（可以打印诶）

数据存储的level

权限分发问题

查询优化部分

基础语句格式/分号的切分

关系表达等式（决策树）

建立索引（hash/B+）

1. R图的设计与化简

范式规范与遵守

数据库与file system的区别

Entity/relationship

Database touched all aspects of our life

Database： large collection of integrated and persistant data（完整性与永久性）

DBMS特征：

1.Efficiency and scalability in data access（可拓展性与高效性）

2.reduced develop time

3.data independence

4.concurrent and recovery

5.data integrity and security（完整性与安全性）

Drawbacks of File system:

冗余：Data redundancy and inconsistency数据冗余严重并难以保证数据一致性。

低效：Difficulty in accessing data难以快速寻得数据

关系弱：Data isolation不同文件之间数据格式不统一

维护：Integrity problem较难去改变或新增数据（不同文件间有大量的冗余问题）

原子性：No atomicity of updates没有原子性操作的保证

并发弱：difficult to concurrent access by multiple users

安全性：security problems

Level of abstraction

Physical level：数据的实际存储；

Logical level：数据在数据库的存储以及数据在上层间的关系

View level：隐藏了数据种类，以及一些因为安全性而不展现的数据；（类似于create view之后可见的具体数据）

Physical independence：the ability to modify the physical schema without changing the logical schema.

Logical independence:protect application programs from changes in logical structure of data.（难以实现，logical 模式的改变往往需要改变应用程序。）

schema表示表的格式（属性结构），instance表示具体的数据。

Schema stands for type; instance stands for variable

Data model:

Conceptual design:entity-relationship model;（对应于实际情况的ER关系，有实体有关系）

Logical design:relational model;（数据库的展现形式，仅有一张张表+其对映的schema）

DDL：数据库的定义语言；（表模式，一致性约束consistency constraints--primary key and referential integrity）如create table

DML：数据库的控制与访问语言；(insert and delete and update)---query language

SQL belongs to nonprocedural DML（只说要什么，不说怎么获得）

DCL：grant，revoke等关键词组成，一般很少使用。

数据库的设计：

1. 需求分析；
2. Conceptual database design（E-R图设计）
3. Logical database design（关系图属性设计）
4. Schema refinement（表格式确定）
5. Physical database design(index, query, tuning......)（真实数据库的创立）
6. Initialize the database （载入数据，分发权限）

ACID：atomicity、consistence、isolation、durability。

Storage manager 包括：transaction manager/authorization and integrity manager/file manager/buffer manager.

Relationship is an association among several entities.

Relation referred to as a table.

Cartesian product笛卡尔积

Multivalued/composite attribute are not atomic.

NULL belongs to every domain.

Tuple表示行元素（一整行），attributes表示列属性。

Tuple无序不重复。

Super key：（超码）可以决定一张表，可能含有多余属性；

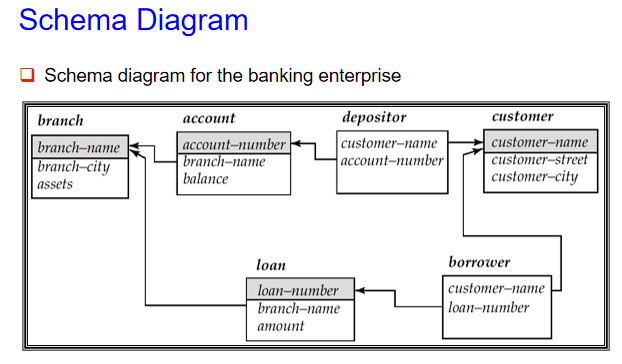
Candidate key：候选码，minimal super key，不含有多余的属性；

Primary key：主码，candidate key被用户显示定义;

Foreign key：外键，referencing relation参照关系（可以为null）/referenced relation被参照关系（主键所在关系表）

Schema diagram：

参照关系指向被参照关系；



Fundamental relation-algebra operation

Select选择；project 投影；union并；- 差；cartesian product笛卡尔积；rename重命名；

Deletion最小单位为tuple，不能仅删除部分数据；

R ← R - E

Insertion：

R ← R ∪ E （注意需要同属性列）

Update：

可以使用拓展版投影，在投影过程中进行计算。

具体见手写（第一个重点模块）：

DDL：create

DML：select from where

DCL：grant revoke

高级SQL语言

自己定义类型：create type 类型名 as 数据类型（感觉类似于define操作）；

也可以定义domain：create domain 域名 as 类型限定；

Domain更多的是一种限制，而type带有强制意义。

Blob(容量)---运输二进制文件,clob(容量)---运输字符文件

建立table时，可以在对映元素后加上 primary key/not null的关键词；

定义外键约束：foreign key (当前表的参照属性名) references 被参照表名（对映表的属性值）

外键交互的两个属性名可以不同

On delete cascade、On update cascade级联处理方式，直接跟在属性/约束定义过程中，逗号前。

一致性约束仅在事务结束时会check。

Assertion

Create assertion assertion\_名称 check（条件）；条件中往往是not exist/exist加上select语句。只有条件被满足时，才不会被拒绝；assertion需要时刻被满足。

//可能有截止符切换的问题，需要再注意一下

Trigger对一些sql操作的级联触发

Create trigger trigger\_名称 after/before[处理时间点] 操作 of 属性值 on 数据表

Referencing new row as nrow for each row[对new row重命名，对于每一插入的row而言/table也可以]

When 处理条件

Begin atomic

操作部分

End

New row 可以用inserted替代nrow，old row 可以用 deleted替代orow

Malicious 恶意的

Authorization授权

Read/insert/update/delete all kinds of authorization

同样也有index/resources（(同意建造新的关系表）/alteration/drop

访问视图也需要像访问表一般，需要一定的授权（视图的授权与表的授权区分）

授权过程基于数据表实现，不能对tuple进行授权；

Revoke权力时，需要层级收回；因而在grant权力时，要避免出现循环授权的情况；

Grant语句，进行授权操作：

Grant 权利列表 on table/view\_name to 授权用户列表（逗号分开） {with grant option}；

{}中内容表示可选，with grant option 表示获得权利的用户有权力继续分配权利；

授权用户可以使用户，也可以是public（所有有效用户均可以拥有此项权利），也可以是一个角色；

其中权利列表可以包括：insert/update/delete/reference（外键引用）/all

Role 角色：拥有特定权利的一组users

Create role role\_名称（建立新role）

Grant 权利名称 on 表 to role\_名（为role附加权利）

Grant role1\_名 to role2\_名（为role2\_名分配role1\_名具有的权利）

Revoke（收回分配的权利）：

Revoke 权利名称 on table\_name from 用户名称 {restrict/cascade}

Cascade表示级联收回分配的权利；restrict：如果需要级联收回权利，则revoke操作失败。

落在应用程序上的授权任务：

优势，应用程序进行授权，可以实现如tuple一般的细粒度操作；

缺点，授权过程可能分布于应用程序的各个部分；

维护，维护困难，需要阅读大量的应用程序代码，才能发现授权问题所在。

Audit Trails 审计部分

特别注意assertion的作用时间域是有关关系/属性的每一次修改，因而错误设置可能导致无法正常插入成对数据。

SQL被宿主语言调用

EXEC SQL END\_EXEC

SQL在宿主语言中定义变量

EXEC SQL BEGIN DECLARE SECTION:

