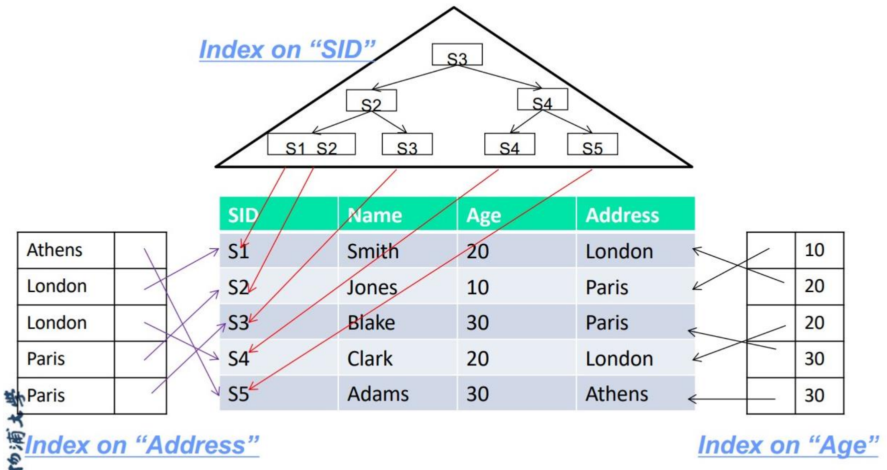
<Indexing Techniques>

Indexing mechanisms can speed up access to the desired data.
[索引机制可以加快对所需数据的访问]



SID	Name	Age	Address
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

The Structure of Index:

Search Key: one or set of attributes used to look up records in a file.

Data file: collection of blocks holding records on disk

Index file: an data structure allowing the DBMS to find particular records in a data file more efficiently.

An index file consists of records (called index entries) of the form:

search-key	pointer

Index files are typically much smaller than the original file.

Relationship: a search key K in the index file is associated with a pointer to a data-file record that has search key K.

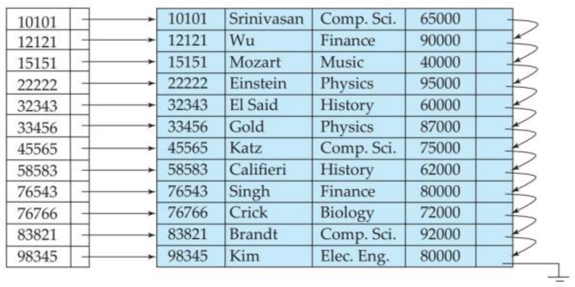
Index Evaluation Metrics [评价指标]:

Access types supported efficiently \ Access time \ Insertion time \ Deletion time \ Space overhead

Multiple Index:

Ordered index where index entries are sorted on the search key value:

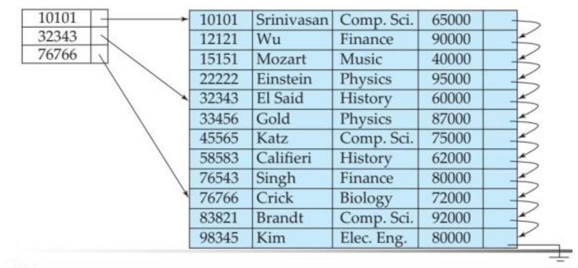
Dense index: index record appears for every search-key value in the file.



Index
10101
12121
15151
22222
32343
33456
45565
58583
76543
76766
83821
98345

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

Sparse Index: contains index records for only some search-key values.



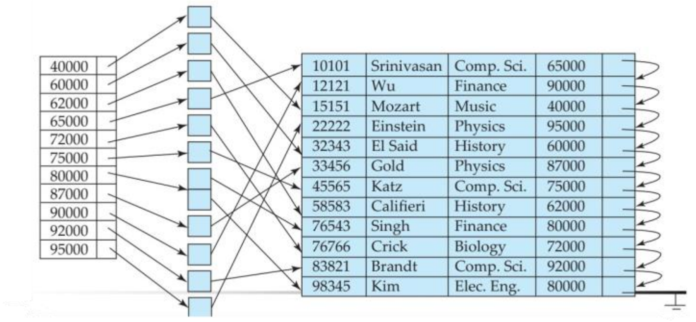
10101
32343
76766

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

Two ways to sort the index:

