

Entity - Relationship Model 实体、关系模型

1. 数据库设计步骤:

- (1) requirement specification
- (2) conceptual - design (E-R diagram)
- (3) logical - design
- (4) physical - design

2. Entity 通过 Relationship 互相联系构成 E-R 图. (前者矩形, 后者菱形)

3. 另外: Relation Set 也可以自带属性. (加 Sets 就是具体的集合)

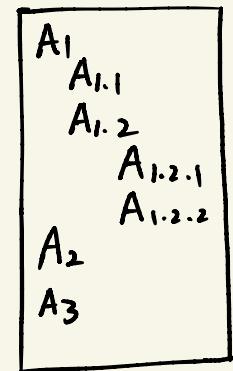
4. Role: 当一个 Entity 通过某 relation 关联自己时,

在 e 和 r 的连接线上会注明, 本 entity 之间分别扮演的角色

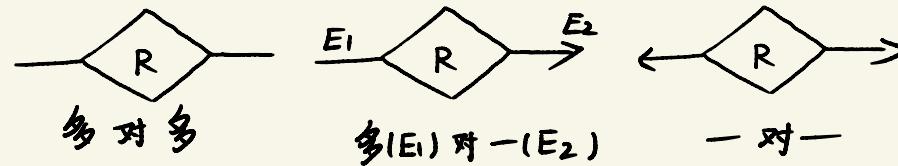
5. 一个 n 元联系, degree 是 n.

6. Attribute

simple / composite	\rightarrow 表示方法:
single-valued / multivalued	
derived attributes	



7. Mapping Cardinality Constraints: 映射基数约束

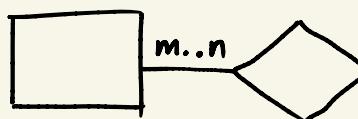


8. Total and Partial Participation

该 Entity 全部参与某 Relationship:

可以不全部参与某 R 的情况:

具体限制



双横线

单横线

对这个 Entity 来说
至少 m 个, 至多 n 个 关系与之相连
若无上限, 则用 * 表示

Figure 1:

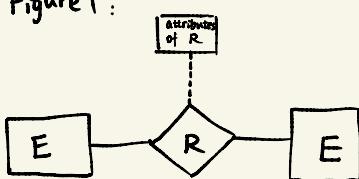
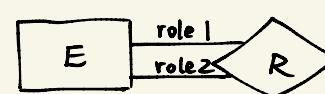


Figure 2:



每个 E2 都有 1 个 E1 与之对应, E1 可能有 E2, 也可能没有, E2 ⊆ E1

9. 在多元关系中，禁止出现多于1个的箭头。

10. Primary Key

(1) Primary Key for Entity Sets: 类似关系模型

(2) Primary Key for Relationship Sets:

R本身的所有自带属性，外加上R所连着的所有E的 primary key 中的部分。
具体而言，对二元关系来说：

多对多关系：所有E的 primary key 取并

(多/一) 对一关系：以“-”部分Entity 的 primary key 作主键

Weak Entity: 弱实体集 (没有 primary key 的实体集)

依赖于 identifying entity set (标识性实体集) 存在，
且是 多 (weak) 对一 (identify) 的关系，

Identifying Relationship 用“双菱形”表示。

Discriminator (分辨符, aka: partial key),

类似于 Entity 的 primary key, 一般用虚线划出。



E-R 图转化为关系模式图：

1. Strong Entity Set: 转化为单独的一个 schema, 属性和 Entity 一样, 主键相同。

2. Weak Entity Set: 转化为单独的一个 schema, 属性是 Weak Entity 的属性, 外加上标识集合的主键。

新表的主键 = 标识集合主键 + 弱实体集的分辨符。

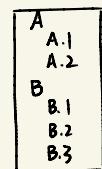
3. 多对多关系，需转化为一个单独的关系模式！

该 schema 的 attributes 和其主键相同，都是所有 E 的主键外加 R 的属性

4. 多对一的关系：无需转化为单独的关系，直接在“多”的那张表格上加上“一”的主键。

5. 一对一的关系：无需转化为单独的关系，在任意一张“一”的表格上加上另一张表格的主键。

对 Composite Attributes 的处理：平铺。



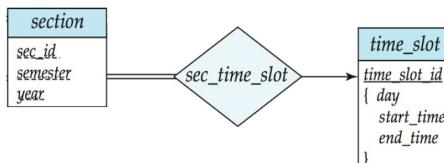
T(A.1, A.2, B.1, B.2, B.3)

对 Multivalued Attributes 的处理：

- A multivalued attribute M of an entity E is represented by a separate schema EM
 - Schema EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
- Example: Multivalued attribute `phone_number` of `instructor` is represented by a schema:
 $\text{inst_phone} = (\text{ID}, \text{phone_number})$
- Each value of the multivalued attribute maps to a separate tuple of the relation on schema EM
 - For example, an `instructor` entity with primary key 22222 and phone numbers 456-7890 and 123-4567 maps to two tuples:
(22222, 456-7890)
(22222, 123-4567)

特殊情况

- Special case: entity `time_slot` has only one attribute other than the primary-key attribute, and that attribute is multivalued
 - `time_slot(time_slot_id)`
`time_slot_detail(time_slot_id, day, start_time, end_time)`
 - Optimization: Don't create the relation corresponding to the entity, just create the one corresponding to the multivalued attribute
`time_slot(time_slot_id, day, start_time, end_time)`
 - Caveat (警告) : `time_slot` attribute of `section` (from `sec_time_slot`) cannot be a foreign key due to this optimization



新建 schema.

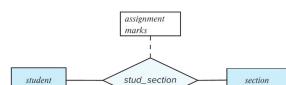
信息冗余，
不同 Entity 间不能有
相同的 Attributes.

UML :

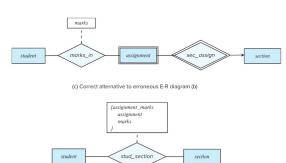
Unified Modeling Language

• 关系属性使用不当

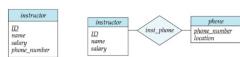
这里一门课可能有很多次作业，不能只用一个实体。



解决方法：



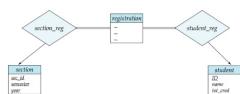
4.2 Use of entity sets vs. attributes



- 第一种方法，明确放一个电话号码。
- 第二种方法，电话号码可以附属更多属性，一个电话号码可以由多人共享。（如办公室的公共电话）

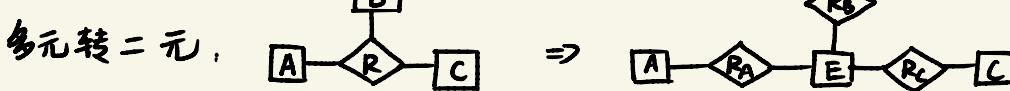
4.3 Use of entity sets vs. relationship sets

Possible guideline is to designate a relationship set to describe an action that occurs between entities.



