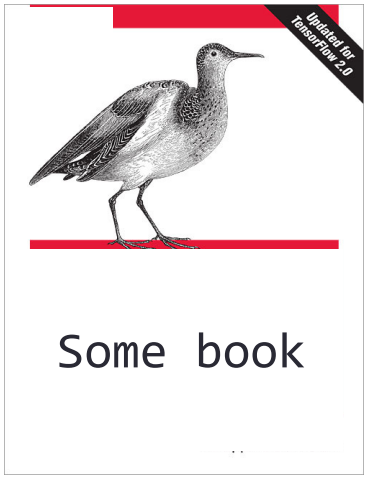
Indexing

CS 4750
Database Systems

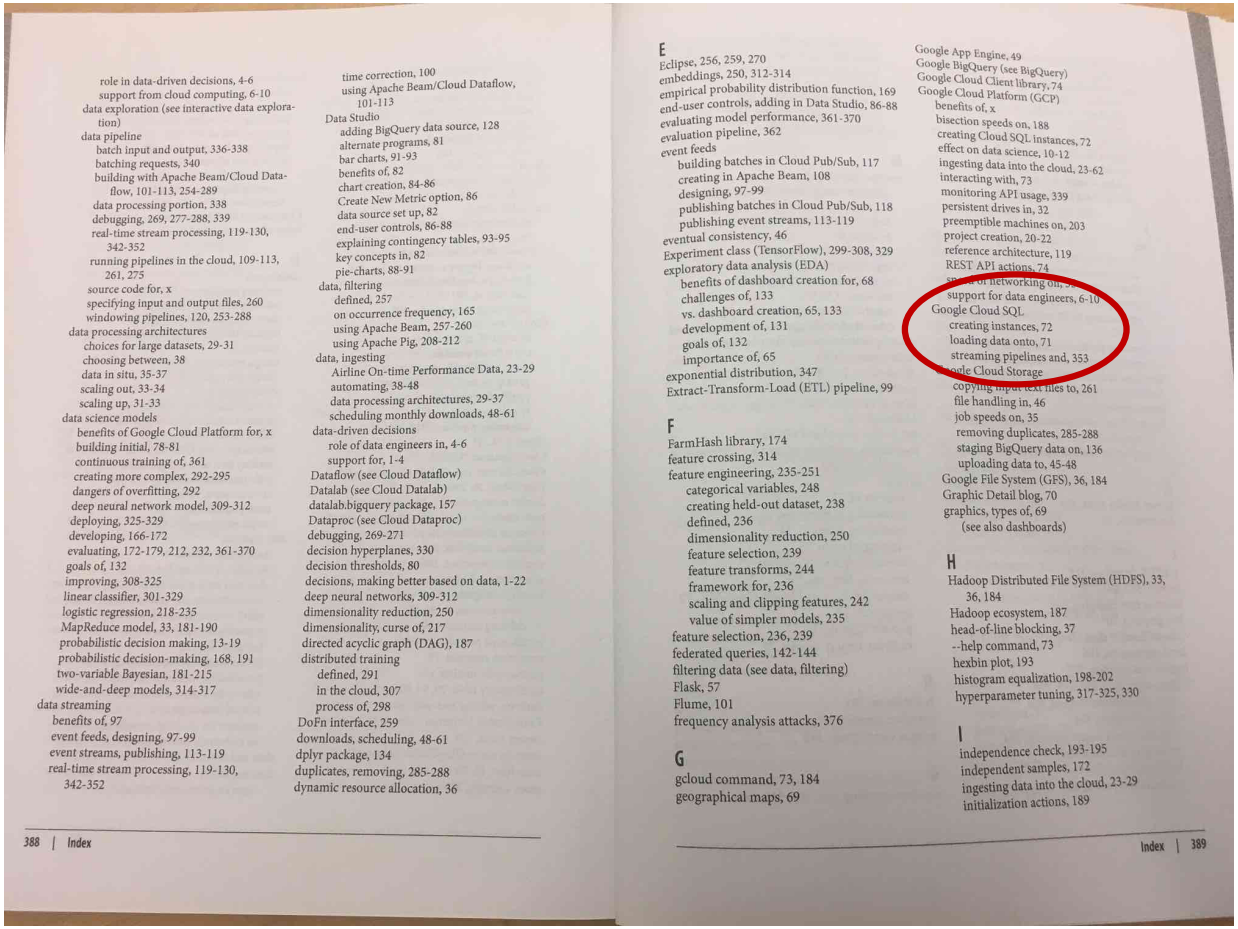
[Silberschatz, Korth, Sudarshan, "Database System Concepts," Ch.14]
[H. Garcia-Molina, J.D. Ullman, J. Widom, Database Systems: The Complete Book, Ch.14]

Indexing in a Book

Find information about "Google Cloud SQL" from



- Look for the keywords in the index
- Find the pages where the words occur
- Read the pages to find the information



Sorted order

Indexing in Database

Example: Find 2nd year students who have taken < 45 credits

There might be 10,000 tuples in `students` relation.

```
SELECT *
FROM students
WHERE year=2 AND credits < 45;
```

Get all 10,000 tuples and test the condition of the `WHERE` clause on each tuple?

- Find which disk block the corresponding record resides
- Fetch the disk block
- Get the appropriate student records

Is there a way to get only the tuples from 2nd year students and then test each of them to see if the credits match?

