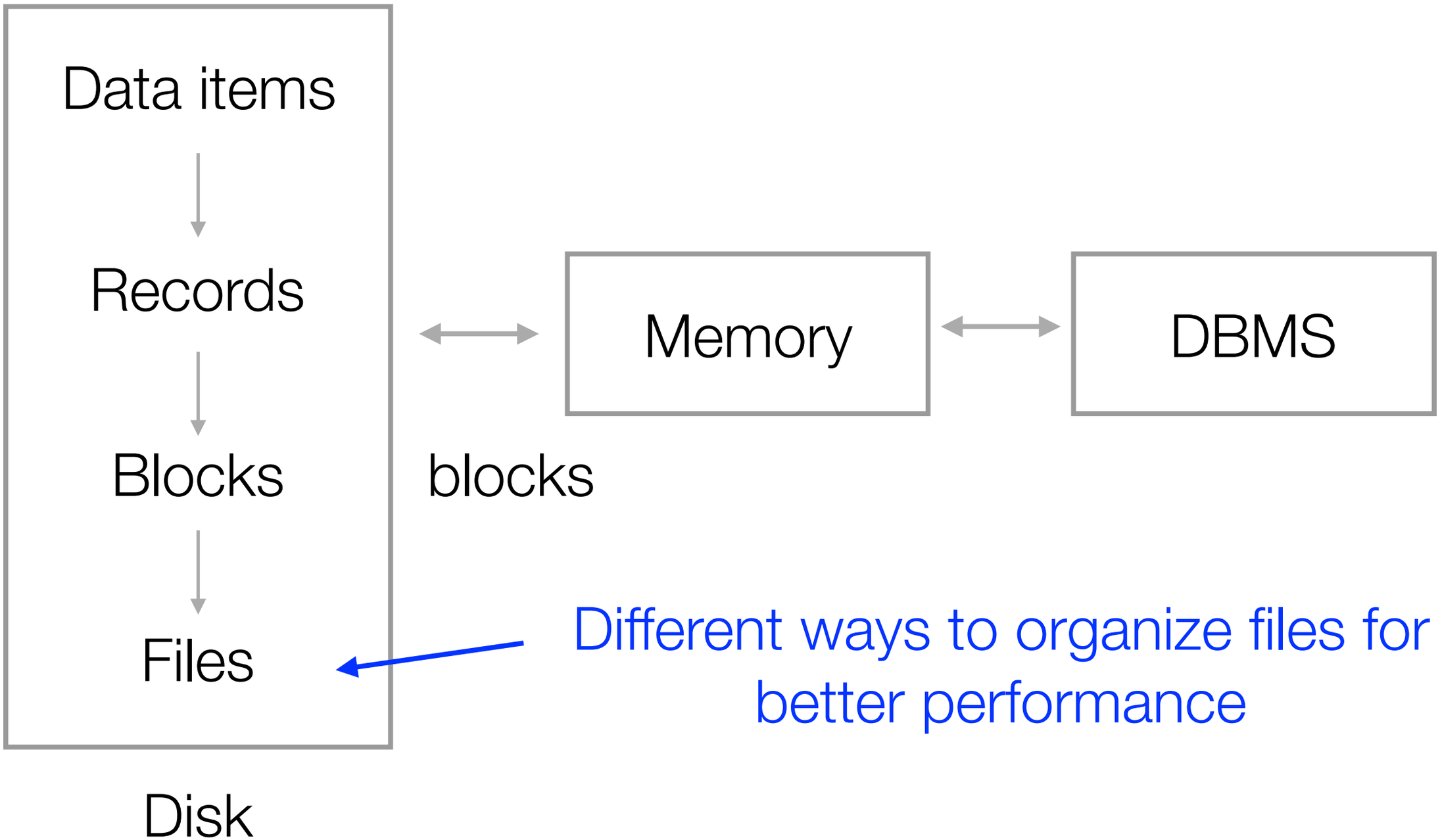
Indexing: Overview & Hashing

CS 377: Database Systems

Recap: Data Storage



Motivation for Index

- Suppose we want to search for employees of a specific age in our company database with a relation:
  SELECT * from Employee where age = 25;

  - Simple scan: O(N) — inefficient to read all tuples to find one

- Idea: Sort the records by age and we know how to do this fast (several efficient algorithms such as merge sort, heapsort, etc.)

  - Binary search: $O(\log_2 N)$

Motivation for Index (2)

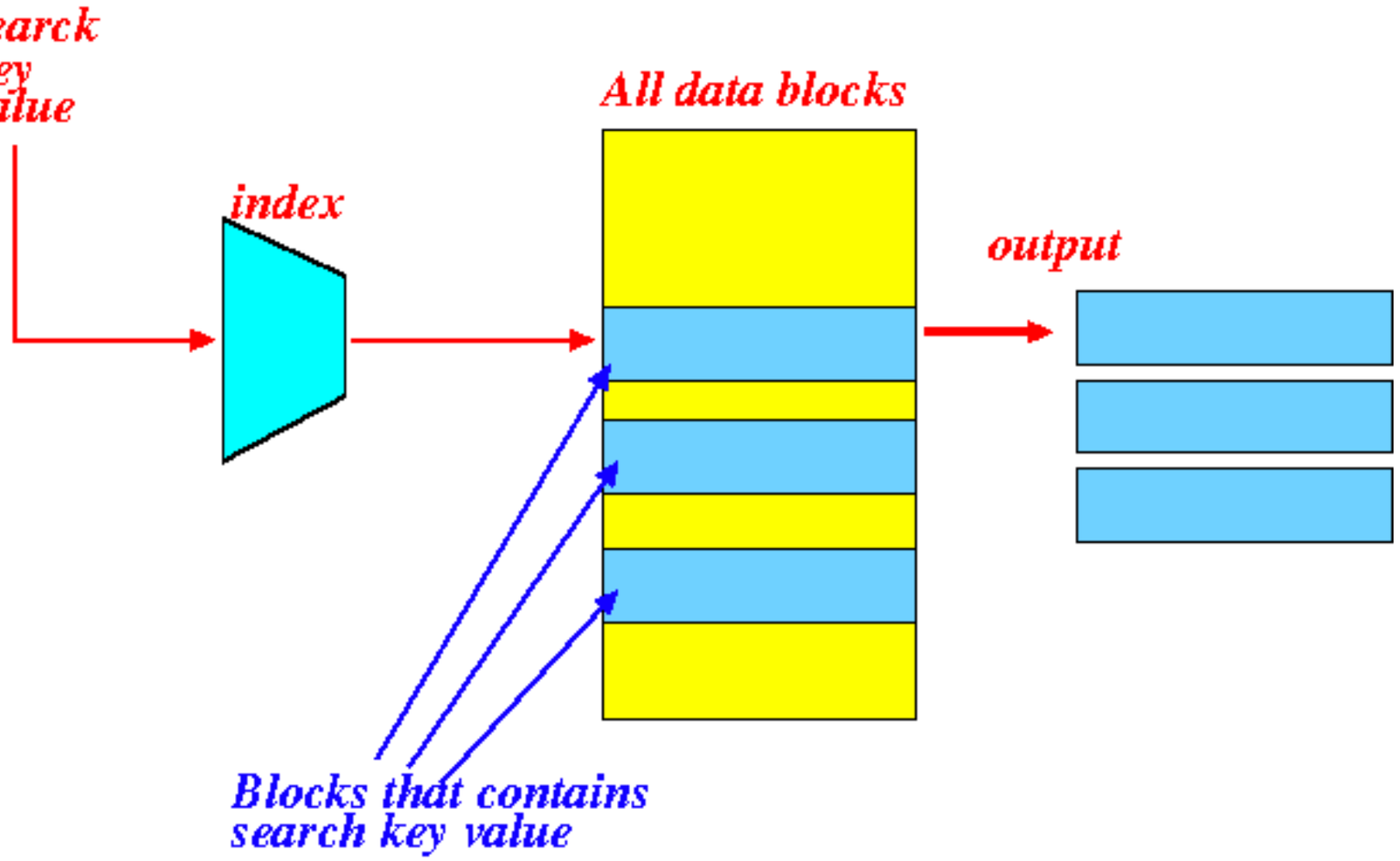
- What if we want to be able to search quickly over multiple attributes (e.g., not just age)?
  - Idea: Keep multiple copies of the records, each sorted by one attribute set — very expensive from a storage perspective
- Are there better techniques that allow better tradeoffs between storage and query speed?

Indexes

- Data structures that organize records via trees or hashing
  - Speed up search for a subset of records based on values in a certain field (search key)
    - Any subset of the fields of the relation can be the search field
    - Search key need not be the same as the key!
- Contains a collection of data entries (each entry with sufficient information to locate the records)

Index Effect

Index maps the search key value to the list of blocks that contains the search key value



