Chapter 14: Indexing

Database System Concepts, 7th Ed.

Chapter 14: Indexing

- Basic Concepts
- Ordered Indices
- B^+-Tree Index Files
- B-Tree Index Files
- Hashing

https://www.vertabelo.com/blog/technical-articles/an-introduction-to-mysql-indexes

Basic Concepts

- Indexing mechanisms used to **speed up access** to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

search-key	pointer

What is an index?

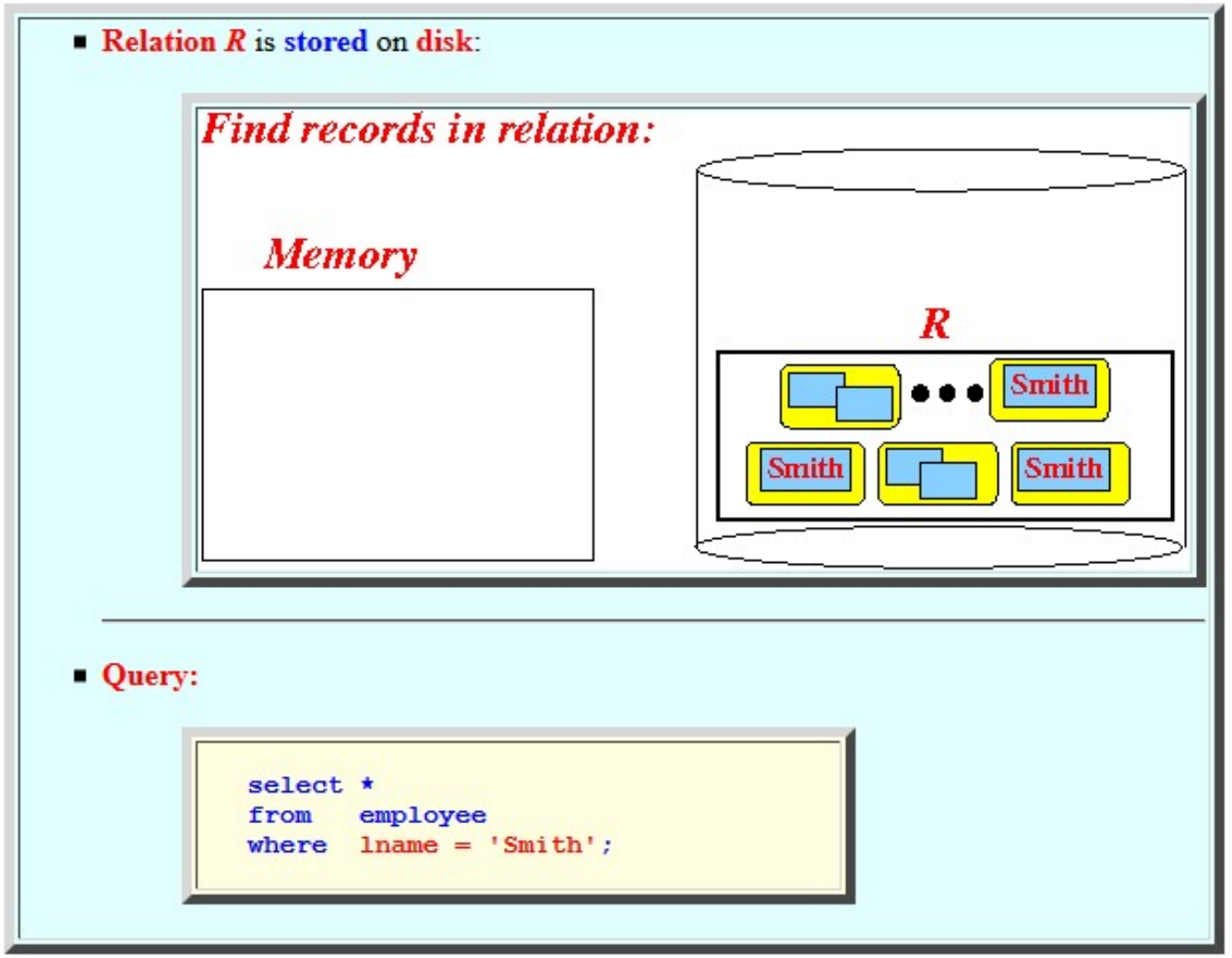
- **Index** = a **data structure** that **enable** the **user** to **find (= locate)** **data items** ***efficiently*** **(quickly)** using **search keys**

  - **Sample** of an **index** (found in the **back** of a **text book**):

Index

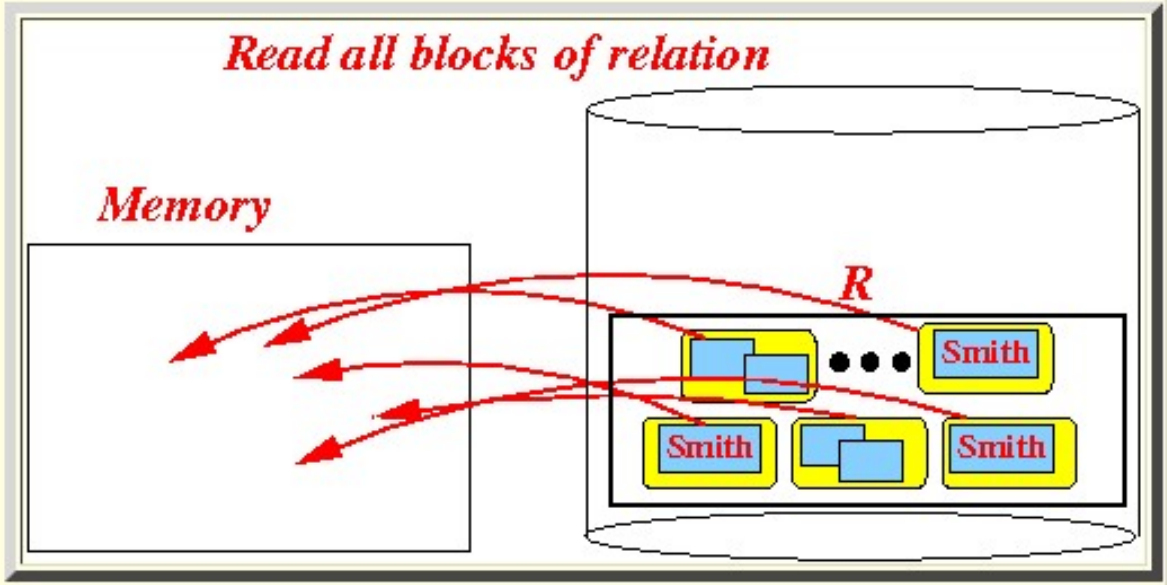
Example

- Consider the following query on relation stored on disk:



Example

- **Without** further information, the **only way** to **process** the **above query** is:
  - **Read** ***all*** **blocks** of the **relation** `employee`



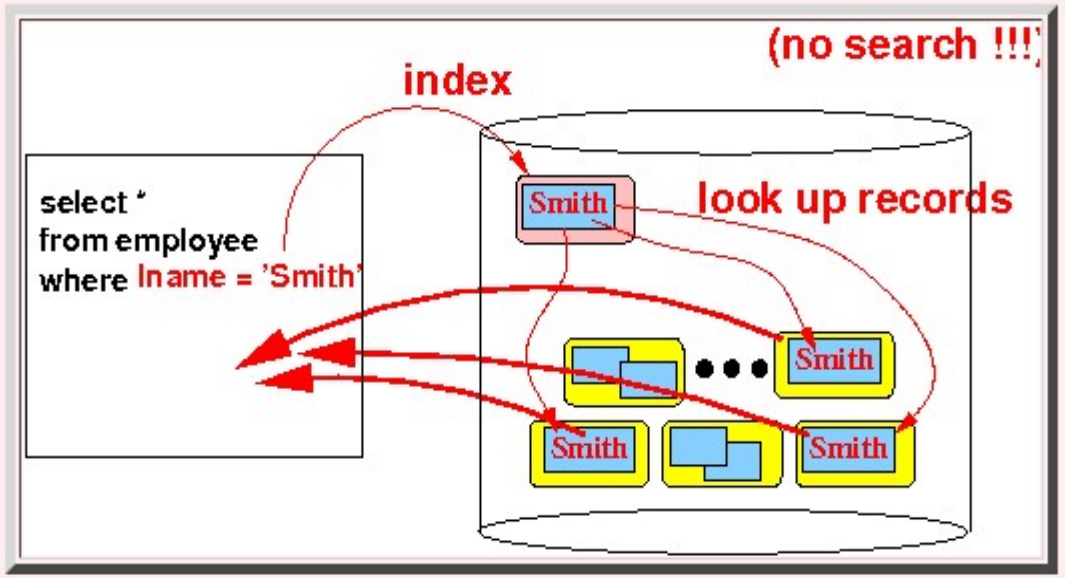
  - **Look** for **tuples** with `lname = 'Smith'`

Example

- More efficient processing:

  - Maintain location information on specific attributes (most useful ones, e.g., key attributes) in the relation

  Graphically:



  We only need to read a setset of the data block

Definitions

- **Search key:**

  - **Search key** = **field(s)** used to **create** the ***additional*** **search information** used to **speed up** a **look up** operation

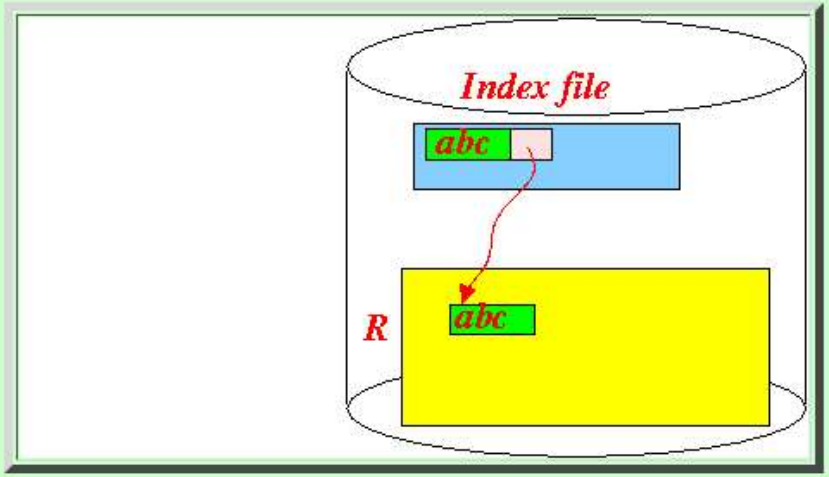
---

- **Index file:**

  - **Index file** = a **file** that store **records** of the following **format**:

```
+------------+----------------------+
| Search key | block/record pointer |
+------------+----------------------+
```

Graphically:



Note:

  - The **size** of an **index file** is **usually** ***much smaller*** than the **size** of a ***data*** **file**

Note

- The pointer is usually a block pointer
  - So the index allow you to locate the block that contain the record quickly