Primary index [集群索引]: an index whose search key specifies the sequential order of the file. （Can be sparse）

Secondary index [非集群索引]: an index whose search key specifies an order different from the sequential order of the file. （Can't be sparse）



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

Tips: have to be dense

索引对于搜索的帮助很大，但增删改会增加开销，每一次需要重新更新索引关系。

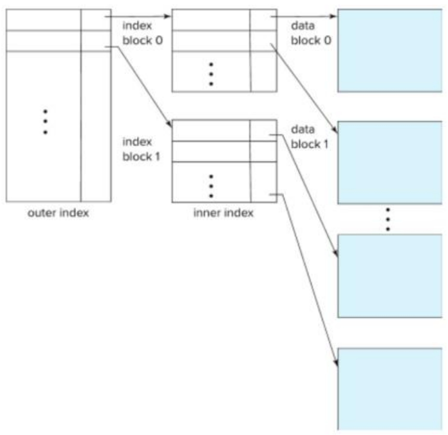
集群索引的顺序非常有效，但非集群索引将会很麻烦（每次访问新的区块文件可能会从一个新的磁盘调取）

So there comes the multilevel index:

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



（套娃，先用集群索引连结顺序磁盘文件，再用稀疏索引连结集群索引）

Hashing index where hashing technique is employed to organize index entries:

Index Definition in SQL：

Create an index:

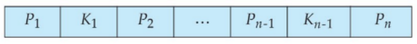
```
create index <index-name> on <relation-name> (<attribute-list>)
e.g. create index b-index on branch(branch_name)
```

To drop an index :

```
drop index <index-name>
```

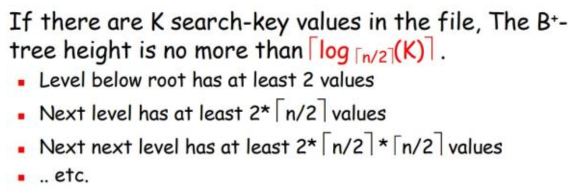
<B+ Tree Indexing>

由于索引顺序文件随着文件数目增多变得查询缓慢、更改麻烦，故引入 B+ Tree 索引结构，它对于增删改只需要小部分的改变结构，只不过空间开销会更大。（advantage>disadvantage）（“short” and “fat”）

Typical B+ Tree node: 

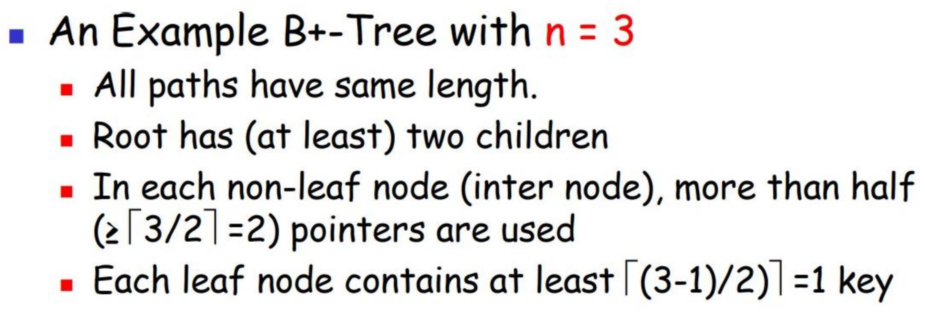
Some feature of B+ Tree:

- n (or sometimes N) is the number of pointers in a node (pointers: P1, P2, …Pn)
- Search keys: K1 < K2 < K3 < . . . < Kn−1
- All paths (from root to leaf) have same length
- Root must have at least two children
- In each non-leaf node (inner node), more than 'half' : n/2(round up) pointers must be used
- Each leaf node must contain at least (n-1)/2(round up) keys
- Sp: If the root is not a leaf, it has at least 2 children
- Sp: If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n−1) values

