



Physical Schema

ECE 356
University of Waterloo
Dr. Paul A.S. Ward

Acknowledgment: slides derived from Dr. Wojciech Golab
based on materials provided by
Silberschatz, Korth, and Sudarshan, copyright 2010
(source: www.db-book.com)

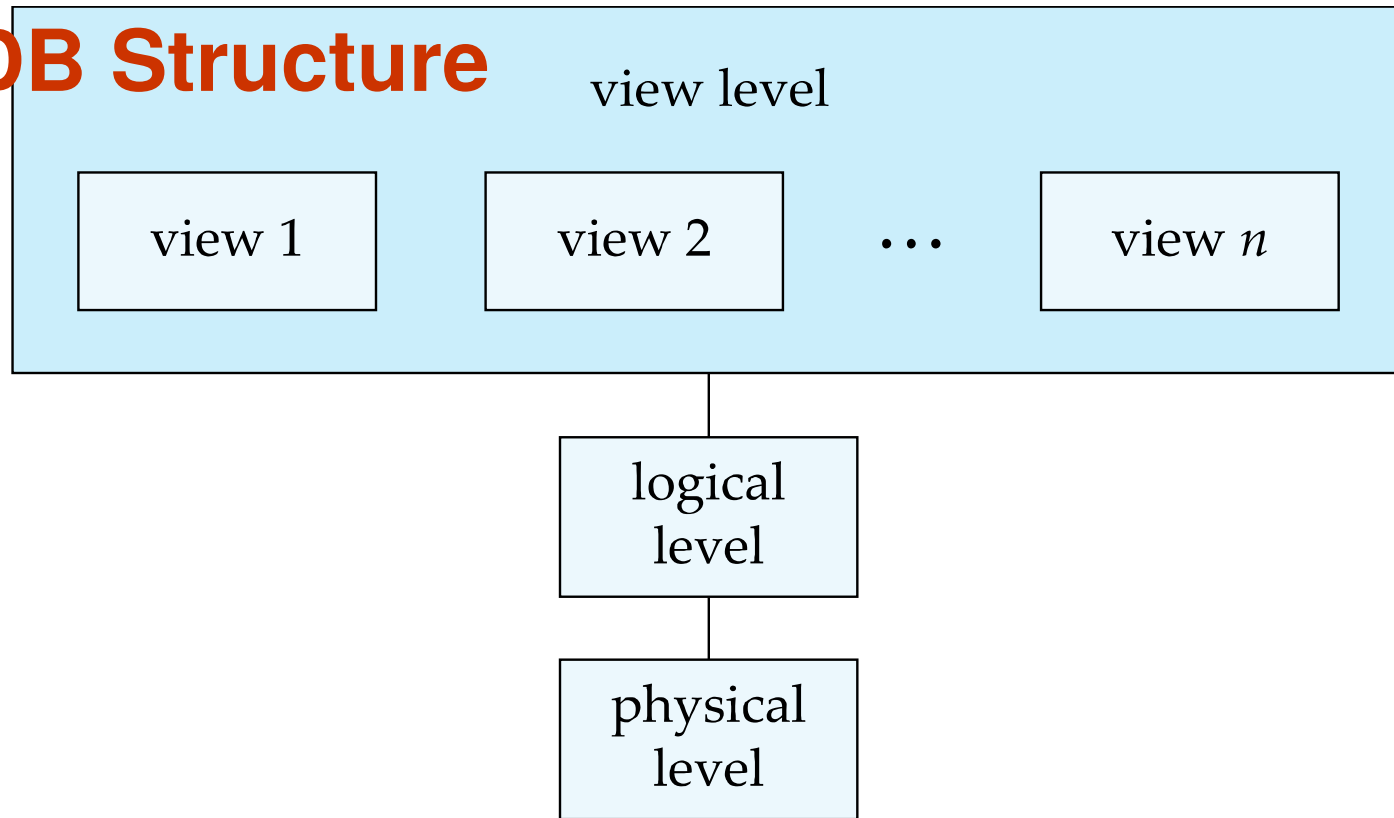


Learning Outcomes

- Storage media
 - File structures
 - Buffer Management
 - Index Techniques
 - Query Optimization
-
- Textbook sections (6th ed.): chapters 10, 11, 12, 13.
 - MySQL reference manual (v5.6) : Chapter 8



Recall DB Structure



■ **Schema** – the shape or structure of the database.

- **logical schema:** Design at logical level
 - ▶ Data types, relations, rows, columns
- **physical schema:** design at physical level
 - ▶ Storage, file formats, buffers, indexes
 - ▶ Why? Performance



Properties of Storage Media

- **speed** with which data can be accessed
 - latency per I/O operation (IOP) vs. throughput
 - reads vs. writes, sequential vs. random access, empty vs. full
- **cost** (e.g., \$ per GB, Joules per bit stored or accessed)
- **density** (e.g., bits per square inch)
- **volatility**
 - **volatile media:** loses content when power is switched off
 - ▶ E.g., CPU registers, cache memory, main memory
 - **non-volatile media:** content persists when power is switched off
 - ▶ E.g., hard drives (HD), solid-state drives (SSD), battery-backed memory
- **reliability**
 - mean time between failures (MTBF)
 - number of write cycles until wear-out



Physical Storage Media

■ CPU registers and cache

- **volatile**, SRAM-based, managed by processor
- capacities commonly ~10 KB (L1), ~100 KB (L2), ~10 MB (L3)

■ main memory

- **volatile**, SDRAM-based, managed by OS/application
- latency 1-10 ns, capacities ~10 GB up to ~1 TB
(increasingly large/inexpensive enough to store the entire DB)

■ flash memory (e.g., NAND flash)

- **non-volatile**, used in solid-state drives (SSDs) and USB drives
- latency 10-100 μ s, capacities ~100 GB to ~1 TB
- drawback 1: cells must be erased before they can be re-written
- drawback 2: cells wear out after 10K–1M write/erase cycles
- **flash translation layer** remaps logical addresses to physical addresses to mask erase latency and worn out pages, also performs **wear leveling** and **sparing** to increase longevity



Physical Storage Media (Cont.)

■ conventional (magnetic) hard disk

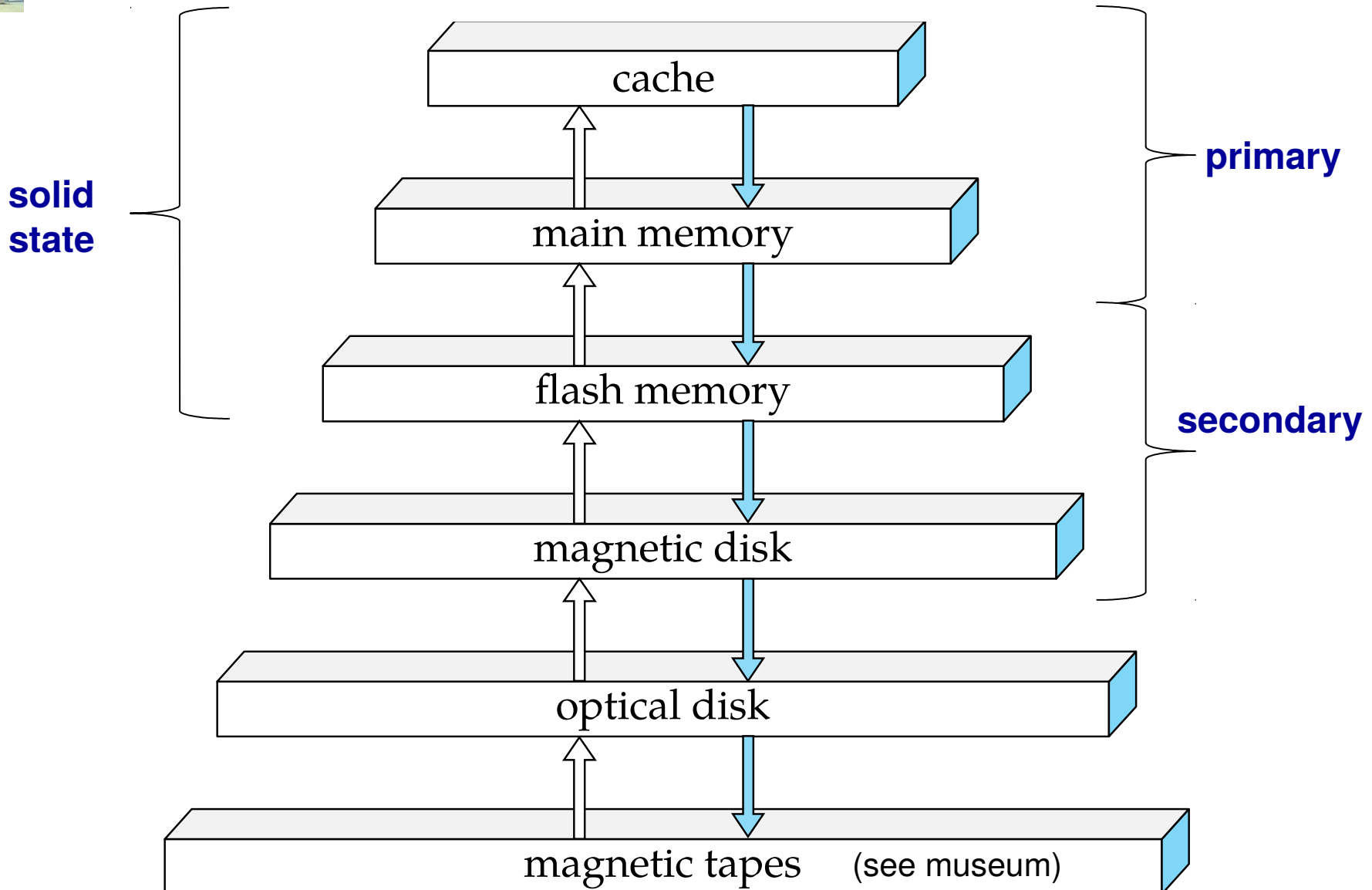
- **non-volatile**, mature technology, less expensive than flash
- most popular medium for **reliable long-term storage** of data
- latency between 3ms (enterprise) and 15ms (mobile)
 - ▶ seek time (< 1 ms) to move read/write heads
 - ▶ rotational latency (a few ms) to position platter under head
- capacities commonly in the hundreds of GB up to several TB
- **much slower for random I/O than for sequential I/O**
(*e.g.*, < 1 MB/s random vs. 150 MB/s sequential)
- susceptible to mechanical vibrations and shocks

■ optical disks (CD, DVD, BlueRay)

- **non-volatile**, least expensive, mostly read-only
- latency on the order of 100ms, capacities up to 100 GB
- used to store DBMS binaries and data sets, record backups

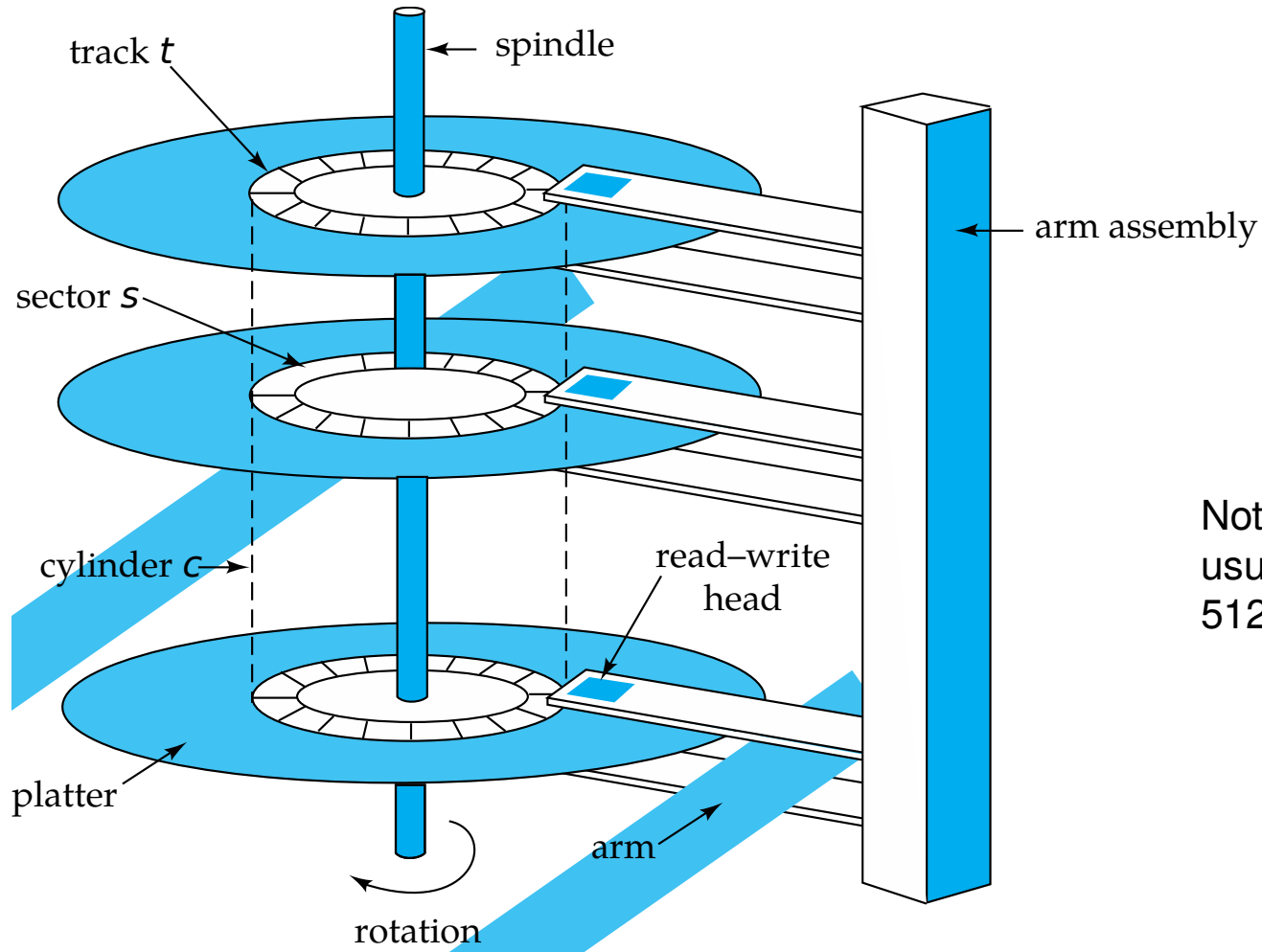


Storage Hierarchy





Hard Disk Drive (HDD) Internals



Note: sector size usually from 512B to 4KB

Note: On-board circuitry controls head movement, schedules I/O requests, and buffers data.

Interesting video: <https://www.youtube.com/watch?v=4iaxOUYaIJU>



Solid State Drive (SSD) Internals

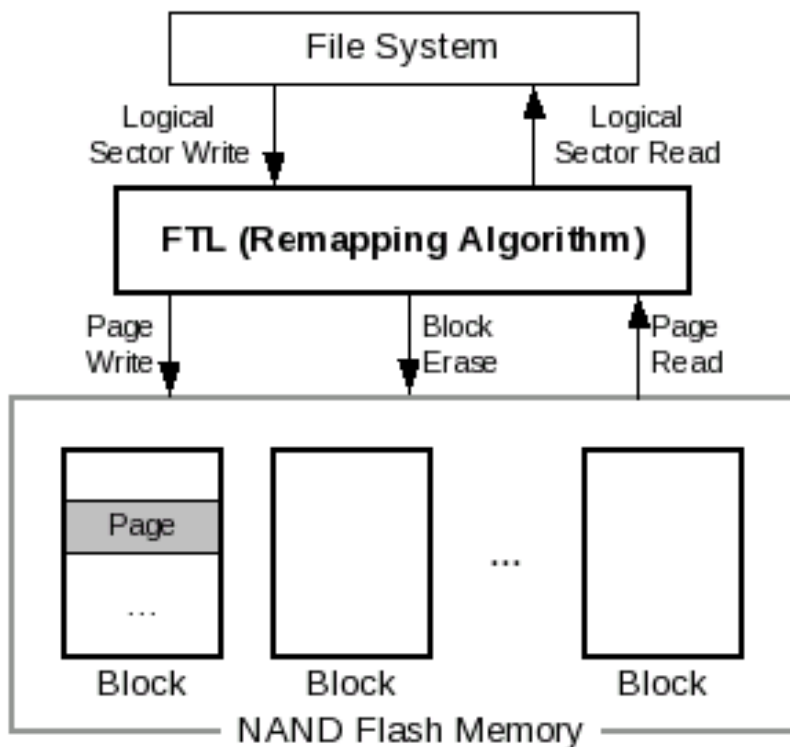
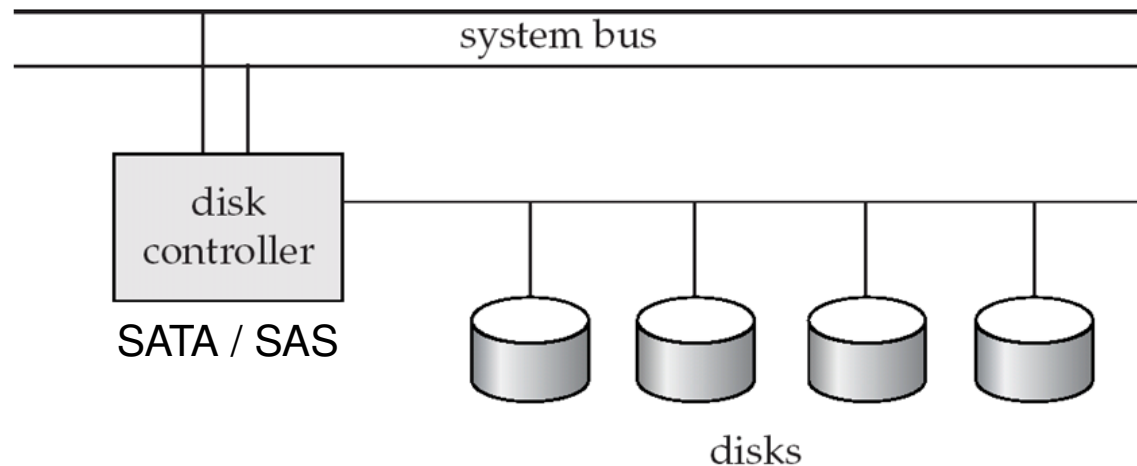


Image source: Hyojun Kim and Seongjun Ahn. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage. FAST 2008.