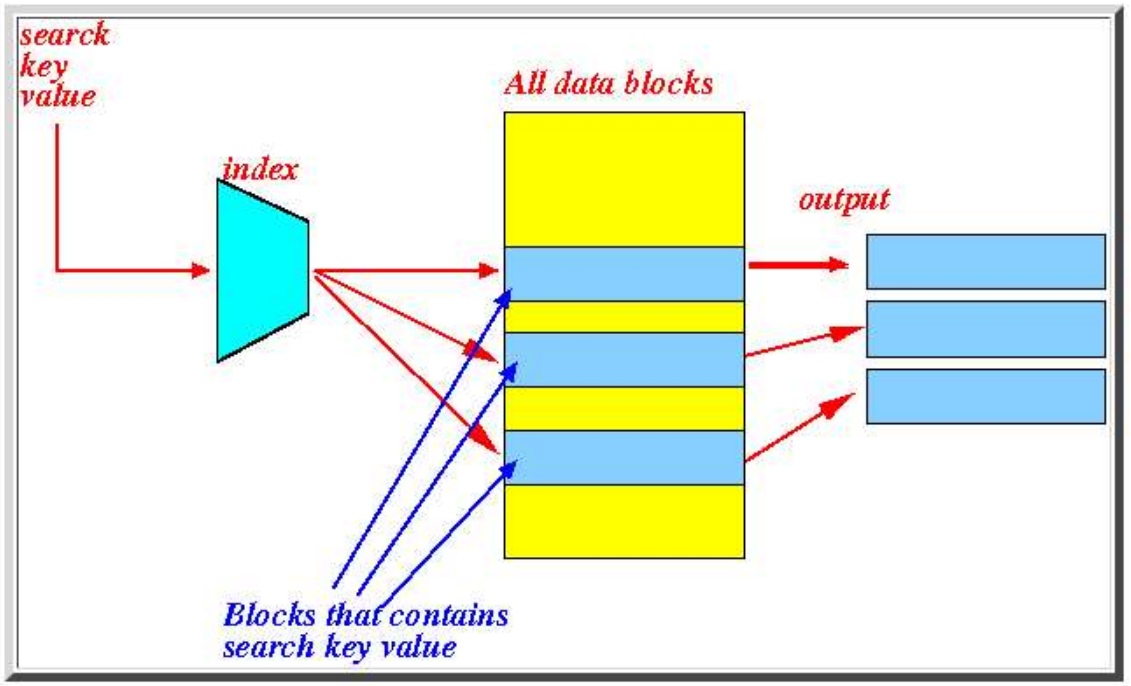
---

- The record is found by a search operation inside the block (after the block is read into main memory)
  - Because the search operation *only* access main memory, the search is relatively quick (because you don't use disk access operations in the search)

Basic Concepts of Indexing

- Speed up data access

An **index** will **achieve** this **effect**:



In other words:

- An **index** performs the following **mapping**:

```
Search key value  ========>  List of blocks that contains
                             the search key value
```

Index File

- An index file is a computer file with an index that allows easy random access to any record given its file key.

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

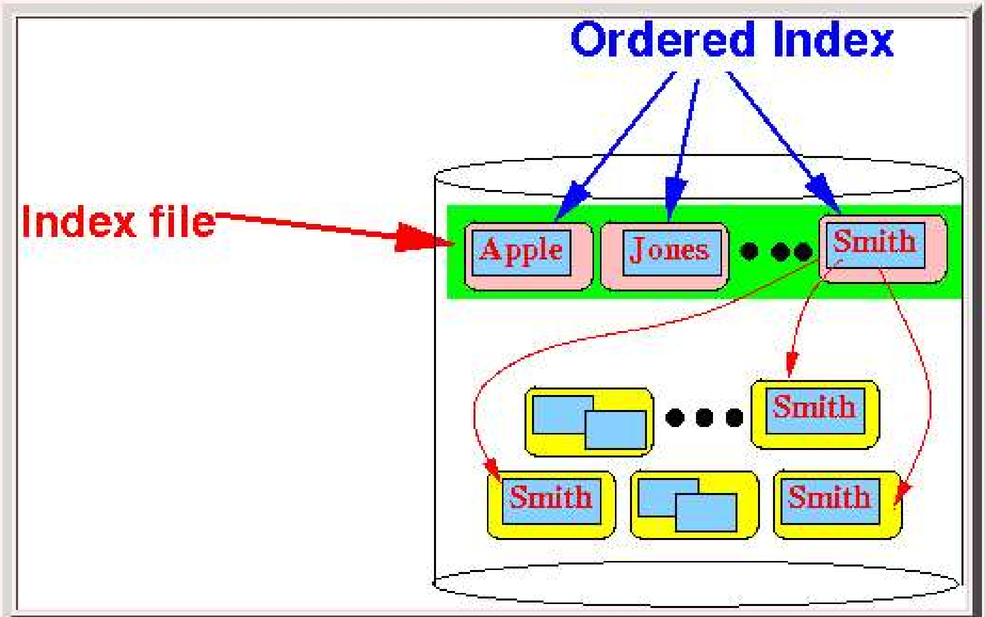
search-key	pointer

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function"

Type of index

- There are **2 basic** kinds of **indexes**:
  - **Ordered indexes**
    - The **search keys** are **stored in sorted order** inside the **index file**

      Schematically:

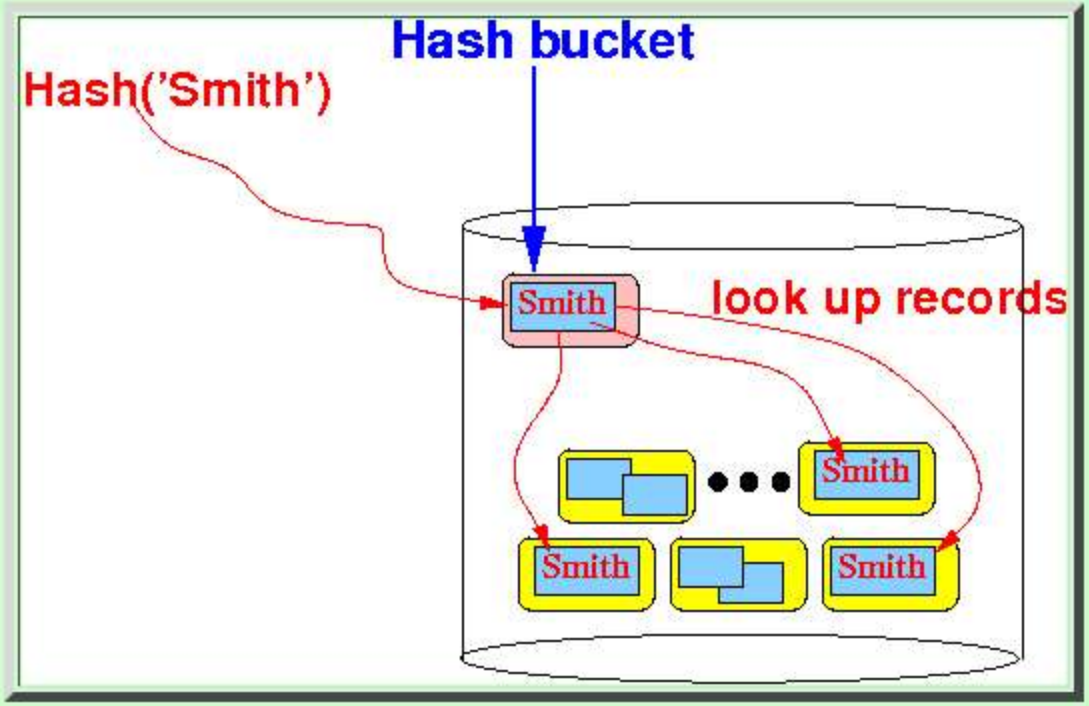


We can **find** a **search key** in the **index file** ***quickly*** using ***binary* search** !!!

Hash Index

- The search keys are stored in hash buckets.
  - Each hash bucket is stored as a 1 or more blocks in the index file

Schematically:



We can find the disk blocks containing the search key *quickly* by using the hash function

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

Ordered Indices

- Index entries are sorted by search key values
  - E.g., author catalog in library.

- The **search keys** are **stored** in **sorted order** in the **index file**

Schematically:

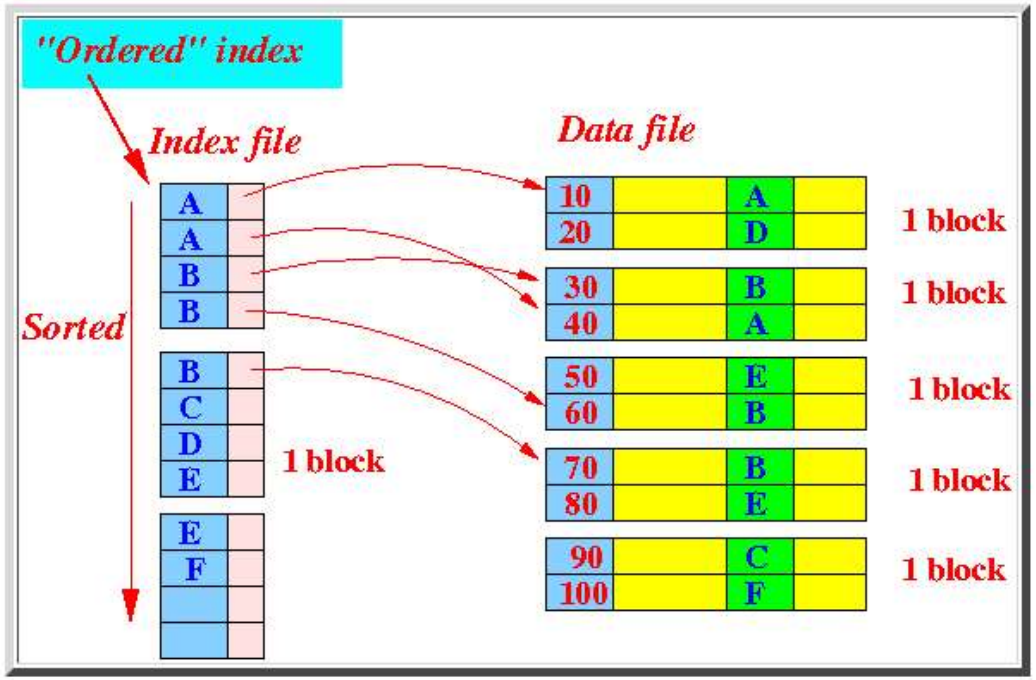
Index file content:

Search key	
cat	p1
dog	p2
elephant	p3
...	

