydir

2.3.2 BitCask 的实现

在知道了 `log` 是如何实现的以后，就可以更好的理解 BitCask 的实现了。回忆一下，代码 2-5 中，`BitCask` 中包含了一个 `log` 以及一个 `keydir`。`log` 用来写入 KV 对，`keydir` 用来维护内存中的索引。

下面先看一下 `BitCask` 中的一些周边函数，然后再看一下如何实现 `Engine` 这个 trait。

先来看看 `BitCask` 的构造函数以及析构函数，`new` 和 `new_compact`。这个两个函数的区别就是 `new_compact` 会在打开的时候自动压缩。关于析构函数，会在 Drop 的时候尝试 flush 文件。

```
/// Opens or creates a BitCask database in the given file.
/// 通过 path 打开或者创建一个 BitCask 数据库
```

```rust
pub fn new(path: PathBuf) -> Result<Self> {
    // 这里非常简单，就是调用前面实现的 Log::new
    log::info!("Opening database {}", path.display());
    let mut log = Log::new(path.clone())?;
    let keydir = log.build_keydir()?;
    log::info!("Indexed {} live keys in {}", keydir.len(), path.display());
    Ok(Self { log, keydir })
}

/// Opens a BitCask database, and automatically compacts it if the amount
/// of garbage exceeds the given ratio and byte size when opened.
/// 打开一个 BitCask 数据库，如果打开的时候垃圾的比例和字节大小超过给定的阈值，就会自动压缩
pub fn new_compact(
    path: PathBuf,
    garbage_min_fraction: f64,
    garbage_min_bytes: u64,
) -> Result<Self> {
    let mut s = Self::new(path)?;

    let status = s.status()?;
    if Self::should_compact(
        status.garbage_disk_size,
        status.total_disk_size,
        garbage_min_fraction,
        garbage_min_bytes,
    ) {
        log::info!(
            "Compacting {} to remove {:.0}% garbage ({} MB out of {} MB)",
            s.log.path.display(),
            status.garbage_percent(),
            status.garbage_disk_size / 1024 / 1024,
            status.total_disk_size / 1024 / 1024
        );
        s.compact()?;
        log::info!(
            "Compacted {} to size {} MB",
            s.log.path.display(),
            (status.total_disk_size - status.garbage_disk_size) / 1024 / 1024
        );
    }

    Ok(s)
}

/// Returns true if the log file should be compacted.
```

```
/// 如果日志文件应该被压缩，就返回 true
fn should_compact(
    garbage_size: u64,
    total_size: u64,
    min_fraction: f64,
    min_bytes: u64,
) -> bool {
    let garbage_fraction = garbage_size as f64 / total_size as f64;
    garbage_size > 0 && garbage_size ⩾ min_bytes && garbage_fraction ⩾ min_fraction
}

/// Attempt to flush the file when the database is closed.
/// 在 Drop 的时候尝试 flush 文件
impl Drop for BitCask {
  fn drop(&mut self) {
      if let Err(error) = self.flush() {
          log::error!("failed to flush file: {}", error)
      }
  }
}
```

代码 2-10　impl Bitcask

上面几个函数比较简单，下面看一下在压缩的时候会使用的函数 `compact` 以及 `write_log` 。

```
impl BitCask {
    /// Compacts the current log file by writing out a new log file containing
    /// only live keys and replacing the current file with it.
    /// 压缩当前的日志文件，写出一个新的日志文件，只包含活跃的 key，并且用它替换当前
的文件
    pub fn compact(&mut self) -> Result<()> {
        let mut tmp_path = self.log.path.clone();
        tmp_path.set_extension("new");
        let (mut new_log, new_keydir) = self.write_log(tmp_path)?;

        std::fs::rename(&new_log.path, &self.log.path)?;
        new_log.path = self.log.path.clone();

        self.log = new_log;
        self.keydir = new_keydir;
        Ok(())
    }

    /// Writes out a new log file with the live entries of the current log file
    /// and returns it along with its keydir. Entries are written in key order.
    /// 写出一个新的日志文件，包含当前日志文件中的活跃条目，并且返回它以及它
的 keydir.
    fn write_log(&mut self, path: PathBuf) -> Result<(Log, KeyDir)> {
```

```
        let mut new_keydir = KeyDir::new();
        let mut new_log = Log::new(path)?;
        new_log.file.set_len(0)?; // truncate file if it exists
        for (key, (value_pos, value_len)) in self.keydir.iter() {
            let value = self.log.read_value(*value_pos, *value_len)?;
            let (pos, len) = new_log.write_entry(key, Some(&value))?;
            new_keydir.insert(key.clone(), (pos + len as u64 - *value_len as
u64, *value_len));
        }
        Ok((new_log, new_keydir))
    }
}
```

代码 2-11　compact / write_log

这里也还是比较简单的，`write_log` 函数会遍历 keydir，将活跃的 key/value 对写入到新的 log 文件中。`compact` 函数会调用 `write_log` 获取新的 log 文件，最后将新的 log 文件替换掉老的 log 文件。

最后我们看一下 Bitcask 对 `Engine` 这个 trait 的实现。

```
impl Engine for BitCask {
    type ScanIterator<'a> = ScanIterator<'a>;

    fn delete(&mut self, key: &[u8]) -> Result<()> {
        self.log.write_entry(key, None)?;
        self.keydir.remove(key);
        Ok(())
    }

    fn flush(&mut self) -> Result<()> {
        // Don't fsync in tests, to speed them up. We disable this here, instead
        // of setting raft::Log::fsync = false in tests, because we want to
        // assert that the Raft log flushes to disk even if the flush is a noop.
        #[cfg(not(test))]
        self.log.file.sync_all()?;
        Ok(())
    }

    fn get(&mut self, key: &[u8]) -> Result<Option<Vec<u8>>> {
        if let Some((value_pos, value_len)) = self.keydir.get(key) {
            Ok(Some(self.log.read_value(*value_pos, *value_len)?))
        } else {
            Ok(None)
        }
    }

    fn scan(&mut self, range: impl std::ops::RangeBounds<Vec<u8>>) ->
```

```
Self::ScanIterator<'_> {
        ScanIterator { inner: self.keydir.range(range), log: &mut self.log }
    }

    fn scan_dyn(
        &mut self,
        range: (std::ops::Bound<Vec<u8>>, std::ops::Bound<Vec<u8>>),
    ) -> Box<dyn super::ScanIterator + '_> {
        Box::new(self.scan(range))
    }

    fn set(&mut self, key: &[u8], value: Vec<u8>) -> Result<()> {
        let (pos, len) = self.log.write_entry(key, Some(&*value))?;
        let value_len = value.len() as u32;
        self.keydir.insert(key.to_vec(), (pos + len as u64 - value_len as u64,
value_len));
        Ok(())
    }

    fn status(&mut self) -> Result<Status> {
        let keys = self.keydir.len() as u64;
        let size = self
            .keydir
            .iter()
            .fold(0, |size, (key, (_, value_len))| size + key.len() as u64 +
*value_len as u64);
        let total_disk_size = self.log.file.metadata()?.len();
        // 8 * keys: key 是 u64, 所以是 8 * key的数量
        let live_disk_size = size + 8 * keys; // account for length prefixes
        let garbage_disk_size = total_disk_size - live_disk_size;
        Ok(Status {
            name: "bitcask".to_string(),
            keys,
            size,
            total_disk_size,
            live_disk_size,
            garbage_disk_size,
        })
    }
}
```

代码 2-12

这里可以看得出来，`BitCask` 实现了 `delete`，`flush`，`get`，`scan`，`set`，`status` 这几个方法。大多数都是调用了 `log` 的方法。

- `delete` 会将 key/value 对写入到 log 文件中，并且在 keydir 中删除这个 key
- `flush` 会将缓冲区的内容写出到存储介质中

- `get` 会从 keydir 中获取 value 的位置以及长度，然后从 log 文件中读取 value
- `scan` 会返回一个 `ScanIterator`，用于遍历 keydir
- `set` 会将 key/value 对写入到 log 文件中，并且在 keydir 中插入这个 key
- `status` 会返回存储引擎的状态

2.3.3 ToyDB 中 BitCask 的取舍

在 ToyDB 中，BitCask 的实现做了相当程度的简化：

1. ToyDB 没有使用固定大小的日志文件，而是使用了任意大小的仅追加写的日志文件。这会增加压缩量，因为每次压缩的时候都会重写整个日志文件，并且也可能会超过文件系统的文件大小限制
2. 压缩的时候会阻塞所有的读以及写操作，这问题不大，因为 ToyDB 只会在重启的时候压缩，并且文件应该也比较小
3. 没有 hint 文件，因为 ToyDB 的 value 预估都比较小，hint 文件的作用不大(其大小与合并的 Log 文件差不多大)
4. 每一条记录没有 timestamps 以及 checksums
5. BitCask 需要 key 的集合在内存中，而且启动的时候需要扫描 log 文件来构建索引
6. 与 LSMTree 不同，单个文件的 BitCask 需要在压缩的过程中重写整个数据集，这会导致显著的写放大问题
7. ToyDB 没有使用任何压缩，比如可变长度的整数

2.4 Memory 存储引擎

TODO： 比较简单，可以直接看代码。

2.5 MVCC 介绍

MVCC[1]是广泛使用的并发控制机制，他为 ACID 事务提供快照隔离[2]，能实现读写并发。快照隔离级别是最高的隔离级别么? 并不是，最高的隔离级别是可串行化隔离级别，但是可串行化隔离级别会导致性能问题，所以快照隔离级别是一个折中的选择。

快照隔离级别会导致写倾斜(Write Skew)问题[3]，但是这个问题在实际中并不常见。详细的讨论可以参考第 6.2.1 节。

2.5.1 MVCC 的实现

ToyDB 在存储层实现了 MVCC, 可以使用任何实现了 `storage::Engine` 存储引擎作为底层存储。在 ToyDB 中，提供了常见的事务操作：

[1]Multi-Version Concurrency Control： https://zh.wikipedia.org/wiki/多版本并发控制

[2]https://jepsen.io/consistency/models/snapshot-isolation

[3]https://justinjaffray.com/what-does-write-skew-look-like/

- `begin`： 开始一个新的事务。(可以是只读事务(read only)，也可以是读写事务)
- `get`： 读取一个 key 的值
- `set`： 设置一个 key 的值
- `delete`： 删除一个 key 的值
- `scan`： 遍历指定范围的 key/value 对
- `scan_prefix`： 遍历指定前缀的 key/value 对
- `commit`： 提交一个事务。(保留更改，对其他事务可见)
- `rollback`： 回滚一个事务。(丢弃更改)