- SSD provides sector read and write operations to file system, just like a magnetic hard disk.
- Internally, an SSD maps **sectors** (logical) to **pages** (physical).
- The SSD can write data one page at a time, but it must **erase** an entire **block** of pages before overwriting data.
- Erasing a block is much slower than reading or writing a page.
- Overwriting the same flash cell repeatedly leads to **wear-out**.
- The flash translation layer (FTL) remaps pages and blocks to enable efficient overwriting, as well as to perform wear leveling.



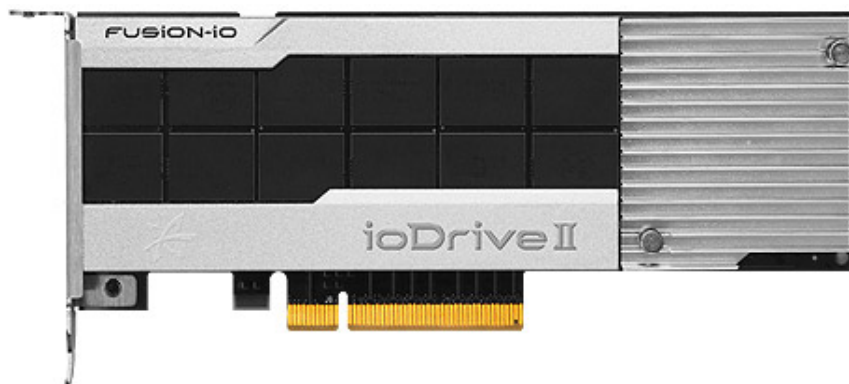
Conventional Block I/O Devices



- **block:** a contiguous sequence of sectors from a single track
 - unit of storage **allocation** and **data transfer**
 - sizes range from 512 bytes to several kilobytes
 - ▶ smaller blocks: more transfers from disk
 - ▶ larger blocks: more space wasted due to partially filled blocks
 - ▶ typical block sizes today range from 4 to 16 kilobytes
- **observation:** secondary storage devices in general are block devices.



Cutting Edge SSDs



\$6000 to \$23000
(2015)

ioDrive2 Capacity	365GB MLC	785GB MLC*	1.2TB MLC*	ioDrive2 3.0TB MLC
Read Bandwidth - 1MB	910 MB/s	1.5 GB/s	1.5 GB/s	1.5 GB/s
Write Bandwidth - 1MB	590 MB/s	1.1 GB/s	1.3 GB/s	1.3 GB/s
Ran. Read IOPS - 512B	137,000	270,000	275,000	143,000
Ran. Write IOPS - 512B	535,000	800,000	800,000	535,000
Ran. Read IOPS - 4K	110,000	215,000	245,000	136,000
Ran. Write IOPS - 4K	140,000	230,000	250,000	242,000
Read Access Latency	68μs	68μs	68μs	68μs
Write Access Latency	15μs	15μs	15μs	15μs
Bus Interface	PCI-Express 2.0 x4			

source: <http://www.fusionio.com/data-sheets/iodrive2/>



RAID

- **RAID: Redundant Arrays of Inexpensive/Independent Disks** – a disk organization technique that uses multiple physical disks to provide the illusion of a single more reliable and/or more performant disk.
 - increases **capacity** and **speed** by using multiple disks in parallel
 - increases **reliability** through redundancy, ensuring survival of data if a (small enough) subset of disks fails

- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
 - *E.g.*, a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days).



Improvement of Reliability via Redundancy

- **Redundancy:** store extra information that can be used to rebuild information lost in a disk failure.
- **Mirroring:** duplicate every disk.
 - One logical disk comprises two physical disks.
 - Every write is carried out on both disks but reads can be served using either disk.
 - If one disk in a pair fails, data remains available in the other. Data loss occurs only if a disk fails and its mirror also fails before the system is repaired (low probability event).
- **Parity bits:** store additional bits to compensate for corrupted ones.
 - Additional bits used to detect and (possibly) correct errors.
- **Mean time to data loss:** depends on mean time to failure, and **mean time to repair**.
 - *E.g.*, MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500×10^6 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes).



Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 - increase throughput by distributing small I/O requests across multiple disks
 - reduce response time by parallelizing large I/O requests
- **Striping**: write data across multiple disks to improve throughput. (Note: mirroring alone only allows us to parallelize reads.)
- **Block-level striping**: with n disks, block i of a file goes to disk number $(i \bmod n) + 1$.
 - Requests for different blocks can run in parallel if the blocks reside on different disks.
 - A request for a long sequence of blocks can utilize all disks in parallel.



RAID Levels

- **RAID levels** are schemes for providing redundancy at lower cost by using disk striping combined with mirroring or parity bits.
- Many RAID levels exist, with different cost, performance and reliability characteristics. We will only cover a few that are used frequently in practice.
- **RAID Level 1+0 (a.k.a. RAID 10):** mirrored disks with block striping.
 - two copies of everything (relatively expensive)
 - best write performance, supports reading in parallel from both mirrors
 - straightforward recovery when disk fails (copy from mirror)
 - good for update-heavy workloads

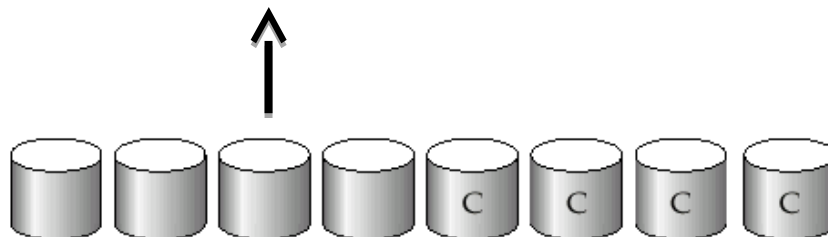


(b) RAID 1: mirrored disks



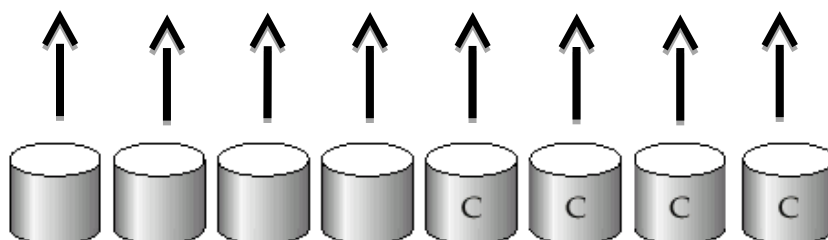
RAID 10 Read Examples

- Small read: read block from one disk.



Note: RAID 10 is able to read multiple small files in parallel.

- Large read: read blocks from all disks in parallel.

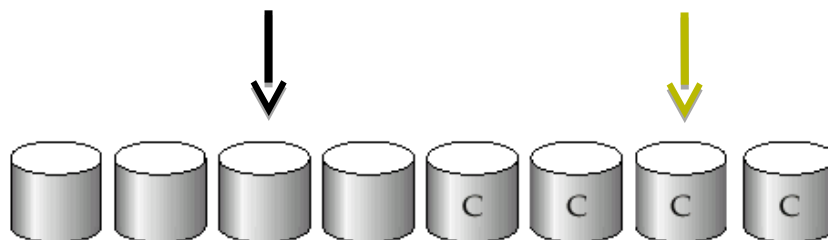


When performing many small random reads or one large read, the throughput is up to 8x better than using one disk. (No penalty.)



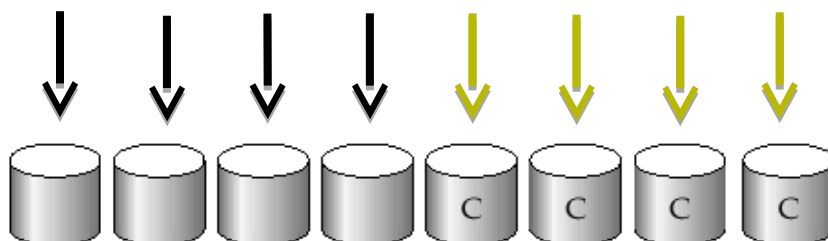
RAID 10 Write Examples

- Small write: write block to one disk plus its mirror.



Note: RAID 10 is able to write multiple small files in parallel.

- Large write: write blocks to four disks in parallel + mirrors.



When performing many small random writes or one large write, the throughput is up to 4x better than using one disk. (2x penalty.)



RAID Levels (Cont.)

■ RAID Level 5: block-interleaved distributed parity

- Partitions data and parity among $N + 1$ disks ($N \geq 2$).
- *E.g.*, with 5 disks, parity block for n -th set of blocks is stored on disk $(n \bmod 5) + 1$, with the data blocks stored on the other 4 disks.
- Recovery entails reading N remaining disks, slower than RAID 10.
- More cost-effective than RAID 10, but slower in some cases.
- Good for reads and large writes, but small writes pay a penalty: they must read-modify-write data to update parity.



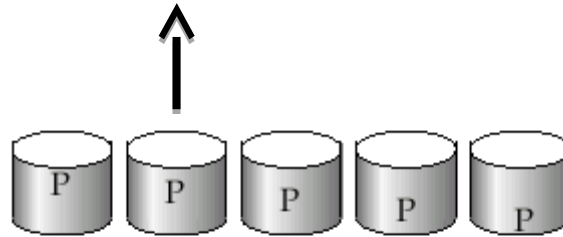
(f) RAID 5: block-interleaved distributed parity

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4



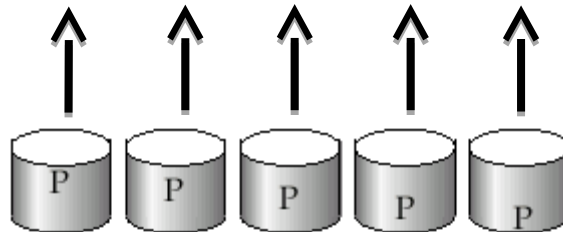
RAID 5 Read Examples

- Small read: read block from one disk.



Note: RAID 5 is able to read multiple small files in parallel.

- Large read: read blocks from all disks in parallel.

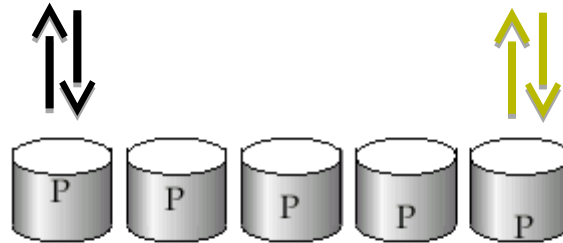


When performing many small random reads, the throughput is up to 5x better than using one disk. (No penalty.) When performing a large read we have to skip over the parity blocks. In this case we can estimate the large read throughput as 4x better than one disk because if we read all five disks in parallel then 4/5 of the blocks are data and 1/5 are parity.



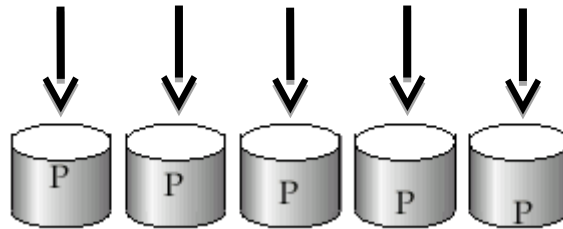
RAID 5 Write Examples

- Small write: read and write one data block + one parity block.



Note: RAID 5 is able to write multiple small files in parallel.

- Large write: write data to all disks in parallel.



When performing many small random writes, there are four physical I/Os for every logical I/O. Therefore the small write throughput is up to $(5/4) \times$ better than using one disk. (4x penalty.) When performing a large write we have to write parity blocks. In this case we can estimate the large write throughput as 4x better than one disk because if we write all five disks in parallel then 4/5 of the blocks are data and 1/5 are parity.



RAID Levels (Cont.)

- Food for thought:
 - How does block size affect RAID performance?
 - ▶ reads vs. writes
 - ▶ small vs. large operations
 - What pattern of access does a database generate?
 - ▶ queries vs. insertions and updates
 - How important is cost vs. fast recovery in practice?
 - How do different RAID levels perform with one failed disk (*i.e.*, before the disk is restored)?



Stable Storage

- A hypothetical (?) form of storage that survives all failures
- Approximated in reality by maintaining multiple copies of data on distinct nonvolatile media (*e.g.*, using RAID)
 - The degree and nature of replication is a function of the threat model:
 - ▶ independent drive failure in a machine (*e.g.*, head crash)
 - ▶ loss of a machine (*e.g.*, motherboard failure)
 - ▶ loss of a rack (*e.g.*, TOR switch failure)
 - ▶ loss of a data center (*e.g.*, cables cut to data center)
 - ▶ loss of a geography (*e.g.*, earthquake in area)
- Plays a crucial role in the database recovery



File Organization

- To a first approximation, we can think of the DB as a collection of **files** representing relations. Each file is a sequence of **records** representing tuples. A record is a sequence of **fields** representing attributes.

- One approach:

- assume record size is fixed
- each file has records of one particular type only
- different files are used for different relations

This case is easiest to implement; will consider variable length records later.

- Observation:

- files usually stored on block I/O device (e.g., hard disk)
- therefore, a file is a sequence of blocks, also called **pages**



Fixed-Length Records

■ Simple approach:

- Store record i starting from byte $n * (i - 1)$, where n is the size of each record.
- Record access is simple but records may cross blocks.
Remedy: do not allow records to cross block boundaries

■ Alternatives for deletion of record i :

- move records $i + 1, \dots, n$ to $i, \dots, n - 1$ (expensive)
- move record n to i (destroys sort order)
- do not move records, but link all free records in a *free list*

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



Free Lists

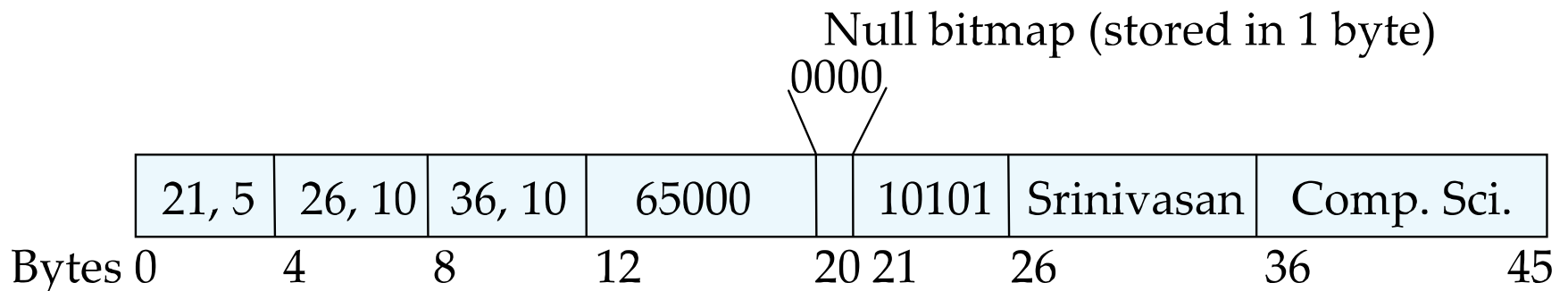
- Store the address of the first deleted record in the file header.
- The i^{th} deleted record records a pointer to the $(i+1)^{st}$ deleted record.
- **Problem:** if the table is sorted by instructor ID then how can we insert a record efficiently without reclaiming a deleted record?

header				
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1				
record 2	15151	Mozart	Music	40000
record 3	22222	Einstein	Physics	95000
record 4				
record 5	33456	Gold	Physics	87000
record 6				
record 7	58583	Califieri	History	62000
record 8	76543	Singh	Finance	80000
record 9	76766	Crick	Biology	72000
record 10	83821	Brandt	Comp. Sci.	92000
record 11	98345	Kim	Elec. Eng.	80000



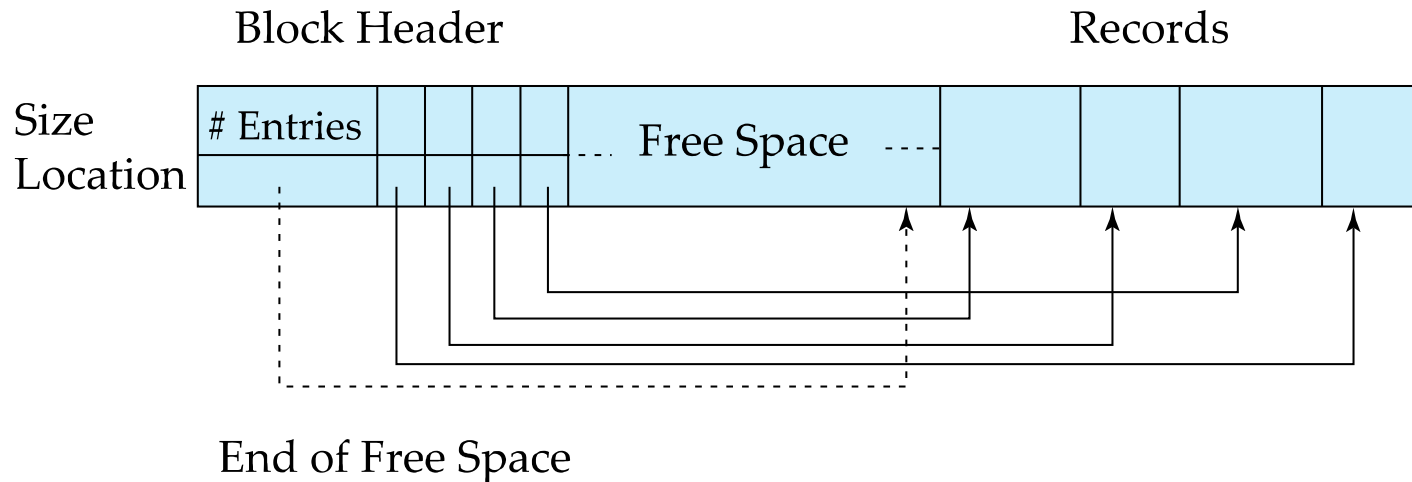
Variable-Length Records

- Variable-length records arise in database systems in several ways:
 - storage of multiple record types in a file
 - record types that allow variable lengths for one or more fields such as varchar strings
 - record types that allow repeating fields (used in some older data models)
- Attributes are stored in a fixed order. Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes.
- Null values represented efficiently using a **null bitmap**.