定义变量（使用需要添加:）

EXEC SQL END DECLARE SECTION；

如果使用游标进行数据表的批量更新时：

需要在select语句后添加“for update of 属性对象”；

在使用游标时，需要在where处添加“current of 游标名称”

使用宿主语言变量时需要使用:作为作用域限定；

DECLARE c CURSOR FOR（游标），类似于迭代器，用于存放从数据库中获得的一个或多个tuple；

游标配合使用：

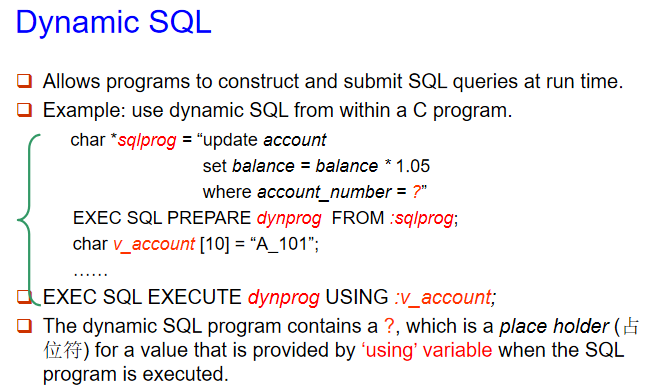
EXEC SQL OPEN c END\_EXEC

EXEC SQL FETCH c INTO 变量名 END\_EXEC（直接获得数据库一个tuple中对映的数据）

EXEC SQL CLOSE c END\_EXEC（最后关闭游标）

sqlca.sqlcode可以作为cursor结束的标识符；

Dynamic SQL：



Prepare关键词语句为指定的select命名语句“？”处填充一个临时变量；

后续使用 EXEC SQL EXECUTE 临时变量名 using 实际使用变量，进行临时变量的替换，从而构建完整的select语句。

SQL injection（SQL注入）

可以使用prepared statement进行防御，将SQL语句的主体写定，只是把用户的输出作为一种数据传入“？”的地方，从而实现SQL injection的防御。

E-R模型的建立：

ER图有entities与relationship构成，entities含有attributes，entity set 表示一个实体集，里面包含了多个同类实体。

实体集相当于一个tuple，表示数据表中的一行数据；attributes value表示一行数据中的一个具体的元素值。

Simple/composite attributes 简单与复合（一个属性包含多个不同的值）属性；

Single/multi-value单值与多值属性；

Derived 派生属性，可以从其他元素的数值推得；

Relationship set 不同实体间的表关系集合；如果是两个实体间的联系，便称为binary。

对于entity（tuple）而言，存在一对一、一对多、多对一、多对多的情况。

Relationship set的超码---参与relationship set中的两个entity set的主键拼合。

用图展示：

Rectangle 表示 entity set

Diamond（菱形） 表示 relationship set；

Ellipses（椭圆）表示entity set的属性；

Double ellipses 表示多值属性；

Dashed（虚线） ellipses 表示派生属性；

用表展示（感觉更加清晰）：

缩进表示复合属性的属性组成；

{}表示多值属性；

属性名+()表示派生属性；

必要时，relationship set也可以具有属性值。

基于不同的role，entity set可以借助relationship set发生自环。

映射基数约束（mapping cardinality constraints）：

（表）在line连接关系中，被箭头指向的表示一；由横线连接的表示多。

Total participation：全参与，double line，每一个entity必须参与relationship的组成。

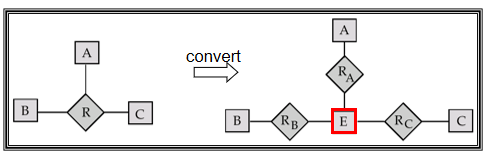
Partial participation：部分参与single line

用数字表达进行化简:

0..\*表示从0到无穷大；1..1表示至少1至多1。

Binary关系与多元（ternary）关系的转换：

多增添一个entity set 对象即可；

Weak entity set（没有主键） 基于 identifying entity set 存在；

双菱形表示weak关系；

Weak entity set 有discriminator（可用于区分所有weak entity的属性）的概念（用下虚线标明），weak entity set的主键由其依赖的strong entity set主键和weak entity set 的discriminator共同构成。

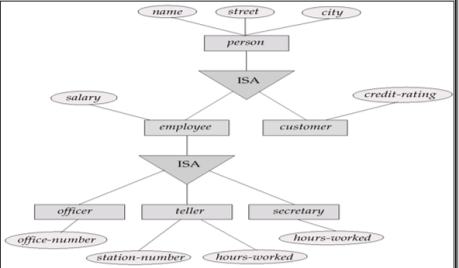
Weak 对 strong往往是多对一的关系。

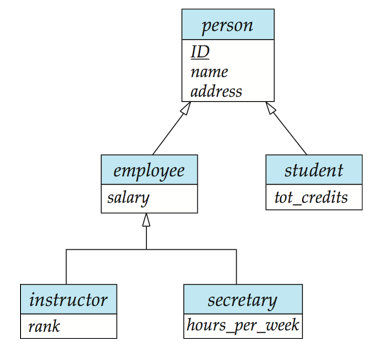
Specialization特殊化/具体化

Generalization：泛化/普遍化

在表现上，specialization与generalization的样式一致。

特殊化的图形式：



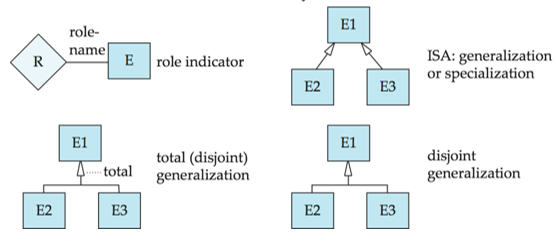
特殊化的表形式：

Condition-defined：根据具体的属性值划分；

User-defined：人为划分；

Disjoint（不相交）：只能属于一个sub entity。在图中用disjoint注明；表中线合并。

overlapping（可重叠）：可以属于多个sub entity。图中默认情况；在表中线分开。



Total（完全泛化）：图中用双线表示，所有high-level entity一定要属于一个sub；表中加total，好像默认disjoint诶。

Partial（部分泛化）：图中单线表示，没有硬性分类要求；

还可以存在复合entity set，必然会存在relationship set 连接relationship set的情况；

建立完E-R图之后，对于表格的处理：

对于多值属性，用一个额外的entity set记录（附加上详细的多值分类信息），加上原表使用的主键；

Relationship set的转换，将连接的entity set的主键拼合+自己relationship set的属性，便可以组件relationship对映的表；两entity set主键的联合成为新table的主键（多对多）；如果是many to one 的情况，来自many的entity set 主键作为新主键（一对一也仅选一方主码作为新表主键）；

在多对一的relationship下，如果many一方entity set属于全参与，则relationship set建立的新table，可以直接并入many一方中（增添一个one entity set 的主码属性即可）；one to one，可以任意挑选一方作为many的合并对象；

复合属性需要拆分，形成额外的多个属性；

Weak entity set加上strong entity set的主键后，成为一张额外的entity set。

Weak与strong entity set对映的relationship set is redundant，所以可以删去。

对于继承关系的处理：全泛化与部分泛化：

1. 建立具有所有属性信息的表格，分别对映于基类和各种派生类；（如果继承关系是total的，则基类甚至不需要记录，但可能存在大量的冗余信息（特别是overlapping的特化））
2. 建立含有所有属性的基类总表，每个子类单独建表，有基类主码与子类特有属性构成；（访问子类的数据时，需要并表查找）