Indexing in Database

- Indexing = data structure technique to optimize the performance of a database by minimizing the number of disk accesses required when query is processed
- Basic algorithm to search – linear. However, complex search queries (especially with joins) impacts performance
- Indexing helps improve performance

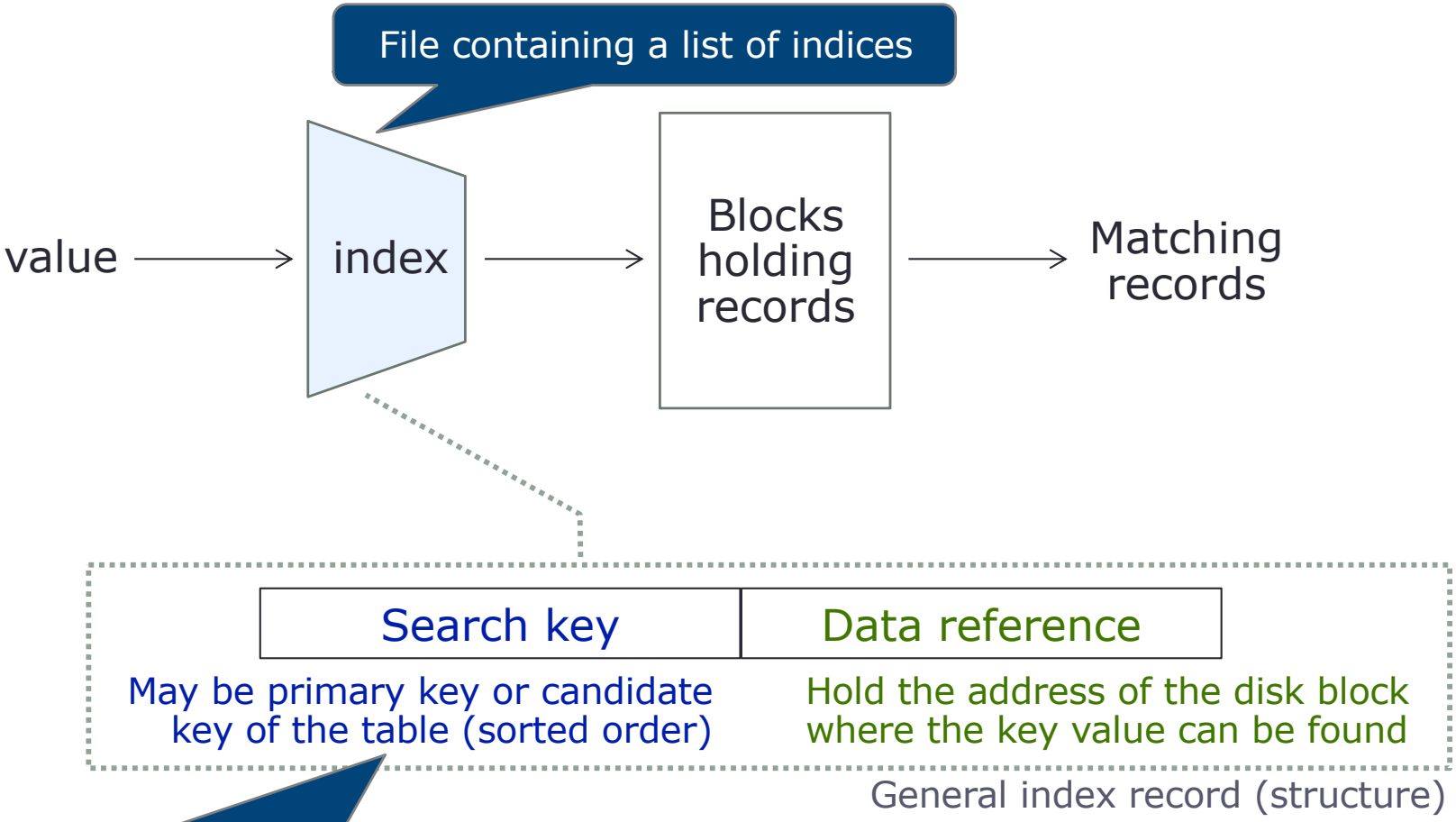
Common structure used by a typical DBMS is B+ tree

The key for the index can be any attribute or set of attributes and need not be the key for the relation on which the index is built.

Rule of thumb: Create an index on the attribute that is used frequently in the search

Indexing in Database

- An index takes a value for some field(s) and finds records with the matching value quickly



Selection of Indexes

- An index on an attribute may speed up the execution of those queries in which a value, or range of values, is specified for that attribute, and may speed up joins involving that attribute.

- On the other hand, every index built for one or more attributes of some relation makes insertions, deletions, and updates to that relation more complex and time-consuming.

Database designers must analyze the trade-off

Which Indexing Technique to Use

Aspects that must be considered:

- Access types
  - Finding records with a specified attribute value (search key), or
  - Finding records with attribute value based on a specified range
- Access time
  - Time needed to find a particular data item or set of data items
- Insertion time
  - Time to find the place to insert + time to insert a new data item + time to update the index structure
- Deletion time
  - Time to find the data item to be deleted + time to delete the data + time to update the index structure
- Space overhead
  - Space occupied by an index structure (vs. performance)

Types of File Organization Mechanism

Sequential file organization (or Ordered index)

- Indices based on sorted ordering of the values
- Generally fast
- Basic / traditional structure that most DBs use

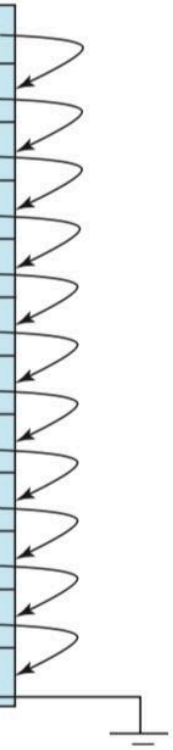
Hash file organization (or Hash index)