Variable-Length Records: Slotted Page Structure



- **Slotted page** header contains:
 - number of record entries
 - end of free space in the block
 - location and size of each record
- Records can be moved around within a page to keep them contiguous as long as the corresponding entry in the header is updated.
- Pointers to records from other structures should not point directly to a record. Instead they should point to the entry for the record in header.



Organization of Records in Files

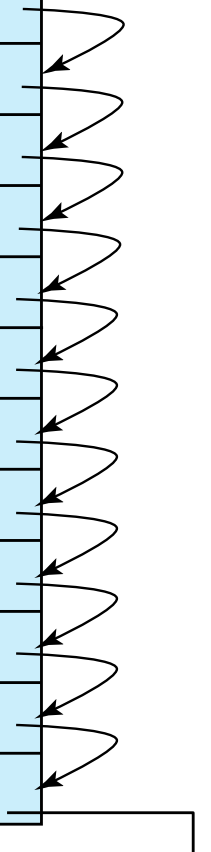
- **Heap** – a record can be placed anywhere in the file where there is space.
- **Sequential** – store records in sequential order, based on the value of the **search key** of each record.
- Records of each relation may be stored in a separate file but in a **multitable clustering file organization** the records of several different relations can be stored in the same file.
 - Motivation: store related records on the same block to minimize I/O costs.
- **Index-organized table** – records are stored using a dictionary structure such as a hash (unordered) or a B-tree (ordered). This type of organization supports fundamental operations (lookup, insert, update, delete) efficiently. In practice tables are often stored as B-trees ordered by the primary key.



Sequential File Organization

- Suitable for applications that require sequential processing of the entire file.
- The records in the file are ordered by a **search key**.

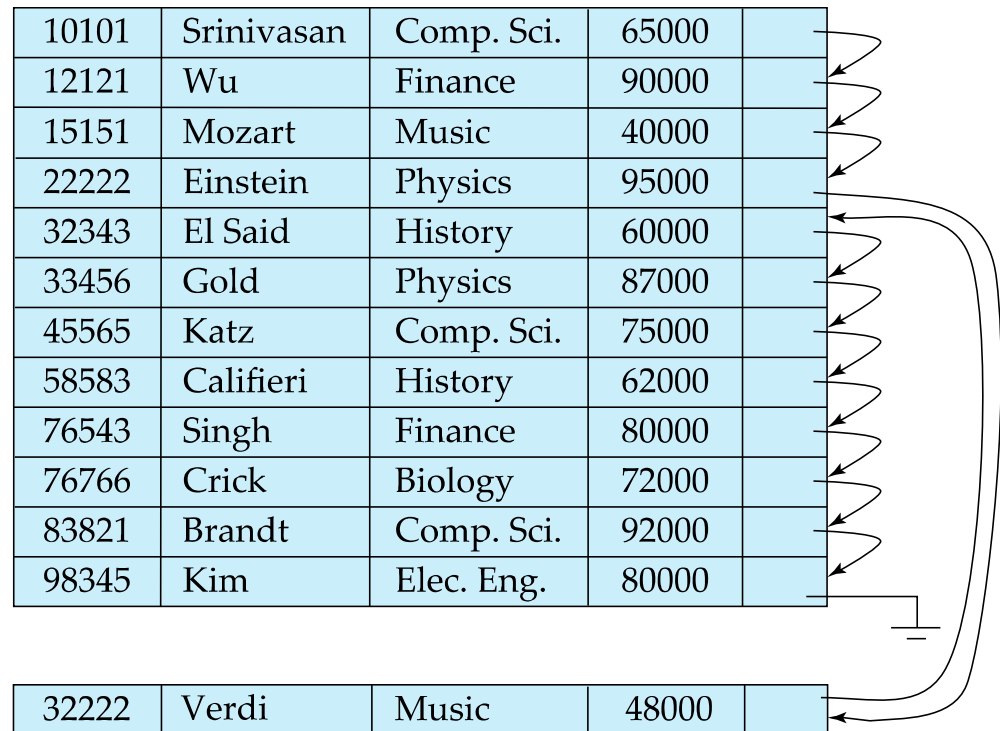
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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





Sequential File Organization (Cont.)

- Deletion – use pointer chains.
- Insertion – locate the position where the record is to be inserted
 - if there is free space insert there
 - if no free space, insert the record in an **overflow block**
 - in either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore the sequential order.





Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization.

department

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

multitable clustering
of *department* and
instructor

Comp. Sci.	Taylor	100000
45564	Katz	75000
10101	Srinivasan	65000
83821	Brandt	92000
Physics	Watson	70000
33456	Gold	87000



Multitable Clustering File Organization (cont.)

- Good for queries involving *department* ⋈ *instructor*, and for queries involving one single department and its instructors.
- Bad for queries involving only *department*.
- Results in variable size records.
- Can add pointer chains to link records of a particular relation.

Comp. Sci.	Taylor	100000	
45564	Katz	75000	
10101	Srinivasan	65000	
83821	Brandt	92000	
Physics	Watson	70000	
33456	Gold	87000	



Data Dictionary Storage

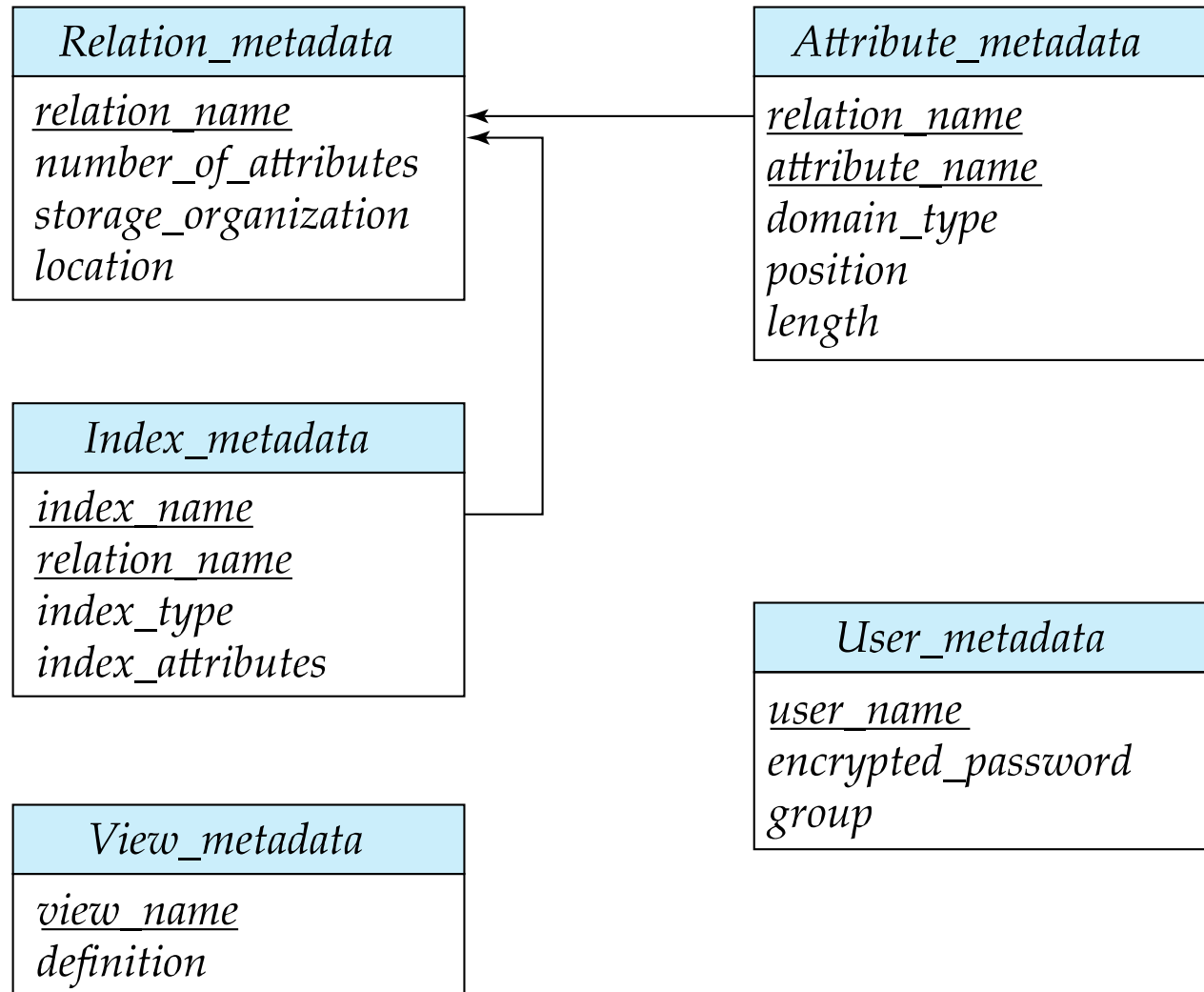
The **data dictionary** (also called **system catalog**) stores **metadata**, in other words data about data, such as:

- information about relations:
 - names of relations
 - names, types and lengths of attributes of each relation
 - names and definitions of views
 - integrity constraints
- user and accounting information, including passwords
- statistical and descriptive data
 - number of tuples in each relation
- physical file organization information
 - how the relation is stored (sequential/hash/B-tree)
 - physical location of the relation
- information about indexes



Relational Representation of Metadata

- Relational representation on disk.
- Specialized data structures designed for efficient access, in memory.





Buffer Management

- Regardless of specific file organization, a file is partitioned into fixed-length storage units called **blocks**, which are units of both storage allocation and data transfer.
- Database systems seek to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- **Buffer or Bufferpool(s)** – portion of main memory available to store copies of disk blocks.
- **Bufferpool manager** – subsystem responsible for
 - allocating buffer space in main memory
 - loading blocks from disk into the buffer pool
 - evicting blocks as needed to create space in the buffer pool
 - ▶ including writing back dirty (*i.e.*, modified) blocks to disk
 - Similar to virtual memory manager (VMM) but optimized specifically for the database (*e.g.*, in terms of eviction policy).



Buffer Replacement Policy

- Consider a nested-loop-join implementation:

```
for (i=0; i<table1.size(); ++i) {  
    for (j=0; j<table2.size(); ++j) {  
        if (table1.joinAttr == table2.joinAttr) {  
            output table1.row, table2.row  
        }  
    }  
}
```

- Assuming both tables are the same size, and an LRU policy is used, what happens if the buffer pool is only big enough for one of the tables?



Indexes and Hashes

- Queries often select a subset of records in a table:
 - **point query**: find record(s) with a specified attribute value
 - **range query**: find records with attribute values in a given range
- In most cases the query returns only a small subset of the records in a table, and we would like to locate these records efficiently.
- Simple file structures have fundamental weaknesses:
 - **heap file**: insertion is fast but a lookup requires a **scan**
 - **sequential file**: supports binary search (logarithmic time) but only on **one search key**, all other queries require a **scan**, and furthermore the use of **overflow blocks** degrades performance
- When a file is stored on a disk, even "efficient" binary search can be quite slow because each record fetched may require a random I/O.
 - What is the cost of file scan (called table scan in DB) if the table is entirely on disk?



Motivation

- An **index** (plural **indexes** or **indices**) is a data structure that speeds up lookup of records by specific search keys.
- Indices boost performance in two ways:
 1. They organize data in ways that benefit specific types of queries.

E.g., if Employee table is in a sequential file sorted by *emp#*, an index on *dept#* can speed up a query for employees in dept. #12
 2. They are usually smaller than the data files they refer to.
 - ▶ The buffer pool manager is more likely to hold them in main memory, which avoids expensive I/O operations.
 - ▶ Even when they reside on disk, they can be searched using fewer I/O operations.



Designing the Physical Schema

- Suppose that the logical schema is decided.
- First, we choose a structure for each relation such as sequential, heap, or index-organized (explained in detail in this lecture).
- Then, we add indices judiciously:
 1. Identify costly queries that would benefit from the index.
 - ▶ *E.g.*, queries with where clauses:
select * from Employee where dept_id = X
select * from Employee where salary between Y and Z
 - ▶ *E.g.*, queries with joins
select * from Employee natural join Department
 2. Choose index type: usually B⁺-tree or hash.
- **Note 1:** Adding an index can make insertions and updates slower.
- **Note 2:** Some indices may be created for you automatically by the DBMS, such as on primary and foreign keys.



Basic Concepts

- **Search key:** an attribute or set of attributes used to look up records.
- An **index** is a collection of records (called **index entries**) of the form

search-key	pointer
------------	---------

- An index looks a lot like a table, and itself requires some concrete structure for storage.
- There are two kinds of indices:
 - **ordered indices:** search keys are stored in sorted order (*e.g.*, variations on a B-tree)
 - **hash indices:** search keys are distributed (nearly) uniformly across buckets using a hash function



Ordered Indices

- **Primary index:** the index whose search key coincides with the sequential order of the file (or structure) that stores the table
 - also called **clustered index**
 - search key usually (but not necessarily) equal to the primary key
- **Secondary index:** an index whose search key specifies an order different from the sequential order of the file (or structure) that stores the table. Also known as a **non-clustered index**.
- **Index-sequential file:** ordered sequential file + a primary index. (Example: see next slide.)



Dense Index Files

- **Dense index:** one index record for every value of the search key in the file.
- *E.g.*, index on *ID* attribute of *instructor* relation.

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



Dense Index Files (Cont.)

- E.g., dense index on *dept_name*, with *instructor* file sorted on *dept_name*.

Biology		76766	Crick	Biology	72000	
Comp. Sci.		10101	Srinivasan	Comp. Sci.	65000	
Elec. Eng.		45565	Katz	Comp. Sci.	75000	
Finance		83821	Brandt	Comp. Sci.	92000	
History		98345	Kim	Elec. Eng.	80000	
Music		12121	Wu	Finance	90000	
Physics		76543	Singh	Finance	80000	
		32343	El Said	History	60000	
		58583	Califieri	History	62000	
		15151	Mozart	Music	40000	
		22222	Einstein	Physics	95000	
		33465	Gold	Physics	87000	



Sparse Index Files

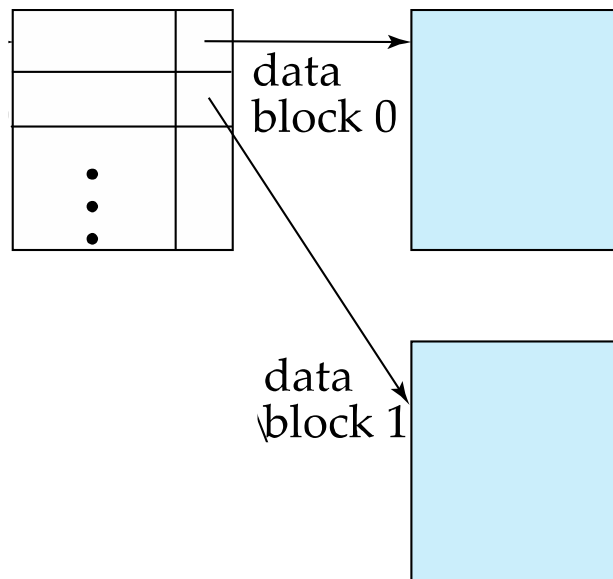
- **Sparse Index:** contains index records for only some of the values of the search key in the file. Possible only when the file is ordered by the same search key.
- To locate a record with search key value K we:
 - find the index record with largest search key value $< K$
 - scan the file forward starting at the record to which the index record points, stop when search key value becomes $> K$

10101		10101	Srinivasan	Comp. Sci.	65000	
32343		12121	Wu	Finance	90000	
76766		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	



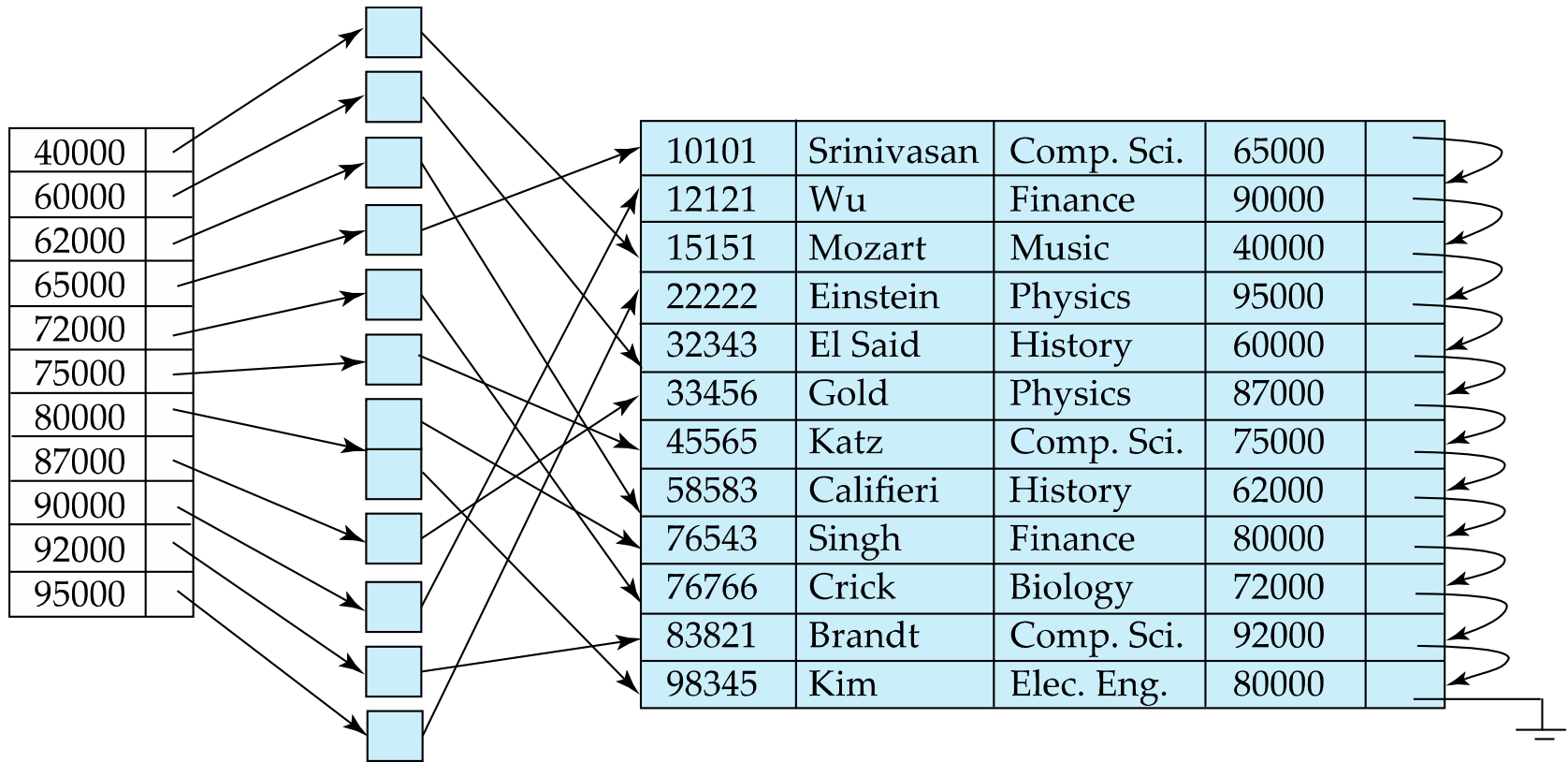
Sparse Index Files (Cont.)