范式：

第一范式；the domain of all attributes of R are atomic；对于关系数据库而言，所有表必须满足第一范式；

非原子性存储：查询复杂，存储冗余且复杂；（anomaly：异常）

多属性一张表：会存在数据冗余，在更新表时可能存在一致性问题，在插入删除时可能需要凭借null属性占位。

Decomposition：分解；lossless-join decomposition（无损连接分解）；

cd9185fad08456600c15853fa450672一种更快速的判断：R1,R2的交集，是否为R1,R2中任意一个的主键，是-->无损分解。

函数依赖：

A函数决定B，B函数依赖于A；

容易判别一个关系表是否满足给定的函数依赖;不易判别一个函数依赖是否在给定关系表上成立。不能简单推断。

Trivial 平凡的函数依赖；

如果A决定B，且B属于A；

如果A决定B，A决定C，则A决定BC；合并律

如果A决定BC，则A决定B，A决定C；分解律

如果A决定B，BC决定D，则AC决定D；伪传递律

闭包：F+--->所有函数依赖关系的集合；

函数依赖具有传递性、自反性（类似于trivial）；A决定B，则AC决定BC；

Sound：只显式显示的依赖；complete：所有的依赖关系（部分通过推导得出）；

闭包覆盖可以寻得超码A，一般还是使用图解更为迅速；

正则覆盖（canonical cover）：

在保留已有函数依赖的基础上，最大程度的进行化简。

1. 左侧的条件唯一；
2. 可以递推获得的可以省略；
3. 在已有函数依赖中，决定依赖的多余条件；
4. 对于依赖的复合结果，可以通过拆分来逐一判断重复性，进行删除更新；

那些多余的属性可以被称为extraneous attributes（无关属性）；

Decomposition类型：

Lossless-join decomposition:无损连接分解；（可以使用决定表格法进行check）

Dependency preservation：依赖保持（所有函数依赖关系均保持---正则覆盖相同即可，如果有冗余函数依赖存在也无所谓）；

BC范式：

1. ->B是平凡的，或A-->B时，A是对映关系下的super key（特别注意，要是super key 至少是主键的全部）；

BC范式保证了“无损连接”要求，但不保证“依赖保持”的分解需求；

BC范式的分解算法：含有主键的关系单独取出成表，并在关系总表中删除对映前式右边的元素；

第三范式：

允许冗余，但保证了分解过程的“依赖保持”与“无损分解”

满足第三范式的条件：

1. ->B是trivial的；

A是super key；

在A-->B的关系中，对于任意B - A中的元素，其是关系表R中的candidate key组成部分；

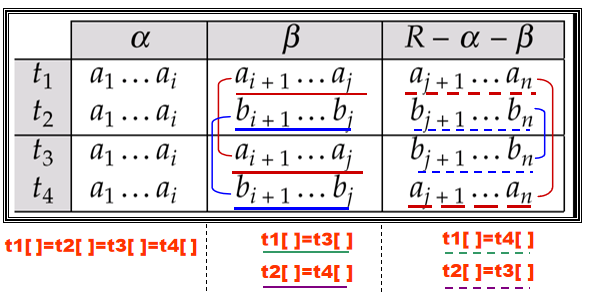
满足BCNF必定满足3NF；

3NF存在更多的冗余信息作为代价；

3NF的分解：直接一个关系一张表即可；最后需要满足一个新表中完全包含了原关系表的主键；

多值依赖与第四范式：

A-->-->B（如此意思）

如果A-->B，则A-->B。

第四范式，便是将BC范式中A-->B的部分改变为A-->-->B即可；只能保证分解是lossless-join的；第四范式的规范化与BC范式的规范化过程一致；////

---------------

Storage：

Primary storage：断点失忆，速度最快；易失性存储器：volatile storage；

Secondary storage：magnetic disks，断电可存储；非易失性存储：non-volatile storage;

Tertiary storage：脱机存储；

存储器的存储速度：

Cache：最快最贵，被系统硬件调用；

Main-memory：较快较小，但是提升速度很快；

Flash memory：断电内存保留；电可擦除器件，读速度同内存，但擦除/写速度较慢；多用于嵌入式设备；

Magnetic-disk：读取速度极慢，必须将数据读取至内存中才可以使用；

对于磁盘而言：

有sector（扇区，最小单元，一般为512Bytes），platter（盘面），read-write head（读写头）；

读写时候会进行校验，若信息读取错误，则会进行对映操作；坏扇区需要进行重映射；为减少控制器的负载，同一时间往往只有一个disk进行操作；

Access time = seek time（寻道时间） + rotational latency（旋转等待时间）；

Data-transfer rate：数据传输的速率；（不同track将存在区别）

Mean time to failure（MTTF 平均故障时间），若存在n块相同的盘，则平均无故障时间会除以相应的n；

Block是a contiguous sequence of sectors（一连串连续的扇区）；

为最小化磁盘臂的移动--->使用电梯算法（elevator algorithm），单向移动磁盘臂直到一端；

File organization：相同文件放在一起；

使用 nonvolatile write buffers 可以加速disk数据的写回--->重排后也可以使数据的写回会有更少的disk arm移动；（先等待，同一写回）

Log disk 记录修改量；

Raid level 0.没有任何校验盘；1.有同数量的校验盘；2.ECC需要三个额外盘；3.bit校验，只有一个多余盘（bit存贮--相当于拆块了）；4.bit校验改成block校验；5.没有特殊的校验盘，校验结果分布于各个盘；6.类似于5，但对于每一块盘都有一个备份；

一般只用level1/5,1花费更多的代价且提供更好的写表现。

Latent failures成功写入的数据被销毁了；

Data scrubbing 不停地scan修补latent failures的数据；

Hot swapping 在系统运行的同时进行disk的替换。

Block and buffer，buffer-replacement policies: LRU/MRU;

LRU（least recently）:最近最少使用策略；

MRU（most recently）：最近最常用策略；

Files--records--fields三个层级；

Fixes-length records--三种删除策略（全部平移，唯一平移，使用link记录下一个record）；

free-list，使用类似于list的方式连接；

使用定长方案时会存在冗余的情况，可以使用anchor block（锚块）与overflow block（溢出块）的策略消去冗余（使用指针指向另一张表）；

Variable-length records

Null value被bitmap标识；

Slotted page structure：需要有header包含总记录数量，空闲指针指向区域，每一record的大小与位置；

从后向前存储数据；索引不需要直接指向record；需要数据类型加数据的形式吧。

\*\*文件的存储顺序

Heap file：无需存放，哪里有空放哪里；

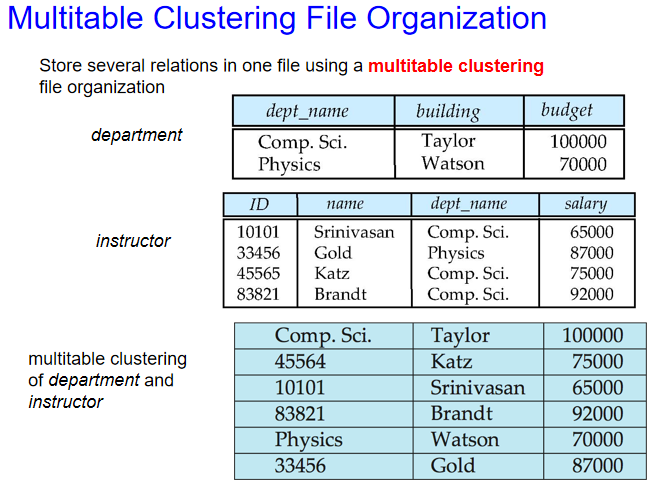
Sequential file：顺序文件（基于search key排列）；特别对于插入，需要指针进行导向，因而有文件定期重排的需求；