- Indices based on a uniform distribution of values across a range of buckets
- Hash function determines a value assigned to a bucket

Example: Sequential File

Instructor

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	

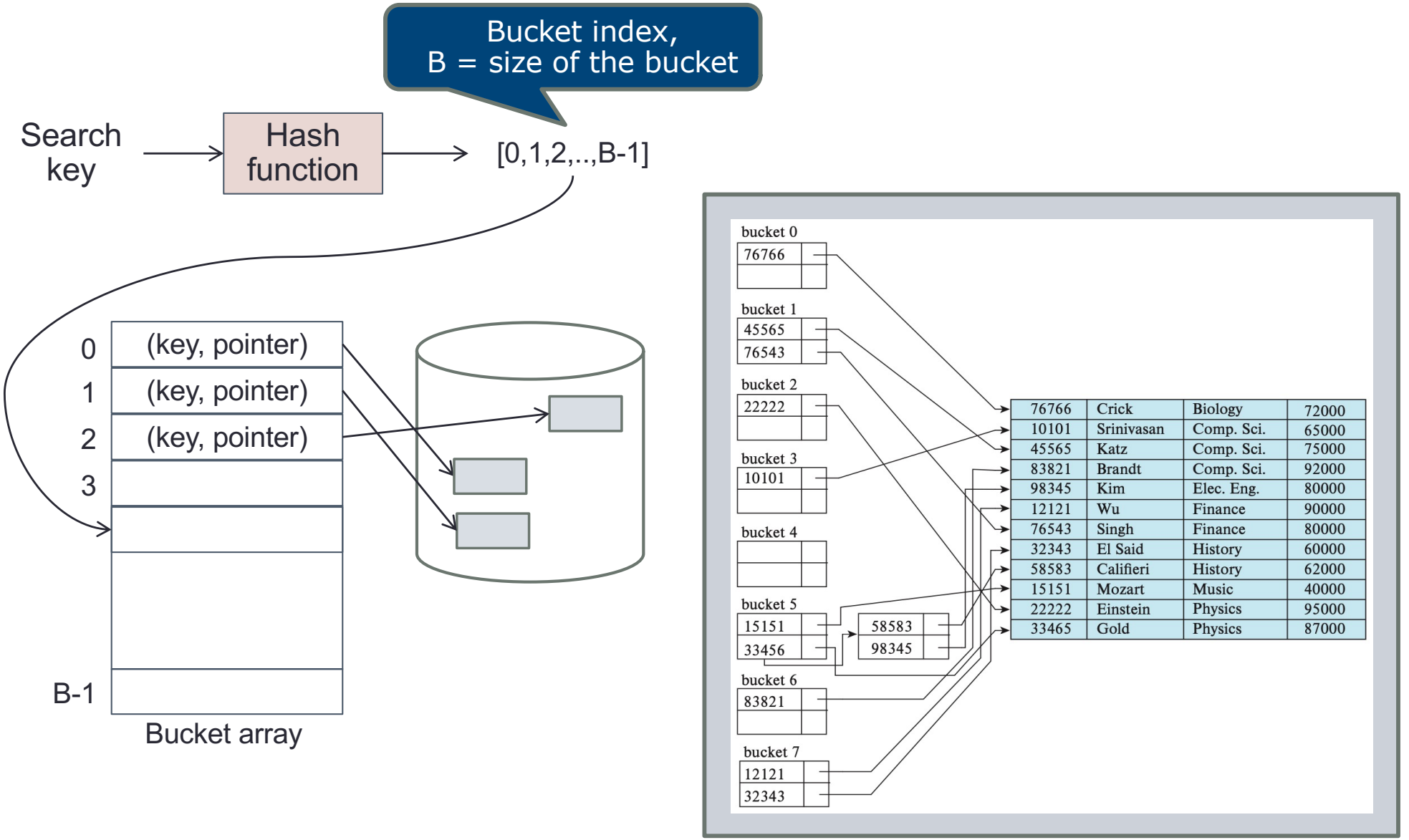


Instructor records

- The records are stored in sorted order of instructor's ID (used as a search key)
- Search key defines the sequential order of the file

[Ref: Figure 11.1, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 477]

Example: Hash File



[Ref: based in part on Figure 24.6, Silberschatz, Korth, Sudarshan, "Database System Concepts," 7th Ed., page 1191]