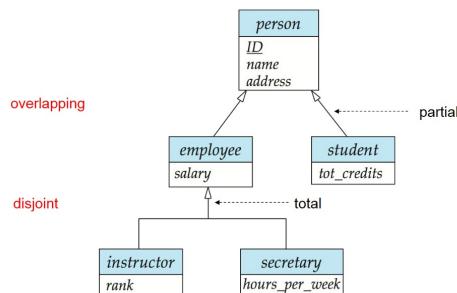
实体可以便于与其他实体建立联系。

如电商，我们可以简单的把客户和商品用 buy 联系起来，但后续还会有付款、物流等情况，我们最好把 buy 实体化为订单。



Extended E-R Features :

Specialization 特化：Top-down, Attribute Inheritance
 Generalization 概化：Bottom-up.



{ overlapping, 重叠, person 可以同时是 employee 和 student }

{ disjoint, 不相交 }

{ total, 全部 employee 属于 instructor 或 secretary }

{ partial, 部分 person 属于 employee 或 student }

Representing Specialization via Schemas

Method 1:

- Form a schema for the higher-level entity
- Form a schema for each lower-level entity set, include primary key of higher-level entity set and local attributes

schema	attributes
person	ID, name, street, city
student	ID, tot_cred
employee	ID, salary

Drawback: getting information about an employee requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema

Method 2:

- Form a schema for each entity set with all local and inherited attributes

schema	attributes
person	ID, name, street, city
student	ID, name, street, city, tot_cred
employee	ID, name, street, city, salary

Drawback: name, street and city may be stored redundantly for people who are both students and employees

Method 3:

- Using single schema for all super /suber entity sets , with a type attribute to differentiate entities.

schema	attributes
person	ID, name, street, city, person_type , tot_cred, salary

Relational Database Design

好的表(没有数据冗余)的特征：只有 candidate key 能决定其它元素

1. Lossless-join decomposition 无损分解。将 Γ 分解为 R_1, R_2

定义： $R_1 \bowtie R_2 = \Gamma$ (自然连接后信息无损失)

充分条件： R_1, R_2 的公共属性是 R_1 或 R_2 的 key.

$$\nabla R_1 \cap R_2 \rightarrow R_1 \text{ 或 } R_1 \cap R_2 \rightarrow R_2$$

2. Functional Dependency 函数依赖

定义： $\alpha \rightarrow \beta$ 表示只要 α 相同， β 一定相同。

(只能追伤，不能跑实：若 $\alpha_1 = \alpha_2, \beta_1 \neq \beta_2$, 非函数依赖；
但如果 $\alpha_1 = \alpha_2, \beta_1 = \beta_2$, 并不一定是函数依赖。)

充要条件： $K \rightarrow R \Leftrightarrow K$ 是 R 的 superkey

$$K \rightarrow R \text{ and no } \alpha \subset K, \alpha \rightarrow R \Leftrightarrow K \text{ 是 } R \text{ 的 Candidate Key.}$$

3. trivial 平凡的。(理解为这种函数依赖是废话)

定义： $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.

4. Closure of a set of Functional Dependencies.

定义： F^+ 是 F 所能推出的所有函数依赖的集合。

Eg. $F = \{A \rightarrow B\}$, 则 $F^+ = \{A \rightarrow A, A \rightarrow B, A \rightarrow AB, B \rightarrow B, AB \rightarrow A, AB \rightarrow B, AB \rightarrow AB\}$

n 个变量的函数依赖集合，其闭包的左侧就有 $2^n - 1$ 种存在。

F^+ 的求法见 Attribute Closure 作用③。

Armstrong's Axioms:

① reflexivity 自反率： $\beta \subseteq \alpha \Rightarrow \alpha \rightarrow \beta$

② augmentation 增补率： $\alpha \rightarrow \beta \Rightarrow r\alpha \rightarrow r\beta$

③ transitivity 传递率： $\alpha \rightarrow \beta, \beta \rightarrow \gamma \Rightarrow \alpha \rightarrow \gamma$

Sound (有效)

Complete (完备)

Additional Rules.

- ① union (合集) $\alpha \rightarrow \beta, \alpha \rightarrow r \Rightarrow \alpha \rightarrow \beta r$
- ② decomposition (分解) $\alpha \rightarrow \beta r \Rightarrow \alpha \rightarrow \beta, \alpha \rightarrow r$
- ③ pseudotransitivity (伪传递) $\alpha \rightarrow \beta, \gamma\beta \rightarrow \delta \Rightarrow \alpha\gamma \rightarrow \delta$

5. Closure of Attributes

定义: $A^+ = \{ \text{所有由 } A \text{ 决定的属性集合} \}$

作用: ① 检验 superkey. 若 $\alpha^+ = \text{all attributes}$, 则 α 是超键。

② 检验 函数依赖: 若 $\beta \subseteq \alpha^+$, 则 $\alpha \rightarrow \beta$

③ 计算 F^+ . I) 对所有 $\alpha \in R$, 计算 α^+ ;

II) 对所有 $S \subseteq \alpha^+$, 有函数依赖 $\alpha \rightarrow S$.

b. Extraneous Attributes (无关属性) $ABC \rightarrow DEF$

定义: 删除某条函数依赖 f 中的属性 P.

