Index：

1. 两种基本的索引：Ordered v.s. Hashed
2. Ordered Indices：
   1. Primary v.s. Secondary：前者是保持文件原序的索引（后者自然就是乱序索引）
   2. Dense v.s. Sparse
   3. 多级索引：为了解决内存不够的问题（如同多级页表）
   4. 插入和删除的话，根据b, c两个维度自行脑补
   5. Secondary必须是dense，Sparse必须是Primary
3. Index增加查找效率和update的overhead
4. 然后就是一坨B+-Tree的内容，基本操作就不说了
   1. Index Strings：可变长，解决方案，对每个串找最长前缀使得能够自己能够被区分，或者存定长的公共前缀
   2. Low fanout ?
5. B-Tree：每个key在整个树上只出现一次，因此非叶结点增加额外的一个槽来指向文件地址（不大用）
6. 索引多个Attributes：注意目标文件是否连续存放
7. 静态Hash：数据结构课上都讲过了
   1. 不足：bucket数量过大或过小都会导致不愿意看到的情况，而导致bucket数量不和谐有两个原因：数据太skew和Hash函数战斗力只有5
   2. 解决方法：定期重新组织一下，或者采用动态Hash
8. 动态Hash：用i表示前缀长度，i随着数据的大小增加收缩，每一个bucket的前缀长度不等
   1. 查找：脑补
   2. 插入：先查找，有位置就放，没位置就分裂
   3. 分裂：
9. Bitmap：对于每一个Attribute的每一个可能的值Value，一个长度等于Tuple数的01串表示哪些Tuple在该Attribute上值为Value。查询时的Not、And和Or就分别对应位Not、And和Or。

Transaction：

1. 要求ACID：
   1. Atomicity：要么不做，要么一做到底；
   2. Consistency：分身的动作要看起来一致
   3. Isolation：不同的Transaction之间必须互不感知
   4. Durability：事情做完要有可持续的效果
2. 几个状态：Active、部分Committed、Committed、Failure、Abort
3. Transaction调度：
   1. Serializability：运行结果跟Serial运行结果一致
   2. 冲突指令：针对同一个数据的RAW、WAR、WAW
   3. 冲突等价：交换不冲突指令，两个调度的结果一致（Conflict Serializability可想而知）
   4. Test：看Precedence Graph是否无环，里面的边都是RAW、WAR、WAW的德行
   5. View等价：如果两个Transaction是生产消费关系，那么消费者看到必须是同样生产者生产出的非变质产品
   6. Test：这玩意儿NP
   7. View Serializability & ! Conflict Serializability 🡪 blind writes
4. 可恢复Transaction：消费者的commit命令在生产者的commit命令之后
5. 一次rollback可能造成连锁式反应
6. Cascadeless调度：消费者的read命令在生产者的commit命令之前

并发控制：

1. 锁机制：
   1. Exclusive (write, X) Lock v.s. Shared (read, S) Lock（哪些操作在什么情况下合法，稍微想一想即可）
   2. 同样的，有饥饿和死锁的问题
   3. 2PC：第一个Phase，大家可以获取和升级，第二个Phase再降级和释放，禁止越权行为。这玩意儿不能防止死锁
   4. 2PC严格版：X锁只能在commit之后释放
   5. 2PC限制版：X和S锁都只能在commit之后释放
   6. Tree Protocol：树上的节点是各个数据项。只允许X锁，第一把锁随便锁，之后的锁能上的前提是其爹被锁了。解锁随便解，但是解完不能重新上。这玩意儿能够防止死锁和保证Conflict Serializability
   7. 其他防止死锁的协议：wait-die，老的遇到情况等，新的遇到情况不等直接被撸下；wound-wait，老的直接把新的撸掉，新的等老的；timeout，等一定时间之后，动不了的话自动被撸下，可能饥饿
2. 锁的粒度（比如锁一个DB，还是area，还是文件，还是记录），精细的（在树的下层）更容易并发，锁起来累；粗糙的（在树的上层）跟难以并发，锁起来方便
3. 三种基于X和S的更加坑爹的锁：
   1. IS：底层那些只是S的锁
   2. IX：底层那些可以X可以S的锁
   3. SIX：某个节点自身上了IS，但是下层被上了IX
4. 时间戳机制：
   1. 如果一个老的进来读东西，发现有年轻的已经写掉了，那么老的被撸掉
   2. 反过来，年轻的读老的写的东西，正常继续
   3. 如果一个老的进来写东西，发现有年轻的已经读掉了，那么老的被撸掉
   4. 如果一个老的进来写东西，发现有年轻的已经写掉了，那么老的被撸掉
   5. 除了d和e，正常写
5. 时间戳机制的问题：一个被撸掉可能导致一连串被撸掉
   1. 方案1：所有写操作放到最后组成一个原子操作，一个人写的时候，其他人统统等着
   2. 方案2：等数据被commit之后再读
   3. 方案3：用commit依赖关系来保证可恢复性…
6. Thomas’ Write Rule：老的写一个过时的数据，直接撸掉这句写操作即可
7. Validation-Based Protocol：三个Phase
   1. Read & execution phase：把要写的东西先写在本地临时变量中
   2. Validation phase：有效性测试，是否这个东西被真正写出去之后会破坏serializability
   3. Write phase：能写则写，否则撸下
   4. 这三个phase可以穿插进行
   5. 遂有三个时间戳，分别是一个Transaction进入前两个phase的时间和完成第三个phase的时间
   6. 一个Transaction要通过验证，必须所有老的在自己验证（进入第二个phase）之前就完成
   7. 当冲突可能性比较小的时候，并发度高