Ordered Index Structures

A file may have several indices, on different search keys

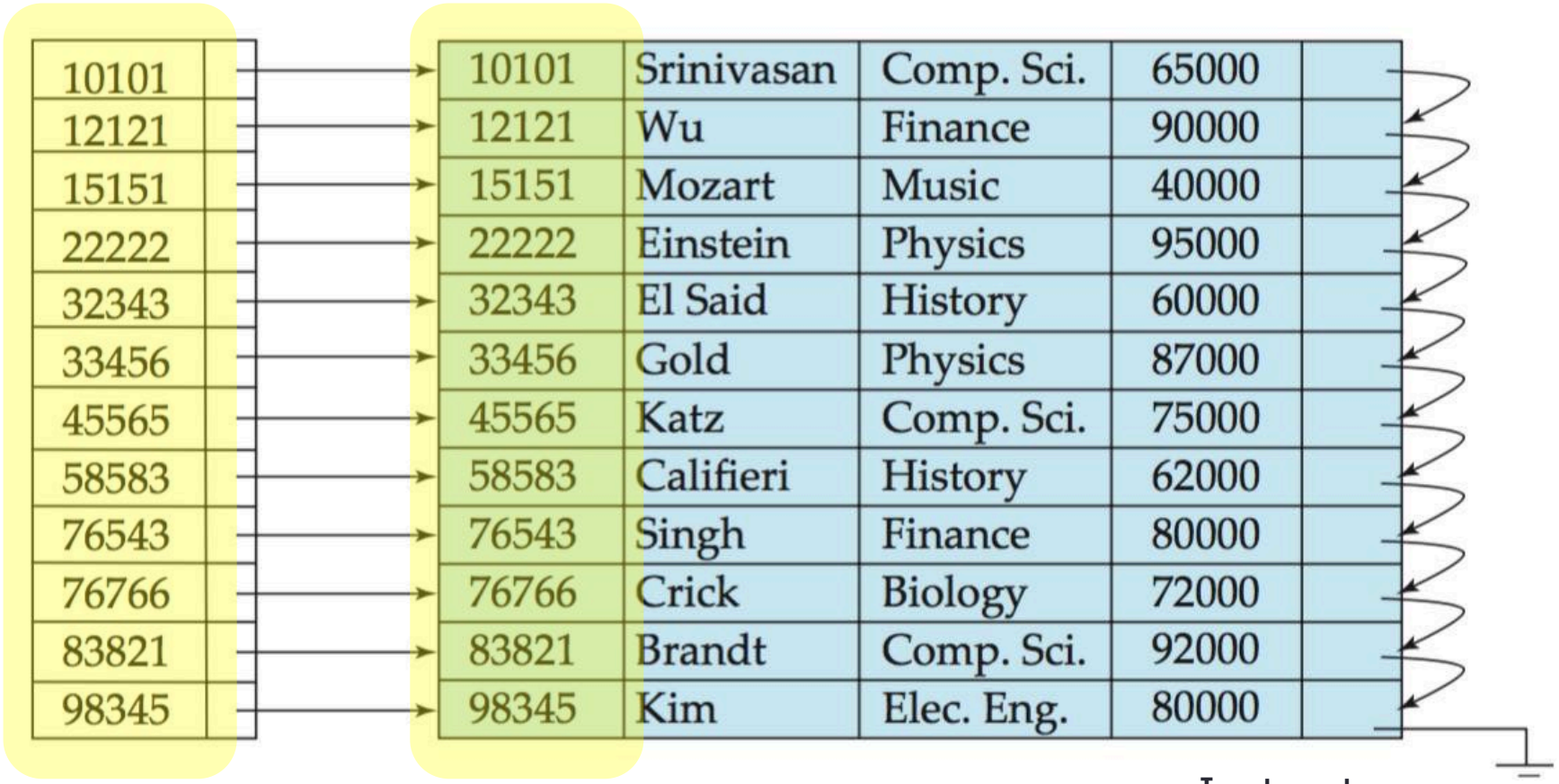
- **Primary index (or clustered index)**
  - Search key defines the sequential order of the file
  - Search key of a clustering index is often the primary key (but not necessarily so)

- **Secondary index (or unclustered index)**
  - Search key specifies an order different from the sequential order of the file
  - Use an extra-level of indirection to implement a secondary index, containing pointers to all the records