- Compared to dense indices:
 - Sparse indices use less space and incur less maintenance overhead for insertions and deletions.
 - Sparse indices are generally slower for locating individual records.
- **Good tradeoff:** sparse index with an index entry for every block in file, corresponding to the lowest search key value in the block.





Secondary Indices Example



Secondary index on *salary* field of *instructor*.

- Index record comprises the search key and a set of pointers to matching file records. The pointers may be record numbers or primary key values (requiring a primary index lookup).
- **Note:** secondary indices are always dense.



Primary vs. Secondary Indices

- Indices enable **faster searches**.
- Updating indices incurs **overhead**: when a table is modified, every index on the table must be updated.
- A sequential scan using primary/clustered index is efficient as long as the sequential file has few overflow blocks, but a sequential scan using a secondary index is expensive for the following reasons:
 - each record access may fetch a new block from disk
 - a block fetch from hard disk requires about 5 to 10ms if it leads to random I/O, versus ~100ns from memory



The Need for Advanced Data Structures

- Sequential files can be used to store both indices and data.
- Sequential files enable efficient search, but suffer from the **degradation problem**:
 - searching becomes slower as file grows due to **overflow blocks**
 - therefore, the file must be **reorganized periodically**
- Solution: use advanced data structures to organize the index file and/or data file.
 - B⁺-trees – good all around, support efficient range queries
 - hashes – good for point queries only
- **Note:** hash indices are more appropriate for in-memory DBs, not supported by MySQL's InnoDB storage engine.



B⁺-Tree Index Files

B⁺-tree indices are an alternative to index-sequential files.

- Disadvantage of indexed-sequential files
 - **degradation problem:** searching becomes slower as file grows due to **overflow blocks**
 - therefore, file must be **reorganized periodically**
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with **small, local, changes**, in the face of insertions and deletions
 - reorganization of entire structure at once is not required to maintain performance
- (Minor) disadvantage of B⁺-trees:
 - computation and space overheads
- The advantages of B⁺-trees usually outweigh their disadvantages. **B⁺-trees are used extensively in practice!**

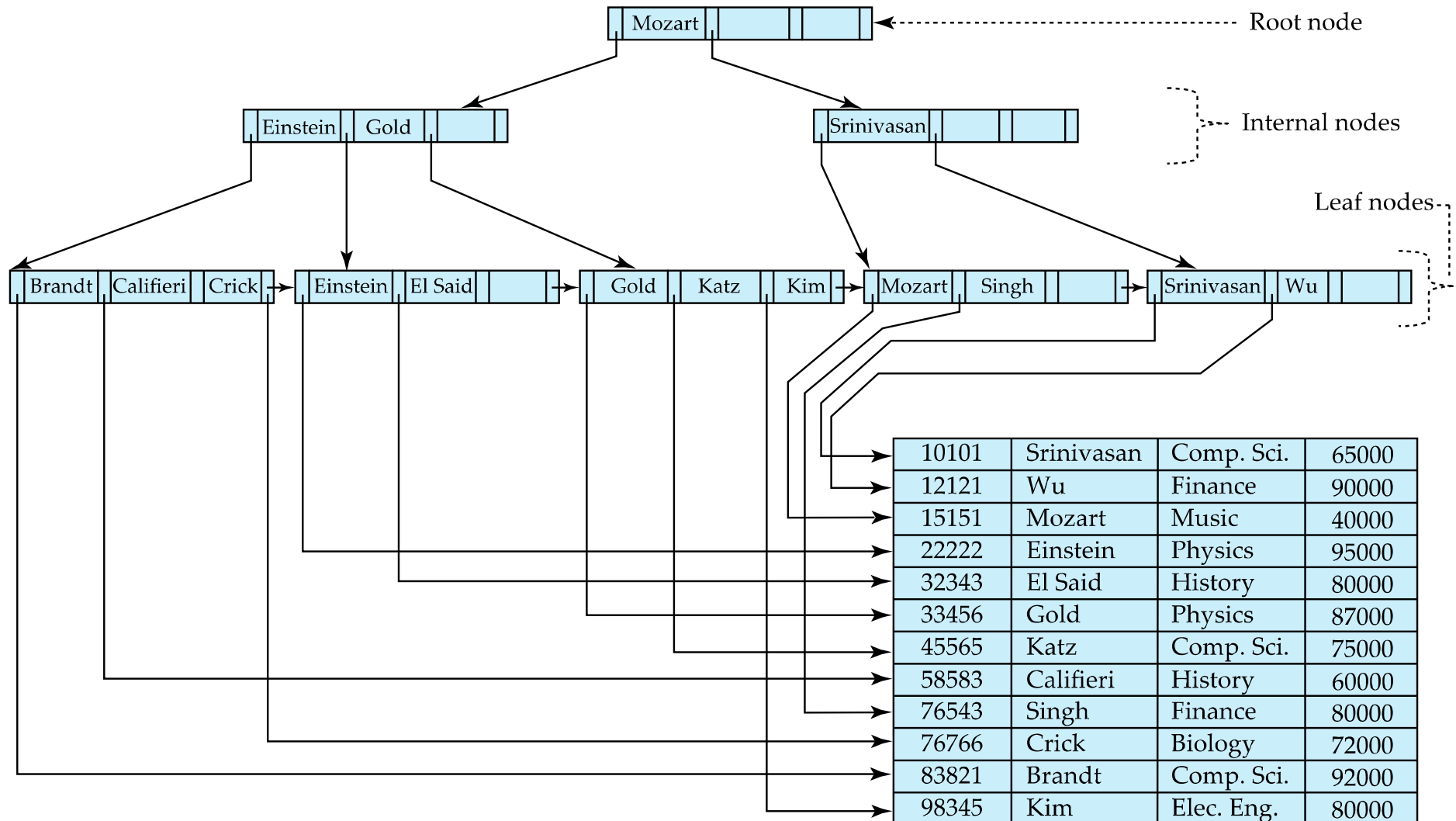


B⁺-Tree Index Files (Cont.)

- A B⁺-tree satisfies many of the useful properties of B-trees:
 - all paths from root to leaf have **equal length**
 - large branching factor, hence **shallow height**
 - each node that is not a root is at least **“half full”**
- A B⁺-tree differs from B-trees (covered in ECE250) as follows:
 - internal nodes only store **separator keys** and pointers to other nodes
 - all the data values are stored in leaf nodes, which are chained together using **sibling pointers**
 - the data values are pointers to records in a file, which can be represented as record numbers or as values of the primary key



Example of B⁺-Tree Index



(sequential file)



Observations about B⁺-Trees

- Since the inter-node connections are pointers, nodes that are close together logically (e.g., siblings) are not always close together in terms of their physical storage on disk.
 - Consequences:
 1. tree traversals may require random I/Os
(note: tree nodes should align with disk blocks)
 2. an in-order walk through the leaf nodes of a B⁺-tree can be slower than scanning a sequential file
- B⁺-trees are shallow:
 - with K key-data pairs and n child pointers per internal node, height $\leq \lceil \log_{\lceil n/2 \rceil}(K) \rceil$
 - insertions and deletions run in time proportional to height
 - internal levels easily buffered in main memory, in which case most B⁺-tree operations require only one random I/O



Indexing Strings with B⁺-trees

- Keys are often variable length strings (*e.g.*, VARCHAR in SQL)
 - possible to use a variable branching factor
 - decision to split determined by space utilization, not the number of keys
- **Prefix compression:**
 - a technique that reduces space overhead
 - keys at internal nodes can be prefixes of full key
 - ▶ retain enough characters to distinguish entries in the sub-trees separated by the key value
 - *E.g.*, “Silas” and “Silberschatz” can be separated by “Silb”
 - keys can also be compressed by sharing a common prefix



Index-Organized Tables

- In an index-organized table, the index stores the rows of the table rather than pointers to the rows.
- Advantage:
 - avoids having to traverse a pointer from the index to the file when performing a row lookup by primary key
- Disadvantage:
 - a full traversal of the index structure is required to scan the entire table, which may incur more random I/Os than scanning a sequential file
- **Note 1:** InnoDB tables are index-organized tables stored in clustered B⁺-tree indices. If you do not declare a primary key then InnoDB will create one for you and build a clustered index.
- **Note 2:** InnoDB also provides in-memory adaptive hash indexing on top of the B-tree indices to speed up lookups on frequently accessed table rows.



Hashing

- Hashing: Map large, but sparse domain to compact domain
- Hash tables may use various collision resolution policies:
 - chaining
 - open addressing
(*e.g.*, linear/quadratic probing, double hashing)
- The cost of hash table operations depends on number of entries, size of table, assumptions on randomness, *etc.*
 - constant time operations on expectation
(but only if hash table is large enough!)
 - linear time operations in the worst case



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.



Example of Hash File Organization

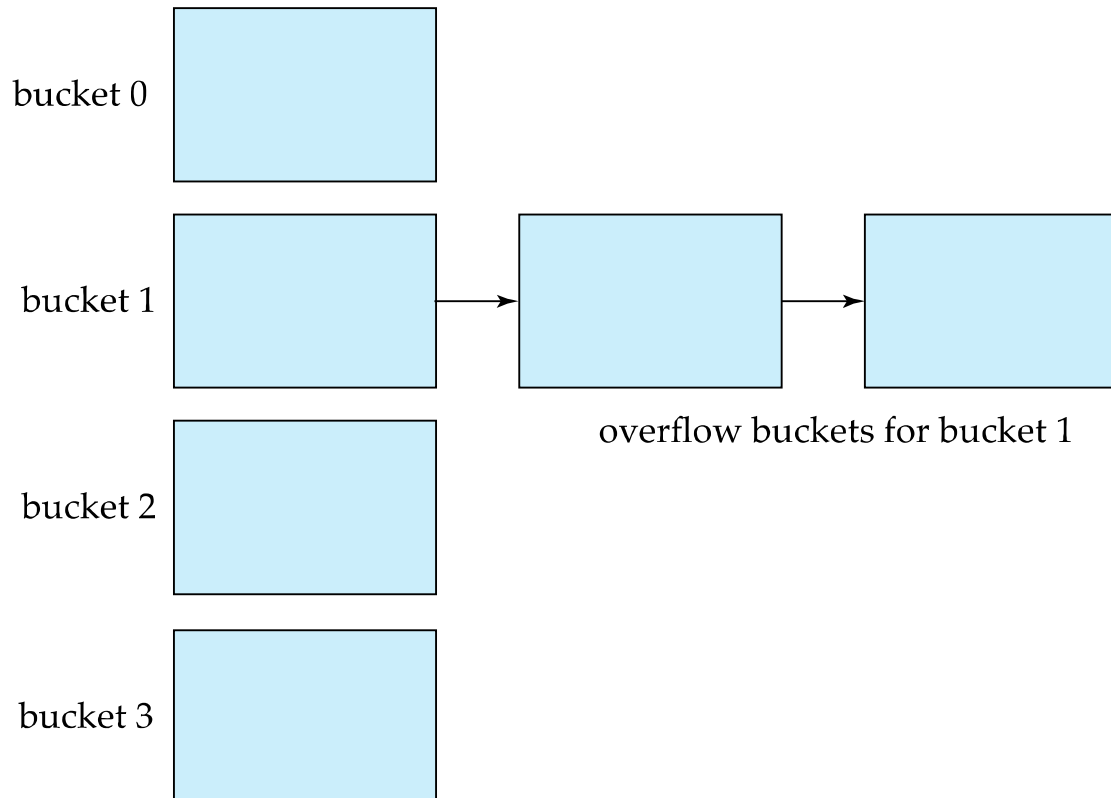
- A hash table on disk must be designed differently from a hash table in main memory.
- Entries stored in **buckets**, each containing multiple entries.
- Buckets are aligned with blocks on disk.
- In this simplified example the hash function returns the sum of the binary representations of the characters in the key modulo the number of buckets
 - *E.g.*, $h(\text{"Music"}) = 1$ $h(\text{"History"}) = 2$
 $h(\text{"Physics"}) = 3$ $h(\text{"Elec. Eng."}) = 3$

Note: Physics and Elec. Eng. collide under h !



Collision Resolution

- A bucket accommodates a limited number of collisions.
- **Overflow buckets** provide additional collision resolution, similar to chaining technique used in in-memory hashing.





Example of Hash Index

bucket 0

76766	

bucket 1

45565	
76543	

bucket 2

22222	

bucket 3

10101	

bucket 4

bucket 5

15151	
33456	

58583	
98345	

bucket 6

83821	

bucket 7

12121	
32343	

76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000
12121	Wu	Finance	90000
76543	Singh	Finance	80000
32343	El Said	History	60000
58583	Califieri	History	62000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
33465	Gold	Physics	87000

hash index on *instructor ID*



Deficiencies of Static Hashing

- In **static hashing** a hash function h maps search key values to a fixed set of bucket addresses.
- Real world databases grow or shrink with time:
 - too few buckets □ performance degrades due to bucket overflow
 - too many buckets □ wasted space
- Naive solution: periodic reorganization of the file with a new hash function.
 - expensive, disrupts ordinary index operations
- Better solution: allow the number of buckets to be modified dynamically.

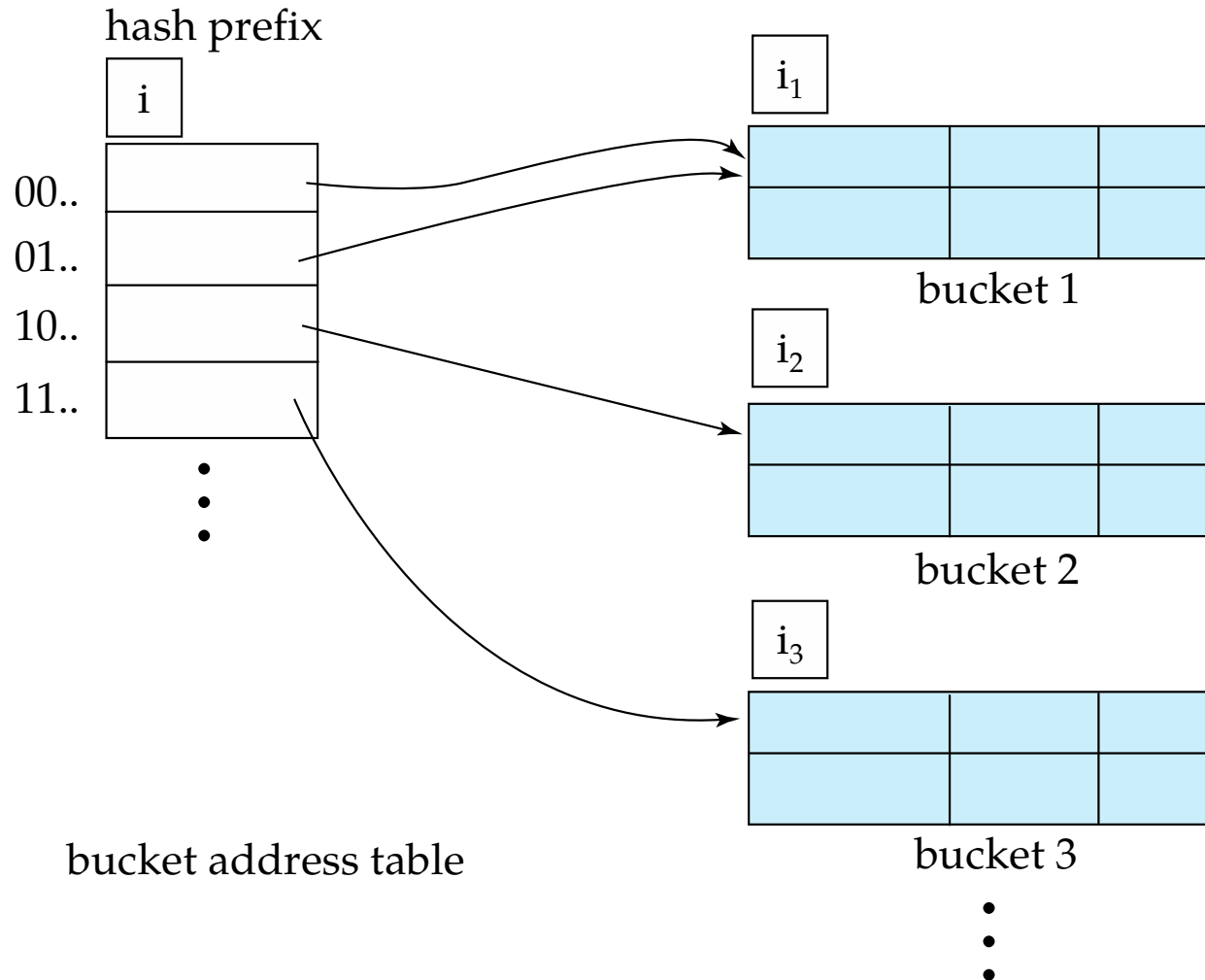


Dynamic Hashing

- **Dynamic hashing** allows the hash function to be modified dynamically.
- **Extendable hashing:** one form of dynamic hashing.
 - Hash function generates values over a large range, typically b -bit integers, with $b = 32$.
 - At any time use only a prefix of the hash function to index into a table of bucket addresses.
 - Let the length of the prefix be i bits, $0 \leq i \leq 32$. Initially $i = 0$.
 - Value of i grows and shrinks as the size of the database grows and shrinks.
 - Buckets located using **bucket address table** of size $= 2^i$.
 - Buckets are split and coalesced dynamically, and so their actual number can be $< 2^i$.
 - Multiple entries in the bucket address table may point to the same bucket.



