Indexing and Hashing: Basic concept, Ordered Indices

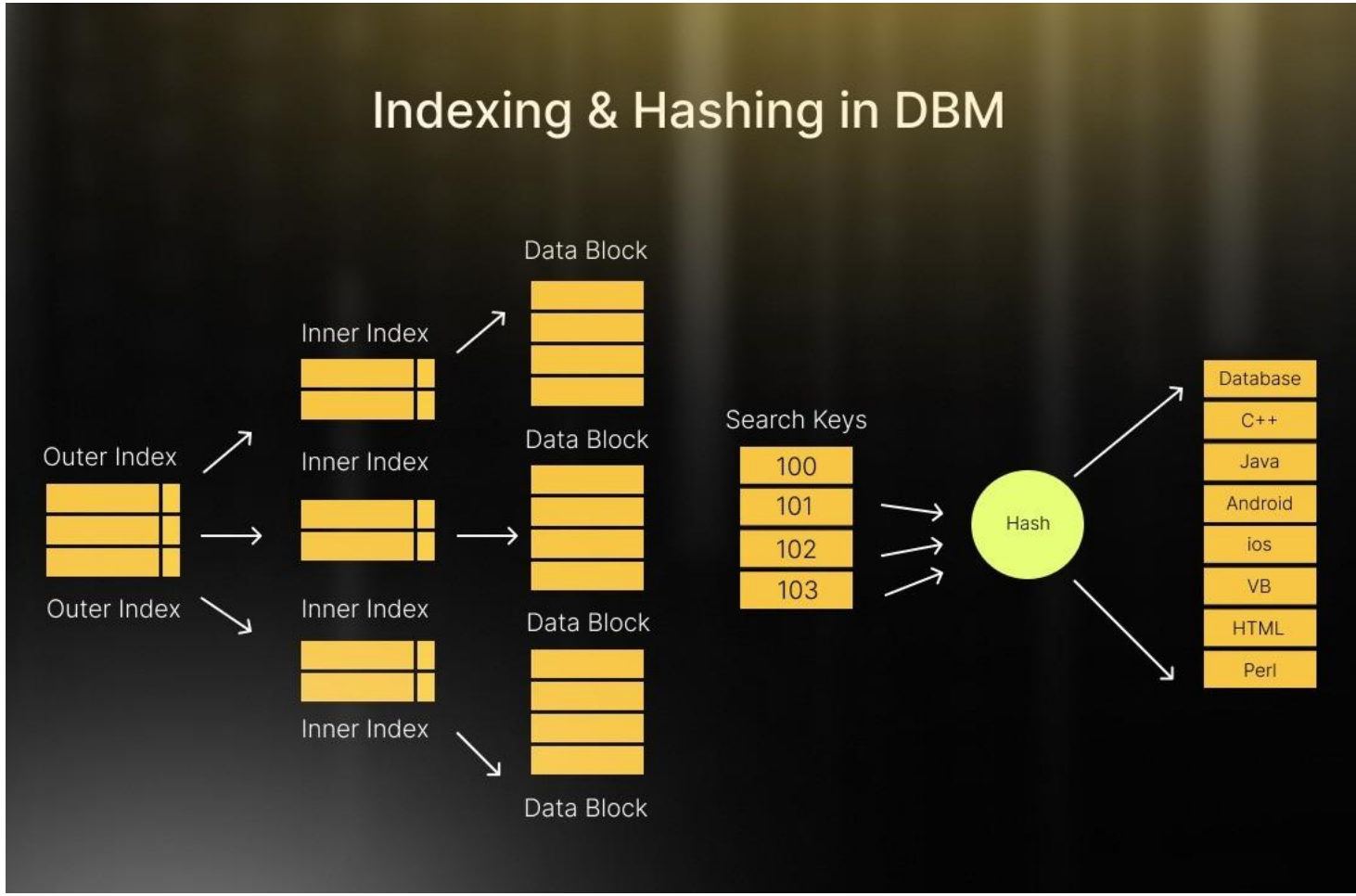
ADBMS

Indexing and Hashing

Introduction

In the ever-evolving world of data management, efficient access and retrieval of information lie at the heart of a well-designed database system.

Two powerful techniques, **Indexing** and **Hashing**, have emerged as key players in optimizing database performance.