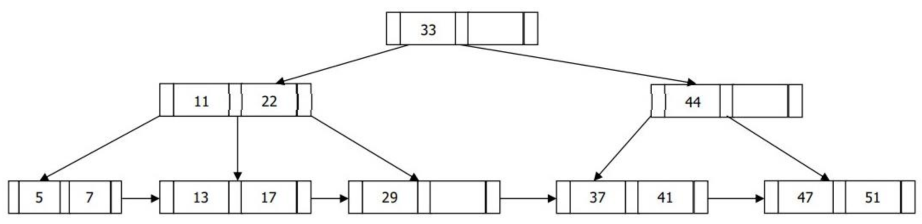


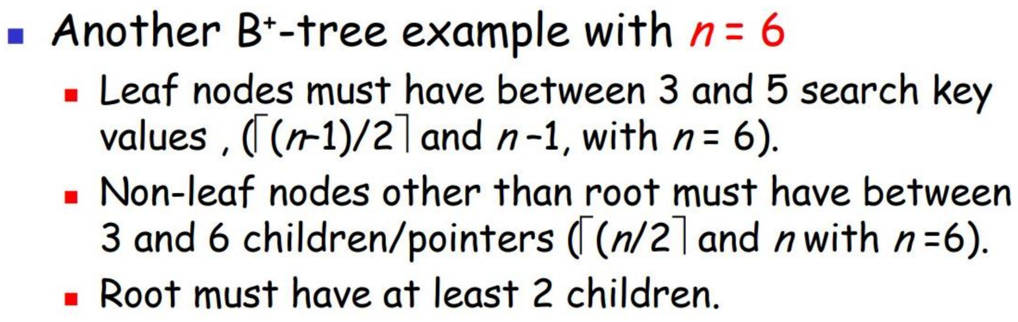
If there are K search-key values in the file, The B+-tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.

- Level below root has at least 2 values
- Next level has at least 2*⌈n/2⌉ values
- Next next level has at least 2*⌈n/2⌉*⌈n/2⌉ values
- .. etc.

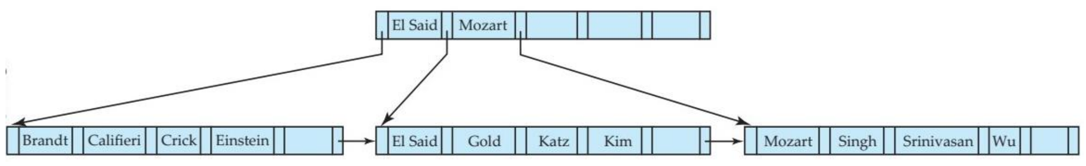


- An Example B+-Tree with n = 3
  - All paths have same length.
  - Root has (at least) two children
  - In each non-leaf node (inter node), more than half (≥⌈3/2⌉=2) pointers are used
  - Each leaf node contains at least ⌈(3-1)/2)⌉=1 key





- Another B+-tree example with *n* = 6
  - Leaf nodes must have between 3 and 5 search key values , (⌈(*n*-1)/2⌉ and *n* -1, with *n* = 6).
  - Non-leaf nodes other than root must have between 3 and 6 children/pointers (⌈(*n*/2⌉ and *n* with *n* =6).
  - Root must have at least 2 children.



Queries on B+-Trees [查询]:

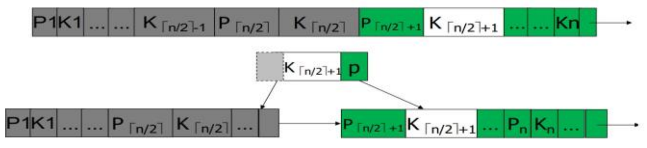
- Find record with search-key value V.
  - 1. C=root
  - 2. While C is not a leaf node
    - 2.1. Let i be least value such that V ≤ K_i.
    - 2.2. If no such i exists, set C = last non-null pointer in C
    - 2.3. Else { if (V = K_i) Set C = P_{i+1} else set C = P_i}
  - 3. Let i be least value such that K_i = V
  - 4. If there is such a value i, follow pointer P_i to the desired record.
  - 5. Else no record with search-key value V exists.

Searching is very efficiently: a node is generally the same size as a disk block, typically 4 kilobytes. And n is typically around 100 (40 bytes per index entry), with 1 million search key values and n = 100. So at most log50(1,000,000) = 4 nodes are accessed in a lookup.

Updates on B+-Trees: Insertion [插入]

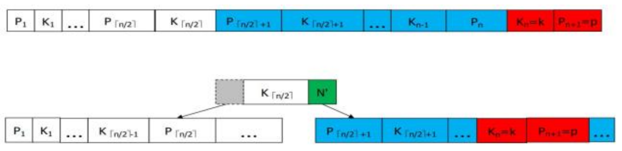
- 1. Find the leaf node in which the search-key value would appear
- 2. If the search-key value is already present in the leaf node
  - 2.1. Add record to the file
  - 2.2. If necessary add a pointer to the bucket.
- 3. If the search-key value is not present, then
  - 3.1. add the record to the main file (and create a bucket if necessary)
  - 3.2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  - 3.3. Otherwise, **split** the node (along with the new (key-value, pointer) entry) as discussed in the next slides.

