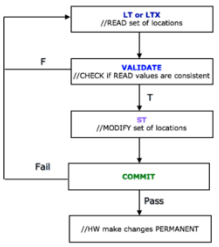
PMFS 做了以下贡献： * 硬件原语支持，`pm_wbarrier`，它能够保证一定将数据写入 NVM。PMFS 通过三条指令共同保证 durability。1）clfush: flush the cacheline;2)sfence:ensure the completion of store;3)pm_wbarrier:ensure the durability of every store to PM。* 重新设计了系统结构，原来从内存到 disk 是块操作，现在 cache 到内存是 byte addressable * 保证数据是正确的，原来的方式是确定一个 atomic point，在 point 之前之后都是确保数据是正确的，关于 point 的寻找，有两种经典操作，一个 COW，一个是 Journaling。前者的方式牵涉到具体的修改，涉及数据结构，而 Journaling 方案是修改前先做 log，log 里面可以加入 atomic 记录，最后后台把 log 上的变化更新上去就是 Journaling。**PMFS 该选择的是混合方案**？如果数据是分散的且小的，使用 Journaling ，因为 COW 粒度很大，如果修改很小，那么很多空间都浪费了，而且如果它很分散，那么我调整的时候要调整很多指针，要很多的指针更新 atomic 完成，atomic 操作不支持辣么多指针同步更新；反过来，集中的大数据用 COW，Journaling 不适合因为它先要写 log，然后还要更新到最终地方，意味着要**写两次**，有一个写放大的效果，当修改数据很多的时候，意味着放大的数据也很多，造成影响很大。**Why do memory reordering and cache play an important role in PMFS consistency? How does PMFS maintain consistency**？因为文件系统的 recovery 需要依赖写入的 order。因为 PMFS 在元数据的记录中使用了 journal 的方式，因此在写入 log 的时候需要保证数据写入的顺序才能保证日志能够完整地记录所有的 commit 以保证 consistency。而 reorder 会导致不可预测的日志状态，所以 order 是非常重要的。一般来说是用 cflush 来做的，但是 cflush 是从 cache 到 memory 并不能保证从 memory 中的 controller 到真实的内存中，cache 的作用也在此。//PMFS 引入了一个硬件原语：pm_wbarrier，用所谓细粒度的 log 来实现 consistency。在可以使用原子写的地方，PMFS 会率先使用原子写。对于 meta data 而言 PMFS 使用了 journal 的方式，因为 meta data 的 size 比较小。对于真正的 data 会使用 copy on write 的方式来保证一致性。在 PMFS 没有被 umount 的时候断电， PMFS 会在下一次 mount 的时候进行 recovery。对于 PMFS 的 allocator 的 consistency，会遍历文件系统的 B 树来进行 recovery。这几方面综合就是 PMFS 为了解决 consistency 问题做的事情。**There are three techniques to support consistency: copy-on-write, journaling and log-structured updates. Please explain the differences among them. Which technique does PMFS choose?** 前两者见上文。Log-structured updates 是 F2FS 基于的 recovery 方式，相比于 journaling 的多次写，LFS 只需要一次追加写。如上题所述，PMFS 采取了一种 hybrid 的实现方式，由本地原子写，CoW 和 journaling 共同实现了 consistency。**What granularity does PMFS use for log? Please explain the reason.** PMFS 针对上述的三种技术，进行了成本的分析，主要是分析写入的数据大小和 pm_wbarrier 操作的次数，发现在对 metadata 的更新的时候，将每个 log entry 设为 64 byte，跟 cacheline 一样，有最小的 overhead。//**RAMCloud：** RAMCloud 通过分布式的方式提供高可靠性，数据可以出现在多台机器内存里，从而达到高可用，当出现一部分 failure 的情况下，可以保证数据不会丢。如果整个 cluster 挂了，数据还是会丢，所以还要考虑 durability，最后还要考虑当机器 crash 之后，能够尽快恢复。问题主要在于 crash，RAMCloud 提出了一个 bufferd logging。