Indexing in DBMS

Indexing serves as a **gateway to rapid data retrieval** in a database. Its primary objective is **to reduce the number of disk accesses required** when processing queries, making it a valuable asset for improving database performance.

How does it work?

When an index is created for a particular field in a database table, it generates a specialized data structure that holds the field value alongside a pointer to its corresponding record. These indexes can be developed using one or more columns from the table, enabling rapid access to data without the need for time-consuming full table scans.

Hashing in DBMS

Hashing revolves around using mathematical functions, known as **hash functions**, to calculate direct locations of data records on a disk.

But how is it different from Indexing, and why is it an invaluable asset for specific database tasks?

Unlike Indexing, Hashing doesn't rely on index structures to access data. Instead, it generates unique addresses for data records using hash functions, which take search keys as parameters. This direct calculation of data locations on the disk allows for faster retrieval, making Hashing an ideal choice for large databases.

Types:

- Static Hashing
- Dynamic Hashing

Consider a table: Faculty(Name, Phone)

Index on "Name"

Name	*Pointer*
Anupam Basu	2
Pabitra Mitra	6
Partha Pratim Das	1
Prabir Kumar Biswas	7
Rajib Mall	5
Ranjan Sen	3
Sudeshna Sarkar	4

Table "Faculty"

Rec #	*Name*	*Phone*
1	Partha Pratim Das	81998
2	Anupam Basu	82404
3	Ranjan Sen	84624
4	Sudeshna Sarkar	82432
5	Rajib Mall	83668
6	Pabitra Mitra	81664
7	Prabir Kumar Biswas	84772

Index on "Phone"

Pointer	*Phone*
6	81664
1	81998
2	82404
4	82432
5	83668
3	84624
7	84772

- How to search on Name?
  - Get the phone number for ‘Pabitra Mitra’
  - Use “Name” Index – sorted on “Name”, search “Pabitra Mitra” and navigate on pointer (rec #)
- How to search on Phone?
  - Get the name of the faculty having phone number = 84772
  - Use “Phone” Index – sorted on “Phone”, search “84772” and navigate on pointer (rec #)

- We can keep the records sorted on “Name” or “Phone” (called primary index) but not on both.

Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - For example:
    - Name in a faculty table
    - 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

- 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”

Index Evaluation Metrics

- Access types supported efficiently. For example,
  - 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

