（第一课）

1. 数据库的存放格式是堆文件
2. 从层次型数据库转变为关系型数据库，基于资源管理器的树状结构的组织没办法科学化。基于层次结构的组织方式不能抽象成科学化的方法，而关系型可以使得数据库的组织简明而合理。将树状结构放入关系型数据库中
3. 在大数据量的情况下如何提高性能：表设计、模式设计，SQL优化等方法。环境为基于硬盘上的关系数据库（SIP是将数据库放入内存中，提高数据库数据分析的效率。在之后十年中，大数据量的分析都会放入memory中进行）
4. 数据库之间的比较，比如Oracle/MySQL/SQL Server/DB2，数据库的连接、组织形式是完全不同的。要提高数据库的性能最本质的是在某一个数据库上提高性能
5. 在基于web的应用中大量地使用非关系型数据库，放宽对一致性的要求而提高吞吐量。因为系统有并发的限制，所以单一请求的性能作为系统的性能指标是不可行的，因此引入吞吐量的概念。计算机系统中，最大的瓶颈是总线级的传输效率，但是在软件中瓶颈可能会在任何位置。一致性的要求会产生大量的回滚操作，从而降低吞吐量。NoSQL认为脏写、不一致性都是可以接受的，只要在一段时间内保证数据库的一致性即可（关系型数据库要求实时保证数据库的一致性）。Google、新浪微博等都是使用NoSQL实现的
6. 推荐教材见PDF
7. 关系代数：同样的代数表达式，所有的操作都是关系和操作符之间的操作，改变代数的形式而不改变最终的结果。每一个表达式都是一个SQL想要得到的结果集的表达式，where是选择操作，select是投影操作。表达式可以转换成多种具有相同结果的不同表达式，每一个表达式都被称为完成某一个结果的某一条路径，从中选出一条最省的路径（运算的时间、中间结果的存储空间），由查询优化器根据关系代数来寻找最快路径
8. 计算机对于表达式的变换是有约束的，所以可以找到一个有限的变换集，寻找一个性能最好的表达式从而形成一个查询。但是计算查询优化的耗时不能过长
9. 完整性约束：组件约束，引用约束
10. 视图：是一个SQL语句，当使用到SQL语句进行查询的时候才生成这个视图，在用的时候才会被执行。视图用在：
    1. 权限：当希望基本表不暴露给别人，所以使用视图来暴露给别人，此时视图可能引用到多张表而形成一个可以被其他应用所看到和使用的视图，从而保护基本表不被别人看到。与接口类似
    2. 重构：用在数据库重构中。完成了数据库重构中，而将原来的表做成完全一样的视图，此时内部的所有重构都不会影响到外部的使用（外部应用使用视图）。如果视图引用到大于4张表，此时应当考虑到性能的问题，但是应用不会看到视图的引用情况，所以使用视图可能会掩盖一些现象
11. 事务处理：需要有时间戳的概念（为了避免脏读），出现脏读需要回滚，但是Oracle不回滚而DB2和SQL Server都会回滚。Oracle称其为一致性读。NULL值就是不知道它是什么值的一个值，但是SQL Server认为空值是一个值。这些问题会导致数据库移植变得十分困难
12. 复杂事情：数据库迁移；数据库重构
13. SQL：SQL的复杂度取决于from后面包含的表数量（笛卡尔积的基数）。没有任何厂商支持完整的SQL99标准，而是只支持最基本的部分（query table/ query view/ select/ insert/ update/ delete）。每一个数据库的存储过程和触发器都不一样
14. 范式（使用范式来降低数据库设计产生的冗余，有冗余则导致维护数据库会带来更多的操作。有时可能会打破范式形成一定的冗余，为了提高数据库的性能，逆范式和反范式只有几种方式）
    1. 1范式（字段都是原子型单值，只被当成一个值来使用，这个值不做拆分进行验证）
    2. 3范式（没有函数依赖，身份证、性别、生日三个字段都存在的话，则不满足第三范式，因为这些字段之间有函数依赖）