当 master 收到写请求，它将要写入的 object append 到内存里的 log（使用 hash table 访问）中，然后再将 log entry 通过网络分发到各个 backup 去。backup 将接收到的信息 buffer（加了电池保证断电不会丢失）到内存中，然后立即向 master 返回一个 RPC 请求，而不需要立即写入到 disk。master 接收到 backup 的请求之后，向 client 返回。对于 backup，在 buffer 满了之后才将 buffer 的数据以 log 的形式异步写回 disk。当出现 crash 的时候要尽快的 recover。原来是从 disk 去 recover，而现在它利用数据分布在多台机器上，因此可以并行的去恢复一台机器，集所有包含该数据的机器的资源去恢复机器，加速 recover。内存场景存储大量数据，因此需要大规模的集群，带来的问题是 crash，但是同样好处是可以集合所有机器去恢复一台机器。原来单点的瓶颈可能在 disk 上，现在因为有上百台机器，每台机器只需要恢复 1%左右的数据，disk 就不会成为瓶颈，但是 IO 可能会成为瓶颈；另外，最后一步的恢复也可能成为瓶颈，因为要把拿来的数据重新变回 log 结构，这件事是单点完成。为了使得最后一步的恢复不成为瓶颈，RAMCloud 一边恢复服务一边恢复数据。它的假设是这些数据不是马上被访问的，访问的数据先替你恢复，更快的去恢复服务，后台再慢慢把数据恢复出来，相当于把数据 recovery 和 service 重新运行 overlay 在一块 **Which policy does RAMCloud use to place segment replicas and how to find the segment replicas during recovery?** 以往的实现是中心化的协调器，这样会造成性能瓶颈。所以 RAMCloud 采纳了去中心化的思想，利用了随机化和微调的方式，来分散备份。有些类似 k choises 的选择，RAMCloud 会先随机选择一些，然后再从中选出最合适的，并且加入了 reject 的机制来保证在乐观并发的情况下不会产生竞争的问题。同时为了保证尽可能贴近最优解，RAMCloud 考虑了硬盘速度以及硬盘上已经有的 segment 的数量来进行微调，保证尽可能的均匀。// recovery 的时候以往的实现是在 coordinator 里维护一个中心化的表，这样的做法向上面提到的一样会造成瓶颈，因此 RAMCloud 在 recovery 的时候会问所有的备份，备份会返回一个它存储的副本列表，整个过程是并行的而且 RAMCloud 用了自称 fast 的 RPC，因此整个过程不会特别慢。**Does RAMCloud support random access? If so, please explain how.**支持。RAMCloud 在每个 master 上都维护有一个哈希表，结构为 <table identifier, object identifier>，通过哈希表，可以进行随机的访问。//**Scalable Lock**:**Lock**。Ticket lock: lock(next_ticket ++); unlock(current ticket++) ; unscalble(bad performance)Unscalable lock: 较短的临界区域会导致性能的陡降，这是对 lock 相关的 cache line 的 contention 导致的。性能下降的主要原因：拿锁时需要从之前 hold 锁的核的 cache 中获取数据，解锁时需要 invalidate 其他核的 cache，然后其他核从持锁核顺序读新数据。性能骤降的原因：一个锁从一个核转移到另一个核的时间是与等待锁的核的数量呈线性关。MCS lock: lock 维护一个队列，每次上锁时在队列末尾添加一个 mcs_node（将原队列末尾的对象的 next 指针设置成自己， is_locked 变量设置成 true），解锁时判断队列中是否还有其他对象，若有将其 mcs_node 的 is_locked 变量设置成 false。MCS lock 的局限性：不可 abort。**Why does the performance of ticket lock collapse with a small number of cores?** 在 ticket lock 的实现中，是维护两个 ticket 变量，当前 ticket 和 next ticket。在 unlock 操作中是将当前 ticket 自增从而使得下一个拿到了 ticket 的 core 获取到了锁。//凡是遵循文章中提出的硬件缓存一致性模型的系统，都会遇到 ticket lock 的 rapid collapse。文章提出的硬件缓存一致性模型用目录的方式类比了 CPU 中的缓存，并且用马尔可夫链模型，对 ticket lock 的问题进行了分析。其中每一个节点代表有几个 core 在等的状态，Arrival and Service Rates 分别代表了在不同状态下锁的获得和释放。其中到达率跟在没有在等的 core 的数量（n - k）成正比，服务率和在等的 core 的数量（k）成反比。