Views

CENG315 INFORMATION MANAGEMENT

The Three Schema Levels

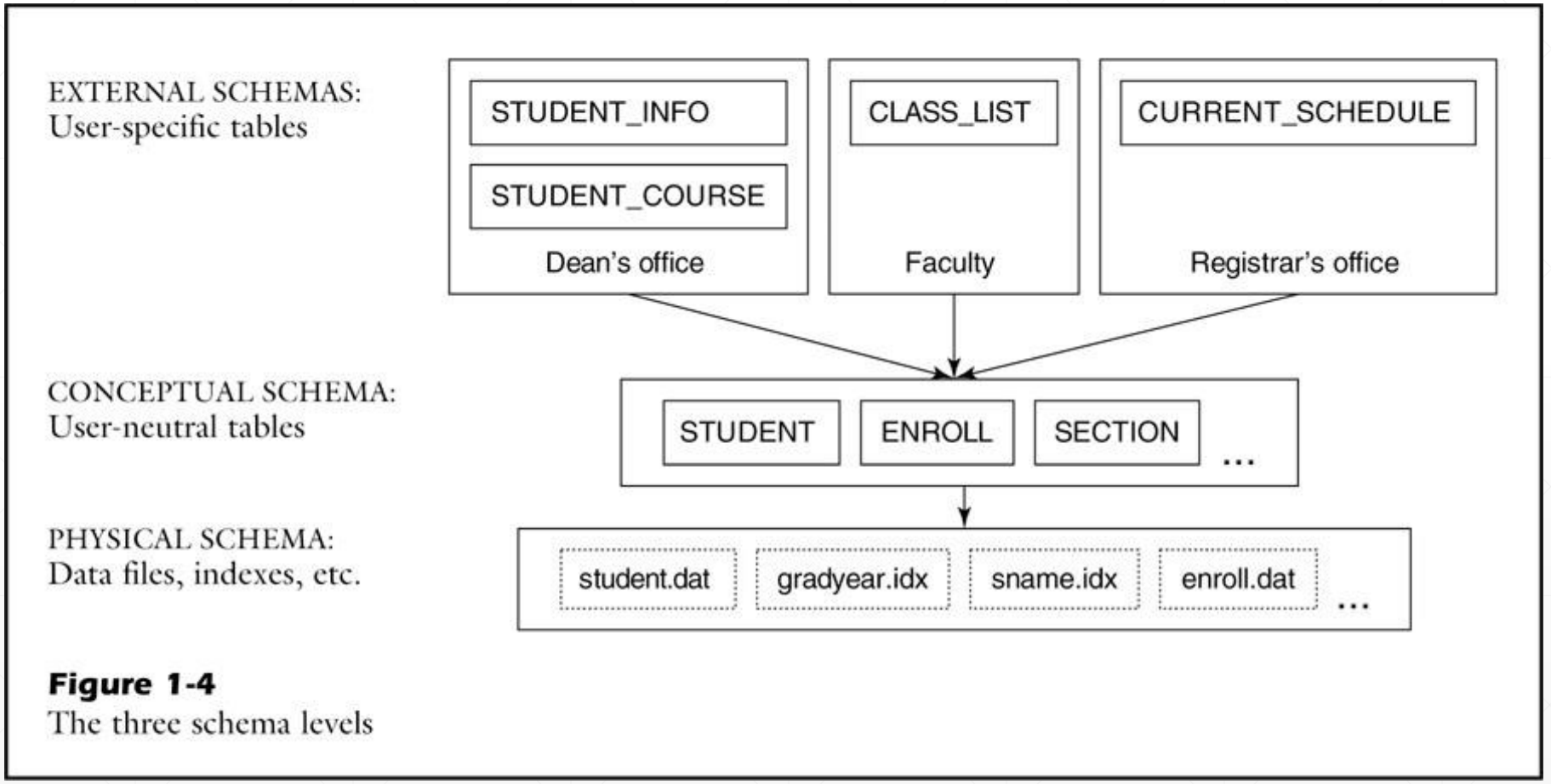


Figure 1-4
The three schema levels

Views

Materialized Views

Indexes

(ordered, hash)

Views

- A view is essentially a named query.
- A user can refer to a view as if it were a table.
  - Requesting that its records be displayed
  - Referencing it in another query or view definition

Why Use Views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access

Views in SQL

- A view is a virtual relation based on the result-set of a SELECT statement.
- Views are customized presentations of data in one or more tables or other views.
- You can think of them as stored queries.
- Views do not actually contain data, but instead derive their data from the tables upon which they are based.

Views in SQL

- Syntax:

```
CREATE VIEW view_name AS
SELECT column_name(s)
FROM table_name(s)
WHERE condition
```

Create a View

- Example: Create a view with names and salaries of employees whose job title is clerk.

```
CREATE VIEW Clerk AS
SELECT ename, sal
FROM emp
WHERE job = 'CLERK';
```

Querying View

- A view could be used from inside a query or from inside another view.

```
SELECT ename
FROM Clerk
WHERE sal > 1000;
```

View

- Have same result as

```
SELECT ename
FROM emp
WHERE job = 'CLERK' AND sal > 1000;
```

Table

Querying View

```
CREATE VIEW EmpDepartments AS
SELECT emp.ename, dept.deptno, dept.dname
FROM emp, dept
WHERE emp.deptno = dept.deptno;
```

Same results:

```
SELECT ename
FROM EmpDepartments
WHERE dname = 'RESEARCH';
```

```
SELECT ename
FROM emp, dept
WHERE emp.deptno = dept.deptno
AND dname = 'RESEARCH';
```

Renaming Attributes in View

- Sometimes, we might want to distinguish attributes by giving different names.

```
CREATE VIEW Clerk (clerkName, clerkSalary) AS
SELECT ename, sal
FROM emp
WHERE job = 'CLERK';
```

Updatable Views

- A view is updatable if every record in the view has a unique corresponding record in some underlying database table.
- Single-table views that do not involve aggregation are the most natural updatable views, because there is an obvious connection between the view records and the stored records.
- Multi-table views are in general not updatable. The problem is that the virtual view record is composed from (at least) two stored records.

Updatable Views (Cont.)

```
create view StudentMajor as
   select s.SName, d.DName
   from STUDENT s, DEPT d
   where s.MajorId=d.DId
```

Is this view updatable?

- Deleting the record [“sue”, “math”] from this view means that Sue should no longer be a math major.
- Database system could
  - delete Sue’s record from STUDENT
  - delete the math record from DEPT
  - modify Sue’s MajorId value to give her a new major
- Since the database system has no reason to prefer one way over the others, it must disallow the deletion.

Modifying Views in SQL

- For a view to be updatable, there must be a one-to-one relationship between the rows in the view and the rows in the underlying table.
- There are also certain other constructs that make a view non-updatable.
  - Aggregate functions
  - DISTINCT
  - GROUP BY
  - HAVING
  - ….

Modifying Views in SQL (INSERT)

```
CREATE VIEW Clerk AS
        SELECT ename, sal
        FROM emp
        WHERE job = 'CLERK';

INSERT INTO Clerk
VALUES ('SUE', 8000);
```

- We need to add attribute **empno** to it's **SELECT** clause because empno cannot be null in emp table (NOT NULL constraint.)