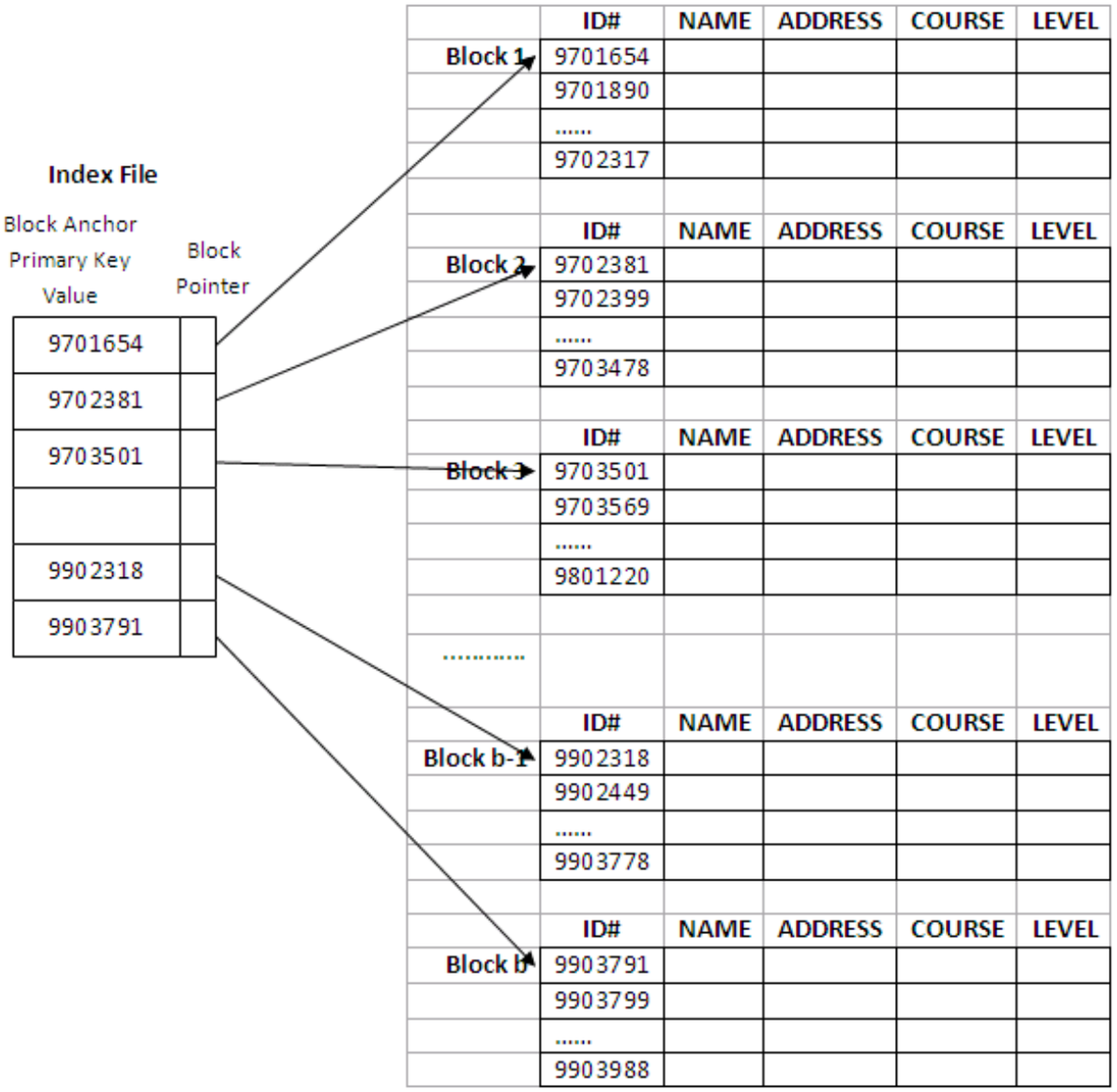
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/intro.html

Index File

- Stores records in the following format:
  Search Key | Block Ptr

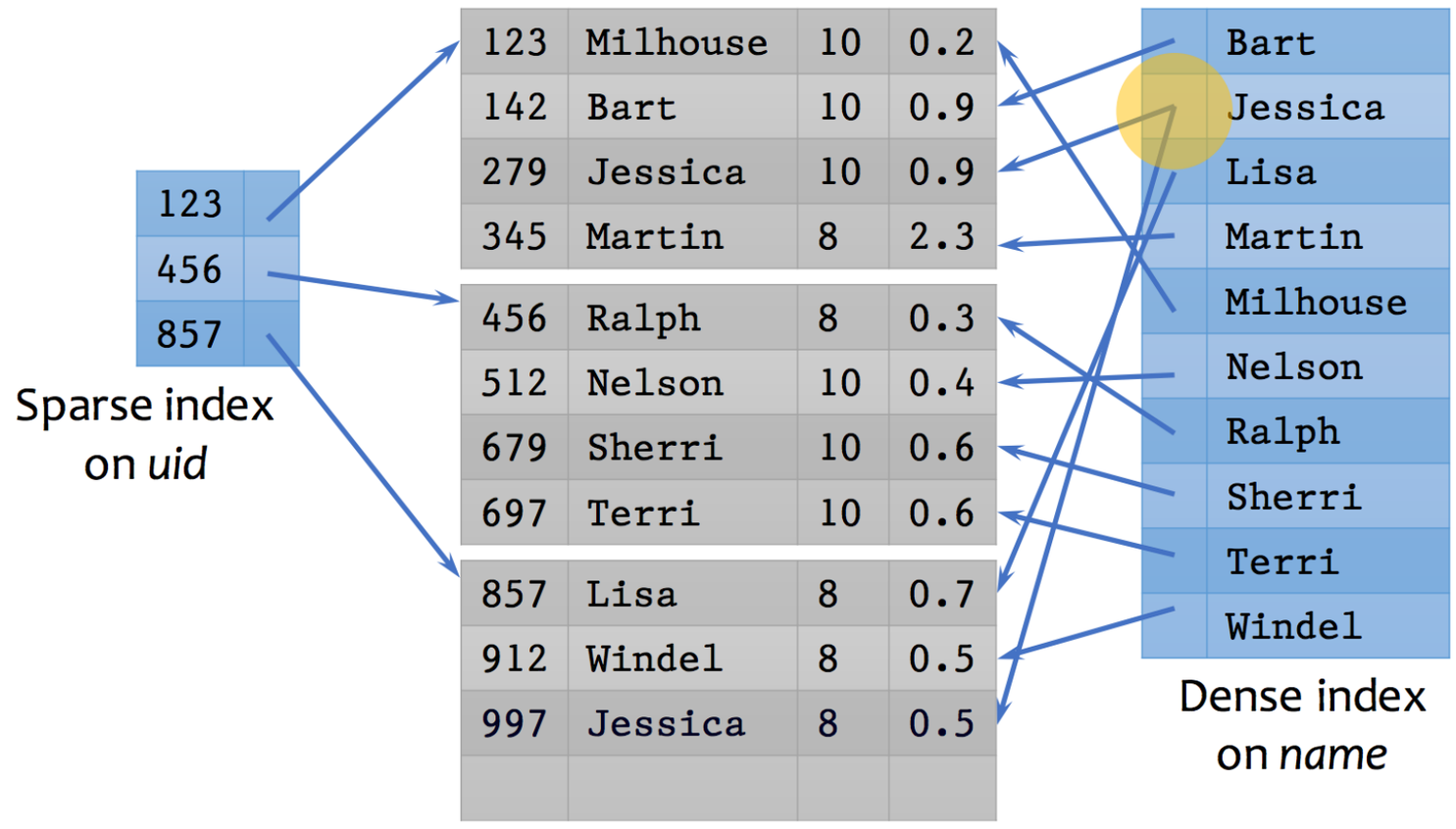
- Size of index file is much smaller than size of a data file

- Allows you to locate the block that contains the record quickly



Dense vs Sparse Index

- Dense: one index entry for each search key value
- Sparse: index entries only for some of the search values



Dense vs Sparse Index (2)

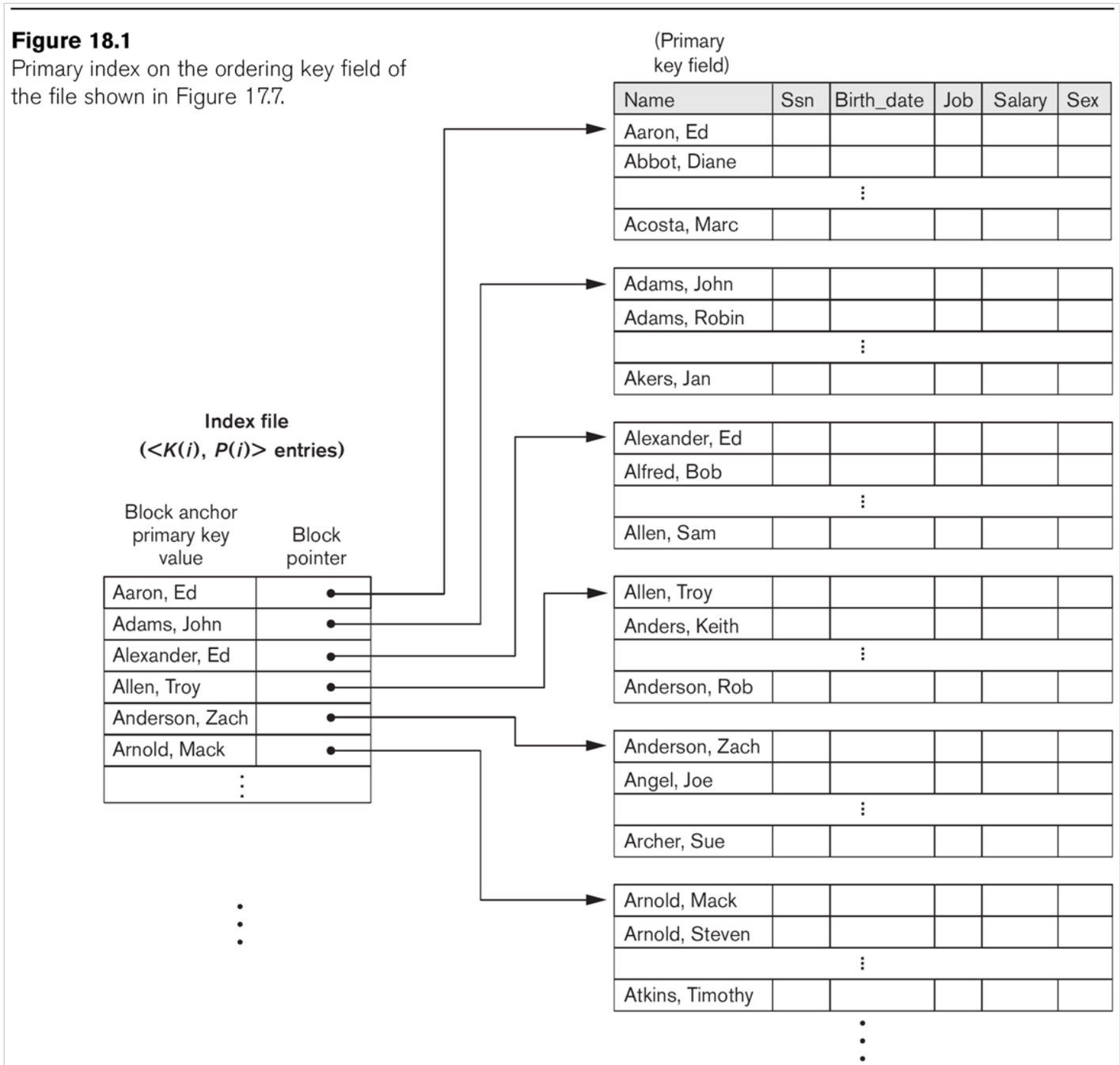
- Sparse:
  - Less index space — may fit in memory (faster)
  - Potentially more varied time to find a record within a block
  - Easier update process
  - Records must be clustered
- Dense:
  - Can directly tell if a record exists without accessing file
  - More index space

Primary Index

- Created for the primary key of a table
    - Usually ordered index whose search key is the sort key for the sequential file
- Typically sparse index
    - Binary search on the index file requires fewer block accesses than on data file

Example: Primary Index

Figure 18.1
Primary index on the ordering key field of the file shown in Figure 17.7.