所以随着 k 的增大，服务率是减小的。这使得模型得到了一个数学上的结论，锁的获取时间和正在等待锁的 core 的数量是成正比的。//因此随着 core 的增加，在 serial section 很小的时候，Sk = 1/(s + ck/2) 中 k 对 Sk 的贡献越大，因此越容易受到 k 的影响。这也就是为什么，在 serial section 很小的时候，只是多了很少一些在等待锁的 cores，就出现了性能的雪崩。**To mitigate the performance collapse problem of ticket lock, we can replace it with MCS lock. Please descibe the MSC lock algorithm in C.**

```
struct QNode {
  QNode* next;
  bool locked;
}
void acquire_lock(QNode** L, QNode* I) {
  QNode* predecessor;
  I->next = NULL;
  predecessor = fetch_and_store(L, I);
  if (predecessor != NULL) {
    I->locked = true;
    predecessor->next = I;
    // Spin
    while(I->locked);
  }
}
```

```
void release_lock(QNode** L, QNode* I) {
  if (I->next == NULL) {
    if (compare_and_store(L, I, NULL)) {
      return;
    }
    while(I->next == NULL);
  }
  I->next->locked = false
}
```

Transaction Memory：txm 能够在内存级别上支持有具有 ACID 的特性的操作。解决 atomic 的问题，无非就是 COW。如果内存用 COW ，需要有一些 policy 去管理内存。Tx Memory 被实现成 cache 级的 memory，因为在 cache 上，本身就要解决数据之间的 consistency 的问题和 track 每一个数据谁访问过了，相当于这种实现可以减少新的硬件。//测试发现 tx memory 提供的性能实际上和 fine grained lock 的性能差不多，fined grained lock 就是每一个数据都有一个 lock 去管理，无关的操作也不会形成冲突，tx memory 也一样；大家都是 fine grained 进行管理，tx memory 的好处是由硬件去 track 操作的变化，软件有软件管理的 overhead；反过来，fined grained lock 是对一个对象的 lock，而非一个地址一个 lock，而 tx memory 是硬件上的保护，虽然 track 开销小，但它是所有的数据都要去 track，平衡下来，性能是差不多的。**Lock 的 limit**: Priority inversion(低权限的拿了锁，从而实现了对高权限的抢占)；convey（频繁的上下文切换会导致性能下降）；dead lock。**具体执行流程：**当事务提交时，会将 XCOMMIT (discard on commit[old value])标记为 EMPTY，将 XABORT (discard on commit[new value])标记为 NORMAL(contains committed data)；当事务废弃时，会将 XABORT 标记为 EMPTY，XCOMMIT 标记为 NORMAL 当事务性缓存需要把部分 CL 替换出去时，首先搜索被标记为 EMPTY (contains no data)的位置，之后再搜索被标记为 NORMAL 的位置，最后才是 XCOMMIT 的位置。LT: ‘XABORT’ -> (’ NORMAL’ ->’ XABORT’ , ADD‘XCOMMIT’) ->’ T_READ’ (收到’ BUSY ‘, ABORT)。**What is the shortcoming of transactional memory compared with conventional locking techniques? Do you have any suggestion to mitigate those problems?** Disadvantages:事务性内存对于 short critical section 的场景下使用比较合适，在大的时候，会使得事务比较频繁地被中止。这也是论文中提到的。除此之外，事务性内存也使得开发者的思维要一定程度的转变，不是很容易被接受。而且目前大多数代码都是使用锁的方式来实现的，代码迁移成本很高。 Suggestions 对于前者，可以向文中所说的那样加入对应的硬件支持。对于后者，可以实现一个 lock to tm 的中间层，这样对于 tm 的接受度会高很多。