Hashing file：散列文件，使用散列函数进行存储；

Clustering file：不同relations（表）被存放于同一file中；

--------------------------

Data dictionary storage：用额外的数据管理数据的存储；

+++

面向列的存储，好奇特；（column-oriented storage）

优点：减少IO读取时间，改变cache表现，矢量处理；

缺点：提升删除与更新开销，重构relation代价；

++++

索引与hash

Ordered indices（顺序索引）/hash indices（散列索引）；

若与对映数据文件本身排列顺序相同的索引称为主索引；（也称为clustering index）；

不是sequential file就没有primary index；

Secondary index辅助索引，search key与file 的 sequential order不同；一定是dense index；

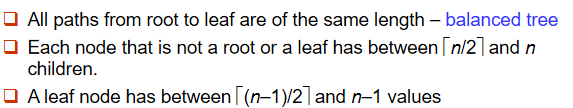
Dense index（稠密索引）；为每一个search key value（相同的key可以只分配一个）分配索引；

Sparse index（稀疏索引）；只有部分search-key values被索引记录；--->只能用于顺序存放文件；

Multilevel index：索引也可以使用嵌套，使得索引至少可以存入内存中；在涉及index的更新时，从下层向上层进行操作；

B+树索引

优点；自动通过最小的改动来组织自身；不需要用重新组织的方式来保持性能；

缺点：需要引入一些多余的操作；

树高的问题：

b1c3d1b4bd93a8ff85a824e5b77a02c插入的split与向上传递；一般为最左侧的leaf data成为上一级node的index；

Deletion，当个数不够时，往往向左合并（可能还需要再分裂（重分配））；删除后的合并，先找同一父节点下node合并，因而不一定就是单向向左。

\*\*\*B tree

每个value在树上只能出现一次，因而有一些value只能出现在非leaf节点中，较混乱；

B tree的优点：可能使用更少的node，有时可以更快的找到对映node的value；

劣势：B tree有更大的non-leaf node，在block大小不变的情况下，每个node的分支减少，因而扇出下降，从而B tree往往会比B+ tree有更深的深度；从实现角度而言，也较B+ tree更为困难；

\*\*static hashing

通过hash函数，将block映射到对映的bucket中，从而方便查找；

Hash函数的定义要尽量的平均映射；

解决overflow的问题，可以使用list串联一系列buckets（closed hashing）；

Hash indices常常是secondary indices；

\*\*dynamic hashing

每次使用hash值的prefix进行记录，根据bucket中数据的多少决定使用的位数；

每个bucket的搜索深度相同，只要bucket满，就可以继续分裂，因而无需分裂的bucket存在多个index指向同一个bucket的情况；overflow的情况可以如前一般处理；

Order更适合于范围查询；hash通常更适合检索具有指定键值的记录。

LSM（log structured merge） tree

多层树结构，新增即插入，删除则只增添标记；

从level 0 （树满）到level 1，直接写入即可，若L1不为空，则是顺序扫描合并leaf node；

从level 1（大小到达阈值）到level 2 以及更高的level，需要使用生成与替换的方法；

当一个level的一棵树填满后，可以开启另一科树，之前的树合并入下一level；

按序合并，只记录最新的数据；

优点：顺序执行I/O操作，leaf为full，浪费空间少，相比于B+ tree，I/O操作减少；

缺点：存在相当数量的冗余；查询需要找多棵树；

++++

BUFFER tree

为B+ tree的每一个node添加一个buffer；插入不是直接到底，直到buffer充满后，插入才开始下沉；对于buffer tree的查找，先在buffer中进行遍历搜寻，没找到再继续向下；读操作更快，但是写有较大的掣肘；

Multiple-Key access

多条件的查找；combine-key

Grid file；

Bitmap indices：bitmap记录位数与record条数相同，分类数与属性对应的种类数相同；

可以用bitmap间的逻辑运算来表示复合条件下的搜索结果；

语法分析与翻译（parsing and translation），check syntax，translate query into extended relational algebra；生成对映的保护计划；

先执行选择操作，再进行笛卡尔积合并；

使用numbers of block transfers and numbers of seeks 来计算cost消耗；（不考虑要把最后结果写回disk）

////

Selection的成本估算：

A1：linear search

1 seek + br block transfers

A2:binary search

100a8e8cfc05da349acfd5141783b60

9bb0a9b7a2caac416f73fface6a80d3如果选择的属性不再key上，则可能有多个项满足，这时有：

其中sc（A，r）表示预测记录结果数，fr是每个block可容纳的最大记录数；

A3: primary index equality on key（寻找单个记录消耗）

c3f7b3a8aacc8e419a9b043fbf07272

A4：primary index equality on nonkey

ddef03cd74de367914a96fe7f63bda3

b表示sc（A，r）/ fr即符合结果的块数；

A5：secondary index on key

如果同样是candidate key

6d213376558f0a5550312c7528ddee4

如果不是一个candidate key

745f642b19ada671cea27ba8ec2b2bd

对于n个结果，需要到leaf后继续seek and transfer n次才行；

范围数据获得：

A6：primary index comparison

A7： secondary index comparison

两者与前类似，先找到分界点，然后使用linear；

多条件合取：

A8：conjunctive selecting on one index

从最小代价开始一个一个条件解决；

A9：conjunctive selection on composite index

使用复合index，大步到位；

A10：conjunctive selection by intersection of identifiers（标识符交集）

使用指针代替记录的思想，先对不同情况求得指针结果，对指针取交集，最后再把指针转换为具体的数据结果；

多条件析取：

A11：disjunctive selection by union identifiers

除非每个条件都有对映index，否则使用linear scan更为方便；

\*sorting的seek and transfer消耗：

External merge sort：

9bf5fcc98bcd0008afc9ab7c8c1f732对于ceiling log部分，因为内存需要存放结果，所以每次最多读入M-1块进行排序；因为总共的块数为br，且内存大小为M，所以经过第一轮排序，我们可以获得br/M块排序后结果块；从而ceiling log为需要进行的外排轮次；

因为外排不能得到最终结果，因而需要写回disk再读出，所以需要\*2，加上第一次分区排序，以及每一次都需要全部遍历一遍所有的block，因而transfers数量如上；

50a2b1da70659c581995665822a49e2Seek：

最开始需要br/M次seek，每一次读入内存时都需要seek到连续读入的blocks，随着merge过程，M可以容纳的block数量逐渐下降，设bb为buffer size，写入内存都通过buffer实现；此时每一轮需要br/bb次seek，从而最终结果如上，最后一次计算数据不需要写回disk，所以可以总体 - 1。

Join消耗：

假设r是outer relation，而s是inner relation；

tT表示 transfer time of a single block;

tS 表示 seek time of a single seek;