Clustering Index

- Created on an ordered non-key field (not unique), known as a cluster field
- One index entry for each distinct value of the field
- Index entry points to the first data block that contains records with that field value
- Insertion and deletion are relatively straightforward

Example: Clustering Index

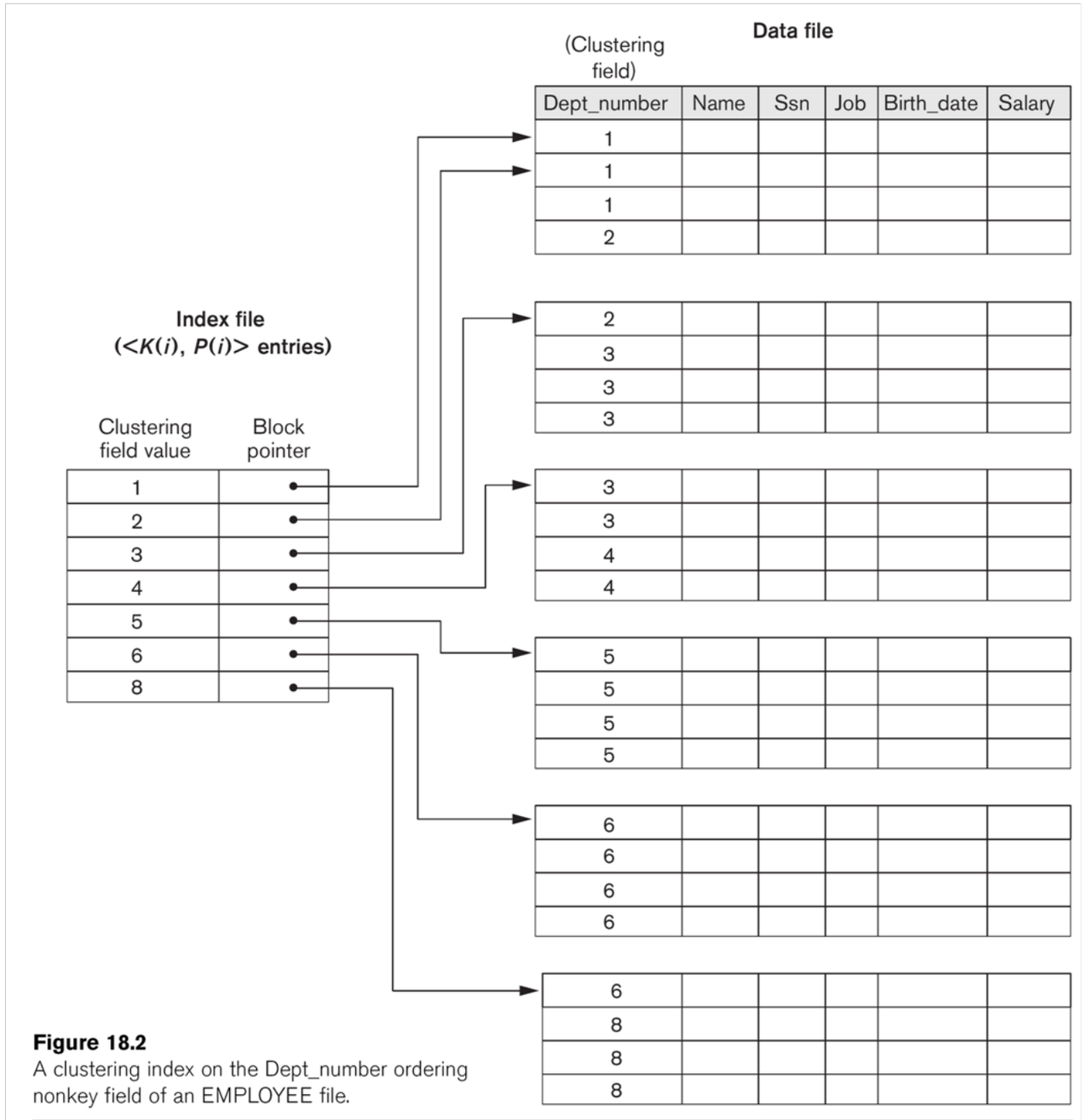
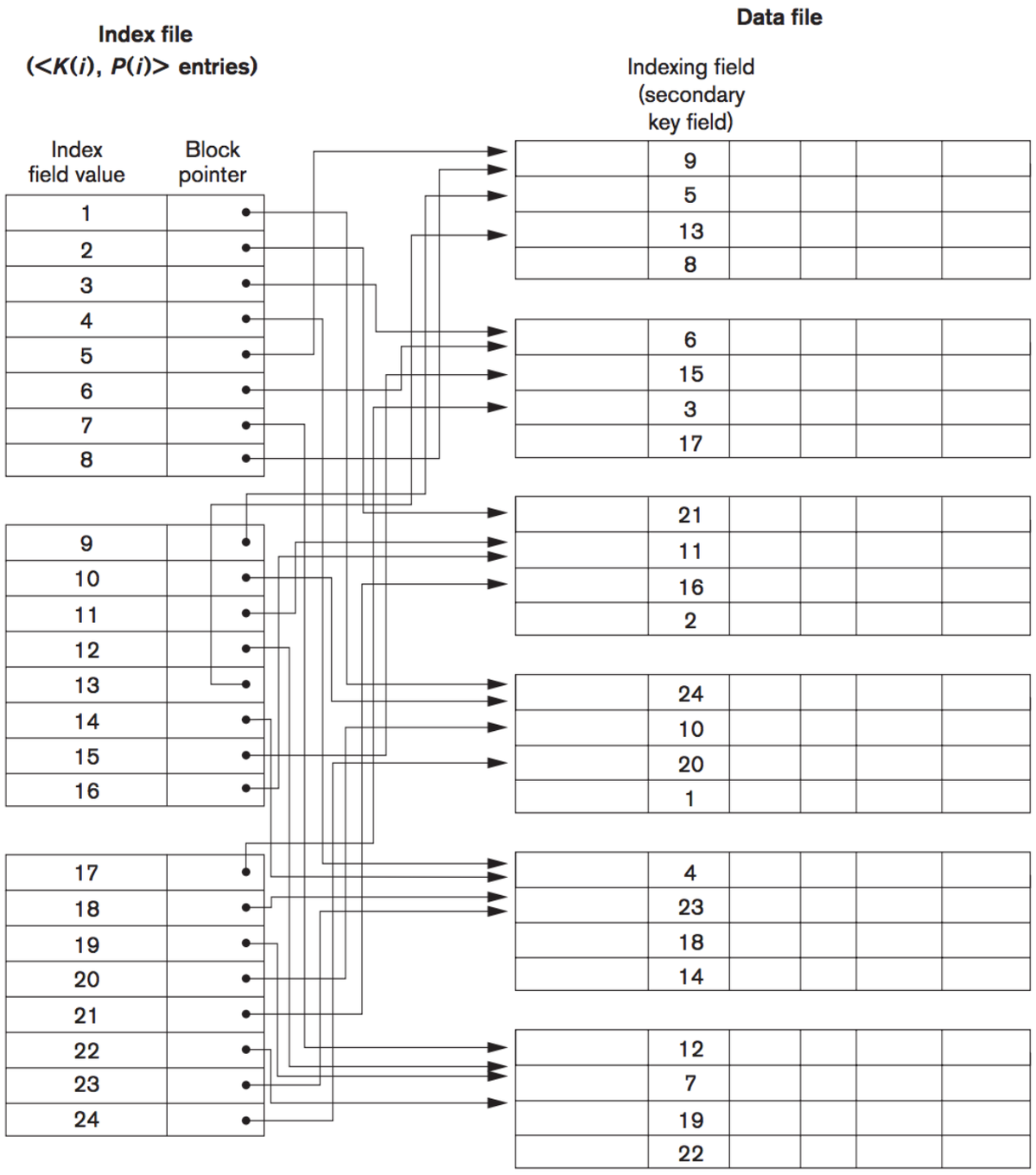


Figure 18.2
A clustering index on the Dept_number ordering nonkey field of an EMPLOYEE file.

Secondary Index

- An ordered index whose search key is not the sort key for the sequential file
    - Unique non-ordering key field results in a dense index with an entry for each record
    - Not unique fields results in an entry for each distinct value (nondense index)
- Multiple secondary indexes for the same file
- Requires more space and longer search time

Example: Secondary Index (Dense)