（第二课）

1. 任何DBMS都可以支持三千个并发连接，超过三千个连接则需要使用其他方式来改善并发的性能。而理论上应用的并发数量无穷大
2. 并发修改针对的是数据库上的某一个独立资源。三千个并发连接可以并行使用同一个数据库，但是不能同时修改其中的一个资源，即对资源的修改是串行的（为了保证数据库的一致性）。这个资源称为原子资源，page/block是数据读取的最小单位。DBMS从硬盘中读取数据不是按记录读取的，而是按块（读取完整的、固定大小的数据块，当中包含多条记录）读取，而后修改其中一条记录
3. Oracle允许脏读，而NoSQL则不考虑数据的一致性
4. 性能在数据库中非常重要。性能可度量，它的可度量的性能指标是吞吐量。性能在设计之前需要被测试（SQL的写法、索引、文件存储方式、表结构的设计都可以影响数据库的性能，其中表结构的设计涉及到范式、反范式、逆范式）。数据库的设计结果中，逻辑设计是表结构，物理设计是索引和物理组织形式
5. 缺省的索引结构为B树索引，而oracle中的索引结构为位图索引。位图索引对于低频字段（可取值的范围小）可以大幅提高存取效率，当位图索引的索引键进行切换修改的时候（N->Y），切换过程中与N相关的字段都会被锁住
6. 每一个SQL的查询都有很多查询路径，但是不能遍历所有的路径，所以必须要设置一个timeout时间段
7. DBA的职责：系统数据备份，系统数据恢复（恢复备份的数据）。开发人员则需要做配置工作，比如表设计、触发器、视图、索引设计、缓冲区大小设计等。普遍认为，索引以下的更底层的任务由DBA完成
8. 对于树结构，oracle中使用with关键字实现递归，或者使用connect by start with来实现自顶向下或者自底向上的查询
9. 二维表就是关系，关系操作是表和表之间的操作。关系不包含重复的数据，但是DBMS中并未这样实现；关系的记录之间没有顺序，如果数据之间有次序要求，则不能被称为一张关系表，而记录的顺序往往会影响查询的效率
10. 数据库中不存在Yes/No或者True/False这样的字段
11. 业务的约束可以放在程序中检查，也可以放在数据库中
    1. 比如：使用check关键字定义数据的取值范围，对字段的有效性进行检查；而购买操作可能包含多条数据库操作，比如库存减一、配送单加一，可以把它们作为一个业务逻辑“购买”，也可以放在程序中操作。
    2. 如果数据库是一个中心级数据库，有多个应用对数据库访问，将业务逻辑放在数据库中可以使得该业务逻辑被多个应用共享，业务逻辑发生变化时只需要修改一个业务逻辑；而某一个字段的取值范围也应当放入数据库中限制
12. 当有多值字段时（比如一个产品有多个历史价格），则满足范式的做法是再使用一个新的数据表，当中存储产品ID和产品价格。这样做的话会导致需要查询多个数据表
13. 在架构层面上解决数据库性能问题：将耗时间的数据库操作放在不密集的环境当中。比如所有的银行跨行清算都是批处理模式，而这种清算都是在深夜进行
    1. 同步操作：存取金额都是同步操作，必须要实时处理的业务
    2. 异步操作：银行清算属于异步操作。异步处理方式处理很多不必要实时处理的业务，从而提高数据库性能
14. 处理数据库的方式
    1. 同步和异步
    2. 集中和分布式

（第四课）

1. 避免在select后使用\*和distinct，因为即使有重复行，也可以在程序中进行处理，但是如果直接用distinct处理可能会导致产生错误的结果
2. 普通查询的过程：先做from，再做select，再做where
3. distinct -> exists
   1. 如果通过不同的SQL写法来完成必须去掉重复行这一约束？使用exists关键字解决distinct的问题，需要用到嵌套查询。
   2. exists嵌套被称为关联嵌套，在子查询中需要用到外层插叙的表，所以子查询不能单独运行。一般而言是检查外部查询的每一行数据（针对外层查询的每一个记录），代入到子查询中作为子查询的条件。exists后的查询只要不为空，则条件值为true。
4. exists查询应用于某一条件在结果集中所占比例较小的情况，在这种情况下使用exists查询能大幅度提高查询效率，因为执行次数少（子查询找到第一个结果就可以返回）。如果某一条件所占比重较大，则可以使用关键字in的嵌套查询
   1. 外层查询所用到的字段的条件所对应的内层查询的字段一定要在表中加上索引，这样才能提高内部查询的执行效率
   2. 内部查询并不需要计算查询结果集，只判断是不是有满足内部查询的结果，换句话说，内部查询并不一定将查询执行完毕，可能在找到第一条满足条件的数据后就完成执行。故而，子查询的计算时间难以估算
   3. 使用exists暗示查询优化器，这个查询的查询计划是：内部查询的查询优化，内部查询的优化是随机方式，内部进行独立的优化；内部查询是单表查询，难以优化
5. exists查询的好处
   1. 内部查询优化完毕之后不会被完整执行
   2. 内部查询中的join关系与外部查询的表没有直接的叉乘关系，而是代入外部查询的结果值到内部查询中。这样做减少了一次叉乘的过程，但是导致内部查询要被代入多次
   3. 避免了distinct所带来的结果集错误的问题，因为它最主要的工作是对外部查询中的表格做选择，而不是对join后的表格做选择
6. exists和in的区别
   1. exists被优化一次后执行多次，但是每一次都不一定要执行完。由于在查询优化器中，优化的时间远远超过执行的时间，所以exists虽然需要执行多次，但是它的效率可能反而较高。执行exists要求外部条件所占比重小，这样能删掉大部分的外部查询结果，故而子查询的执行次数少
   2. in被优化一次后只需要执行一次，但是必须是完整的执行，然后再与外部查询的结果比较找到交集。如果外层查询的条件所占比例较大的话，则in的查询效率都会远远超过exists
7. 使用查询计划告诉DBMS的查询工作
8. 一个订单有多个状态的话，根据三范式的要求，必须将订单状态放在另一个表中，将订单号作为该表的外键，与状态一起作为该表的主键
9. 在订单和客户的例子中（PDF27页），orderstatus被使用多次，通过查询的写法减少它的遍历次数可以优化查询的过程。可以将两个子查询结合在一起，找到最后的状态并判断它是不是完成状态。
   1. 解决的方法是，将查询每一个订单的最后结果的子查询的结果关系表作为外部查询的from中所要引用到的表，则这样该查询只需要执行一次。
   2. 这种优化方法主要用于处理一张表被使用多次的情况（降低一张表的使用次数主要是在from后加入子查询，该子查询针对该使用多次的表，而且往往需要用到group by以及相应的aggregation操作）
10. 查询优化的两种方式
    1. 当需要使用多次join的情况下，考虑使用exists和in来进行查询优化
    2. 降低连接表的个数（降维），当有一张表被多次访问时，要考虑合并其中的查询

