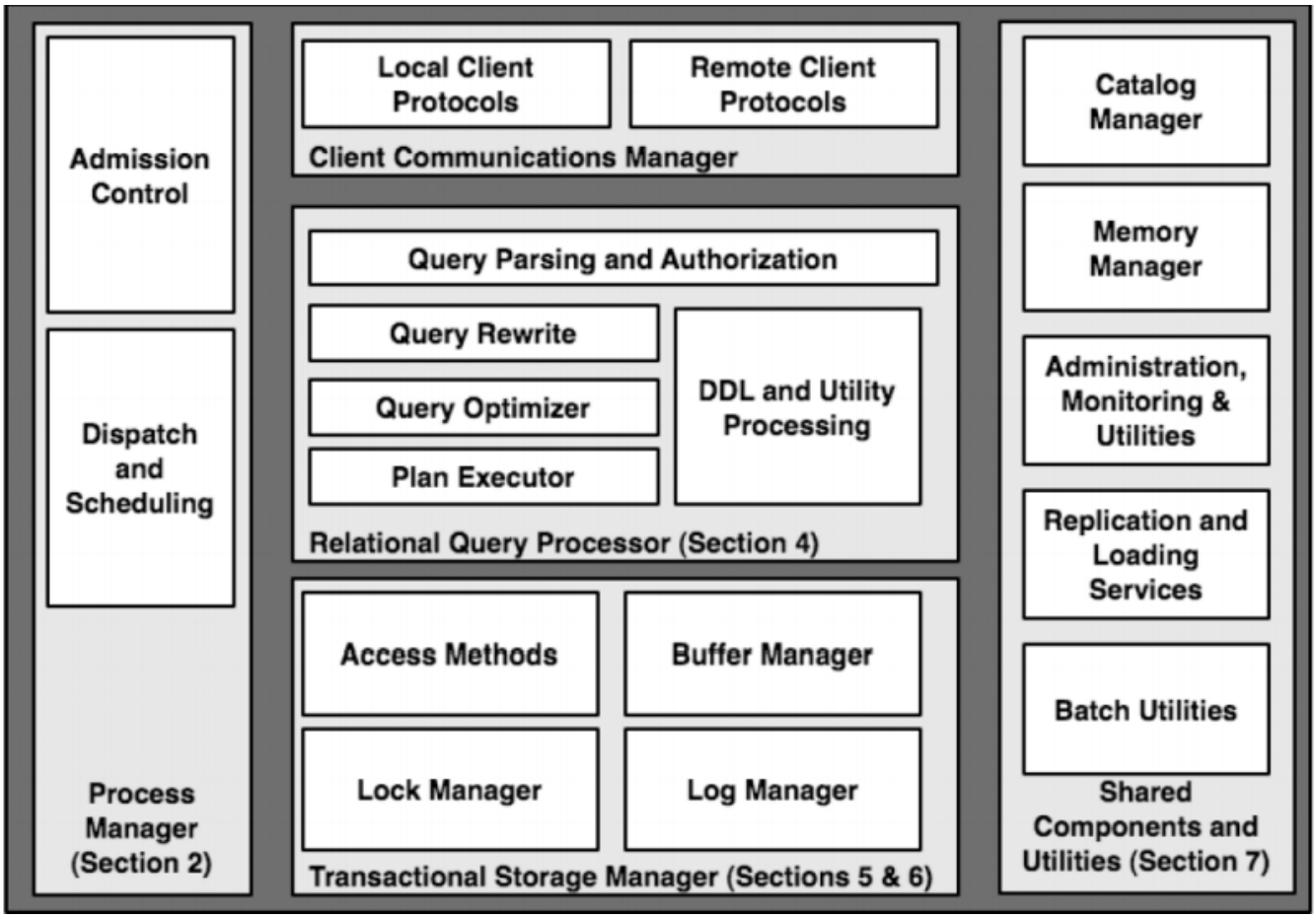
数据库系统架构论文阅读笔记

我看的是中文版，英文原版一百多页有毒啊，不过一些翻译不好的地方以及专有名词还是建议看原版

我是边看论文导读边做笔记的，感谢PingCAP哈哈

导论



关系型数据库五个部分

- Client Communication Manager

负责管理调用者与数据库服务器之间的连接

- Process Manager

在收到客户端第一个请求之后，DBMS会为之分配一个计算线程。主要工作是确保线程数据以及控制输出是通过通信管理器与客户端连接的。

- Relational Query Processor

1.检查用户是否有权进行该查询

2.把用户sql查询语句编译为中间查询计划

3.编译完成后结果查询计划会交给查询执行器

4.查询执行器会包含一系列处理查询的操作，典型处理查询任务包括:连接，选择，投影，聚集，排序等

