



Storage and File Structure Indexing and Hashing Query Processing and Optimization Transactions

Database System Concepts, 6th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



Classification of Physical Storage Media

- Speed with which data can be accessed
- Cost per unit of data
- Reliability
 - data loss on power failure or system crash
 - physical failure of the storage device
- Can differentiate storage into:
 - **volatile storage:** loses contents when power is switched off
 - **non-volatile storage:**
 - ▶ Contents persist even when power is switched off.
 - ▶ Includes secondary and tertiary storage, as well as battery-backed up main-memory.



Physical Storage Media

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- **Main memory:**
 - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
 - generally too small (or too expensive) to store the entire database
 - ▶ capacities of up to a few Gigabytes widely used currently
 - ▶ Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
- **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.



Physical Storage Media (Cont.)

□ Flash memory

- Data survives power failure
- Data can be written at a location only once, but location can be erased and written to again
 - ▶ Can support only a limited number (10K – 1M) of write/erase cycles.
 - ▶ Erasing of memory has to be done to an entire bank of memory
- Reads are roughly as fast as main memory
- But writes are slow (few microseconds), erase is slower
- Widely used in embedded devices such as digital cameras, phones, and USB keys



Physical Storage Media (Cont.)

□ Magnetic-disk

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
 - ▶ Much slower access than main memory (more on this later)
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Capacities range up to roughly 1.5 TB as of 2009
 - ▶ Much larger capacity and cost/byte than main memory/flash memory
 - ▶ Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
 - ▶ disk failure can destroy data, but is rare



Physical Storage Media (Cont.)

□ Optical storage

- non-volatile, data is read optically from a spinning disk using a laser
- CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- Blu-ray disks: 27 GB to 54 GB
- Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- Reads and writes are slower than with magnetic disk
- **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data



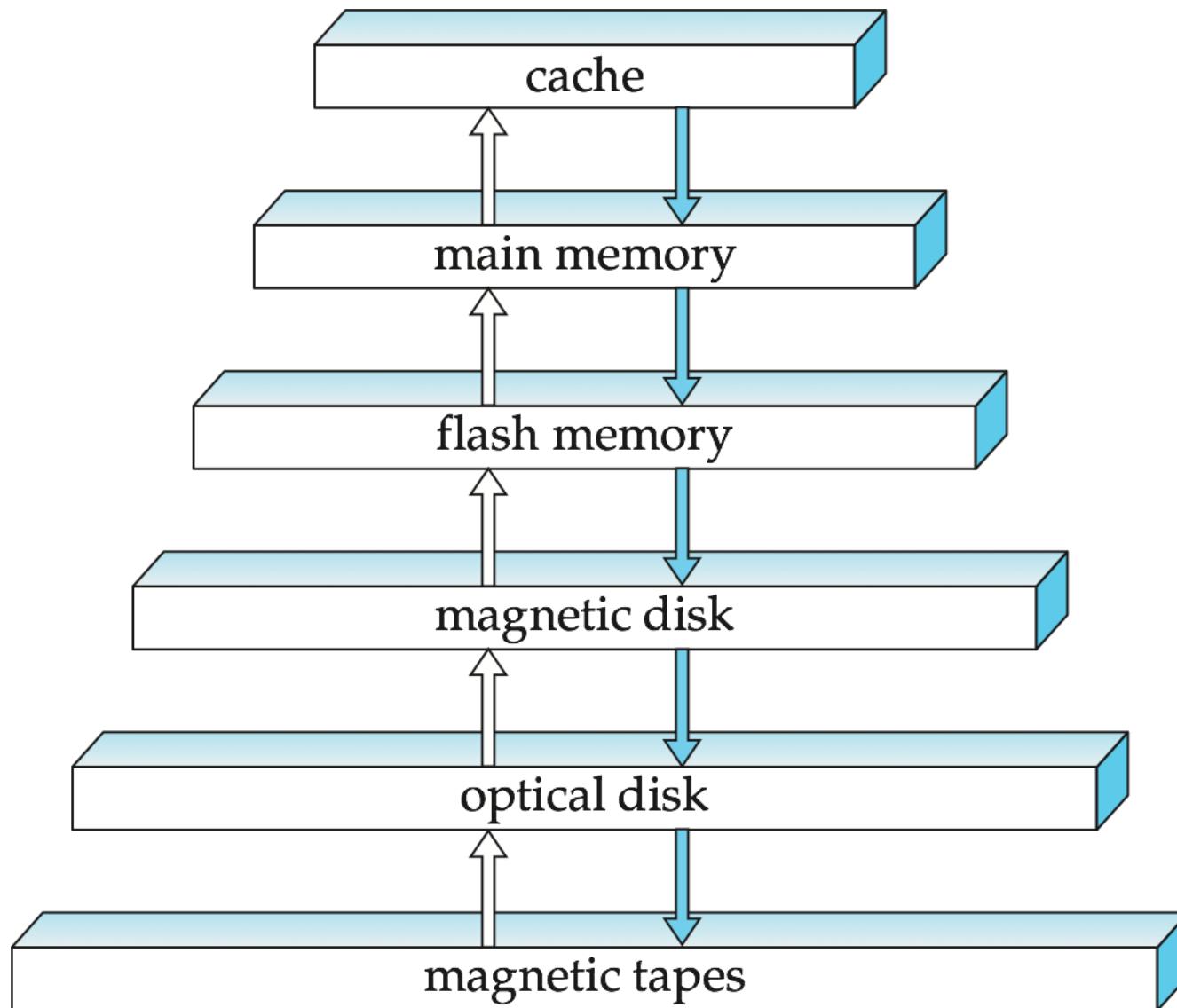
Physical Storage Media (Cont.)