1. Nested-loop join：

作用目标为tuple（行），因而有：

3fa465de21029215ca2238f965ae857Outer每行取一次，inner以块对映，加上外层去块需要br次；

Seek直接等效为nr次seek inner与br次seek outer即可；

1. block nested-loop join

区别于tuple，基于block进行join

8f1f86f6a1c937620a83f4d09472944考虑当次结构时：

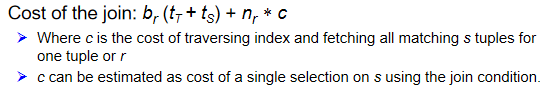
分析基本同上，对于seek inner/outer均只需要br次seek即可；

若memory支持每次读入多块，假设memory大小为M，则有：

8c496cf2d537cc6339bea8dde94a598

Memory需要预留两块空间，一块用于存储merge的另一方数据，一块用于存储merge结果。

1. Index nested-loop join

br次遍历outer，且对于每一条br中的tuple，需要c的时间来进行inner index搜索每一个tuple；（tT与tS近乎可以省略（合并））

1. merge-join

23a0dc78f4f0a22c1ac8b2edc7ede54Transfer相当于每个块遍历一遍即可；对于seek而言，根据buffer的大小，每次可以读入的跨个数则为br（bs）/bb，因而seek的个数如此；

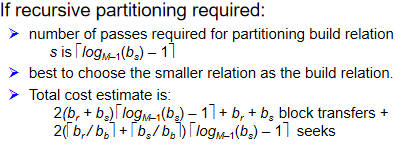
1. hash join

若不考虑hash table过大的情况，即hash table可以存储与内存之中，则有：

d65453447e580973724b74274b66718其中2\*（br + bs）用于outer与inner的分块（读入加写出），再用（br + bs）的读入进行对映块的合并，因为分块的数量一定大于等于inner关系与outer关系的分块数量，从而需要4\*nh的额外transfer（nh为hash函数的分块数量）;

对于seek而言：划分时的读入与写出需要br/bb与bs/bb（以块进入以块读出），但在划分后的读入中，每次只需要nh（r/s关系表各一次）即可--->已经划块分类。

若需要迭代使用hash 函数，则对映的seek与transfer需求如下：

nh太小忽略不计（maybe）；

1. complex joins

多条件析取/合取搜索；可以一层层求取，或者结果求交集/并集......

另外一些可选的策略：

Duplicate elimination：删除重复项，只记录第一项；

Projection：对于结果tuple实行预测；

Aggregation：分类后实行聚合函数；

Evaluation expressions：

Materialization：计算expression的结果并存放进入disk；

Pipelining：计算结果直接向后传递；

在计算对映seek与transfer消耗时，考虑最坏情况，即B+ tree每个节点只有一半会被占用。

////

++++

Query optimization：

Evaluation plan：defines exactly what algorithm is used for each operation（使用什么算法性能最优）, and how the execution of the operations is coordinated（如何协调不同结果）.

先获得等价的逻辑关系表达式，再通过不同的算法实现获得替代的表达式，最后根据预测结果挑选消耗最少的计划。

等价关系式的要求仅仅建立于合法数据库之上。

Equivalence rules：

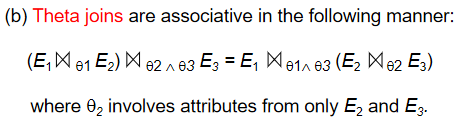
Conjunctive selection 可以转变为串行的select操作；

Selection operations are commutative（可交换的）；

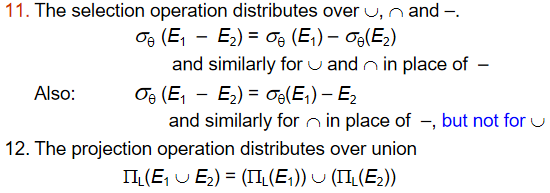
一连串映射中（如果映射相互包含），只有最后一个映射生效--->限制不断框定；

Selection可以融合进入自然连接（笛卡尔积中）；

Natural join operations are associative5b5a430416229ed33f3f22e9818df99；（自然连接可联合）；

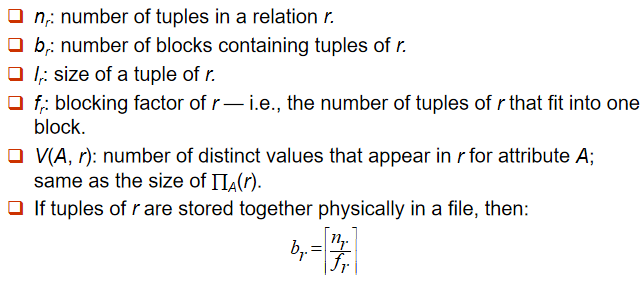
等价过程中，总要先选后连，先投后连--->宗旨，连的最少；

可以根据最后输出的需求，扩展中间步骤的投影属性个数，目的只要使表变小即可；

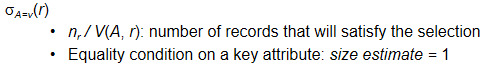


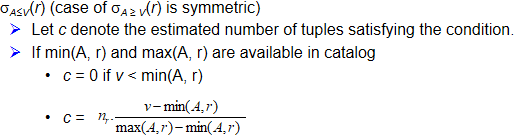
先完成selection与projection操作；

生成等价表达式：将所有适用的等价规则应用于每个等价表达式的每个子表达式，直到不能产生新的表达式为止；

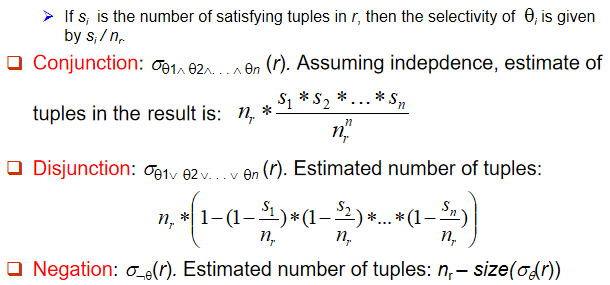
预测结果数量的方法：

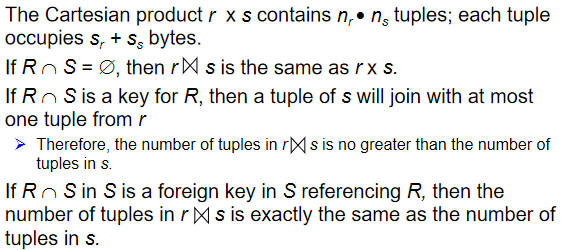
使用histograms，对一个属性的值进行大致的分段（等宽分段/等数量分段）；

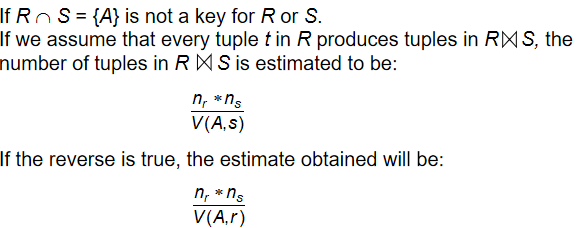
对于范围查询而言：类似于概率的算法即可；



对于conjunctive selection类似于概率的计算；独立计算个数，使用s/n，类似于概率计算即可：



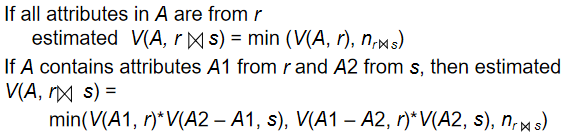
对于join的size估计：



两者计算的最小值即为较为精确的预测结果项；

Select：

9984bf3ff46a8bcc6c5c44c38c784cfJoin：



Cost-based optimization：基于开销，选择最优路径进行计算；

Heuristic optimization：启发式优化，把投影select向下沉，且先执行小表的连接操作；

Left-deep join trees，自下而上，从左至右开始执行evaluation plan，每次运算结果与右侧tree继续进行对映计算（可以考虑使用pipeline的方法用以优化性能）；

SQL语言对于nested SQL语句的处理往往类似于函数查询过程，有较大的的性能损失；

使用join往往比使用nested SQL语句更加高效；但nested SQL语言并不是总能转化至from语句中，tuple的重复数量可能会出现问题；

可以通过将nested 转换为 with table来优化性能；---->decorrelation.