Modifying Views in SQL (INSERT) (Cont.)

```
CREATE VIEW Clerk AS
    SELECT empno, ename, job, sal
    FROM emp
    WHERE job = 'CLERK';
```

Then

```
    INSERT INTO Clerk
    VALUES (1234, 'SUE', 'CLERK', 8000);
```

View

EMPNO	ENAME	JOB	SAL
1234	SUE	CLERK	8000

Table

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
1234	SUE	CLERK	-	-	8000	-	-

Modifying Views in SQL (DELETE)

- Suppose we wish to delete all employees with “MS” in their name from the updateable view Clerk.

```
DELETE FROM Clerk
WHERE ename LIKE '%MS%’;
```

- It is turned into the base table delete

```
DELETE FROM emp
WHERE ename LIKE '%MS% AND job = 'CLERK';
```

Modifying Views in SQL (UPDATE)

- UPDATE from an updateable view

```
UPDATE Clerk
SET sal = 1000
WHERE ename = 'JAMES';
```

- It is turned into the base table update

```
UPDATE emp
SET sal = 1000
WHERE ename = 'JAMES' AND job = 'CLERK';
```

Modifying Views in SQL (DROP)

- DROP view: All views can be dropped, whether or not the view is updateable.

  DROP VIEW Clerk

- **DROP VIEW** does not affect any tuples of the underlying relation (table) emp.

- However, **DROP TABLE** will delete the table and also make the view Clerk unusable.

  DROP TABLE emp;

Read Only Views

```
CREATE VIEW Manager AS
    SELECT empno, ename, sal
    FROM emp
    WHERE job = 'MANAGER'
    WITH READ ONLY;
```

```
DELETE FROM Manager
WHERE ename = 'JONES';
```

ORA-42399: cannot perform a DML operation on a read-only view

The Virtues of Controlled Redundancy

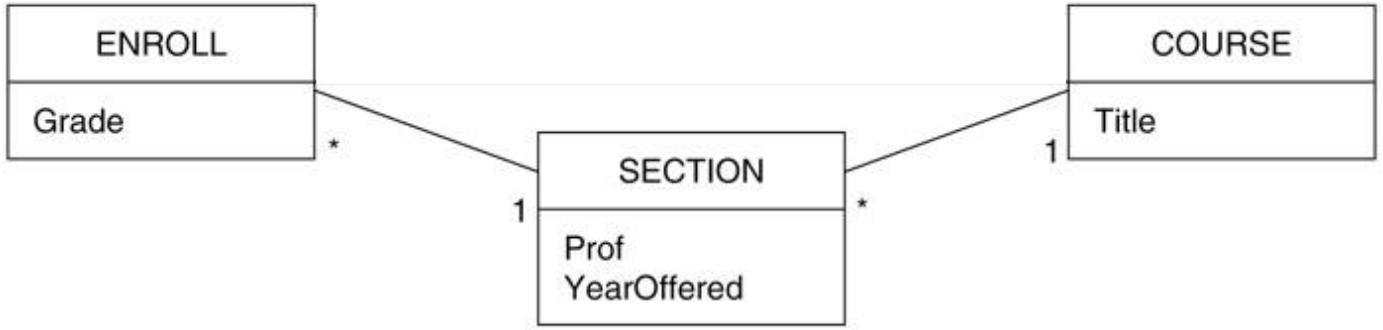
- A well-designed database should not contain redundant data.
- Redundancy is bad because it opens the door for inconsistent updates, and inconsistent updates lead to a corrupted database.
- Redundant data is valuable when it enables faster ways to execute queries.
- In order to avoid inconsistency problems, redundancy should be controlled by the database system.
- Materialized view is a form of **controlled redundancy**.
- Controlled redundancy is useful, because the database system can ensure that the database remains consistent.

Materialized Views

- A view is never stored it is only displayed. It is a virtual table.
- A materialized view is a table that contains the output of a view.
- A materialized view is stored on the disk.
- Each time that the underlying tables are updated, the database system must also update the materialized table so that it is consistent with those updates.

Materialized Views (Cont.)

- A view is worth materializing when its benefits outweigh its cost.
- The benefit of keeping a view increases when it is accessed frequently, and its output is expensive to recompute each time.
- The cost increases when the view is updated frequently, and its output is large.
- The database designer is responsible for determining which views to materialize.
- The database designer must estimate the frequency of each query and update.



(a) A class diagram with a redundant relationship

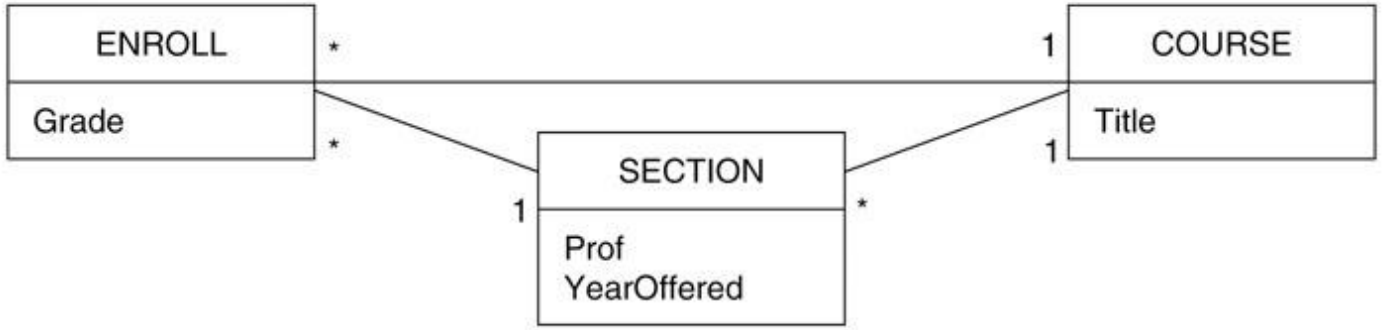
```
ENROLL(EId, SectionId,              Grade)
SECTION(SectId, CourseId, Prof, YearOffered)
COURSE(CId, Title)
```

(b) The corresponding relational schema

Figure 6-1
The effect of a redundant relationship

Controlled Redundancy

- Suppose we want to know the titles of the courses in which grades of “F” were given.
- ENROLL (EId, StudentId, SectionId, Grade)



(a) A class diagram with a redundant relationship

```
ENROLL(EId, SectionId, CourseId, Grade)
SECTION(SectId, CourseId, Prof, YearOffered)
COURSE(CId, Title)
```

(b) The corresponding relational schema

Figure 6-1
The effect of a redundant relationship

Controlled Redundancy

- Suppose we want to know the titles of the courses in which grades of “F” were given.
- ENROLL (EId, StudentId, SectionId, Grade)

```
select c.Title
from COURSE c, ENROLL e
where c.CId=e.CourseId and e.Grade='F'
```

(a) A query that uses the redundant field

```
select c.Title
from COURSE c, SECTION k, ENROLL e
where c.CId=k.CourseId and k.SectId=e.SectionId
and e.Grade='F'
```

(b) An equivalent query that does not use the redundant field

Figure 6-2
The value of a redundant *CourseId* field

Controlled Redundancy

- The redundant CourseId field makes it possible to join COURSE and ENROLL directly.
- If this redundant field did not exist, then our query would have had to include SECTION as well, as shown in Figure 6-2(b).

- ENROLL (EId, StudentId, SectionId, Grade)
- SECTION (SectId, CourseId, Prof, YearOffered)

```
create materialized view ENROLL_PLUS_CID as
   select e.*, k.CourseId
   from ENROLL e, SECTION k
   where e.SectionId=k.SectId
```

Figure 6-3
A materialized view that adds *CourseId* to ENROLL

Controlled Redundancy Materialized Views

create materialized view

The general form of the statement also gives the creator the option of specifying certain parameter values, such as how frequently the system should update the materialized table.

Materialized Views (Cont.)

- What is the difference between modified ENROLL table of Figure 6-1(b) and the materialized view ENROLL_PLUS_CID?
  - They contain the same records.
  - The difference is in how the redundant CourseId values are kept consistent.
  - In the modified ENROLL table, the user was responsible for ensuring their consistency, whereas in ENROLL_PLUS_CID, consistency is handled by the database system.

Materialized Views (Cont.)