若删除 P 后, f 左侧依然能推出右侧, 则 P 多余。(“推出”具体表现为同包)

7. Canonical Cover 正则覆盖

定义: 是能推出 F^+ 的最小函数依赖集合 F_C , (F_C 中任何函数依赖不含无关属性且 F_C 函数依赖的左半部分均唯一且不重复)。

计算方法: loop { ① 找合项: $\alpha \rightarrow \beta, \alpha \rightarrow r$ 合并为 $\alpha \rightarrow \beta r$ }
② 删除无关属性。

更实用的方法: 画图, 去掉无意义的箭头。

8. BCNF, Boyce-Codd Normal Form

定义: R is in BCNF with respect to a set F

if: $\forall (\alpha \rightarrow \beta) \in F^+$ 都满足: α 是 superkey (或 $\beta \sqsubseteq \alpha$ (trivial))

一般直接用 F 即可。

算法: 不断把左侧不是 key 的函数依赖分解出来。

loop { if $\exists (\alpha \rightarrow \beta)$ 是非平凡且 α 不是 key, 则 $R = (\alpha \cup \beta) + \underline{(R - (\beta - \alpha))}$ }

在 β 与 α 没有重复属性的情况下: $R = (\alpha \cup \beta) + (R - \beta)$

9. Dependence Preservation 保持依赖

定义: 分解前的任意函数依赖, 都可以通过分解后的函数依赖推出.

* 理解为函数依赖的无损分解.

充要条件: $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$

将 R 分解为 BCNF 的过程无法确保一定是 依赖保障的! 但一定可以是 lossless 的

10. 3NF, Third Normal Form → 保持 lossless-join、dependency preserving.

定义: \forall 函数依赖 $\alpha \rightarrow \beta$, 要么平凡, 要么 α 是超键,

要么 $(\beta - \alpha)$ 的所有属性都是 candidate key.

3NF 可以保持函数依赖!

算法: ① 求出 F 的 F_c ② 对所有 $(\alpha \rightarrow \beta) \in F_c$, 新增一个关系 $R_i = \alpha\beta$

③ 如果所有 R_i 中, 由不包含 R 的 candidate key, 则新增 $R_i = \text{any candidate key of } R$

④ 删除所有冗余的关系 R_i (if $R_j \sqsubseteq R_k$, then delete R_j)

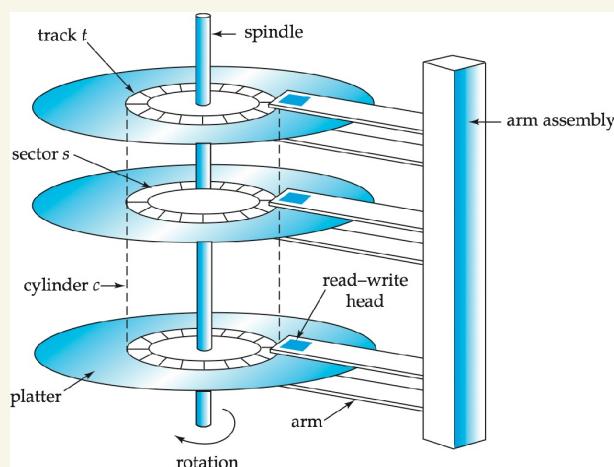
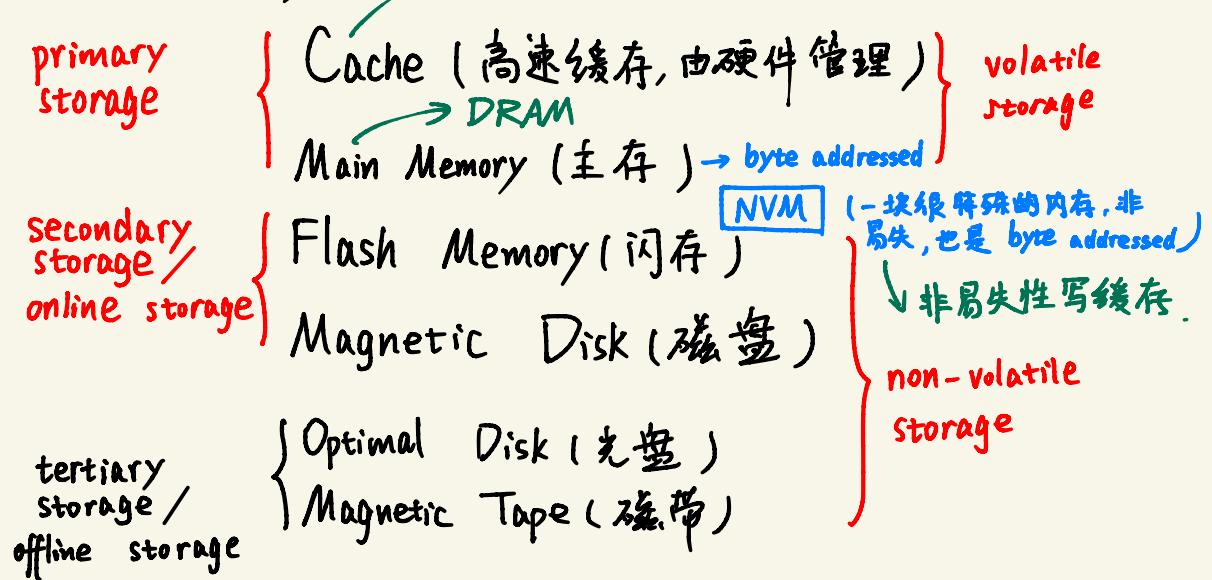
对不“好”的关系, 要把它们分解为“好”的关系, 且保持 函数依赖、信息无损。

(分解成 BCNF 或 3NF)

Physical Storage Systems

可以分成 voltage storage 和 non-voltage storage (易失 / 非易失)
从 speed . cost . reliability 衡量.

Memory Hierarchy:



一张磁盘会有 50K~100K 的磁道，
半径小的磁道约有 500~1000 帧区，
半径小的磁道约有 1000~2000 帧区。
一个 帧区一般是 512 bytes.

