**Integrity Constraint Types:**

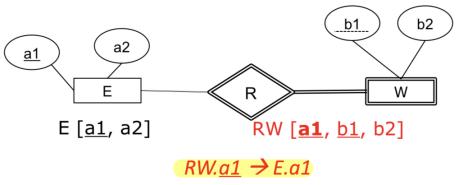
1. Domain constraints 2. Key constraints（key不能重复） 3. Entity constraints（key不能有null值） 4. Referential integrity constraints 5.User-defined constraints

**Mapping ER Diagrams to Relational Models:**

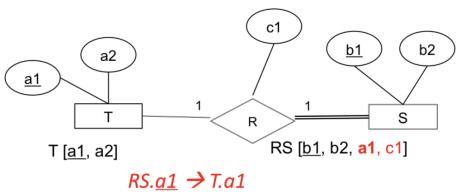
1. Entity Mapping (create relations)

把所有strong entity和simple attributes画出来，如果是composite attribute只写simple.

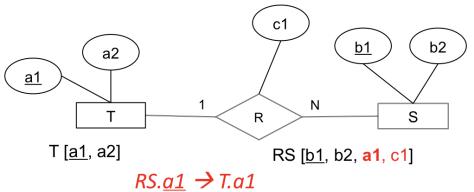
1. Weak Entity Mapping (create relations)



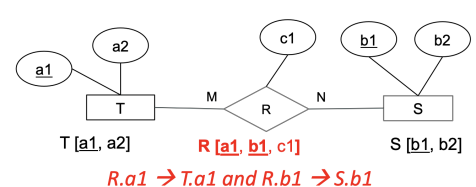
1. Binary 1:1 Relationship Mapping (define foreign keys)



1. Binary 1: N Relationship Mapping (define foreign keys)



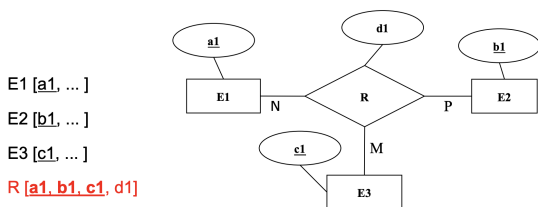
1. Binary M:N Relationship Mapping (create relations and define foreign keys)



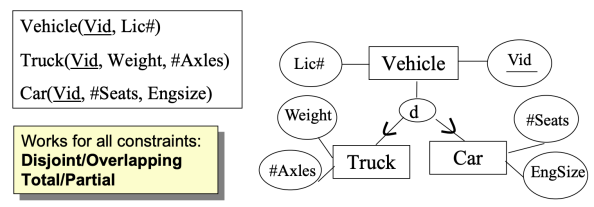
1. Multi-valued Attribute Mapping (create relations and define foreign keys)



1. N-ary Relationship Mapping (create relations and define foreign keys)



1. Super & Sub-classes (mapping of EER)



**Anomalies**

1.**Modification anomaly**修改异常：由于数据冗余导致的数据不一致

2.**Deletion anomaly**删除异常：由于一些attributes的删除导致的某些attributes的丢失

3.**Insertion anomaly**插入异常：无法插入某些attributes而不具备其他attributes的情况

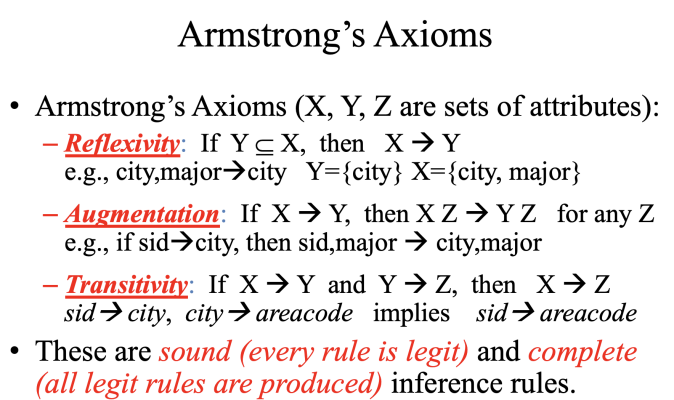
**Functional Dependencies**

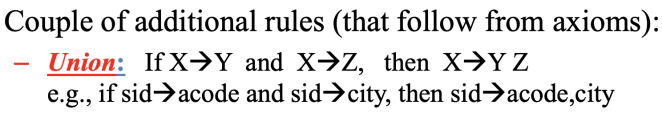
**功能依赖**是一个属性可以决定另一部分属性，而成为key是要能决定所有在这个实体中其它的属性

必须通过定义来体现：Given some instance of R, we can check if it violates some FD f, but we cannot tell if f holds over R!

**superkey**：可以唯一的识别关系，但是不一定是最小的（key才是最小的可以唯一的识别关系）

**Closure：**the set of all FDs implied by F隐含在F中的所有FD的集合。



截屏2023-05-30 10.23.27

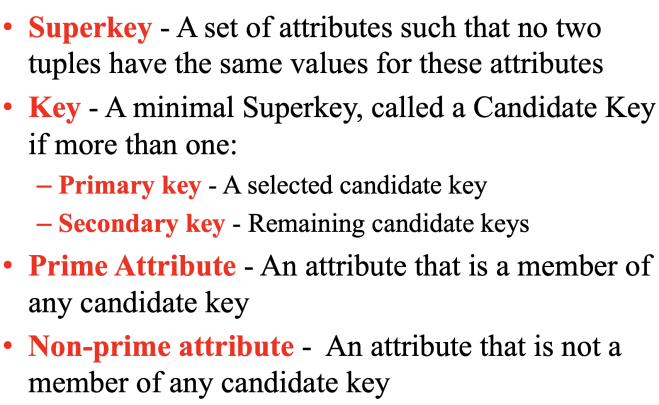
看一组属性是否是超级键意味着检查它的闭合是否是所有的属性。

**Normalization**

Normalization is a process that aims at achieving better designed relational database schemas using

1. Functional Dependencies 2.Primary Keys

The schemas that satisfy certain conditions are said to be in a given ‘**Normal Form**’.