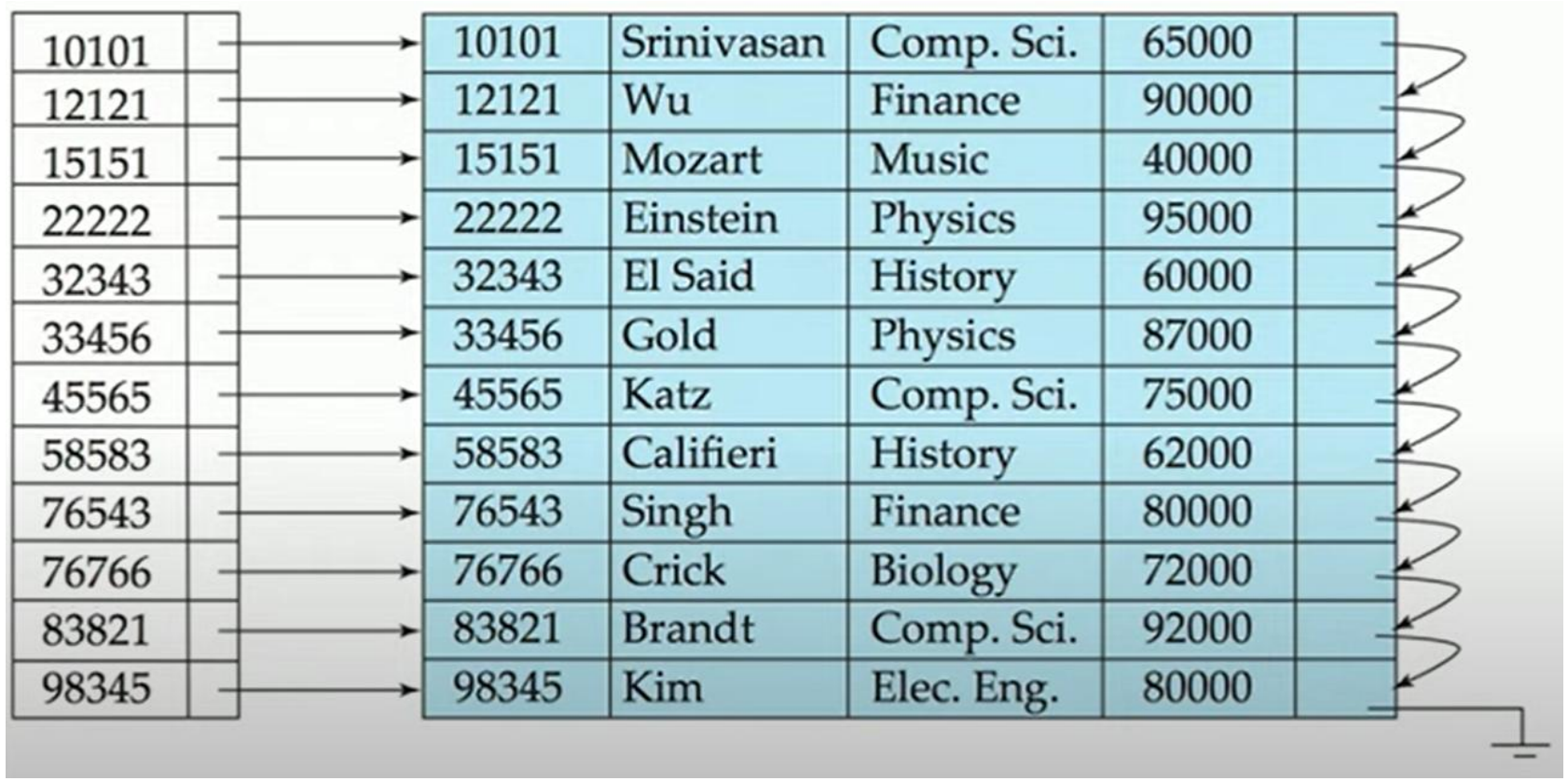
Ordered Indices

Ordered indices

- In an **ordered index**, index entries are stored sorted on the search key value. For example, author catalog in library
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file
- **Index-sequential file**: ordered sequential file with a primary index