- transactional storage manager

存储管理器：负责所有的数据接口和操作调用

存储系统主要是包括了管理**磁盘的基本算法和数据结构**，比如基本的表和索引

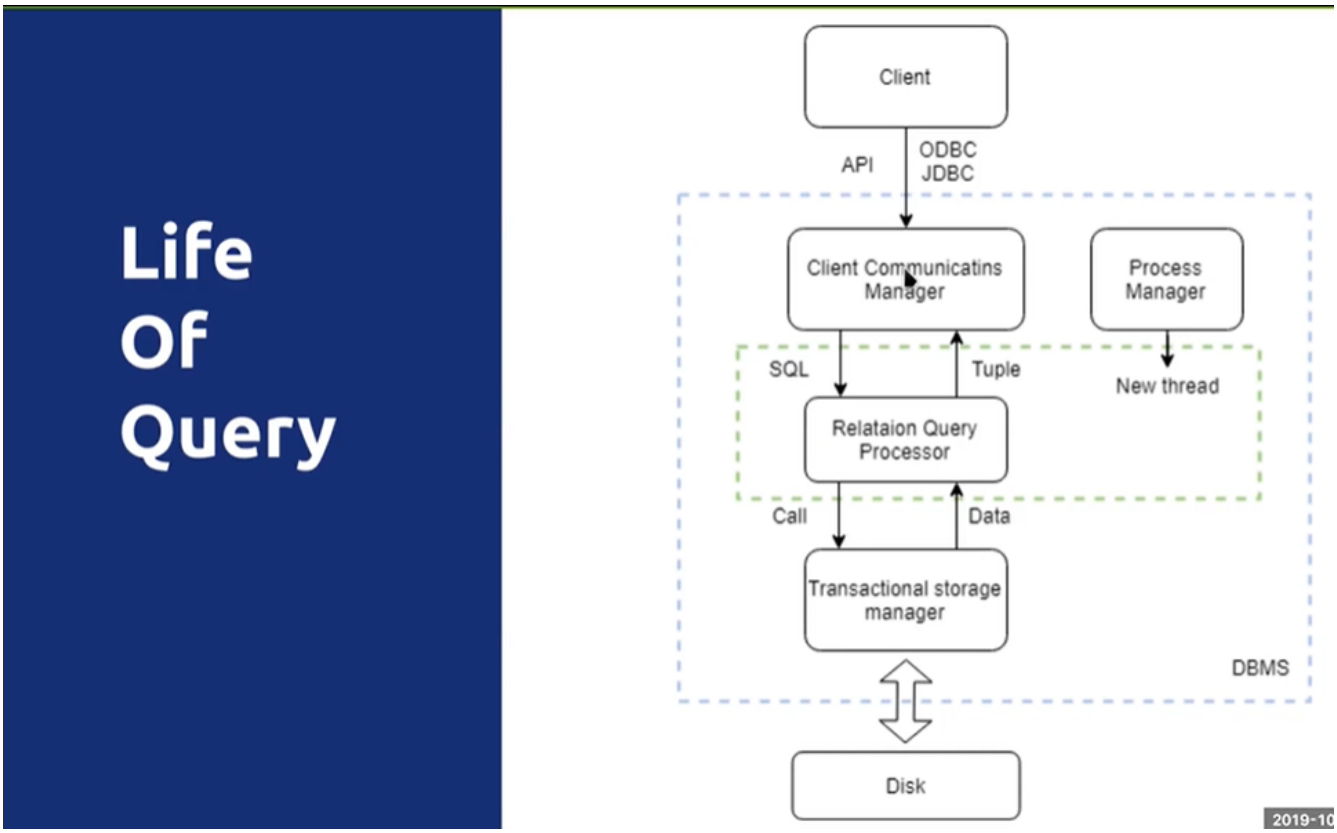
它还会包括一个buffer manager，用来控制内存缓冲区和磁盘的数据传输

还有就是lock manager,保证ACID

log manager用于实现undolog(撤销操作的完整性,实现原子性关键)和redolog(事务操作的持久性实现)

- 查询结束后，把数据库的数据组织成结果元组，放入client communication manager的buffer中，由其负责发送给调用者。
- 而上图的右侧的组件会独立于任何查询，使得数据库保持稳定性和整体性。例如CatalogManager，在数据的传输，分解与查询优化过程中会用到目录。MemoryManager也广泛应用于整个DBMS运行中动态分配和释放内存的场合。