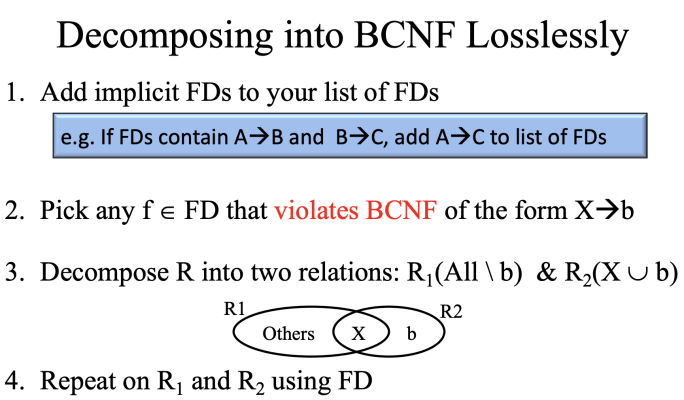
**1NF：**

第一范式要求数据库中的每个属性都是原子的，即不可再分。

**2NF：**

要求数据库中的每个非主属性都完全依赖于关键属性（主键），而不是部分依赖。

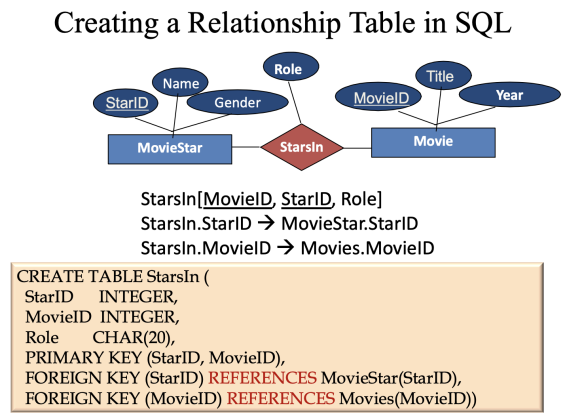
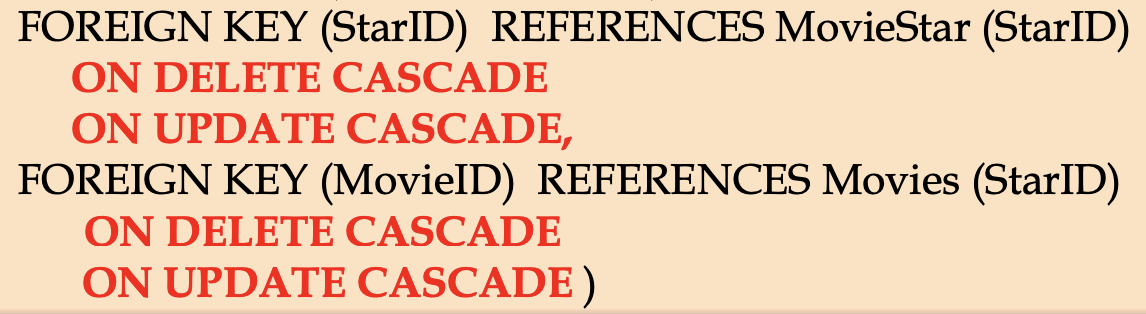
**BCNF:**

****

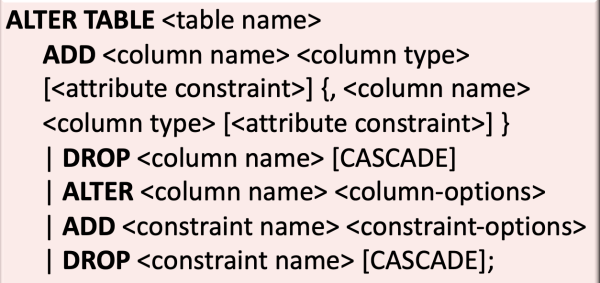
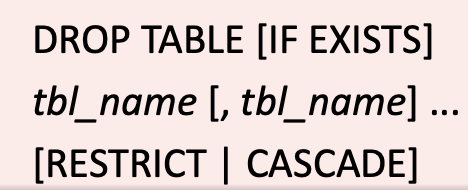
**SQL:**

1. DDL(Data Definition Language)

statements (CREATE TABLE, ALTER TABLE, DROP TABLE)

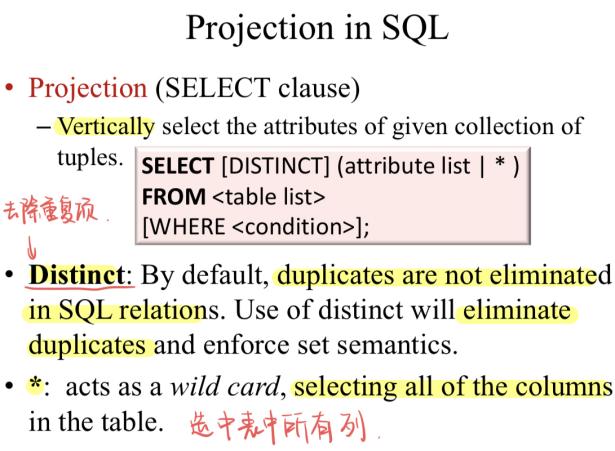
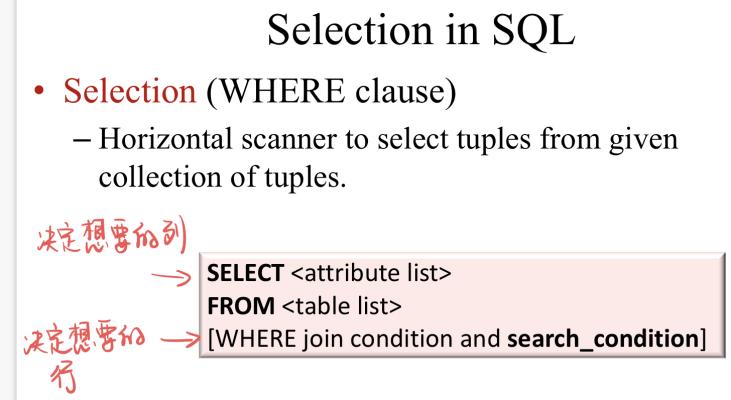
ON DELETE (SET NULL | SET DEFAULT | CASCADE)设置为空｜默认值｜级联

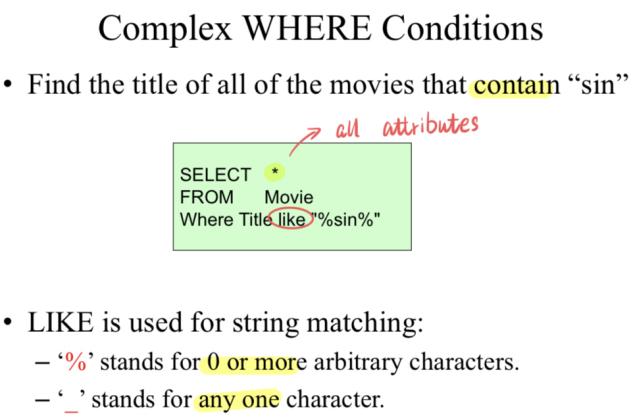
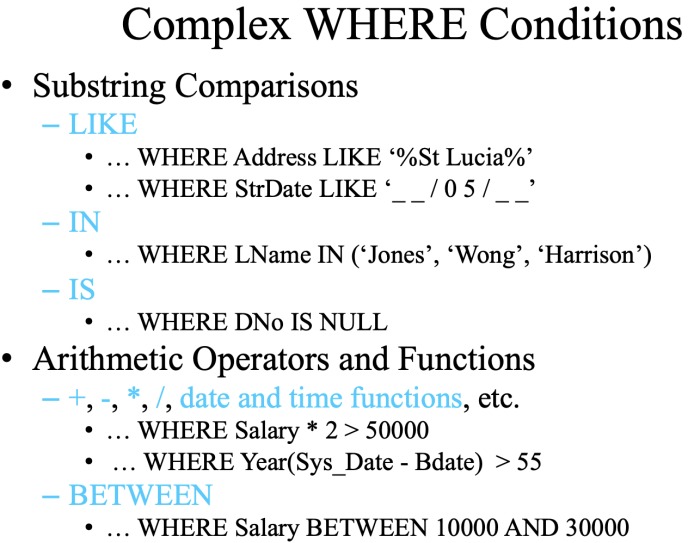
 