Example: Primary Index

Search key defines the sequential order of the file

Fast but can result in unnecessary indices and big space needed

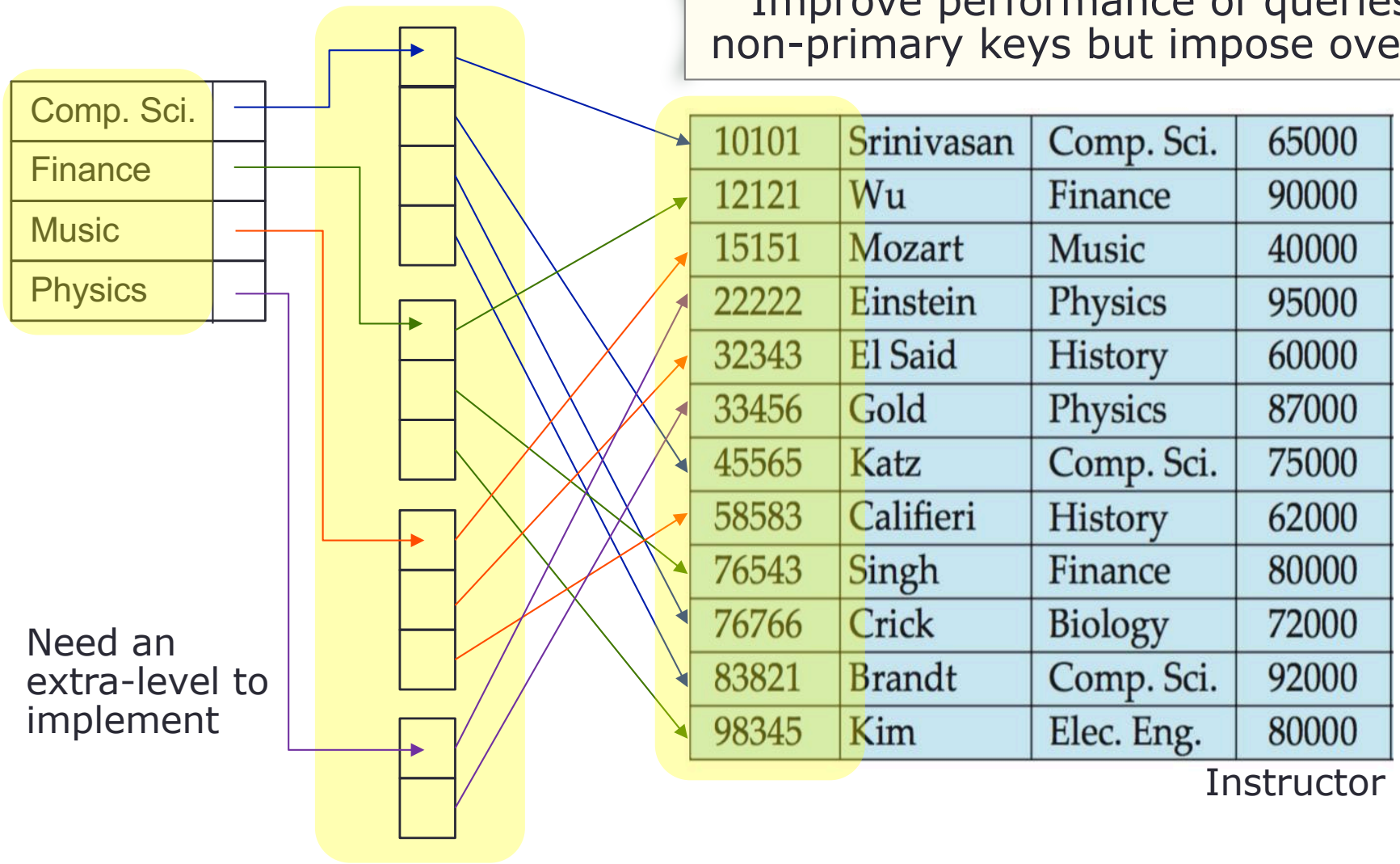


[Ref: Figure 11.2, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 478]

Example: Secondary Index

Search key specifies an order different from the sequential order of the file

Improve performance of queries on non-primary keys but impose overhead



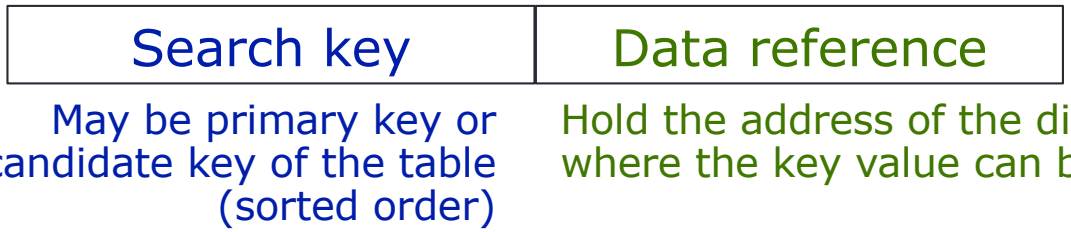
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	62000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000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

Instructor

[based in part on Figure 11.6, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 484]

Ordered Index

- Created on the basis of the key of the table
- Ordered file with fixed, two fields



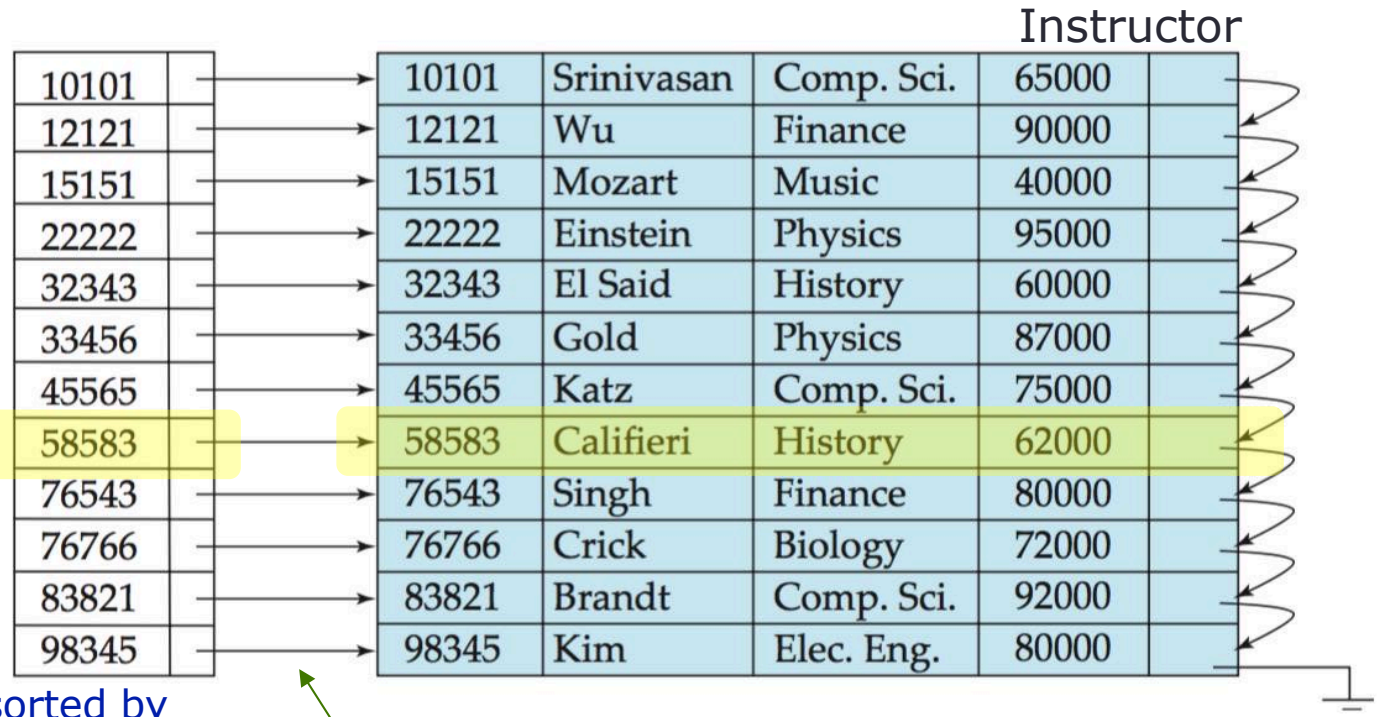
- Unique to each record (i.e., 1:1 mapping)
- Since primary keys are stored in sorted order, the performance of the search operation is quite efficient
- Two types:
  - Dense index
  - Sparse index