Ordered Index

- **Ordered index** = an **index file** where the **index entries** are *sorted* (in the **order** of the **search key**)

Example: an **ordered index**



Note:

- Take **sorted** with a **grain of salt**

  We will discuss **B-tree** that store the **keys** in a **tree structure**

- There is an **ordering** of the **keys** in a **B-tree**, but the **ordering** is not *sequential*

Ordered Indices

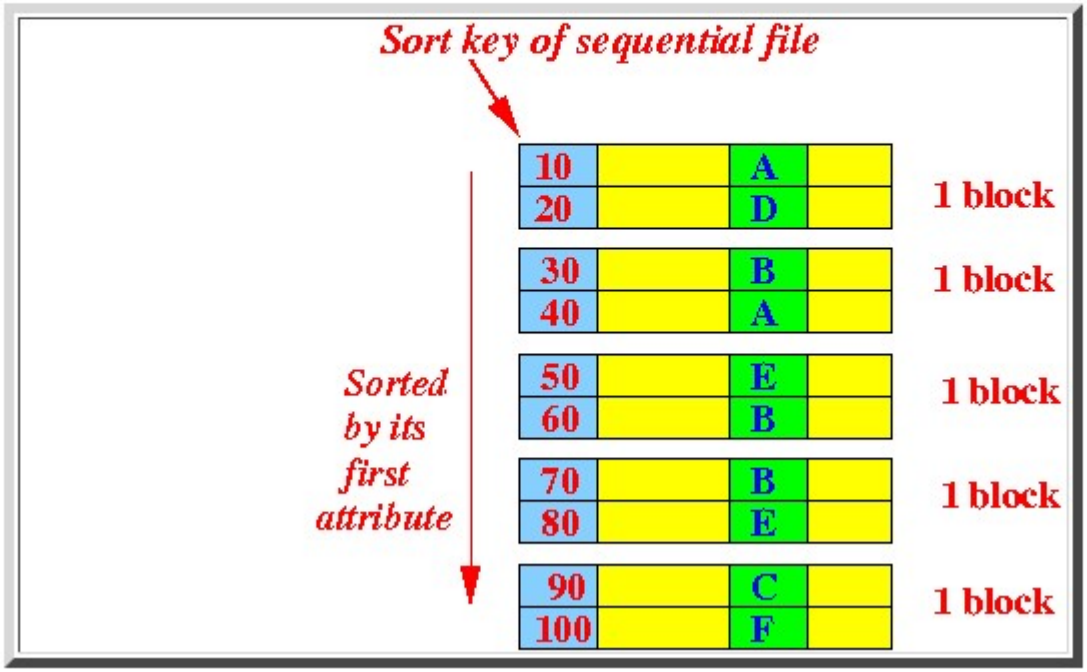
- **Primary index** (clustering index)
  - the index whose search key specifies the sequential order of the file
    - Index-sequential files**:** files ordered on a primary index
  - The search key of a primary index is usually the primary key

- **Secondary index** (non-clustering index)
  - An index whose search key specifies an order different from the sequential order of the file.

Sequential File (IBM's terminology....)

- **Sequential file** = a **file** whose **records** are *sorted* by some *attribute(s)* (usually its **primary key**)

Example: a **sequential file**



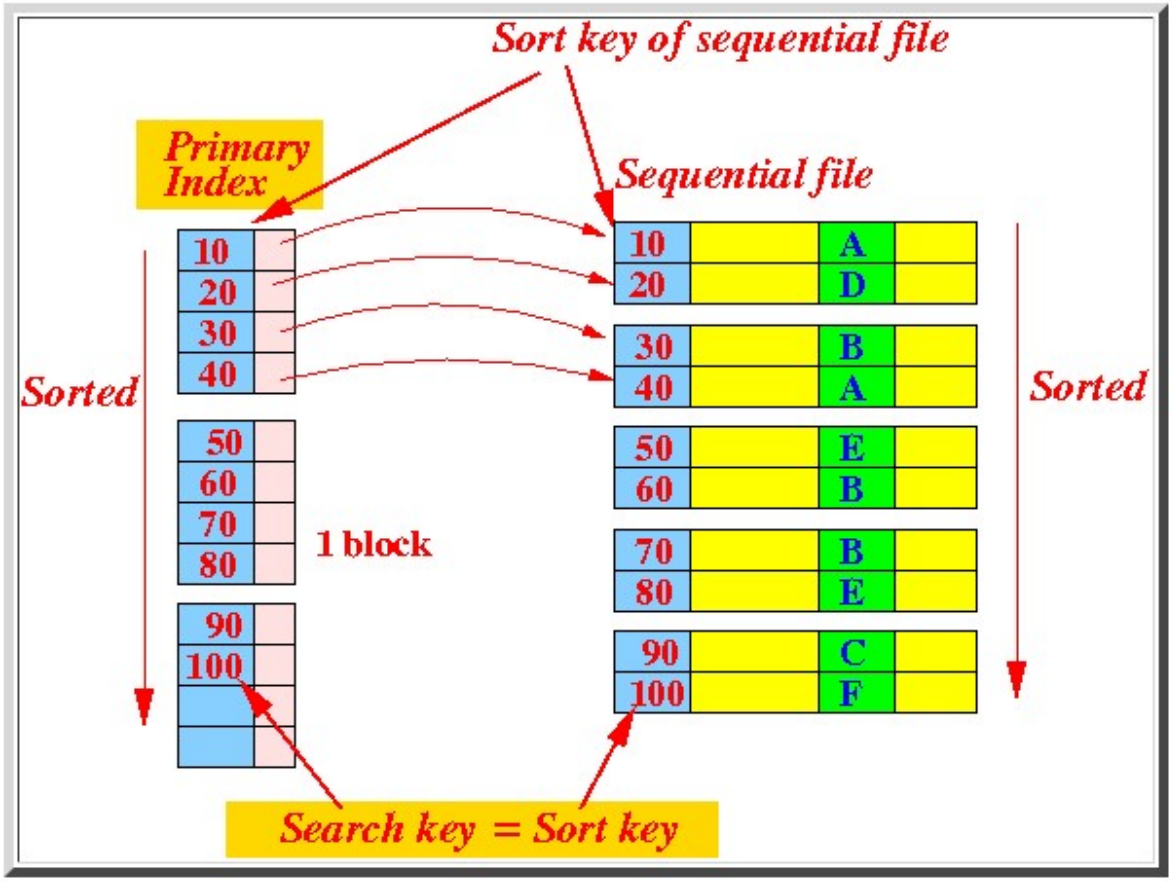
- **Sort key:**

- **Sort key** = **field(s)** whose values are used to **sort/order** the **records** in a **sequential file**

Primary Index (clustering index)

- **Primary index** = an **ordered index** whose **search key** is *also* the *sort key* used for the **sequential file**

Example: a primary index

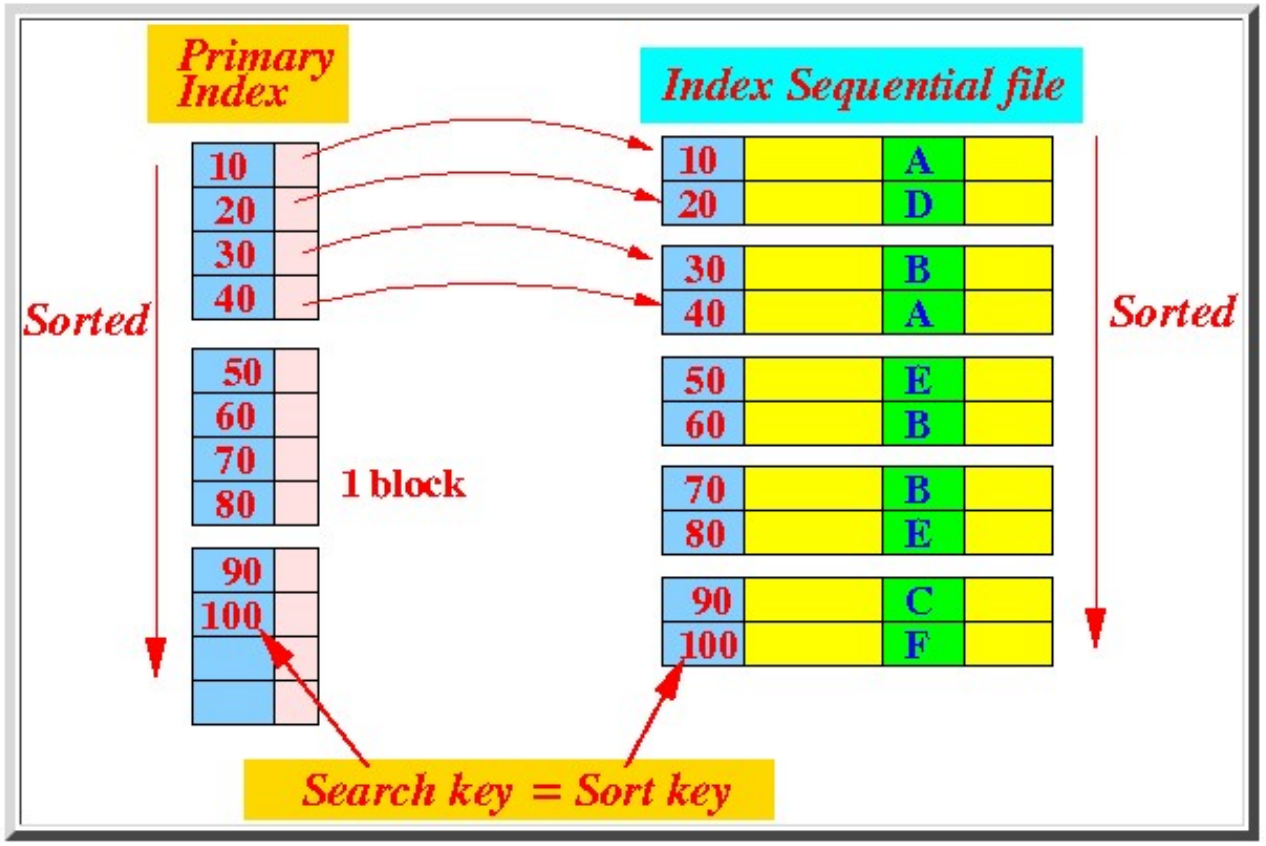


Index File

- **Index sequential file**

  - **Index sequential file** = a **sequential file** that has a ***primary*** **index**