（第六课）

1. 外键和索引
   1. 如果没有外键和引用的话，一次修改会导致多次修改
   2. 大系统普遍取消外键的关联，取消参照完整性（降低在更新主表时候的过多引用）是提高数据库性能的一个措施。如果有大量的外键关联，则做一次主表查询可能会导致连接多个代码表
   3. 如果有外键的话，则需要对外键加上索引
2. 每个数据库都会有一个自动生成（递增）的序列号
   1. 自己解决：在插入一行数据的时候，需要先找到当前的最大值，然后对当前值递增作为新数据的值。可以使用一个表，记录其他表的当前序列号的最大值，每次更新后修改最大值。它带来的问题是串行化的插入，因为这个表需要被串行化的访问，所以每一次对该数据库的任意一张表插入新数据的动作都必须是串行的
   2. 为了解决上述的串行化访问的问题，必须采用系统所提供的自动递增序列号的方式
   3. 但是即便系统自动生成的序列号可以并发进行（导致插入数据可以并发进行），但是索引不能并发（如果索引是连续的，则一般会在一个块中），即便插入数据表的操作可以并发，但是索引仍旧需要串行化操作
   4. 解决上述问题的方式：反向键索引/逆向索引。当42、43、44三个记录同时插入时，可以插入到不同块中实现并发插入，但是插入索引还是串行化的（索引键连续，可能在同一块中）。可以将键逆向变成24、34、44，此时它们不再连续，就可能不在同一个块中，可能可以实现并发插入
   5. 使用上述方法，不仅在堆文件中可以实现并发插入（插入记录），同时也可以在基本文件中实现并发插入（插入索引）
   6. 质疑：毫无意义的系统自动生成键作为主键，是否有意义？（毫无意义的字段放到表中是否有意义？）
      1. 如果不使用自动生成键，可能不能保证主键的取值一定是唯一的（比如身份证都可能会重复）
      2. 不习惯使用非数值型的字段做表的主键
      3. 从索引结构的角度来看和对数值型索引的处理方式来看，字符串索引的处理速度远远小于对数值索引的处理速度
3. 没有使用索引的情况
   1. 谓词中没有使用索引的最前列，所有的B+树都是在索引的第一个键值上构建索引
   2. count(\*)在全表中算行数，不使用到索引。如果键有控制，count()操作仍旧算这些记录，但是索引中不记录空值（B树索引不对空值构建索引，子节点中都有值）
   3. 对于一个有索引的列做函数查询或者隐形函数查询（主要是时间和类型变化这两种隐形函数查询）
   4. 使用索引是否会带来效率的提高（查询优化器根据统计信息进行判断）
      1. 如果使用到索引反而会更慢。查询优化器未必会使用索引，使用前会先判断使用索引有没有价值，根据查询的统计信息判断是使用索引的评分高还是不使用索引的评分高
      2. 如果查询优化器发现当前的查询不该使用索引，所以它所制定的查询计划不包括所提供键的索引查询。查询优化器根据统计信息进行判断，如果统计信息是错误的，则查询优化器的判断也是错误的
   5. 使用索引的基本要求是索引可用，且索引对查询有帮助
   6. 访问索引的方式是点状访问，得出的结果是满足条件的所以快的编号。索引所做的工作是：通过索引找到所需要的记录；读取这个记录（通过堆文件来读取记录）
4. 数据库的物理实现
   1. 数据库的数据存在数据库所设计的文件中，数据文件在绝大多数情况下是堆文件。我们看到的是该文件的逻辑过程（二维表的读取），而不能看到它的物理过程（记录的存放、读取）
   2. 数据库表是存储方式的抽象过程，它屏蔽了所有的存储过程的物理细节。比较关键的两点是如何读取数据以及数据的组织过程
   3. 尽管物理结构与SQL没有直接关系，但是使用SQL的好坏却受物理结构的影响