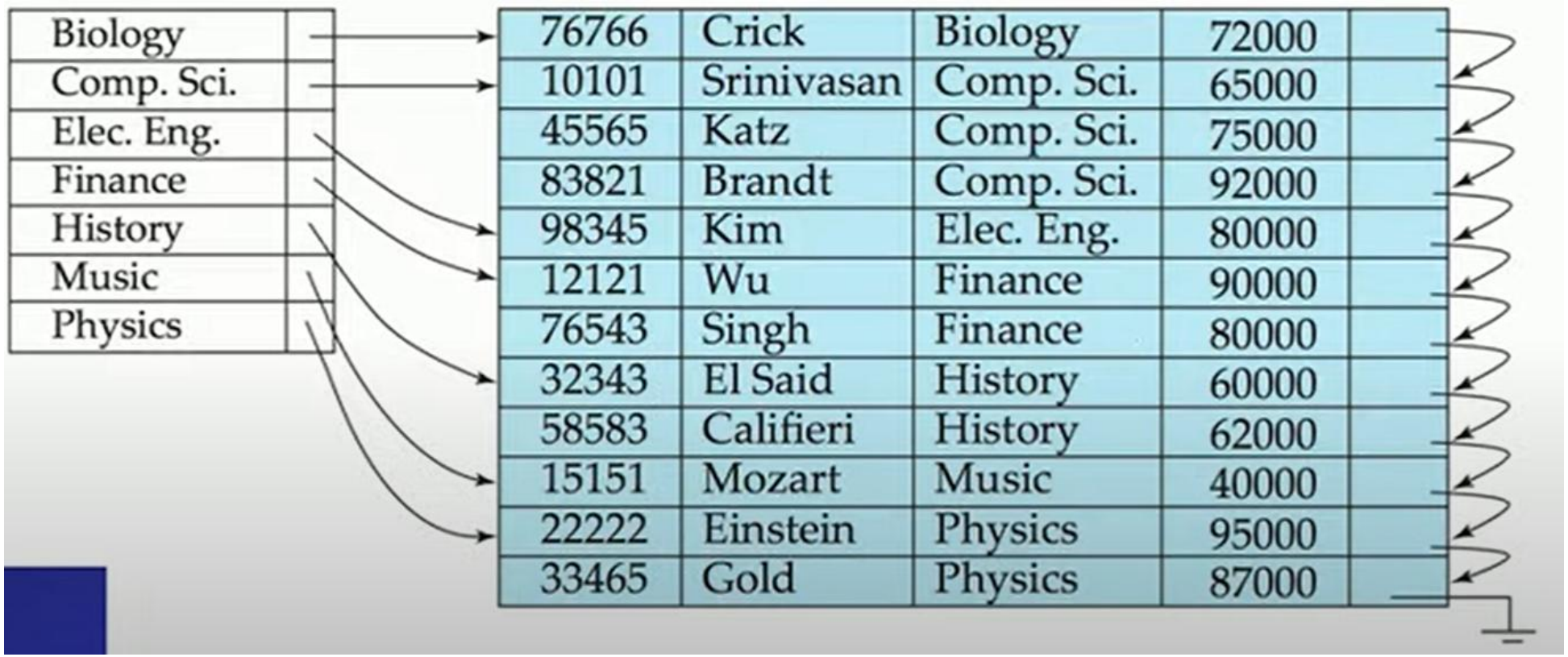
Dense Index Files

- Dense index – Index record appears for every search-key value in the file.
- E.g. index on ID attribute of instructor relation



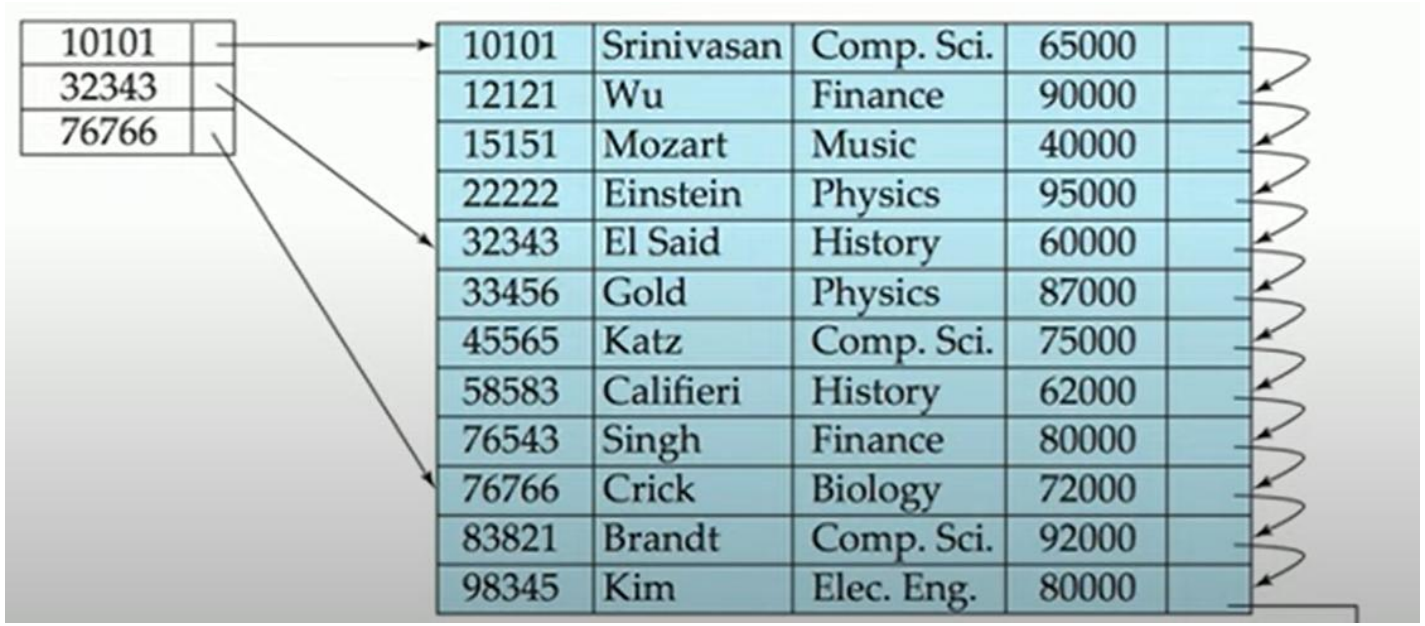
Dense Index Files (Cont.)