1. DML(Data Manipulation Language)

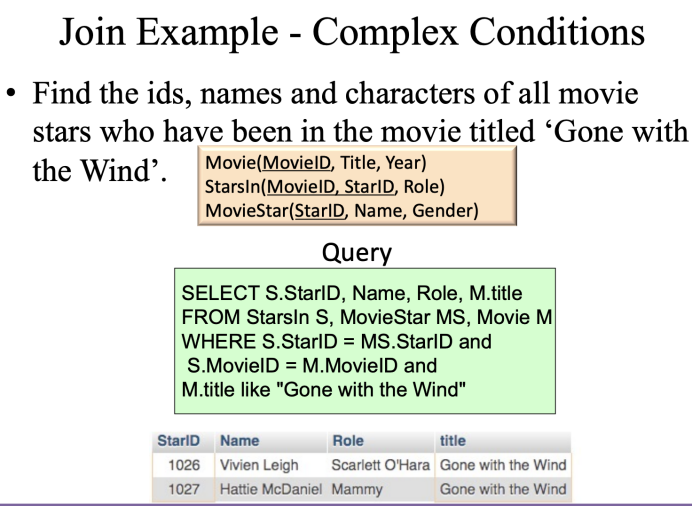
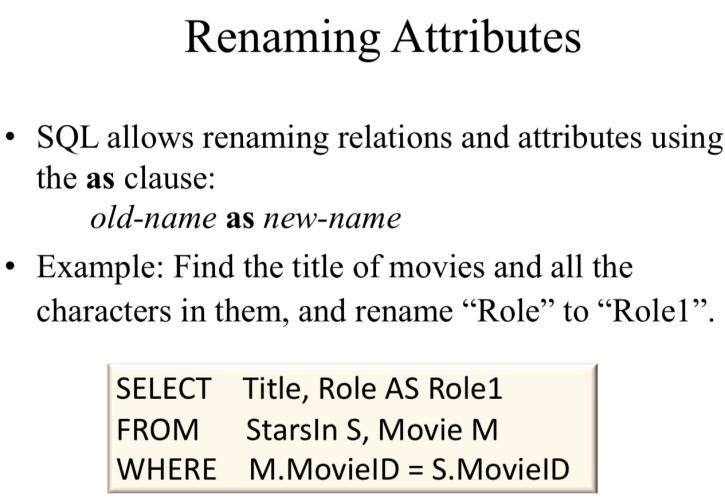
statements (INSERT, DELETE, UPDATE, SELECT)

**--------------------------------------------------SELECT**：

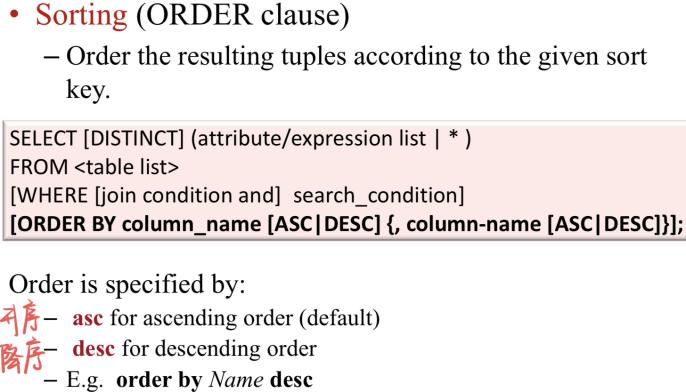
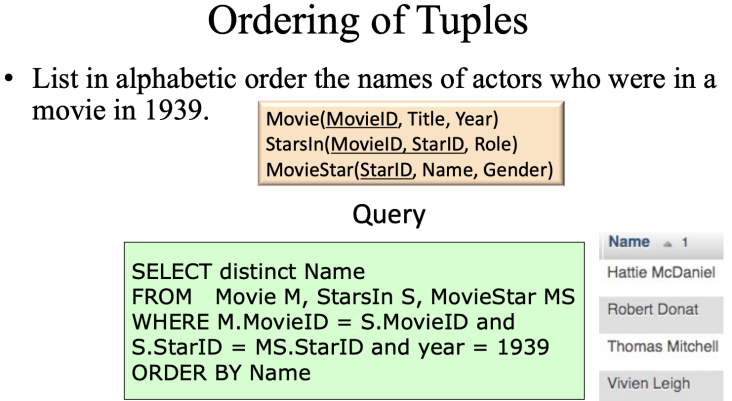
 

Join:

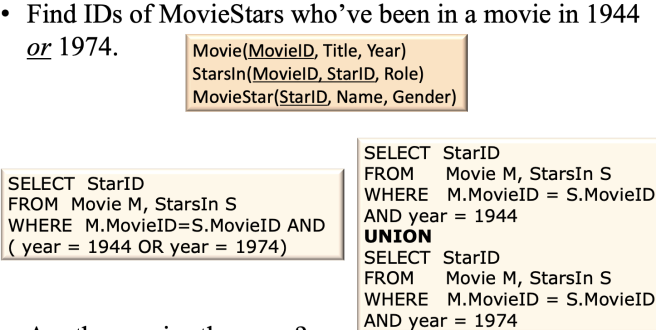
 

**-------------------------------------------------------------Sorting:**

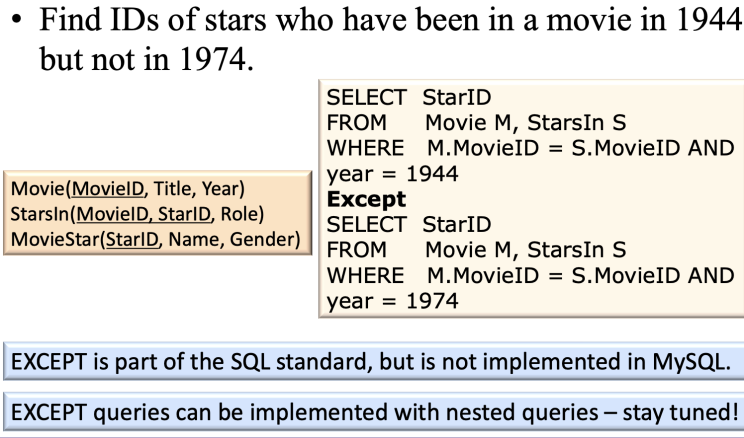
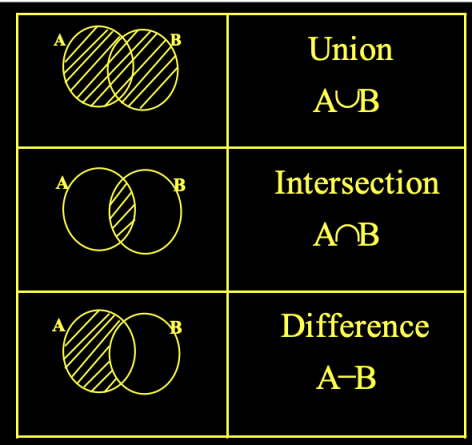
 

**----------------------------------------------------Set Operations:**

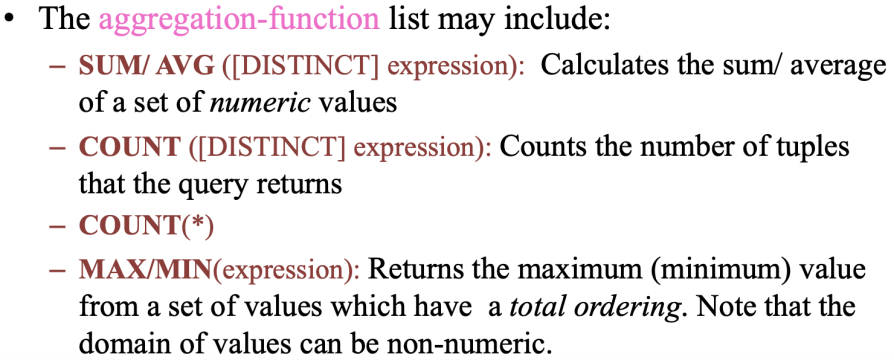
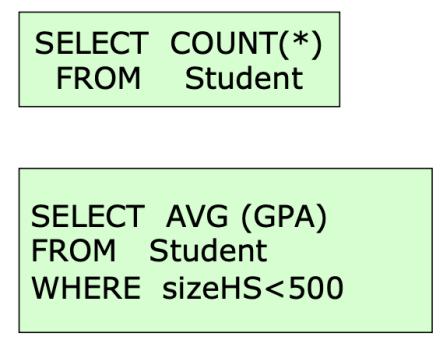
Union: Intersection:

Except:

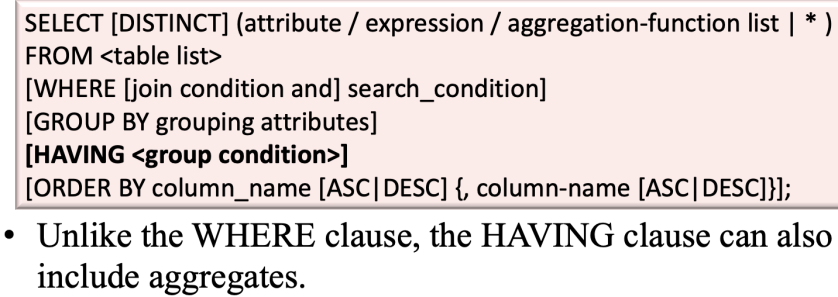
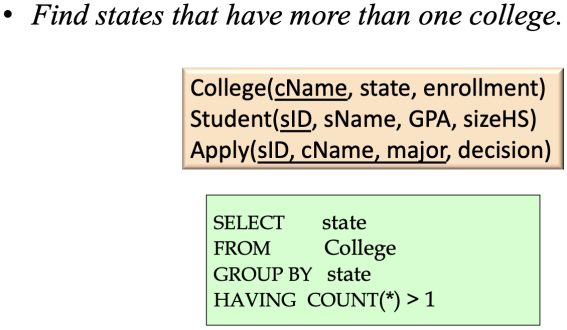
 

**Aggregation：**

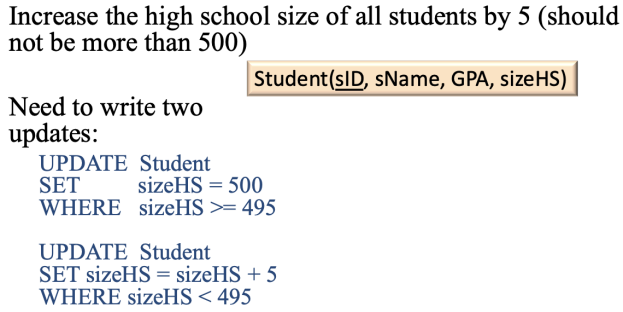
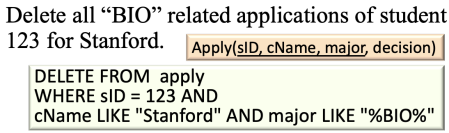
** **

聚合表达只能出现在having和select中

**--------------------------------------------------GROUP BY & HAVING:**

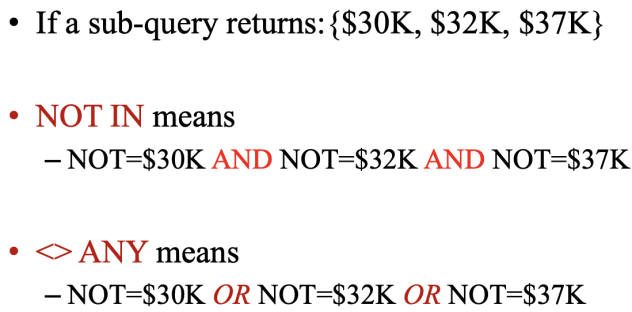
**------------------------------------INSERT, DELETE and UPDATE statements:**

** **

**-------------------------------------Nested Queries:**

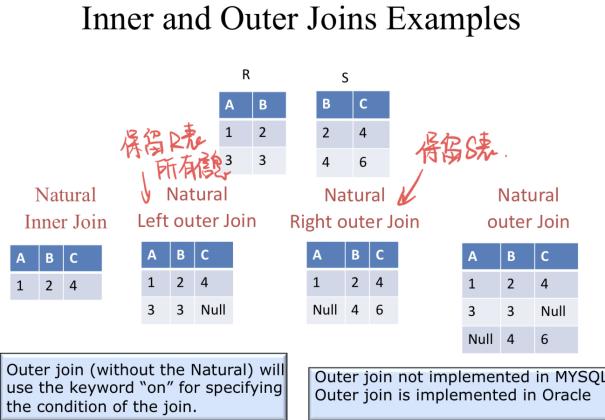
ANY: Evaluates to true if **one** comparison is true （“IN” = “= ANY” ）

ALL: Evaluates to true if **all** comparisons are true



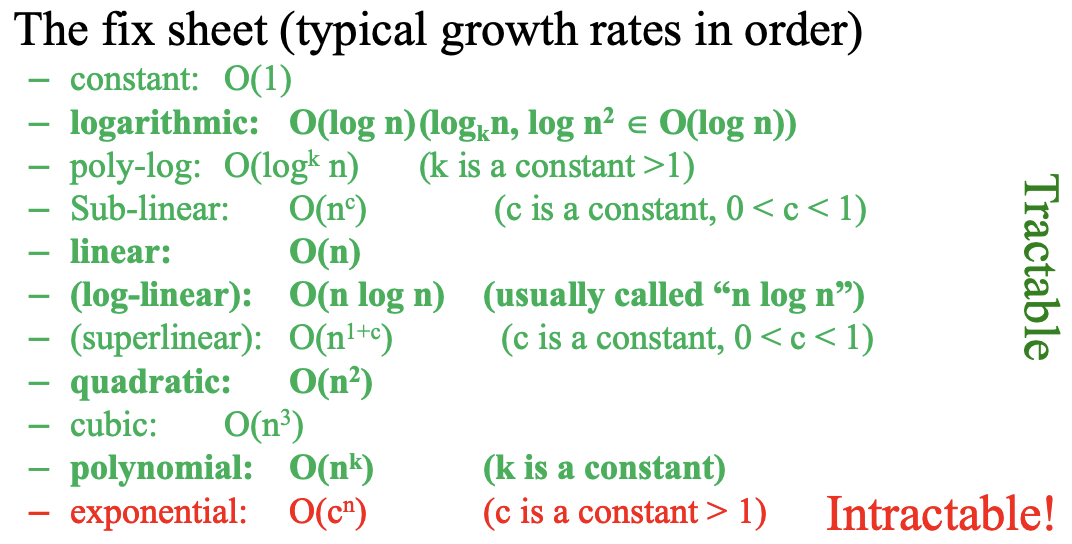
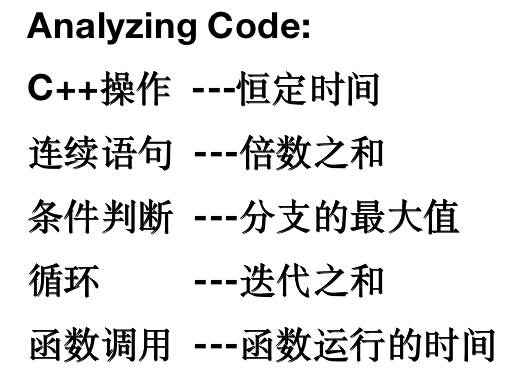
sub-query子查询不能包括ORDER BY子句。可选的DISTINCT关键字可以有效地对子查询的结果进行排序.

推荐：展现的数据从多个表来的时候使用join，而当需要用总值比较其他值的时候使用Nested

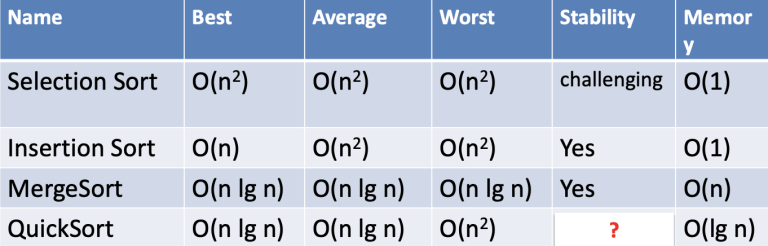
**Join:**  

**Big O:**

O(n!)>O(2^n)>O(n^2)>O(nlogn)>O(n)>O(log n)=O(1)

 ****

**Sorting:**

****

stability: 如果两个键值相同的对象在经历排序后输出的顺序与他们现在的顺序相同，那么该程序算法就称为稳定的

A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.