MVCC，顾名思义，是通过多个版本来管理事务的。版本的先后通过 `Version` 来界定，可以说， `Version` 就是一个逻辑时间戳。通过 `Version` 用来标记每一个写入的 `Key` ，就可以在一个确定的时刻(`Version` 一定的时刻)，知道一个 `Key` 的值是什么。

```
// x 表示 tombstone 值
时间(Version)
5
4  a4
3      b3       x
2
1  a1       c1  d1
   a    b    c   d    Key
```

上面就是一个例子，在 `Version = 2` 的时候，可以看到 `a` 的值是 `a1` ， `c` 的值是 `c1` ， `d` 的值是 `d1` 。在 `Version = 4` 的时候， `a` 的值是 `a4` ， `b` 的值是 `b3` ， `c` 的值是 `c1` 。(没提到就是空值)

这里有必要说一下，上面一段话中多次提到 `Key` ，但是表示有不同的意思。 `Key` 分为存储引擎中的 `Key` 和事务管理(MVCC)中的 `Key` 枚举类。在本书后续部分，大家需要根据上下文来理解。

事务管理的 `Key` 枚举类的值会通过序列化成存储引擎中的 `Key` 来存储。这一点对于理解下面 MVCC 的实现非常重要。关于序列化的实现，会在 第 5 章 详细说明[4]，这里只需要知道实现了 `serde` 的 `Deserialize` 和 `Serialize` 两个 `trait` 就可以通过 `encode()` 将一个对象序列化成 `vec<u8>` ，通过 `decode()` 把 `vec<u8>` 反序列化成一个对象就可以了。

另外，通过 `serde` 的 `with = "serde_bytes"` 和 `borrow` 可以将 `Cow` 类型的数据序列化成 `vec<u8>` ，这个在 `Key` 枚举类中会用到。

下面来看一下 `Key` 枚举类的定义。

```
/// MVCC keys, using the KeyCode encoding which preserves the ordering and
/// grouping of keys. Cow byte slices allow encoding borrowed values and
/// decoding into owned values.
/// MVCC keys，使用 KeyCode 编码，保留了 key 的顺序和分组.
/// Cow 允许对 borrowed 的值进行编码，并且解码成 owned 的值.
#[derive(Debug, Deserialize, Serialize)]
```

[4]关于序列化也可以阅读一下： https://research.cs.wisc.edu/mist/SoftwareSecurityCourse/Chapters/3_5-Serialization.pdf

```rust
pub enum Key<'a> {
    /// The next available version.
    /// 下一个可以使用的 version.
    NextVersion,
    /// Active (uncommitted) transactions by version.
    /// 在给定 version 下, 活动的(未提交的)事务.
    TxnActive(Version),
    /// A snapshot of the active set at each version. Only written for
    /// versions where the active set is non-empty (excluding itself).
    /// 每个 version 下的活动集的快照. 只有在活动集非空的时候才会写入.
    TxnActiveSnapshot(Version),
    /// Keeps track of all keys written to by an active transaction (identified
    /// by its version), in case it needs to roll back.
    /// 用于跟踪所有被活动事务(通过 version 标识)写入的 key, 以防需要回滚.
    TxnWrite(
        Version,
        #[serde(with = "serde_bytes")]
        #[serde(borrow)]
        Cow<'a, [u8]>,
    ),
    /// A versioned key/value pair.
    /// 一个带有 version 的 key/value 对.
    Version(
        #[serde(with = "serde_bytes")]
        #[serde(borrow)]
        Cow<'a, [u8]>,
        Version,
    ),
    /// Unversioned non-transactional key/value pairs. These exist separately
    /// from versioned keys, i.e. the unversioned key "foo" is entirely
    /// independent of the versioned key "foo@7". These are mostly used
    /// for metadata.
    /// 非事务的 key/value 对. 这些和带有 version 的 key 是独立的.
    /// 例如, unversioned key "foo" 和 versioned key "foo@7" 是完全独立的.
    /// 这些主要用于元数据.
    Unversioned(
        #[serde(with = "serde_bytes")]
        #[serde(borrow)]
        Cow<'a, [u8]>,
    ),
}
```

代码 2-13　Key 枚举类

看到这里应该看的一头雾水，不过没有关系，再粗略看一下 `KeyPrefix`，然后再统一解释。

```rust
/// MVCC key prefixes, for prefix scans. These must match the keys above,
/// including the enum variant index.
```

```
/// MVCC key 的前缀，用于前缀扫描. 这些必须和上面的 key 匹配，包括枚举变量的索引.
#[derive(Debug, Deserialize, Serialize)]
enum KeyPrefix<'a> {
    NextVersion,
    TxnActive,
    TxnActiveSnapshot,
    TxnWrite(Version),
    Version(
        #[serde(with = "serde_bytes")]
        #[serde(borrow)]
        Cow<'a, [u8]>,
    ),
    Unversioned,
}
```

代码 2-14　KeyPrefix 枚举类

现在来理解一下 `Key` 和 `KeyPrefix`。它们都实现了 `Deserialize/Serialize` 这两个 trait，所以里面的枚举值就可以序列化和反序列化。换句话说，就是可以将 `Key` 和 `KeyPrefix` 中的枚举值序列化成 `vec<u8>`，然后存储到存储引擎中。

`NextVersion`：顾名思义，表示的是下一个 Version。当事务开始的时候，会从 `NextVersion` 获取下一个可用的 version 并且递增它。

`TxnActive`：表示的是活动中事务

`TxnActiveSnapshot`：表示的是活动事务的快照

`TxnWrite`：

`Version`：

`Unversioned`：

其实在测试用例中说明了问题。

```
#[test_case(KeyPrefix::NextVersion, Key::NextVersion; "NextVersion")]
#[test_case(KeyPrefix::TxnActive, Key::TxnActive(1); "TxnActive")]
#[test_case(KeyPrefix::TxnActiveSnapshot,
Key::TxnActiveSnapshot(1); "TxnActiveSnapshot")]
#[test_case(KeyPrefix::TxnWrite(1), Key::TxnWrite(1, b"foo".as_slice().into());
"TxnWrite")]

#[test_case(KeyPrefix::Version(b"foo".as_slice().into()),
Key::Version(b"foo".as_slice().into(), 1); "Version")]

#[test_case(KeyPrefix::Unversioned, Key::Unversioned(b"foo".as_slice().into());
"Unversioned")]
fn key_prefix(prefix: KeyPrefix, key: Key) {
    let prefix = prefix.encode();
    let key = key.encode();
```

```
        assert_eq!(prefix, key[..prefix.len()])
    }
```

代码 2-15　Key 与 KeyPrefix 的测试用例

看完了 `Key` 和 `KeyPrefix`，就来具体看一下 MVCC 的实现了。不过还有一个前置点是需要理解 ToyDB 中 `MVCC` 与 `Transaction` 的关系。

先看一下 MVCC 的代码。

```
/// An MVCC-based transactional key-value engine. It wraps an underlying storage
/// engine that's used for raw key/value storage.
///
/// While it supports any number of concurrent transactions, individual read or
/// write operations are executed sequentially, serialized via a mutex. There
/// are two reasons for this: the storage engine itself is not thread-safe,
/// requiring serialized access, and the Raft state machine that manages the
/// MVCC engine applies commands one at a time from the Raft log, which will
/// serialize them anyway.
/// 基于 MVCC 的事务键值引擎. 它包装了一个底层的存储引擎, 用于原始的 key/value 存储.
/// 虽然它支持任意数量的并发事务, 但是单个读写操作是顺序执行的, 通过互斥锁串行化.
/// 有两个原因: 存储引擎本身不是线程安全的, 需要串行访问, 以及管理 MVCC 引擎的 Raft
状态机
/// 会从 Raft 日志中一次应用一个命令, 这样也会将它们串行化.
pub struct MVCC<E: Engine> {
    pub engine: Arc<Mutex<E>>,
}

impl<E: Engine> MVCC<E> {
    /// Creates a new MVCC engine with the given storage engine.
    /// 使用给定的存储引擎创建一个新的 MVCC 引擎.
    pub fn new(engine: E) -> Self { Self { engine: Arc::new(Mutex::new(engine)) } }
    /// Begins a new read-write transaction.
    /// 开始一个新的读写事务.
    pub fn begin(&self) -> Result<Transaction<E>>
{ Transaction::begin(self.engine.clone()) }
    /// Begins a new read-only transaction at the latest version.
    /// 开始一个新的只读事务, 在最新的 version 下.
    pub fn begin_read_only(&self) -> Result<Transaction<E>> {
        Transaction::begin_read_only(self.engine.clone(), None)
    }
    /// Begins a new read-only transaction as of the given version.
    /// 开始一个新的只读事务, 在给定的 version 下.
    pub fn begin_as_of(&self, version: Version) -> Result<Transaction<E>> {
        Transaction::begin_read_only(self.engine.clone(), Some(version))
    }
    /// Resumes a transaction from the given transaction state.
    /// 从给定的事务状态恢复事务.
```

```
    pub fn resume(&self, state: TransactionState) -> Result<Transaction<E>> {
        Transaction::resume(self.engine.clone(), state)
    }
    /// Fetches the value of an unversioned key.
    /// 获取一个非版本化 key 的值.
    pub fn get_unversioned(&self, key: &[u8]) -> Result<Option<Vec<u8>>> {
        self.engine.lock()?.get(&Key::Unversioned(key.into()).encode())
    }
    /// Sets the value of an unversioned key.
    /// 设置一个非版本化 key 的值.
    pub fn set_unversioned(&self, key: &[u8], value: Vec<u8>) -> Result<()> {
        self.engine.lock()?.set(&Key::Unversioned(key.into()).encode(), value)
    }
    /// Returns the status of the MVCC and storage engines.
    /// 返回 MVCC 和存储引擎的状态.
    pub fn status(&self) -> Result<Status> {
        let mut engine = self.engine.lock()?;
        let versions = match engine.get(&Key::NextVersion.encode())? {
            Some(ref v) => Version::decode(v)? - 1,
            None => 0,
        };
        let active_txns = engine.scan_prefix(&KeyPrefix::TxnActive.encode()).count()
as u64;
        Ok(Status { versions, active_txns, storage: engine.status()? })
    }
}

/// MVCC engine status.
/// MVCC 引擎状态.
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Status {
    /// The total number of MVCC versions (i.e. read-write transactions).
    /// MVCC 版本的总数(即读写事务).
    pub versions: u64,
    /// Number of currently active transactions.
    /// 当前活动事务的数量.
    pub active_txns: u64,
    /// The storage engine.
    /// 存储引擎.
    pub storage: super::engine::Status,
}
```

代码 2-16　MVCC

在代码中可以看到，MVCC 中更多的对 Transaction 以及存储引擎的一层封装。在一个事务想要开启的时候，先通过 `MVCC` 来开启一个（读写/只读）事务，之后关于事务的处理就不再属于 `MVCC` 来管理，转而到了稍后会讲到的 `Trancation` 中。