Recovery：

1. Failure，可以是逻辑错误、系统错误或者硬盘错误
2. Volatile storage v.s. Nonvolatile storage，内存对硬盘之类
3. Stable Storage：这货是Nonvolatile的升级版，yy出来的。一种近似实现方式是把数据分若干个Copy
4. 注意input(X)、output(X)和read(X)、write(X)的区别
5. 日志
   1. 一条记录的样子：谁(T)把谁(X)从啥(V1)写成了啥(V2)
   2. Immediate修改 v.s. deferred修改：前者可以随时写（要留日志），后者commit之前只能本地写
   3. Committed被认为是其commit日志写到了Stable Storage里面，实际内容可能还在Buffer里面没到硬盘
6. 并发Transaction，大家用一个Buffer和Log
7. 如果一个人修改了某个数据，那么在他committed或者aborted之前，别人都不能动这块数据
8. 开始了未commit的撤销，开始了commit的重做
9. 老是重做似乎很花时间，于是用Checkpoint存档：在日志中加一条存档命令。存档的时候，把Buffer中的数据以及日志信息统统写到硬盘。并且一切Update活动暂停
10. 日志一般存在Buffer中。Write-Ahead Logging（WAL）：在数据被写到硬盘之前，相关的日志必须先写入硬盘
11. Buffer管理：
    1. No-force policy v.s. force policy：commit之后是否强制flush被更新的block
    2. Steal policy：被更新block可以随时flush，即使还未commit
    3. Flush的时候相关数据不得更新，可以要一把锁，这种持续时间很短的锁叫做latch
    4. Dual Paging(?)
12. Fuzzy Checkpointing：为了允许数据在过长的存档过程中可以被更新而设计，实际上只是区分了哪些数据已经被存档哪些还没有
13. 远程备份，primary端和备份端通过网络连接：
    1. One-safe：primary端日志写完立即commit；
    2. Two-very-safe：primary端和备份端日志都写完才commit
    3. Two-safe：折衷，两端都active的话，同Two-very-safe，只有primary端active的话，同One-safe

Architecture：

1. Centralized
   1. single用户（OS） v.s. multi用户（server）
   2. back-end v.s. front-end
   3. Transaction servers（也叫query server or SQL server） v.s. data servers：前者通常在relational database system中使用，后者通常在OO-based system中使用
2. Transaction server进程：Server p，Lock管理p，写数据p，写日志p，存档p，监控p的p
3. Shared memory里面是Buffer池，日志Buffer和Lock Table，query plan cache之类，大家都可以看到
4. Data servers：
   1. 船page v.s. 船item：前者可以做prefetch
   2. 锁可以和数据一起船过来船过去
   3. 数据在本地缓存（cache一致性问题）
   4. 锁缓存
5. 粗粒度并行 v.s. 细粒度并行：貌似就是多核v.s. 众核？
6. Speedup和Scaleup：一个是单纯核的增加，一个是核和问题规模一并增加。前者理想上是一条直线，后者理想上也是一条直线，只不过平行于横轴
7. 现实同理想有差距：
   1. 多进程的启动时间可能dominate计算时间
   2. 类似于数据依赖的干扰
   3. Load imbalance
8. 不同核之间通过BUS、MESH、HYPERCUBE连接（实际上就是1D、2D、3D的区别）
9. 不同并行架构：
   1. Shared Memory
   2. Shared Disk
   3. Shared Nothing
   4. Hierarchical：顶层Shared Nothing，下层可以是Shared Memory
   5. 前三者的瓶颈都在communication上，有shared就shared的那部分的access折腾，没shared就直接communication折腾
10. Homogeneous v.s. Heterogeneous：前者软件和schema在各个sites都一样，给人造成一个只有单一sites的错觉；后者可以不一样，于是能够整合各种现有的数据库
11. 分布式系统的trade-off：
    1. 共享数据（公有制）：一个site能否看到一些其他site的东西
    2. 自主性（私有制）：每一个site能够保留一些本地数据
    3. Availability v.s. redundancy：比如通过复制数据来使得更多的Transaction能够同时进行，代价是增加各种开销
12. 分布式系统通常使用2PC用来保证atomicity

分布式系统

1. 各个不同site的Transaction之间互相独立，但是可能access其他site的数据
2. 分布式存储：复制v.s.分段，也可以并用
3. Full replication：在所有site均有完整备份；Full redundant DB：在所有site都存了整个DB。
4. 复制的好处是availability和并行度高，并且减少数据传输；坏处是更新代价增加，并发控制代价增加。一种解决方案是选一个作为primary copy
5. 横分段v.s.竖分段：这里横竖是指切整张表的分割线的方向。目的都是为了相应的并行处理，以及有针对性地存放在最需要的地方
6. Transparency（透明度）：系统中关于数据的存储和管理，应该对用户隐藏多少
7. 数据项命名原则：
   1. 每一个数据项的名字必须全局唯一
   2. 能够有效定位一个数据项
   3. 能够透明的（？）改变一个数据项的位置
   4. 每一个site能够自主的新建一个项
   5. 中心化方法，用一个Name Server来折腾，前三条可以满足，第四条满足不能。同时有中心化方法的通病：单节点高负载带来性能瓶颈和Single Point Failure
   6. 非中心化方法，用Alias，并且在各个site存放Alias到真名的映射来解决透明度问题
8. 分布式Transaction：一个Transaction可以访问不同site的数据
   1. 每一个site有一个Transaction manager，用来维护日志和参与协调工作
   2. 每一个site还有一个Transaction coordinator，用来启动执行，发布subtransaction到相应的site，协调自身Transaction的终止（一个终止可能导致所有sites一连串commit和abort的连锁反应）