A materialized view is a view whose contents are computed and stored.（物化视图会保留在内存/disk上）

Materialized view maintenance需要根据insert/delete/update等不断更新（重新生成）物化view，因而提出了incremental view maintenance的概念；基于已有的物化view，我们每次仅计算需要在原view上修改的内容（add or minis），从而更加简便的更新物化view；

6b71190d9f6181302cf5beebb8fa22a维护了view new与view old两个概念；

对于数据表，R介导属性列，而r介导具体tuple；

View对于projection而言，则需要维护对映的count记录，删除部分记录并不意味着projection view中对映的记录会消失，可能依然存在其他同属性记录；

维护sum的结果时，我们即需要有sum的直接结果，也需要维护对映的count记录，count用于check当前group下是否存有tuple--->sum = 0并不意味着该group下不存在tuple；

维护view后并不一定要使用view，特别是在查找等问题上，如果不同表上均含有index，则不需要使用已连接的view，直接使用index进行搜索可能会获得更加高性能的结果。

++++

Transaction（事务）：

Execution can be inconsistent, but when commit the database must be consistent.

Atomicity：所有操作完全做了或完全没做；

Consistency：数据库始终具有一致性，事务执行前后数据不会向矛盾；

Isolation：即使数据库可以被并发访问，但是并发访问的过程及结果是相互独立的，事务运行之间互不干扰；

Durability：事务一旦提交之后，其作出的改变是永久性的，不会再发生改变；

一些并行运行事务的概念：  
 Active：事务正在进行（为提交）属于活跃状态；

Partially committed：部分提交，事务已经部分完成，但是尚未提交；

Failed：在恢复之后，对映事物不能正常执行；

Aborted：事务回滚，因而数据库状态返回至上一事务完成时样式；

Commit：正常提交；

Shadow database：

数据表被不停拷贝，每个并行使用/操作的仅是原表的一个复制；如果事务aborted则删除对映的copy，若commit则将对映的指针指向copy database然后删除old one.

Concurrent 并行操作的优势：

1. 提高processor与disk的利用率，从而提升相应的吞吐量（throughput）；
2. 减少事务的平均响应时间，短事务不需要等待长事务；

Schedules，表示不同事物间指令在并发运行时的完成顺序；

Serial：串行的；

n个事务有n！中串行调度方式，串行调度必能保持数据库的一致性，但效率较低；

Serializability：可串行性；

Conflict serializability；冲突可串行化；

View serializability：视图可串行化；

事务内部操作仅讨论read and write

Conflict serializability：冲突可串行化；

冲突表示，事务A与B对于item Q的处理先后顺序会得出不同的Q结果；

只有read操作时不会相互冲突的；

如果可以把一个调度转换为一个对映的顺序执行调度，则称这个调度为“冲突可串行化”；这两个调度则称为：conflict equivalent，冲突等价；

View serializability：视图可串行化（满足下列三个条件）

首读：第一次读取item Q的事务操作需要保持不变；

写读：事务间写读item Q的顺序不能发生改变，不然会产生conflict；

末写：最后write item Q的事务操作需要保持不变；

冲突可串行化一定是视图可串行化；

Recoverable schedules：

Cascading rollback：a single transaction failure leads to a series of transaction rollbacks.

Cascadeless schedules：该调度下，每一事务的rollback之间并不会相互影响；

在SQL语言中，事务总是被隐式的触发；如果执行成功，则事务也是被隐式提交；

可以使用precedence graph来check 某一调度是否可以被serializability；

在precedence graph中，每个事务表示一个vertices；对于两个冲突的事务，我们使用有向线段连接它们，其中最先用到冲突item的作为有向线段的起始段；

如果graph中不存在回路，则该调度是“冲突可串行化”的，且根据有向线段的指向，我们可以便捷地给出与当前调度等价的serial调度；

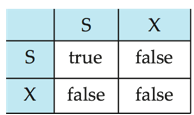
检验视图可串行化是一个NP-hard问题，因而常常只是使用一些充分性check方案；

Concurrency control：

Exclusive（X） lock：可以对item进行读和写，往往称为写锁；

Shared（S） lock：只能进行item的读，因而被称为读锁；

读写锁兼容关系如下：

只有添加锁后才能进行相应的操作；

如果没有对映的locking protocol进行管理，则可能出现死锁（deadlock）的情况，两个事务间相互等待必要锁的释放与获得；

同样，starvation（饿死）也是一个并行运行时常常存在的问题；因为X锁的获得难于S锁，因而一连串S锁的获得请求，会导致对X锁需求的事务一直无法得到对映的锁，从而饿死；

Two-phase locking protocol：二阶段锁协议（2PL）

事务主要分为两个阶段：

1. Growing phase，不断获得锁；
2. Shrinking phase，不断释放锁；

只要开始释放锁，事务就不可以再获得新的锁；转折点称为lock point.

2PL保证了冲突序列化的要求（借助于lock point进行证明）；不能保证the freedom of deadlock；

为了防止cascading rollback的发生，可以使用strict two-phasing locking 协议，此协议要求事务一定要持有所有需要exclusive clock直到commit或者abort；

Lock-table：往往是存在于内存之中的哈希表，且以数据作为index；

锁表中用不同颜色表示上锁状态（蓝色）与锁等待状态（白色），用箭头表示等待关系（时序记录，除非读锁可以直接获得）；如果事务abort，则所有当前事务的等待锁都需要被删除；lock manager会记录相应事务的锁分配情况；

Graph-based protocols：

构图方法类似于前述precedence graph；

一种graph protocol：

Tree-protocol：

只支持上X锁；

每一事务上的第一把锁可以任意选取，进而上其他锁前，一定要获得对映锁的父锁权限，才可以进行相应的上锁；已经被其余事务上锁的item不能被第二个事务继续上锁；

优点：可以保证调度为冲突可序列化，且freedom of deadlock；

介导更短的等待时间，增加事务的并行度；

缺点：并不保证可恢复性与级联自由；

父锁锁定的要求导致事务锁定其本身不需要的item；

2PL protocol与tree protocol互不包容但互有交集；

Timestamp-based protocol：

给予每一个事务一个时间戳（time-stamp）；

该协议对每一item（data）标定两个time-stamp；

1. timestamp：记录最近（最大）成功写的事务时间戳；
2. timestamp：记录最近（最大）成功读的事务时间戳；

！！有没有等于注意一下！！

如果事务的S（Q）小于等于对映item的W（Q），则数据已被重写，对映事务需要rollback；

如果事务的S（Q）大于等于对映item的W（Q），在成功读取数据，需要将item的S（Q）进行对映的更新操作（max事务时间戳/item本身S时间戳）；

如果如果事务的W（Q）小于对映item的S（Q），写被拒绝，事务需要rollback；如果如果如果事务的W（Q）小于对映item的W（Q），写被拒绝，事务需要rollback（其实可以只是拒绝，而没必要进行rollback）；其余情况，写操作被接受，需要用事物对映的时间戳对W（Q）进行更新；