//**NoSQL**: **Data Model**:是一个稀疏的、分布式的、持久化存储的多维度排序 Map。Map 由 key 和 value 组成：Map 的索引 key 是由行关键字、列关键字以及时间戳组成；Map 中的每个 value 都是一个未经解析的 byte 数组。即 (row:string, column:string,time:int64) -> string。一个存储的数据可能为{“aaa” {“A:bb”：{“15”：“hi”}}}。**Table,Tablet 和 SStable 的关系**：Tablet 是从 Table 中若干范围的行组成的一个相当于原 Table 的子表，所以多个 Tablet 就能组成一个 Table。而 SSTable 是 Tablet 在 GFS 文件系统中的**持久化存储的形式**，即 Tablet 在 Bigtable 中，是存在一个一个 SSTable 格式的文件中的。**Building Blocks**：GFS，提供存储日志和数据文件；SSTable,BigTable 数据在内部使用 SSTable 文件格式存储；Scheduler；Chubby,负责 master 的选举，保证在任意时间最多只有一个活动的 Master。**Refinement**：Locality groups：将不经常一起访问的列分开不同的组，每个组存放在单独的 SSTable 中，加速读；Compression：时间戳不同值相同，不同列中近似的值，邻近行中相似的值；Two phase：先在长窗口压缩长字符串，然后在短窗口寻找重复值；Bloom Filter:加速 SSTable 的查找。**Describe what will happen when a read operation or write operation arrives.**读取和写入是以 Tablet 作为一个 Unit 进行的。在进行写操作的时候，Tablet Server 会先做一些检查，保证请求的合法以及权限问题。权限的检查是通过检查一个在 Chubby 中的列表进行的，这个列表会被 Client Library 缓存住。一次被允许的写操作会先进入 Commit Log，在处理 Log 的时候采取了批处理来提高吞吐。在操作被 Commit 后，它的内容会被插入 MemTable 里，当 MemTable 的 size 超过一个阈值的时候，会让当前的 MemTable 进入一个 frozen 的状态，随后创建一个新的 MemTable，Frozen 的 MemTable 就可以以 SSTable 的形式写入 GFS。在进行读操作的时候，Tablet Server 会在做了一些检查保证合法后，在 MemTable 和 SSTable 的一个 merge 后的 view 中来进行读操作，这样可以保证可以读到最新的值。**Describe which applications are BigTable suitable for and not suitable for.** 由于 BigTable 针对数据存储进行包括压缩等各方面的优化，以及在事务的一致性上做出了让步，BigTable 对于那些需要海量数据存储，高扩展性以及大量的数据处理，但又不要求强一致性的应用是十分适合的，比如 Google Earth 等。也因此，对于那些需要强一致性，需要同步更改多行数据的应用来说，BigTable 是不合适的。



DTX：Google 的 Bigtable 和 Megastore 存在一些问题，前者对于复杂可变的模式，或者需要大范围强一致性复制的场合不适用；后者的写吞吐量很差，即使是它的半关系型数据模型和对实施复制的支持很好，但是也非完美的选择。Spanner 同 BigTable 的区别如下,从简单的 key-value store 加强到 temporal multi-version database 数据以半关系型的 table 组织；支持 tx 语义；支持 SQL 查询。**系统架构**： Universe，整个部署；universemaster：单例，维护所有 zones；placementdriver：单例，负责在 zones 之间迁移数据；zone：部署节点，是管理配置的单位，也是物理隔离的单位；**Zone 内部**：zonemaster：每个 zone 一个，负责将数据分配给当前 zone 的 spanserver；location proxy：每个 zone 有多个，为 client 提供定位到需要的 SpanServer 的服务；spanserver：每个 zone 有成百上千个，为 client 提供数据服务。