**Selection Sort(选择排序)**

方法：从第一个位置开始，第一个的数跟剩余每个位置遍历一遍比较大小，如果其他位置的数小于备选的数就从这个数开始继续遍历，直到遍历结束，最小的数放到第一位，接着从第二位开始遍历，以此类推，直到最后一位结束排序。

选择排序**不稳定！**

**Insertion Sort(插入排序)：**

方法：

1.将待排序序列的第一个元素视为已排序序列。

2.遍历待排序序列，从第二个元素开始将其插入到已排序序列中的正确位置。

3.比较当前元素与已排序序列中的元素，找到合适的位置插入。

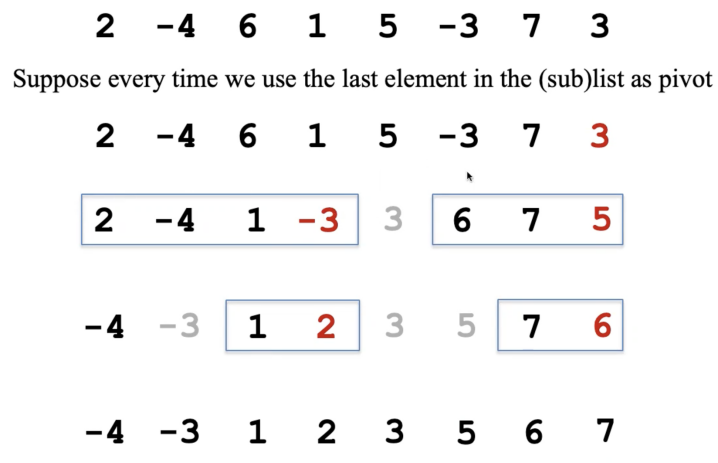
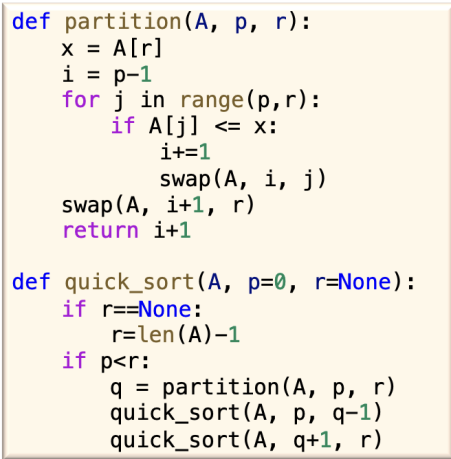
4.将当前元素插入到正确的位置，并将已排序序列中的元素后移，为新元素腾出位置。

5.重复步骤2至步骤4，直到遍历完整个待排序序列。

**Merge Sort(合并排序):**

方法：先一分为二，然后左右两边分别排序一对一比大小，之后两个两个排序后再合并，以此类推。

**Quick Sort(快速排序)：**

****

**使用场景：**

**选择排序：**适用于写操作昂贵而读操作相对便宜的场景，如在使用闪存存储器或外部硬盘进行原地排序时。在这种情况下，选择排序的优点是只需要进行O(n)次写操作。然而，选择排序的时间复杂度为O(n^2)，在大规模数据集上性能较差。通常情况下，其他更高效的排序算法更常用。

**插入排序**：

1.数据接近有序：当数据接近有序状态时，插入排序是一种很好的选择，因为它能快速完成排序，只需要少量的比较和移动操作。

2.小规模问题：对于规模较小的问题，插入排序是一种高效的选择，因为它的额外开销较低，实现简单且容易理解

**合并排序**：适用于外部排序，即在需要对无法完全放入主内存的大量数据进行排序时使用。它通过将数据分割成较小的块，对每个块进行排序，并最后合并这些有序块来得到完整的排序结果。这种算法能够高效地处理海量数据，最小化对外部存储器的访问。

**Insertion Sort**:

def secretFunction(aList):

for i in range(1, len(aList)):

tmp = aList[i]

k = i

while k > 0 and tmp < aList[k - 1]:

aList[k] = aList[k - 1]

k = k - 1

aList[k] = tmp

def **selectionSort**(arr):

for i in range(len(arr)):

min\_index = i

for j in range(i + 1, len(arr)):

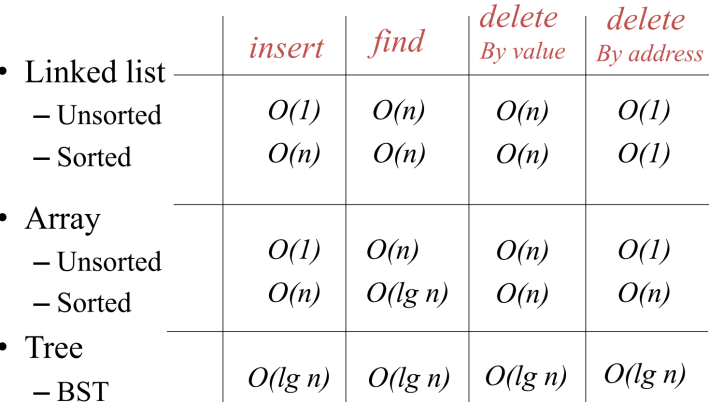
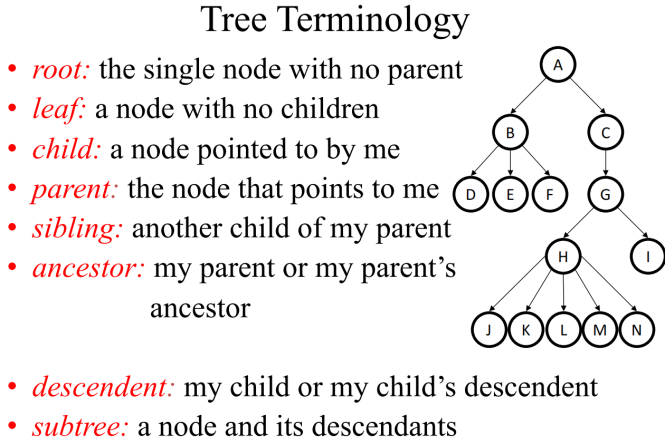
if arr[j] < arr[min\_index]:

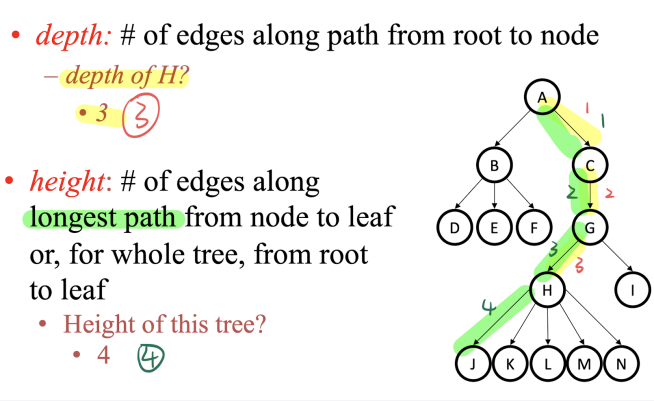
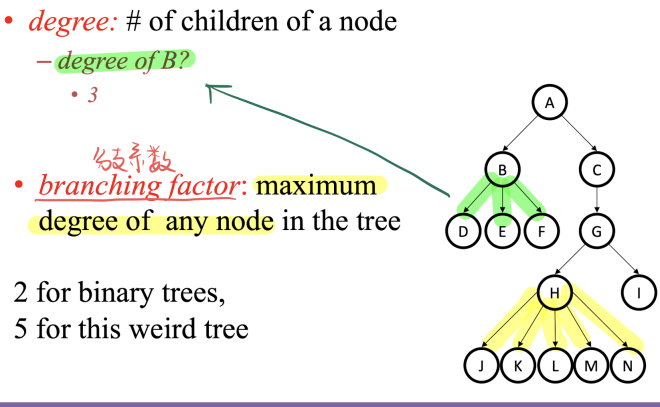
min\_index = j

arr[i], arr[min\_index] = arr[min\_index], arr[i]

return arr

**Trees**

 ****

** **

**complete:** 完整的二叉树，就是所有层的节点都被填满

**nearly complete:** 除了最后一层，所有层的节点都被填满，并且最后一层是从左到右填满的

搜索树（**Search Tree**）具有以下特性：1.左子树中的所有键值小于根节点的键值。2.右子树中的所有键值大于根节点的键值。

the running time of the find function in BSTs is **O(height)**

只有在相当平衡的情况下，使用BST才是有效的。BSTs是否平衡，在很大程度上取决于被添加的值的顺序。

successor（继承者）：对于给定节点，后继节点是在搜索树中大于该节点值的最小节点。

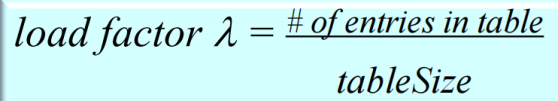
Predecessor（前驱）：对于给定节点，前驱节点是在搜索树中小于该节点值的最大节点。

Build Tree：Worst case: O(n2) Average case assuming all orderings equally likely turns out to be O(n lg n).