Splitting a Leaf Node



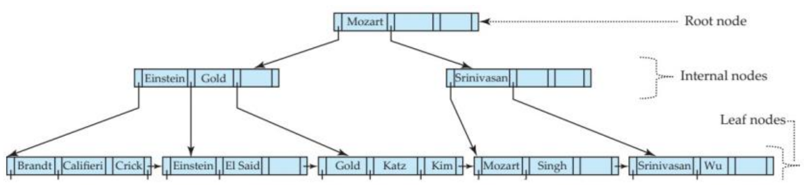
P1 K1 … … K ⌈n/2⌉-1 P ⌈n/2⌉ K ⌈n/2⌉ P ⌈n/2⌉+1 K ⌈n/2⌉+1 … … Kn
K ⌈n/2⌉+1 p
P1 K1 … … P ⌈n/2⌉ K ⌈n/2⌉ …
P ⌈n/2⌉+1 K ⌈n/2⌉+1 … Pn Kn …

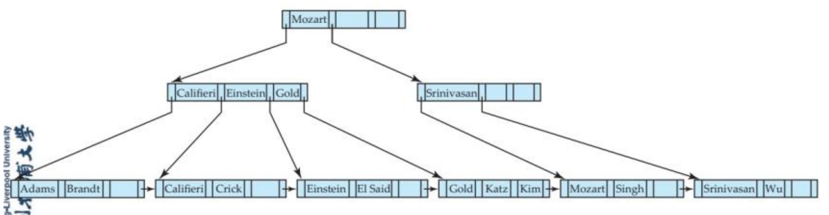
Splitting a Non-leaf Node



P1 K1 … P ⌈n/2⌉ K ⌈n/2⌉ P ⌈n/2⌉+1 K ⌈n/2⌉+1 … Kn-1 Pn Kn=k Pn+1=p
K ⌈n/2⌉ N'
P1 K1 … K ⌈n/2⌉-1 P ⌈n/2⌉ …
P ⌈n/2⌉+1 K ⌈n/2⌉+1 … Kn=k Pn+1=p …

Diagrammatize [图解]:





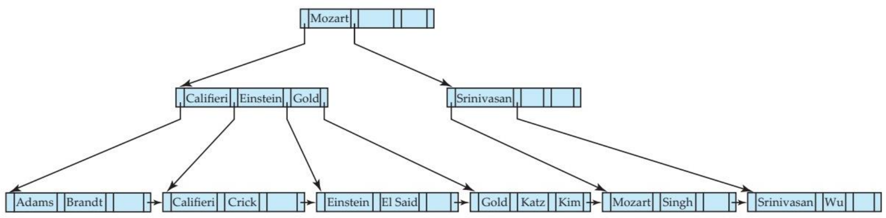
B+-Tree before and after insertion of "Adams"

Updates on B+-Trees: Deletion [删除]

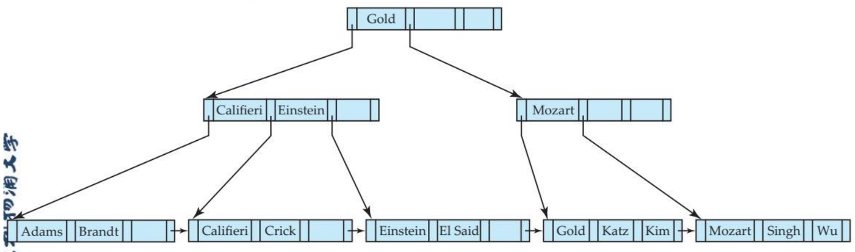
- Find the record to be deleted, and remove it from the main file and from the bucket (if present).
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty.
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then merge siblings.
- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then redistribute pointers.
- The node deletions may cascade upwards till a node which has ⌈n/2⌉ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