磁盘控制器，Disk Controller

计算机与实际磁盘硬件间的接口，一般在磁盘驱动单元内实现。
Disk Controller 为它所写的每个扇区附加“校验和”checksum (根据写入数据计算得出)。读回数据时，再计算 checksum，如果不一致说明 data 被破坏。还会执行 remapping of bad sector，检测到坏块扇区就 remap。

磁盘性能度量

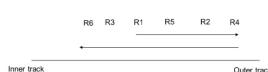
1. 访问时间 access time $\left\{ \begin{array}{l} \text{寻道时间 seek time (4~10 ms)} \\ \text{旋转延迟 rotation latency (4~11 ms)} \end{array} \right.$
 $\text{Average} = \frac{1}{2} \text{ worst}$
2. Data-Transfer Rate (数据传输率) → 以 Block 为单位
最大 $50\text{MB} \sim 200\text{MB/s}$, 内侧磁道传输率明显低于外侧。
3. IOPS (每秒 I/O 操作): 每秒随机访问块的数量
前置知识: ① Disk 是 Block Addressed. ($50 \sim 200$)
每个 Block $4\text{KB} \sim 16\text{KB}$ 。(也就是说包含几个扇区)
② 两种访问模式:
 - sequential access, 连续的请求访问连续的块, 可能位于相同或相邻磁道。
 - random access
4. MTTF (Mean Time To Failure) 平均故障时间

$$\text{MTBF} = \text{MTTF} + \text{MTTR} \quad (\text{Mean Time Between Failure}) \quad (\text{Mean Time to Replacement})$$
$$\text{Reliability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

优化磁盘访问的方法:

2.2 Optimization of Disk-Block Access 1

- Buffering: in-memory buffer to cache disk blocks
数据读进来就丢，比较可惜，所以我们把它放在一个地方，万一后面需要使用可以不用再读。
- Read-ahead(Prefetch): Read extra blocks from a track in anticipation that they will be requested soon
预取，读某块时预测邻近几块也会被访问，于是就一起取到内存中。要有依据地预取，不然无用的数据会占用缓存。
- Disk-arm-scheduling algorithms re-order block requests so that disk arm movement is minimized
elevator algorithm



• File organization

- Allocate blocks of a file in as contiguous a manner as possible
预先分配得到的内存是连续的
- Files may get fragmented
 - Sequential access to a fragmented file results in increased disk arm movement
 - Some systems have utilities to defragment the file system, in order to speed up file access
- Nonvolatile write buffers (非易失性写缓存)
speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
把要写的数据先写到一个快速的非易失的缓存里，如 NVM。这时上面的程序可以继续执行了，NVM 再择机将数据写回到磁盘。
- Log disk (日志磁盘)
a disk devoted to writing a sequential log of block updates

闪存被广泛应用于相机、手机之中。

Flash

NOR	}	read: 一次读 1 page. (类似于磁盘的sector)
NAND <small>更便宜</small>		

write: 只能写一次！需被擦除后才能重写。

固态硬盘 SSD: 内部用闪存 NAND 实现，但是提供与 Disk 相同的面向块 Block 的接口。

	Magnetic Disk	Solid State Disk
Retrieve a page	5-10 milliseconds	20-100 microseconds
Random access	Random 50 to 200 IOPS	reads: 10,000 IOPS writes: 40,000 IOPS
Data transfer rate	200M	500M(SATA), 3G(NVMe)
Power consumption	higher	lower
Update mode	in place	erase → rewrite
Reliability	MTTF: 500,000 to 1,200,000 hours	erase blocks: 100,000 to 1,000,000 erases

SSD 更快 (微秒级)
Disk (毫秒)
SSD 的 IOPS 更高。
传输更快。

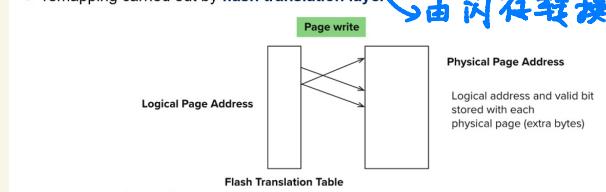
Erase 操作：

以 Block 为单位，每个 Block 由 pages 组成。(256KB~1MB, 128~256page)
(Disk 的 Block 由 sectors 组成) 擦 $10^5 \sim 10^6$ 次易坏

具体实现：Remapping Logical Page Address to Physical Page Address to avoid waiting for erase.

这一映射被记录在 Flash Translation Table 中。

- also stored in a label field of flash page 存放在 flash 中。
- remapping carried out by flash translation layer → 由闪存转换层执行。



Wearing Level：优先更新少被更新的数据，减少磨损。

(OS 与 Disk 打交道的最小单位
→ Block Addressed, 以 Block 为单位读写)

对 Disk 来说：

物理层面：disk → track → sector, 磁盘读写基本单位

逻辑层面：block (虚拟而得)，最小逻辑储存单位

Dada Storage Structures

File : A sequence of Records

Record : A sequence of Fields.

Fixed - Length Records .

第*i*条 record 的 byte : $n \cdot (i-1)$, 其中 n 是每条 record 的大小。

有两个问题：


nivasan	Comp. Sci.	65000
zart	Music	40000
nstein	Physics	95000
old	Physics	87000

 Block 不一定是 n 的整数倍 \Rightarrow 询问一个 Record 可能要跨块！ Solution: 在一个 Block 中仅分配能完整容纳的 Record 数目，舍弃多余字节。
 ① 删除麻烦。

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	15151	Mozart	Music	40000
record 2	22222	Einstein	Physics	95000
record 3	33456	Gold	Physics	87000
record 4	58583	Califieri	History	62000
record 5	76543	Singh	Finance	80000
record 6	76766	Crick	Biology	72000
record 7	83821	Brandt	Comput. Sci.	92000
record 8	98345	Kim	Elec. Eng.	80000

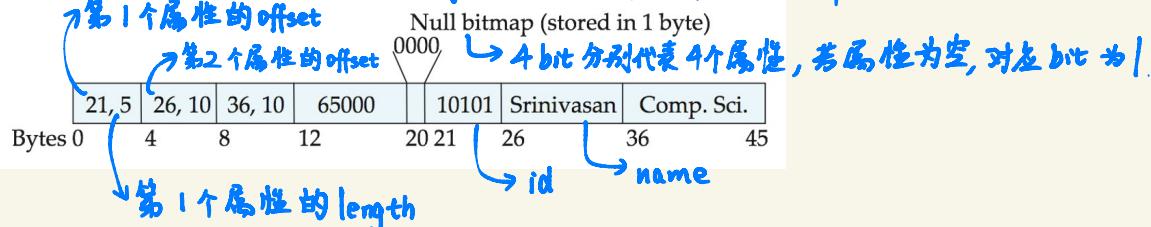
Solution 1: 移动后续所有 record X 开销大

Solution 2. 把最后一条 record 移上来 X 可能溢出

Solution 3: 由于插入比删除更频繁，先添加 file header(文件头)
标记所有已删除的 record，形成 free list (自由链表)。每次插入时，
先往自由链表里插，满了才会插到文件末尾。

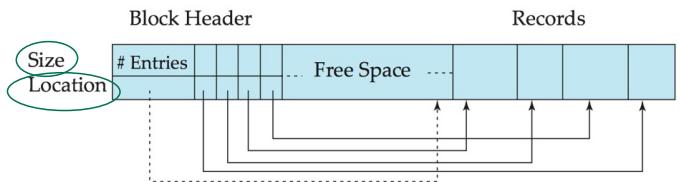
Variable - Length Records 变长记录

Instructor(id, name, dept_name, salary) 前3个不定长，第四个定长8字节



不走长的需记录 (offset, length)

Slotted Page Structure



这里的 page 和 Block 一个含义

从末尾处开始连续分配空间

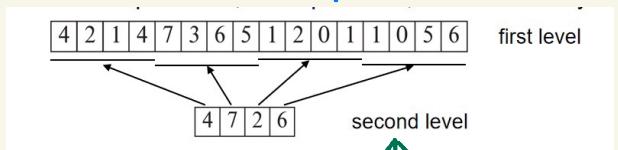
* Record Pointer 需指向 Head 中的 entry, 而非直接指向 Record.