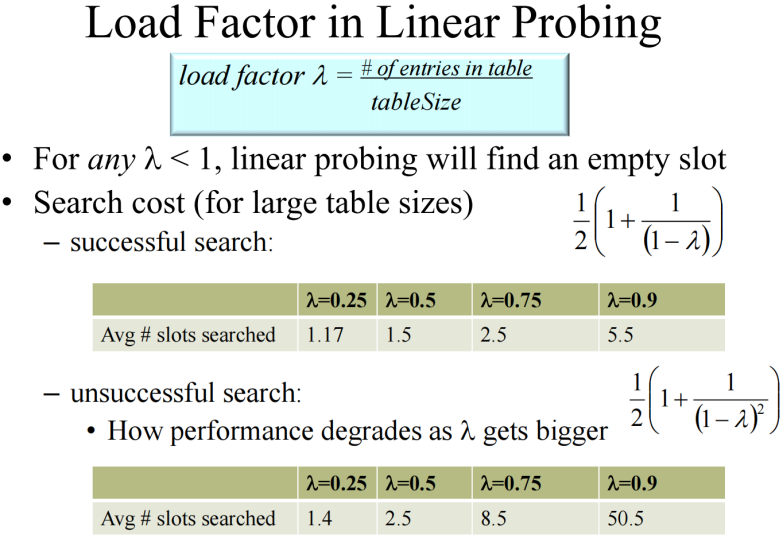
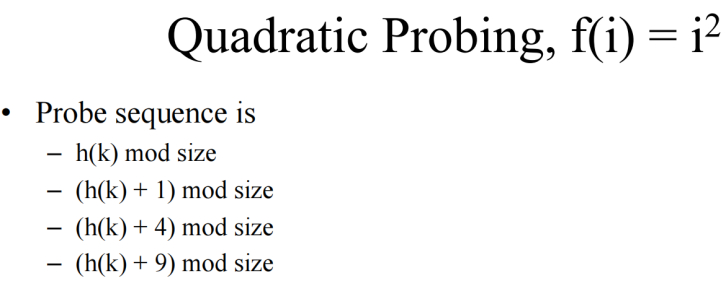
**Hash Function**

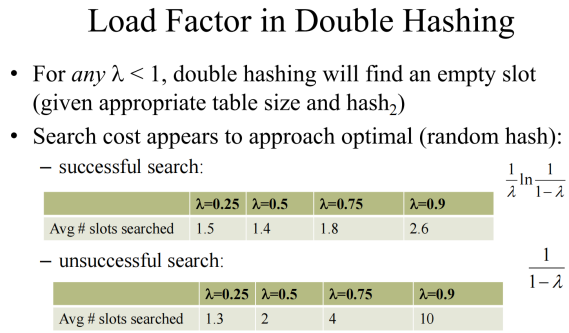
Goals of a hash function： 1. reproducible mapping from key to table entry 2. evenly distribute keys across the table

1. separate commonly occurring keys (neighbouring keys?) 4. complete quickly

 λ 可以大于1，但性能随着链式的长度增加而减小

**Search cost：**不成功的cost（on average λ ), 成功的cost（on average λ/2 )，期望的就是1/2到1。

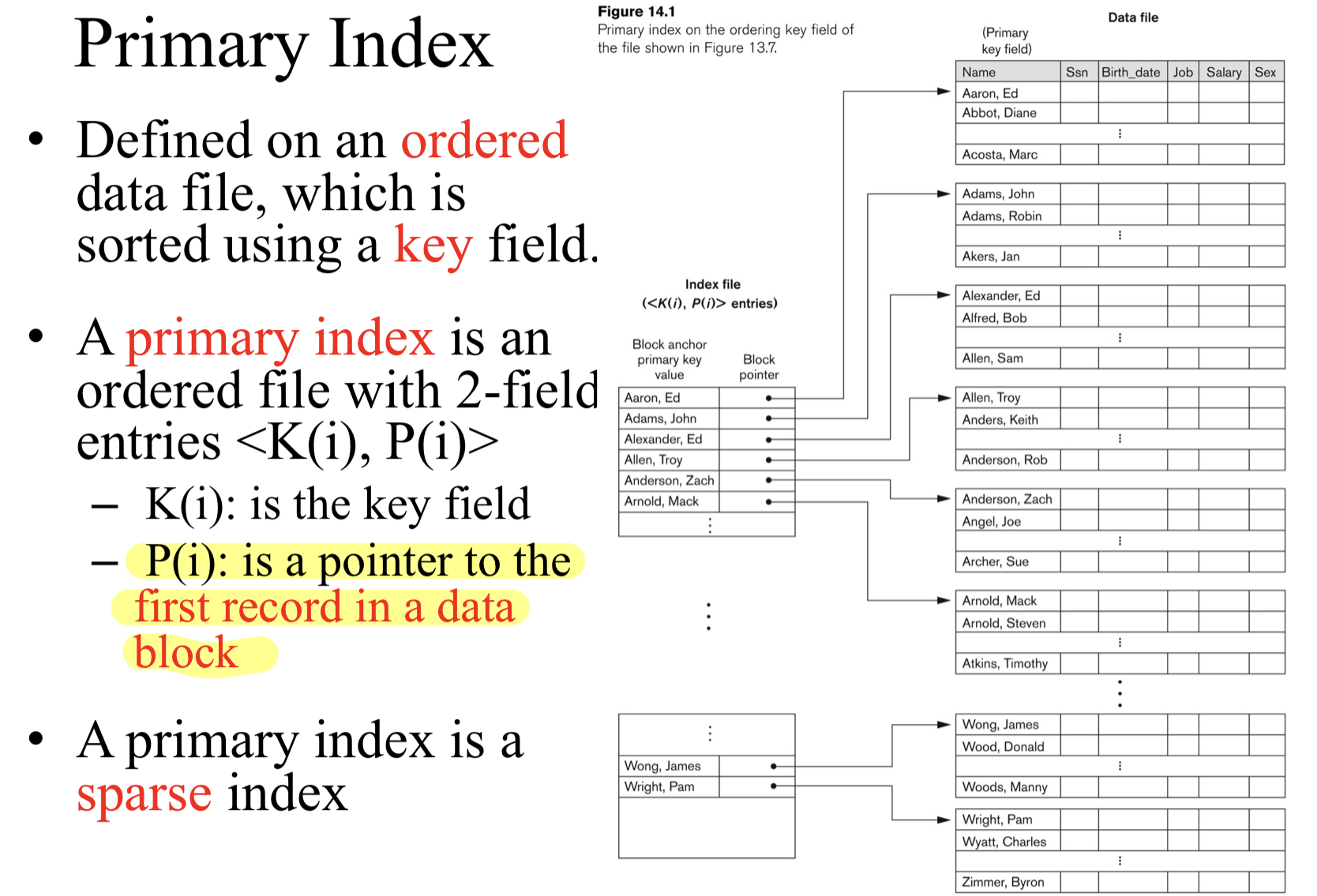
 

**Indexing:**

– A dense index(稠密索引) has an index entry for every search key value (and hence every record) in the data file.