**Span Server**:***tablets***：存数据的最小单位，概念和 BigTable 的 tablet 相近（一个 tablet 往往有多个备份副本，会存在其他 zone 的 span server 上）；***Paxos state machine***：每个 span server 维护一个用来选举（使用场景：当需要修改本地某个 tablet 时，由于需要同时修改其他 span server 上的副本，此时用每个 span server 上的 Paxos 状态机来处理一致性的问题(选出 leader 等)），tablet 副本的集合组成 Paxos group；写请求由 Paxos leader 负责，读请求由任意足够 up-to-date 的 tablet 所在 span server 执行都行；***lock table***：（单 Paxos group 中选为 leader 可用）标示该 Paxos group 中对应数据的上锁情况；***txn mngr***：（多 group leaders 中选为 coordinator leader 可用）当要执行的 txn 跨多个 Paxos group 时，需要对这些 groups leader 进行再选举，选举出来 coordinator leader & non-coordinator-participant leader。前者使用 txn mngr 来协调这个 txn（协调方法：txn mngr 对其下管理的 Paxos leaders 执行 2PL）**True time API**：TT.now()：返回一个时间段[earliest, latest]，保证被调用的一刻，所有 spanserver 的本地时间都处在这个范围内；TT.after(t)，TT.before(t)：检查是否所有 spanserver 都经历了 t；或都还没有经历 t。**What is external consistency? What's the difference between external consistency and serializability?** External consistency：如果事务 2 发生在事务 1 提交之后，那事务 2 的时间戳要比事务 1 提交的时间戳要大。Serializability 是数据库隔离性上的一个级别，所有的事务被序列化来执行的。EC 是一致性上的概念，在分布式场景下，EC 是更难做到的，因为时序对时间的精度要求很高，在分布式场景下，有可能出现因为不同机器系统时间不一致导致事务 2 拿到一个比事务 1 提交的时间戳更小的时间戳。S 是隔离性上的概念，如果做到了 EC，就一定可以做到 S。**How does Spanner achieve the external consistency?** Spanner 之前是使用全局的时钟来解决 EC 的。但是 Sp 创新地使用了原子钟和 GPS 来作为全局的时钟，以此来实现 EC。在事务的执行中，Sp 会保证，每个事务的 commit timestamp 都会在其 start 和 commit 之间。Sp 依赖的底层容器集群系统 Borg 会维护一个 True Time API，这个 API 会返回精度为 ε 的时间区间[t-ε,t+ε]。因此每个事务会在 start 和 commit 的时候分别调用一次 True Time API，拿到两个时间区间[t1-ε,t1+ε]和[t2-ε,t2+ε]，因此在区间[t1+ε,t2 -ε]之间的时间都是可用的，如果 t1 和 t2 很接近，那最多需要等 2ε。**What will happen if the TrueTime assumption is violated? How the authors argue that TrueTime assumption should be correct?** 这会导致 True Time API 不能再用来保证 EC，文章中提到，CPU 造成的问题比时钟问题多六倍，因此跟硬件造成的错误相比，时钟造成的问题不值一提，可以被视为是值得信任的。//**RDMA**:NewSQL 最大的问题在于为了做一件事，额外做的事情开销很大，有用的操作可能只占总消耗的 10%，但是为了支持该操作，需要提供 ACID，需要有 logging，需要有 locking 和 latching，为了支持不同的查询需要有 index，还要有 buffer 去支持 isolation，这些开销很高。DrTM 从多个角度解决：recovery，它有 NVM，用 in-memory 的 logging 来提高；locking，使用 RDMA 去提升；latching 这种单机里面的锁，使用 tx memory 去提升； buffer，使用完全内存的存储，这样就不需要 buffer 了。**DrTM 系统**。DrTM 使用了 HTM 的 ACI（没有 D）的特性来处理本地的 Tx 执行，使用单向的 RDMA 来处理多 HTM 事务的执行，减少了系统 latch 和 lock 处理花费的时间。DrTM 实现可以分成两层:事务层：处理事务，所有并发控制在这一层做；存储层：处理存储，不考虑并发控制，从而使得逻辑更加清晰，实现更加简洁。Transaction Layer：HTM（本地）+ Locking（远程）来实现事务（ACI），D 由 Logging + UPS（断电时将内存刷回持久化存储)来保证。**事务读写**：本地，HTM；RDMA 指令无条件 abort 目标机器冲突的 HTM 事务，意味着无法通过 HTM 事务中的 RDMA 直接访问远程记录。为此，DrTM 使用 2PL 在 HTM 事务中实际执行之前将所有远程记录安全地累积到本地缓存中，直到本地提交 HTM 事务将本地的更新写回到其他机器。