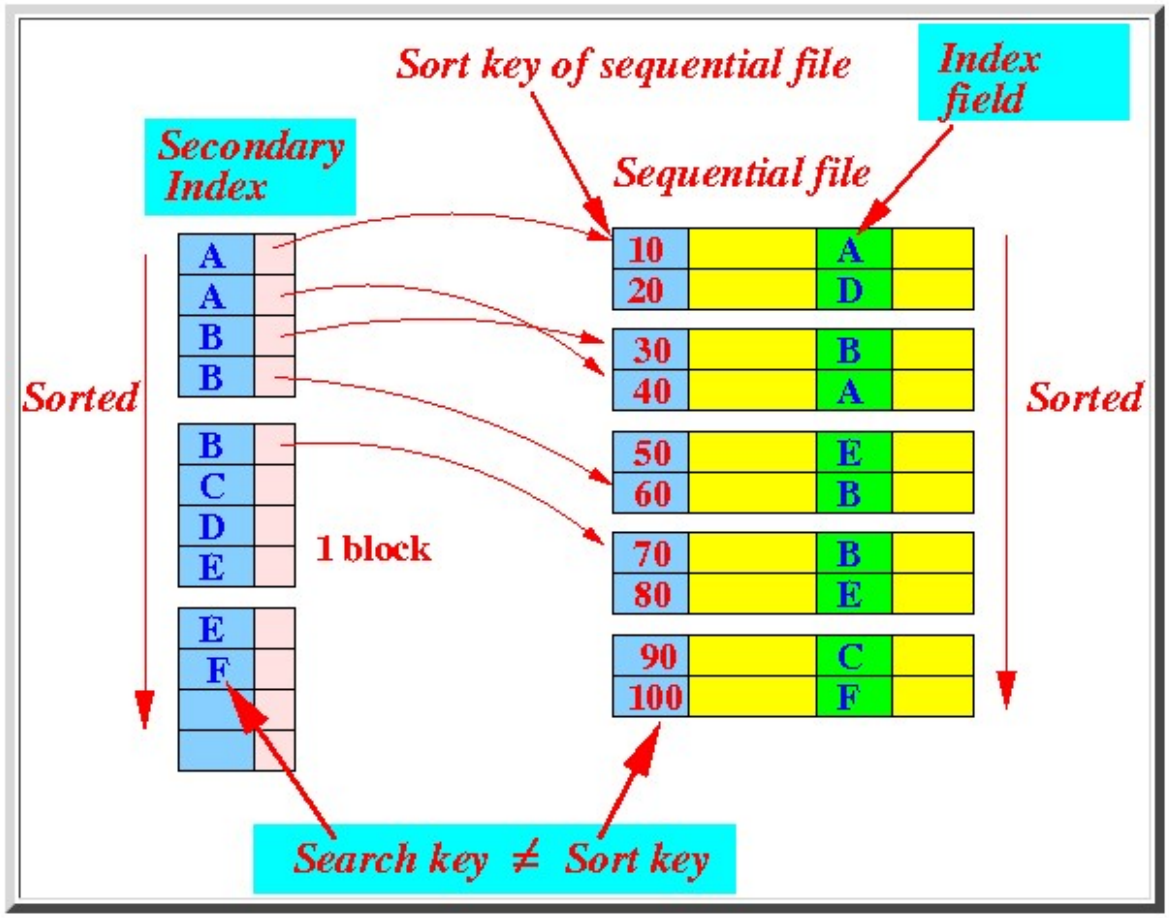
Example: an **index sequential file**



Secondary Index (non-clustering index)

- **Secondary index** = an **ordered index** whose **search key** is *NOT* the ***sort key*** used for the **sequential file**

Example: a **secondary index**



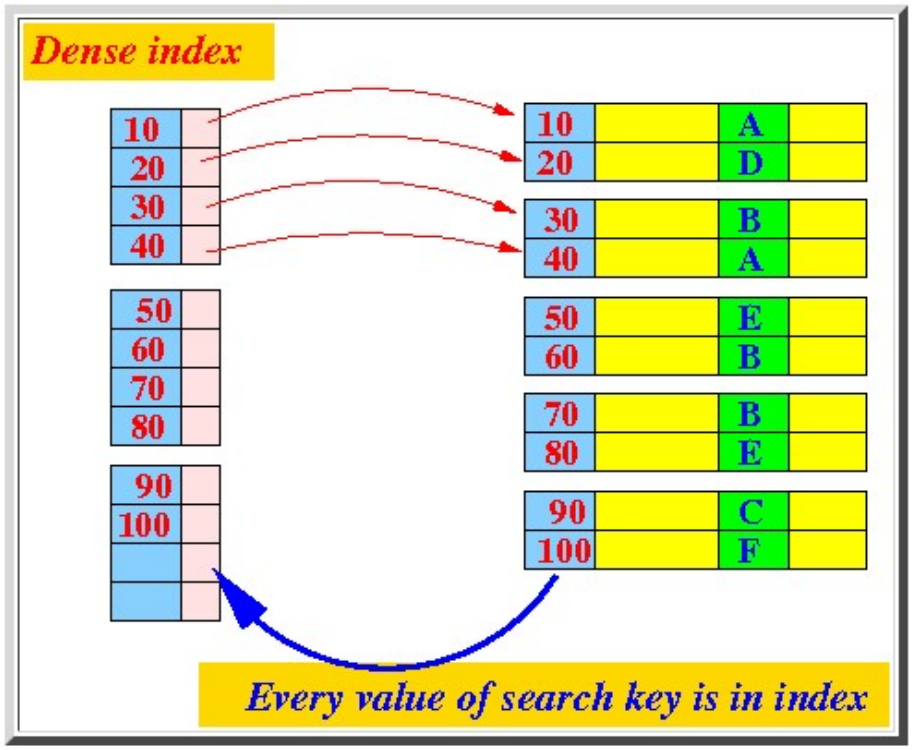
Dense Index

- *Dense* **index** = the **index file** contains an **index entry**:

```
+-----------+-----------------------+
| Search key | block/record pointer |
+-----------+-----------------------+
```

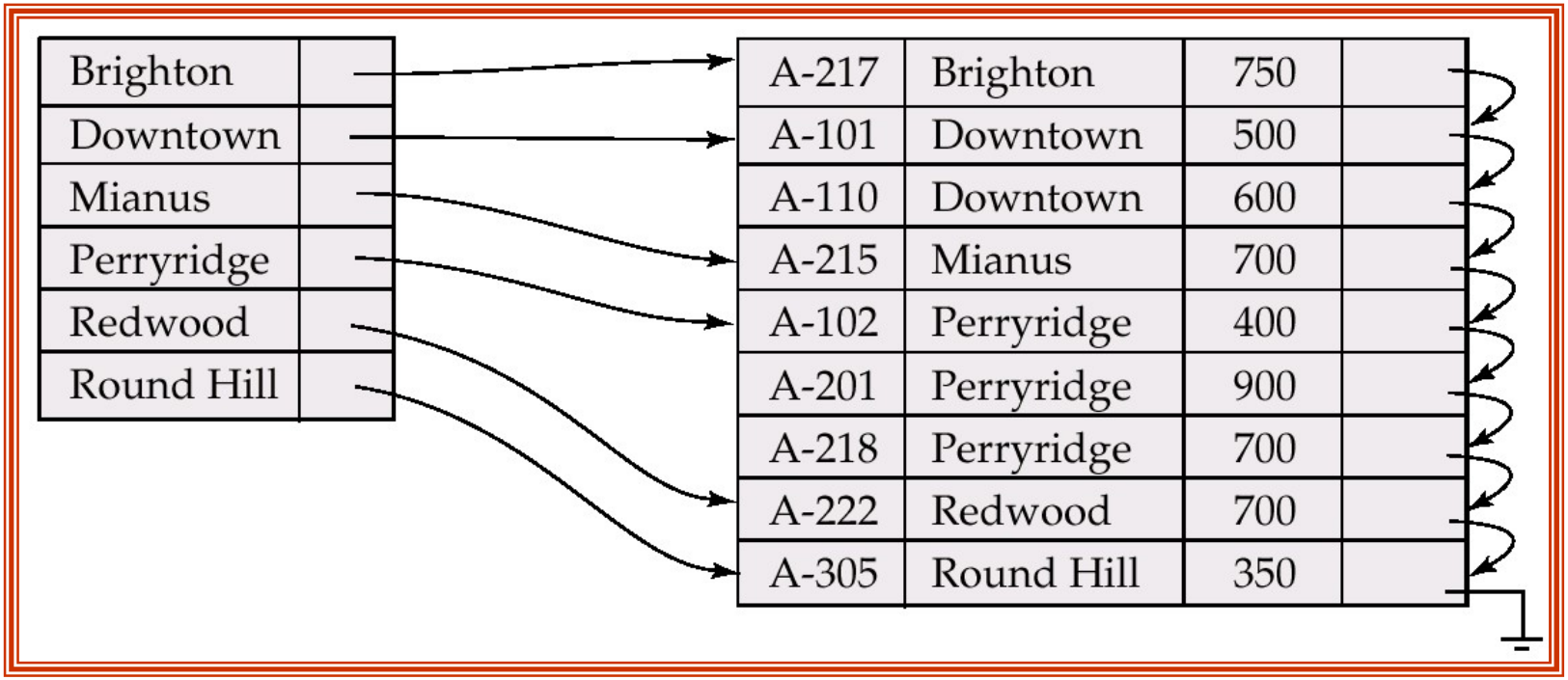
for *every* **value** of the **search key** in the **data file**

Example: a **dense** index



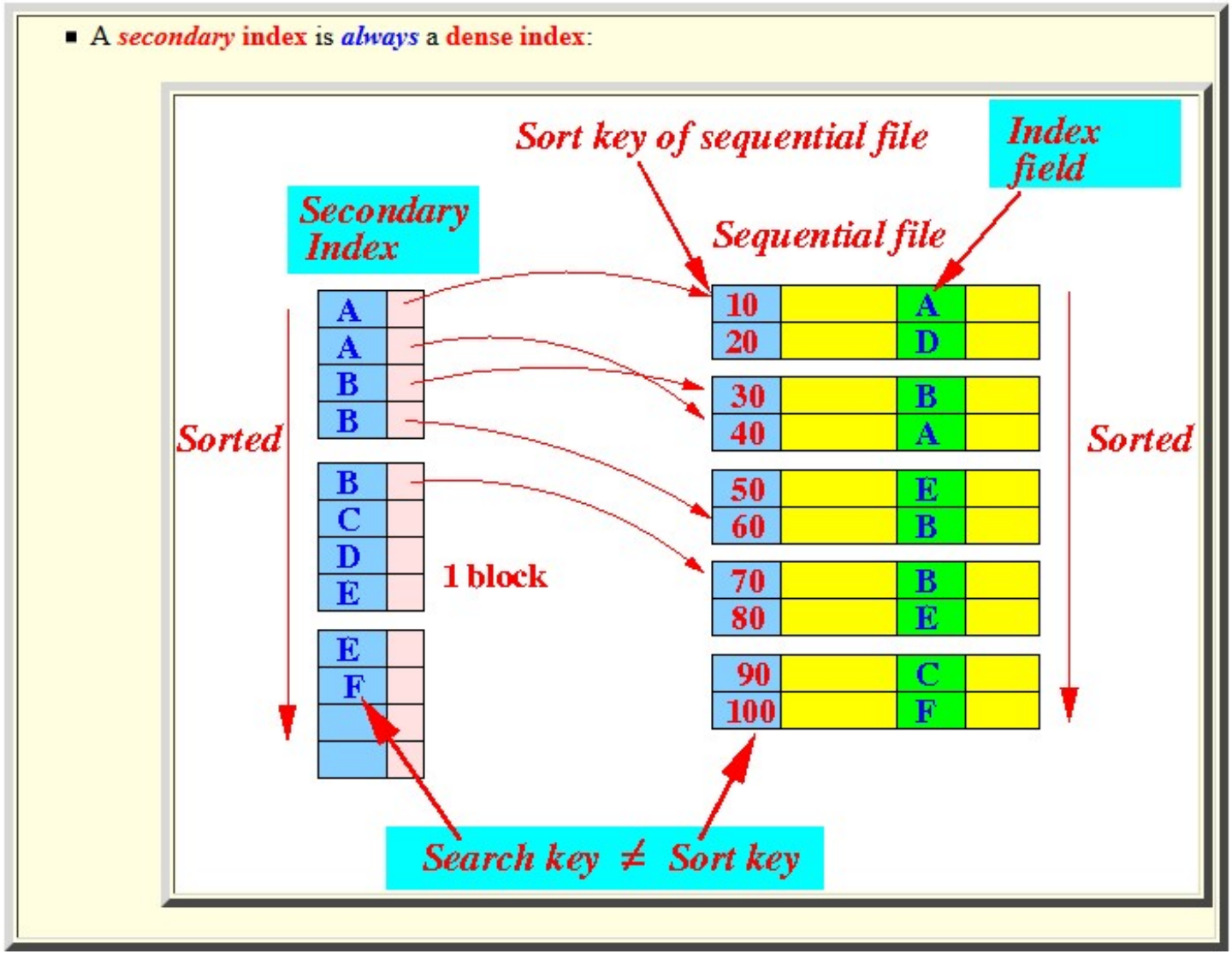
Dense Index Files

- Dense index — Index record exists for every search-key value in the file.