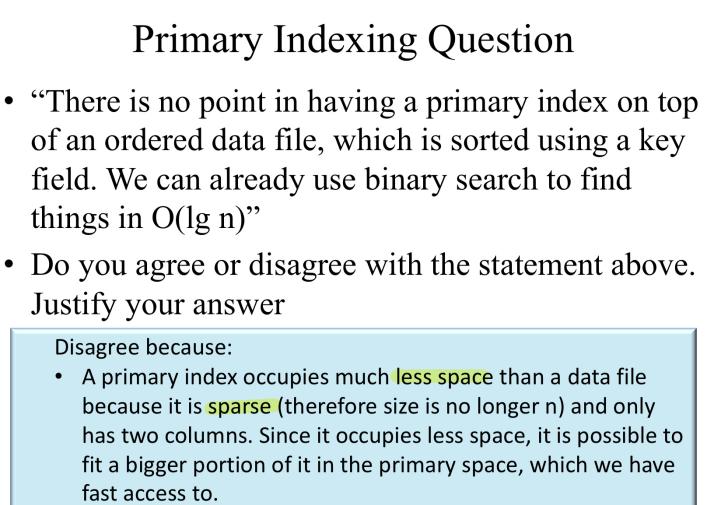
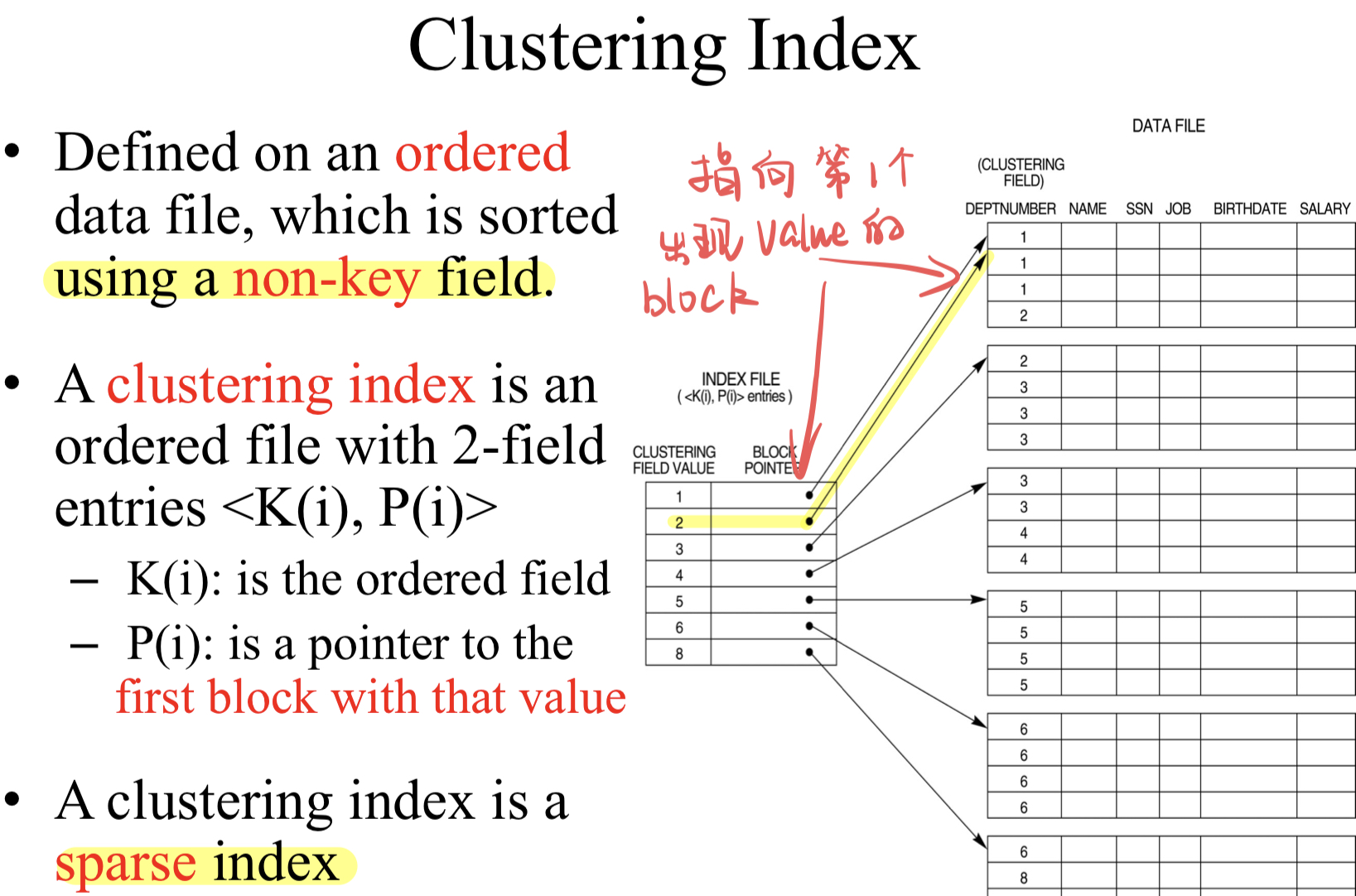
– A sparse (or nondense) index(稀疏索引) has index entries for only some of the search values

A **primary index** occupies much less space than a data file because:

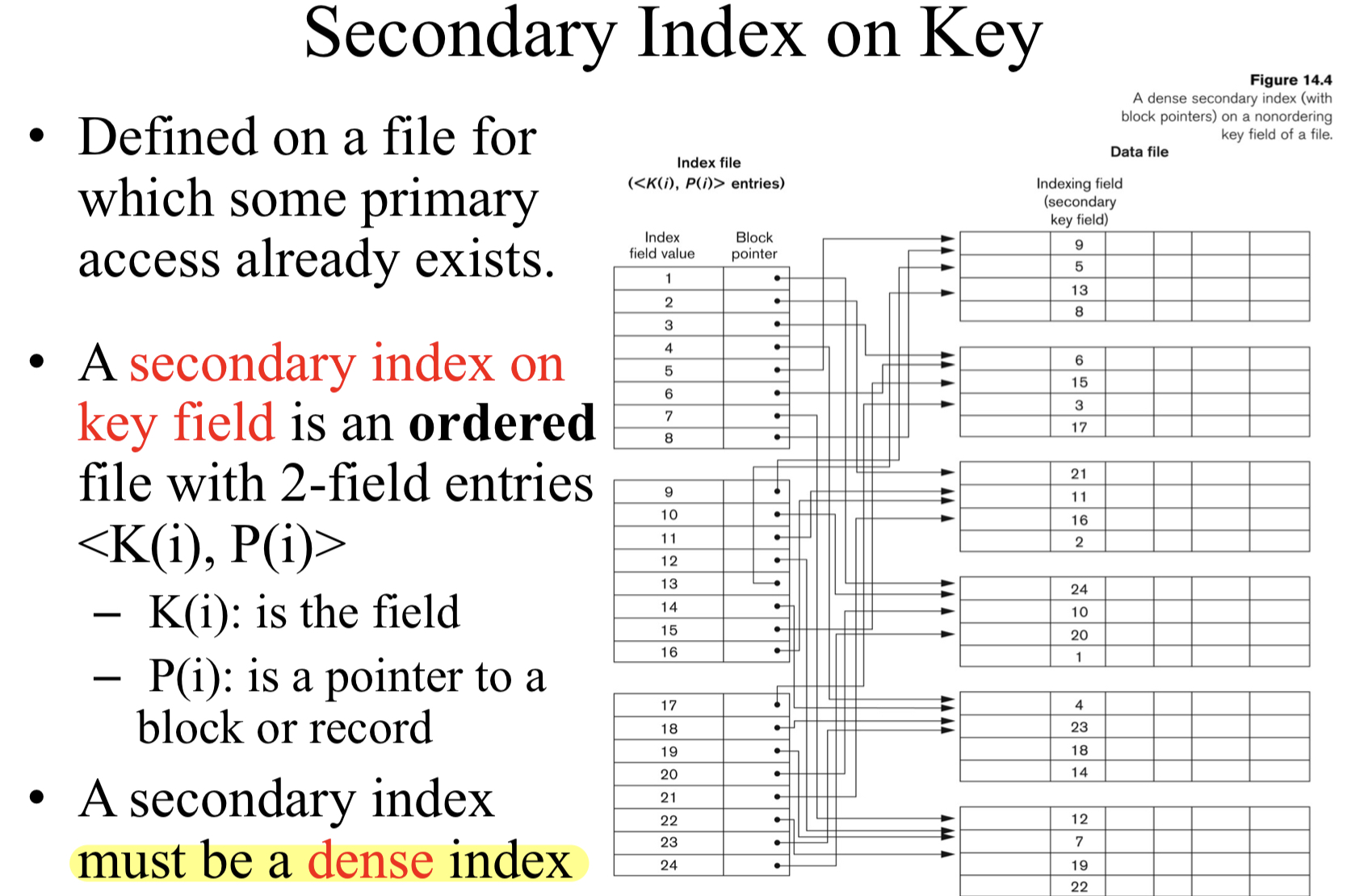
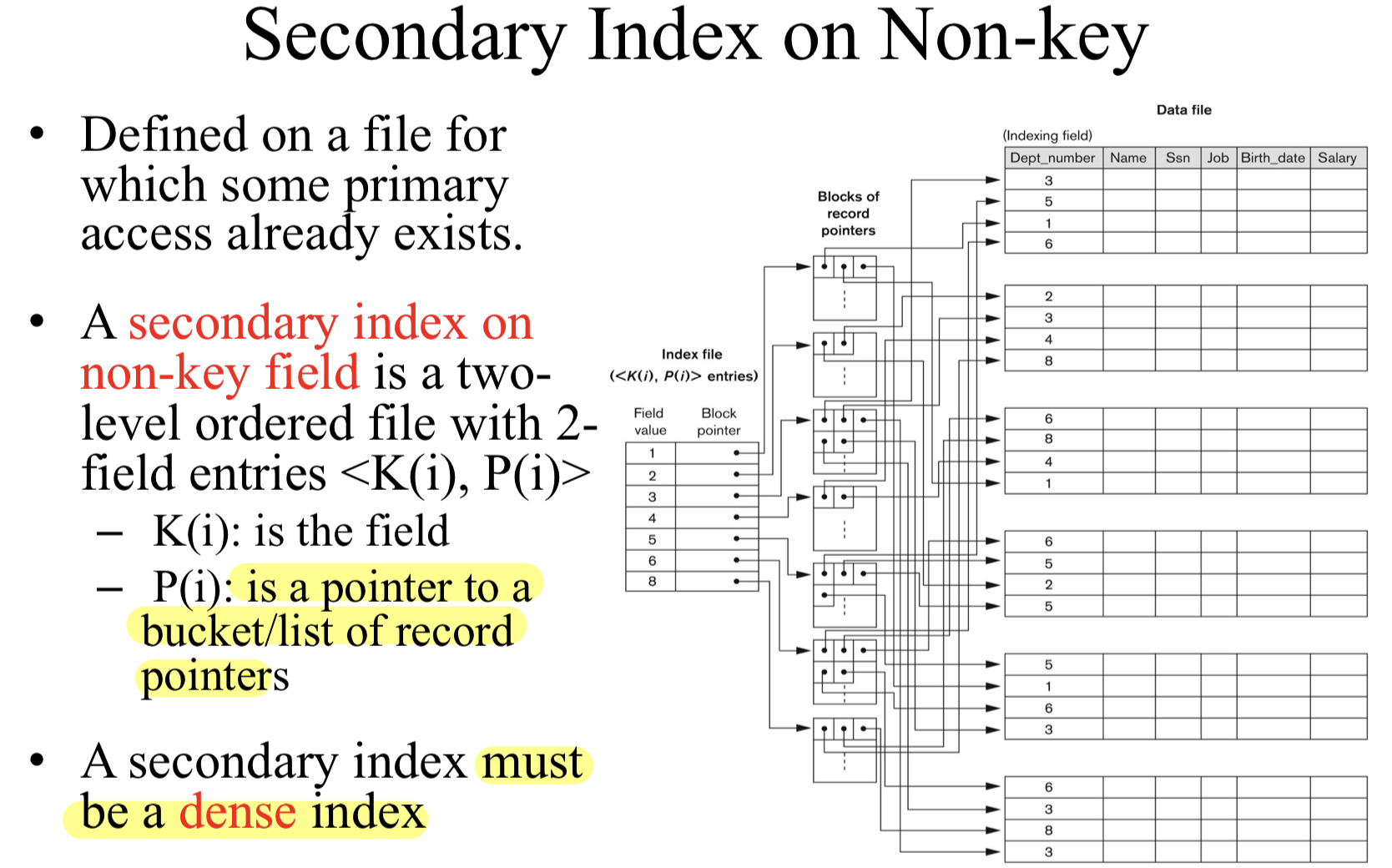
– It is sparse, so few index entries – Each entry has only two columns

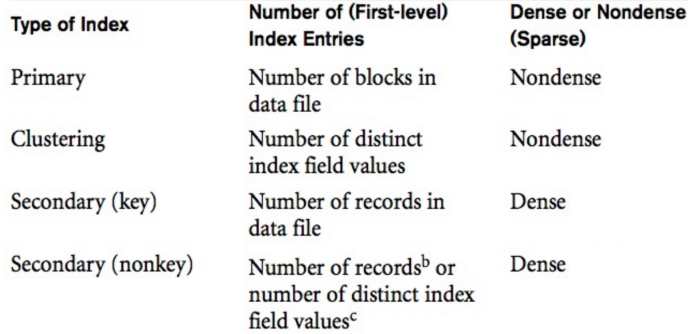
Even tough both will be using Binary Search, searching a primary index is much faster than search the data file.

Clustering Index的潜在问题：Since records are ordered, insertion and deletion cause problems,To overcome the problem of insertion: – Reserve blocks for each distinct value.

由于记录是有序的，插入和删除操作会带来问题。为了克服插入问题：- 为每个不同的值预留块。

non-key:secondary index(dense)

key: primary index(nondense)

equal: hash-based index

range: B+ tree

muti-values: grid-file

**Tree-structured indexes**树状索引是一种以树状结构组织搜索键的索引方法，

支持查找、插入和删除操作，并且能够同时支持**范围搜索(range searches)**和**相等性搜索(equality searches)**

B+ Trees:B+树是基于B树的一种索引结构，B树是对二叉搜索树的一种泛化。B+树是自平衡的，对于搜索、插入和删除操作的运行时间为对数级别（O(log n))。

优点：在插入和删除操作时，B+树可以自动进行小范围的本地调整，从而保持平衡。不需要对整个文件进行重新组织，即可保持性能。

1.Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions. 2.Reorganization of entire file is not required to maintain performance.

缺点：需要额外的插入和删除开销。需要更多的存储空间。1.Extra insertion and deletion overhead. 2.Space overhead

**Hashed-based Indexes**将搜索键和相关的记录指针组织成哈希文件结构。

哈希索引的特点包括：1.哈希索引通常是处理相**等性选择（equality selections）的最佳选择**。2.哈希索引无需遍历树结构，可以直接计算出记录所在的位置。3.哈希索引**无法**有效地支持**范围搜索（range searches）**