5. 数据库存在大量的冲突
   1. 并发用户数量大的系统
      1. 紧凑：数据的紧凑程度和查询效率有正向相关关系。紧凑意味着一个块中能够存取更多的数据，如果有一定的顺序的话，数据的紧凑度直接影响查询效率。如果块中能存更多的记录，则一次查询所需要读取的块的数量会降低（全表遍历的效率会提高）
      2. 分散：从并发的角度来看，数据越紧凑则并发数越低。数据紧凑则块数少，则能够支持的并发数量少。（3000个并发是一个很大的并发概念）。为了提高并发数量，需要将数据存储的更加分散。所以需要在查询效率和并发数量中间取一个平衡点
      3. 任何提高查询效率的手段，都会带来某些性能的下降。整体提高数据库应用效率的措施，应当是提高了大部分事务效率而降低了小部分事务的效率。频繁事务效率提高的同时，非频繁事务的效率会下降。可能会使用批处理的方式来处理频繁事务，而使用实时处理的方式来处理非频繁事务
6. IOT索引组织表（根据索引结构存储基本文件数据，只有Oracle提供这种存储方式）
   1. 数据的组织不再是随机地向某一个块中插入，而是使用一个B+树索引结构，在主键索引中存储表中的所有数据（按照主键的顺序在索引中存储所有的数据）
   2. 优点：它是一个有顺序的存储方式，而不是随机的。针对一定范围的（有层次的范围）查询是非常高效的
   3. 缺点：更改文件的顺序会造成极大的更新代价，这种代价在随机文件中就不存在
   4. 本质：把一个数据库文件存成index的方式
7. 聚簇索引
   1. 比如有两个表staff和department，分别有10个块和3个块用于存储数据记录。clustered index的存储方式将这两个表中融合在一起存储，比如在一个块中存储department表中ID=1的记录，以及staff表中外键=1的所有记录。块中的存储没有顺序
   2. 优点：在查询中需要join的情况（select name from staff where dID = …），这样的存储方式会提高查询的效率
8. 数据库分组
   1. 原因：数据库出错需要备份，数据库使用redu、undo日志来进行备份和回复（rollback），备份过程只能祈祷不要出错，如果出错，则整个备份活动失败。为了防止备份出错，则可以将记录分组，存在不同的物理设备上，这样即使出错，也只是一个分组的备份出错
   2. sybase最早提出partition，希望能从管理数据库的角度降低DBA的心理压力。partition提供了一种数据管理的方法
   3. 分组可以提高并发性
   4. 举例：当物理并发满足并发性要求后，会考虑提高数据密集度来提高查询效率，所采用的方式是滑动窗口
      1. 如将存储分成13个区，前12个区存当前12个月的数据，最后一个区存历史数据；当时间进入到下一个月后，则为新的月新开一个区，而将倒数第二个区并入历史数据中
      2. 使用这种方式，当月的数据存入独立的空间中，必定会提高查询效率，但是降低了并发大小的可能性，所以可以将当月数据放入并发性较高的区，而将历史数据方式并发性较低的区
      3. 并发时，当前月所在的区的并发率最高，资源竞争最激烈。数据集中提高查询效率，必然导致并发性下降，所以应当从区的内部提高并发性（比如提供很多存储块）
      4. 以上均为物理存储，表面来讲仍然是一个逻辑表
   5. 分区方式
      1. 根据记录的值，将原来逻辑表中的数据按照行分到不同的分区中
      2. 根据字段分区，将不同的列存入不同的分区中，重点照顾频繁访问到的列
      3. 范围分区、列表分区，希望在查询过程中让数据更加具体，而哈希分区希望在更大程度上提高并发性