我们以这幅图来简单总结下Query执行的生命周期



接下来对比下TiDB的SQL层的执行流程

1.Client发送调用，具体而言是一个网络包

2.协议层解析这个网络包，拆包，把SQL解析出来

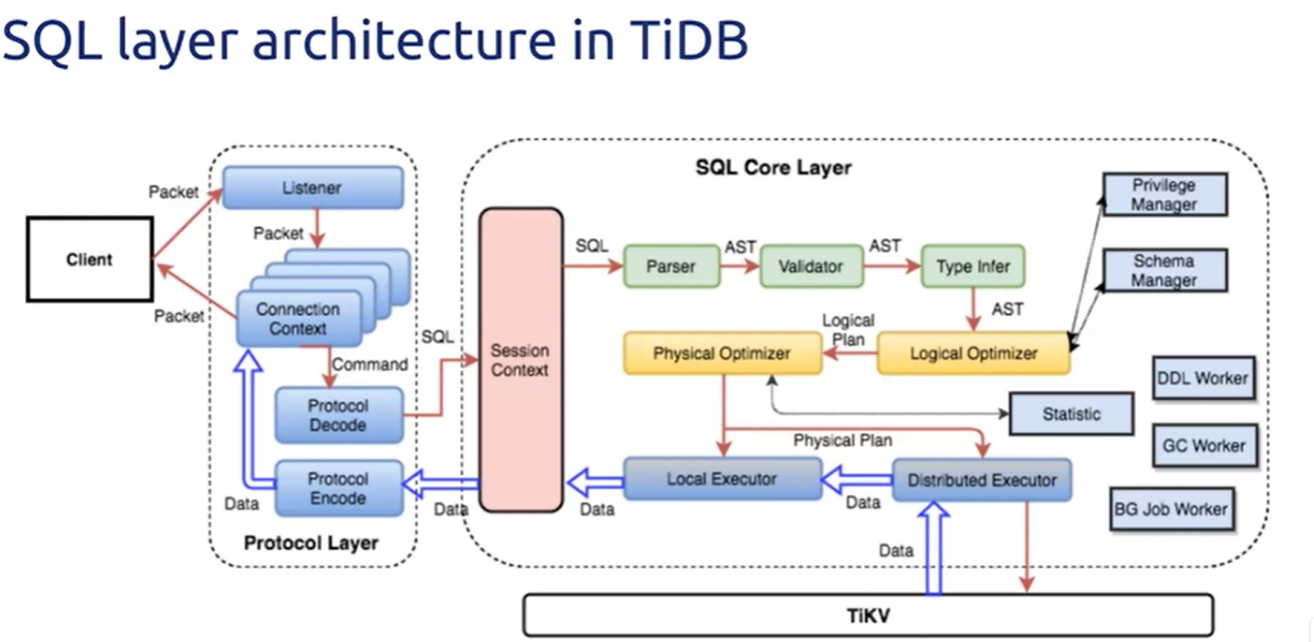
3.SQL经过词法语法语义分析转换成AST抽象语法树

4.语法树传入Logical Optimizer生成逻辑计划

5.逻辑计划经过Physical Optimizer生成物理计划

6.物理计划分发给本地或者分布式的执行器

7.执行器负责远程从TIKV调数据



进程模型

- 每个DBMS worker拥有一个进程
- 每个DBMS worker拥有一个线程
- 进程池

共享数据和进程空间

两种主要的缓冲区：

- 磁盘IO缓冲区

主要考虑不同两种IO中断:数据库请求与日志请求

1. 数据库IO中断请求：缓冲池

当一个线程需要从数据库读取一个数据页的时候，会产生一个IO请求并且在缓冲池中分配内存空间来存放从磁盘读取的数据。

当要把缓冲区的页存入磁盘时候，线程也会产生一个IO请求把缓冲池中的页存入磁盘中的目的地址中。

2.日志IO请求: 日志尾部

每一个事务都会产生一条日志entry，它们会暂时存储在内存队列中。然后会有一个独立的进程或者线程负责将其以FIFO顺序刷新到磁盘中。

通常采用的日志刷新方法是WAL，即一个事务在日志记录刷新到日志存储器之前不能被成功提交。就是先写日志再提交事务，而日志刷新可能会被推迟一段时间以实现一个IO请求批量提交记录。

- 客户端通信缓冲区

SQL通常被用于"pull"拉模型，即客户端可以通过重复发送SQL FETCH请求不断获取结果元组，而DBMS会尽可能再FETCH流到来之前做数据pre-fetch工作。

客户端可以用游标机制，在客户端存储近期即将被访问的结果，而不再依赖**操作系统通信缓冲区**做数据的pre-fetch

- 锁表

Lock Manager模块负责管理，锁表会由所有的DBMS worker共享

准入控制（Admission Control）

首先我们来讨论一般什么情况下会发生"抖动"呢?

- 可能是由内存问题造成的

可能缓冲池满了，放不下磁盘调来的的页，需要频繁在缓冲池和磁盘中调页淘汰页

可能是一些内存消耗比较大的操作如排序，哈希连接等导致

- 可能是因为锁竞争导致的

可能发生死锁，事务需要回滚以及重启

因此一个多用户的系统通常都要设计**准入控制**模块，实现满载情况下的"graceful degradation": 具体指的是事务的延迟会随着事务到达率增加，但是吞吐量会一直维持在峰值。

那具体如何实现呢？可以从两个层面实现：

1.**在用户请求到达的时候进行准入控制**可以通过确保客户端链接数处在一个临界值中，避免对**网络链接数，本质是socket数量**的过度消耗，一般可以在应用层，事务处理层和网络服务层实现

2.在**DBMS内核的Query Processor模块**实现，即在**执行查询计划的时候进行准入控制**

Query Processor在把query进行转换和优化生成执行计划之后,会继续决定是否推迟执行一个查询或者是否使用更少的资源来执行查询或者是否需要额外的限制条件来执行查询。