- How does the database system can ensure the consistency of a materialized view in the face of changes to the underlying tables?
- The most obvious approach: delete the existing version of the materialized view, and then recompute it from scratch using the updated tables
  - Too time-consuming to be practical
- The system needs to be able to update the materialized table incrementally.
- The database system makes incremental changes to a materialized view each time one of its underlying tables changes.

Materialized Views (Cont.)

- What is a materialized view?
  - A materialized view is a redundant table that is maintained by the database system.
- A database system has a query optimizer, whose job is to find the most efficient implementation of user queries.
- A database system can use materialized views to optimize an SQL query, even when that query does not mention those views.
- The decision to use a materialized view in a query should be made by the query optimizer, not the user.

Why Use Materialized Views?

- Hide some data from some users
- Make some queries easier / more natural
- Modularity of database access
- Improve query performance

References

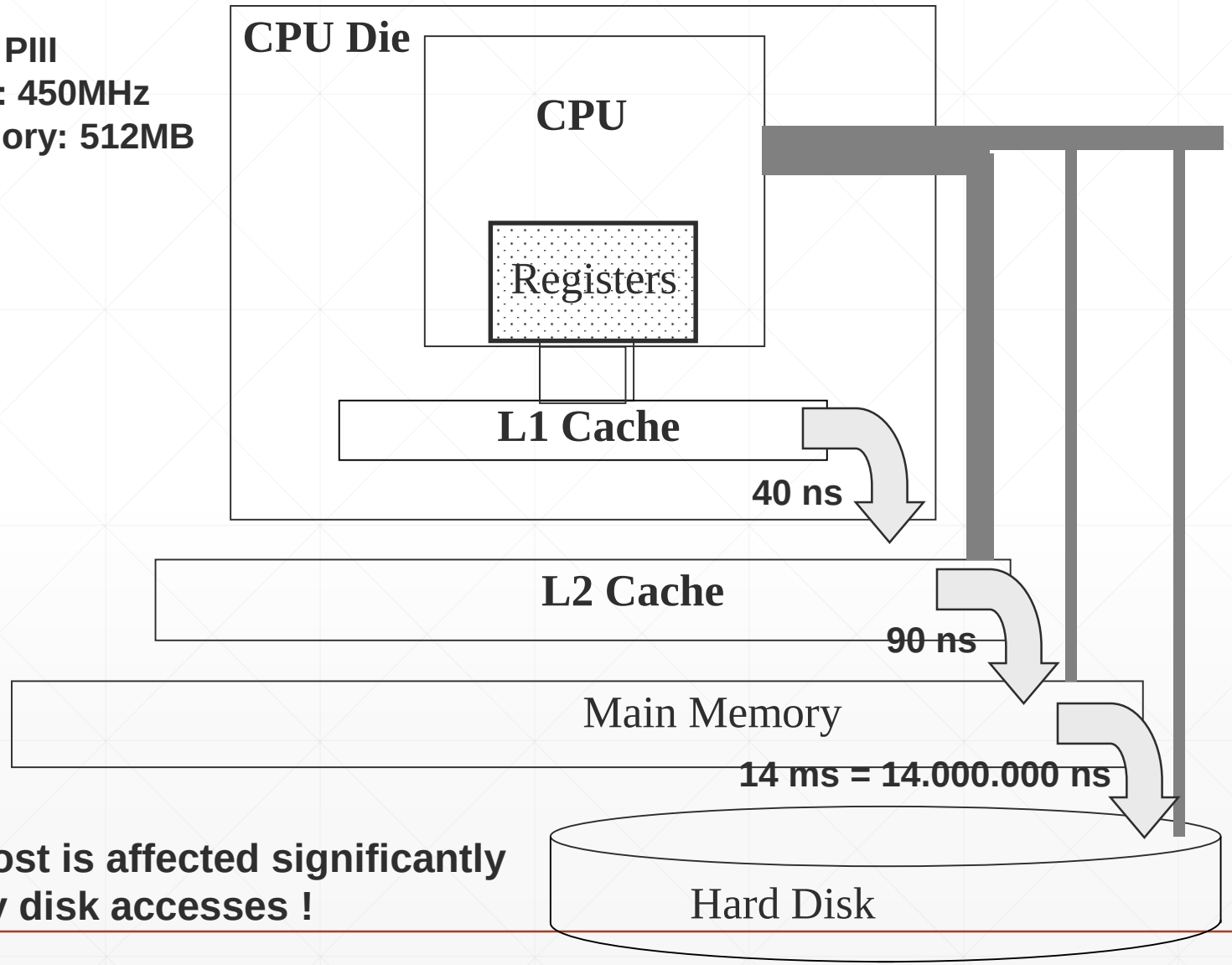
- Edward Sciore, Database Design and Implementation, Wiley, 2009.
  - Chapter 4, 6
- Stanford DB Class
- Docs Oracle

Indexes

CENG315 INFORMATION MANAGEMENT

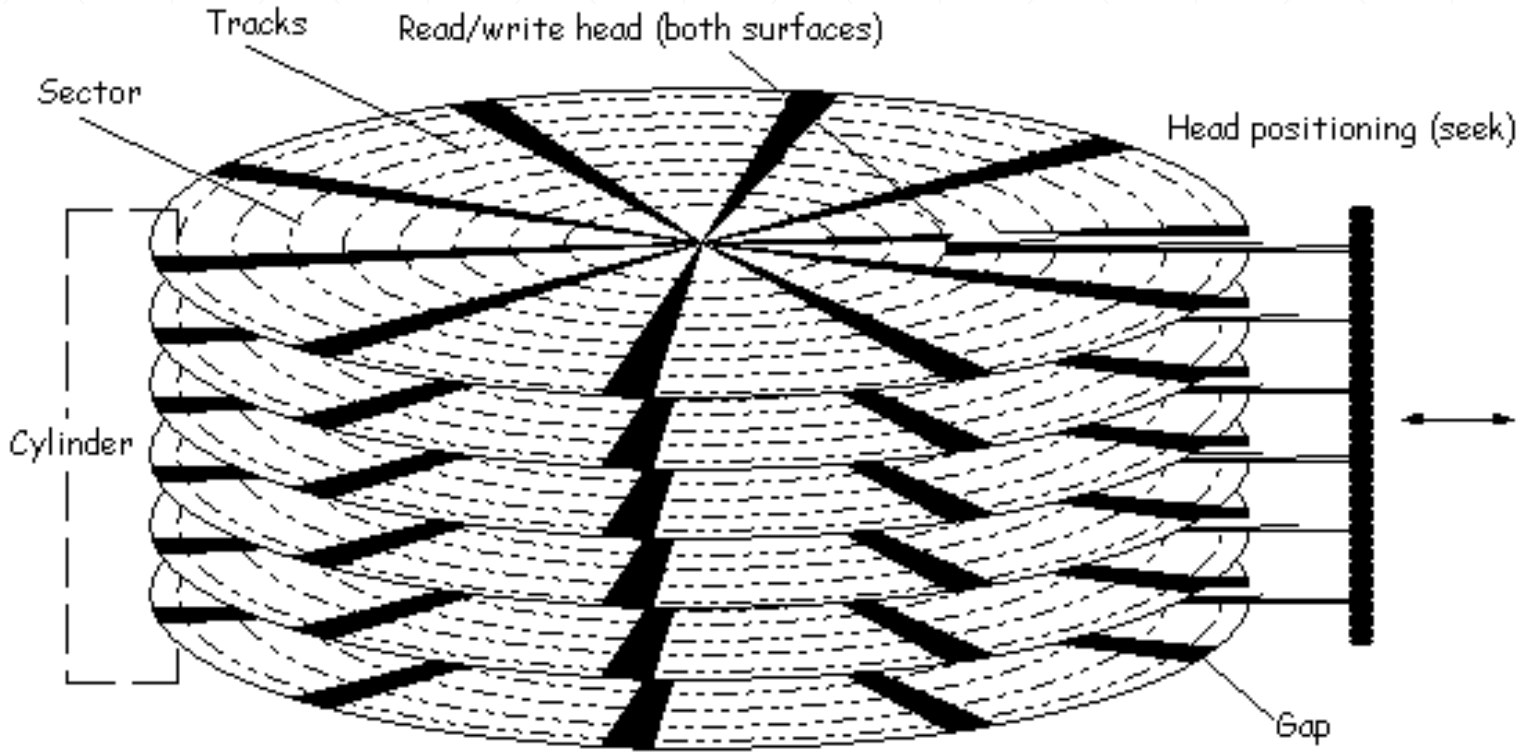
The Memory Hierarchy

Intel PIII
CPU: 450MHz
Memory: 512MB



Cost is affected significantly by disk accesses !