（第七课）

1. 分区
   1. 范围分区：根据字段的值把记录分散在不同的物理存储空间。分区最核心的目标是为了方便管理
   2. 滑动窗口：按照时间分区
      1. 可以使得最常使用到的数据被聚集在一起，这样可以提高检索效率。绝大部分银行系统都是针对最近的数据加以访问
      2. 可以把最需要的数据（当月、当前的数据）放到最快的物理设备当中，通过物理的部署状况来提高查询效率
   3. 哈希分区：把数据按照某种不同的比例平均分在不同的物理区域，将数据分布的话出现错误后回滚的压力较小
   4. 分区缺点：从本质上来说降低了并发的个数，但是在数据量非常庞大的情况下，降低并发所带来的缺陷远远小于分区所提高的性能
      1. 并发和数据密集的冲突是一对不能被解决的矛盾，或者读和写是一对不能被解决的矛盾
      2. 应当找到频繁发生的事情，提高该事情的性能，并降低该事情所消耗的资源
      3. 由于强制的数据聚合可能会导致其他数据的分散，所以不同的查询请求也可能会形成性能上的矛盾
   5. 分区注意：希望分区能够平均分布，均衡；避免更新操作频繁地移动分区，因为分区的移动会消耗大量的资源（以时间做partition key的话，数据不会轻易移动）
2. 事务的三种状态：等待W；正在被处理P；结束D
   1. 如果某一个事务正在被处理，则它所涉及的数据表（分区）需要被锁起来，如果大量的事务同时涉及同一个分区，则会导致并发数量的下降
   2. 通过服务类型进行分区：应当把数据均衡地分布到不同的分区中，比如T1处理第一个分区，T2处理第二个分区，则这两个事务可以完全地并发，不会导致并发数量下降
   3. 通过状态进行分区：分成三个区域，W区域存储等待的数据，P区域存储正在被处理的数据，D区域存储已经结束的数据。将所有等待的数据放在同一个区域中，则减少了轮询所消耗的时间（不这样做的话，则需要轮询寻找第一个需要被处理的数据）。每一个记录在移动分区的时候（状态变化），不存在并发的冲突（不需要将分区锁起来）。这种方案的最大特征：通过记录在分区之间的不断移动，可以降低某一个分区对同一个资源/记录的竞争压力
3. 除了堆文件，所有的物理环境都能带来复杂性；存储方式的变化会导致性能的下降。解决这些问题的方法只有测试
4. 层次机构
   1. 关系型数据库无法直观地解决层次式问题，所以需要一种变换。关系型数据表中的字段之间是平级且等价的，没有层次关系
   2. 层次式结构（树状结构）不能直接放在关系型数据库中，需要变换一种形式。树状结构中，节点之间有父子关系，存在兄弟节点，有根节点也有子节点
   3. 简单树状结构要求：一个节点只有一个父节点；所有的节点类型都是一样的。如果存在多个父节点，则为物料单BOM
   4. 简单树状结构的例子：档案位置（楼->层->房间->橱->柜），找到档案是一个自顶向下的遍历过程；风险分析（解决对冲基金的风险问题，一个基金可能包含多种基金、股票，甚至有可能包含平级的基金。计算一个基金的风险，要计算这个基金的组成部分的加权风险）
5. 将简单树状结构存储到数据库中（建模）
   1. 邻接模型，使用ID来存储每一个节点，使用PID（父节点ID）来存储树状结构。这种方式假设兄弟节点无序。这种方式不支持多父节点，即物料单结构。在单父节点的情况下，邻接模型是最重要的存储方式
   2. 物化路径模型，PathID（1，1.1，1.2，1.1.1，1.2.1，…），使用层次式的路径明确地标识出来。它允许节点之间有顺序（因为路径的标识有顺序），比如家族族谱
   3. 嵌套集合模型，每一个节点都有一个左编号和右编号
      1. 某一节点（A节点）的所有节点，它的左右编号都在A节点的左右编号之间
      2. 按照深度遍历进行左编号，遍历到叶节点后再原路返回进行右编号，而后再继续深度遍历进行左编号
      3. 按照此遍历方式，根节点的左编号是编号1，右编号是最后一个编号N，而它的第一个直接子节点的左编号是2，它的最后一个直接子节点的右编号是N-1
      4. 当发生更新之后，整个结构的编号都要发生变化，所以增加节点和删除节点会带来很多开销，所以较少使用
      5. 使用这个方法，找到某一个节点的子节点很方便，所有左编号和右编号在这个节点的左右编号之间的节点，都是它的子节点