- Dense index on dept_name, with instructor file sorted on dept_name



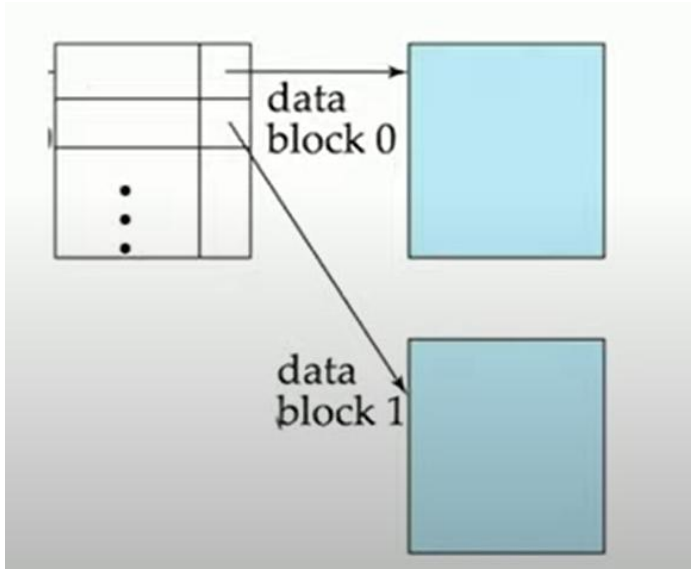
Sparse Index Files

- 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

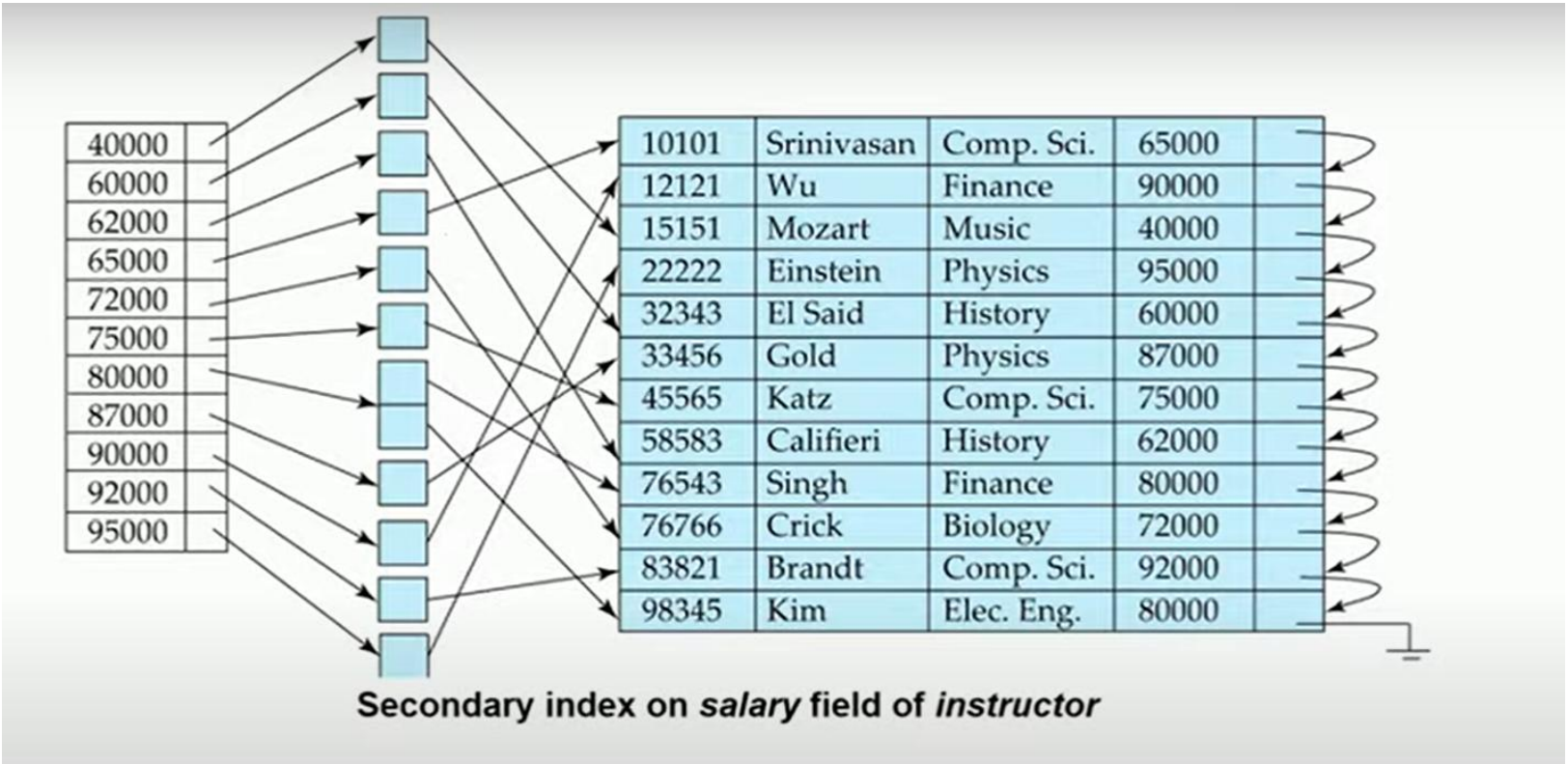


Sparse Index Files (Cont.)

- Compared to dense index:
  - Less space and less maintenance overhead for insertion and deletions
  - Generally slower than dense index for locating records
- **Good trade-off**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block