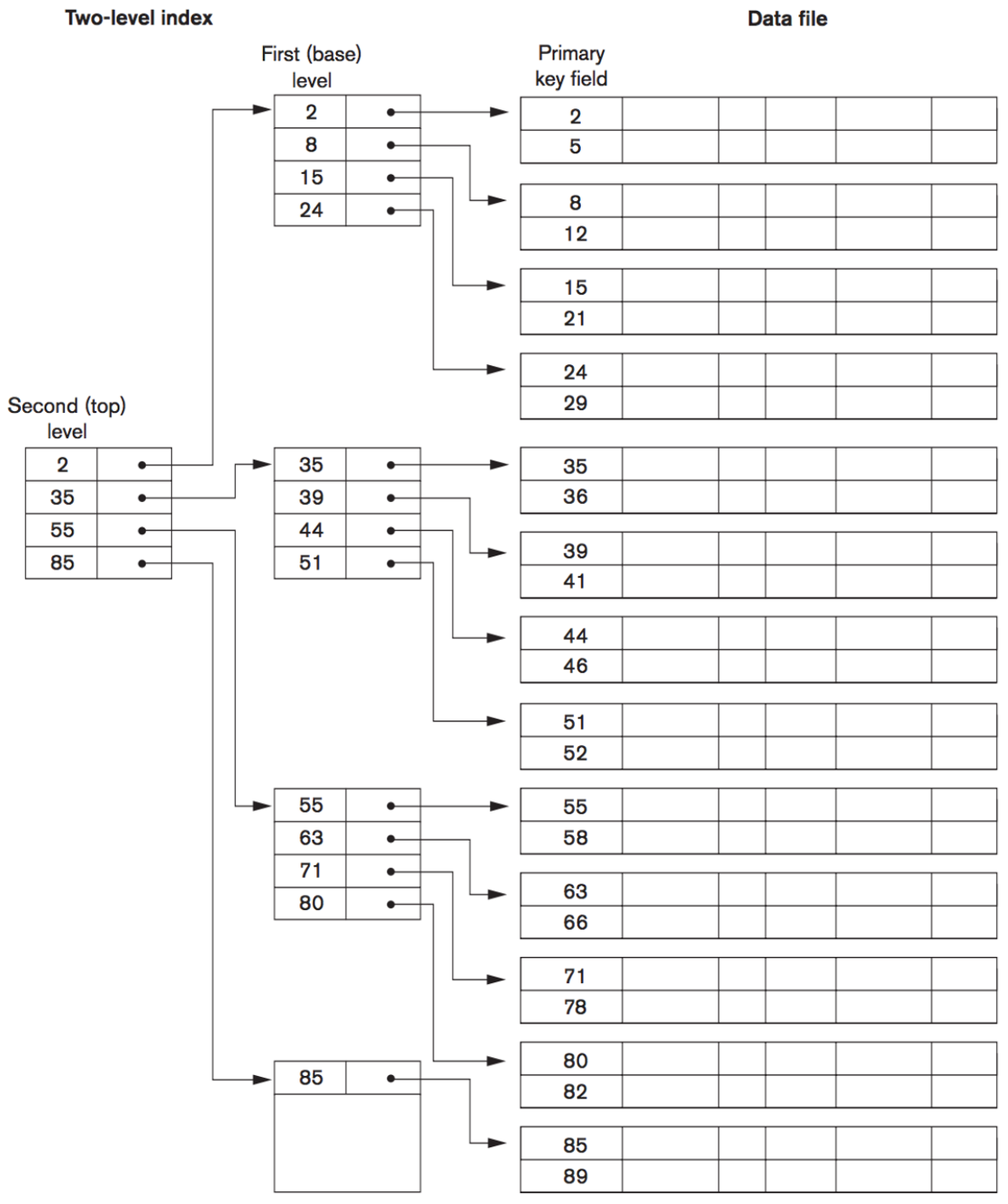


Summary of Types of Indexes

	Search key used for ordering of file	Search key not used for ordering of file
Search key is key of relation	Primary index — sparse	Secondary index (key) — dense
Search key is not key of relation	Clustering index — sparse	Secondary index (nonkey) — dense or sparse

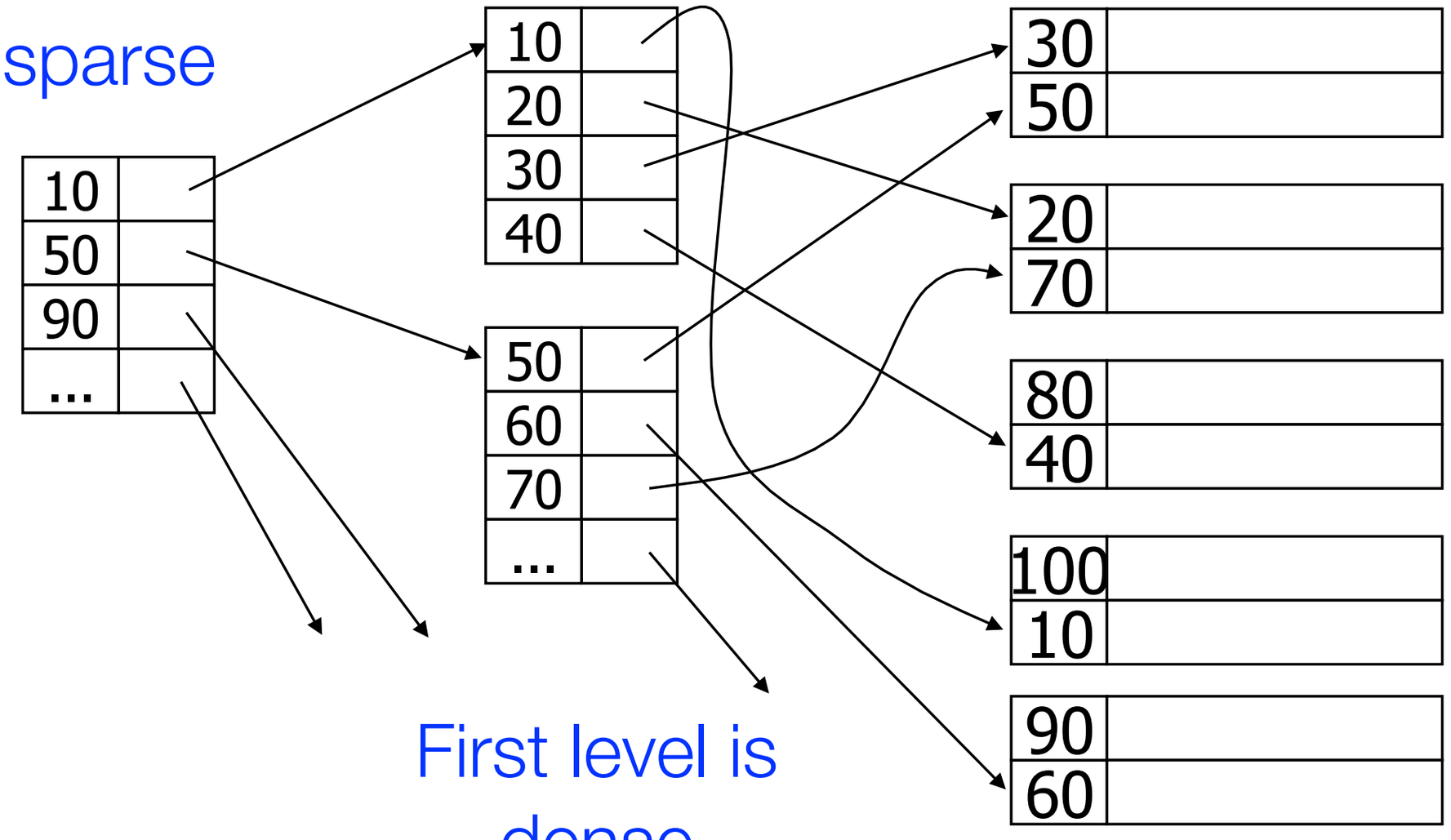
Multi-level Index

- What if index can't fit in memory?
- What if we want faster search than $\log_2(n)$?
- Solution: An index file is also a data file — create an index on the index file



Example: Multi-level Index

Second level is sparse



First level is dense

unordered file (according to search key)

SQL Index

- PRIMARY KEY declaration automatically creates a primary index
- UNIQUE key automatically creates a secondary index
- Additional secondary indexes can be created on non-key attributes
  CREATE INDEX <indexName> ON <Relation>(<attr>);
- Example: Company Database
  CREATE INDEX employeeAgeIdx ON Employee(Age);

Index Structures

- Hash index
  - Good for equality search
  - In expectation: O(1) I/Os and CPU performance for search and insert
- B+ tree index
  - Good for range and equality search
  - $O(\log_F(N))$ I/O cost for search, insert, and delete

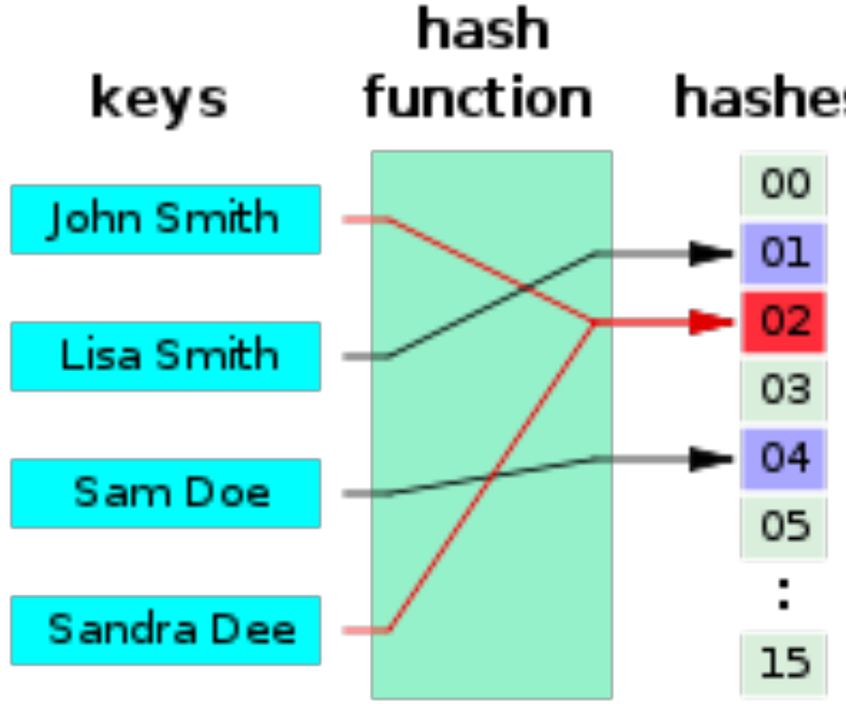
Hash Function

- A hash function, h, is a function which transforms a search key from a set K, into an index in a table of size n

  h: K —> {0, 1, …, n-2, n-1}

- Bucket is a location (slot) in the bucket array (or the hash table)

- Different search keys can be hashed into the same bucket



Hash Function Properties

- Minimize collisions (different hash keys should hash to different values whenever possible)

- Uniform — each bucket is assigned the same number of search key values from the set of all possible values

- Random — each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file

- Be easy and quick to compute

Hash Function Usage

- The mark of a computer scientist is their belief in hashing
  - Possible to insert, delete, and lookup items in a large set in O(1) time per operation
- Widely used in a variety of applications
  - Crytpography, table or database lookup, caches for large datasets, finding duplicate records, finding similar “items”, etc.