（第八课）

1. 作业2
   1. A -> B有一个班次G7001，中间途径C、D。现在可能购买A->C的车票，C->B的车票，这种情况下怎么在数据库表结构中间解决这个问题。即实现区间购买票的记录，以及对剩余票的检索
   2. 抽象和实际的问题。G7001是一个抽象的概念，真正跑的车次是今天早晨8：00出发的G7001。对于这样的火车编号来说，它一天最多只开一班，但是有可能隔天发车，比如每周2、4、6、1发车（或者每隔1/2天发车；或者每隔两天，同时逢10号开），需要判断这班火车今天是否开车。探讨如何将这个内容记录在数据库中（可以存储规则，也可以存储当前时刻表。不好使用存储、比对字符串的方式）
   3. 火车上有一组对应的班组人员（班组成员的数量会有规则，即为潜在的约束），包括一个司机、一个列车长、若干列车员、一个车警。班组人员可能会变动，而且每天的G7001不可能都是同一个班组，所以需要一个排班表。可能会查询某一天的某一个班次的班组人员，也可能会统计某个车警一周工作的天数
   4. 每一列真实运行的班次都有可能晚点，比如在C点晚点，如何计算实际到达B点的时间。有些情况下以上内容可以记录在数据库中，有些情况下可以即时计算
   5. A->B->C->D，要记录在B->C之间有哪些具体的座位卖出去了，哪些具体的座位没有卖出去
2. 作业要求（作业1和作业2都要如此）
   1. 以文本提交作业结果
   2. 提供一个数据库模式（即数据库表结构），最简单的表结构T(column1, column2, column3, column4…)
   3. 提供一个满足查询的SQL语句
   4. 可以提供多种数据库设计，并且运行在不同的平台之上，并说明它们的不同
   5. 以create table的方式构建表结构，使用关键字check来设置表结构的约束
3. 例子：普法战争
   1. 自顶向下查询，查询Vandamme所管辖的所有部队
      1. 邻接模型
         1. 查找某一个节点的所有子节点，这是一个典型的递归过程。首先需要找到这个节点的所有直接子节点，查出来之后放在一个临时表中；然后遍历临时表，找到表里每一个节点的直接子节点，继续放入临时表中。直到没有新的节点放入临时表中，则递归结束，临时表存放了一开始的节点的所有子节点
         2. 实现递归过程，Oracle提供的关键字connect by。而SQL标准实现递归的关键字是with，DB2最早使用with。with是一个基于关系操作的递归过程，而connect by完全地去关系化，是一个基于过程的查询方式，所以它的查询效率非常高
      2. 对于物化路径的存储方式，子节点的前缀就是父节点的路径，找到某个节点的子节点，只需要匹配它们的前缀即可。但是物化路径的效率不如邻接模型的效率高，因为物化路径最关键的部分是字符串比较，它最大的问题是拆分字符串
         1. 可以使用01010102来代替路径1.1.1.2，计算它的深度就是路径字符串长度除以二，即strlen(“01010102”)/2
         2. 使用一个系统自动生成的ID，而不是使用物化的路径作为字段ID
      3. 嵌套集合模型，子节点的left\_num和right\_num都在根节点的left\_num和right\_num的范围之内。它无法直接获得节点的层次
   2. 自底向上查询，查询某一个子节点的所有祖先节点
      1. 关系操作可以合并相同的父节点，很容易地找到所有的祖先结果集，但是不能做到很好的层次排序
      2. MySQL有一个机制，可以解决字串索引的问题，substring index
      3. 因为自底向上需要遍历多个节点，而自顶向下只需要遍历一个节点，所以物化路径的方式，自底向上的查询效率远远低于自顶向下的查询效率
      4. 对于嵌套集合模型，自顶向下和自底向上的查询效率相同，它们的差异仅仅体现在自底向上查询的排序过程，这个排序比较耗时间
   3. 树状结构的聚合计算
      1. 在部队的树状结构中，叶节点往往存储一个数值，说明了一个团的士兵人数。可以将兄弟叶节点聚合，得到它们的父节点的数值，即一个旅的士兵人数
      2. 使用oracle的connect by关键字进行递归。connect by ...start with方式，只能获取自顶向下和自底向上的最终结果，而不保存中间的递归过程，所以每一级别都需要做一次自底向上的操作，效率降低

（第九课）

1. 要封装数据库，JDBC仍然不方便，因为存在关系和关系对象之间的mismatch
   1. 一个对象不可能存在一张关系表中，如果这样存储，则应当为对象数据库而不是关系数据库。一个对象可能存在多张表中
   2. ORM，封装数据访问的过程。数据封装层主要用来屏蔽业务逻辑，屏蔽OO与关系之间的差异性，设置关系表和对象之间的关系
2. 数据库封装的优点
   1. 应用程序员能够专注于（应用的）业务问题
   2. 可以实现公共的面向的对象的业务规则
   3. 仍旧能够利用数据库的特性提高应用程序的性能