当事务开始的时候，从 `Key::NextVersion` 获取下一个可用的 version 并且递增它，然后通过 `Key::TxnActive(version)` 将自身记录为活动中的事务。它还会将当前活动的事务做一个快照，其中包含了事务开始的时候其他所有活动事务的 version，并且将起另存为 `Key::TxnActiveSnapshot(id)` 。

key/value 保存为 `Key::Version(key, version)` 的形式，其中 `key` 是用户提供的 key，`version` 是事务的版本。事务的 key/value 的可见性如下：

- 对于给定的 key，从当前事务的版本开始对 `Key::Version(key, version)` 进行反向的扫描。
- 如果一个版本位于活动集(active set)中的时候，跳过这个版本。
- 返回第一个匹配记录（如果有的话），这个记录可能是 `Some(value)` 或者 `None` 。

写入 key/value 的时候，事务首先要扫描其不可见的 `Key::Version(key, version)` 来检查是否存在任何冲突。如果有找到一个，那么需要返回序列化错误，调用者必须重试这个事务。如果没有找到，事务就会写入新记录，并且以 `Key::TxnWrite(version, key)` 的形式来跟踪更改，以防必须回滚的情况。

当事务提交的时候，就只需要删除其 `Txn::Active(id)` 记录，使其更改对其他后续的事务可见就可以了。如果事务回滚，就遍历所有的 `Key::TxnWrite(id, key)` 记录，并删除写入的 key/value 值，最后删除 `Txn::Active(id)` 记录就可以了。

这个方案可以保证 ACID 事务的快照隔离： 提交是原子的，每一个事务在开始的时候，看到的都是 key/value 存储的一致性快照，并且任何写入冲突都会导致序列化冲突，必须重试。

为了实现时间穿梭查询，只读事务只需加载过去事务的 `Key::TxnActiveSnapshot` 记录就可以了，可见性规则和普通事务是一样的。

2.5.2 MVCC 在 ToyDB 中的取舍

1. 旧的 MVCC 版本永远不会被删除，会导致存储空间的浪费。但是这简化了实现，也允许完整的数据历史记录。
2. 事务 id 会在 64 位后溢出，没有做处理。

2.6 总结

第三章 共识算法 Raft

```
src/raft
├── log.rs # 定义了Raft中的Log
├── message.rs # 定义了Raft中Node之间交互的信息
├── mod.rs
├── node.rs # 定义了Node(Leader, Follower, Candidate)以及其行为
├── state.rs # 定义了Raft中的状态机
└── testscripts
    └── ...
```

代码 3-1　Raft 算法

3.1 Raft 介绍

Raft 共识算法是一种分布式一致性算法, 它的设计目标是提供一种易于理解的一致性算法。Raft 算法分为三个部分：领导选举、日志复制和安全性。具体的实现可以参考 Raft 论文 [5][6][7]。

Raft 有三个特点：

1. 线性一致性（强一致性）：一旦一个数据被 Client 提交，那么所有的 Client 都能读到这个数据。（就是永远看不到过时的数据）
2. 容错性：知道大多数节点在运行，就可以保证系统正常运行。
3. 持久性：只要大多数节点在运行，那么写入就不会丢失。

Raft 算法通过选举一个 Leader 来完成 Client 的请求以及将数据复制到其他节点。请求一旦被大多数节点所确认后就会执行。如果 Leader 挂了，那么会重新选举一个 Leader。在一个集群中，需要至少有三个节点来保证系统的正常运行。

有一点需要说明，Raft 并不提供水平扩展。当 Client 有一个请求时，只能由单一的 Leader 来处理, 这个 Leader 很快就会称为系统的瓶颈。而且每一个节点会存储整个数据集完整的副本。

系统通常会将数据分片到多个 Raft 集群中，并在它们之间使用分布式事务协议来处理这个问题。不过与现在的这一章关系不大。

3.1.1 Raft 中日志与状态机

Raft 维护了 Client 提交的任意写命令的有序 Log。Raft 通过将 Log 复制到大多数节点来达成共识。如果能共识成功，就认为 Log 是已经提交的，并且在提交之前的 Log 都是不可变的。

当 Log 已经提交以后, Raft 就可以将 Log 中存储的命令应用到节点本地的状态机。每个 Log 中包含了 index、term 以及命令。

· index：Log 的索引，表示这个 entry 在 Log 中的位置。

[5]Raft Paper：https://raft.github.io/raft.pdf

[6]Raft Thesis：https://web.stanford.edu/~ouster/cgi-bin/papers/OngaroPhD.pdf

[7]Raft 官网：https://raft.github.io

· term：当前 entry 被 Leader 创建的时候的任期。
· 命令：会被应用在状态机的命令。

注意，通过 index 和 term 可以**确定**一个 entry。

表 3-1 Log 可视化

Index	Term	Command(命令)
1	1	`None`
2	1	`CREATE TABLE table (id INT PRIMARY KEY, value STRING)`
3	1	`INSERT INTO table VALUES (1, "foo")`
4	2	`None`
5	2	`UPDATE table SET value = 'bar' WHERE id = 1`
6	2	`DELETE FROM table WHERE id = 1`

状态机必须是确定的，只有这样，才能所有的节点达到相同的状态。Raft 将会在所有节点上独立的，以相同的顺序应用相同的命令。但是如果命令具有非确定性行为（比如随机数生成、与外部通信），会产生分歧。

3.1.2 Leader 选举

在 Raft 中，每一个节点都处于三个身份中的一个：

1. Leader：处理所有的 Client 请求，以及将 Log 复制到其他节点。
2. Follower：只是简单的响应 Leader 的请求。可能并不知道 Leader 是谁。
3. Candidate：在想要选举 Leader 时，节点会变成 Candidate。

Raft 算法都依赖一个单一的保证：在任何时候都只能有一个有效的 Leader（旧的、已经被替换掉的可能仍然认为自己是 Leader，但是不起作用）。Raft 通过领导者选举机制来强制保证这一点。

Raft 将时间划分为 term，term 与时间一样，是一个严格单调递增的值。在一个 term 内只能有一个 Leader，而且不会改变。每一个节点会存储当前已知的最后 term。并且在节点直接发送信息的时候会附带当前 term（如果收到大于当前 term 的会变成 Follower，收到小于当前 term 的会忽略）。

可以把在 Leader 选举过程分成两个部分：

1. 谁想成为 Leader
2. 如何给想成为 Leader 的 Node 投票

如果 Follower 在选举超时时间内没有收到 Leader 的心跳信息，会变成 Candidate 并且开始选举 Leader。如果一个节点与 Leader 失联（可能是网络环境等原因），那么他会不断的自行选举，直到网络恢复。因为他不断的自行选举，他的 term 会变得非常大，一旦恢复以后会大于集群中的其他节点，此时会在其任期内进行选举，也就是扰乱了当前的领导者（本来集群正常运行，每一个突然收到了特别大的 term 信息的节点，都会变成 Follower，然后整个集群被迫重新开始选举），为了解决这个问题，提出了 PreVote（Raft+

每个节点刚开始的时候都是 Follower，并且都是不知道谁是 Leader。如果收到来自当前或者更大 term 的 Leader 发来的信息，会变成知道 Leader 是谁的 Follower。否则，在等待一段时间（根据定义的选举超时时间）后，会变成 Candidate 并且开始选举 Leader。

Candidate 会将自己的 term 号+1，并且向其他节点发送投票请求。收到请问投票的信息后，节点开始相应投票。每个节点只能投一次票，先到先得，并且有一个隐藏的语义是：Candidate 会投给自己一票。

当 Candidate 收到大多数节点（>50%）的投票后，就会变成 Leader。然后向其他节点发送心跳信息，以断言并且保持自己的 Leader 身份。所有节点在收到 Leader 的心跳信息后，会变成 Follower（无论当时投票给了谁）。Leader 还会定期发送心跳信息，以保持自己的 Leader 身份，新 Leader 还会一个空 item，以便安全提前前面 term 的 item（Raft paper 5.4.2）。

有成功就会有失败，可能因为平局或者多数节点失联，在选举超时时间内没有选出 Leader。这时候会重新开始选举（term+1），直到选出 Leader。

为了避免多次平局，Raft 引入了随机化的选举超时时间。这样可以避免多个节点在同一时间内开始选举（Raft paper 5.2）。

如果 Follower 在选举超时时间内没有收到 Leader 的心跳信息，会变成 Candidate 并且开始选举 Leader。如果一个节点与 Leader 失联（可能是网络环境等原因），那么他会不断的自行选举，直到网络恢复。因为他不断的自行选举，他的 term 会变得非常大，一旦恢复以后会大于集群中的其他节点，此时会在其任期内进行选举，也就是扰乱了当前的领导者（本来集群正常运行，每一个突然收到了特别大的 term 信息的节点，都会变成 Follower，然后整个集群被迫重新开始选举），为了解决这个问题，提出了 PreVote（Raft thesis 4.2.3）。不过这就属于 Raft 的优化问题了，会在我的另外一本书中讨论。

3.1.3 日志复制与共识

当 Leader 收到了来自 Client 的写请求的时候，会将这个请求追加到其本地的 Log 中，然后将这个请求发送给其他节点。其他节点会尝试将收到的 item 也追加到自己的 Log 中，并且向 Leader 发送相应信息。

一旦大多数节点确认了追加操作，Leader 就会提交提交这个 item，并且应用到本地状态机，然后将结果返回给 Client。

然后在下一个心跳的时候，告诉其他节点这个 item 已经提交了，其他节点也会将这个 item 提交并且应用到本地状态机。关于这个的正确性不是必须的（因为其成为 Leader，它们也会提交并且应用该 item，否则也没必要应用它）。

Follower 也有可能比不会应用这个 item，可能因为落后于 Leader、日志出现了分歧等（Raft paper 5.3 节）等情况。

Leader 发送给其他 Node 的 Append 中包含了 item 的 index 以及 term, index+term（如表 3-1 所示），如果两个 item 拥有相同的 index+term，那么这两个 item 是相同的，并且之前的 item 也是相同的（Raft Paper 5.3）。

如果 Follower 收到了 index+term 不匹配的 item，会拒绝这个 item。当 Leader 收到了拒绝信息，会尝试找一个与 Follower 的日志相同的点，然后将这个点之后的 item 发送给 Follower。这样可以保证 Follower 的日志与 Leader 的日志相同。

Leader 会通过发送一个只包含 index+term 的信息来探测这一点，也就是逐个较小 index 来探测，直到 Follower 相应匹配。然后发送这个点之后的 item（Raft paper 5.3）。

Leader 的心跳信息也会包含 last_index 以及 term，如果 Follower 的日志落后于 Leader，会在给心跳的返回信息中说明，Leader 会像上面一样发送日志。