Hash Index

- Hash function, h, distributes all search-key values to a collection of buckets

- Each bucket contains a primary page plus overflow pages

- Buckets contain data entries

- Entire bucket has to be searched sequentially to locate a record (since different search-key values may be mapped to same bucket)

Example: Hash Index

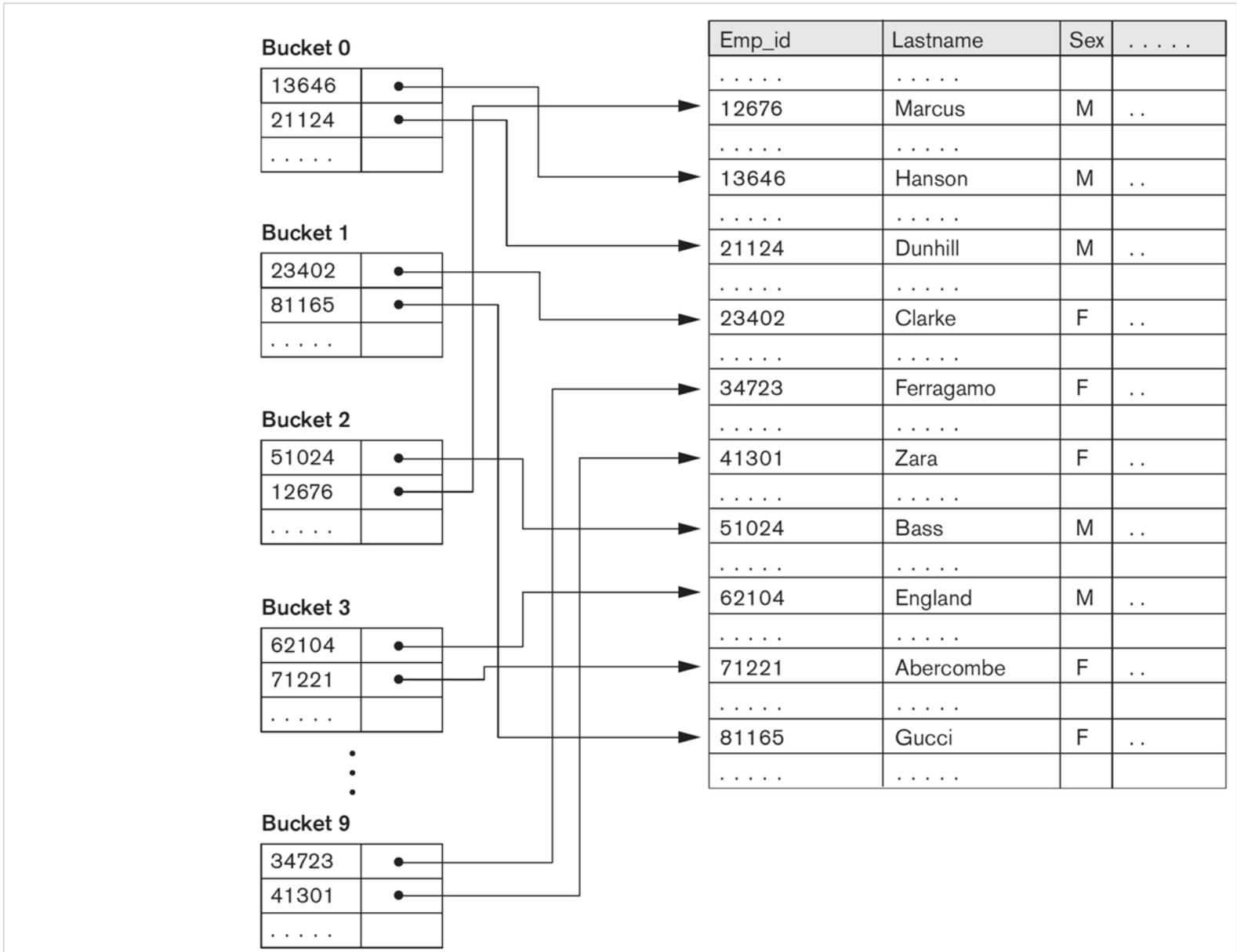


Figure 18.15
Hash-based indexing.

Bucket Overflows

- Causes of bucket overflows
  - Insufficient buckets
  - Skew in distribution of records
    - Multiple records with the same search-key value
    - Hash function produces non-uniform distribution
- Overflow chaining links the buckets together to handle when a certain bucket has a large number of entries

Example: Overflow Hash

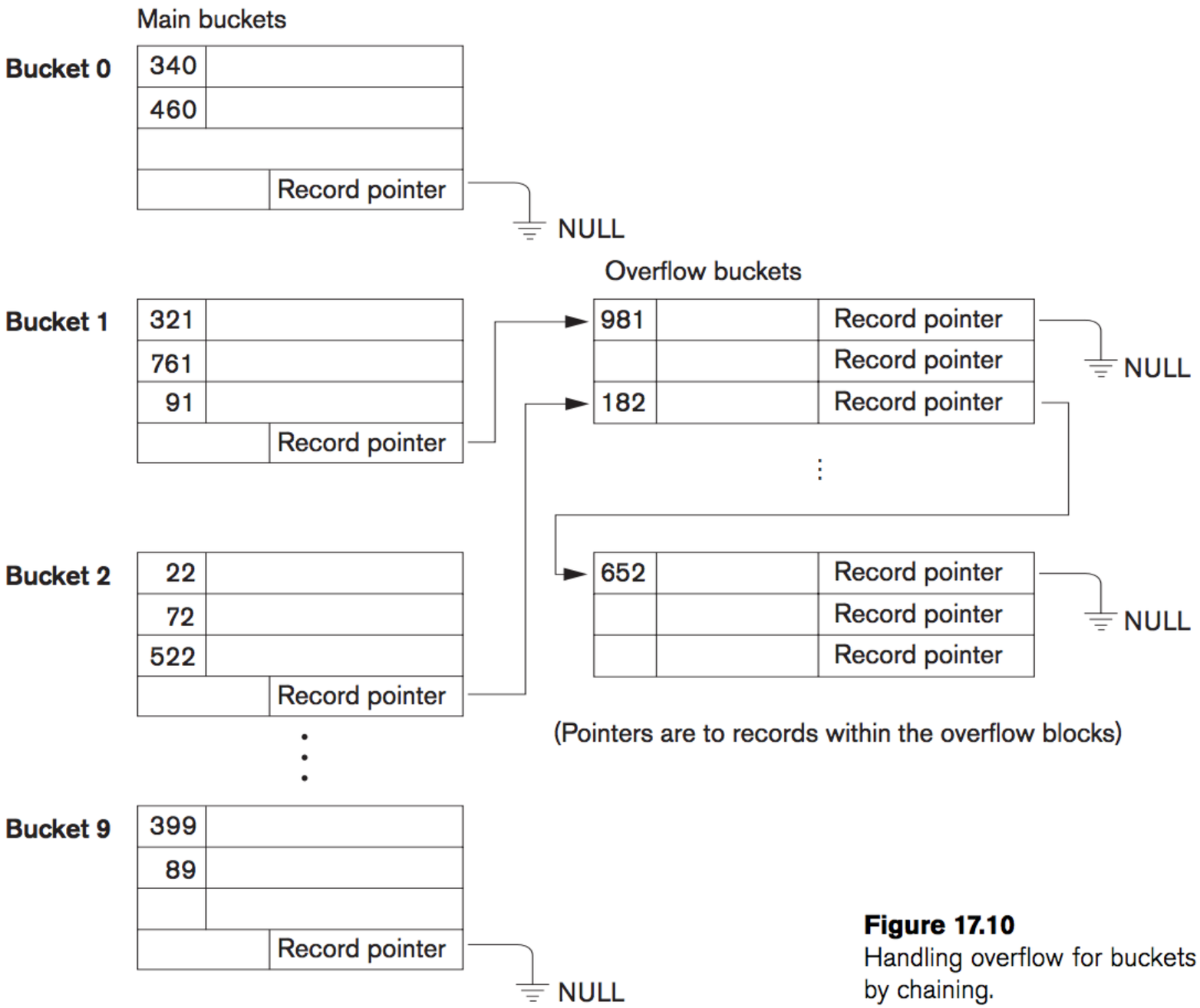
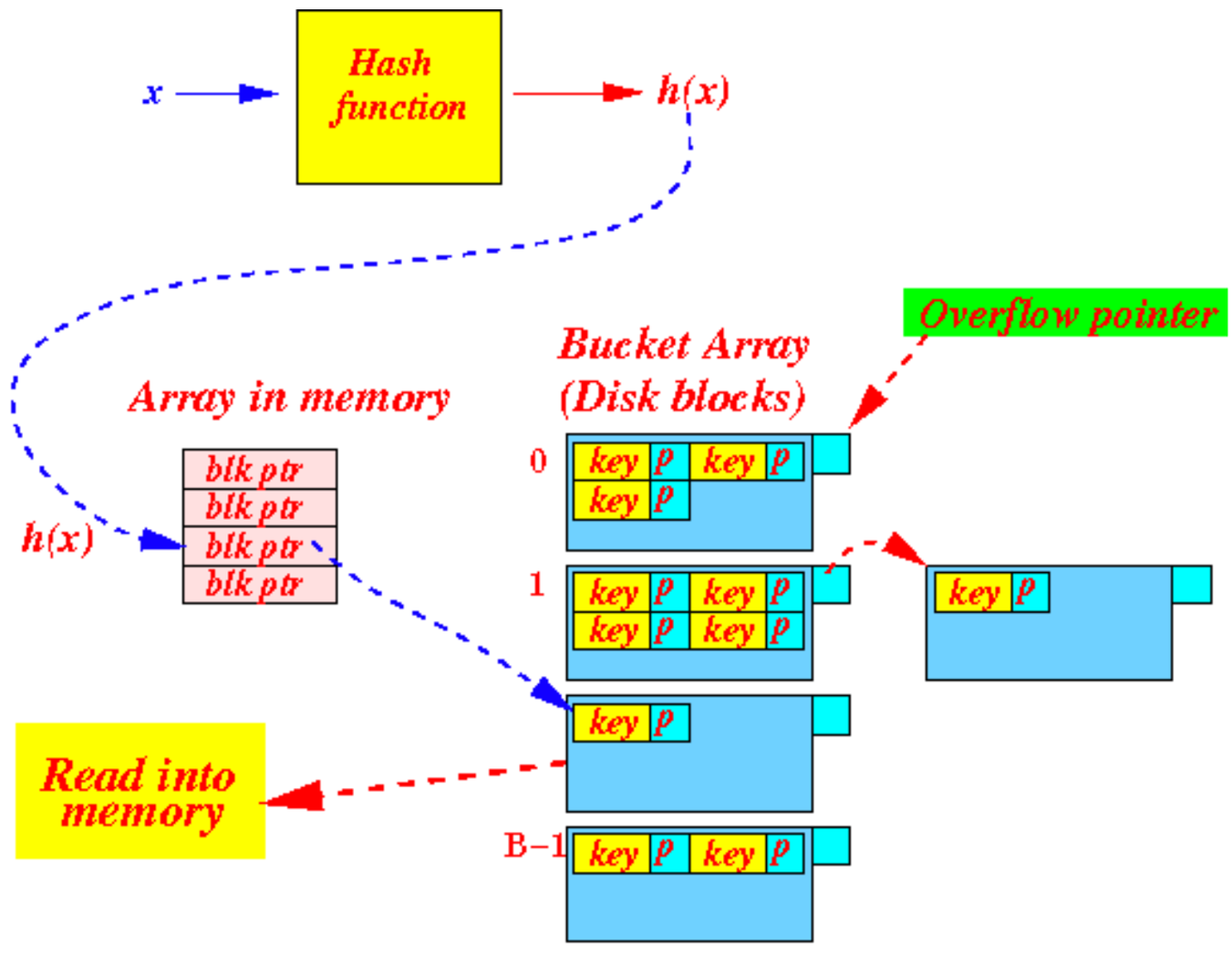


Figure 17.10
Handling overflow for buckets by chaining.