（第十课）

1. 数据库方法学
   1. Database planning，数据库设计的计划
   2. System definition，最主要的是定义系统的边界
   3. Requirements collection and analysis，需求分析和收集，收集表单和行为
   4. Database design，逻辑设计（与具体的数据库无关）和物理设计（与具体的数据库有关）
      1. 比如在构建索引的时候，就从逻辑数据库设计进入物理数据库设计。
      2. 数据库系统与文件系统的差别，在于并发修改（而不是并发读）。数据库系统是处于并发修改的最好载体；数据量大并不足以使文件系统转变成数据库，而如果数据关系复杂的话，则需要将信息存入数据库中；必须要安全存储的信息需要存到数据库中。正常情况下，文件系统的效率远远高于数据库的效率
   5. DBMS selection，选择哪一个具体数据库。如果有遗留数据库，则选择遗留的数据库系统；或者某个数据库系统有其他数据库没有的特性，比如Oracle
   6. Application design
   7. Prototyping
   8. Implementation
   9. Data conversion and loading，数据转换指的是数据库的迁移，loading指的是对遗留数据库的重构
   10. Testing
   11. Operational maintenance
2. 逻辑设计（ER图）。ER图最大的问题是，其在UML中没有标准
   1. MDA——模型驱动架构。画完UML图后，程序结构和逻辑就可以生成
   2. 强实体和弱实体。强实体是可以由自己的主键标识自己的实体，比如课程有课程号，学生有学号，都是强实体。如果选课是由课程号和学好来限制的话，则其没有标识自己的主键，属于弱实体；如果选课有一个选课流水号来标识自己的话，则它也算是强实体
   3. 关于人与公司的雇佣关系，可以有两种设计表达：（1）人与公司是多对多的关系，它们产生一个雇佣关系，雇佣关系中有雇佣的属性；（2）人与公司没有直接的关系，人可以有多份合同说明他所在的公司以及相关的雇佣属性，公司也可以有多份合同说明它雇佣的员工。第一种中，雇佣关系是弱实体，一个员工和一个公司只可能产生一个关联实例；第二种中，合同是强实体，一个员工和一个公司可以产生多个关联实例（因为合同由合同号来标识）
   4. 注意ER模型中的一对一、一对多、多对一、多对多的关系，以及一个实体是否强制参与某个关系
      1. 1:1，一对一全部参与可以放在一张表中，并且不会有空值
      2. 一对一部分参与必须分在两张表中，如果放在一张表中则会出现很多空值。如果是一个全部参与一个部分参与的话，则将全部参与的实体的主键放在部分参与的表中作为外键（以此来描述他们的对应关系）；如果两个都是部分参与的话，则参与更多的一方的主键放入参与更少的一方的表中作为外键，这样能产生较少的空值
      3. 多对多会产生一个关系表，将相关的两个实体的主键作为复合外键，作为关系实体的主键。这样关系产生的实体是一个弱实体
   5. 强制性地表达表中的完整性约束，no action，cascade，set null，set default，no check。这些关键字说明了主键被删除后，外键应该怎么办
   6. 数据库泛化的问题。比如设置一个Staff实体为父实体，它有manager、salesPersonnel、secretary三个子实体，它们共享父实体中的属性。这种结构会产生以下的约束
      1. 参与约束，participation constraint，是不是所有的实例都存在于某一个子实体中，父实体只是一个抽象的概念。即是不是所有的员工都属于某一个职位，如果有一个员工不是子实体中的一员，则他为父实体的实例，这属于部分参与
      2. disjoint constraint，互斥约束。一个实例是不是只属于一个子实体。如果一个人，他既是manager又是sales，则manager和sales这两个子实体是overlap的，不是disjoint的
   7. 数据库泛化的约束产生的结果
      1. mandatory + nondisjoint，完全参与且不互斥，则所有的父实体+子实体都使用一张表，虽然可能会因为某个实例不属于某一个子实体而产生空值，但是产生的空值比较少，所以可以
      2. optional + nondisjoint，部分参与且不互斥，则将父实体做成单独的一张表，所有的子实体做成一张表
      3. mandatory + disjoint，完全参与，但是互斥，则父实体产生一个主表，子实体分别分成一个表
      4. optional + disjoint，部分参与，但是互斥，则父实体产生一个主表，子实体分别分成一个表。只要是disjoint的情况，都要将子实体拆分成独立的子表
   8. 逆范式/反范式（打破三范式的规则来生成冗余数据）
      1. 合并1：1关系
      2. 部分合并1：\*关系，不复制外关键字而复制常用的字段
         1. 比如staff和department之间有一对多的关系，将department中常被连接查询的字段复制并迁移到staff表中（比如department的名字是一般需要查询到的，则将名字放入staff表中，这样查询staff所属部门的名字时就不需要连接两个表了），降低连接查询的数量
         2. 使用“一致性控制”模式来保证两个表中冗余字段的一致性，这种控制放在UI界面中进行，使用combo list或者check box进行选择，而不是获取用户的输入
      3. 在1：\*关系中复制外关键字
      4. 在\*：\*关系中复制属性
         1. 比如多对多关系的orders、articles的关系，有一个关系表中保存了oid和aid。如果要知道某个订单中某个产品的名字和价格的话，则需要三表连接
         2. 在现今的电子商务系统中，article的某些属性会直接复制到关系表中，即关系表中除了oid和aid之后，还包括price（price现在已经是缺省放入关系表的字段）、article name、数量
   9. 为了避免查询和更新这两个不可调和的矛盾，可以将更新和查询放在两张表中，从工作表提取出查询表，专门用于查询。这个方法演化成了数据仓库。这个方法只适用于查询的实时性要求不高的情况
3. 物理设计