3.1.4 Client 的请求

Client 会将请求提交给本地的 Raft 节点。但是就像上面提到的，它们仅仅在 Leader 上处理，所以 Followers 会将请求转发给 Leader（Raft thesis 6.2）。

关于写请求，会被 Leader 追加到 Log 中，然后发送给其他节点。一旦大多数节点确认了这个请求，Leader 就会提交这个请求，并且应用到本地状态机。一旦应用到状态机中，Leader 会通过日志查找的写请求的结果，并且返回给 Client。确定性的错误（外键违规）会返回给客户端，非确定性的错误（IO 错位）会直接使节点崩溃（这样就可以避免副本状态产生分歧）。

关于读请求，仅在 Leader 上处理，不需要 Raft 的日志的复制。但是为了确保强一致性，Leader 会在处理读请求的时候，会通过一次心跳来避免脏读（避免其他地方选举出来了新的 Leader 而导致的 Leader 身份失效）。一旦可以确认 Leader 的身份，Leader 就会将读取结果返回给 Client。

3.2 Raft 之 Message

```
/// 发送者和接收者的消息信封(在Message上包装了一层额外信息).
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub struct Envelope {
    pub from: NodeID,      // 从哪里来
    pub term: Term,        // 发送者的当前term
    pub to: NodeID,        // 到哪里去
    pub message: Message, // 具体发送的信息
}

/// Raft的Node之间的消息，消息是异步发送的，可能会丢失或者重排序.
/// 在实践中，它们通过TCP连接发送，并通过crossbeam通道确保消息不会丢失或重排序，前提是连接保持完好.
/// 消息的发送以及回执都是通过单独的TCP连接.
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
```

```
pub enum Message {
    /// Candidates 从其他Node中获取投票，以便成为Leader.
    /// 只有当Candidate的日志至少与投票者一样新时，才会授予投票.
    Campaign {
        last_index: Index, // Candidate最后一个日志的index
        last_term: Term,   // Candidate最后一个日志的term
    },

    /// Followers 只有在Candidate的日志至少与自己一样新时，才会投票.
    /// Canidate会隐式的投票给自己.
    CampaignResponse {
        /// true表示投票给Candidate，false表示拒绝投票.
        /// 拒绝投票不是必须的，但是为了清晰起见，还是会发送.
        vote: bool,
    },

    /// Leader 会定期发送心跳，这样有几个目的:
    /// 1. 通知节点当前的 Leader，防止选举.
    /// 2. 检测丢失的 appends和 reads，作为重试机制.
    /// 3. 告知Followers 已经提交的 indexes，以便他们可以应用 entries.
    /// Raft论文中使用了一个空的AppendEntries RPC来实现心跳，但是我们选择添加一个以更好的分离心跳消息.
    Heartbeat {
        /// last_index 是 Leader 最后一个日志的index，Term 是Leader 的当前 term.
        /// 因为它在选举成功后会追加一个空的entry. Follower 会将这个与自己的日志进行比较，以确定是否是最新的.
        last_index: Index,
        /// 表示的是 Leader 的最后一个提交的日志的 index.
        /// Followers 会使用这个来推进他们的 commit index，并应用 entries.
        /// (只有在本地日志与 last_index 匹配时才能安全地提交这个).
        commit_index: Index,
        /// Leader在这个term中的最新读序列号.
        read_seq: ReadSequence,
    },

    /// Followers 回应 Leader 的心跳，以便 Leader 知道自己还是 Leader.
    HeartbeatResponse {
        /// 如果不为0，表示Follower的日志与Leader的日志匹配，否则Follower的日志要么是不一致的，要么是落后于Leader.
        match_index: Index,
        /// 心跳的 读 序列号.
        read_seq: ReadSequence,
    },

    /// Leaders replicate log entries to followers by appending to their logs
    /// after the given base entry.
```

```
    ///
    /// If the base entry matches the follower's log then their logs are
    /// identical up to it (see section 5.3 in the Raft paper), and the entries
    /// can be appended -- possibly replacing conflicting entries. Otherwise,
    /// the append is rejected and the leader must retry an earlier base index
    /// until a common base is found.
    ///
    /// Empty appends messages (no entries) are used to probe follower logs for
    /// a common match index in the case of divergent logs, restarted nodes, or
    /// dropped messages. This is typically done by sending probes with a
    /// decrementing base index until a match is found, at which point the
    /// subsequent entries can be sent.
    /// Leaders 复制日志到 Followers, 通过在给定的 base 之后 entry 追加到他们的 log
中.
    Append {
        /// 即将添加的 entry 之前的 entry 的 index
        base_index: Index,
        /// 即将添加的 entry 之前的 entry 的 term
        base_term: Term,
        /// 即将被添加的 entry 集合, index 从 base_index + 1 开始
        entries: Vec<Entry>,
    },

    /// Followers accept or reject appends from the leader depending on whether
    /// the base entry matches their log.
    AppendResponse {
        /// If non-zero, the follower appended entries up to this index. The
        /// entire log up to this index is consistent with the leader. If no
        /// entries were sent (a probe), this will be the matching base index.
        match_index: Index,
        /// If non-zero, the follower rejected an append at this base index
        /// because the base index/term did not match its log. If the follower's
        /// log is shorter than the base index, the reject index will be lowered
        /// to the index after its last local index, to avoid probing each
        /// missing index.
        reject_index: Index,
    },

    /// Leaders need to confirm they are still the leader before serving reads,
    /// to guarantee linearizability in case a different leader has been
    /// estalished elsewhere. Read requests are served once the sequence number
    /// has been confirmed by a quorum.
    Read { seq: ReadSequence },

    /// Followers confirm leadership at the read sequence numbers.
    ReadResponse { seq: ReadSequence },
```

```
    /// A client request. This can be submitted to the leader, or to a follower
    /// which will forward it to its leader. If there is no leader, or the
    /// leader or term changes, the request is aborted with an Error::Abort
    /// ClientResponse and the client must retry.
    ClientRequest {
        /// The request ID. Must be globally unique for the request duration.
        id: RequestID,
        /// The request itself.
        request: Request,
    },

    /// A client response.
    ClientResponse {
        /// The ID of the original ClientRequest.
        id: RequestID,
        /// The response, or an error.
        response: Result<Response>,
    },
}
```

代码 3-2　Message 类型

3.3 Raft 之 Node

3.4 Raft 之 Log

3.5 其他部分

3.6 总结

第四章 SQL 引擎

```
src/sql
├── engine
│   ├── engine.rs # 定义了SQL引擎的接口.
│   ├── local.rs # 本地存储的SQL引擎.
│   ├── mod.rs
│   ├── raft.rs # 基于Raft的分布式SQL引擎.
│   └── session.rs # 执行SQL语句, 并处理事务控制.
├── execution
│   ├── aggregate.rs # SQL的聚合操作, 如GROUP BY, COUNT等.
│   ├── execute.rs # 执行计划的执行器.
│   ├── join.rs # SQL的连接操作, 如JOIN, LEFT JOIN等.
│   ├── mod.rs
│   ├── source.rs # 负责提供数据源, 如表扫描, 主键扫描, 索引扫描等.
│   ├── transform.rs # SQL的转换操作, 如投影, 过滤, 限制, 排序等.
│   └── write.rs # SQL的写操作, 如INSERT, DELETE, UPDATE等.
├── mod.rs
├── parser
│   ├── ast.rs # 定义了SQL的抽象语法树(ast)的结构.
│   ├── lexer.rs # SQL的词法分析器, 将SQL语句转换为Token.
│   ├── mod.rs
│   └── parser.rs # SQL的语法分析器, 将Token转换为AST.
├── planner
│   ├── mod.rs
│   ├── optimizer.rs # 执行计划的优化器.
│   ├── plan.rs # 执行计划的结构与操作.
│   └── planner.rs # SQL解析以及执行计划的生成.
├── testscripts
│   └── ...
└── types
    ├── expression.rs # 定义了SQL的表达式.
    ├── mod.rs
    ├── schema.rs # 定义了SQL的表结构以及列结构.
    └── value.rs # 定义了SQL的基本数据类型以及数据类型枚举.
```

代码 4-1　SQL 引擎的代码结构

在看所有代码之前，通过它们对外的接口来了解整个 SQL 引擎的结构。除了第一个基本数据类型以外，我们通过一条 SQL 的生命周期来排序这些接口。

```
// src/sql
// ├── ...
// └── types
//     ├── expression.rs # 定义了SQL的表达式.
//     ├── mod.rs
```

```
//     ├── schema.rs # 定义了SQL的表结构以及列结构.
//     └── value.rs # 定义了SQL的基本数据类型以及数据类型枚举.
pub use expression::Expression;
pub use schema::{Column, Table};
pub use value::{DataType, Label, Row, Rows, Value};
```

代码 4-2　types 对外暴露的接口

这里暴露的东西还是比较简单的，就是 SQL 的基本数据类型，表结构，列结构，表达式以及多表查询的 Lable 等。

```
// src/sql
// ├── ...
// └── engine
//     ├── engine.rs # 定义了SQL引擎的接口.
//     ├── local.rs # 本地存储的SQL引擎.
//     ├── mod.rs
//     ├── raft.rs # 基于Raft的分布式SQL引擎.
//     └── session.rs # 执行SQL语句，并处理事务控制.
pub use engine::{Catalog, Engine, Transaction};
pub use local::{Key, Local};
pub use raft::{Raft, Status, Write};
pub use session::{Session, StatementResult};
```

代码 4-3　engine 对外暴露的接口

`local.rs`， `raft.rs` 定义的两个引擎本别处理本地以及分布式事务

在 `engine` 模块中， `Session` 通过 `Engine` 接口与具体的引擎交互， `Session` 里面有个方法 `execute`，用于执行 SQL 语句。`Session` 里面的 `StatementResult` 用于表示 SQL 语句的执行结果。

```
// src/sql
// ├── ...
// └── parser
//     ├── ast.rs # 定义了SQL的抽象语法树(ast)的结构.
//     ├── lexer.rs # SQL的词法分析器，将SQL语句转换为Token.
//     ├── mod.rs
//     └── parser.rs # SQL的语法分析器，将Token转换为AST.
pub use lexer::{is_ident, Keyword, Lexer, Token};
pub use parser::Parser;
```

代码 4-4　parse 对外暴露的接口

在 `execute` 中，会调用 `Parser` 来解析 SQL 语句，解析的流程大概是： `Lexer` 负责将 SQL 语句转换为 Token， `Parser` 负责将 Token 转换为 AST。