- Slotted page (分槽页) header contains: “以” 包括,
 - number of record entries **records 数量**
 - end of free space in the block **Free Space 末端指针**
 - location and size of each record **每条记录的 (offset, length) 对, 也称为该 record 的 entry.**

Records Organization

Heap : 记录可以存储在任意位置, 且通常不会被移动.

Free-Space Map : 一个数组, 记录每个块的自由空间比例



比如数组每一次有 3 bits. ($\frac{2^3}{2^8}$)
Free-map 中的数据越大, 就越空

还可以使用多层 Free-Space Map

取第1层中的最大数据.

Sequential File : records 按照某一 search key 排列.
适合于需要 sequential processing whole file 的.

插入 : freelist / overflow block

溢出块的使用.

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
 - if there is free space insert there
 - if no free space, insert the record in an overflow block
 - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order

10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp. Sci.	92000	
98345	Kim	Elec. Eng.	80000	

32222	Verdi	Music	48000	
-------	-------	-------	-------	--

时不时重排文件.

Multiple Clustering File Organization 多表聚簇文件组织

- Store several relations in one file using a **multitable clustering** file organization

	<i>dept_name</i>	<i>building</i>	<i>budget</i>	
<i>department</i>	Comp. Sci. Physics	Taylor Watson	100000 70000	
<i>instructor</i>	<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
	10101 33456 45565 83821	Srinivasan Gold Katz Brandt	Comp. Sci. Physics Comp. Sci. Comp. Sci.	65000 87000 75000 92000
multitable clustering of <i>department</i> and <i>instructor</i>				
	Comp. Sci. 45564 10101 83821 Physics 33456	Taylor Katz Srinivasan Brandt Watson	100000 75000 65000 92000 70000	87000

- good for queries involving **join** *department* \bowtie *instructor*, and for queries involving one single department and its instructors
- bad for queries involving only *department*
- results in variable size records
- Can add pointer chains to link records of a particular relation

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	

适用于有 Join 的情况。
单独的属性不适合多表聚簇。

Table Partitioning : 会根据某一属性的值将一个关系中的记录划分为更小的关系。

Ex : 可根据 year 将表 T 分成 T-2018 , T-2019 .
则一般情况下对表做操作时, 需要同时调用
T-2018 , T-2019 , 除非有条件限制 year = 2018 / year = 2019 .

Data Dictionary Storage (System Catalog)

↳ Store Metadata : data about data.

- Information about **relations**
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
 - number of tuples in each relation
- Physical file organization information
 - How relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices

Storage Access

Block : Data Allocation / Transfer 的基本单位.

Buffer Management, LRU 方案 (Least Recently Used)

当 Buffer 满的时候, 移出最近最少使用的.

Access	1	4	8	1	5	2	3	2	4
Buffer	1	4	8	1	5	2	3	2	4
	1	1	1	4	4	8	1	5	1
	1	1	1	4	4	8	1	5	1
	1	1	1	4	4	8	1	5	1

4 Buffer Pages.

Replacement happens 3 times

Pinned Block : 不被允许写回到 Disk 上的 Memory Blocks

Pin : Before reading / writing.

Unpin : When reading / writing is complete.

记录 pin count. 当 pincount = 0 时, Block 可被 evicted.

■ Shared and exclusive locks on buffer

- Needed to prevent concurrent operations from reading page contents as they are moved/reorganized, and to ensure only one move/reorganize at a time
- Readers get **shared lock**, updates to a block require **exclusive lock**
- Locking rules:**
 - Only one process can get exclusive lock at a time
 - Shared lock cannot be concurrently with exclusive lock
 - Multiple processes may be given shared lock concurrently

■ LRU can be a bad strategy for certain access patterns involving repeated scans of data

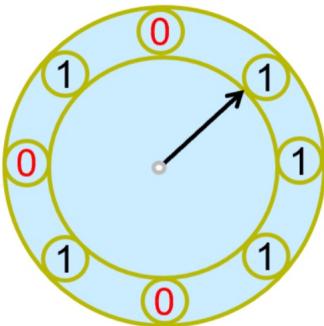
- For example: when computing the **join** of 2 relations r and s by a nested loops
 - for each tuple *tr* of *r* do
 - for each tuple *ts* of *s* do
 - if the tuples *tr* and *ts* match ...

Toss-immediate Strategy : 某个 Block 里的所有数据一处理完, 立即释放空间。

MRU : 在处理的 Block 就 Pinned, 一处理完就 Unpinned → most recently used

Forced Output, 为保证磁盘上的数据一致性, 把 Block 强制写回 Disk.

有时候我们用 Clock 来近似实现 LRU



Arrange block into a cycle, store one reference_bit per block

When pin_count reduces to 0, set reference_bit = 1
reference_bit as the 2nd chance bit

被访问后，置 1

```

do for each block in cycle {
    if (reference_bit == 1)
        set reference_bit=0;
    else if (reference_bit == 0)
        choose this block for replacement;
} until a page is chosen;

```

开销小，但不是精确的LRU.