具体依赖的是**Query Optimizer**提供的信息（查询所需资源）以及系统当前的资源

具体来说Query Optimizer可以提供的信息有：

1.确定要查询的磁盘设备以及顺序和随机磁盘IO次数的估计

2.根据计划的算子估计CPU负载与所要查询的元组个数

3.评估查询数据结构的内存使用情况，包括连接，排序，哈希连接等操作消耗的内存

其中第三点最为重要，因为我们也说了"抖动thrashing"的主要原因还是因为内存压力，准入控制主要还是为了解决系统的内存压力问题的。

并行架构：进程和内存的协调

本部分侧重讨论进程模型与**Memory Coordination**问题

Shared-Memory

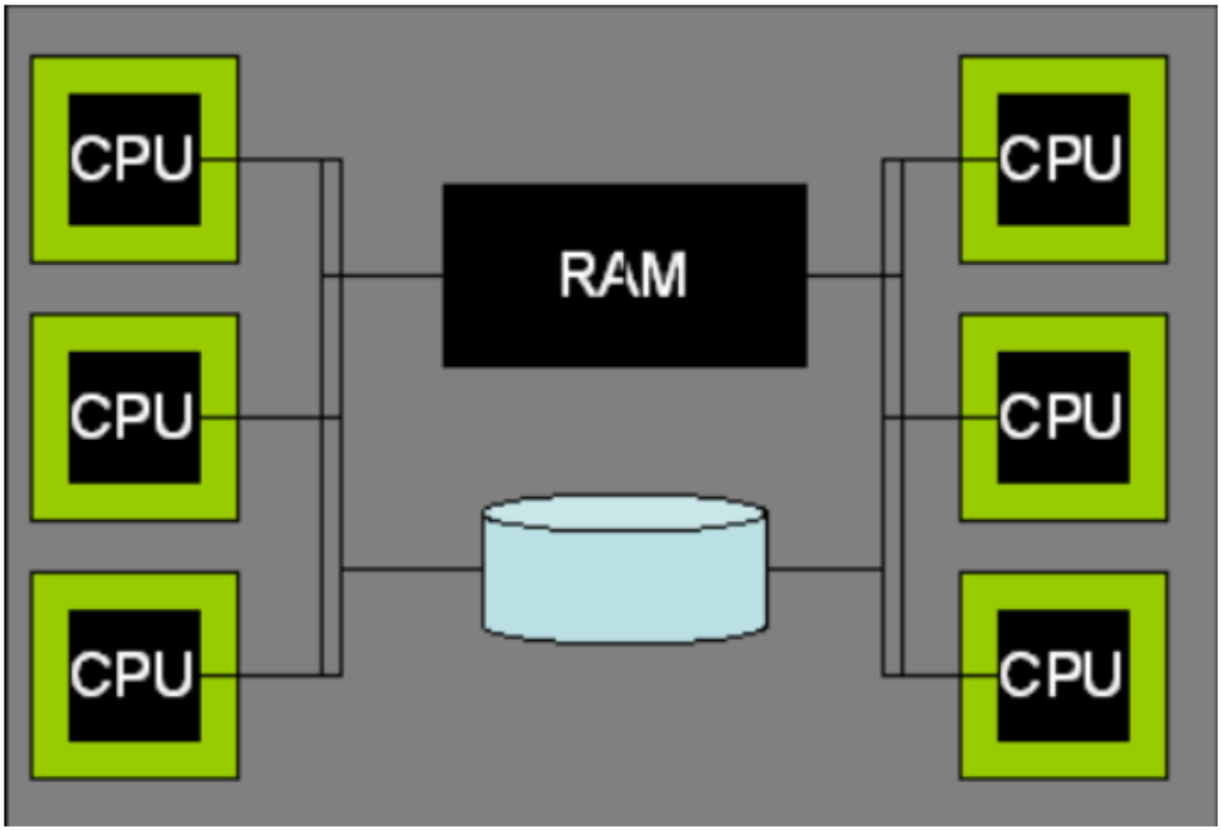
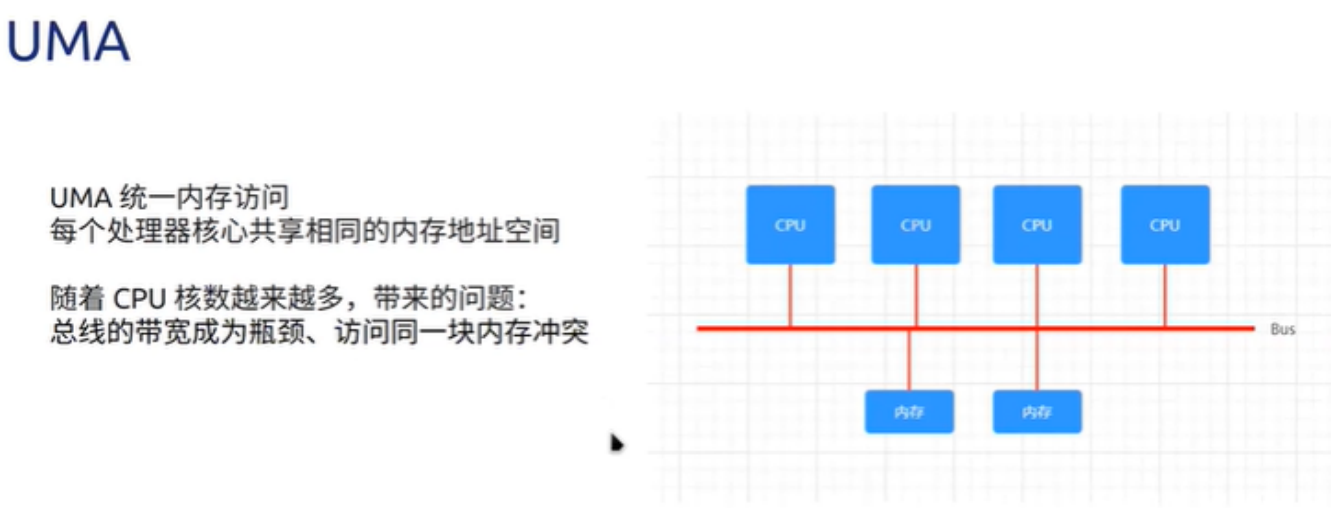