The Hard (Magnetic) Disk



- The time for a ***disk block access***, or ***disk page access*** or ***disk I/O***

  access time = seek time + rotational delay + transfer time

- IBM Deskstar 14GPX: 14.4GB
  - Seek time: 9.1 msec
  - Rotational delay: 4.17 msec
  - Transfer rate: 13MB/sec, that is, 0.3msec/4KB

Cost of a Query for Disk based Databases

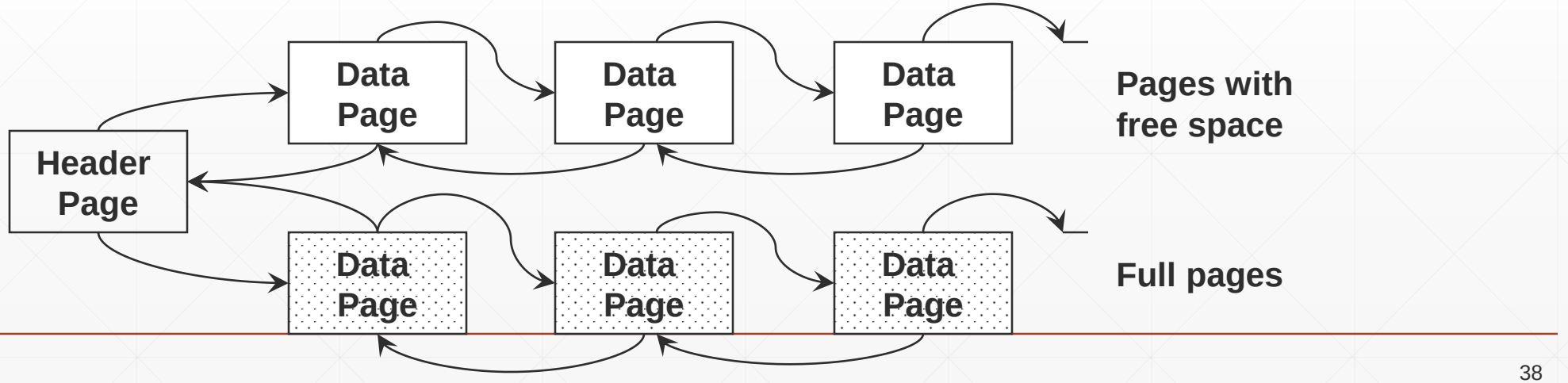
- Cost measure: *number of page accesses*

- Basic Query Types
  - Exact match (point query)
    - Q1: Find me the book with the name “Database”
  - Range query
    - Q2: Find me the books published between year 2003-2005

Basic File Organization : Heap Files

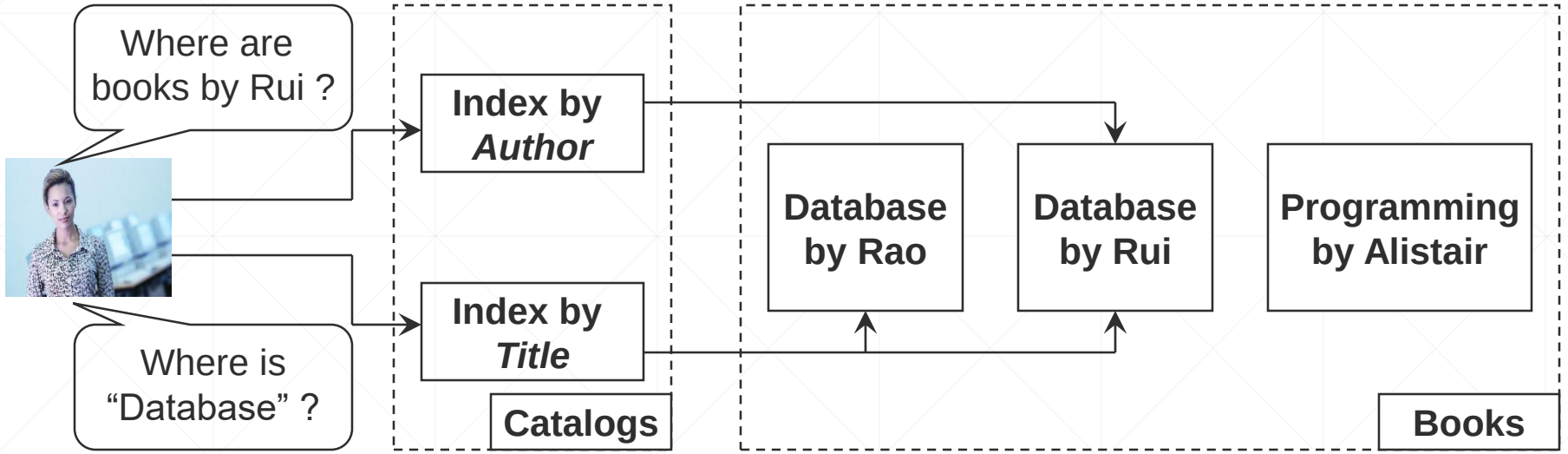
- **File** : a logical collection of data, physically stored as a set of pages.

- **Heap File** (Unordered File)
  - Linked list of pages
  - The DBMS maintains the header page: <*heap_file_name*, *header_page_address*>
  - Operations
    - Insertion
    - Deletion
    - Search
  - Advantage: Simple
  - Disadvantage: Inefficient



The Concept of Index

- Searching a Book…



- Index
  - A data structure that helps us find data quickly

- Note
  - Can be a separate structure (we may call it the index file) or in the records themselves.
  - Usually sorted on some attribute

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

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order (ISAM, B^+-Tree).
  - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function” (static, dynamic).

Index Evaluation Metrics

- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute (like Q1)
  - or records with an attribute value falling in a specified range of values (like Q2)
- Access time
- Insertion time
- Deletion time
- Space overhead

Query, Key and Search Key

- Queries
  - Exact match (point query)
    - Q1: Find me the book with the name “Database”
  - Range query
    - Q2: Find me the books published between year 2003-2005

- Searching methods
  - Sequential scan – too expensive
  - Through index – if records are sorted on some attribute, we may do a binary search
    - If sorted on “book name”, then we can do binary search for Q1
    - If sorted on “year published”, then we can do binary search for Q2

- Key vs. Search key
  - Key: the indexed attribute
  - Search key: the attribute queried on