掌握自下而上，叶子节点和非叶子节点的拆开与合并方法。由于 B+树所有关键字存储在叶子节点出现,内部节点(非叶子节点并不存储真正的 data)。

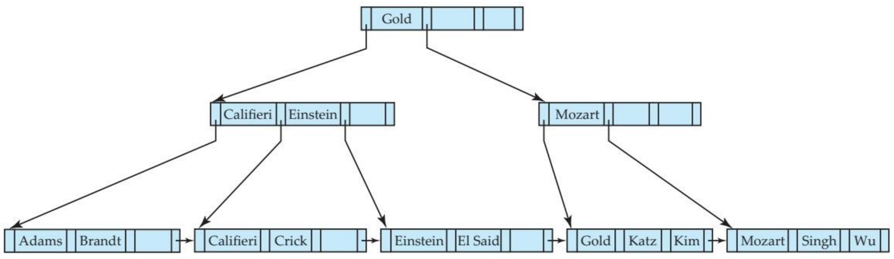
Diagrammatize [图解]:



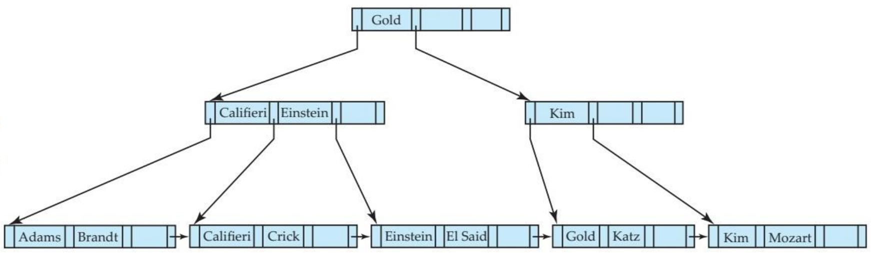
Before and after deleting “Srinivasan”

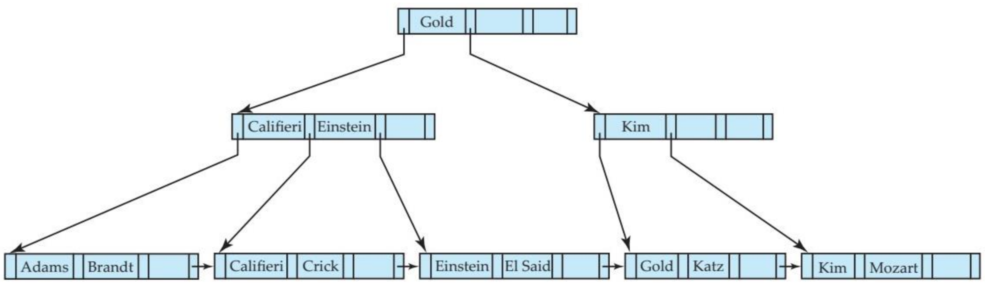


- Deleting “Srinivasan” causes merging of under-full leaves

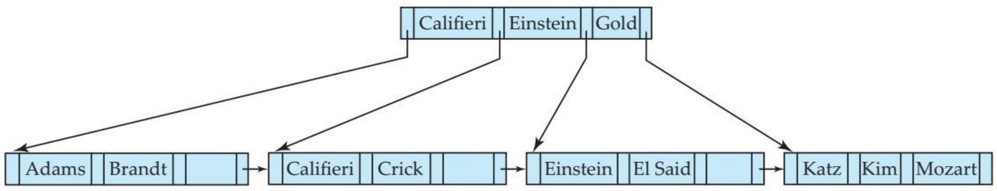


Before and after deleting “Singh and Wu”





Before and after deleting “Gold”



结合 tutorial 后对 B+树的白话理解：

首先，n 就是每个页的大小，然后它的指针就有 n 个（分布在中间和左右）,中间空白 n-1 个，以下全局有关的增、删、改、查基本都是基于[round up](n/2)进行定位的

对于叶节点：

一般操作都是先从根开始走定位到叶节点，之后遵守每个叶中的元素不超过 n 的原则进行拼接或者拆分。至少元素得有[round up](n/2)个到 **n-1** 个。

对于非叶节点：

对于叶节点的操作一般都会影响到其父辈非叶节点的结构，普通情况就是减少或增加非叶节点中的指针，特殊情况是需要联合爷辈一起组合为一级或者把父辈中的一个节点拆给爷辈，叶节点删除或者更改某个数据非叶节点可以不理睬。至少每个根的孩子得有两个。