图 3-1 共享内存体系架构

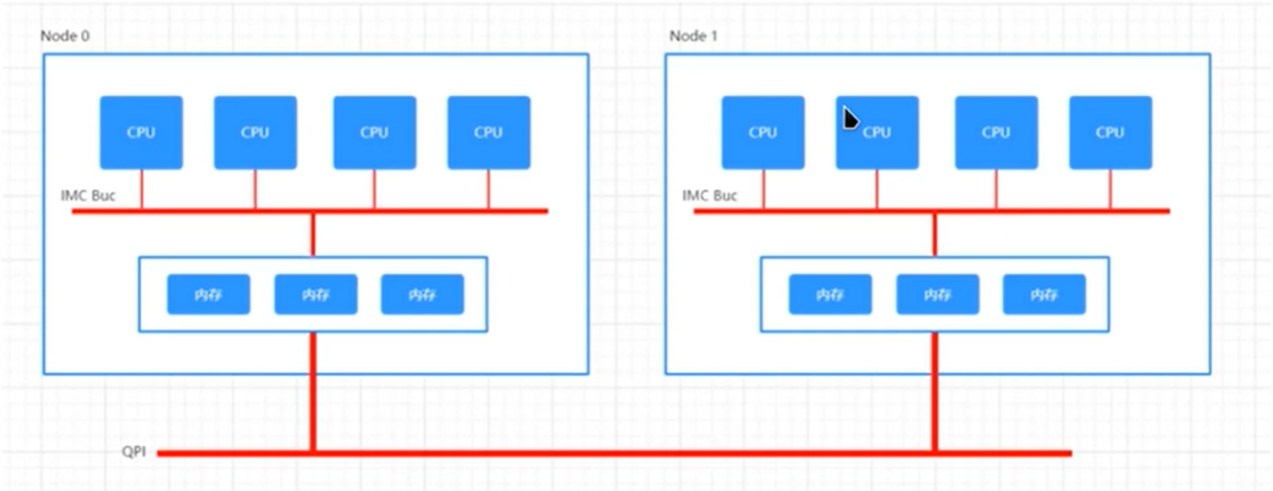
所有处理器共享相同的内存地址空间

- UMA（均衡访问模型）



- NUMA（非均衡访问模型）

NUMA



避免了对内存共享系统总线的占用

主要做法

- 多个CPU与部分内存构成一个Node，Node内部的CPU通过IMC Bus进行通信
- 不同Node之间通过QPI通信
- QPI的延迟远比IMC要大，因此访问远程内存的时间远比访问近程内存要慢

TiDB这种架构不适合在NUMA上跑，原因是假如是Node1接受客户端请求，但是数据在Node0上，会导致请求走QPI总线，导致延迟过高。

解决方法: 提前绑核，把所有CPU绑定在同一个Node

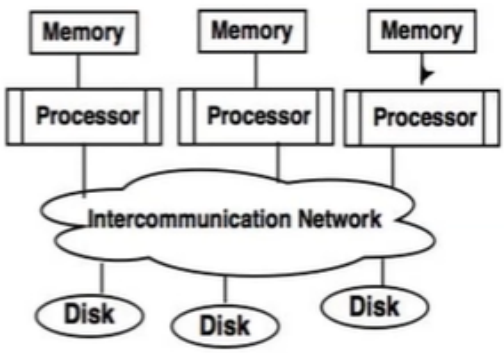
Shared-Disk

Shared-Disk

特点：管理成本低，单节点故障不会影响其他节点

依赖分布式锁管理设备，高速缓存一致性协议

代表产品：Amazon Aurora



Shared disk system in Parallel Databases

依赖分布式锁：不同处理器通过内部网络相互通信，多个处理器同时访问磁盘必定会带来并发安全问题，因此需要引入分布式锁管理磁盘的读取和写入。

高效缓存一致性协议: 每个CPU与内存不是直接打交道的，是通过多级缓存来缓冲CPU和内存处理速度能力的差异。CPU处理速度远远高于内存处理速度。每个CPU绑定的缓存不一样，可能缓存写入内存的时候会出现不一致情况。