AST 就是可以被下面的 `Planner` 所使用的执行计划。

```
// src/sql
// ├── ...
```

```
// └── planner
//     ├── mod.rs
//     ├── optimizer.rs # 执行计划的优化器.
//     ├── plan.rs # 执行计划的结构与操作.
//     └── planner.rs # SQL解析以及执行计划的生成.
pub use plan::{Aggregate, Direction, Node, Plan};
pub use planner::{Planner, Scope};

#[cfg(test)]
pub use optimizer::OPTIMIZERS;
```

代码 4-5　planner 对外暴露的接口

`execute` 从上一步获得了 AST，然后调用 `Plan::build()` 来生成执行计划。生成的执行计划会被 `Plan` 中的 `optimize()` 方法调用 optimize.rs 中的优化方法来优化。

```
// src/sql
// ├── ...
// └── execution
//     ├── aggregate.rs # SQL的聚合操作, 如GROUP BY, COUNT等.
//     ├── execute.rs # 执行计划的执行器.
//     ├── join.rs # SQL的连接操作, 如JOIN, LEFT JOIN等.
//     ├── mod.rs
//     ├── source.rs # 负责提供数据源, 如表扫描, 主键扫描, 索引扫描等.
//     ├── transform.rs # SQL的转换操作, 如投影, 过滤, 限制, 排序等.
//     └── write.rs # SQL的写操作, 如INSERT, DELETE, UPDATE等.
pub use execute::{execute_plan, ExecutionResult};
```

代码 4-6　execution 对外暴露的接口

最后执行计划会被 `Plan::execution` 执行，执行计划的结果会被 `ExecutionResult` 返回。

当然这里只是简单的说一下功能，具体的链路比现在的还要复杂一些。最终会在 第 4.6 节 更详细描述脉络。

4.1 Type

4.1.1 基本数据类型

现在看一下基本数据类型的定义。

为了简化，这里是支持少量的数据类型，不支持复杂数据类型等。

另外需要说一下, NULL 以及 NaN 都被认为是不等于本身的值, 所以理论上 `NULL ≠ NULL`，`NaN ≠ NaN`。但是在实际的代码中，NULL 和 NaN 都认为是可以比较且相等的。这是为了对允许对这些值进行排序和处理（比如索引查找、桶聚合等场景）。

另外浮点数中的 `-0.0 == 0.0`，`-NaN == NaN` 都认为返回 `true`。存储的时候会将 `-0.0` 规范化为 `0.0`，`-NaN` 规范化为 `NaN`。

```rust
/// 支持的数据类型
#[derive(Clone, Copy, Debug, Hash, PartialEq, Serialize, Deserialize)]
pub enum DataType {
    Boolean, // 布尔类型
    Integer, // 64位有符号整数
    Float,   // 64位浮点数
    String,  // UTF-8编码的字符串
}

/// 数据的值
#[derive(Clone, Debug, Serialize, Deserialize)]
pub enum Value {
    Null,           // 空值
    Boolean(bool),  // 布尔类型
    Integer(i64),   // 64位有符号整数
    Float(f64),     // 64位浮点数
    String(String), // UTF-8编码的字符串
}

// 这里定义了 Value 相等的比较
impl std::cmp::PartialEq for Value {
    fn eq(&self, other: &Self) -> bool {
        match (self, other) {
            // ...
            // 这里可以看到上面提到的 NaN == NaN 的情况
            // 另外
            // let a: f64 = 0.0;
            // let b: f64 = -0.0;
            // println!("{}", a == b); // true
            (Self::Float(l), Self::Float(r)) => l == r || l.is_nan() && r.is_nan(),
            // ..省略大部分简单代码..
            (l, r) => core::mem::discriminant(l) == core::mem::discriminant(r),
        }
    }
}

impl std::hash::Hash for Value {
    fn hash<H: std::hash::Hasher>(&self, state: &mut H) {
        core::mem::discriminant(self).hash(state);
        // Normalize to treat +/-0.0 and +/-NAN as equal when hashing.
        // 这里调用了 normalize_ref 方法，这个方法会将 -0.0 规范化为 0.0
        // -NaN 规范化为 NaN
        match self.normalize_ref().as_ref() {
            Self::Null => {}
            Self::Boolean(v) => v.hash(state),
            Self::Integer(v) => v.hash(state),
```

```
            Self::Float(v) => v.to_bits().hash(state),
            Self::String(v) => v.hash(state),
        }
    }
}

// 这里定义的全序比较，看一下就好
impl Ord for Value {
    fn cmp(&self, other: &Self) -> std::cmp::Ordering {
        use std::cmp::Ordering::*;
        use Value::*;
        match (self, other) {
            (Null, Null) => Equal,
            (Boolean(a), Boolean(b)) => a.cmp(b),
            (Integer(a), Integer(b)) => a.cmp(b),
            (Integer(a), Float(b)) => (*a as f64).total_cmp(b),
            (Float(a), Integer(b)) => a.total_cmp(&(*b as f64)),
            (Float(a), Float(b)) => a.total_cmp(b),
            (String(a), String(b)) => a.cmp(b),

            (Null, _) => Less,
            (_, Null) => Greater,
            (Boolean(_), _) => Less,
            (_, Boolean(_)) => Greater,
            (Float(_), _) => Less,
            (_, Float(_)) => Greater,
            (Integer(_), _) => Less,
            (_, Integer(_)) => Greater,
            // String is ordered last.
        }
    }
}

// 这里是实现偏序比较，在定义了全序比较以后，偏序比较直接调用全序比较就可以了
impl PartialOrd for Value {
    fn partial_cmp(&self, other: &Self) -> Option<std::cmp::Ordering> {
        Some(self.cmp(other))
    }
}
```

代码 4-7　基本数据类型

```
impl Value {
    /// 加法操作
    pub fn checked_add(&self, other: &Self) -> Result<Self> { ... }
    /// 除法操作
    pub fn checked_div(&self, other: &Self) -> Result<Self> { ... }
```

```rust
    /// 乘法操作
    pub fn checked_mul(&self, other: &Self) -> Result<Self> { ... }
    /// 幂运算
    pub fn checked_pow(&self, other: &Self) -> Result<Self> { ... }
    /// 取余操作
    pub fn checked_rem(&self, other: &Self) -> Result<Self> { ... }
    /// 减法操作
    pub fn checked_sub(&self, other: &Self) -> Result<Self> { ... }
    /// 返回数据类型
    /// 这里可以看到Rust的表达能力挺强的
    pub fn datatype(&self) -> Option<DataType> {
        match self {
            Self::Null => None,
            Self::Boolean(_) => Some(DataType::Boolean),
            Self::Integer(_) => Some(DataType::Integer),
            Self::Float(_) => Some(DataType::Float),
            Self::String(_) => Some(DataType::String),
        }
    }
    /// 返回 true 如果值是未定义的(NULL 或 NaN).
    pub fn is_undefined(&self) -> bool { ... }

    /// 原地规范化一个值.
    /// 目前将 -0.0 和 -NAN 规范化为 0.0 和 NAN, 这是主键和索引查找中使用的规范值.
    pub fn normalize(&mut self) {
        if let Cow::Owned(normalized) = self.normalize_ref() {
            *self = normalized;
        }
    }

    /// 将一个 borrow 的值规范化.
    /// 目前将 -0.0 和 -NAN 规范化为 0.0 和 NAN, 这是主键和索引查找中使用的规范值.
    /// 当值发生变化时, 返回 Cow::Owned, 以避免在值不变的常见情况下分配内存.
    pub fn normalize_ref(&self) -> Cow<'_, Self> {
        if let Self::Float(f) = self {
            if (f.is_nan() || *f == -0.0) && f.is_sign_negative() {
                return Cow::Owned(Self::Float(-f));
            }
        }
        Cow::Borrowed(self)
    }

    /// 如果值已经规范化, 返回 true.
    /// 这里还挺有意思, 通过 Cow::Borrowed 来判断是否规范化.
    /// 如果是 Cow::Owned(_), 说明已经规范化了, 返回false.
    /// 这里的 Cow::Borrowed(_), 说明传进去的值是已经规范化的, 返回true.
```

```
    pub fn is_normalized(&self) -> bool {
        matches!(self.normalize_ref(), Cow::Borrowed(_))
    }
}


impl std::fmt::Display for Value {
    fn fmt(&self, f: &mut std::fmt::Formatter) -> std::fmt::Result { ... }
}

/// 定义了 Value 与 Rust 基本数据类型的相互转换.
impl From<bool> for Value { fn from(v: bool) -> Self { Value::Boolean(v) } }
impl From<f64> for Value { fn from(v: f64) -> Self { Value::Float(v) } }
impl From<i64> for Value { fn from(v: i64) -> Self { Value::Integer(v) } }
impl From<String> for Value { fn from(v: String) -> Self { Value::String(v) } }
impl From<&str> for Value { fn from(v: &str) -> Self
{ Value::String(v.to_owned()) } }
impl TryFrom<Value> for bool { ... }
impl TryFrom<Value> for f64 { ... }
impl TryFrom<Value> for i64 { ... }
impl TryFrom<Value> for String { ... }

/// 定义了 Value 到 Cow<'_, Value> 的转换.
impl<'a> From<&'a Value> for Cow<'a, Value> {
    fn from(v: &'a Value) -> Self { Cow::Borrowed(v) }
}
```

代码 4-8　Value 的具体实现

这一部分看起来是比较简单的，但是写起来需要考虑的东西还是挺多的，比如 macth 中的顺序之类的，都是一些小细节。但是看起来难度并没有很大，这里就不再展开了。

下面看一下其他的类型。

```
/// A row of values.
pub type Row = Vec<Value>;

/// A row iterator.
pub type Rows = Box<dyn RowIterator>;

/// A row iterator trait, which requires the iterator to be both clonable and
/// object-safe. Cloning is needed to be able to reset an iterator back to an
/// initial state, e.g. during nested loop joins. It has a blanket
/// implementation for all matching iterators.
pub trait RowIterator: Iterator<Item = Result<Row>> + DynClone {}
impl<I: Iterator<Item = Result<Row>> + DynClone> RowIterator for I {}
dyn_clone::clone_trait_object!(RowIterator);
```

代码 4-9

```rust
/// A column label, used in query results and plans.
#[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
pub enum Label {
    /// No label.
    None,
    /// An unqualified column name.
    Unqualified(String),
    /// A fully qualified table/column name.
    Qualified(String, String),
}

impl std::fmt::Display for Label {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
        match self {
            Self::None => write!(f, ""),
            Self::Unqualified(name) => write!(f, "{name}"),
            Self::Qualified(table, column) => write!(f, "{table}.{column}"),
        }
    }
}

impl Label {
    /// Formats the label as a short column header.
    pub fn as_header(&self) -> &str {
        match self {
            Self::Qualified(_, column) | Self::Unqualified(column) =>
column.as_str(),
            Self::None => "?",
        }
    }
}

impl From<Label> for ast::Expression {
    /// Builds an ast::Expression::Column for a label. Can't be None.
    fn from(label: Label) -> Self {
        match label {
            Label::Qualified(table, column) => ast::Expression::Column(Some(table),
column),
            Label::Unqualified(column) => ast::Expression::Column(None, column),
            Label::None => panic!("can't convert None label to AST expression"), //
shouldn't happen
        }
    }
}

impl From<Option<String>> for Label {
```

```
    fn from(name: Option<String>) -> Self {
        name.map(Label::Unqualified).unwrap_or(Label::None)
    }
}
```