Index Definition in SQL

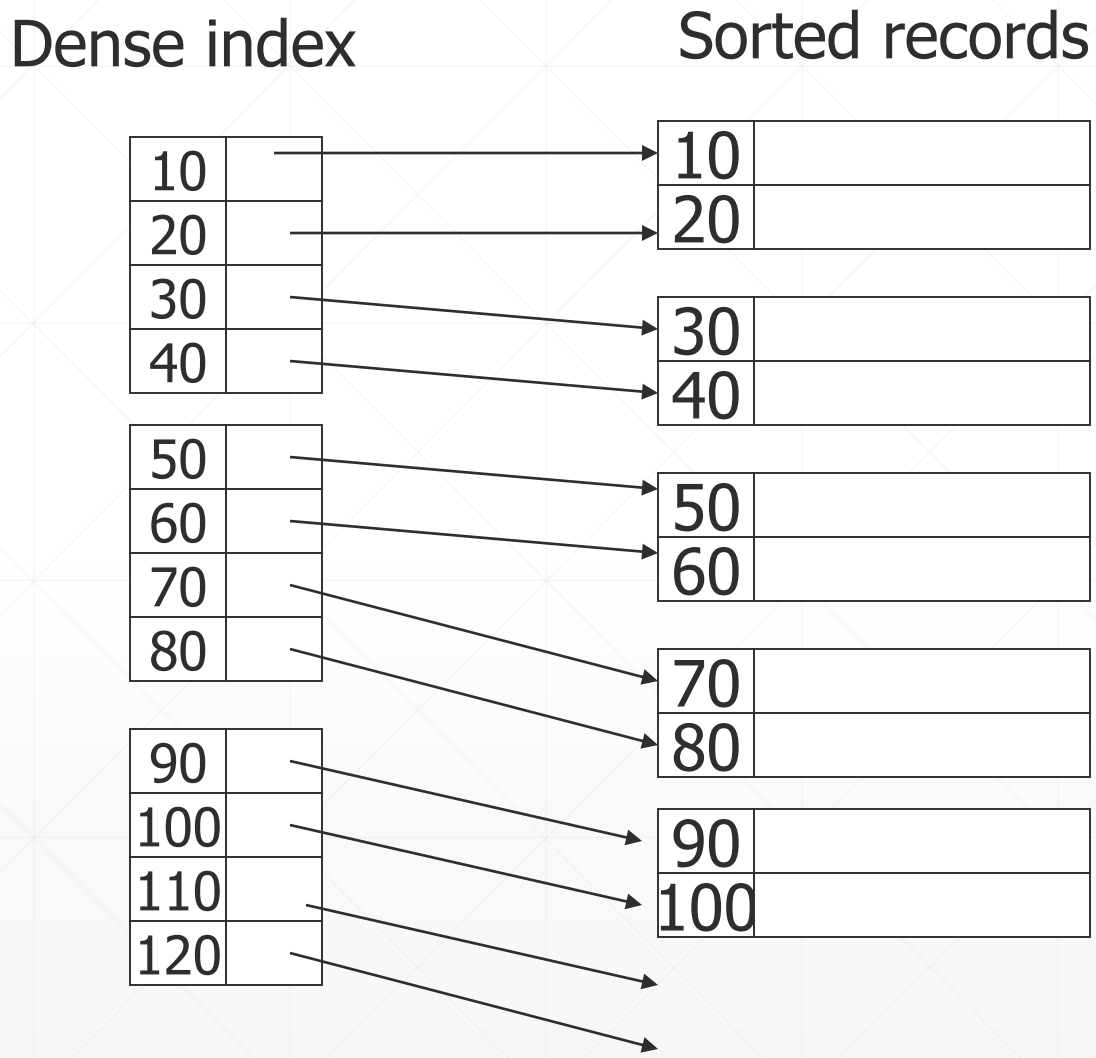
- Create an index

  create index <index-name> **on** <relation-name>
  (<attribute-list>)

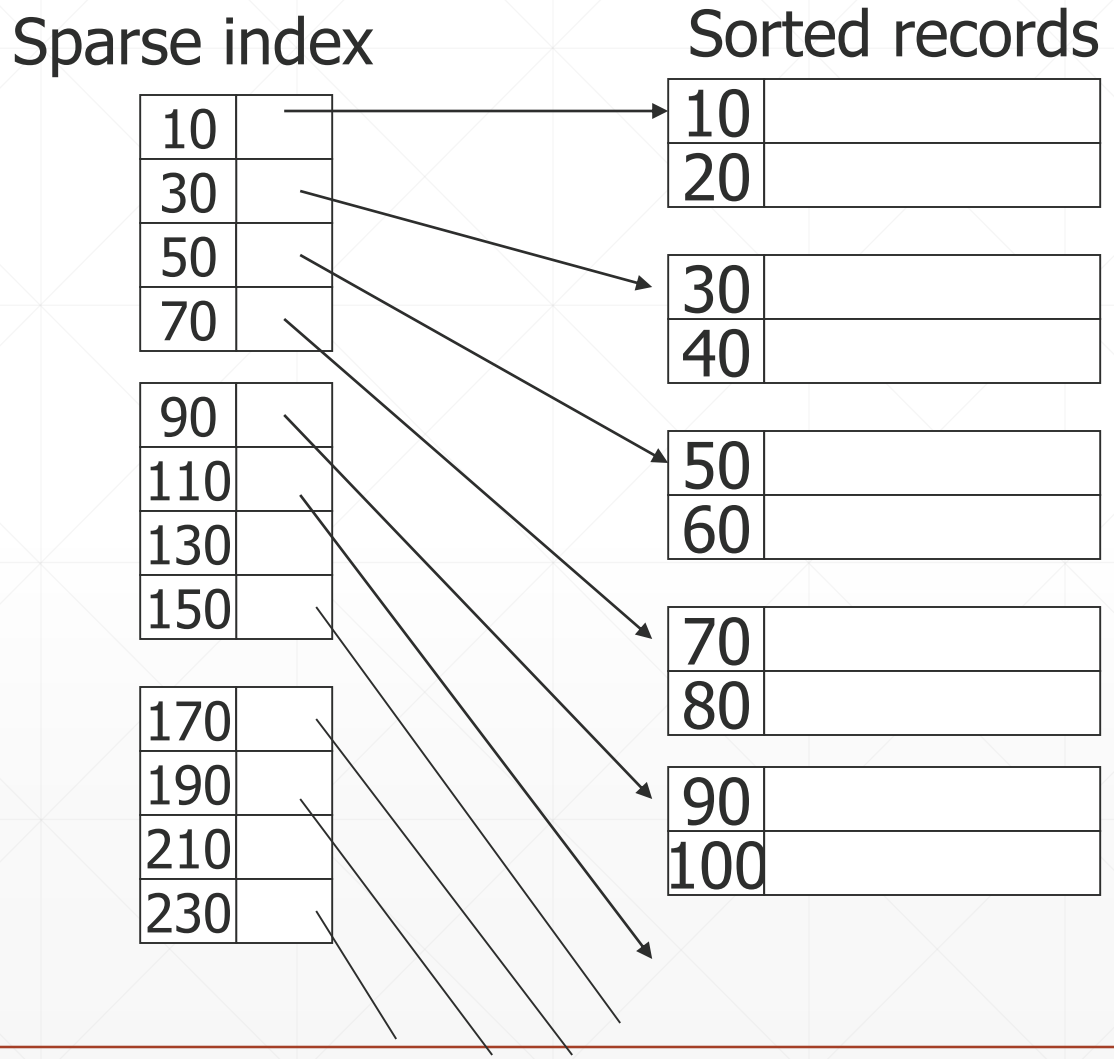
  E.g.: **create index** *a-index* **on** *book(author)*