Column - Oriented Storage : 把某一关系的不同属性分别存储，也叫 Columnar Representation (不以行 (row/record) 为存储 unit)

■ Benefits: 排列存储的优点

- Reduced IO if only some attributes are accessed
- Improved CPU cache performance
- Improved compression
- Vector processing on modern CPU architectures

■ Drawbacks

- Cost of tuple reconstruction from columnar representation
- Cost of tuple deletion and update
- Cost of decompression

■ Columnar representation found to be more efficient for **decision support** than row-oriented representation

■ Traditional **row-oriented representation** preferable for transaction processing

■ Some databases support both representations

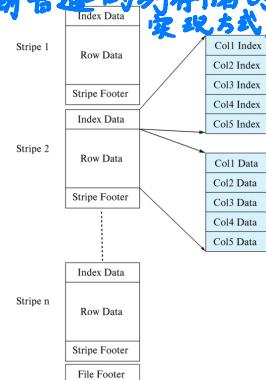
- Called **hybrid row/column stores**

ORC、Parquet 是目前普遍的列存储的实现方式

■ ORC(Optimized Row Columnar) and Parquet

- File formats of Hadoop HDFS
- **Apache ORC** is an open source, self-describing type-aware columnar file format designed for Hadoop workloads.
- **Apache Parquet** is an open source, column-oriented data file format designed for efficient data storage and retrieval.
- File formats with columnar storage inside file
- Very popular for big-data applications

■ Orc file format shown on right:



Indexing 索引

Index File 由以下结构的 records (称为 index entries) 构成:

search-key	pointer
------------	---------

(search-key 是某些 attributes)

Index Evaluation Metrics (评估指标)

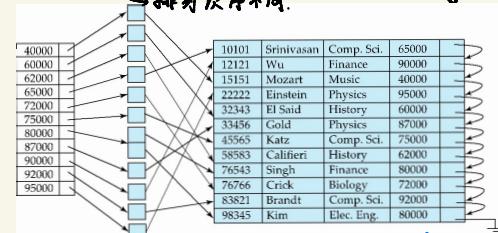
Access type、Access time、insertion time、
Deletion time、Space overhead。

↓ Access Type 的两种分类:

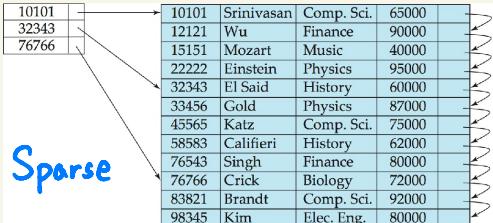
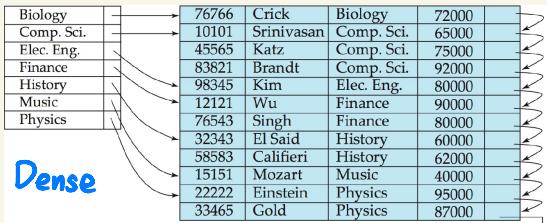
- Point query: records with a specified value in the attribute
- Range query: or records with an attribute value falling in a specified range of values.

Ordered Index : 被索引的文件本身也被 sorted. (按属性A被排序)
(顺序索引) ↓
Index entries are sorted in search-key.

分为两类 { primary index: A = Search-Key (clustering index)
secondary index: A ≠ Search-Key (nonclustering index) ↓
排列次序相同
排列次序不同.

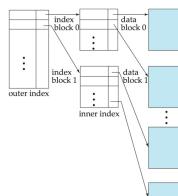


Index 也可分为 { Dense Index 稠密: 所有 search key 都在索引文件里.
Sparse Index 稀疏: 部分 search key 在索引文件里.



相比而言, 稠密索引更快, 稀疏索引空间更小、删插更快.

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

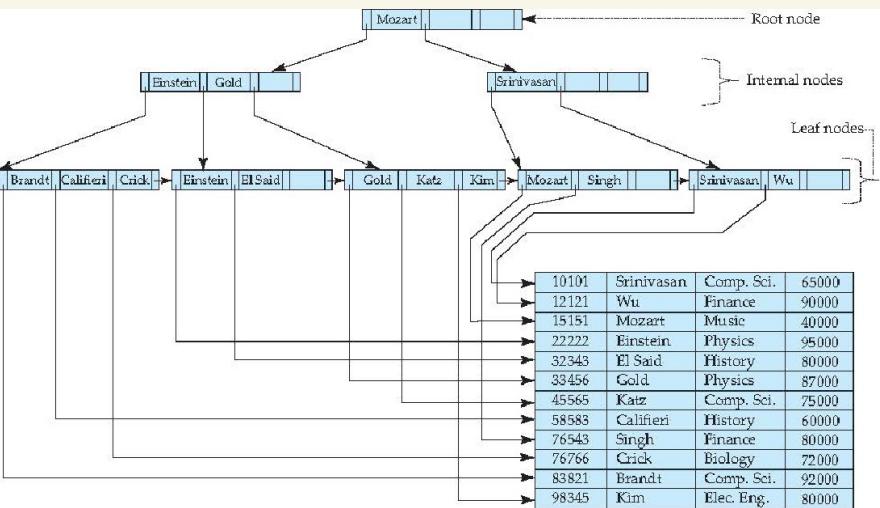


外层 sparse
内层 primary

Multilevel Index
多级索引

B⁺ Tree Index (一般会把 node 做成 Disk Block 的大小, 比如 4KB)

对于特定的 n, 根结点有 $\lceil \frac{n}{2} \rceil$ 个孩子, 非根非叶有 $\lceil \frac{n-1}{2} \rceil$ 个孩子.



$$P_1 | K_1 | P_2 | K_2 | \dots | P_n | K_n | P_{n+1}$$

↳ fanout (n) 的计算
 $n = \frac{\text{BlockSize} - 4}{\text{SearchKeySize} + 4} + 1$

其中 4 是一个指针的大小

对叶节点: $i=1, 2, \dots, n-1$: P_i 指向被索引文件中 search-key 值为 K_i 的 record.
 $i=n$: P_n 指向下一个叶节点
叶节点中最多 $n-1$ 个值, 最少 $\lceil \frac{n-1}{2} \rceil$ 个值.

对非叶节点 (也被称为 internal node):

若非 root, 则最多 n 个指针, 最少 $\lceil \frac{n}{2} \rceil$ 个指针。指针数称作 fanout.
对任意一个 P_i , 其指向子树的 search-key 要比 P_i 前的 K 大, P_i 后的 K 小。

对一棵 B⁺ 树来说, 如果有 K 个 search-key,

$$\text{则 } \text{height} \leq \lceil \log_{\lceil \frac{n}{2} \rceil} (K) \rceil$$

在最坏情况下, 插入 / 删除的 I/O 操作数 = $O(\log_{\lceil \frac{n}{2} \rceil} (K))$
若使用范围查询, 检索 M 个指针, 可能最多需要访问 $\lceil \frac{M}{n} \rceil + 1$ 个叶节点.