代码 4-10

4.1.2 Schema

在 Schema 中定义了 Table 以及 Column 的结构。

```
/// A table schema, which specifies its data structure and constraints.
///
/// Tables can't change after they are created. There is no ALTER TABLE nor
/// CREATE/DROP INDEX -- only CREATE TABLE and DROP TABLE.
#[derive(Clone, Debug, PartialEq, Deserialize, Serialize)]
pub struct Table {
    /// The table name. Can't be empty.
    pub name: String,
    /// The primary key column index. A table must have a primary key, and it
    /// can only be a single column.
    pub primary_key: usize,
    /// The table's columns. Must have at least one.
    pub columns: Vec<Column>,
}

/// A table column.
#[derive(Clone, Debug, PartialEq, Deserialize, Serialize)]
pub struct Column {
    /// Column name. Can't be empty.
    pub name: String,
    /// Column datatype.
    pub datatype: DataType,
    /// Whether the column allows null values. Not legal for primary keys.
    pub nullable: bool,
    /// The column's default value. If None, the user must specify an explicit
    /// value. Must match the column datatype. Nullable columns require a
    /// default (often Null), and Null is only a valid default when nullable.
    pub default: Option<Value>,
    /// Whether the column should only allow unique values (ignoring NULLs).
    /// Must be true for a primary key column.
    pub unique: bool,
    /// Whether the column should have a secondary index. Must be false for
    /// primary keys, which are the implicit primary index. Must be true for
    /// unique or reference columns.
    pub index: bool,
    /// If set, this column is a foreign key reference to the given table's
```

```
        /// primary key. Must be of the same type as the target primary key.
        pub references: Option<String>,
    }
```

代码 4-11　Colume 以及 Table 的结构

这里的表结构还是比较简单的，只有基本的表名，主键所在的列号，所有的列。每一列有列名，数据类型，是否允许为空，默认值，是否唯一，是否有索引等。

```
impl std::fmt::Display for Table {
    fn fmt(&self, f: &mut std::fmt::Formatter<'_>) -> std::fmt::Result {
      // ...
    }
}

impl Table {
    /// Validates the table schema, using the catalog to validate foreign key
    /// references.
    pub fn validate(&self, catalog: &impl Catalog) -> Result<()> {
        if self.name.is_empty() {
            return errinput!("table name can't be empty");
        }
        if self.columns.is_empty() {
            return errinput!("table has no columns");
        }
        if self.columns.get(self.primary_key).is_none() {
            return errinput!("invalid primary key index");
        }

        for (i, column) in self.columns.iter().enumerate() {
            if column.name.is_empty() {
                return errinput!("column name can't be empty");
            }
            let (cname, ctype) = (&column.name, &column.datatype); // for formatting
convenience

            // Validate primary key.
            let is_primary_key = i == self.primary_key;
            if is_primary_key {
                if column.nullable {
                    return errinput!("primary key {cname} cannot be nullable");
                }
                if !column.unique {
                    return errinput!("primary key {cname} must be unique");
                }
                if column.index {
                    return errinput!("primary key {cname} can't have an index");
                }
```

```
            }

            // Validate default value.
            match column.default.as_ref().map(|v| v.datatype()) {
                None if column.nullable => {
                    return errinput!("nullable column {cname} must have a
default value")
                }
                Some(None) if !column.nullable => {
                    return errinput!("invalid NULL default for non-nullable
column {cname}")
                }
                Some(Some(vtype)) if vtype != column.datatype => {
                    return errinput!("invalid default type {vtype} for {ctype}
column {cname}");
                }
                Some(_) | None => {}
            }

            // Validate unique index.
            if column.unique && !column.index && !is_primary_key {
                return errinput!("unique column {cname} must have a secondary
index");
            }

            // Validate references.
            if let Some(reference) = &column.references {
                if !column.index && !is_primary_key {
                    return errinput!("reference column {cname} must have a
secondary index");
                }
                let reftype = if reference == &self.name {
                    self.columns[self.primary_key].datatype
                } else if let Some(target) = catalog.get_table(reference)? {
                    target.columns[target.primary_key].datatype
                } else {
                    return errinput!("unknown table {reference} referenced by
column {cname}");
                };
                if column.datatype != reftype {
                    return errinput!("can't reference {reftype} primary key of
{reference} from {ctype} column {cname}");
                }
            }
        }
        Ok(())
```

```
    }

    /// Validates a row, including uniqueness and reference checks using the
    /// given transaction.
    ///
    /// If update is true, the row replaces an existing entry with the same
    /// primary key. Otherwise, it is an insert. Primary key changes are
    /// implemented as a delete+insert.
    ///
    /// Validating uniqueness and references individually for each row is not
    /// performant, but it's fine for our purposes.
    pub fn validate_row(&self, row: &[Value], update: bool, txn: &impl Transaction) -
> Result<()> {
        if row.len() != self.columns.len() {
            return errinput!("invalid row size for table {}", self.name);
        }

        // Validate primary key.
        let id = &row[self.primary_key];
        let idslice = &row[self.primary_key..=self.primary_key];
        if id.is_undefined() {
            return errinput!("invalid primary key {id}");
        }
        if !update && !txn.get(&self.name, idslice)?.is_empty() {
            return errinput!("primary key {id} already exists");
        }

        for (i, (column, value)) in self.columns.iter().zip(row).enumerate() {
            let (cname, ctype) = (&column.name, &column.datatype);
            let valueslice = &row[i..=i];

            // Validate datatype.
            if let Some(ref vtype) = value.datatype() {
                if vtype != ctype {
                    return errinput!("invalid datatype {vtype} for {ctype}
column {cname}");
                }
            }
            if value == &Value::Null && !column.nullable {
                return errinput!("NULL value not allowed for column {cname}");
            }

            // Validate outgoing references.
            if let Some(target) = &column.references {
                match value {
                    // NB: NaN is not a valid primary key, and not valid as a
```

```
                    // missing foreign key marker.
                    Value::Null => {}
                    v if target == &self.name && v == id => {}
                    v if txn.get(target, valueslice)?.is_empty() => {
                        return errinput!("reference {v} not in table {target}");
                    }
                    _ => {}
                }
            }

            // Validate uniqueness constraints. Unique columns are indexed.
            if column.unique && i != self.primary_key && !value.is_undefined() {
                let mut index = txn.lookup_index(&self.name, &column.name,
valueslice)?;
                if update {
                    index.remove(id); // ignore existing version of this row
                }
                if !index.is_empty() {
                    return errinput!("value {value} already in unique column
{cname}");
                }
            }
        }
        Ok(())
    }
}
```

代码 4-12

4.1.3 表达式

```
    /// An expression, made up of nested operations and values. Values are either
    /// constants or dynamic column references. Evaluates to a final value during
    /// query execution, using row values for column references.
    ///
    /// Since this is a recursive data structure, we have to box each child
    /// expression, which incurs a heap allocation per expression node. There are
    /// clever ways to avoid this, but we keep it simple.
    #[derive(Clone, Debug, PartialEq, Serialize, Deserialize)]
    pub enum Expression {
        /// A constant value.
        Constant(Value),
        /// A column reference. Used as row index when evaluating expressions.
        Column(usize),

        /// Logical AND of two booleans: a AND b.
        And(Box<Expression>, Box<Expression>),
```

```
        /// Logical OR of two booleans: a OR b.
        Or(Box<Expression>, Box<Expression>),
        /// Logical NOT of a boolean: NOT a.
        Not(Box<Expression>),

        /// Equality comparison of two values: a = b.
        Equal(Box<Expression>, Box<Expression>),
        /// Greater than comparison of two values: a > b.
        GreaterThan(Box<Expression>, Box<Expression>),
        /// Less than comparison of two values: a < b.
        LessThan(Box<Expression>, Box<Expression>),
        /// Checks for the given value: IS NULL or IS NAN.
        Is(Box<Expression>, Value),

        /// Adds two numbers: a + b.
        Add(Box<Expression>, Box<Expression>),
        /// Divides two numbers: a / b.
        Divide(Box<Expression>, Box<Expression>),
        /// Exponentiates two numbers, i.e. a ^ b.
        Exponentiate(Box<Expression>, Box<Expression>),
        /// Takes the factorial of a number: 4! = 4*3*2*1.
        Factorial(Box<Expression>),
        /// The identify function, which simply returns the same number: +a.
        Identity(Box<Expression>),
        /// Multiplies two numbers: a * b.
        Multiply(Box<Expression>, Box<Expression>),
        /// Negates the given number: -a.
        Negate(Box<Expression>),
        /// The remainder after dividing two numbers: a % b.
        Remainder(Box<Expression>, Box<Expression>),
        /// Takes the square root of a number: √a.
        SquareRoot(Box<Expression>),
        /// Subtracts two numbers: a - b.
        Subtract(Box<Expression>, Box<Expression>),

        // Checks if a string matches a pattern: a LIKE b.
        Like(Box<Expression>, Box<Expression>),
    }
```