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

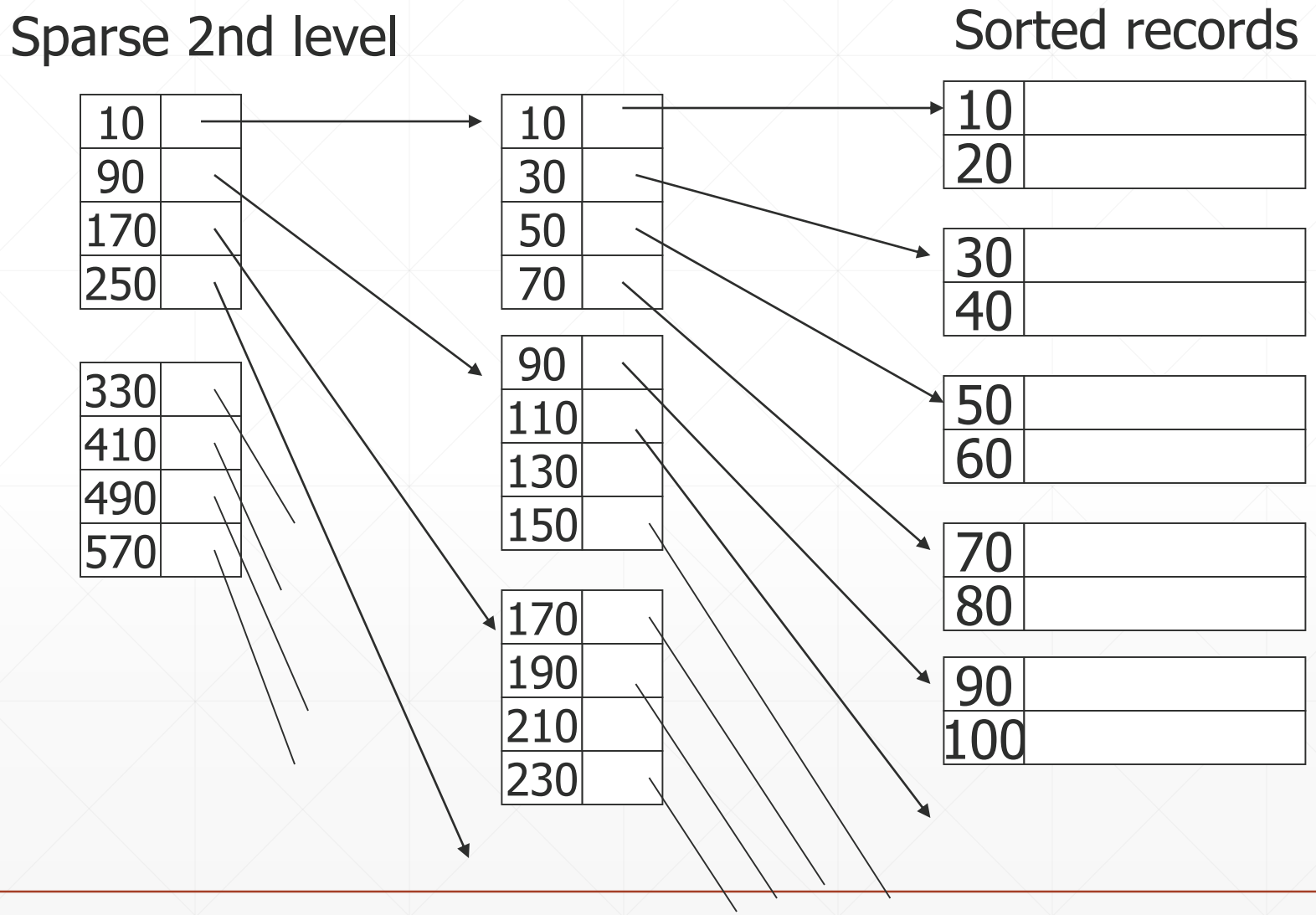
Simple Index File (Clustered, Dense)



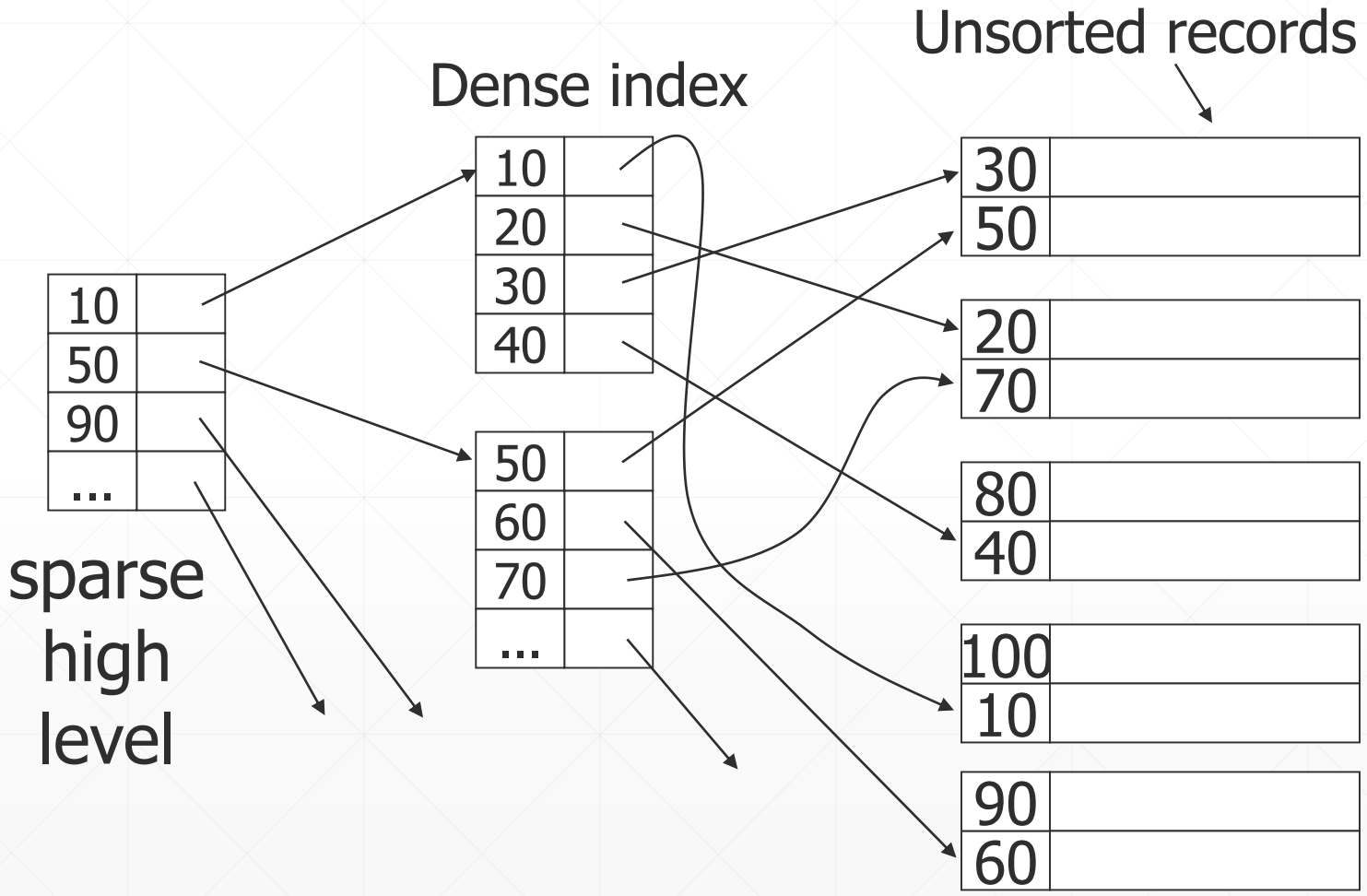
Simple Index File (Clustered, Sparse)