General Extendable Hash Structure



in this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$
(see next slide for details)



Use of Extendable Hash Structure

- Each bucket j stores a value i_j . All the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j :
 1. Compute $h(K_j) = X$.
 2. Use the first i high order bits of X as an offset into the bucket address table, and follow the pointer to appropriate bucket.
- To insert a record with search-key value K_j :
 - Follow same procedure as look-up and locate the bucket, say j .
 - If there is room in the bucket j insert record in the bucket.
 - Else the bucket must be split and insertion re-attempted. May need to create an overflow bucket. (See next slide.)



Insertion in Extendable Hash Structure (Cont)

To split a bucket j when inserting record with search-key value K_j :

- If $i > i_j$ (more than one pointer to bucket j):
 - Allocate a new bucket z , and set $i_j = i_z = (i_j + 1)$.
 - Update the second half of the bucket address table entries originally pointing to j , to point to z .
 - Remove each record in bucket j and reinsert (in j or z).
 - Re-compute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full).
- If $i = i_j$ (only one pointer to bucket j):
 - If i reaches some limit b , or too many splits have happened in this insertion, create an overflow bucket.
 - Else:
 - ▶ Increment i and double the size of the bucket address table.
 - ▶ Replace each entry in the table by two entries that point to the same bucket.
 - ▶ Re-compute new bucket address table entry for K_j
Now $i > i_j$ so use the first case above.



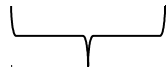
Deletion in Extendable Hash Structure

- To delete a key value:
 - Locate it in its bucket and remove it.
 - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
 - If possible, coalesce a bucket with a “*buddy*” bucket having the same value of i_j and the same hash prefix of length $i_j - 1$.
 - Decreasing bucket address table size is also possible.
 - ▶ Note: Expensive operation, should be done only if the number of buckets becomes much smaller than the size of the bucket address table.



Use of Extendable Hash Structure: Example

<i>dept_name</i>	$h(\text{dept_name})$
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

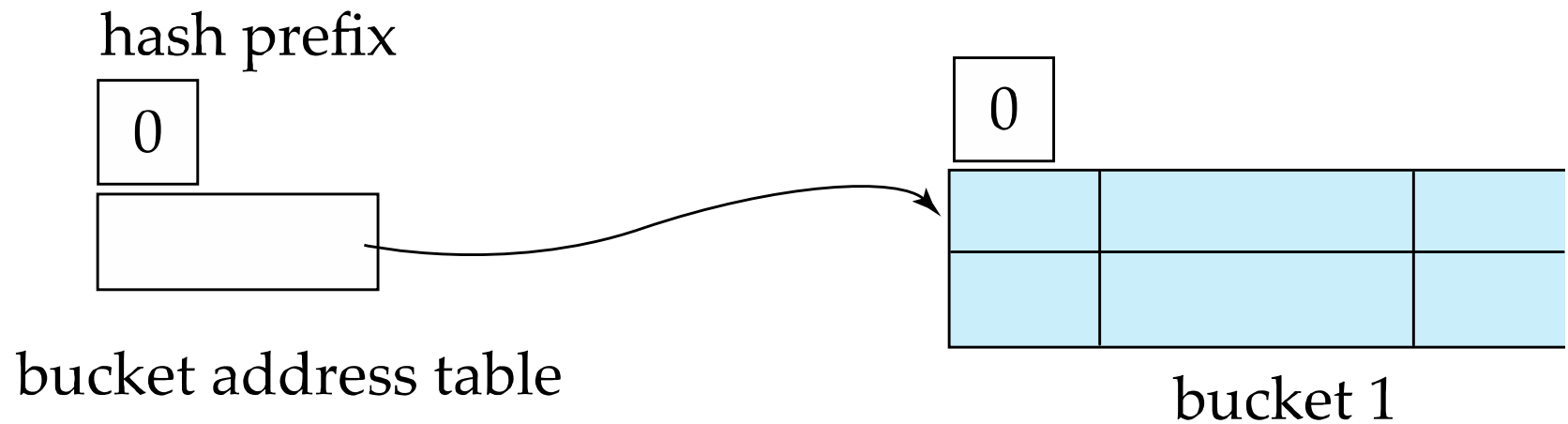


initially using only these bits, starting with most significant bit (MSB)



Example (Cont.)

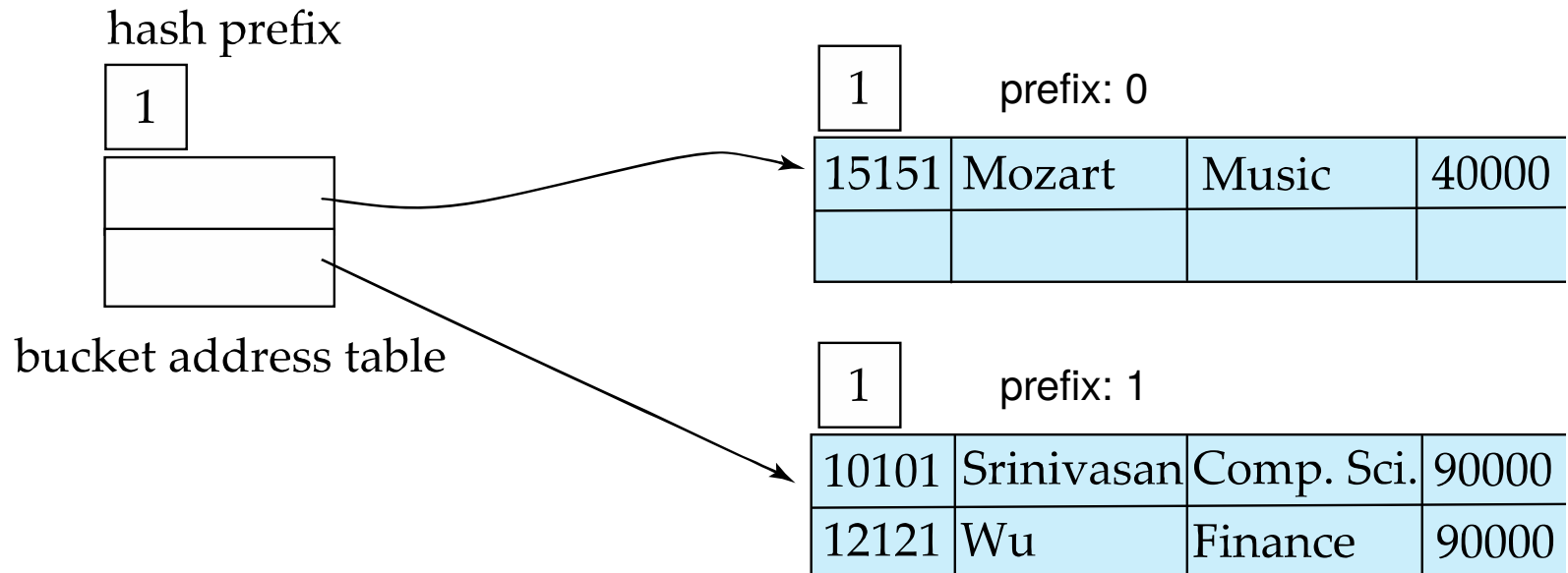
- Initial hash structure. Bucket size = 2 records.





Example (Cont.)

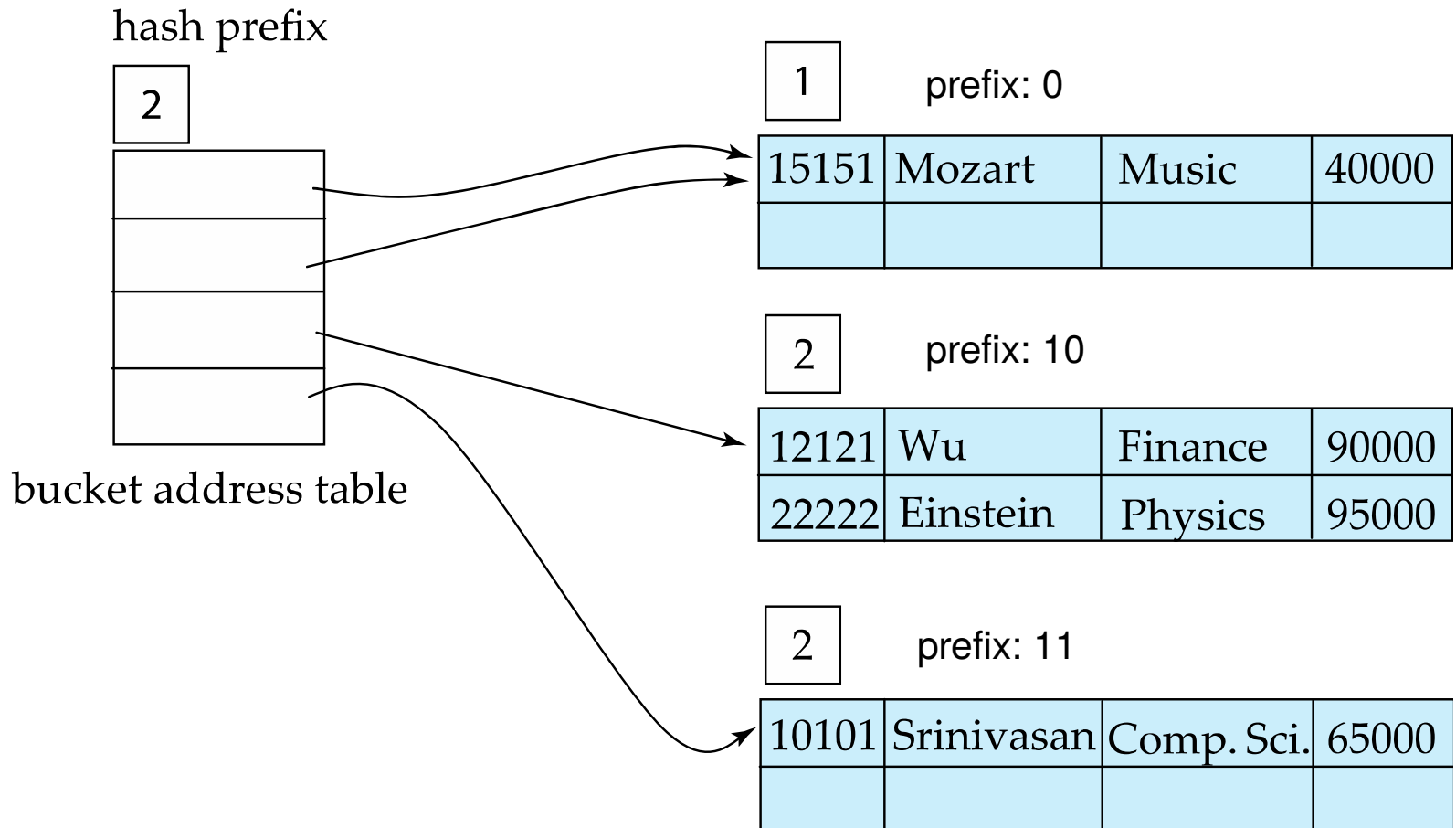
- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records.





Example (Cont.)

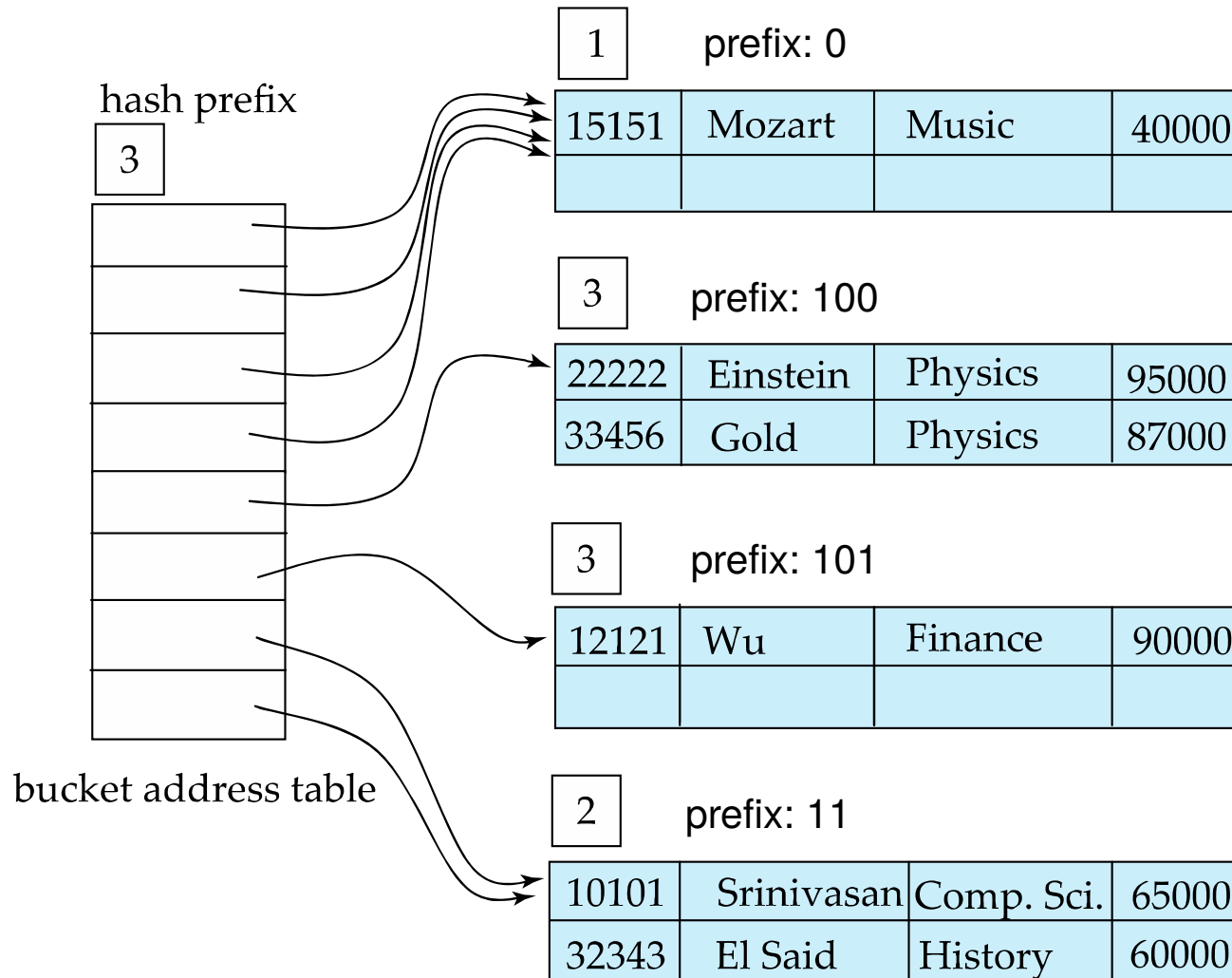
- Hash structure after insertion of Einstein record.





Example (Cont.)

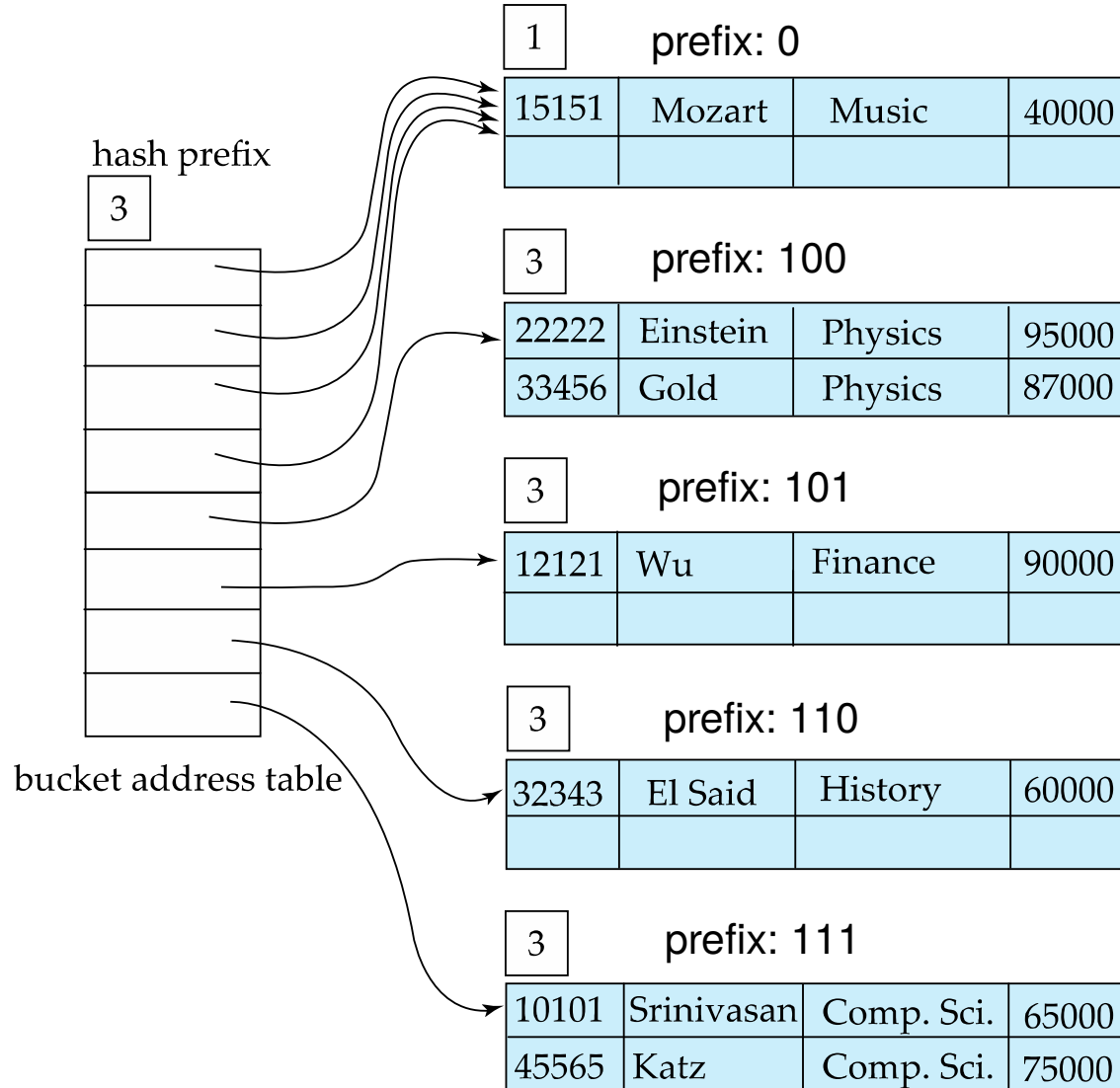
- Hash structure after insertion of Gold and El Said records.