Secondary Indices Example



Secondary index on *salary* field of *instructor*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

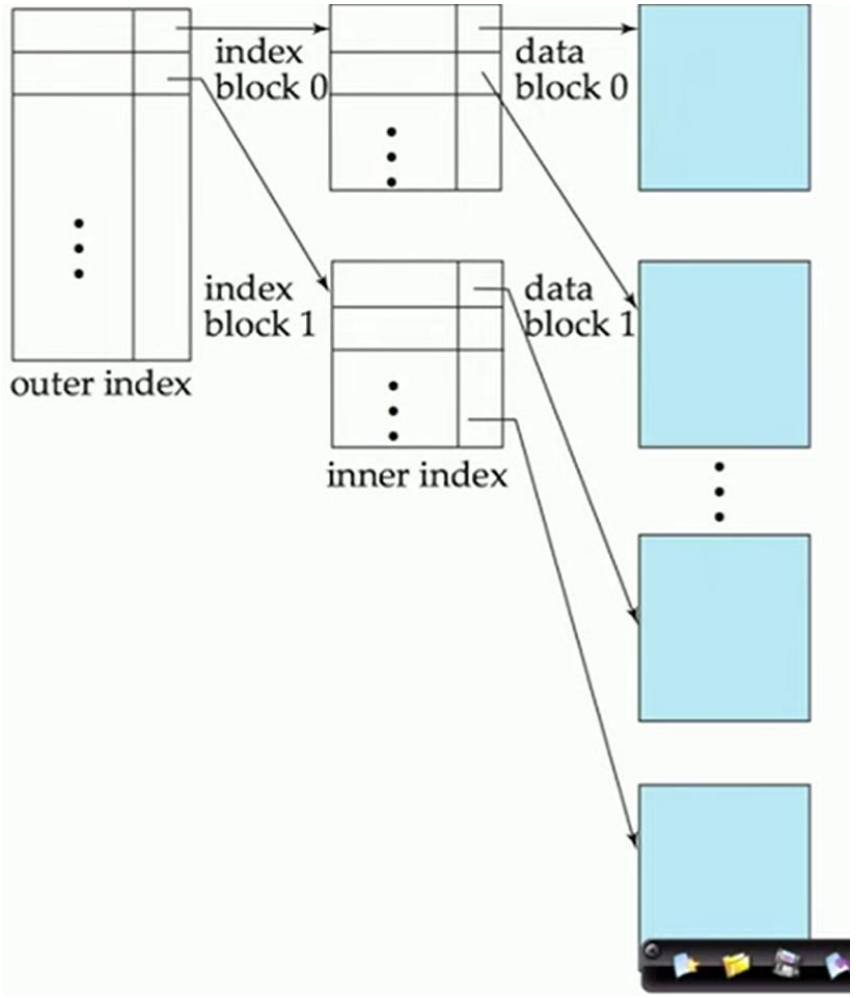
Primary and Secondary Indices

- Indices offer substantial benefits when searching for records
- BUT: Updating indices imposes overhead on database modification – when a file is modified, every index on the file must be updated
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds, versus about 100 nanoseconds for memory access