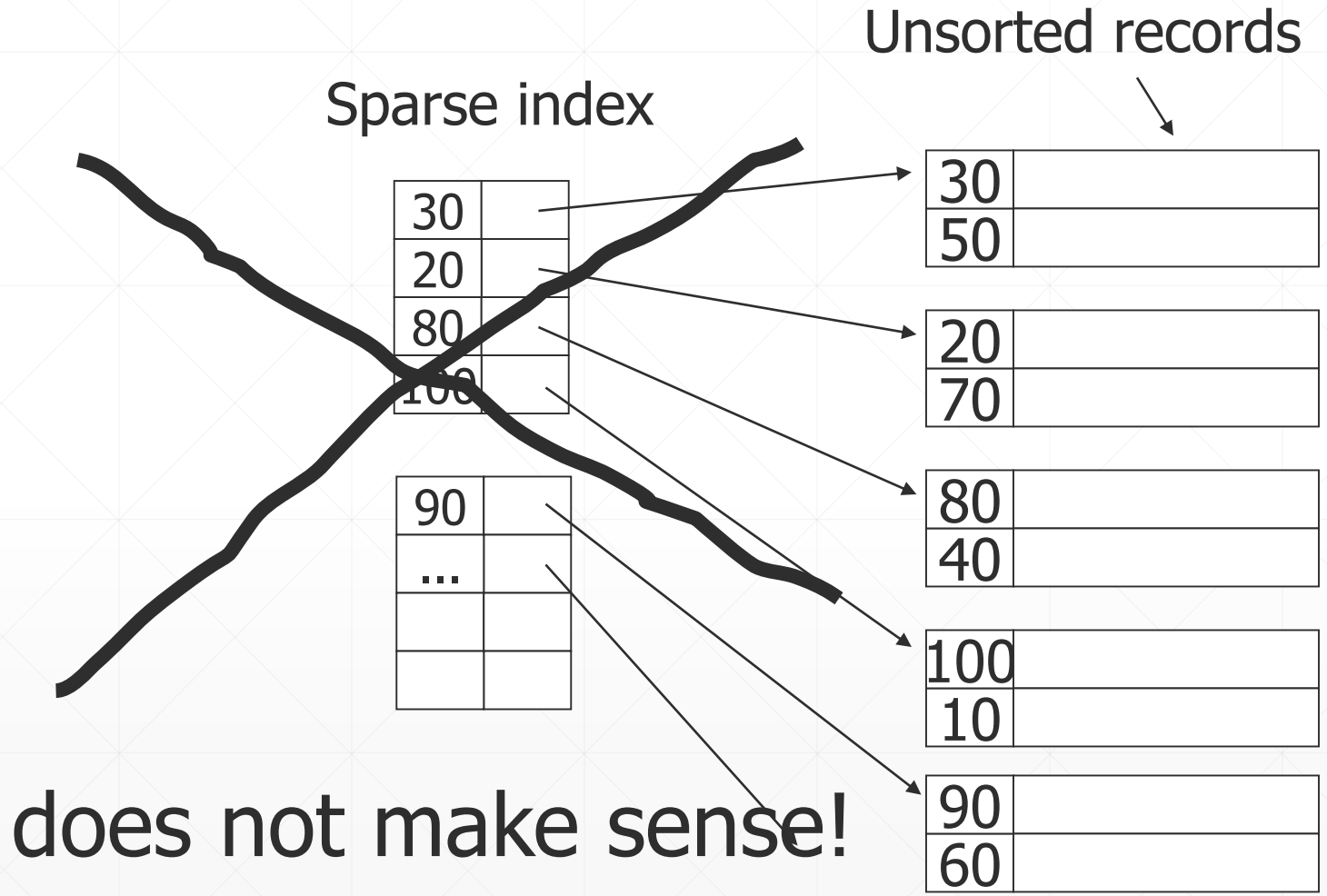
Simple Index File (Clustered, Multi-level)



Simple Index File (Unclustered, Dense)

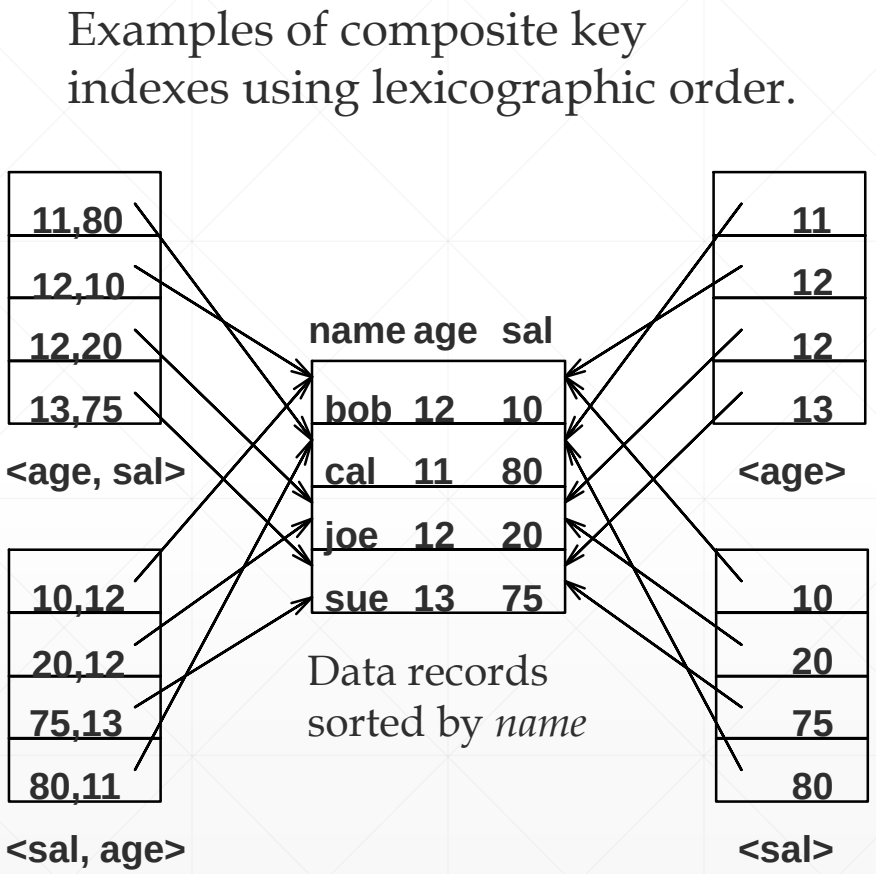


Simple Index File (Unclustered, Sparse ?)



Indexes on Composite Keys

- Q3: age=20 & sal=10
- Index on two or more attributes: entries are sorted first on the first attribute, then on the second attribute, the third …
- Q4: age=20 & sal>10
- Q5: sal=10 & age>20
- Note
  - Different indexes are useful for different queries