因为时间戳的读/写表现一定要不断增大，因而时间戳协议可以保证序列化的要求；同时，其保证没有死锁发生；同理于tree protocol 它不保证可恢复性与cascade free；

为解决上述两个协议的弊端，可以要求事务在最后再执行写操作，且写操作是唯一的，当一个事务在进行写操作时，另一个事务不允许进行写操作；

----！！！！！

\*\*\*Validation-based protocol

分为三阶段执行：

1. 事务只向临时变量写入；
2. 在写入前验证写入过程是否会产生序列化冲突；
3. 如果验证通过，则可以进行写入；

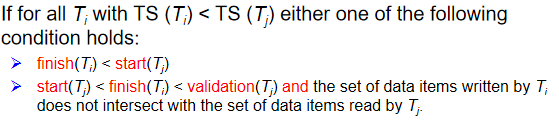
每个事务的每个阶段均有对映的一个时间戳；

序列化过程由不同事务的验证阶段时间戳排序定义；

Multiple Granularity（多粒度）

Fine granularity（细粒度）：可以提供更高的并发性；但上锁解锁代价也相应上升；

Coarse granularity（粗粒度）：较低的并发性，较低的锁管理需求；

Intention lock（注意力锁）

Intention lock为当前点的所有祖先（一条path）均显示上对映的锁；使得更高的level在并行性check时，可以先判断一部分并通过一部分；

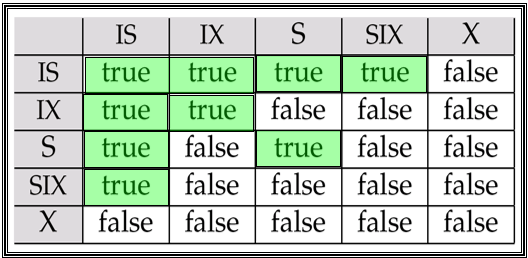
Intention lock的种类：

Intention-shared（IS，共享型意向锁）：表明后代有node上了S锁；

Intention-exclusive（IX，排它型意向锁）：表明后代有人上了X锁；

Shared and intention-exclusive（SIX，共享排它型意向锁）：同一个事务中，在较高的级别获得了S锁，而在较低的级别依然可以分配X锁，因而在高级别对映的node中，便需要S与IX锁的同时存在；注意是统一事务中；

意向锁的相容矩阵如下（保持最大通过度要求）：

意向锁需要从上至下添加；解锁过程则为自下向上，且事务的加减锁符合2PL协议；

优点：增强了并发性，降低了加锁的开销；

\*Multiversion schemes

使用多版本方案，保留数据表旧版本以提升并行访问的性能；

多版本使用时间戳排序；

每次成功写入均添加至最新版本中；当进行读操作时，则根据事务的时间戳进行合适版本的数据读取；这样使得数据的读取不需要等待，可以立即返回合适的版本；

每一个item都有一组对映的版本数据；每一version of item均有W（创造时间）/R（最新成功读入时间）-timestamp；创造一个新version后使用事务的timestamp进行初始化；

多写同时间戳根据处理顺序先后进行覆盖写入，写时间戳若有先后则需要创建新的item version；

\*Multiversion Two-Phase Locking

Update transaction，需要占有所有read and write lock直到事务结束；

写事务会在每写一个version时生成一个对映的时间戳，而read事务仅在事务开始时生成一个唯一的timestamp；

Deadlock handing解决死锁问题：

Deadlock prevention：死锁预防；

Deadlock detection and deadlock recovery：死锁的检查与恢复；

Deadlock prevention：

1.事务需要获得所有用到的item对映的lock再开始执行，也被称为：“conservative 2PL”（保守的二阶段协议）；但会带来并发度较差，难以预测的劣势；

2.对所有数据（item）使用进行排序，要求事务只能按顺序锁定数据项，（基于图）不能形成一个循环；

3.timeout-based schemes：

事务会等待需要的锁一段时间；若一段时间后仍为获得锁，则事务进行roll back，因此，死锁便不会发生；

4.wait-die与wound-wait策略（需要使用到时间戳进行辅助）；

如果事务因策略roll back，其原始的时间戳可以继续继承，防止“年轻”事务饿死的情况发生；但是超时的阈值较难设定；

Wait-die策略（non-preemptive）：“老”事务等待“年轻”事务去释放他们需要的锁，“年轻”事务不等待直接进行roll back；

Wound-wait策略（preemptive具有优先权）：“老”事务在未获得的锁上面会强迫“年轻”事务死去；而“年轻”事务会一直等待“老”事务的执行直到其释放锁；

Deadlock detection：

同前构图规则，根据事务间依赖关系，绘制事务依赖图；称为wait-for graph，仅当wait-for graph中不存在cycle，则表示该调度没有死锁；

Deadlock recovery：

使用roll back的方法去解开死锁，但选取哪一个事务去roll back有待商榷；如果选取过于随意，则又可能有starvation的情况发生；

Total roll back：abort事务，并重新执行；

Partial roll back：只roll back部分事务，只要死锁能解开即可；

将write与read操作进行拓展，我们考虑select与delete等操作：

因为select/delete/update等操作与write/read的先后顺序不能进行替换，因而需要在执行操作前对其上X锁；如果相互之间不上锁而并发进行，则会发生一种“phantom phenomenon（幽灵现象）”，因而我们需要对表上锁；

如果直接对全表上锁，则会导致并发性大幅降低，比如聚合函数与查询是可以并行的，因而考虑到index lock，我们规定进行insert/delete/update操作时，需要先能申请到对映index处的lock，有lock则可以继续执行；就此，我们每次只需要对index进行上锁即可，很大程度解决了并发性的问题，使得聚合函数与查询语句不再发生冲突；

对索引上锁过程要求2PL协议会导致较低的并发度，因为query对索引访问频次远远高于对数据表的访问；另一方面，在保证正确性的基础上，对索引进行非序列化的并发访问是可以接受的；

索引访问不关心数值，只关心节点的正确性；

\*\*另有一些索引相关的协议，什么锁所有子节点，然后释放本身；锁子节点前，释放父节点锁；插入/删除、拆分合并等模式下，升级锁为排它锁等等策略；一些协议的使用可能导致更多的死锁（搜索向下进入死锁，更新向上进入死锁）。