Dense Index

- A record is created for every search key value
- Need more space to store index records
- Example: a search key is a primary key

Find instructor with ID "58583"

A primary index that is dense on a primary-key, ordered table is redundant (requiring unnecessary spaces)



Instructor file is sorted by instructor's *ID* (every search key valued)

Point to the real record on the disk

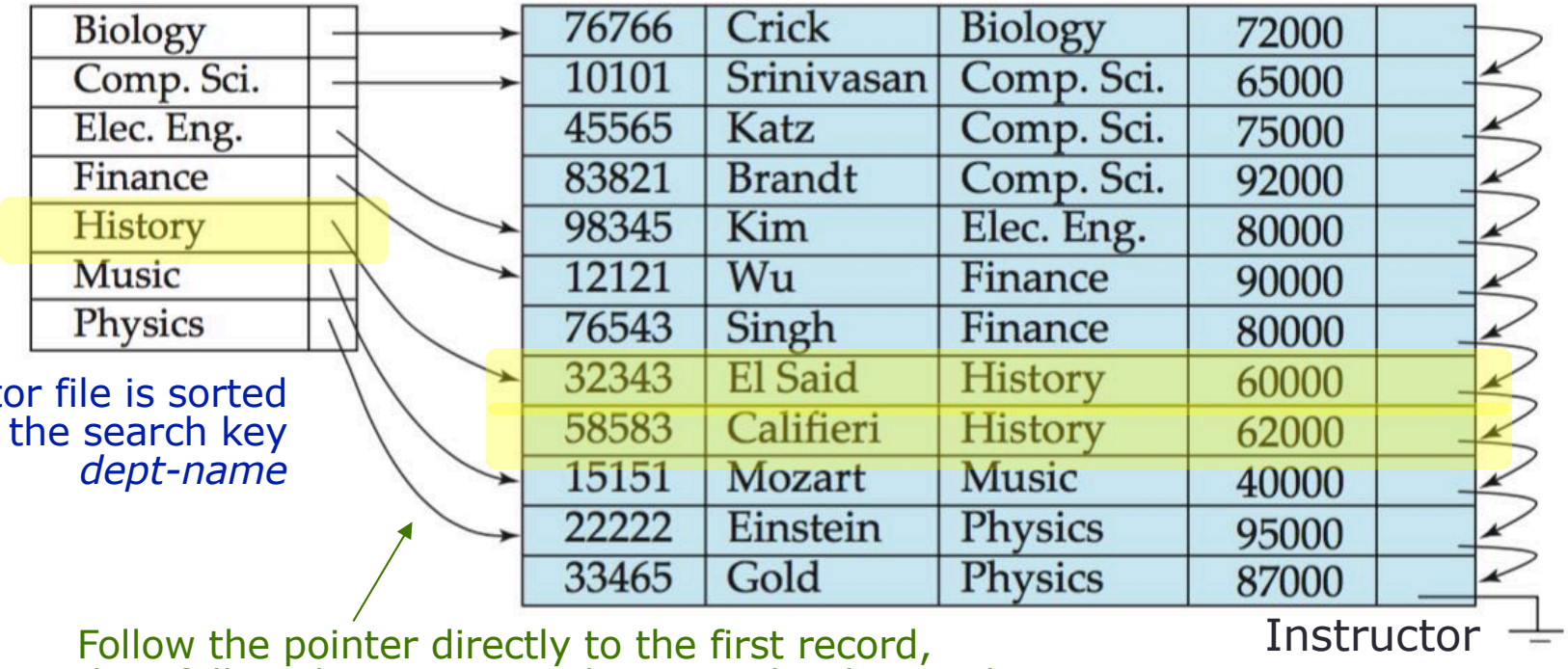
[Ref: Figure 11.2, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 478]

Dense Index

- Support range queries
- Example: a search key is not a primary key

Find a history instructor with ID "58583"

Pointer points to the first data record with the search-value. The rest of the records are sorted on the same search key



Instructor file is sorted on the search key *dept-name*

Follow the pointer directly to the first record, then follow the pointer in that record to locate the next record in search key order until the desired record is found

[Ref: Figure 11.4, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 480]