Note

- A *secondary* index is *always* a dense index:



Sparse Index Files

- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value *K* we:
  - Find index record with largest search-key value *K*
  - Search file sequentially starting at the record to which the index record points
- Less space and less maintenance overhead for insertions and deletions.
- Slower than dense index for locating records.
- Good tradeoff: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.

Sparse Index

- *Sparse* **index** = the **index file's search entries**

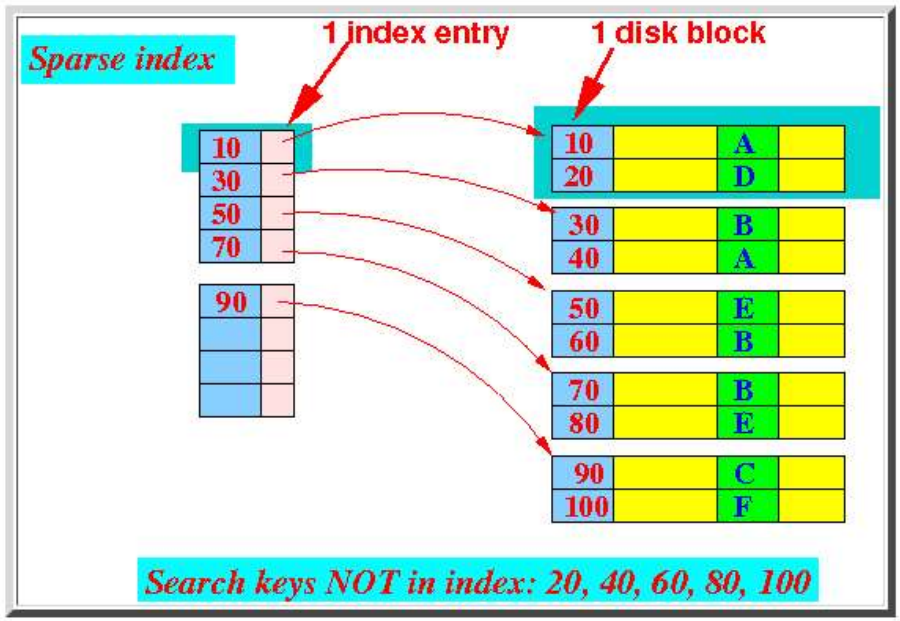
```
+-----------+-----------------------+
| Search key | block/record pointer |
+-----------+-----------------------+
```

does **not** contain ***every*** **value** of the **search key** of the **data file**

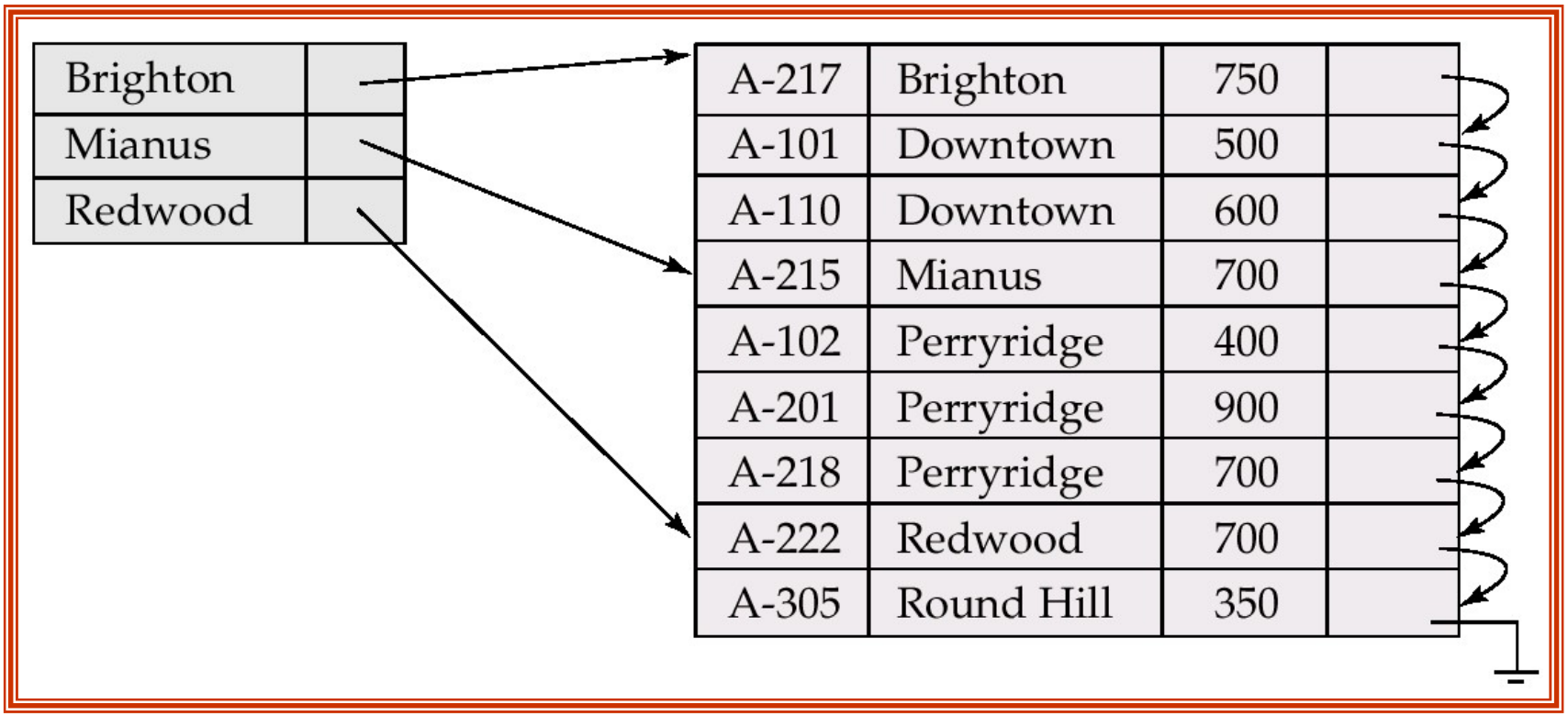
- The ***search*** **key** of a **sparse index** must be the **sort key** of the **data file** !!!

- A ***sparse*** **index** contains:
  - ***one*** **index** for each **(data)** ***block***

- **Organization** of a *sparse* index



Example of Sparse Index Files

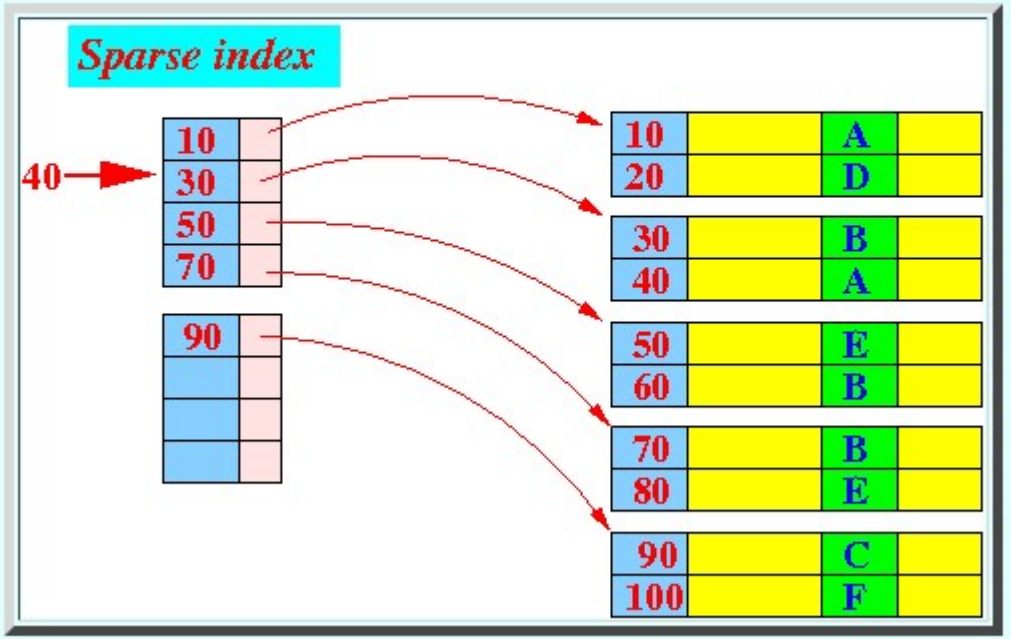


How to use a sparse index

Example:

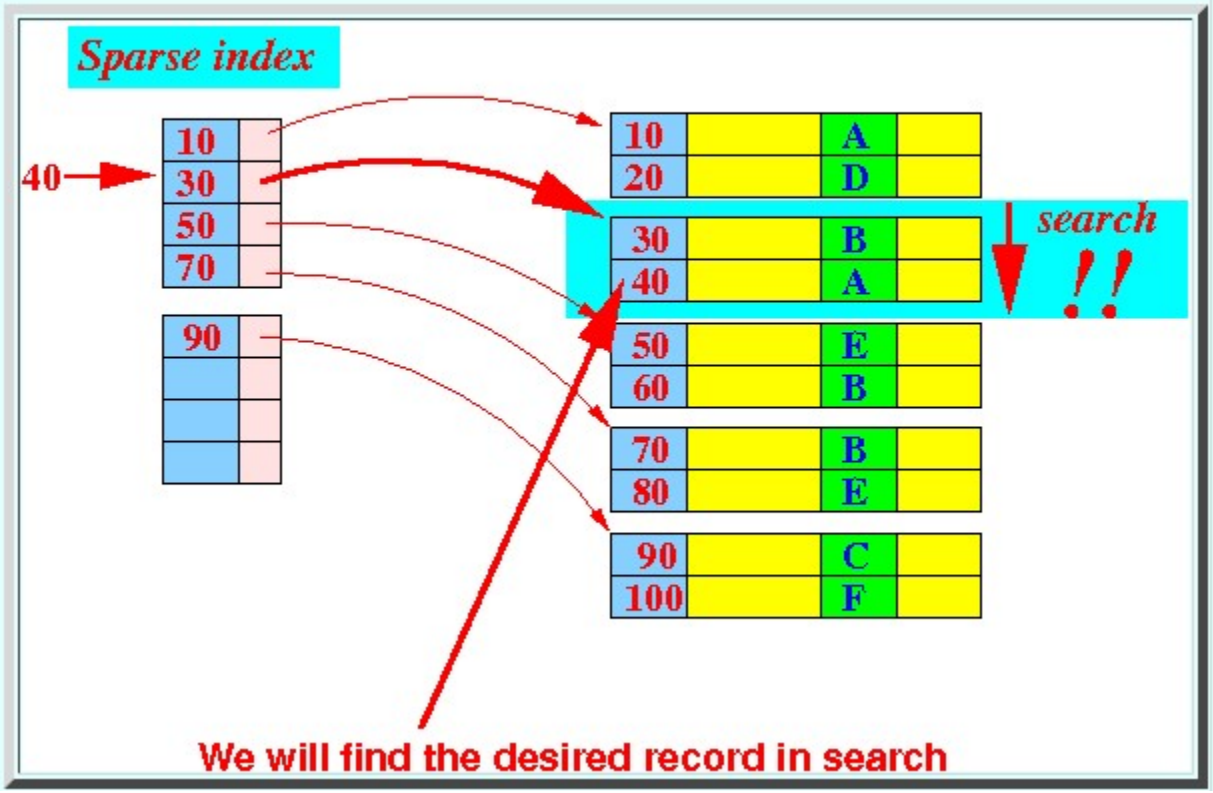
- **Look up** the **record** with **search key = 40**