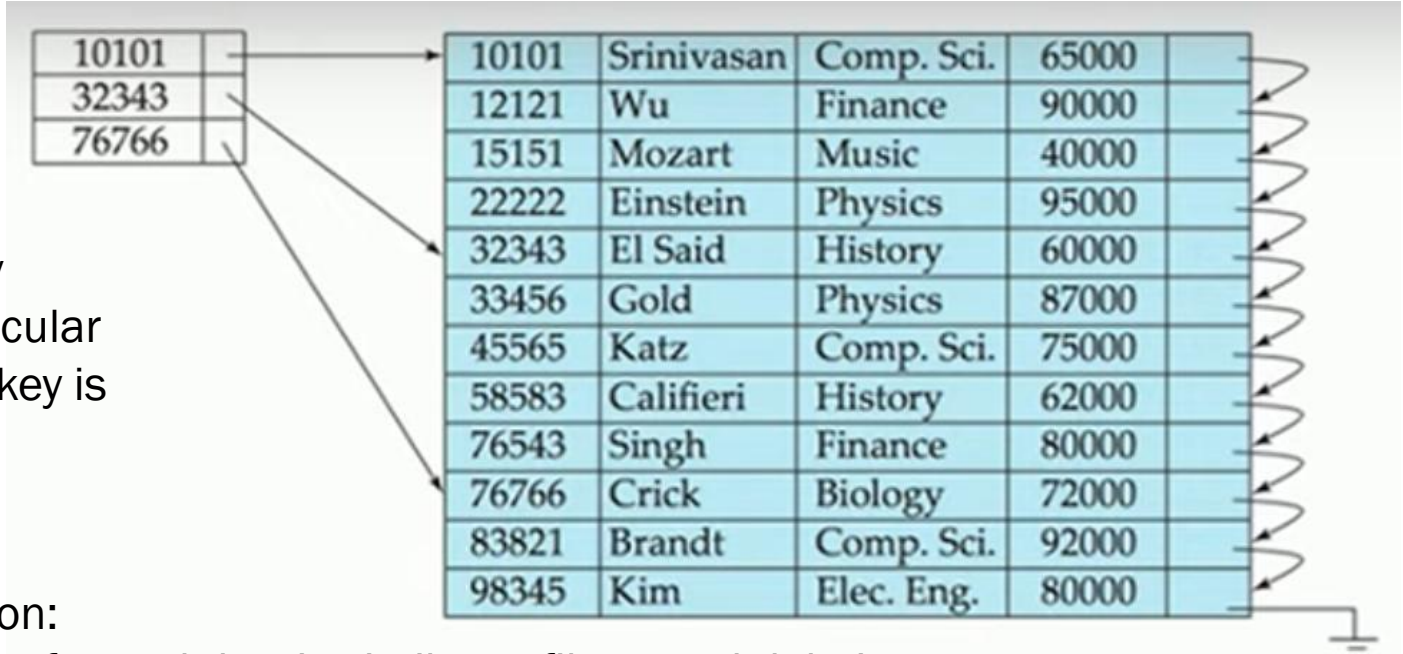
Multilevel Index

- 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

Multilevel Index (Cont.)



Index Update: Deletion



- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index entry deletion:
  - Dense indices – deletion of search-key is similar to file record deletion
  - Sparse indices –
    - If an entry for search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order)
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced

Index Update: Insertion

- **Single-level index insertion**:
  - Perform a lookup using the search-key value appearing in the record to be inserted
  - **Dense indices** - if the search-key value does not appear in the index, insert it
  - **Sparse indices** - if index stores an entry for each block of the file, no change needs to be made to be made to the index unless a new block is created
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index
- **Multilevel insertion and deletion**: algorithms are simple extensions of the single-level algorithms

Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition
  - Example 1: In the instructor relation stored sequentially by ID, we may want to find all instructors in a particular department
  - Example 2: as above, but where we want to find all instructors with a specified salary or with salary in a specified range of values
- We can have a secondary index with an index record for each search-key value

Thank You!