Local TX vs. Local TX：HTM；Distributed TX vs. Distributed TX：2PL；Local TX vs Distributed TX：abort local TX；Lease-based protocol：read right。**事务流程**：获取 remote read set 和 write set 的锁，XBEGIN，远端读写直接操作本地的 cache，本地读写操作先检查相关数据是否上锁，若上锁则 Abort，若未上锁，检查 END_TIME；提交前同样需要检测 end_time，若超时，则 ABORT，否则 XEND；释放所有锁，并写回 remote 数据。**K-v store**：DrTM 的内存存储层为上层事务层提供了一般的 kv 存储接口。此接口的最常见用法是按给定键读取或写入记录。为了优化不同的访问模式，DrTM 提供了 B +树形式的有序存储和散列表形式的无序存储。DrTM 提供了两类存储：有序存储：基于 B+ Tree，没有充分利用 RDMA 的特性；无序存储：基于 Hash，充分利用了 RDMA 的特性，并且在事务层的保护下，不需要考虑冲突检测问题。DrTM 利用 HTM 强大的原子性和 RDMA 的强一致性来设计 HTM / RDMA 友好的哈希表。首先，DrTM 通过利用 HTM 的强原子性将竞争检测与 hash table 分离，其中键值对上的所有本地操作（例如，READ / WRITE / INSERT / DELETE）都受到 HTM 事务的保护，因此任何冲突的访问都将中止 HTM 交易。这显着简化了 race detection 的数据结构和操作。其次，DrTM 使用单向 RDMA 操作对远程键值对执行 READ 和 WRITE，而不涉及主机。最后，DrTM 将键和值及其对应元数据分离到一个**解耦**的内存区域，从而产生两级查找。这使得利用单向 RDMA READ 进行查找变得高效，因为一个 RDMA READ 可以获取 key 的 cluster。此外，分离的键值对可以实现 RDMA 友好，基于位置和主机透明的缓存。**无需存储实现**如下图。有序存储实现只需要知道它没有充分利用 RDMA 特性。**DrTM 的优点**：分离了存储操作与冲突检测；之前的实现偏向于 RDMA(而不是 Local)，增加了延迟；本地内存操作还是比 RDMA 快，但是之前的实现没有考虑本地缓存。**Why DrTM has the deadlock issue and how does it avoid the deadlock?** 在 DrTM 的 fallback handler 中，不能像传统的实现那样，一个简单的锁就可以解决问题，而是 fh 通过 2PL，对于任何 record 都是以 remote 的形式进行访问。这里就有可能产生死锁。因为涉及到 remote locks 的顺序。为了避免这个问题，DrTM 声明一个全局的释放和申请锁的顺序，避免死锁的问题。**Limitation of DrTM?** 首先，DrTM 没有做到很好的可用性，这是它最大的局限性。还有就是需要硬件特性的支持，导致在很多现有的硬件上没有办法完全复刻 DrTM 的工作，而需要一些适配性的工作。//**Tail latency**：原因：resource contention，skewed access patterns，Queueing Delays，background activities(gc，log compactions，data reconstruction）。**集群变大，延迟放大**：在大型分布式环境下，一个组件的延迟在 service level 就会被放大。如上所述，若每个单独的组件延迟很短时间，把他们整合起来的延迟时间就可能会很长。尤其是一些 request 的相应必须收集并行的、众多的、带有微小延迟的 servers 的 response 的时候，集合起来的性能延迟将是不可忽视的。**tail-tolerant techniques**：Hedged requests：把同一个请求发给多份备份，使用第一个返回的结果（改进，延迟发送第二次请求直到第一次请求超过延迟预期的 95%）；Tied request：将请求发给多台 server 并加上 tag，第一个请求开始执行时，移除其他队列上的同样 tag 的请求（改进：选取负载最低的 server）；Micro-partitions：将任务划分成多份（超过服务器数量），动态地将子任务分配给服务器；Selective replication：预测可能的负载不均衡，创造额外的复制品；Latency-induced Probation：暂时排除特别慢的机器，对排除地机器发送 shadow 请求，一旦发现问题减缓，把这些机器再加进来；Good-enough Schemes：返回一个足够好的部分结果，跳过不重要的子系统；Canary Requests：在向所有子服务器发送请求之前，先找一两个测试一下，如成功才发送。//**NFV**:*特点：1）*灵活:很容易添加新的功能 2）模块化：通过组合元素来实现功能 3）Open：允许开发者添加 Elements4）效率：与硬件性能相差不大。