- **Find** the **search key** ≤ **40**:



How to use a sparse index

- *Search* in the **block** for **search key 40**:



- ***Only* primary indexes *can*** be a **sparse index**

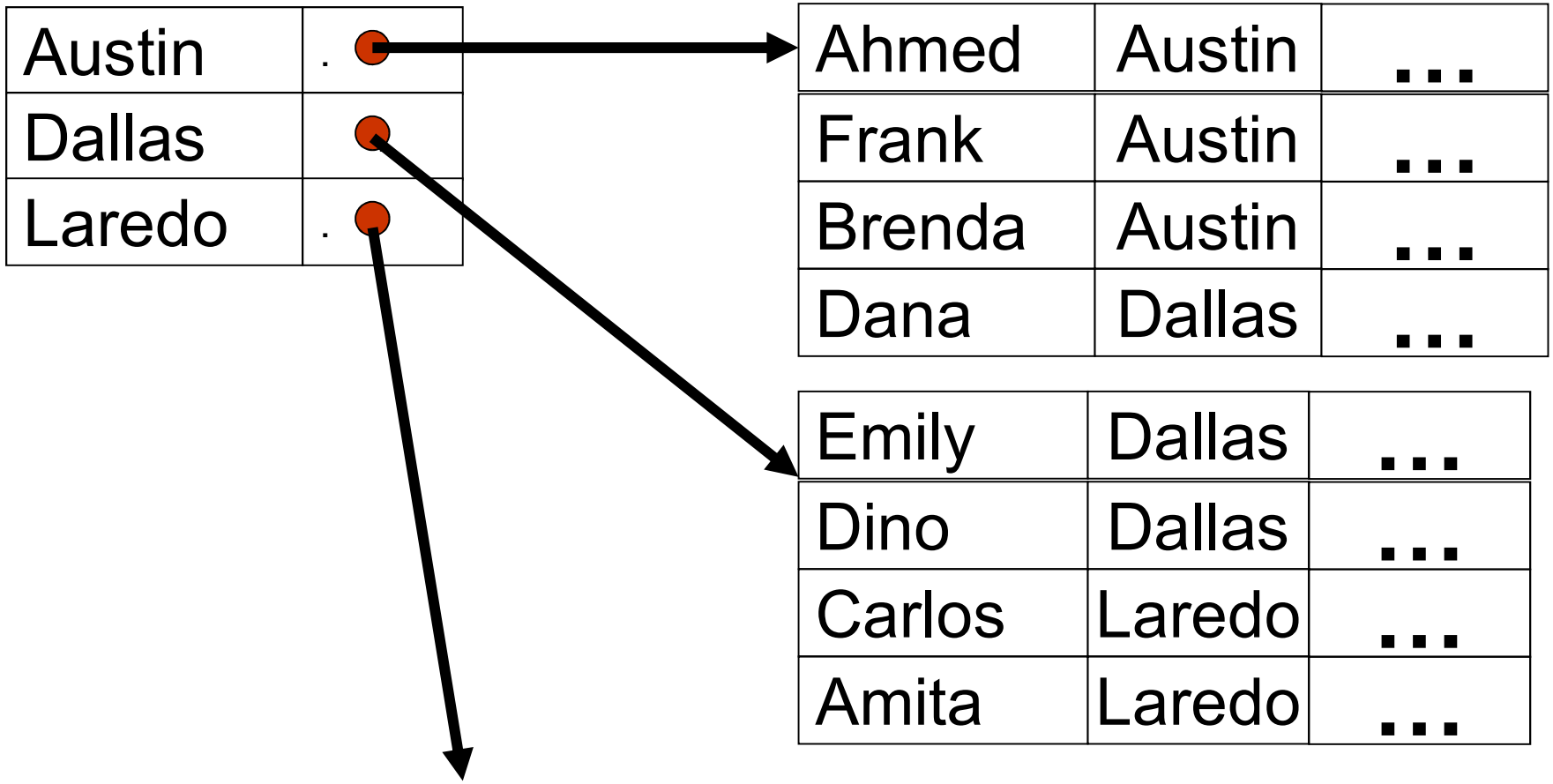
  (Using the **technique** above: the **sparse index** stores the ***first* key** in each **data block**)

Examples

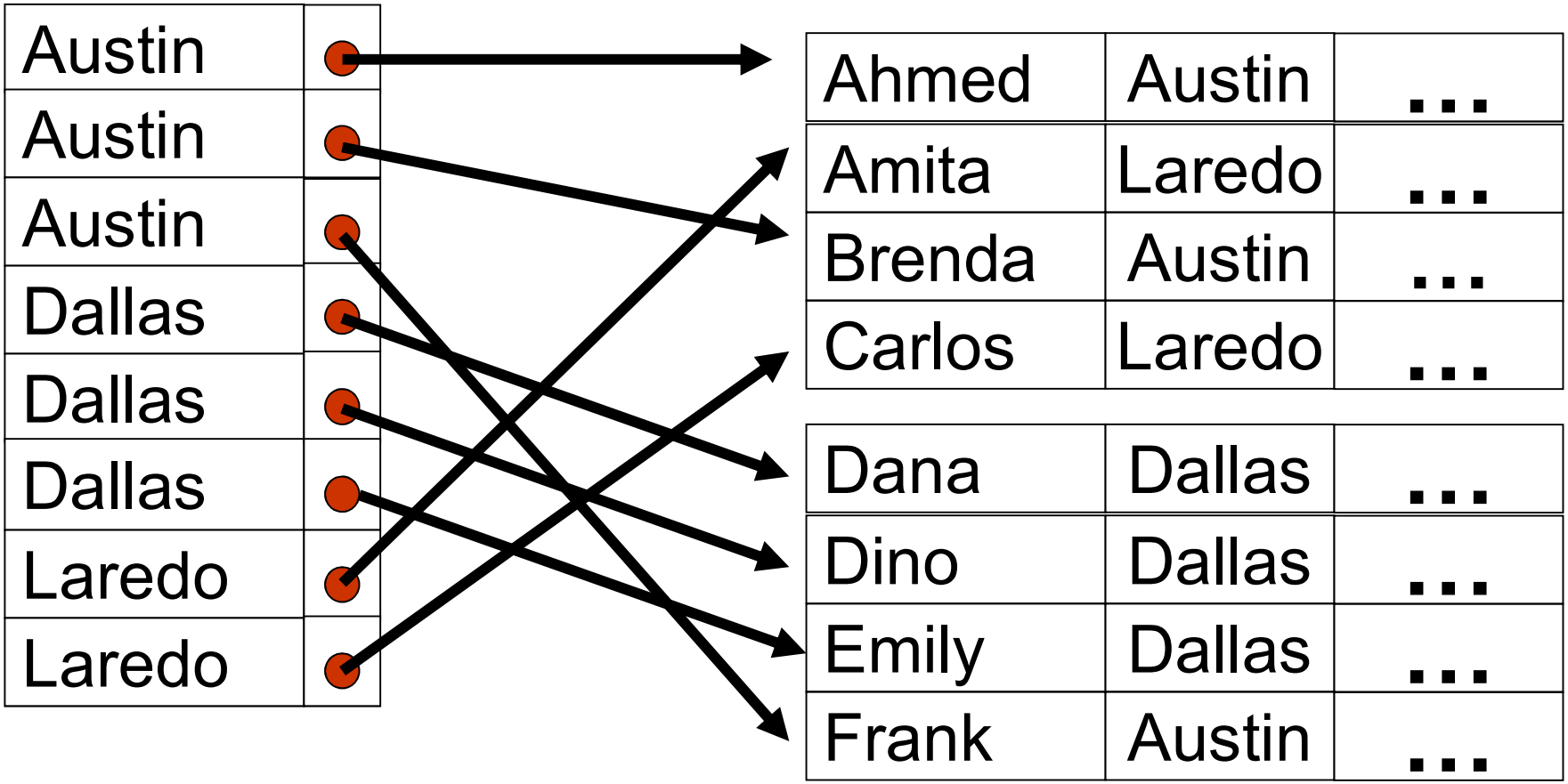
Dense Index:

Sparse Index:

Sparse clustering index



Dense clustering index



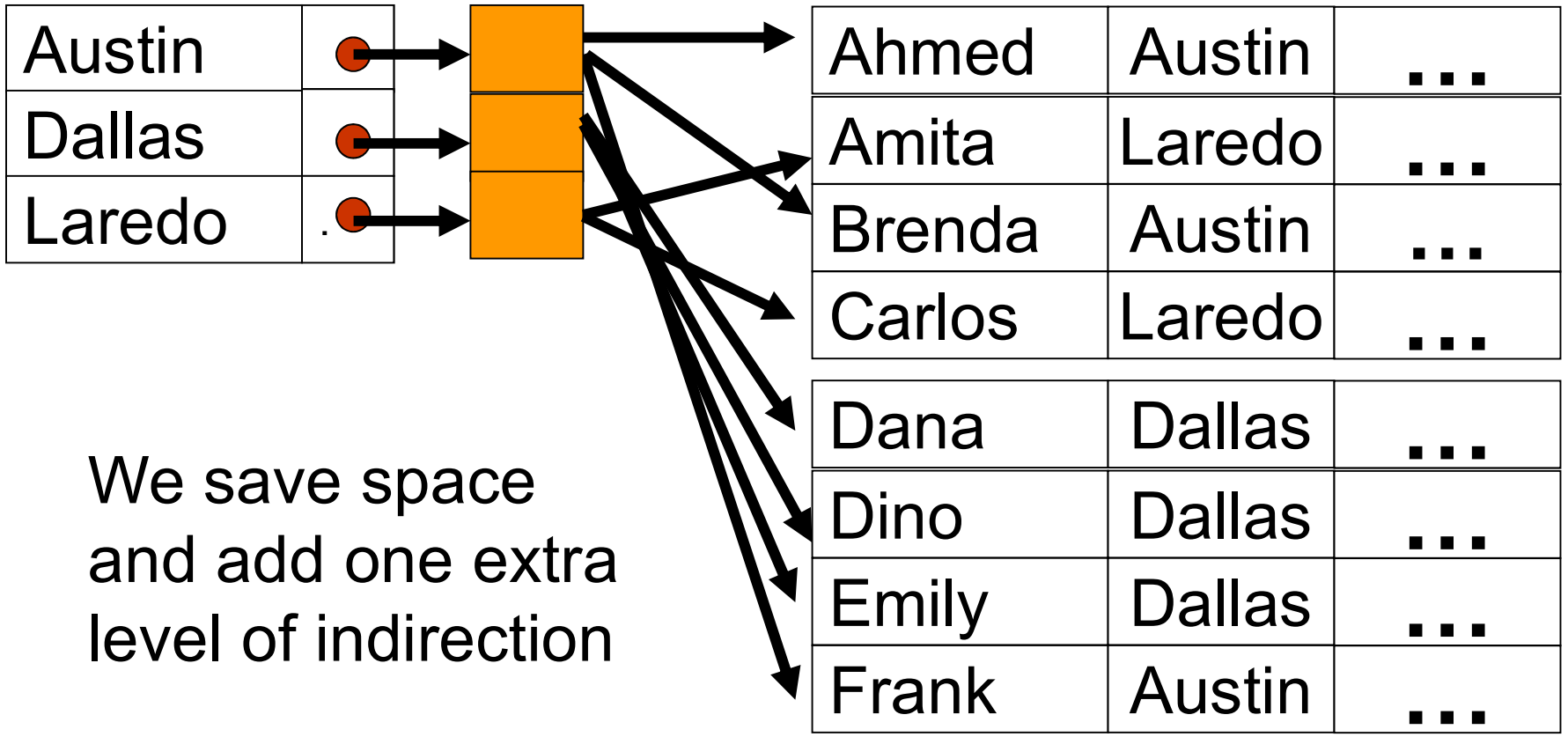
Indexing the index

- When index is very large, it makes sense to index the index
  - Two-level or three-level index
  - Index at top level is called ***master index***
    - Normally a ***sparse index***

- "We can solve any problem by introducing an extra level of indirection, except of course for the problem of too many indirections."

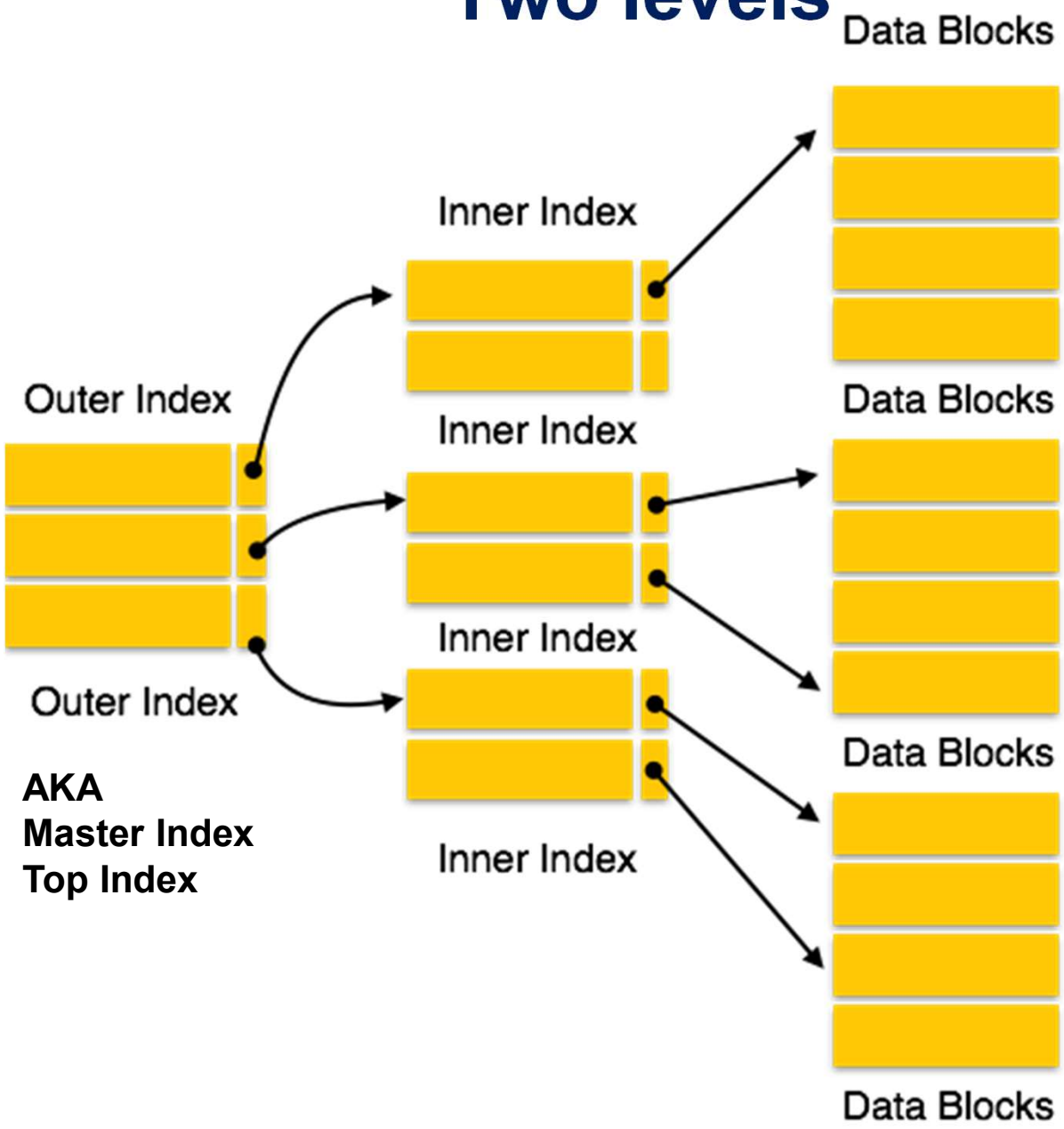
David John Wheeler

Another realization



We save space and add one extra level of indirection

Two levels



AKA
Master Index
Top Index

Multilevel Index

- **Fact:**
  - An **index file** is ***also*** a **data file**

    (It contains **data** !!!)

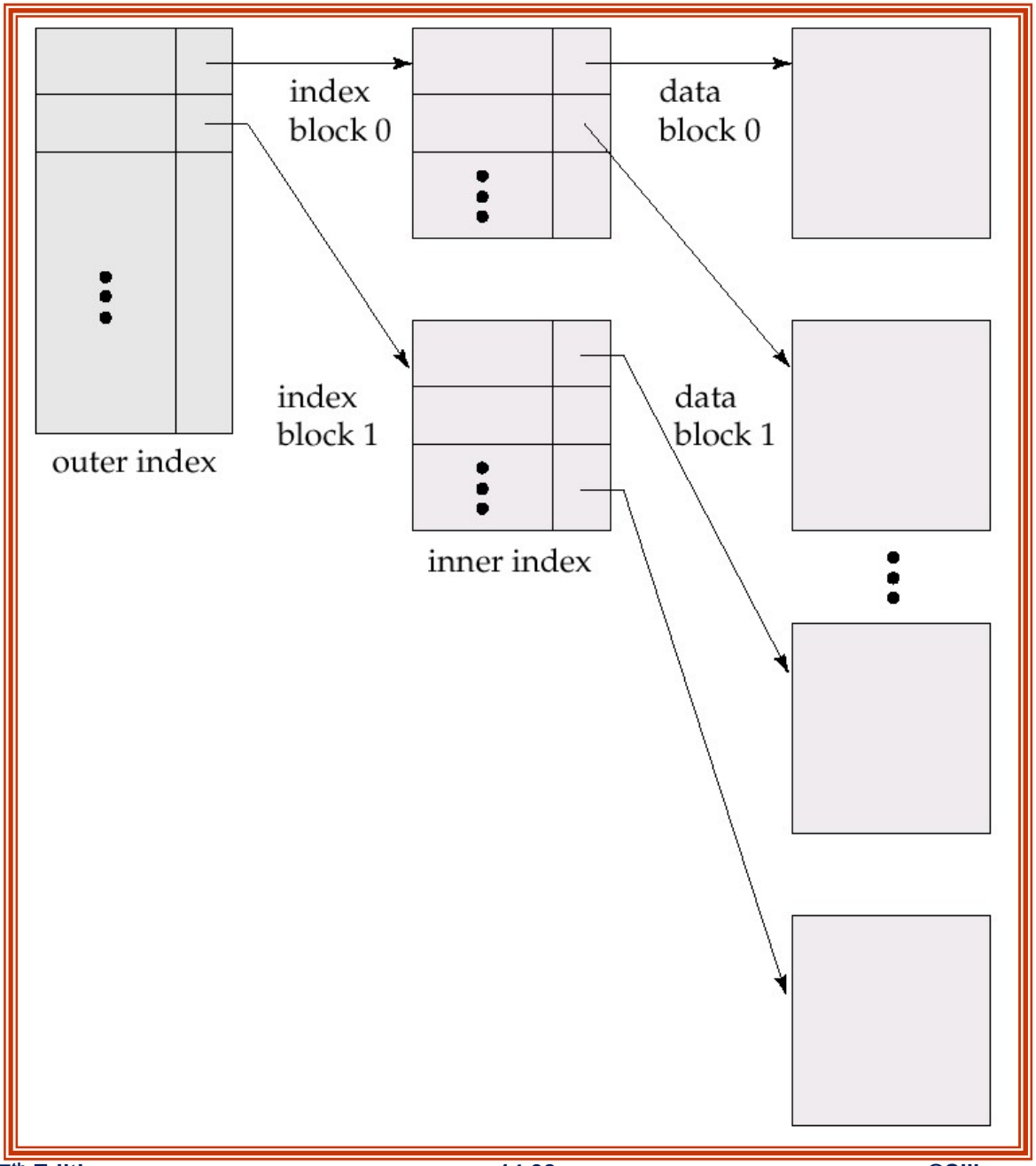
Therefore:

- We can **create** an (**primary**) **index** on an **index file** !!!!
  - ***Multi*-level** index....