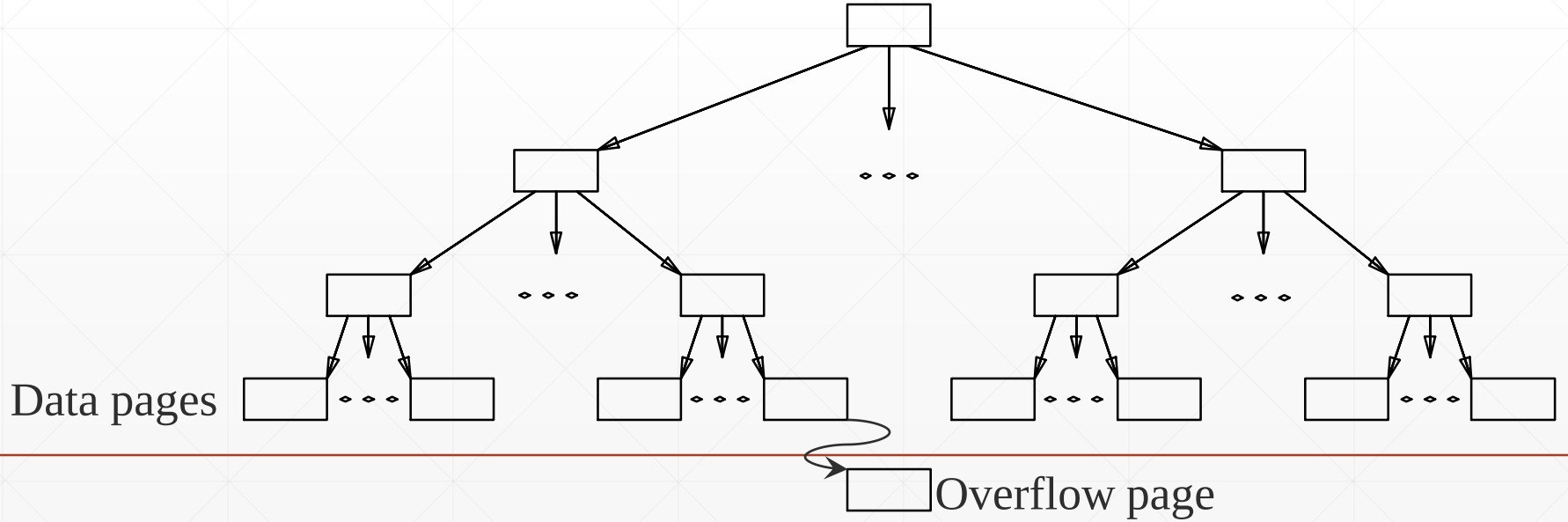


Properties of Indexes

- Alternatives for entries in an index
  - An actual data record
  - A pair (key, ptr)
  - A pair (key, ptr-list)
- Clustered vs. Unclustered indexes
- Dense vs. Sparse indexes
  - Dense index on clustered or unclustered attributes
  - Sparse index only on clustered attribute
- Primary vs. Secondary indexes
  - Primary index on clustered attribute
  - Secondary index on unclustered attribute

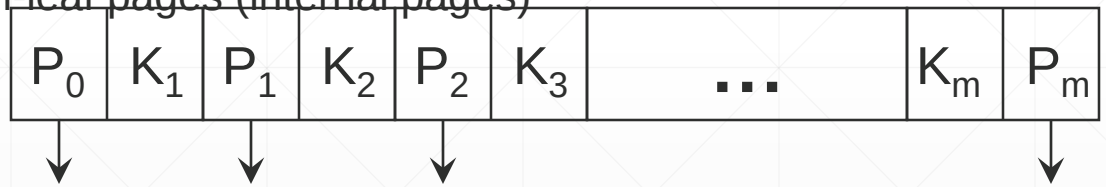
Indexed Sequential Access Method (ISAM)

- Tree structured index
- Support queries
  - Point queries
  - Range queries
- Problems
  - Static: inefficient for insertions and deletions

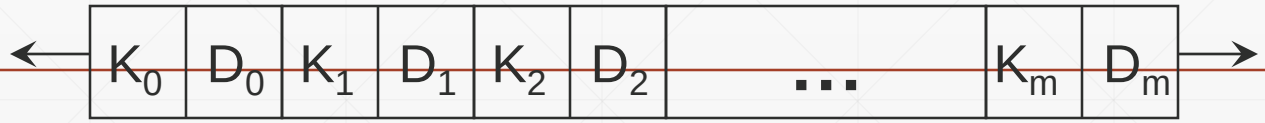


The B+-Tree: A Dynamic Index Structure

- Grows and shrinks dynamically
- Minimum 50% occupancy (except for root).
  - Each node contains *d* <= *m* <= *2d* entries. The parameter *d* is called the ***order of the tree***.
- Height-balanced
  - Insert/delete at $log_f N$ cost (*f* = fanout, *N* = No. leaf pages)
- Pointers to sibling pages
  - Non-leaf pages (internal pages)

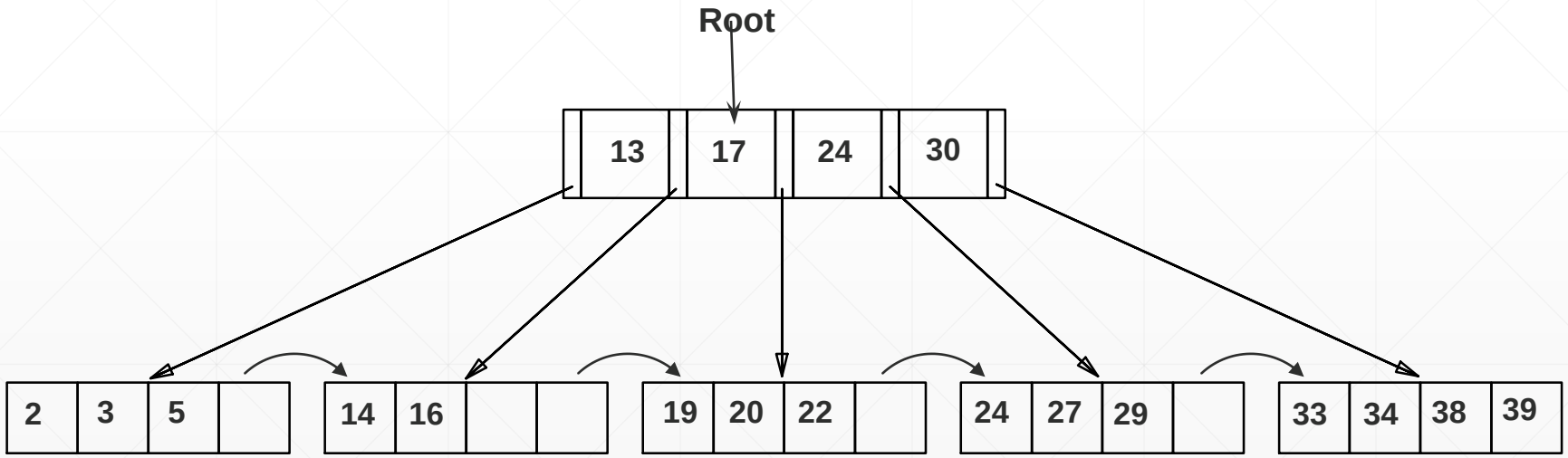


  - Leaf pages
    - If directory page, same as non-leaf pages; pointers point to data page addresses
    - If data page



Searching in a B+-Tree

- Search begins at root, and key comparisons direct it to a leaf (as in ISAM)
- Search for 5, 15, all data entries >= 24 ...
- What about all entries <= 24 ?



Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function.**
- Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

Example of Hash File Organization

Hash file organization of instructor file, using dept_name as key
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the *i*th character is assumed to be the integer *i.*
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g. h(Music) = 1 h(History) = 2
    h(Physics) = 3 h(Elec. Eng.) = 3

Example of Hash File Organization

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

22222	Einstein	Physics	95000
33456	Gold	Physics	87000
98345	Kim	Elec. Eng.	80000

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

bucket 7

Hash file organization of instructor file, using dept_name as key (see previous slide for details).

Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization (ISAM)
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B^+-trees

Dropping Index with SQL

DROP INDEX *index_name* ON *table_name*;

- Indexes should not be used on small tables.
- Tables that have frequent, large batch updates or insert operations.
- Indexes should not be used on columns that contain a high number of NULL values.
- Columns that are frequently manipulated should not be indexed.

Advanced Aspects of Indexes

- https://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_5010.htm

•Normal indexes. (By default, Oracle Database creates B-tree indexes.)

•**Bitmap indexes**, which store rowids associated with a key value as a bitmap

•**Partitioned indexes**, which consist of partitions containing an entry for each value that appears in the indexed column(s) of the table

•**Function-based indexes**, which are based on expressions. They enable you to construct queries that evaluate the value returned by an expression, which in turn may include built-in or user-defined functions.

•**Domain indexes**, which are instances of an application-specific index of type *indextype*

References

- Many contents are from

Database Management Systems, 3rd edition.
Raghu Ramakrishnan & Johannes Gehrke
McGraw-Hill, 2000.

Database Systems Concepts, 6th edition.
Silberchatz, Korts & Sudarshan

- Some are from the internet