通常有两种解决方法:

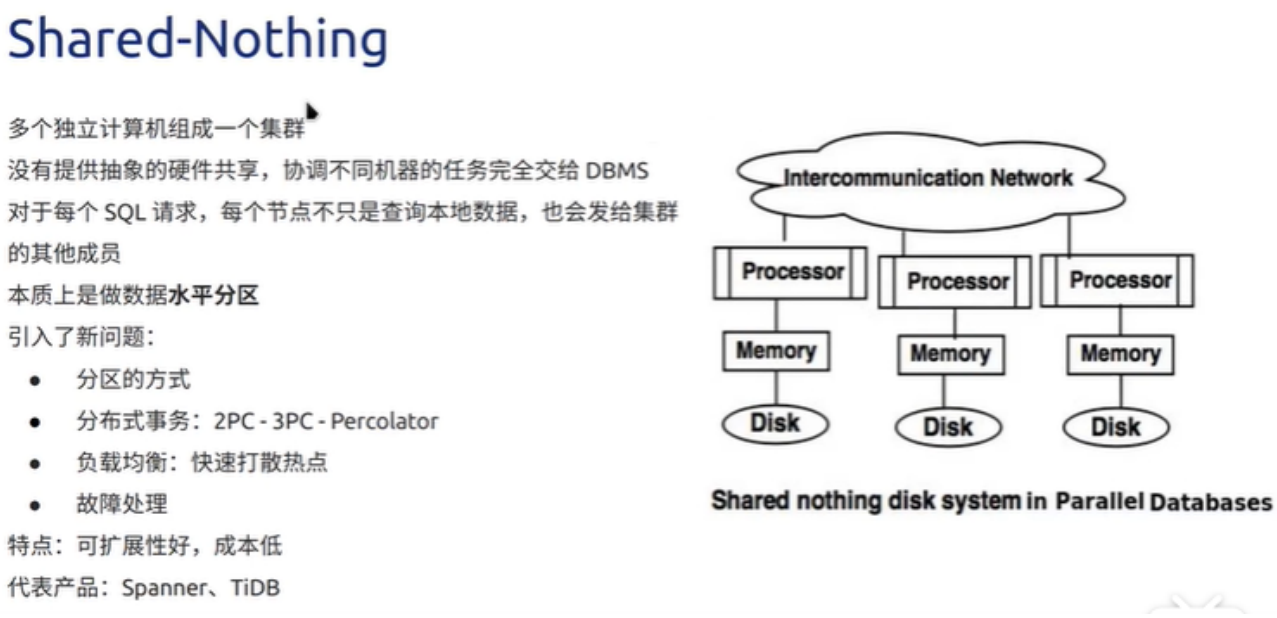
1.总线锁，某个核心独占总线，其他CPU不能通过总线与内存通信，这样做效率无疑会很低

2.MESI缓存一致性协议

若干个CPU核心通过ringbus连到一起。每个核心都维护自己的Cache的状态。如果对于同一份内存数据在多个核里都有cache，则状态都为S（shared）。一旦有一核心改了这个数据（状态变成了M），其他核心就能瞬间通过ringbus感知到这个修改，从而把自己的cache状态变成I（Invalid），并且从标记为M的cache中读过来。同时，这个数据会被原子的写回到主存。最终，cache的状态又会变为S

Shared-Nothing

分布式架构



1.分区可以怎么分？hash,partition,round-robin等

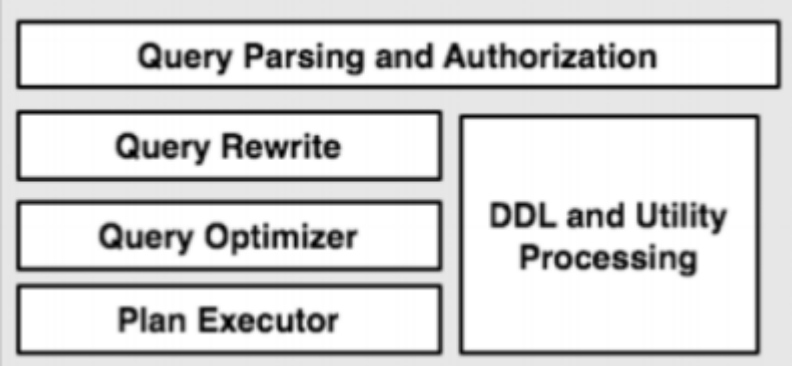
2.分布式架构通常会出现网络分区问题，分布式事务的解决方案也有一个演变过程

2PC -> 3PC -> Percolator(Google提出，可以了解下)