Pull 和 Push **push** 和 **pull** 是 Click 里 elements 直接的两种连接方式。 Push 是从源 Element 到下游的元素，通过事件触发，比如有包到达。在 Push 连接方式里，上游的元素会提交一个包到下游的元素；Pull 是从目的地元素到上游元素。在 Pull 的连接方式里，是下游的元素发送包请求到上游的元素。当输出接口准备好发包时，会开始一个 pull 过程：该过程会一直向后遍历图直到遇到某个元素‘吐出’一个包。**包调度**在 Click 里，调度器就是一个有着多个输入端口，一个输出端口的 pull element， 并通过一定的调度策略来决定从这多个 input 进来的包应该如何共享这个 Output。 在论文里提到 click 实现了两个调度器：Round-Robin Sched 是对 input 进行轮询；Priority Scheduler 是每次都从第一个 input 开始 pull packets。**Dropping Policies**:当包数目超过配置的最大长度时，这些包都会被扔掉。Dropping 策略：1）RED：Random Early Detection。该 Element 以下游的最近的队列长度作为 Dropping 的依据 2),RED over multiple queues：如果 RED 的下游有多个队列，那就将这些队列的长度都加起来作为 Dropping 依据 3）weight RED：每个包根据它的优先级有不同的 Drop 的概率。//**Graph Query**： Graph Query 和 Graph Analytics 主要关注的是不同的 workloader。GA 访问整个数据，去分析整个图上面的特征。GQ 解决的是具体问题，访问的是一小部分相关的数据，然后做计算。GA 是一个个的去做任务，因为时间很长，并行反而会抢占资源，CPU 还要互相协调，只要每个任务做的时候资源全用上，一个个做没关系；GQ 每个都是小的任务，因此并行。**现有 GQ 问题**：先前的在大数据集上经历高的查询延迟，大多数先前的设计具有较差的资源利用率，每个查询被顺序地处理。使用 triple 存储和 triple join 方法。存在的问题：使用三元组存储过度依赖 Join，特别是分布式环境下的 merge/hash join 操作；scan-join 操作会产生大量的中间冗余结果；现有的工作使用一些方法可以加速 join 操作，但是索引会导致大量的内存开销。使用 Graph store 和 Graph exploration，存在的问题：之前的工作表明，最后一步 join 相应的子查询结果会造成一个潜在的性能瓶颈。特别是查询那些存在环的语句，或者有很大的中间结果的情况下。**Wukong：Graph-based Model**：Wukong 最大的不同在索引的存储方式。之前的基于设计都是用独立的索引数据结构来存索引，Wukong 把索引同样当做基本的数据结构（点和边）来存储。并且会考虑分区来存储这些索引。Wukong 中有两种不同类型的索引结构，分别是 Predicate Index 和 Type Index 索引。Wukong 提出了谓词索引（PI）来维护所有使用其特定谓词标记的主体和对象入边和出边。索引顶点本质上充当从谓词到相应的主体或对象的倒排索引。Wukong 还提出了一种 TI 方便查询一个 Subject 属于的类型。与先前方法（使用单独的数据结构管理索引）不同，Wukong 将索引作为 RDF 图的基本部分（顶点和边）处理，同时还考虑了这些索引的分割和存储。这样做有两个好处，第一点就是在进行图上的遍历或者搜索的时候可以直接从索引的节点开始，不用做额外的操作。第二点是这样使得索引的分布式存储变得非常简单，复用了正常的数据的存储方式。**Predicate-based KV Store**：Wukong 使用了一种基于谓词的键值存储，在这样的结构中，相比原来的存储方式尽管会产生很多的冗余，但这种细粒度的存储使得搜索速率高效许多。**What is full-history pruning and what's the difference compared with the prior pruning approach? Why can Wukong adopt full-history pruning?** Full-history 就是说所有的历史记录都会被记录下来。在探索查询的每一个阶段中，通过 RDMA READ 读取其他机器上的数据，进行裁剪。裁剪那些没有必要的冗余数据。之前是只记录一次的。之所以可以这样做是因为一方面 RDF 的查询都不会有太多步，而且 RDMA 在低于 2K bytes 的时候性能都是差不多的，所以 Wukong 可以这样做。//**Bug**：**How does STACK identify unstable code?** STACK 有一个假设，是关于 Reach(e) 和 Undef(e) 的。STACK 会在 Assumption Δ 被允许和不允许的情况下分别模拟编译。先模拟假设不成立的情况进行一次编译；模拟假设成立的情况进行一次编译；查看前两步的执行结果有没有区别，有区别的地方就是 unstable code。