Hash Index Query

- Compute hash value h(x)
- Read the disk block pointed to by the block pointer h(x) into memory
- Linear search the bucket for x, ptr(x)
- Use ptr(x) to access x on disk



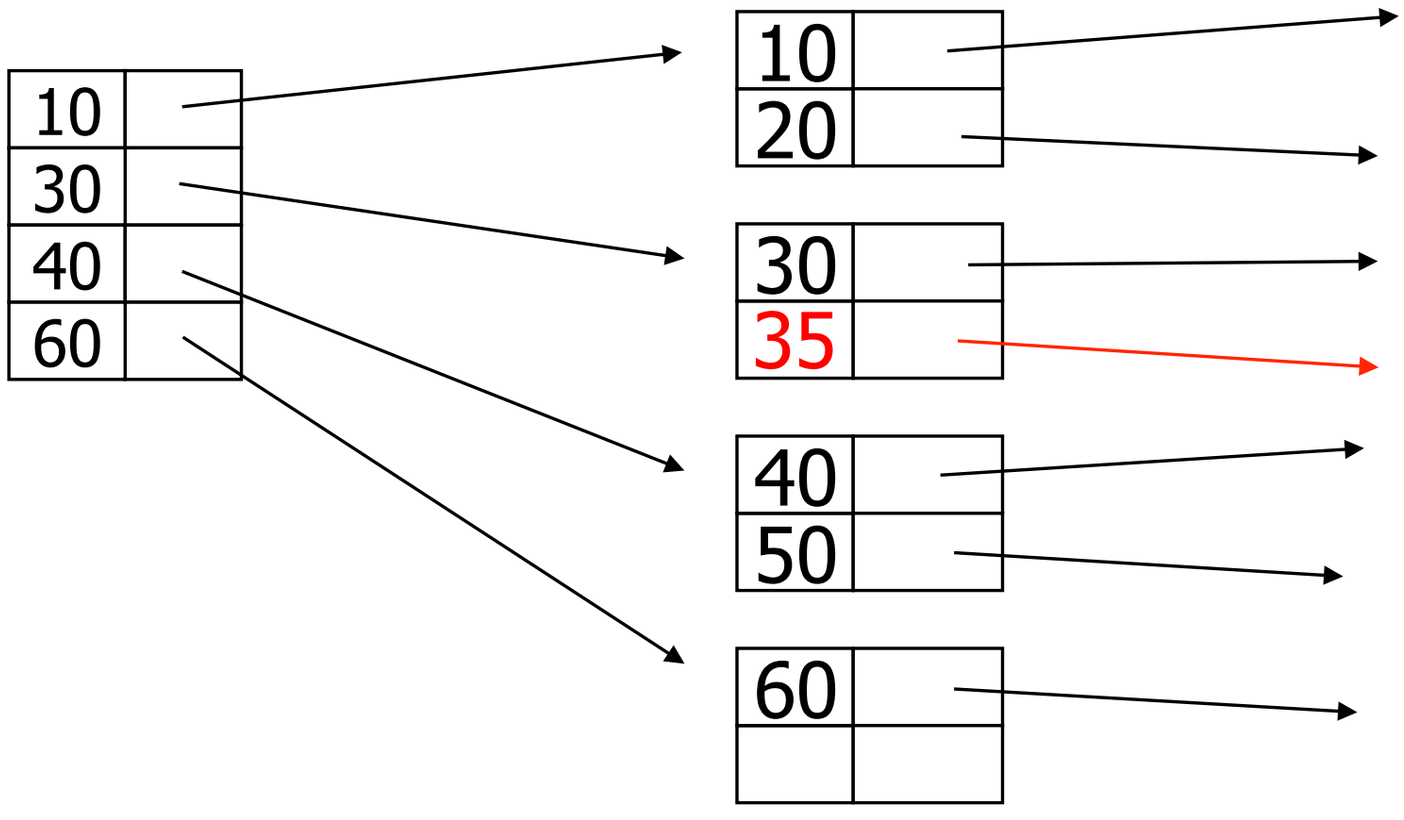
http://www.mathcs.emory.edu/~cheung/Courses/554/Syllabus/3-index/hashing.html

Hash Index Insert/Deletion

- Hash the new item h(x)
- Find the hash bucket for the item
- Add/delete item from hash bucket
- If there is insufficient space, allocate an overflow bucket and then add it to the overflow bucket

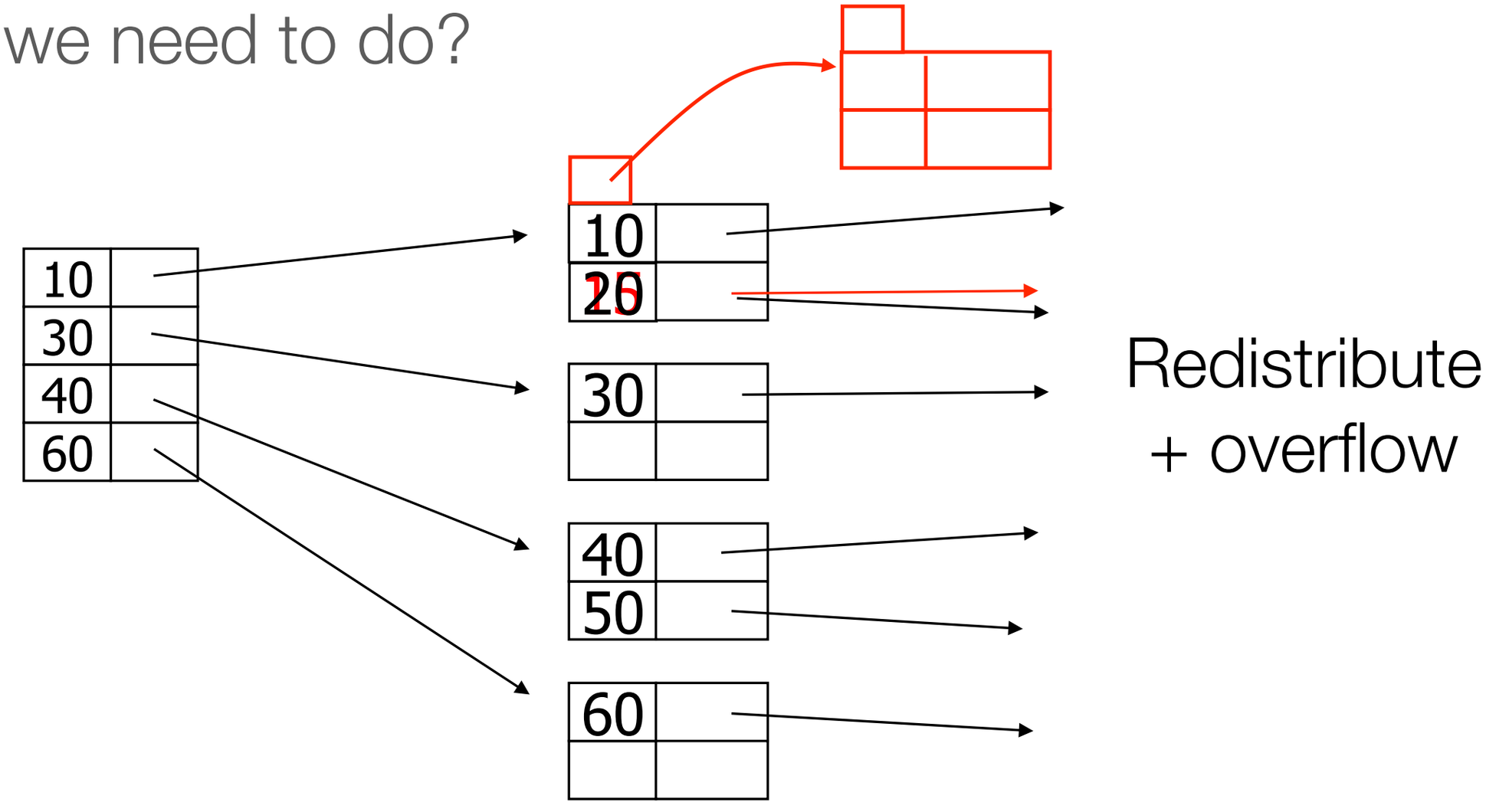
Example: Multi-level Index Insertion

- Insert a tuple with value 35 in the search key
- What do we need to do?



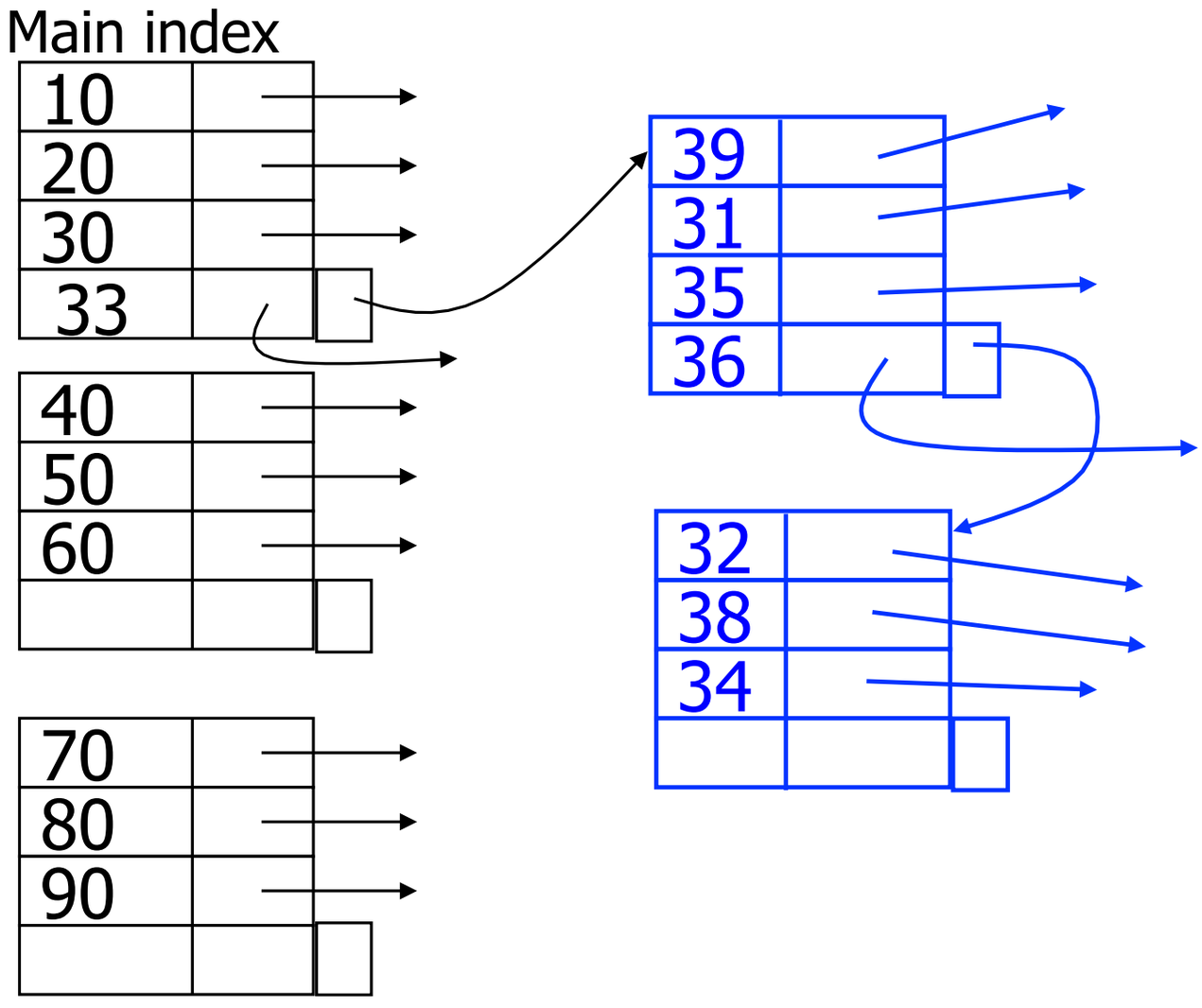
Example: Multi-level Index Insertion

- Insert a tuple with value 15 in the search key
- What do we need to do?