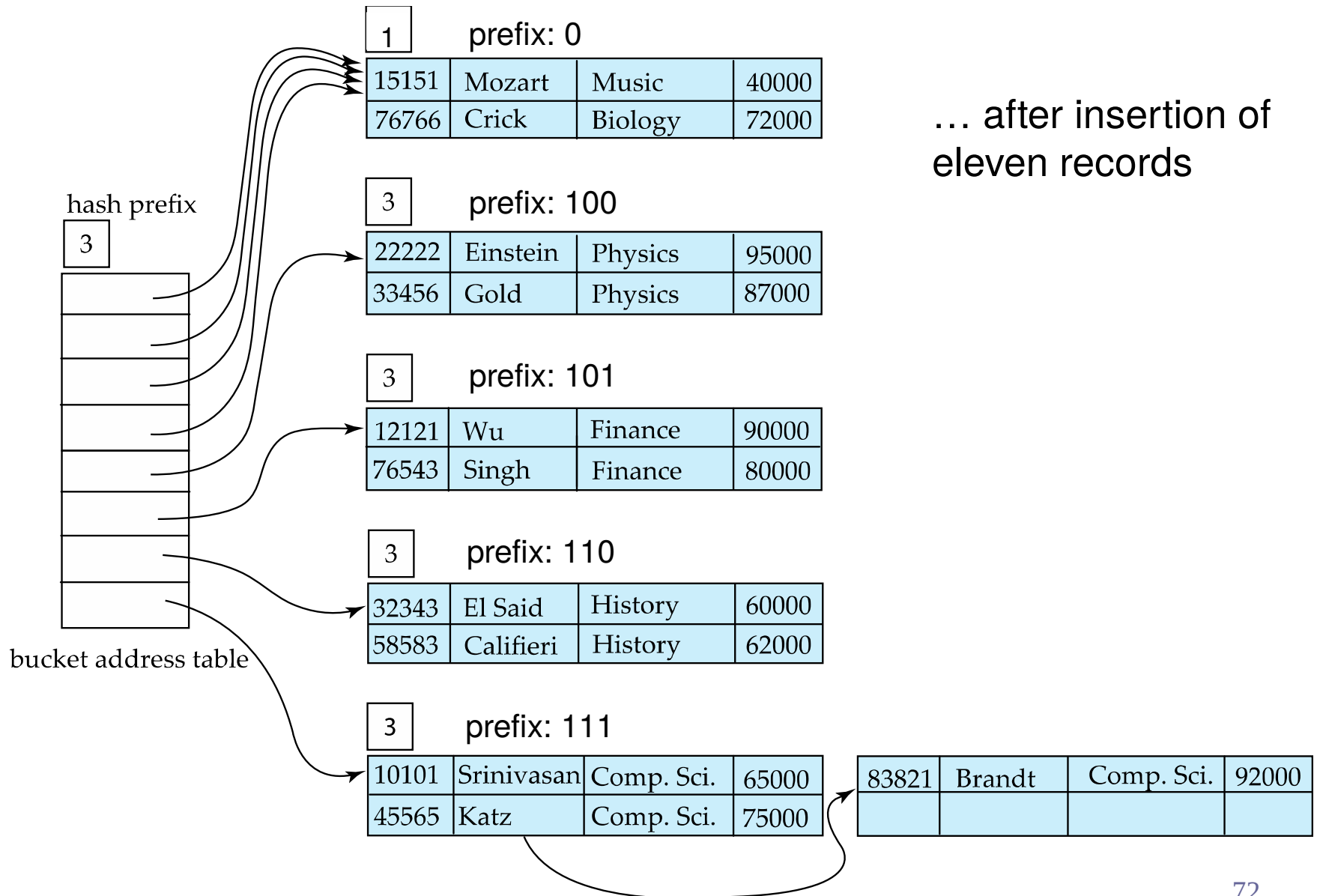
Example (Cont.)

- Hash structure after insertion of Katz record.





Example (Cont.)





Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
 - performance does not degrade as records are inserted
 - minimal space overhead for the bucket address table

- Disadvantages of extendable hashing:
 - additional level of indirection to find desired record
 - bucket address table may itself become very big
 - ▶ solution: store it using a B⁺-tree structure!
 - changing the size of bucket address table can be expensive



Multiple-Key Access

- Multiple indices may be useful for certain types of queries.
- Example:

select *ID*

from *instructor*

where *dept_name* = "Finance" **and** *salary* = 80000

- Possible strategies for processing query using indices on single attributes:
 1. Use index on *dept_name* to find instructors with department name Finance; test *salary* = 80000
 2. Use index on *salary* to find instructors with a salary of \$80000; test *dept_name* = "Finance".
 3. Use *dept_name* index to find pointers to all records pertaining to the "Finance" department.
Similarly use index on *salary*.
Take intersection of both sets of pointers obtained.



Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute.
 - *E.g., (dept_name, salary)*
- Composite search keys are compared using **lexicographic ordering**: $(a_1, a_2) < (b_1, b_2)$ if
 - $a_1 < b_1$, or
 - $a_1 = b_1$ and $a_2 < b_2$



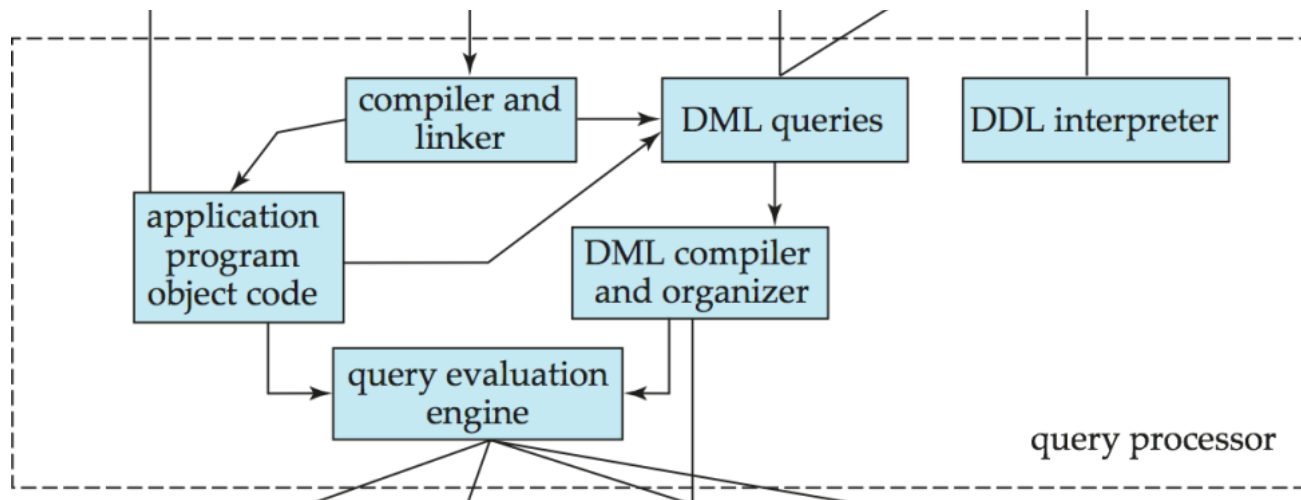
Indices on Multiple Attributes

- Suppose we have an index on the composite search key (*dept_name*, *salary*).
- With the **where** clause
 where *dept_name* = "Finance" **and** *salary* = 80000
the index on (*dept_name*, *salary*) can be used to fetch only records that satisfy both conditions.
 - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the two conditions.
- Can also efficiently handle
 where *dept_name* = "Finance" **and** *salary* < 80000
- But cannot efficiently handle
 where *dept_name* < "Finance" **and** *salary* = 80000
 - The index can be used to find records that satisfy the first condition but do not always satisfy the second condition.



Query Processing

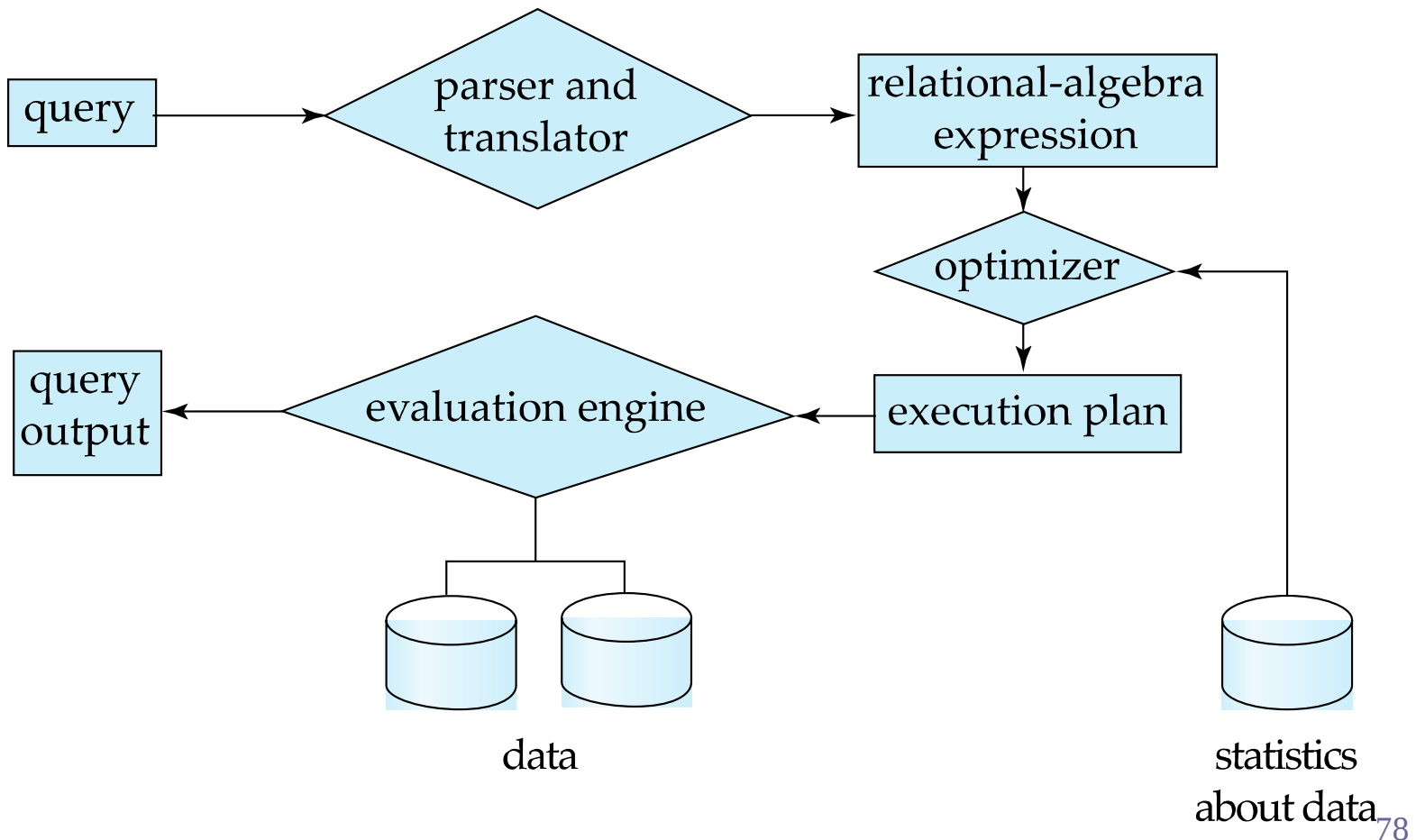
- One of the fundamental components of a DBMS is the **query processor**, which includes a **query evaluation engine**.
- The query processor translates every SQL query submitted by an application to a concrete **evaluation plan**. The evaluation engine then executes that plan by interacting with the **storage manager** layer.
- The storage manager is responsible for fetching rows from tables, inserting and updating rows, enforcing integrity constraints, as well as performing concurrency control and managing transactions.





Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Query Optimization and Evaluation

- One relational algebra expression may have many equivalent expressions.
 - Example 1: $(instructor \bowtie teaches) \bowtie course$ is equivalent to $instructor \bowtie (teaches \bowtie course)$
 - Example 2: $\sigma_{salary < 75000}(\Pi_{salary}(instructor))$ is equivalent to $\Pi_{salary}(\sigma_{salary < 75000}(instructor))$
- Furthermore, each relational algebra operation can be evaluated using one of several different algorithms. As a result, there are many ways to evaluate a given relational algebra expression.
- An **evaluation plan** is an annotated expression specifying the detailed evaluation strategy for a given query.
 - Example 1: join instructor with teaches first, then join with course.
 - Example 2: use an index on *salary* to find instructors with salary < 75000



Query Optimization and Evaluation

- Optimizing a query entails reasoning about the following:
 - equivalent relational algebra expressions for the query
 - possible evaluation plans for each candidate RA expression
 - cost of each candidate evaluation plan
- Assuming that we can settle on a meaningful definition of “cost”, optimizing a query entails finding the equivalent RA expression and evaluation plan that minimize that cost.
- Cost is **estimated** in practice using statistical information recorded in the system catalog. Examples of statistics include:
 - # of tuples in a table, # of entries in an index
 - size of each tuple or index entry, height of a B⁺-tree index
 - **cardinality** of an attribute: # distinct attribute values stored
 - **selectivity** of an index:
cardinality of indexed attribute(s) / total # of index entries



Cost Measures

- Cost measures attempt to capture the time needed to process a query. The textbook generally considers the worst case cost, which occurs when memory is scarce and hence tables and indexes are in secondary storage (one disk). In this case the cost of processing a query is dominated by I/O operations.
- For simplicity the book considers only the cost of reading data from disk, and ignores disk writes as well as CPU and network.
- The cost measure incorporates two quantities:
 - the total **number of block transfers** from disk
(t_T – time to transfer one block)
 - the total **number of seeks**
(t_S – time for one seek)
- Therefore, the total cost for b block transfers plus s seeks is
$$\mathbf{b \cdot t_T + s \cdot t_S}$$



Selection Using Table Scan

- Goal: find records matching a selection predicate P over a relation r .
- The number of tuples returned is:
 - at most one if P tests for equality on a superkey
 - possibly many if P tests for inequality, or equality on a search key that is not a superkey
- **Algorithm A1: linear search** (*i.e.*, scan) over file
 - scan file blocks contiguously and test all records to see whether they satisfy the predicate P
 - generally slow but works for any possible predicate P and any possible ordering of records in the file
- **Case 1:** assuming relation r occupies b_r contiguous blocks, cost is:
$$b_r \cdot t_T + t_s$$
- **Case 2:** assuming P tests for equality on a superkey, the cost is:
$$(b_r / 2) \cdot t_T + t_s \quad (\text{on avg., assuming a matching record is found})$$
$$b_r \cdot t_T + t_s \quad (\text{assuming no matching record is found})$$



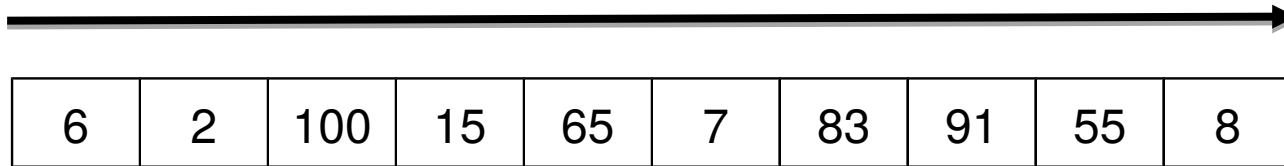
Algorithm A1 Illustrations

■ Case 1: any predicate P

- always scans entire file, cost = $b_r \cdot t_T + t_s$

one seek required only
to access first block

linear search from beginning to end,
requires b_r block transfers



file with b_r contiguous blocks

■ Case 2: P tests for equality on superkey

- scan half of the file on average if record found
cost = $(b_r / 2) \cdot t_T + t_s$
- scan entire file if record not found
cost = $b_r \cdot t_T + t_s$



Selections Using Index

- **Algorithms A2 and A3:** **index scan** using **primary/clustered** B⁺-tree index. (Table is stored separately in a sequential file.)
- Notation: in this lecture h_i denotes the number or levels in the B⁺-tree index, which equals the height of the B⁺-tree plus one.
- **A2:** P tests for equality, search key is a superkey. One seek and transfer required for each level of the B⁺-tree, plus one more to retrieve a matching record from the sequential file. The cost is:

$$(h_i + 1) \cdot (t_T + t_S)$$

- **A3:** P tests for equality, search key is not a superkey. Index scan may return multiple records, in contrast to A2. If b denotes the number of blocks in the sequential file containing matching records then the cost is:

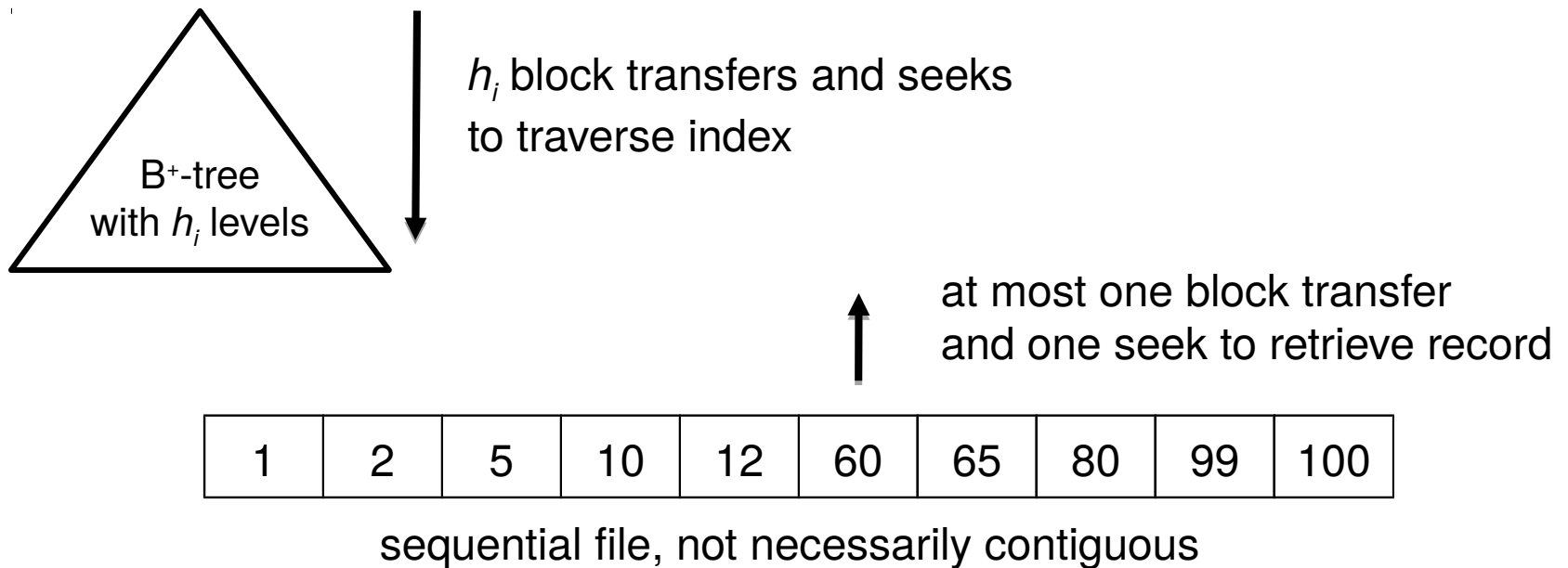
$$h_i \cdot (t_T + t_S) + t_S + t_T \cdot b$$

(Assumes blocks of sequential file are stored contiguously.)