代码 4-13　Expression 的定义

```
impl Expression {
    /// Formats the expression, using the given plan node to look up labels for
    /// numeric column references.
    pub fn format(&self, node: &Node) -> String {
        use Expression::*;
```

```
        // Precedence levels, for grouping. Matches the parser precedence.
        fn precedence(expr: &Expression) → u8 {
            match expr {
                Column(_) | Constant(_) | SquareRoot(_) ⇒ 11,
                Identity(_) | Negate(_) ⇒ 10,
                Factorial(_) ⇒ 9,
                Exponentiate(_, _) ⇒ 8,
                Multiply(_, _) | Divide(_, _) | Remainder(_, _) ⇒ 7,
                Add(_, _) | Subtract(_, _) ⇒ 6,
                GreaterThan(_, _) | LessThan(_, _) ⇒ 5,
                Equal(_, _) | Like(_, _) | Is(_, _) ⇒ 4,
                Not(_) ⇒ 3,
                And(_, _) ⇒ 2,
                Or(_, _) ⇒ 1,
            }
        }

        // Helper to format a boxed expression, grouping it with () if needed.
        let format = |expr: &Expression| {
            let mut string = expr.format(node);
            if precedence(expr) < precedence(self) {
                string = format!("({string})");
            }
            string
        };

        match self {
            Constant(value) ⇒ format!("{value}"),
            Column(index) ⇒ match node.column_label(*index) {
                Label::None ⇒ format!("#{index}"),
                label ⇒ format!("{label}"),
            },

            And(lhs, rhs) ⇒ format!("{} AND {}", format(lhs), format(rhs)),
            Or(lhs, rhs) ⇒ format!("{} OR {}", format(lhs), format(rhs)),
            Not(expr) ⇒ format!("NOT {}", format(expr)),

            Equal(lhs, rhs) ⇒ format!("{} = {}", format(lhs), format(rhs)),
            GreaterThan(lhs, rhs) ⇒ format!("{} > {}", format(lhs), format(rhs)),
            LessThan(lhs, rhs) ⇒ format!("{} < {}", format(lhs), format(rhs)),
            Is(expr, Value::Null) ⇒ format!("{} IS NULL", format(expr)),
            Is(expr, Value::Float(f)) if f.is_nan() ⇒ format!("{} IS NAN",
format(expr)),
            Is(_, v) ⇒ panic!("unexpected IS value {v}"),

            Add(lhs, rhs) ⇒ format!("{} + {}", format(lhs), format(rhs)),
```

```
            Divide(lhs, rhs) => format!("{} / {}", format(lhs), format(rhs)),
            Exponentiate(lhs, rhs) => format!("{} ^ {}", format(lhs), format(rhs)),
            Factorial(expr) => format!("{}!", format(expr)),
            Identity(expr) => format(expr),
            Multiply(lhs, rhs) => format!("{} * {}", format(lhs), format(rhs)),
            Negate(expr) => format!("-{}", format(expr)),
            Remainder(lhs, rhs) => format!("{} % {}", format(lhs), format(rhs)),
            SquareRoot(expr) => format!("sqrt({})", format(expr)),
            Subtract(lhs, rhs) => format!("{} - {}", format(lhs), format(rhs)),

            Like(lhs, rhs) => format!("{} LIKE {}", format(lhs), format(rhs)),
        }
    }

    /// Formats a constant expression. Errors on column references.
    pub fn format_constant(&self) -> String {
        self.format(&Node::Nothing { columns: Vec::new() })
    }

    /// Evaluates an expression, returning a value. Column references look up
    /// values in the given row. If None, any Column references will panic.
    pub fn evaluate(&self, row: Option<&Row>) -> Result<Value> {
        use Value::*;
        Ok(match self {
            // Constant values return themselves.
            Self::Constant(value) => value.clone(),

            // Column references look up a row value. The planner ensures that
            // only constant expressions are evaluated without a row.
            Self::Column(index) => match row {
                Some(row) => row.get(*index).expect("short row").clone(),
                None => panic!("can't reference column {index} with constant
evaluation"),
            },

            // Logical AND. Inputs must be boolean or NULL. NULLs generally
            // yield NULL, except the special case NULL AND false == false.
            Self::And(lhs, rhs) => match (lhs.evaluate(row)?, rhs.evaluate(row)?) {
                (Boolean(lhs), Boolean(rhs)) => Boolean(lhs && rhs),
                (Boolean(b), Null) | (Null, Boolean(b)) if !b => Boolean(false),
                (Boolean(_), Null) | (Null, Boolean(_)) | (Null, Null) => Null,
                (lhs, rhs) => return errinput!("can't AND {lhs} and {rhs}"),
            },

            // Logical OR. Inputs must be boolean or NULL. NULLs generally
            // yield NULL, except the special case NULL OR true == true.
```

```
            Self::Or(lhs, rhs) => match (lhs.evaluate(row)?, rhs.evaluate(row)?) {
                (Boolean(lhs), Boolean(rhs)) => Boolean(lhs || rhs),
                (Boolean(b), Null) | (Null, Boolean(b)) if b => Boolean(true),
                (Boolean(_), Null) | (Null, Boolean(_)) | (Null, Null) => Null,
                (lhs, rhs) => return errinput!("can't OR {lhs} and {rhs}"),
            },

            // Logical NOT. Input must be boolean or NULL.
            Self::Not(expr) => match expr.evaluate(row)? {
                Boolean(b) => Boolean(!b),
                Null => Null,
                value => return errinput!("can't NOT {value}"),
            },

            // Comparisons. Must be of same type, except floats and integers
            // which are interchangeable. NULLs yield NULL, NaNs yield NaN.
            //
            // Does not dispatch to Value.cmp() because sorting and comparisons
            // are different for f64 NaN and -0.0 values.
            #[allow(clippy::float_cmp)]
            Self::Equal(lhs, rhs) => match (lhs.evaluate(row)?, rhs.evaluate(row)?) {
                (Boolean(lhs), Boolean(rhs)) => Boolean(lhs == rhs),
                (Integer(lhs), Integer(rhs)) => Boolean(lhs == rhs),
                (Integer(lhs), Float(rhs)) => Boolean(lhs as f64 == rhs),
                (Float(lhs), Integer(rhs)) => Boolean(lhs == rhs as f64),
                (Float(lhs), Float(rhs)) => Boolean(lhs == rhs),
                (String(lhs), String(rhs)) => Boolean(lhs == rhs),
                (Null, _) | (_, Null) => Null,
                (lhs, rhs) => return errinput!("can't compare {lhs} and {rhs}"),
            },

            Self::GreaterThan(lhs, rhs) => match (lhs.evaluate(row)?,
rhs.evaluate(row)?) {
                #[allow(clippy::bool_comparison)]
                (Boolean(lhs), Boolean(rhs)) => Boolean(lhs > rhs),
                (Integer(lhs), Integer(rhs)) => Boolean(lhs > rhs),
                (Integer(lhs), Float(rhs)) => Boolean(lhs as f64 > rhs),
                (Float(lhs), Integer(rhs)) => Boolean(lhs > rhs as f64),
                (Float(lhs), Float(rhs)) => Boolean(lhs > rhs),
                (String(lhs), String(rhs)) => Boolean(lhs > rhs),
                (Null, _) | (_, Null) => Null,
                (lhs, rhs) => return errinput!("can't compare {lhs} and {rhs}"),
            },

            Self::LessThan(lhs, rhs) => match (lhs.evaluate(row)?,
rhs.evaluate(row)?) {
```

```
                #[allow(clippy::bool_comparison)]
                (Boolean(lhs), Boolean(rhs)) => Boolean(lhs < rhs),
                (Integer(lhs), Integer(rhs)) => Boolean(lhs < rhs),
                (Integer(lhs), Float(rhs)) => Boolean((lhs as f64) < rhs),
                (Float(lhs), Integer(rhs)) => Boolean(lhs < rhs as f64),
                (Float(lhs), Float(rhs)) => Boolean(lhs < rhs),
                (String(lhs), String(rhs)) => Boolean(lhs < rhs),
                (Null, _) | (_, Null) => Null,
                (lhs, rhs) => return errinput!("can't compare {lhs} and {rhs}"),
            },

            Self::Is(expr, Null) => Boolean(expr.evaluate(row)? == Null),
            Self::Is(expr, Float(f)) if f.is_nan() => match expr.evaluate(row)? {
                Float(f) => Boolean(f.is_nan()),
                Null => Null,
                v => return errinput!("IS NAN can't be used with {}",
v.datatype().unwrap()),
            },
            Self::Is(_, v) => panic!("invalid IS value {v}"), // enforced by parser

            // Mathematical operations. Inputs must be numbers, but integers and
            // floats are interchangeable (float when mixed). NULLs yield NULL.
            // Errors on integer overflow, while floats yield infinity or NaN.
            Self::Add(lhs, rhs) =>
lhs.evaluate(row)?.checked_add(&rhs.evaluate(row)?)?,
            Self::Divide(lhs, rhs) =>
lhs.evaluate(row)?.checked_div(&rhs.evaluate(row)?)?,
            Self::Exponentiate(lhs, rhs) =>
lhs.evaluate(row)?.checked_pow(&rhs.evaluate(row)?)?,
            Self::Factorial(expr) => match expr.evaluate(row)? {
                Integer(i) if i < 0 => return errinput!("can't take factorial of
negative number"),
                Integer(i) => (1..=i).try_fold(Integer(1), |p, i|
p.checked_mul(&Integer(i)))?,
                Null => Null,
                value => return errinput!("can't take factorial of {value}"),
            },
            Self::Identity(expr) => match expr.evaluate(row)? {
                v @ (Integer(_) | Float(_) | Null) => v,
                expr => return errinput!("can't take the identity of {expr}"),
            },
            Self::Multiply(lhs, rhs) =>
lhs.evaluate(row)?.checked_mul(&rhs.evaluate(row)?)?,
            Self::Negate(expr) => match expr.evaluate(row)? {
                Integer(i) => Integer(-i),
                Float(f) => Float(-f),
```

```
                Null => Null,
                value => return errinput!("can't negate {value}"),
            },
            Self::Remainder(lhs, rhs) =>
lhs.evaluate(row)?.checked_rem(&rhs.evaluate(row)?)?,
            Self::SquareRoot(expr) => match expr.evaluate(row)? {
                Integer(i) if i < 0 => return errinput!("can't take negative
square root"),
                Integer(i) => Float((i as f64).sqrt()),
                Float(f) => Float(f.sqrt()),
                Null => Null,
                value => return errinput!("can't take square root of {value}"),
            },
            Self::Subtract(lhs, rhs) =>
lhs.evaluate(row)?.checked_sub(&rhs.evaluate(row)?)?,

            // LIKE pattern matching, using _ and % as single- and
            // multi-character wildcards. Inputs must be strings. NULLs yield
            // NULL. There's no support for escaping an _ and %.
            Self::Like(lhs, rhs) => match (lhs.evaluate(row)?, rhs.evaluate(row)?) {
                (String(lhs), String(rhs)) => {
                    // We could precompile the pattern if it's constant, instead
                    // of recompiling it for every row, but this is fine.
                    let pattern =
                        format!("^{}$", regex::escape(&rhs).replace('%',
".*").replace('_', "."));
                    Boolean(regex::Regex::new(&pattern)?.is_match(&lhs))
                }
                (String(_), Null) | (Null, String(_)) | (Null, Null) => Null,
                (lhs, rhs) => return errinput!("can't LIKE {lhs} and {rhs}"),
            },
        })
    }

    /// Recursively walks the expression tree depth-first, calling the given
    /// closure until it returns false. Returns true otherwise.
    pub fn walk(&self, visitor: &mut impl FnMut(&Expression) -> bool) -> bool {
        if !visitor(self) {
            return false;
        }
        match self {
            Self::Add(lhs, rhs)
            | Self::And(lhs, rhs)
            | Self::Divide(lhs, rhs)
            | Self::Equal(lhs, rhs)
            | Self::Exponentiate(lhs, rhs)
```

```
            | Self::GreaterThan(lhs, rhs)
            | Self::LessThan(lhs, rhs)
            | Self::Like(lhs, rhs)
            | Self::Multiply(lhs, rhs)
            | Self::Or(lhs, rhs)
            | Self::Remainder(lhs, rhs)
            | Self::Subtract(lhs, rhs) => lhs.walk(visitor) && rhs.walk(visitor),

            Self::Factorial(expr)
            | Self::Identity(expr)
            | Self::Is(expr, _)
            | Self::Negate(expr)
            | Self::Not(expr)
            | Self::SquareRoot(expr) => expr.walk(visitor),

            Self::Constant(_) | Self::Column(_) => true,
        }
    }

    /// Recursively walks the expression tree depth-first, calling the given
    /// closure until it returns true. Returns false otherwise. This is the
    /// inverse of walk().
    pub fn contains(&self, visitor: &impl Fn(&Expression) -> bool) -> bool {
        !self.walk(&mut |e| !visitor(e))
    }

    /// Transforms the expression by recursively applying the given closures
    /// depth-first to each node before/after descending.
    pub fn transform(
        mut self,
        before: &impl Fn(Self) -> Result<Self>,
        after: &impl Fn(Self) -> Result<Self>,
    ) -> Result<Self> {
        // Helper for transforming boxed expressions.
        let xform = |mut expr: Box<Expression>| -> Result<Box<Expression>> {
            *expr = expr.transform(before, after)?;
            Ok(expr)
        };

        self = before(self)?;
        self = match self {
            Self::Add(lhs, rhs) => Self::Add(xform(lhs)?, xform(rhs)?),
            Self::And(lhs, rhs) => Self::And(xform(lhs)?, xform(rhs)?),
            Self::Divide(lhs, rhs) => Self::Divide(xform(lhs)?, xform(rhs)?),
            Self::Equal(lhs, rhs) => Self::Equal(xform(lhs)?, xform(rhs)?),
            Self::Exponentiate(lhs, rhs) => Self::Exponentiate(xform(lhs)?,
```

```
xform(rhs)?),
            Self::GreaterThan(lhs, rhs) => Self::GreaterThan(xform(lhs)?,
xform(rhs)?),
            Self::LessThan(lhs, rhs) => Self::LessThan(xform(lhs)?, xform(rhs)?),
            Self::Like(lhs, rhs) => Self::Like(xform(lhs)?, xform(rhs)?),
            Self::Multiply(lhs, rhs) => Self::Multiply(xform(lhs)?, xform(rhs)?),
            Self::Or(lhs, rhs) => Self::Or(xform(lhs)?, xform(rhs)?),
            Self::Remainder(lhs, rhs) => Self::Remainder(xform(lhs)?, xform(rhs)?),
            Self::SquareRoot(expr) => Self::SquareRoot(xform(expr)?),
            Self::Subtract(lhs, rhs) => Self::Subtract(xform(lhs)?, xform(rhs)?),

            Self::Factorial(expr) => Self::Factorial(xform(expr)?),
            Self::Identity(expr) => Self::Identity(xform(expr)?),
            Self::Is(expr, value) => Self::Is(xform(expr)?, value),
            Self::Negate(expr) => Self::Negate(xform(expr)?),
            Self::Not(expr) => Self::Not(xform(expr)?),

            expr @ (Self::Constant(_) | Self::Column(_)) => expr,
        };
        self = after(self)?;
        Ok(self)
    }

    /// Converts the expression into conjunctive normal form, i.e. an AND of
    /// ORs, which is useful when optimizing plans. This is done by converting
    /// to negation normal form and then applying De Morgan's distributive law.
    pub fn into_cnf(self) -> Self {
        use Expression::*;
        let xform = |expr| {
            // We can't use a single match, since it needs deref patterns.
            let Or(lhs, rhs) = expr else { return expr };
            match (*lhs, *rhs) {
                // (x AND y) OR z → (x OR z) AND (y OR z)
                (And(l, r), rhs) => And(Or(l, rhs.clone().into()).into(), Or(r,
rhs.into()).into()),
                // x OR (y AND z) → (x OR y) AND (x OR z)
                (lhs, And(l, r)) => And(Or(lhs.clone().into(), l).into(),
Or(lhs.into(), r).into()),
                // Otherwise, do nothing.
                (lhs, rhs) => Or(lhs.into(), rhs.into()),
            }
        };
        self.into_nnf().transform(&|e| Ok(xform(e)), &Ok).unwrap() // infallible
    }

    /// Converts the expression into negation normal form. This pushes NOT
```

```rust
    /// operators into the tree using De Morgan's laws, such that they're always
    /// below other logical operators. It is a useful intermediate form for
    /// applying other logical normalizations.
    pub fn into_nnf(self) -> Self {
        use Expression::*;
        let xform = |expr| {
            let Not(inner) = expr else { return expr };
            match *inner {
                // NOT (x AND y) -> (NOT x) OR (NOT y)
                And(lhs, rhs) => Or(Not(lhs).into(), Not(rhs).into()),
                // NOT (x OR y) -> (NOT x) AND (NOT y)
                Or(lhs, rhs) => And(Not(lhs).into(), Not(rhs).into()),
                // NOT NOT x -> x
                Not(inner) => *inner,
                // Otherwise, do nothing.
                expr => Not(expr.into()),
            }
        };
        self.transform(&|e| Ok(xform(e)), &Ok).unwrap() // never fails
    }

    /// Converts the expression into conjunctive normal form as a vector of
    /// ANDed expressions (instead of nested ANDs).
    pub fn into_cnf_vec(self) -> Vec<Self> {
        let mut cnf = Vec::new();
        let mut stack = vec![self.into_cnf()];
        while let Some(expr) = stack.pop() {
            if let Self::And(lhs, rhs) = expr {
                stack.extend([*rhs, *lhs]); // push lhs last to pop it first
            } else {
                cnf.push(expr);
            }
        }
        cnf
    }

    /// Creates an expression by ANDing together a vector, or None if empty.
    pub fn and_vec(exprs: Vec<Expression>) -> Option<Self> {
        let mut iter = exprs.into_iter();
        let mut expr = iter.next()?;
        for rhs in iter {
            expr = Expression::And(expr.into(), rhs.into());
        }
        Some(expr)
    }
```

```
    /// Checks if an expression is a single column lookup (i.e. a disjunction of
    /// = or IS NULL/NAN for a single column), returning the column index.
    pub fn is_column_lookup(&self) -> Option<usize> {
        use Expression::*;
        match &self {
            // Column/constant equality can use index lookups. NULL and NaN are
            // handled in into_column_values().
            Equal(lhs, rhs) => match (lhs.as_ref(), rhs.as_ref()) {
                (Column(c), Constant(_)) | (Constant(_), Column(c)) => Some(*c),
                _ => None,
            },
            // IS NULL and IS NAN can use index lookups.
            Is(expr, _) => match expr.as_ref() {
                Column(c) => Some(*c),
                _ => None,
            },
            // All OR branches must be lookups on the same column:
            // id = 1 OR id = 2 OR id = 3.
            Or(lhs, rhs) => match (lhs.is_column_lookup(), rhs.is_column_lookup()) {
                (Some(l), Some(r)) if l == r => Some(l),
                _ => None,
            },
            _ => None,
        }
    }

    /// Extracts column lookup values for the given column. Panics if the
    /// expression isn't a lookup of the given column, i.e. is_column_lookup()
    /// must return true for the expression.
    pub fn into_column_values(self, index: usize) -> Vec<Value> {
        use Expression::*;
        match self {
            Equal(lhs, rhs) => match (*lhs, *rhs) {
                (Column(column), Constant(value)) | (Constant(value), Column(column))
=> {
                    assert_eq!(column, index, "unexpected column");
                    // NULL and NAN index lookups are for IS NULL and IS NAN.
                    // Equality shouldn't match anything, return empty vec.
                    if value.is_undefined() {
                        Vec::new()
                    } else {
                        vec![value]
                    }
                }
                (lhs, rhs) => panic!("unexpected expression {:?}", Equal(lhs.into(),
rhs.into())),
```

```
            },
            // IS NULL and IS NAN can use index lookups.
            Is(expr, value) => match *expr {
                Column(column) => {
                    assert_eq!(column, index, "unexpected column");
                    vec![value]
                }
                expr => panic!("unexpected expression {expr:?}"),
            },
            Or(lhs, rhs) => {
                let mut values = lhs.into_column_values(index);
                values.extend(rhs.into_column_values(index));
                values
            }
            expr => panic!("unexpected expression {expr:?}"),
        }
    }

    /// Replaces column references with the given column.
    pub fn replace_column(self, from: usize, to: usize) -> Self {
        let xform = |expr| match expr {
            Expression::Column(i) if i == from => Expression::Column(to),
            expr => expr,
        };
        self.transform(&|e| Ok(xform(e)), &Ok).unwrap() // infallible
    }

    /// Shifts column references by the given amount.
    pub fn shift_column(self, diff: isize) -> Self {
        let xform = |expr| match expr {
            Expression::Column(i) => Expression::Column((i as isize + diff) as
usize),
            expr => expr,
        };
        self.transform(&|e| Ok(xform(e)), &Ok).unwrap() // infallible
    }
}
```