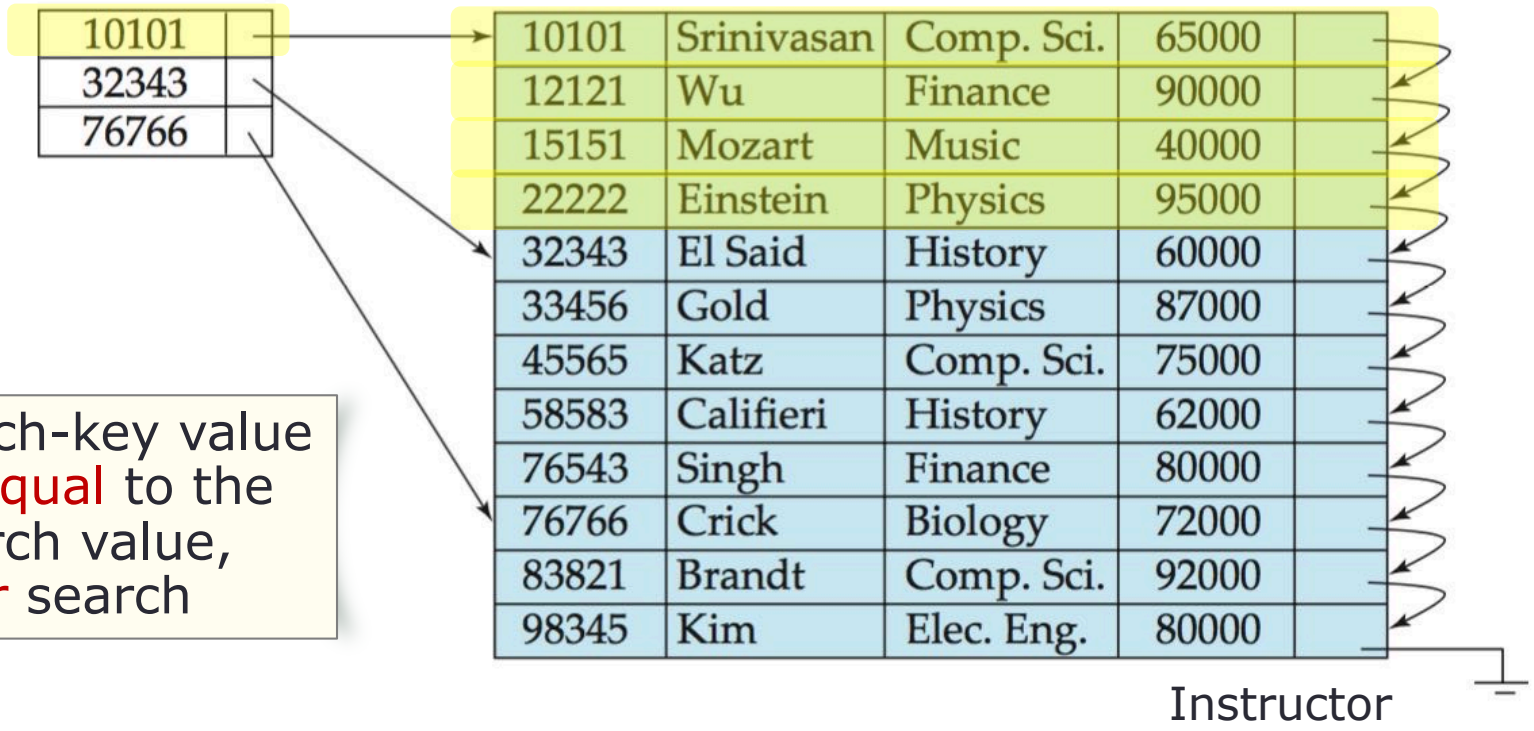
Dense Index: Lookup

- Given a search key *K*, the index is scanned
  - When *K* is found, the associated pointer to the data file recorded is followed and the block containing the record is read in main memory
- When dense indexes are used for non-primary key, the minimum value is located first
  - Consecutive blocks are loaded in main memory until a search key greater than the maximum value is found
- The index is usually kept in main memory. Thus one disk I/O has to be performed during lookup
- Since the index is sorted, a binary search can be used.
  - If there are *n* search keys, at most $log_2 n$ steps are required to locate a given search key
- Query-answering using dense indices is efficient

Sparse Index

- Used when dense indices are too large
- One key-pointer pair per data block
- Can be used only if the relation is stored in sorted order of the search key

Find instructor with ID "22222"



Start with search-key value less than or equal to the desired search value, then linear search

[Ref: Figure 11.3, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 479]

Sparse Index: Lookup

- Given a search key *K*,
  - Search the sparse index for the greatest key ≤ *K*, using binary search
  - Retrieve the pointed block to main memory to look for the record with search key *K* (linear search vs. binary search)
- The index is usually kept in main memory. Thus one disk I/O has to be performed during lookup
- Efficient in space but may require more computation time due two binary searches
  - Search on the sparse index
  - Search on the retrieved data block

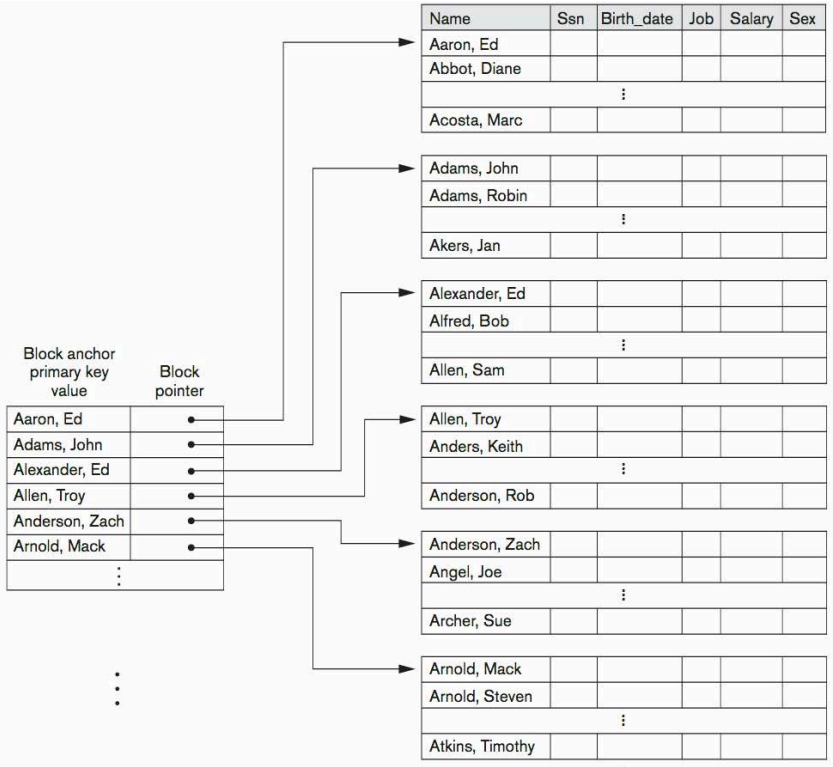
Dense Index vs. Sparse Index

- A primary index that is dense on an ordered table is redundant
- Thus, a primary index on an ordered table is always sparse