Algorithm A2 Illustration

- **Algorithm A2: index scan using primary B⁺-tree index, P tests for equality on superkey**

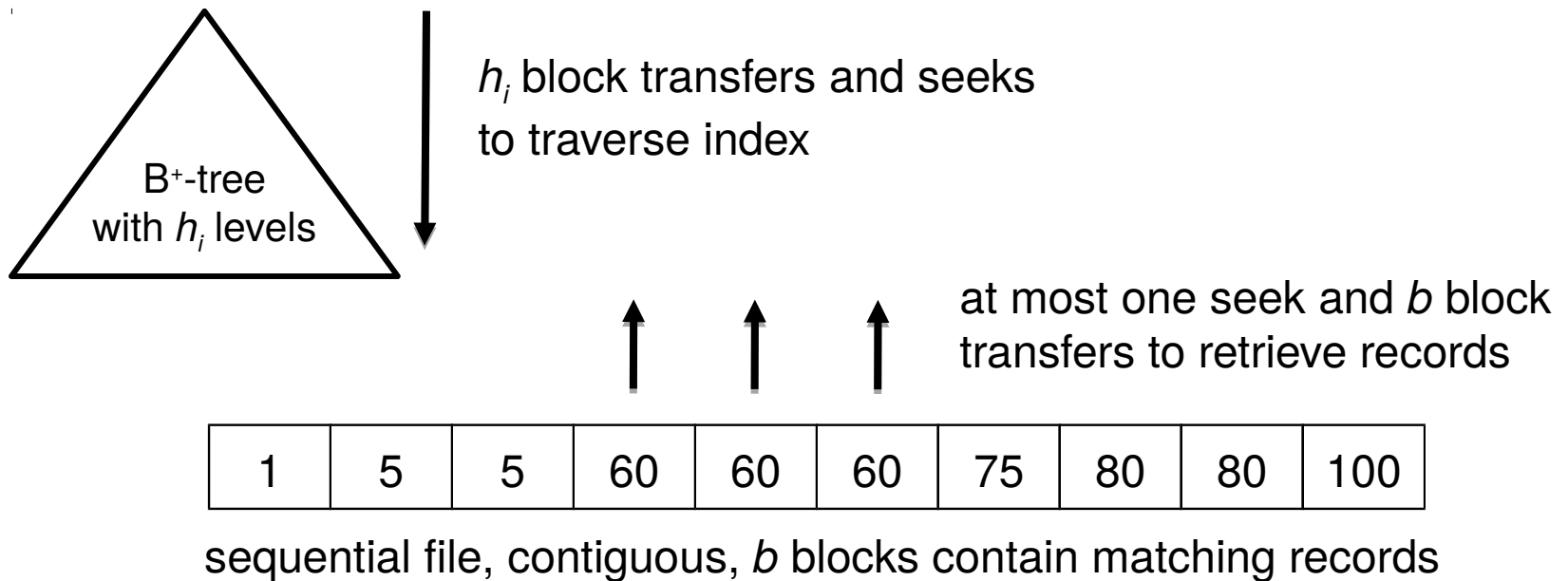


$$\text{cost} = (h_i + 1) \cdot (t_T + t_S)$$



Algorithm A3 Illustration

- Algorithm A3: index scan using primary B⁺-tree index, P tests for equality on non-superkey



$$\text{cost} = h_i \cdot (t_T + t_S) + t_S + t_T \cdot b$$



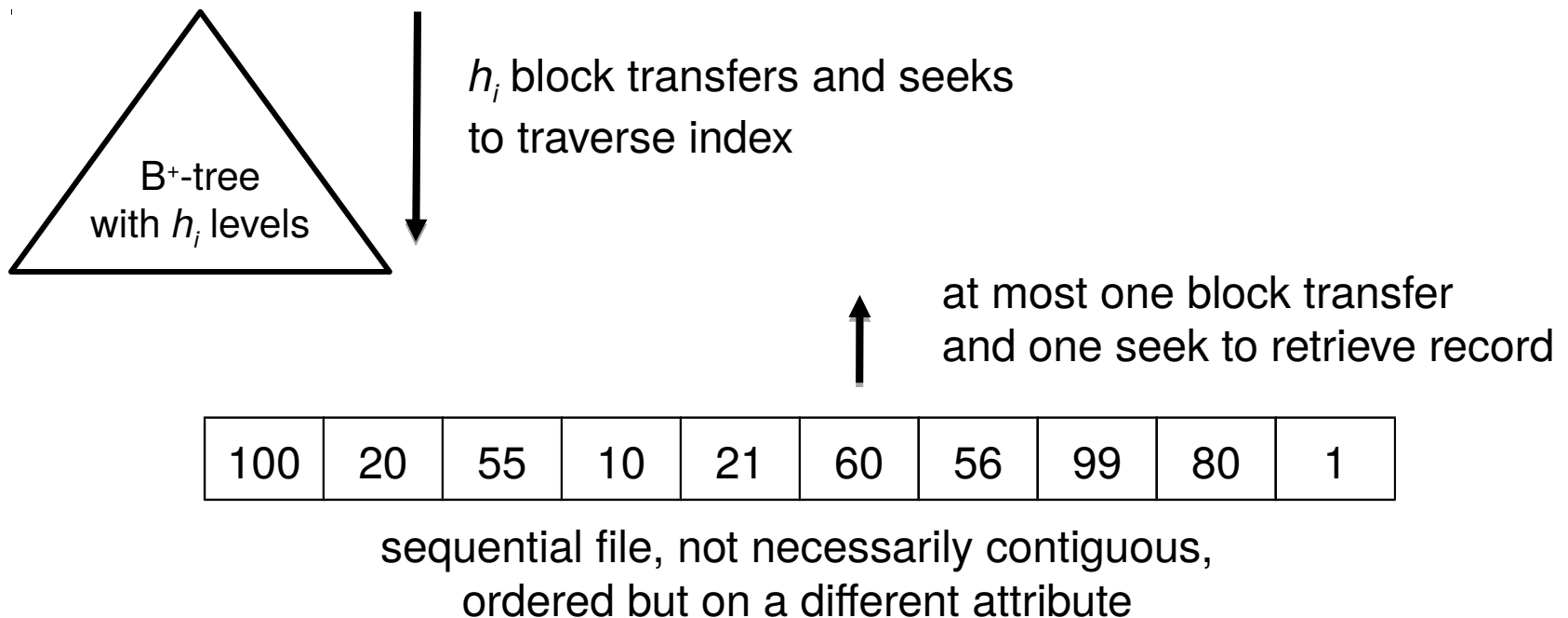
Selections Using Index

- **Algorithm A4: index scan** using **secondary** B⁺-tree index. (Table is stored separately in a sequential file.)
- **Case 1:** P tests for equality, search key is a superkey.
 - Analogous to Algorithm A2. Cost:
$$(h_i + 1) \cdot (t_T + t_S)$$
- **Case 2:** P tests for equality, search key is not a superkey.
 - First traverse B⁺-tree as in Algorithm A3, then fetch matching records, which may not be stored in contiguous blocks because the index is secondary. If there are n matching records the cost is:
$$(h_i + n) \cdot (t_T + t_S)$$
 - Note: in the worst case $n = \#$ rows in table.



Algorithm A4 Illustration

- **Algorithm A4 case 1: index scan using secondary B⁺-tree index, *P* tests for equality on superkey**



$$\text{cost} = (h_i + 1) \cdot (t_T + t_S)$$

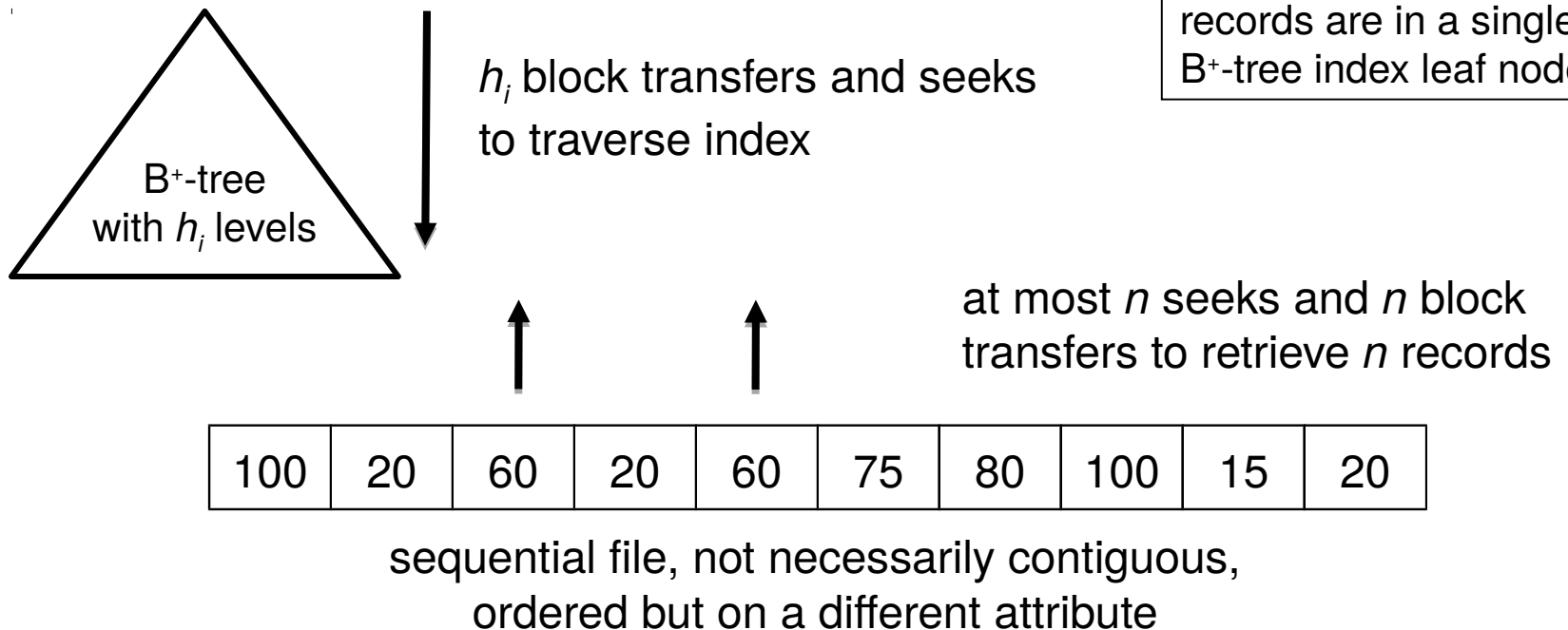


Algorithm A4 Illustration

- Algorithm A4 case 2: index scan using secondary B⁺-tree index, *P* tests for equality on non-superkey

Assumption:

Pointers to matching records are in a single B⁺-tree index leaf node.



$$\text{cost} = (h_i + n) \cdot (t_T + t_S)$$



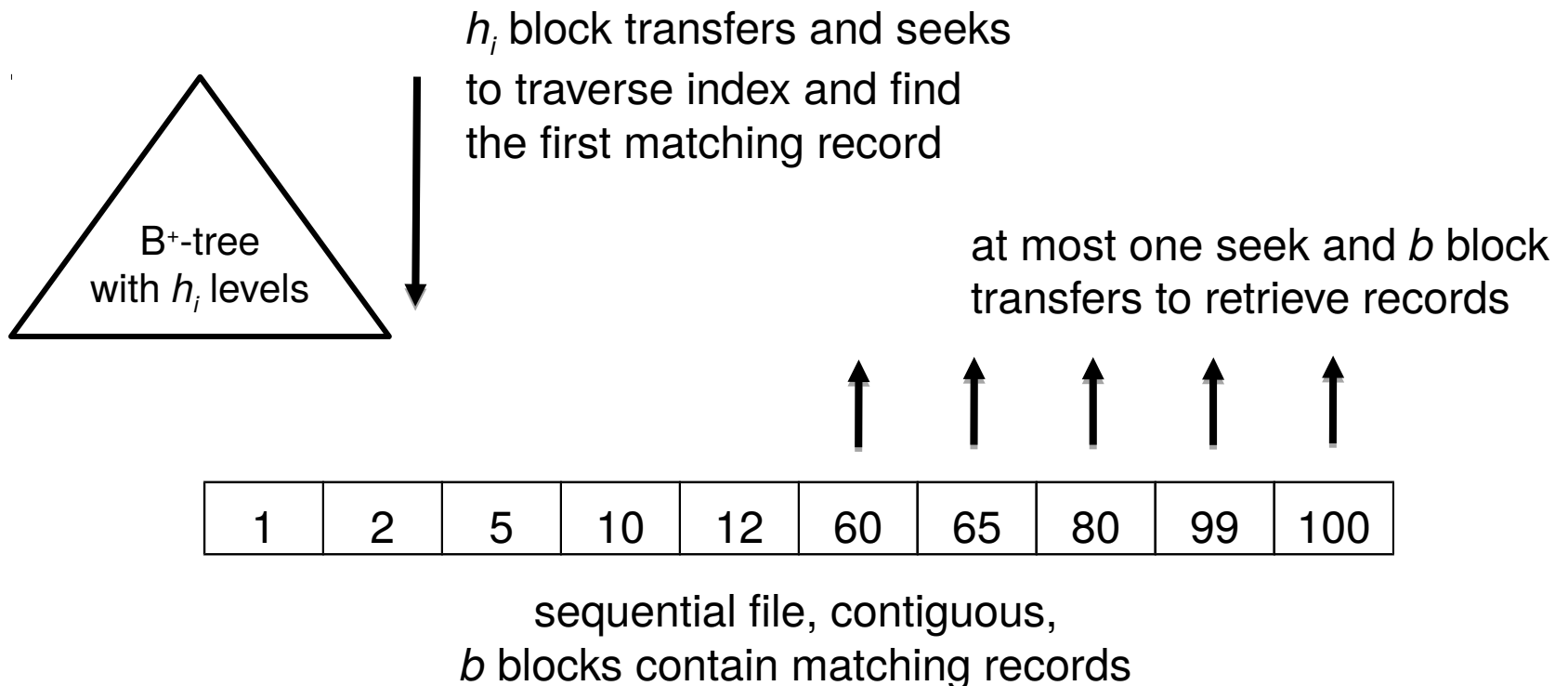
Selections Using Index

- **Algorithms A5 and A6: index scan** using B⁺-tree index, predicate P tests for inequality (e.g., $\sigma_{A \leq V}(r)$ or $\sigma_{A \geq V}(r)$).
- **A5: primary/clustered B⁺-tree index**
 - Procedure and cost identical to Algorithm A3 (testing equality, search key is not a superkey).
 - Example: To compute $\sigma_{A \geq V}(r)$ first lookup $A = V$ using the index and then scan b contiguous blocks from V to the end of the file.
- **A6: secondary B⁺-tree index**
 - Procedure and cost similar to Algorithm A4, Case 2 (testing equality, search key is not a superkey).
 - Example: To compute $\sigma_{A \geq V}(r)$ first lookup $A = V$ using the index, then scan k leaf nodes to find pointers to records matching $A > V$, finally retrieve records n from table. The cost is:
$$(h_i + k + n) \cdot (t_T + t_S)$$
- Food for thought: how do the cost estimates change if an index-organized table is used (i.e., B⁺-tree only with no sequential file)?