代码 4-14　Expression 的实现

4.1.4 总结

Type这一部分是比较简单的，没有太多的复杂的逻辑。主要是定义了一些基本的数据结构，比如表结构，列结构，表达式等。表达式是一个递归的数据结构，可以表示复杂的逻辑表达式。这一部分的代码主要是定义了这些数据结构，以及对这些数据结构的一些操作。

4.2 Engine

4.2.1 SQL 引擎的 Engine 接口

4.2.2 本地存储的 SQL 引擎

4.2.3 基于 Raft 的分布式 SQL 引擎

4.2.4 Session

4.2.5 总结

4.3 Parse

4.3.1 抽象语法树

4.3.2 词法解析

4.3.3 语法解析

4.3.4 总结

4.4 Planner

4.4.1 Plan 结构

4.4.2 执行计划的生成

4.4.3 执行计划的优化

4.4.4 总结

4.5 Execution

4.5.1 执行器

4.5.2 扫描操作

4.5.3 聚合操作

4.5.4 连接操作

4.5.5 转换操作

4.5.6 写操作

4.5.7 总结

4.6 总结

第五章 数据编码

5.1 Keycode

5.2 Bincode

5.3 Format

5.4 总结

第六章 附录

6.1 BitCask 的论文解读

6.1.1 参考

1. https://riak.com/assets/bitcask-intro.pdf
2. https://arpitbhayani.me/blogs/bitcask/

6.2 隔离级别

6.2.1 写倾斜(Write Skew)问题

6.2.2 参考

1. https://justinjaffray.com/what-does-write-skew-look-like/