分布式事务是日后需要深入了解的领域

3.避免单个节点成为热点，这会导致该节点成为系统的瓶颈，同时要把热点节点的数据打散到别的节点上去

关系查询处理器



这部分实践可以以**Apache Calcite**来作为学习。

查询处理与鉴权

Parser主要任务：

1.检查这个查询语法，是否被正确定义

2.解决名字与引用

a. 把表名规范化为<数据库，模式，表名>，并且检查表是否被注册

3.把查询转换为优化器使用的内部形式

4.核实这个用户是否被授予权限执行该查询

查询重写

Logical Optimizer 负责简化和标准化查询，无需改变查询语义

- 视图展开
- 简化常量运算表达式

eg: a > 1 + 1 => a > 2

- 谓词逻辑重写

eg: R.x < 10 and R.x = S.y => R.x < 10 AND S.y < 10 AND R.x = S.y

左部分要全表扫描y

而有部分如果在y上建立了索引，可以利用索引来找y

- 语义的优化

如冗余连接的消除

Query Optimizer

也就是**Physics Optimizer**,把内部查询表达转换为一个高效的查询计划，指导数据库如何去取表，如何排序，如何join。

- 计划空间
- 选择代价估计(Cost Model)

- 搜索算法
- 并行
- 自动调优

比较详细的请看原文。

A Note on Query Compilation and Recompilation

“预处理”：将查询传递到 Parser、Rewriter 和 Optimizer，把生成的查询执行计划存储起来，并且再后面的“执行”语句中使用。
适用于用程序变量代替常量的动态查询。
难点：优化器选择的查询计划不一定总是最优的。

“缓存”：在查询计划的缓存中存储这些动态查询执行计划。
随着数据库的变化，需要重新优化预编译计划：

- IBM - 在多次调用中获得可预测的性能，而不是每次查询计划都获得最优性能
- Miscrosoft - 希望数据库系统实现自我调优，更积极地重新优化执行计划

Query Executor

负责执行一个具体的查询计划。

这里的查询计划是一个把很多操作连接在一起的数据流程图，封装了基本表的访问和各种查询执行算法。

查询执行器常用模型：迭代器模型

存储的管理

数据库如何管理硬盘的数据?

DBMS与物理硬盘的交互方式:

1.DBMS直接与底层面向磁盘的块模式设备驱动程序进行交互

2.DBMS使用标准的OS文件系统设施

空间控制

事实：顺序读写比随机读写要快10到100倍

时间控制

双缓存：数据库缓冲池一份数据，OS缓存又一份，会导致数据冗余。而内存中复制数据其实是非常消耗CPU的过程。因此写操作应该直接把数据写入到磁盘，不要走缓存

缓冲管理

通常用一个哈希表来维护缓冲池中帧和磁盘中块的映射关系

缓冲管理

DBMS 会在自己的内存空间中实现一个大型共享缓冲池

数据库一般使用一块内存空间（frame）和硬盘上的内容进行一一映射，这个映射不涉及到数据内容的转换以避免额外的 CPU 开销。

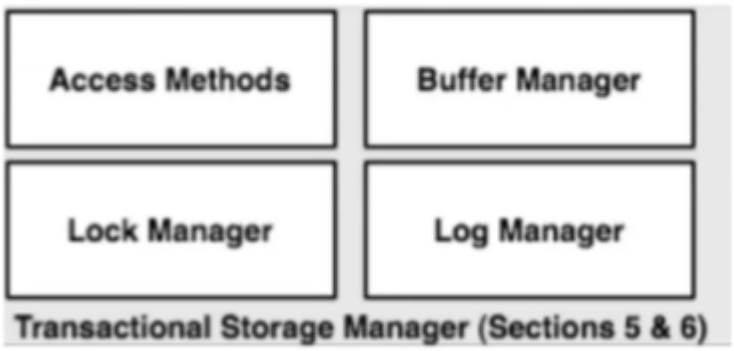
元信息 Dirty flag 决定是否需要将 frame 写回磁盘。

增强版的 LRU 页面置换策略

操作系统的页面替换策略（LRU 和 CLOCK）往往表现糟糕。

大多数系统使用 **LRU-2** 方案进行全表扫描。

事务和并发控制恢复



1.并发控制的Lock Manager

2.错误恢复的日志管理

3.用于分离IO的缓冲池

4.底层磁盘上管理数据的Access Methods(索引的维护，磁盘数据什么方式组织？)

三大并发控制方法：

1.Strict two-phase locking (2PL)

上锁阶段：能上多个不互斥的读锁和一个与所有锁都互斥的写锁，

解锁阶段：只能解锁

事务在读任何数据之前需要一个共享锁，而在写之前需要一个排他锁。 一个事务所拥有的锁，会一直保持到事务结束时才自动释放。当一个事务等待锁的时候，会中断然后移动到等待队列。

2.MVCC

每一个数据都有一个版本号（undolog）

删除数据不会删除磁盘上数据，只是标记已经删除

为过去某一时间点的数据 库状态保存一个一致的副本，即便在某一固定时间点之后数据库状态发生了改变， 我们也可以读到数据库的一个过去的状态

3.OCC(乐观并发控制)