Potential Problems

- What happens after many insertions?



overflow pages may not be sequential!

Static Hashing Issues

Databases grow and shrink with time, while static hashing assumes a fixed set of B bucket addresses

- If initial number of buckets is too small, performance will degrade due to too much overflow
- If space allocated for anticipated growth or database shrinks, buckets will be underutilized and space will be wasted

Fixing Static Hashing

- One solution: periodic re-organization of the file with new hash functions
  - Expensive — rehash all keys into a new table!
  - Disrupts normal operations
- Another (better) solution: dynamic hashing techniques that allow size of the hash table to change with relative low cost

Extendible Hashing (Fagin, 1979)

Main idea:

- Directory of pointers to the buckets
- Double the number of buckets by splitting just the bucket that overflowed
- Directory is much smaller than file, so doubling it is cheaper

Extendible Hashing Structure

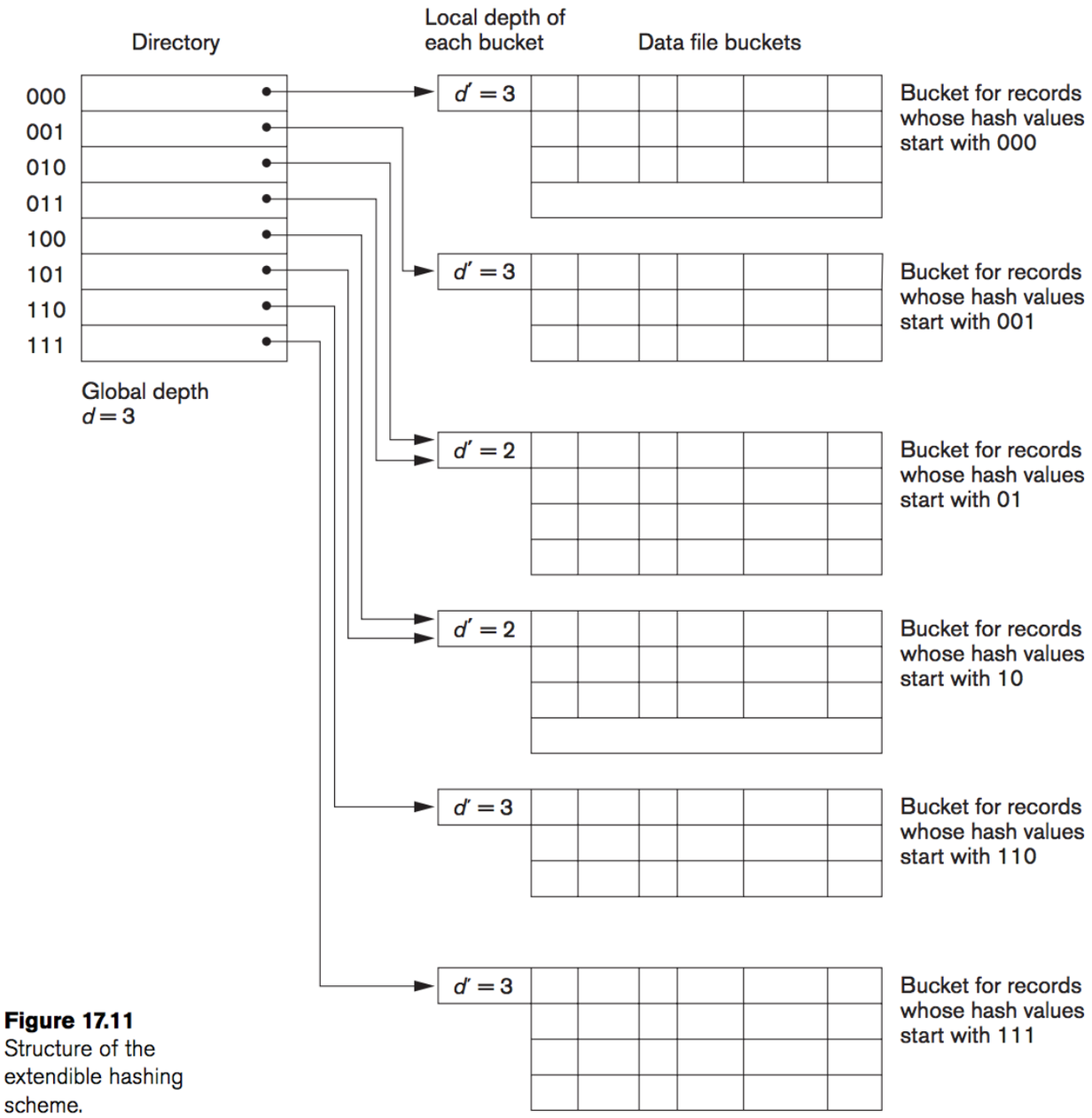
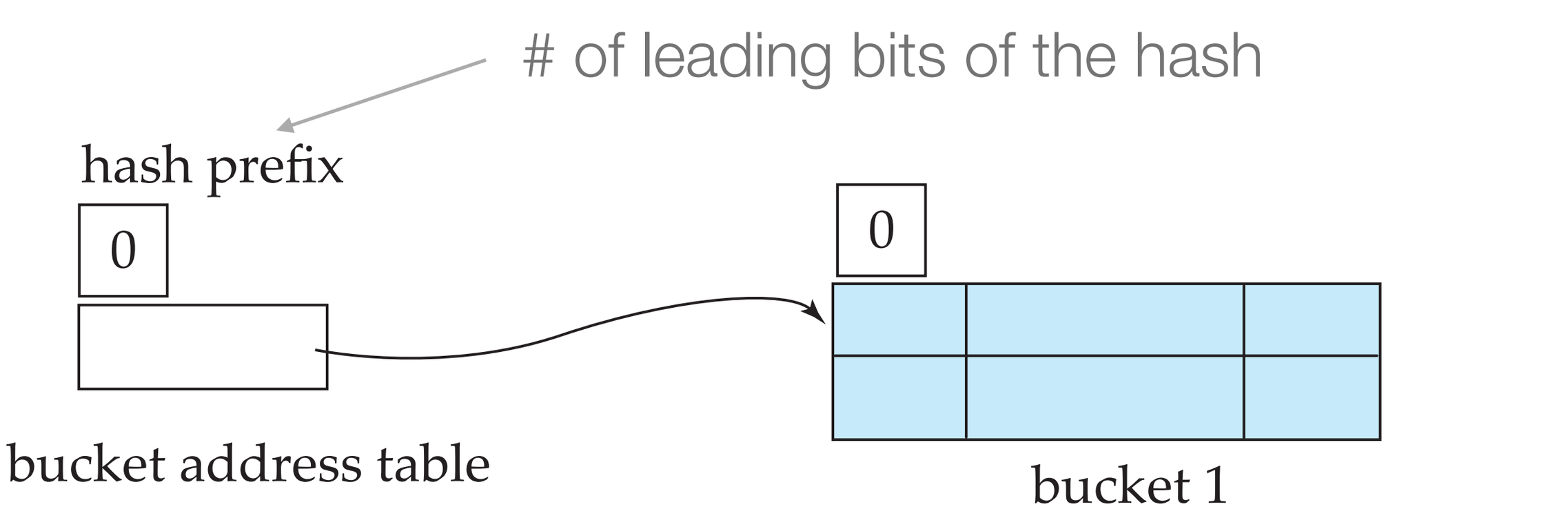


Figure 17.11
Structure of the extendible hashing scheme.

Example: Extendible Hashing Structure

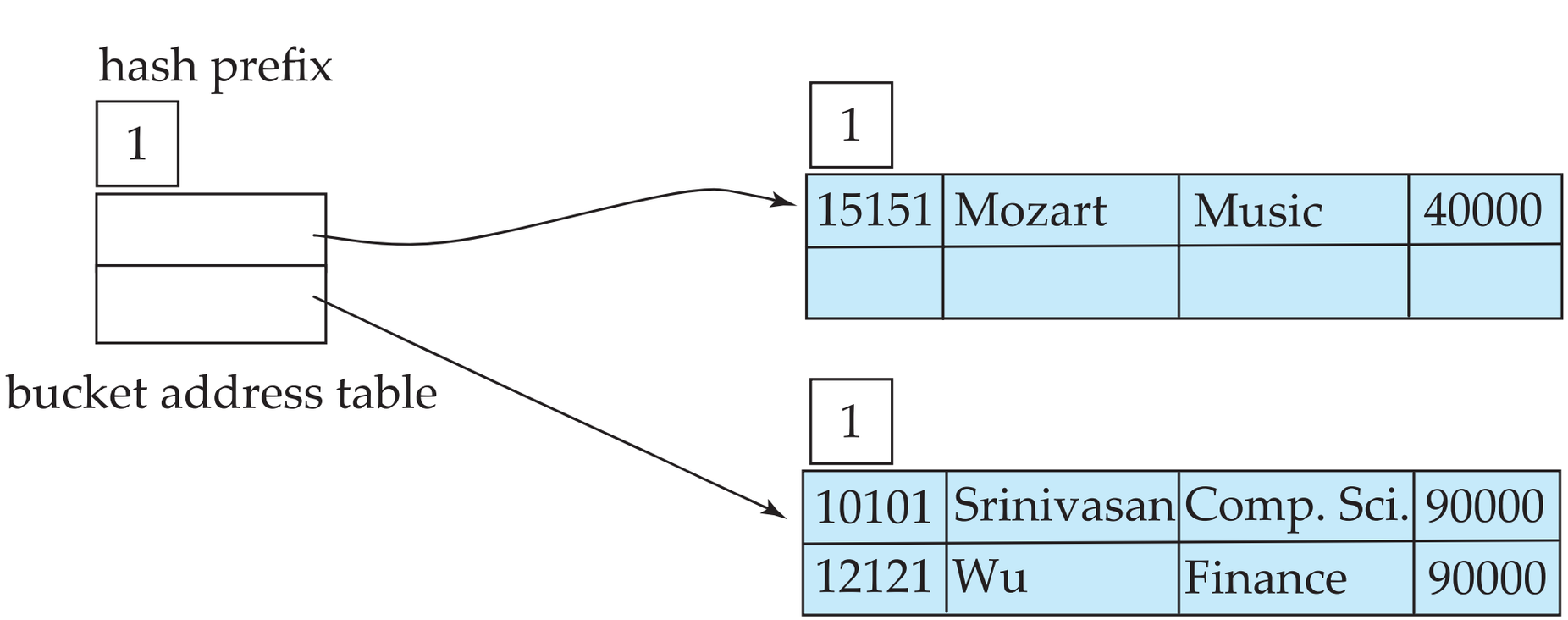
Instructor(ID, Lname, Department, Salary)
and we want to build a secondary index on the department attribute