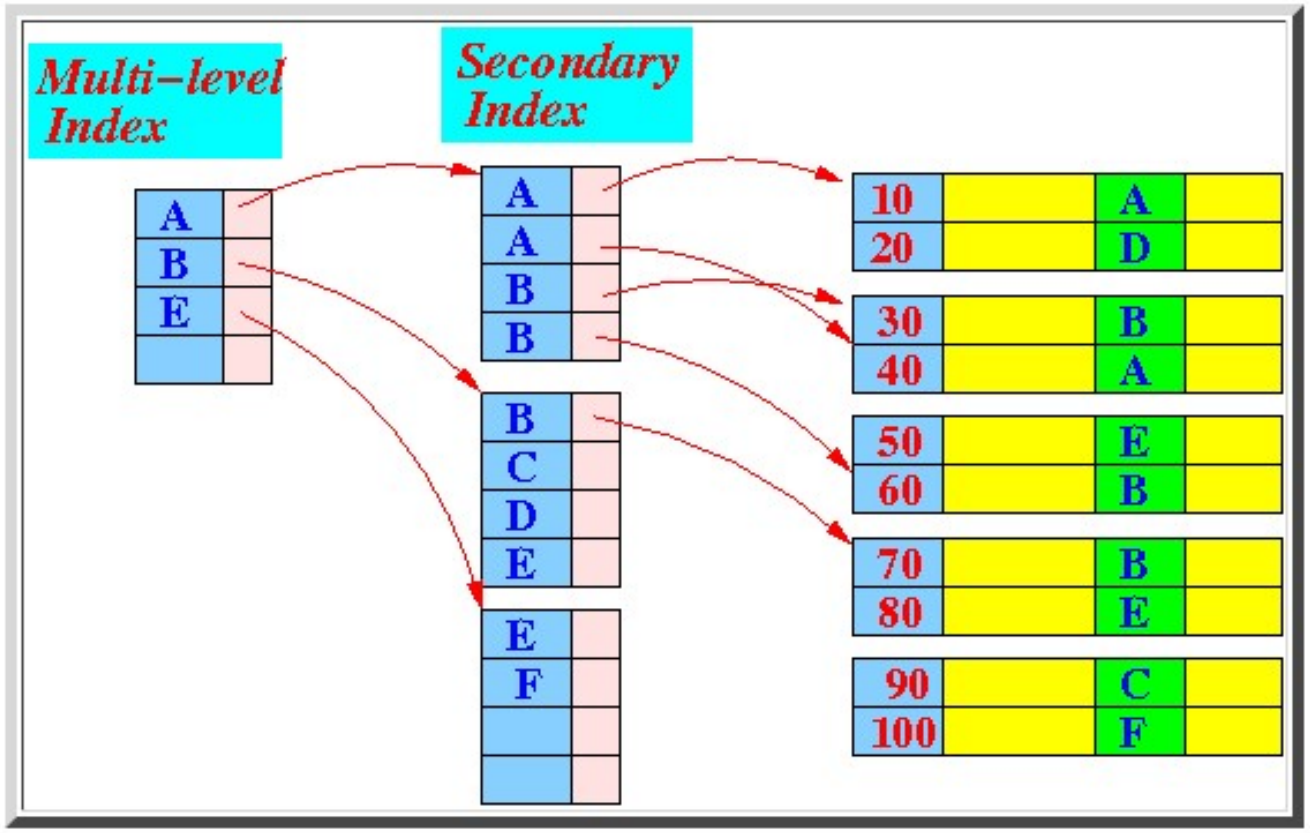
Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, 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.

Multilevel Index (Cont.)

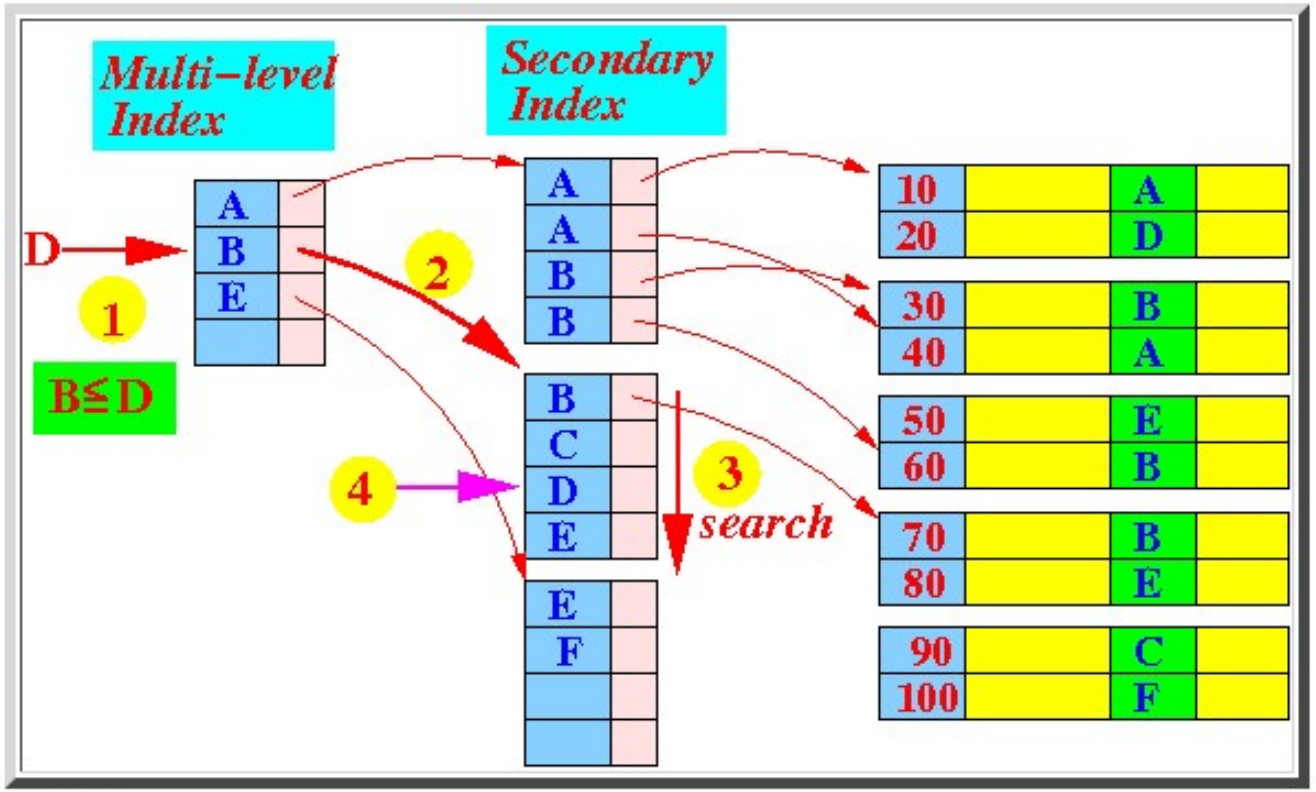


Multilevel Index Example



Multilevel Index

- How to find the search key = D



- Note:

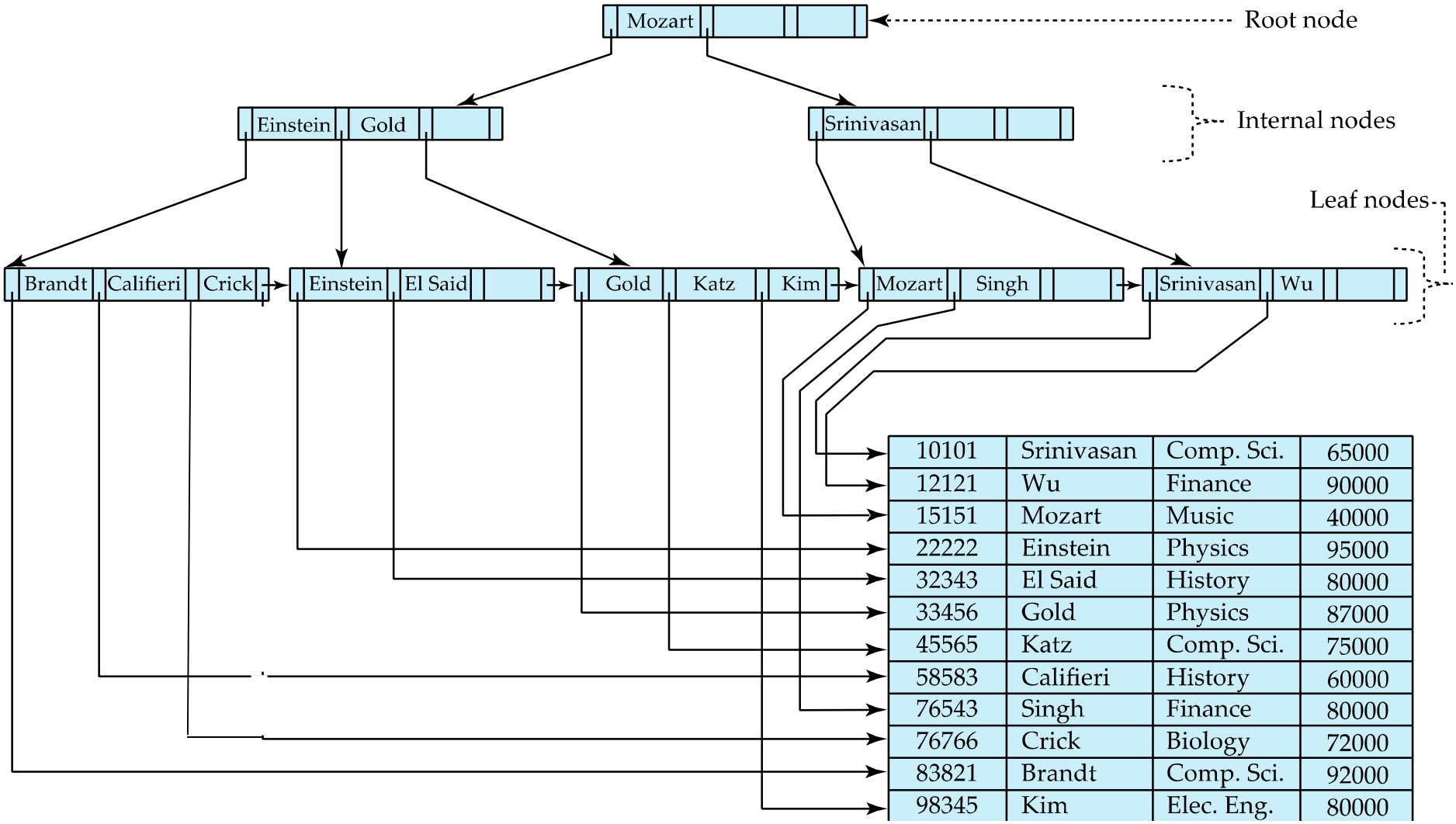
  - We will soon discuss a *dynamic* multi-level index data structure:
    - B-tree / B^+-tree

B-tree and B+-tree

Please watch the video:

https://www.youtube.com/watch?v=aZjYr87r1b8

Example of B+-Tree



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

Creation of Indices

- **E.g.**
  create index *takes_pk* **on** *takes* (*ID,course_ID, year, semester, section*)
  drop index *takes_pk*
- Most database systems allow specification of type of index, and clustering.
- Indices on primary key created automatically by all databases
  - Why?
- Some database also create indices on foreign key attributes
  - Why might such an index be useful for this query:
    - $takes \bowtie \sigma_{name='Shankar'}(student)$
- Indices can greatly speed up lookups, but impose cost on updates
  - Index tuning assistants/wizards supported on several databases to help choose indices, based on query and update workload

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)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

  drop index <index-name>

- Most database systems allow specification of type of index, and clustering.

End of Chapter 14