Algorithm A5 Illustration

- **Algorithm A5: index scan using primary B⁺-tree index, P tests for inequality (e.g., $\sigma_{A \geq V}(r)$)**

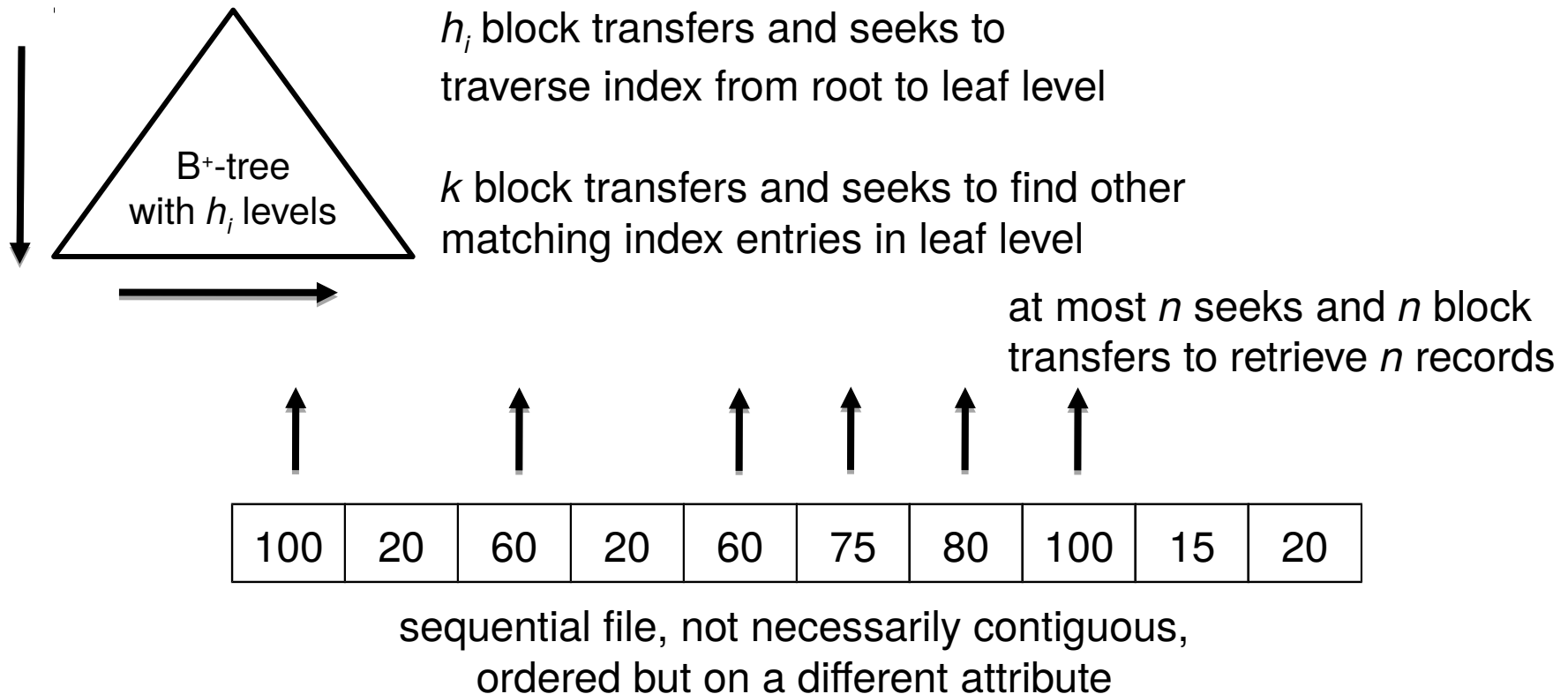


$$\text{cost} = h_i \cdot (t_T + t_S) + t_S + t_T \cdot b$$



Algorithm A6 Illustration

- **Algorithm A6: index scan using secondary B⁺-tree index, P tests for inequality (e.g., $\sigma_{A \geq v}(r)$)**



$$\text{cost} = (h_i + k + n) \cdot (t_r + t_s)$$



Join Operations

- A join combines data from a pair of **inputs**, which can be tables or intermediate query results.
- We will focus on the **equi-join**, which is a theta-join where the predicate theta tests attribute values for equality.
 - e.g., $r \bowtie_{r.A = s.B} s$
- A **natural join** is a special case of an equi-join.
- Several different algorithms can be used to evaluate joins. We will focus on variations of the **nested loops join**, which iterates over the rows of the two inputs.
- Examples of cost estimates will use instances of *student* and *takes* from the university schema with the following parameters:
 - number of records of *student*: 5,000 *takes*: 10,000
 - number of blocks of *student*: 100 *takes*: 400
- Notation: n_x and b_x denote the # of records and blocks (respectively) in relation x .



(Ordinary) Nested Loops Join

- High-level idea: to compute $r \bowtie_{\theta} s$ using a pair of for loops:

for each tuple t_r **in** r **do begin**

for each tuple t_s **in** s **do begin**

 test pair (t_r, t_s) to see if it satisfies the join condition θ

 if it does, add $t_r \bullet t_s$ (concatenation of tuples) to the result

end

end

- Relations r and s are called the **outer relation** and **inner relation** of the join, respectively.
- The algorithm is very **general**: it does not require any indexes and it can be used with any condition θ .
- The algorithm is also **expensive** since it examines every pair of tuples in the two relations.



(Ordinary) Nested Loops Join

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is:

$$(n_r \cdot b_s + b_r) \cdot t_T + (n_r + b_r) \cdot t_S$$

- Intuition: each block in the inner relation s is read once for each record in the outer relation r .
- The cost estimate in our running example is as follows:
 - Case 1: *student* as the outer relation:
 - ▶ $5000 \cdot 400 + 100 = 2,000,100$ block transfers
 - ▶ $5000 + 100 = 5,100$ seeks
 - Case 2: *takes* as the outer relation
 - ▶ $10000 \cdot 100 + 400 = 1,000,400$ block transfers
 - ▶ $10000 + 400 = 10,400$ seeks
- If the smaller relation fits entirely in memory, use that as the inner relation. This reduces the cost down to $(b_r + b_s) \cdot t_T + 2 \cdot t_S$. For example, if *student* fits in memory then the cost estimate is only 500 block transfers + 2 seeks.



Block Nested Loops Join

- Insight from last slide: the cost of the nested loops join can be reduced by taking advantage of main memory.
- The **block nested loops join** is a variant of the nested loops join in which every block of the inner relation is paired with every block of the outer relation.

```
for each block  $B_r$  of  $r$  do begin
    for each block  $B_s$  of  $s$  do begin
        for each tuple  $t_r$  in  $B_r$  do begin
            for each tuple  $t_s$  in  $B_s$  do begin
                test  $(t_r, t_s)$  to see if it satisfies join
                condition  $\theta$ 
                if it does, add  $t_r \cdot t_s$  to the result
            end
        end
    end
end
```




Block Nested Loops Join

- The worst case occurs when only one block of each relation is held in memory at a given time. The cost is:

$$(b_r \cdot b_s + b_r) \cdot t_T + (2 \cdot b_r) \cdot t_s$$

- Intuition: each block in the inner relation s is read once for each block in the outer relation r .
- If the inner relation fits entirely in memory then the cost is the same as for the ordinary nested loops join:

$$(b_r + b_s) \cdot t_T + 2 \cdot t_s$$

- The algorithm can be optimized for a specific memory budget M (number of blocks that can be held in memory simultaneously).
- Approach: Scan the outer relation $M - 2$ blocks at a time, and use the remaining two blocks to buffer the inner relation and the output. The cost is:

$$(\lceil b_r / (M - 2) \rceil \cdot b_s + b_r) \cdot t_T + (2 \lceil b_r / (M - 2) \rceil) \cdot t_s$$



Indexed Nested Loops Join

- If an index is available on the join attribute(s) of the inner relation, looking up tuples using the index can be more efficient than scanning the inner relation. (Index lookup replaces inner for loop!)

for each tuple t_r **in** r **do begin**

 use index to compute the set S_r of tuples t_s in s such that

 the pair (t_r, t_s) satisfies the join condition θ

 for every pair (t_r, t_s) in S_r , add $t_r \cdot t_s$ to the result

end

- Assuming that one block of r is buffered in memory at a given time, and letting c denote the average cost of traversing the index and fetching all the matching s tuples for one tuple t_r of r , the cost is:

$$b_r (t_r + t_s) + n_r \cdot c$$

- If indexes are available on the join attributes of both r and s , it may be better to use the relation with fewer tuples as the outer relation.



Indexed Nested Loops Join

- Example: compute $student \bowtie takes$, with $student$ as the outer relation.
- Assume $takes$ has a primary B⁺-tree index on the attribute ID (foreign key but not superkey), which contains 20 entries in each tree node. Since $takes$ has 10000 tuples, the B⁺-tree has 4 levels. Assume that only one additional I/O is needed to find all matching $takes$ records.
- Recall that $takes$ has 10000 records and 400 blocks, whereas $student$ has 5000 records and 100 blocks.
- Worst case cost of block nested loops join:
 - $400 \cdot 100 + 100 = 40,100$ block transfers
 - $2 \cdot 100 = 200$ seeks
- Worst case cost of indexed nested loops join:
 - 100 block transfers and seeks for outer relation ($student$)
 - $5000 \cdot 5 = 25,000$ block transfers and seeks for index lookups
 - total = 25,100 block transfers and seeks
(fewer block transfers than block nested loops join, more seeks)



Other Join Algorithms

- The nested loops join can accommodate any condition θ and does not require that the inputs be sorted or indexed. It can take advantage of main memory and indexes if available.
- Other algorithms (not covered in detail in this lecture) may offer better performance than the nested loops join in special cases.
- **Merge join:** merges data from inputs that are sorted on the join attribute(s). The inputs can be sorted explicitly if they are not already sorted (**sort-merge join**), which incurs additional cost especially when an input does not fit in main memory.
- **Hash join:** builds a hash table on the join attribute(s) for one of the inputs, and scans the other input, probing each row against the hash table. Good for large, unsorted, non-indexed inputs. The hash is generally built on the smaller input, which tends to generate a smaller structure that is more likely to fit in memory.



Illustration of Sort-Merge Join

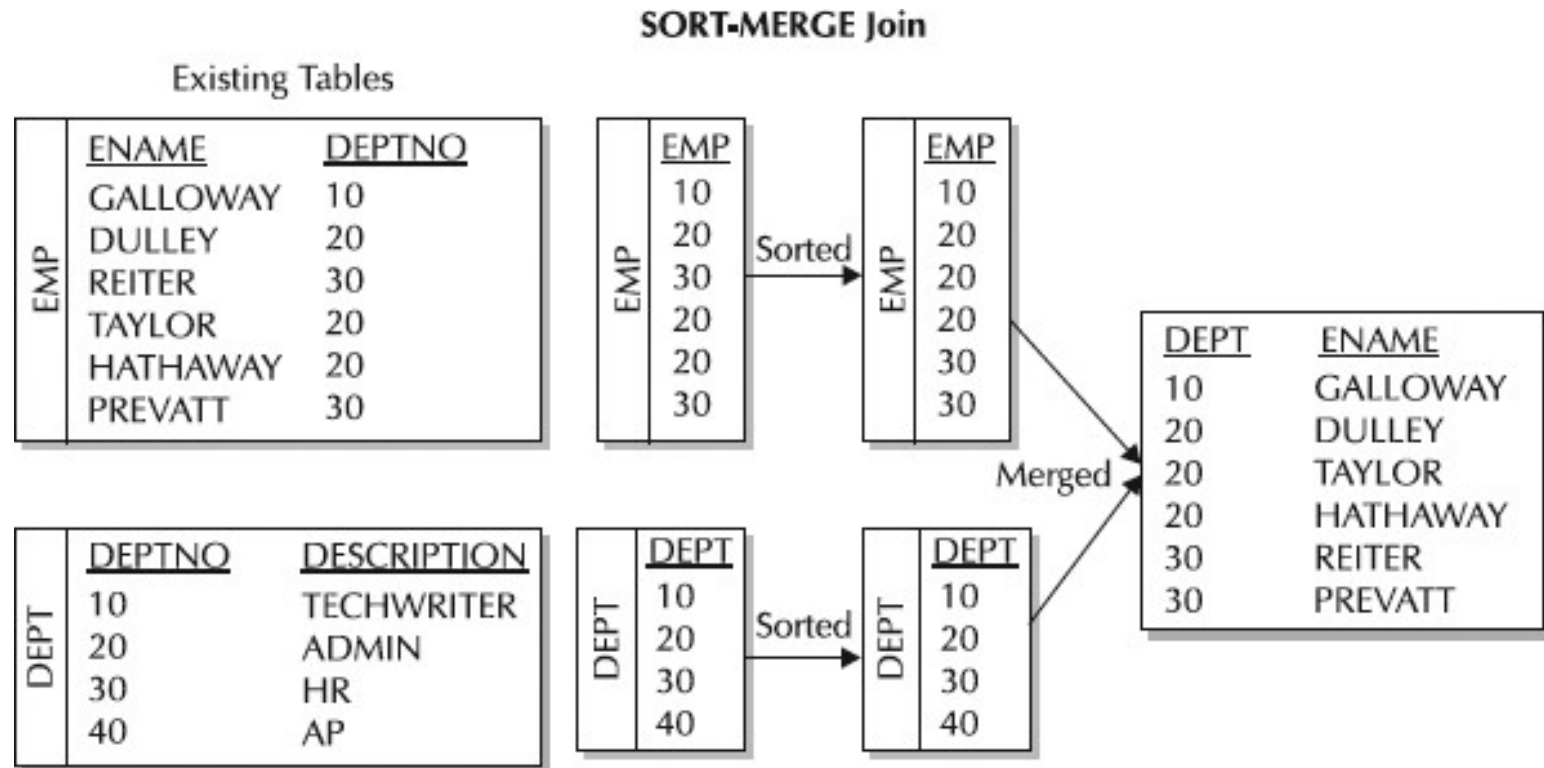


image source: http://logicalread.solarwinds.com/media/371414/0467_001.jpg



Illustration of Hash Join

HASH Join

1. Builds a hash table for EMP table in memory

EMP	<u>ENAME</u>	<u>DEPTNO</u>
	GALLOWY	10
	DULLEY	20
	REITER	30
	TAYLOR	20
	HATHAWAY	20
	PREVATT	30

1 Record Returned

<u>ENAME</u>	<u>DEPTNO</u>
GALLOWAY	10

The loop would continue until each of the DEPTNOs in the DEPT table have been checked against those in the EMP table.

IN MEMORY!!!

(If the **HASH_AREA_SIZE** is large enough)

2. Takes each record from DEPT and compares to HASH table.

DEPT	<u>DESCRIPTION</u>	<u>DEPTNO</u>
	TECHWRITER	10
	ADMIN	20
	HR	30
	AP	40

image source: http://logicalread.solarwinds.com/media/371419/0468_001.jpg



Engineer's Dilemma

- On the one hand, we know that:
 - Adding indexes can speed up some queries drastically.
- On the other hand, we know that:
 - Indexing slows down insertions and updates.
 - Looking up an index repeatedly during a join is not necessarily faster than scanning the inner relation (see indexed nested loops joins vs block nested loops join).
 - The DBMS may create the most important indexes automatically (*e.g.*, on primary keys and foreign keys), and it may not offer many choices of index structure (*e.g.*, InnoDB in MySQL 5.6 only supports B⁺-tree indexes).
 - It is difficult to outsmart a good query optimizer, which has access to detailed statistics about tables and uses elaborate optimization algorithms.



Engineer's Dilemma

- Therefore, designing a good physical schema may require the following:
 - Using a basic understanding of query evaluation, making an educated guess as to what index or indexes might benefit the most important queries.
 - Understanding how an index is used by the query optimizer by inspecting the evaluation plan for a given query.
 - Determining the impact of adding an index by measuring performance differences empirically.
- In this lecture we will focus on the first two points.



Making the Educated Guess

- Example 1: where clause.

select *ID*

from *instructor*

where *dept_name* = "Finance" **and** *salary* < 80000

An index on either *dept_name* or *salary* could be useful.

Separate indexes on *dept_name* and *salary* could be better.

A single index on (*dept_name*, *salary*) could be even better. Why?

- Example 2: join.

select * **from** *instructor* **natural join** *department*

Depending on the join order, an index on either the primary key

department.dept_name or the foreign key *instructor.dept_name* could be useful.

Note: InnoDB will automatically create B⁺-tree indexes on all primary and foreign keys. As a result, in this case there is no need to create an additional index.



Making the Educated Guess

- Example 3: order by.

```
select dept_name, budget
from department
order by budget
```

An index on *budget* could be used to find all the departments and their budgets without additional sorting.

- Example 4: group by.

```
select dept_name, avg(salary)
from instructor
group by dept_name
```

An index on *dept_name* would make it possible to efficiently identify the groups (*i.e.*, departments), and compute the members of the group (*i.e.*, instructors in a given department).



Index Extensions

- **Index extension:** when the DBMS automatically appends the primary key to each secondary index entry (e.g., InnoDB).
 - Example: a secondary index on *department.building* has entries of the form (*building*, *dept_name*).
- Index extensions have important advantages:
 - They make it possible to physically relocate table rows without having to update all secondary indexes.
 - They enable efficient evaluation of queries that refer to both the primary key and non-key attributes. (See next slide!)
- They also have some disadvantages:
 - Retrieving a department record given the *building* requires an additional primary index lookup.
 - The secondary index becomes larger and less likely to remain buffered in main memory.



Covering Indexes

- **Covered query**: a query that can be evaluated using indexes only, without accessing the tables.
- **Covering index** (with respect to a covered query): an index that is used to evaluate the covered query.
- Example:

```
select ID from instructor  
where dept_name = "Finance" and salary < 80000
```

Suppose that a secondary B⁺-tree index is defined on (*dept_name*, *salary*). Due to extension the entries of this index are triples of the form (*dept_name*, *salary*, *ID*). Such an index is a covering index for the above query.



Index Size

- The smaller the index, the more likely it is to remain buffered in main memory and the lower the cost of accessing that index.
- Several techniques can be used to reduce the size of an index.
 - **Technique 1:** shorten the primary key.
 - ▶ Example: Instead of making *department.dept_name* the primary key, create a shorter fixed-length **surrogate key** by adding an auto-increment *ID* attribute to *department*.
 - ▶ If index extension is used, this optimization also benefits all secondary indexes on *department* !
 - **Technique 2:** index only a prefix of a column.
 - ▶ Example: Instead of indexing the *department.building* attribute in full, index only the first 5 characters:
create index *building_part* on *department*
(*building*(5))



Understanding the Table Structure

show table status where *Name* = 'department'

■ MySQL 5.0 output (transposed):

Name: department

Engine: InnoDB

Version: 10

Row_format: Compact

Rows: 7

Avg_row_length: 2340

Data_length: 16384

...



Understanding the Index Structure

show indexes in *instructor*

■ MySQL 5.0 output (transposed):

Table: instructor	Table: instructor
Non_unique: 0	Non_unique: 1
Key_name: PRIMARY	Key_name: dept_name
Seq_in_index: 1	Seq_in_index: 1
Column_name: ID	Column_name: dept_name
Collation: A	Collation: A
Cardinality: 12	Cardinality: 12
...	...
Index_type: BTREE	Index_type: BTREE



Understanding the Evaluation Plan

explain select * from *instructor* natural join *department*

- MySQL 5.0 output (transposed):

id: 1	id: 1
select_type: SIMPLE	select_type: SIMPLE
table: department	table: instructor
type: ALL	type: ref
possible_keys: PRIMARY	possible_keys: dept_name
key: NULL	key: dept_name
key_len: NULL	key_len: 23
...	...
rows: 7	rows: 1

- Which join order did the query optimizer choose?
- Which index was used to evaluate the join?