对B⁺树来说，按随机顺序插入， $\frac{2}{3}$ 是满的；有序插入， $\frac{1}{2}$ 是满的。

Non-Unique Search Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i, A_p) , which is unique
 - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- Extra storage overhead for keys
- Simpler code for insertion/deletion
- More I/O operations are needed to fetch the actual records
 - If the index is clustering, all accesses are sequential
 - If the index is non-clustering, each record access may need an I/O operation
- Widely used

非唯一性搜索码。

原始搜索码 + 额外属性
= 复合搜索码 $\xrightarrow{\text{uniquifier}}$

B+- tree : height and size estimation

- person(pid char(18) primary key,
- name char(8),
- age smallint,
- address char(40));
- block size : 4K
- 1000000 persons
- person record size = 18+8+2+40 = 68
- Records per block = 4096/68 = 60,235 \rightarrow 60
- blocks for storing 1M persons = $\lceil 1000000/60 \rceil$ = 16667
- B+ tree n(fan-out) = $(4096-4)/(18+4) + 1$ = 187
- $\lceil n/2 \rceil = 94$, inner pointers : 94~187
- $\lceil n-1/2 \rceil = 93$, leaf values : 93~186
- 2 levels: min=2*93 = 186 max= 187*186 = 34,782
- 3 levels: min=2*94*93 = 17484 max= 187*187*186 = 6,504,234
- 4 levels: min=2*94*94*93 = 1,643,496 max= 187*187*187*186 = 1,216,291,758

B⁺ Tree File Organization.

① Leaf nodes 放 records 本身, 而非 pointers。
(因为 records 本身比 pointer 大, 叶节点中 records 数量少于非叶中的 pointer)

② 可以提高至少半满的需求 (如. $\frac{2}{3}$ 满) 来提高空间利用率

Other Issues

Record relocation and secondary indices.

1. 记录迁移和次级索引:

当数据库中的一条记录被移动到别的位置时，所有指向这条记录的次级索引都需要更新。次级索引是指不基于主键建立的索引，它们帮助我们快速访问那些非主键字段。

2. B+树分裂:

B+树是一种常见的数据库索引结构，它帮助快速地查找和插入数据。当B+树的一个节点存储满了，它需要分裂成两个节点。这个分裂过程在有很多次级索引时会变得复杂且代价昂贵，因为每次分裂可能都需要更新很多索引指针。

3. 使用主索引搜索键代替记录指针:

为了减少因记录移动或节点分裂导致的次级索引更新，一种解决方案是在次级索引中使用记录的主索引搜索键，而不是直接存储指向记录的指针。这样，即使记录移动了位置，次级索引也无需更新，因为它引用的是不变的搜索键。

使用 primary index 的 search-key 替代 secondary index 的 pointer. → Queries 更贵
Node Splitting 便宜

Variable Length Strings as keys : Fanout 也会变动；
且节点分裂不再是根据 pointer 数量，而是 空间利用率。

■ Prefix compression

- Key values at internal nodes can be prefixes of full key
 - Keep enough characters to distinguish entries in the subtrees separated by the key value
 - E.g. "Silas" and "Silberschatz" can be separated by "Silb"

Bulk Loading : 批量装载. (开销大)

Solution 1: Insert in sorted order.

→ IO表现提升，但大多叶节点只是 half full (空间浪费)

Solution 2: Bottom-up B⁺-tree Construction

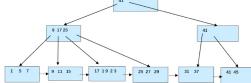
→ Create Tree Layer by Layer

Example: build a B+-tree for the following 16 entries, assuming the fan-out of the B+-tree is 4:

(23 25 27 29 1 5 7 9 11 13 31 37 41 45 15 17 19)
→ (1 5 7 9 11 15 17 19 23 25 27 29 31 37 41 45)

Estimated cost :

1 seek + 9 block transfers



如果要排序的内容较大，无法放下内存，可以使用外部排序。

fanout 可以计算出来。

可以用 level-order 写到磁盘里，便于顺序访问所有索引，此时块是连续的。（便于顺序访问所有数据项）

这里的代价就是建好后，一次 seek 后全部写出去 (9 blocks)

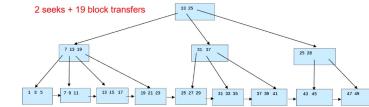
Example: insert following 9 entries into previous B+-tree

(21 33 35 39 43 47 49 3 13)

→ (1 3 13 21 33 35 39 43 47 49)

Estimated cost :

2 seeks + 19 block transfers



Merge two existing two B+-trees, to create a new B+-tree using the Bottom-UP Build algorithm, as in LSM-tree Index.

把刚刚那棵 B+ 树叶子节点（即遍历所有数据）需要 1seek+6blocks。随后和上面的数据合并后，写回磁盘时需要 1seek+13blocks。

Indices on Multiple Keys

Suppose we have an index on combined search-key
(*dept_name*, *salary*).

- With the **where** clause
 - where *dept_name* = "Finance" **and** *salary* = 80000
the index on (*dept_name*, *salary*) can be used to fetch only records that satisfy both conditions.
 - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle
where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
where *dept_name* < "Finance" **and** *balance* = 80000
 - May fetch many records that satisfy the first but not the second condition

Index in Main Memory

存储器（内存）和磁盘/闪存的访问速度有很大差异：

- 内存访问比磁盘/闪存访问快得多。
- 但是，相比缓存，内存访问还是较慢。缓存是更快的存储区域，专门用于临时存储频繁访问的数据，以加快访问速度。

随机访问和缓存未命中

随机访问指的是非顺序的访问数据：

- 在大B+-树节点内进行二分查找时，需要访问很多不同的位置，这会导致缓存未命中（cache miss）。
- 缓存未命中意味着所需数据不在缓存中，需要从较慢的内存中读取数据，从而导致速度变慢。

缓存友好的数据结构

为了减少缓存未命中，使用更适合缓存的数据结构更为理想，即缓存友好（cache conscious）的数据结构：

- 较小的节点更适合缓存：小节点可以更容易地全部放入缓存中，从而减少缓存未命中。

B+-树优化策略：

- 使用大节点来优化磁盘访问：大节点可以减少从磁盘读取的次数，因为每次读取的数据量大。
- 在节点内使用小节点构成的树结构来优化缓存访问：在大节点内部，使用更小的节点（例如小的树结构而不是数组）来组织数据，使得这些小节点可以放入缓存中，减少缓存未命中。