- Dense indices are faster in general
- Sparse indices require less space and impose less maintenance for insertions and deletions

- Try to have a sparse index with one entry per block

Try to keep index size small



[Ref: Figure 18.1, Elmasri Navathe, "Fundamentals of Database Systems," 6th Ed., page 634]

More Info on Indexing

- When a primary key is created for a table, a table is ordered based on the primary key
- At least one sparse index is created on that record to reduce search time
- If a column (or some columns) is declared as unique, a secondary index is created
- Every index introduces more data and more overhead (especially when doing insert, delete, or update)

Can we create (or add) indices to just any table (or DB)? – **No!**

Read-heavy DBs – **can index a lot** (if space allows)
Write-heavy DBs – **index sparingly** (take a balanced approach)
Write-ONLY DBs – **one or no index**

Multi-Level Indices

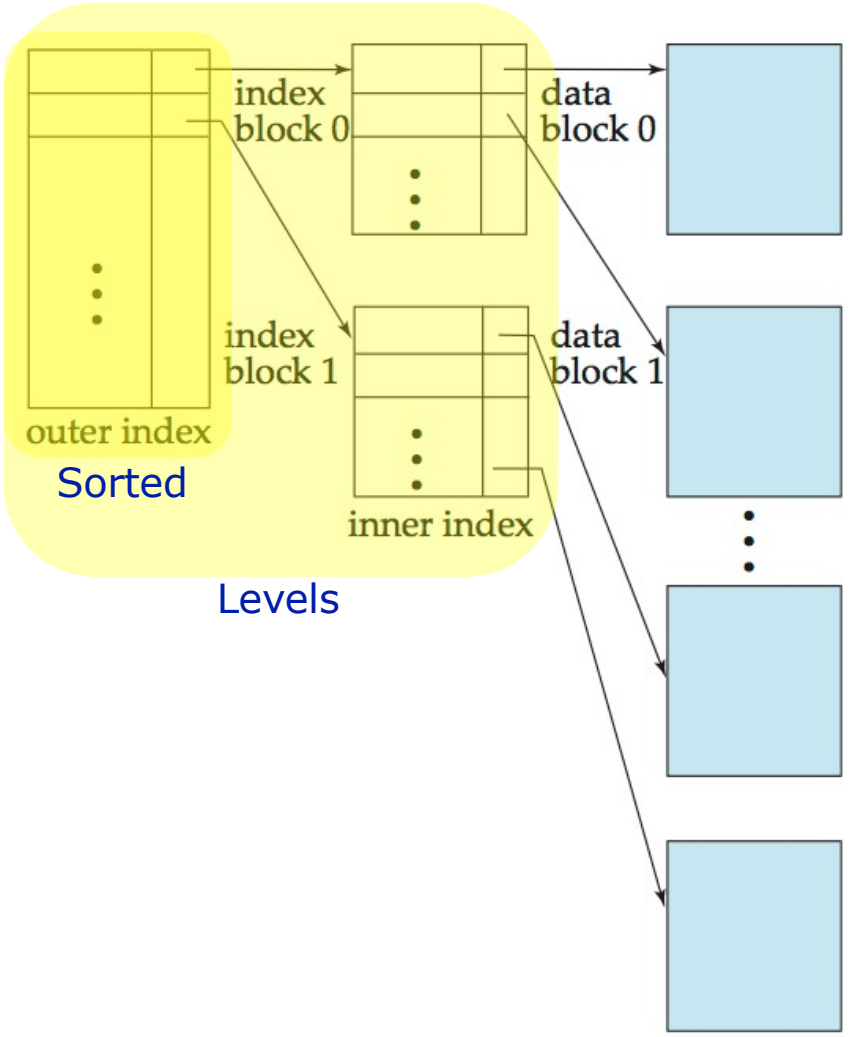
- If an index is small enough to be kept entirely in main memory, the search time to find an entry is low

- If index is too large to be kept in main memory, index blocks must be fetched from disk when required. One search results in several disk-block reads
  - If no overflow blocks in the index → use binary search
  - If overflow blocks → use sequential search

- Solution:
  - Use a sparse index on the index

Example: Two-Level Sparse Index

- Use binary search on outer index
- Scan index block until the correct record is found
- Scan block pointed to for desired record
- For very large files, add additional level of indexing to improve search performance
- Must update indices at all levels when perform insertion or deletion



[Ref: Figure 11.5, Silberschatz, Korth, Sudarshan, "Database System Concepts," 6th Ed., page 481]

Updating Indices

All associated indices must be updated when a record is inserted into or deleted from a file

Insertion:

- Find a place to insert
- For dense index:
  - Insert search key value if not present
- For sparse index:
  - No change unless a new block is created
  - If the first search key value appears in the new block, insert the search key value into the index

Deletion:

- Find the record
- If it is the last record, delete that search key value from index
- For dense index:
  - Delete the search key value
- For sparse index:
  - Delete the search key value
  - Replace the key value's entry index with the next search key value if not already present