http://codex.cs.yale.edu/avi/db-book/db6/slide-dir/PPT-dir/ch11.ppt

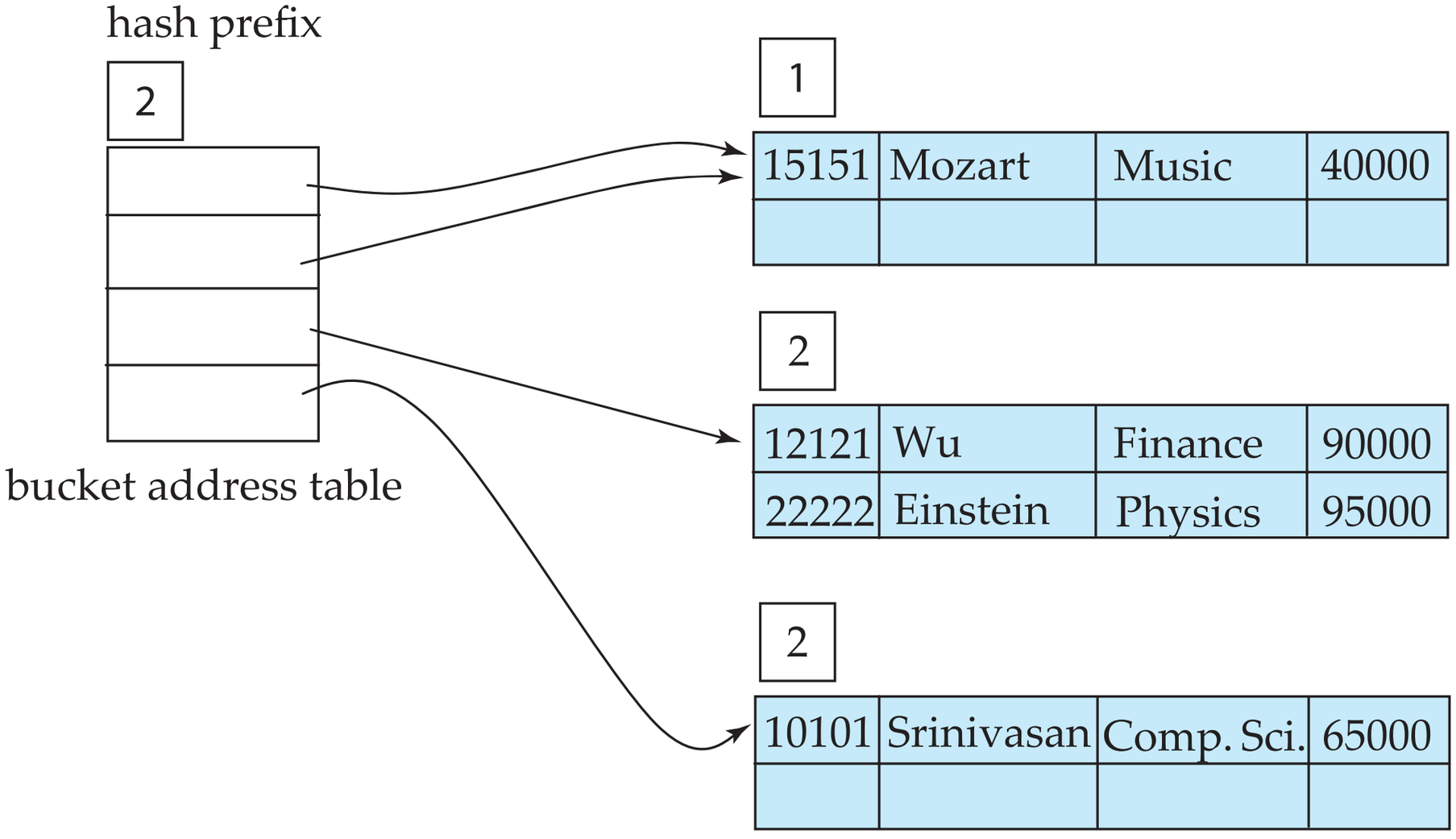
Example: Extendible Hashing Structure (2)

Insert 3 new tuples: (15151, Mozart, Music, 40000),
(10101, Srinivasan, Comp. Sci, 90000),
(12121, Wu, Finance, 90000)

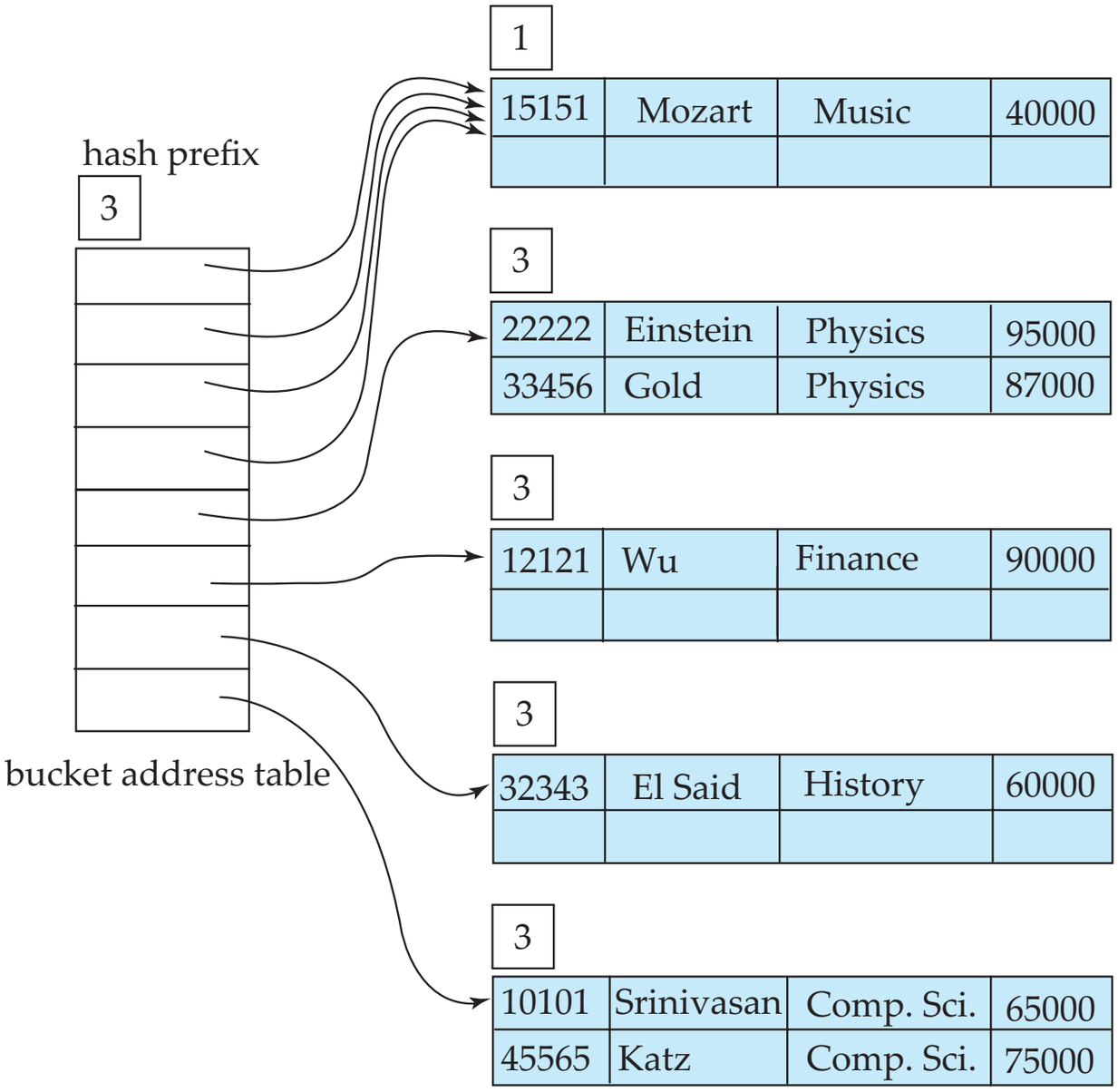


Example: Extendible Hashing Structure (3)

Insert new tuple Einstein whose first 1 bit hash matches the 2nd bucket and overflows => increase the hash prefix and have the new bucket



Example: Extendible Hashing Structure (4)



Other Dynamic Hashing Schemes

- Dynamic hashing (Larson, 1978): precursor to extendible hashing with the main difference in the organization of the directory with tree-structured directory with internal nodes and leaf nodes

- Linear hashing (Litwin, 1980): allows incremental growth without needing a directory at the cost of more bucket overflows

Dynamic Hashing Properties

- Benefits
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages
  - Additional level of indirection on lookup (2 block access instead of one)

Ordered Files vs Hashing

- Relative frequency of insertions and deletions
  - Ordered files are much more expensive to keep sorted
  - Cost of periodic re-organization in hashing
- Is average access time more important than worst-case access time?
- What types of queries are expected?
  - Hashing is good for equality
  - Ordered files are preferred if range queries are common

Indexing: Recap

- Motivation for Index
- Types of indexes
  - Dense vs sparse
  - Primary vs secondary vs clustering
- Hash index file
  - Static vs dynamic