Recovery system（恢复系统）：

Failure classification（事务中断分类）：

Transaction failure（事务层面的错误）：

Logical errors：可能因为data overflow，错误input等问题出错；

System errors：出现了一些语法问题，像是deadlock的情况，系统需要进行相应的解决；

System crash（操作系统的奔溃）：

像power failure或hardware/software failure等情况，易失性数据（内存）丢失；但假设磁盘并未受到损坏；

Disk failure（磁盘损坏）：

Disk destroys或者发生了head crash，磁盘需要被校验与重建恢复；

Storage structure：

Volatile storage：像主存，断电即失；

Nonvolatile storage：disk，tape，flash-memory等存储器件；

Stable storage：理想化的存储，在任何时间发生后，数据均可以得到保留；在异地非易失性存储器上进行多备份存储；

Block传输过程的不一致性：

Partial failure: destination block has incorrect information；

Total failure: destination block was never updated；

Output/input是buffer与disk之间的数据交换，而对于每一work而言，read/write是其对映area与buffer间的交互办法；因而即使事务已经提交，也存在尚在内存buffer中而未写入disk的情况；

Log-based recovery（基于日志的恢复方案）：

Log存放于stable storage；

事务开始时<Ti start>，在write写入之前，需要<Ti, X, V1, V2>记录什么事务对什么item进行了从V1至V2的数值转变；

事务结束之后，加上<Ti commit>的关键字；

Deferred database modification：直到事务commit之后，事务中进行的write操作才开始写入内存之中；

（以下均是在immediate的框架下提出的）

Immediate database modification：事务未commit之前，已经进行的write操作也可以写入内存之中；内存块的写入顺序可以与磁盘块的写入顺序不一致；

其中对映有undo和redo两类操作--->idempotent都是幂等操作；

Redo与undo操作也需要被记录，在undo某一事务后，需要加上<T0, abort>日志表示undo过程的结束；

提出Checkpoint来区块化事务的进行；

Shadow paging：maintain two page tables during the lifetime of a transaction –the current page table（每当page需要被修改时才复制产生）, and the shadow page table（存储不发生任何修改--->用于恢复，直到事务提交后才将指向copy pointer的指针指向current page）.

优点：不需要花费开销记录日志；且恢复是廉价而平凡的；

缺点：copy page是十分昂贵的（有时只需要记录需要修改的node即可）；commit的开销很大，需要进行整表刷新；每次有删除old page的额外操作；

strict two-phase locking：2PL with X-locks held until the end of the transaction.

Recovery with concurrent transaction；

使用redo/undo list表记录需要重做或abort的事务；

从底向上逐条遍历，直到最近的check point为止；

If the record is <Ti commit>, add Ti to redo-list；

If the record is <Ti start>, then if Ti is not in redo-list, add Ti to undo-list；

If the record is <Ti abort>, add Ti to undo-list；

在check point中的活跃事务，如果不在redo list中，则加入到undo list中；

再次自底向上scan log，直到undo list中的所有事务的start都被覆盖；再从上向下scan，根据undo list与redo lost，对每个事务进行相应的处理；

Rule of log record buffering：

Log record 需要按照顺序输出；

只有当log record已经被送至stable storage上时（包括最重要的commit语句），事务T才可以被提交；

日志应先于数据写至disk；

WAL（write-ahead logging）规则；

No-force policy，在事物提交后，也不要求内存中数据立马写入disk中；

Force policy，要求commit需要保证所有内存data已经写入disk中；

Steal policy，即使事务还没有提交，也允许data写入disk中；

Lock与latches不同，latches是只持续一小段时间的lock，往往在内存中block写入disk时使用；

对于buffer management 一般使用virtual memory的方法，虽然虚拟内存也有一定的缺点（page in disk的额外I/O操作），但直接主存存储灵活性较低，且分区是个大考验；

Failure with loss of nonvolatile storage：

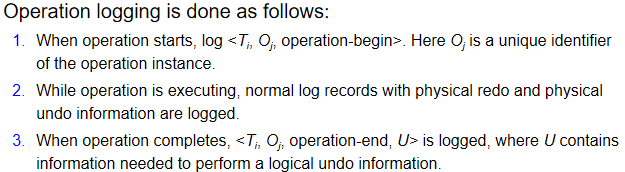
考虑到非易失性存储的loss，需要定期将数据库内容更新至stable storage中，完成这项操作后，需要在日志中添加<dump>标签；

----前

Advanced recovery techniques：

像B+ tree的更新（insert and delete），使用物理日志记录初始态和终止态可能不妥当，可以考虑使用逻辑日志来记录对象item的变化量；

逻辑日志的规则如下：

在逻辑记录未完成时，undo等价于物理日志处理；如果逻辑日志已完成，则直接使用改变量进行undo即可；逻辑日志的redo操作同物理日志；

在恢复的过程中的一些规则：

对于物理日志的起始态终止态改变，undo过程只需要记录<事务名，对象，值>即可，表示item被修改为对映的数值；

对于完全完成的逻辑日志，直接根据变化量进行undo即可，之后可以跳过逻辑日志中间的记录内容；在日志中添加<事务，事件，operation-abort>来记录undo操作；

如果碰到<事务，时间，abort>则可以直接跳过当前逻辑日志内容；直接寻找<T，start>并在日志中记录<T，abort>；

实际recovery过程：

从最后一个check point向下scan：

重复日志记录的history，将内存恢复至奔溃状态前；因为内存会恢复至奔溃状态前，因而只需要维护undo list即可，在scan的过程中，check point处的活跃事务，以及<事务，start>将被放入undo list中；若碰到对映abort/commit标签，则从undo list中取出对映的事务；

在从下向上遍历，完成undo list的清空；注意undo过程也要进行相应的日志填写，在碰到<事务，start>时向日志写入<事务，abort>；可以超越undo过程可以超越checkpoint.

Check point的check过程会要求所有事务停止，因而引入fuzzy checkpoint的概念，先只是声明此处有checkpoint，而checkpoint实际的工作运行，则根据系统的空闲程度进行工作。在checkpoint类中，只有last\_checkpoint是最近check且可以回溯的checkpoint；

最重要的恢复系统：ARIES；

需要使用log sequence number(LSM)对日志进行序列编号；

只处理dirty page，从而减少undo的工作量；

Each page contain a pageLSN which is the LSN of the last log record whose effects are reflected on the page；

Compensation log record（CLR）：补偿日志记录；

Dirty page table：记录尚在buffer中的被update的pages；

Contains：page对映的pageLSN，以及recLSN---最早修改当前page且未写入disk中的事务LSN；

ARIES的恢复过程：

Analysis pass，check point to end；redo pass，the earliest redo transaction to the end；undo pass，the end to the earliest undo transaction；

含有prevLSN前置的日志记录；undonextLSN指向需要undo的相邻事务，跳过不需要undo的事务；

Analysis pass：获取dirty table 将redoLSN（用于指示redo的开始地点）记录flush为min of recLSN，如果没有page is dirty 则设置为checkpoint的LSN；（redoLSN的只需要在checkpoint处更新即可）

遍历过程中，根据日志的记录，不断更新dirty table中的内容；有abort与commit的事务不需要undo，但需要redo--->有dirty table进行记录；

Redo pass，从redoLSN起始点开始，通过redo将内存恢复至crash前的状态；在redo过程中，如果一个事务不再dirty table中获得对映的LSN小于dirty table中对映page 的redoLSN，则直接跳过；

Undo pass支持partial rollback；从checkpoint向前走直至清空undo list；

ARIES’ s features：

Recovery independence：page可以独立恢复；

Savepoints：事务可以记录savepoints并且rollback到对映的savepoints；