总结

我们需要在设计B+-树时，兼顾磁盘访问和缓存访问的效率。为此，可以使用较大的节点来减少从磁盘读取的次数，同时在这些大节点内部，用较小的树结构来组织数据，以减少缓存未命中的情况。这样既能提高磁盘访问的效率，也能提高内存访问的效率。

Index on Flash

1. "Random I/O cost much lower on flash 20 to 100 microseconds for read/write":

这句话的意思是，在闪存设备上进行随机读写操作的成本比较低，大约在20到100微妙之间。这说明闪存在访问数据时速度非常快。

2. "Writes are not in-place, and (eventually) require a more expensive erase":

这里说的是，在闪存中写入数据不是在原有数据的位置上直接修改，而是需要写入到一个新的位置。原有的数据位置上的信息随后会被标记为无效，并且在之后某个时刻需要通过擦除操作来清除。擦除操作相对来说比较耗时且成本较高。

3. "Optimum page size therefore much smaller":

由于写入操作需要移动数据到新位置并且旧数据需要擦除，所以理想的数据页（page，数据存储的基本单位）大小应该尽可能小。这样可以减少需要擦除的数据量，从而提高效率。

4. "Bulk-loading still useful since it minimizes page erases":

尽管每次写入都可能涉及数据页的擦除，但是批量加载（bulk-loading）仍然是一个有用的策略。批量加载是指一次性写入大量数据，这种方法可以减少整体的页面擦除次数，因为数据是以集中的方式写入的。

5. "Write-optimized tree structures (i.e., LSM-tree) have been adapted to minimize page writes for flash-optimized search trees":

这句话提到了一种被称为LSM树（Log-Structured Merge-tree）的数据结构，它是专门为写入操作优化的。在闪存中使用LSM树可以最小化写入操作的次数，因此减少了页面擦除的需求。LSM树通过合并和延迟写入的技术来优化性能，特别适用于闪存这种介质。

不是直接修改，
而是“擦除”。

使用 Bulk-Loading
能减小开销。

优化 Write 的两种方法：

{ LSM Tree
Buffer Tree

Log Structured Merge Tree

1. "Records inserted first into in-memory tree (L0 tree)":

首先，所有新的记录都会被插入到一个内存中的树结构，称为L0树。这个树结构完全存储在内存中，因此数据的插入速度非常快。

2. "When in-memory tree is full, records moved to disk (L1 tree)":

当L0树的容量达到上限时，它里面的记录会被移动到磁盘上的另一个树结构，称为L1树。这个过程通常涉及到数据的批量移动，以减少频繁的磁盘写入操作。

3. "B+-tree constructed using bottom-up build by merging existing L1 tree with records from L0 tree":

在将记录从L0树移动到L1树时，使用一种名为B+树的数据结构来组织这些数据。这个B+树是通过自底向上的方式构建的，即通过合并已经存在于L1树中的数据和从L0树中来的新数据来完成。

4. "When L1 tree exceeds some threshold, merge into L2 tree":

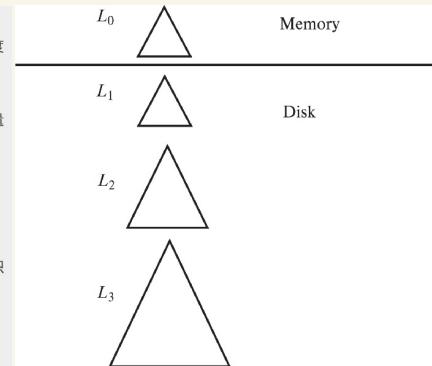
当L1树的大小超过某个预设的阈值时，它会被合并到下一层级的树结构，即L2树。这样的合并操作有助于维护数据的组织和提高查询效率。

5. "And so on for more levels":

这种层级的合并操作不仅限于L1和L2，而是可以继续向下进行多个层级，每一层都是前一层的扩展，用于处理更多的数据。

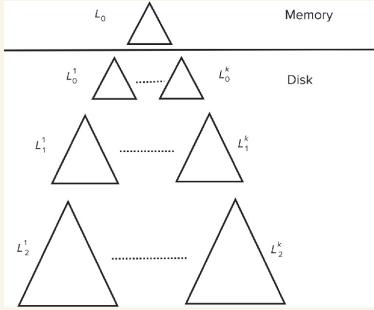
6. "Size threshold for Li+1 tree is k times size threshold for Li tree":

每一层树的大小阈值是上一层的k倍。这意味着随着数据层级的加深，每层可以存储的数据量也在成倍增加，这样设计可以有效地管理庞大的数据集，并减少顶层（在内存中的树）的频繁操作，从而优化整体的性能。



优点：①顺序插入 ②叶节点 full，空间利用率高。③和 B+ 树相比 I/O 少。
缺点：①查询复杂度高（搜索多层）②合并过程有大量复制。

⇒ Stepped Merge Index : Disk 上每层有 K 棵树



Stepped Merge Index

当K个索引处于同一层时，
合并写回下一层。

→ Bloom Filter

Query 开销更大！

Query a tree only if Boolean
Filter returns a positive result.

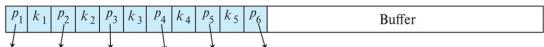
首先，LSM树允许我们通过添加特殊的“delete”条目来处理删除操作。当我们执行查找时，系统会同时检查原始条目和删除条目，但只返回没有匹配删除条目的原始条目。当合并树时，如果找到一个匹配的删除条目和原始条目，两者都会被删除。更新操作则是通过插入新条目和添加对应的删除条目来完成的。

LSM树最初是为了适应磁盘上的索引而引入的，因为它能够最大程度地减少磁盘上的写操作。但是，它也非常适合用于闪存等基于固态存储的索引，因为可以最大程度地减少擦除操作，延长固态存储器的使用寿命。

Buffer Tree

- Key idea: each internal node of B*-tree has a buffer to store inserts
 - Inserts are moved to lower levels when buffer is full
 - With a large buffer, many records are moved to lower level each time
 - Per record I/O decreases correspondingly
- Benefits
 - Less overhead on queries
 - Can be used with any tree index structure
 - Used in PostgreSQL Generalized Search Tree (GiST) indices
- Drawback: more random I/O than LSM tree

Internal node



Buffer