（第十一课）SAP + HANA

1. 数据以列的形式存在数据库中。当需要查询某几个列的数据时，只需要将这几个列中的数据移入内存之中

（第十二课）

1. 列数据库
   1. column family: 任何的column family都可以分布式地存储
      1. @ikirly/name:”Liujia”，在列数据库中，ikirly是关键字，列是name，value是Liujia。根据列进行排序
      2. @某微博流水号/File:“内容”/Reply:”回复”
      3. 在存单独信息（“个人信息”、“微博信息”）时候，不考虑其互相的关系。与关系数据库相比，这里体现在不存储外键
   2. super column: 将数据库中的column family联系在一起
      1. @微博流水号: /A/content /A/time /A/picture /B/content /B/time /C/content /C/time，说明了微博的内容、时间、图像以及回复的内容、时间以及转发的内容、时间的相关关系
   3. 所有的column family都是按顺序存储的，通过key联系在一起。列数据库可以将value压缩，处理的过程也不用解压缩，只需要在显示的时候解压缩即可
   4. 行数据在存储的时候，每一块都需要保留一块空间，因为每一次update都可能会使得块中数据所占空间增加；但是列数据库不能update数据，只能insert/delete，就比如微博中只能增加/删除微博，不能修改微博
   5. 列数据库最早用于做数据分析，但是由于有某种意义上的关系存在，列数据库不被认为是NoSQL，但是NoSQL由列数据库发展出来
2. 数据库封装的优点
   1. 仍旧能够利用数据库的特性提高应用程序的性能，在封装成对业务性能做处理，比如索引和并发
   2. 可以使得程序员更多地关注业务，而不是底层信息
3. 数据库封装成的架构
   1. 应用程序 -> 数据库封装层 -> 数据库，这里的数据库可以不是通用的
   2. 如果多种应用通过数据库封装成访问数据库的话，则数据库需要是通用的
   3. 如果有多个数据库，在多个数据库之上又有多个数据库封装层（一个封装层可以封装多个数据库），则可以有多个应用访问这多个封装层。在这种情况下，数据库封装层可以屏蔽数据库的实现细节
4. 实现策略
   1. 蛮力方式
      1. 不是一种数据库封装策略，而是手动进行一些封装方式，这是没有数据库封装层时最常见的方式
      2. 优点：简单而快捷，是最直接的方式，能够支持糟糕的数据库设计
      3. 缺点：对象和数据耦合，一旦出现问题便是直接的问题，而且方案数据库的重构，且访问代码难以重用；应用程序开发者需要懂SQL
   2. 数据访问对象DAO
      1. 封装了业务对象所需要的数据库访问逻辑，典型方式是每一个业务对象都要有一个数据访问对象
      2. 使用DAO，应用程序访问DAO层，而后DAO层根据访问情况生成SQL，并用该SQL在数据库中执行，最后将返回的结果返回给应用程序
      3. DAO需要这一层自己拼成SQL语句
      4. 优点：高内聚且低耦合，由于低耦合所以易于数据库重构，可以屏蔽糟糕的数据库设计，且访问对象可以重用
   3. 持久化框架
      1. 框架通过访问Mapping信息，获取数据库的元数据，而后拼成SQL，使用该SQL访问数据库，最后返回结果
      2. 优点：低耦合，完全屏蔽了数据库的实现细节，有利于数据库重构
      3. 缺点：对性能有影响，而且无法支持过于复杂的映射
   4. 服务（Service）
      1. 由一个服务提供数据封装的功能。往往但是不限于用来封装对遗留功能和数据的访问
      2. 程序的一个请求发给Service，Service内部拼成SQL（也可能是调用其他部分拼SQL）并访问数据库，最后返回结果
      3. 用Service访问数据库，调用方不知道Service具体是怎么实现对数据库的访问的，但是DAO和持久化框架这两种方法中，都是封装层拼SQL并进行访问
      4. 优点：服务是完全平台无关的
      5. 缺点：远程调用都会导致远程的物理实现细节的屏蔽，从而导致无法进行统一的优化