不适合并发特别高的场景，事务提交时解决事务冲突问题

lock & latch

数据库会自己实现一个锁控制系统，称为Locking。数据库系统会自己维护一个Lock Table记录Transaction/Lock/Object之间的关系，这样当Abort一个Transaction的时候，就能将与其关联的所有Lock都释放掉。此外，由于加锁的顺序是由用户存取数据的顺序驱动的，因此死锁检测也是一个必不可少的工作。这种锁系统主要是针对Transaction的。

数据库系统也会在访问数据结构时使用更细粒度的锁对数据及数据结构进行保护，这种锁称为Latching。一般来说Latching都是由操作系统或者硬件指令提供的基础设施，因此对于Latching的使用要避免出现死锁的情况

	lock	latch
对象	事务	线程
保护	数据库内容	内存数据结构
持续时间	整个事务过程	临界资源
模式	行锁、表锁、意向锁	读写锁、互斥量
死锁	通过 waits-for graph、time out 等机制进行死锁检测与处理	无死锁检测与处理机制。仅通过应用程序加锁的顺序（lock leveling）保证无死锁的情况发生
存在于	Lock Manager 的哈希表中	每个数据结构的对象中

事务隔离级别

事务隔离级别

ANSI SQL 规定了几种隔离性级别：

1. READ UNCOMMITTED
2. READ COMMITTED
3. REPEATABLE READ
4. SERIALIZABLE

MVCC 和 OCC 中几种隔离级别：

5. CURSOR STABILITY
6. SNAPSHOT ISOLATION
7. READ CONSISTENCY

游标稳定：

这个等级是为了解决已提交读的更新丢失问题。假设有两个事务 T1 和T2。 T1 以“已提交读”模式运行，读取数据项 X（假设是银行账户值），记录这个值，然后根据记录的值重写数据项 X（假设为原始账户增加￥100）。 T2 同样读写了X（假设从账户取走￥300）。如果 T2 的行为发生在 T1 的读和写之间，那么 T2 对于账户的修改将丢失，即对于我们的例子而言，该账户最终将增加￥100 而不是减少￥200。游标稳定中的事务将根据查询游标在最近读取的数据项上加一个锁，当游标移走（如数据被提取）或者事务中止时释放该锁。游标稳定允许事务对个别数据项目按照“读—处理—写”的顺序来操作，其间避免了其他事务的更新干扰

快照隔离：

一个以快照隔离方式运行的事务，只对自身开始时的数据版本进行操作，不受在这个时间点后发生的其他事务对该数据的改变的影响。这是 MVCC 在数据库产品中的主要应用之一。当事务开始时，它从一个单调递增的计数器中得到一个开始时间戳，当它成功提交时得到一个终止时间戳。对于一个事务 T 而言，只有当具有与 T 重叠的开始/结束时间戳的其他事务不去写事务 T 要写的数据时，事务 T 才会提交。 这种隔离模型更依赖于多版本并发的实现， 而不是锁机制 .

日志管理器

主要职责：

1.保证已提交事务持久性

2.协助中止事务回滚以确保原子性

3.系统崩溃或者非正常关机时候使得系统恢复

日志管理器在磁盘上维护一系列日志记录并且在内存中维护一个数据集，宕机的时候内存中数据结构要通过日志和数据库中的数据进行重建。

WAL（Write Ahead Log)

- 每个对数据页的修改都应该产生一条日志，这个日志必须在内存页落盘之前落盘，**日志先落盘数据才落盘**
- 数据库日志记录必须按顺序flush写入磁盘，日志r flush表明r之前的日志记录都被刷新
- 事务提交之前，再提交返回成功之前，提交日志记录必须flush到磁盘中

如何提高WAL性能?

1.Direct模式

数据项原地更新，直接覆盖原来数据，而不是先删再插入

2.Steal模式

如果事务粒度很大，最后提交的时候写磁盘可能会成为瓶颈，因此可以在事务执行过程中写，一点一点地写

3.Not-force模式

事务commit时候不需要把数据页落盘（因为已经写了redo日志）

如何减少日志大小?

可以只记录逻辑操作（如sql语句）

而不是物理操作，如元组插入后的字节范围情况，对文件以及索引块中的字节

但代价是**逻辑上的撤销重做性能下降**

因此通常采用的是记录物理操作和逻辑操作，物理日志用于重做，逻辑日志用于撤销

如何避免崩溃恢复过长

可以使用recovery log sequence number来记录日志顺序

使用checkpoints周期性记录recover LSN，可以直接从checkpoint开始恢复，不需要从头恢复

数据库的备份机制

通常通过日志机制来进行备份

将数据库日志（一般指Bin-Log）近实时的写入另一个位置，然后再进行恢复。这里又有2种做法，一种是从日志中重建SQL进行回放，另一种就是直接回放数据变更。前者通用性好，可以跨不同数据库vendor进行数据复制；后者性能更高

扩展阅读

论文阅读笔记知乎 阿里云、Amazon、Google云数据库方案架构与技术分析 怎么打造一个分布式数据库 细说分布式数据库的过去、现在与未来

开源数据库的现状方向与未来