□ Tape storage

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive \Rightarrow storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
 - ▶ hundreds of terabytes (1 terabyte = 10^9 bytes) to even multiple **petabytes** (1 petabyte = 10^{12} bytes)



Storage Hierarchy



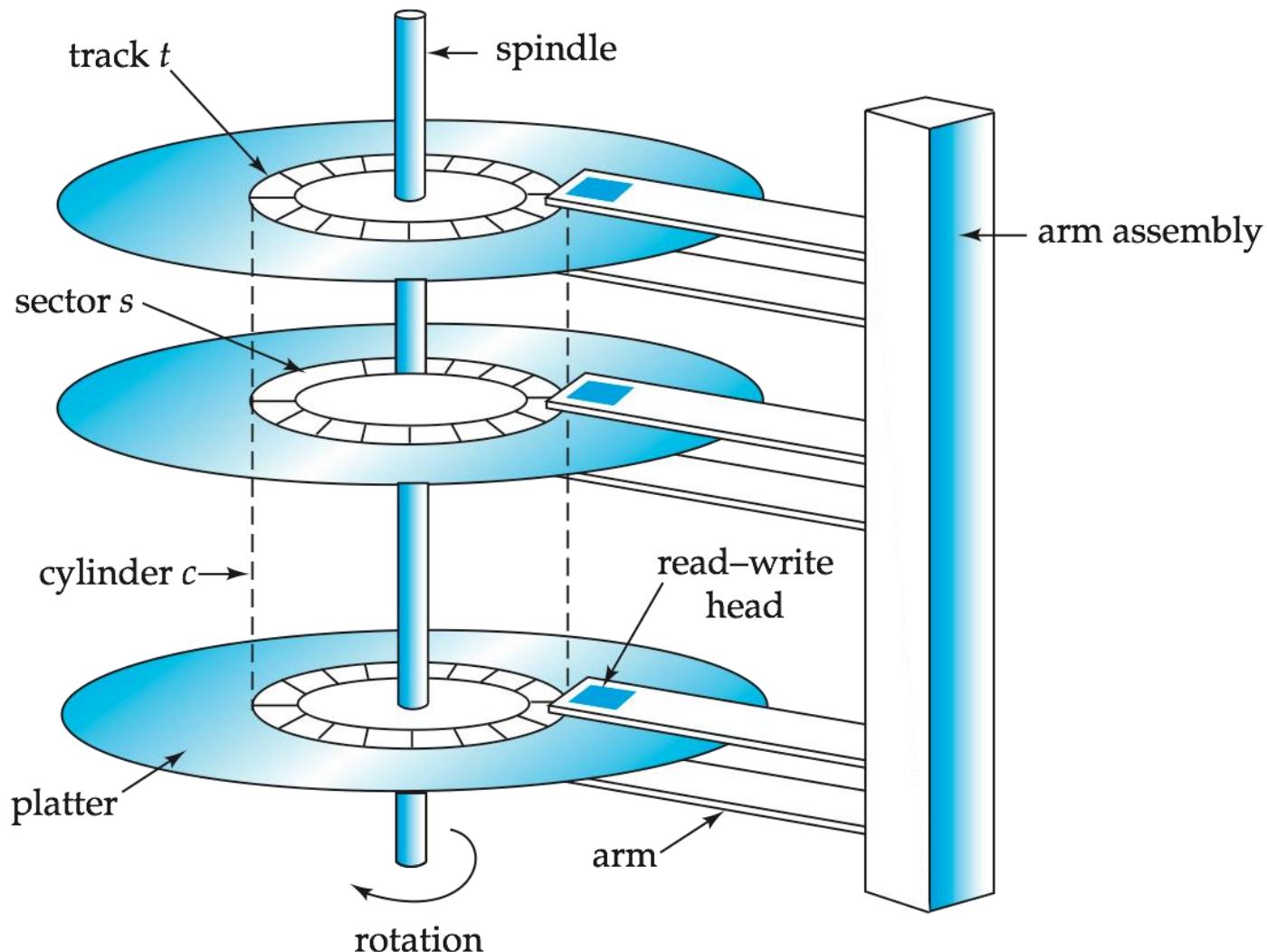


Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
 - also called **on-line storage**
 - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
 - also called **off-line storage**
 - E.g. magnetic tape, optical storage



Magnetic Hard Disk Mechanism



NOTE: Diagram is schematic, and simplifies the structure of actual disk drives



Magnetic Disks

- **Read-write head**
 - Positioned very close to the platter surface (almost touching it)
 - Reads or writes magnetically encoded information.
- Surface of platter divided into circular **tracks**
 - Over 50K-100K tracks per platter on typical hard disks
- Each track is divided into **sectors**.
 - A sector is the smallest unit of data that can be read or written.
 - Sector size typically 512 bytes
 - Typical sectors per track: 500 to 1000 (on inner tracks) to 1000 to 2000 (on outer tracks)
- To read/write a sector
 - disk arm swings to position head on right track
 - platter spins continually; data is read/written as sector passes under head
- Head-disk assemblies
 - multiple disk platters on a single spindle (1 to 5 usually)
 - one head per platter, mounted on a common arm.
- **Cylinder** i consists of i^{th} track of all the platters



Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
 - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
 - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- **Disk controller** – interfaces between the computer system and the disk drive hardware.
 - accepts high-level commands to read or write a sector
 - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
 - Computes and attaches **checksums** to each sector to verify that data is read back correctly
 - ▶ If data is corrupted, with very high probability stored checksum won't match recomputed checksum
 - Ensures successful writing by reading back sector after writing it
 - Performs remapping of bad sectors



Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
 - **Seek time** – time it takes to reposition the arm over the correct track.
 - ▶ Average seek time is 1/2 the worst case seek time.
 - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
 - ▶ 4 to 10 milliseconds on typical disks
 - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
 - ▶ Average latency is 1/2 of the worst case latency.
 - ▶ 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
 - 25 to 100 MB per second max rate, lower for inner tracks
 - Multiple disks may share a controller, so rate that controller can handle is also important
 - ▶ E.g. SATA: 150 MB/sec, SATA-II 3Gb (300 MB/sec)
 - ▶ Ultra 320 SCSI: 320 MB/s, SAS (3 to 6 Gb/sec)
 - ▶ Fiber Channel (FC2Gb or 4Gb): 256 to 512 MB/s



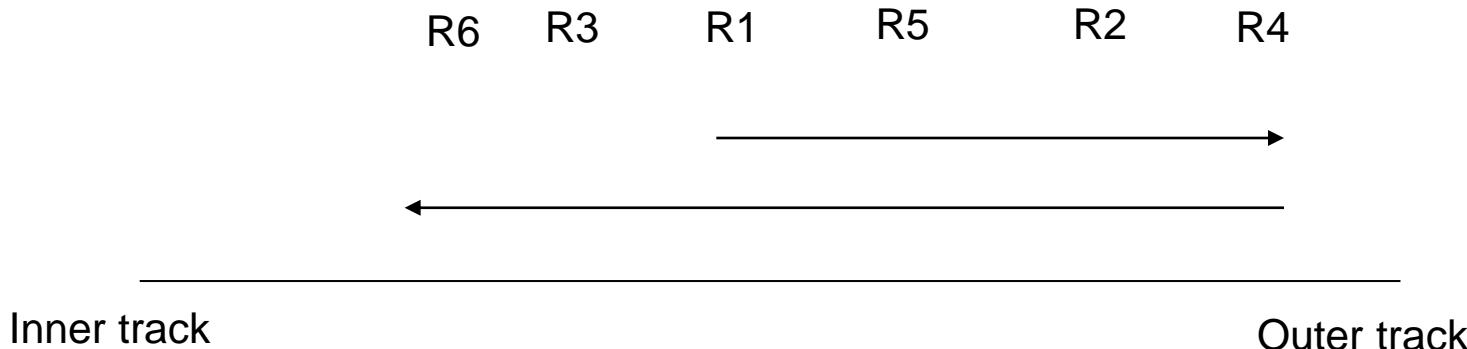
Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
 - Typically 3 to 5 years
 - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
 - ▶ E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
 - MTTF decreases as disk ages



Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
 - data is transferred between disk and main memory in blocks
 - 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
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
 - **elevator algorithm:**





Optimization of Disk Block Access (Cont.)

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
 - E.g. Store related information on the same or nearby cylinders.
 - Files may get **fragmented** over time
 - ▶ E.g. if data is inserted to/deleted from the file
 - ▶ Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
 - ▶ Sequential access to a fragmented file results in increased disk arm movement
 - Some systems have utilities to **defragment** the file system, in order to speed up file access



Optimization of Disk Block Access (Cont.)

- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
 - Non-volatile RAM: battery backed up RAM or flash memory
 - Even if power fails, the data is safe and will be written to disk when power returns
 - Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
 - Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
 - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
 - Used exactly like nonvolatile RAM
 - Write to log disk is very fast since no seeks are required
 - No need for special hardware (NV-RAM)
- File systems typically reorder writes to disk to improve performance
 - **Journaling file systems** write data in safe order to NV-RAM or log disk
 - Reordering without journaling: risk of corruption of file system data



Flash Storage

- used widely for storage
- requires page-at-a-time read (page: 512 bytes to 4 KB)
- transfer rate around 20 MB/sec
- **solid state disks**: use multiple flash storage devices to provide higher transfer rate of 100 to 200 MB/sec
- erase is very slow (1 to 2 millisecs)
 - ▶ erase block contains multiple pages
 - ▶ after 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used



RAID

- **RAID: Redundant Arrays of Independent Disks**
 - disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
 - ▶ **high capacity** and **high speed** by using multiple disks in parallel,
 - ▶ **high reliability** by storing data redundantly, so that data can be recovered even if a disk 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)
 - Techniques for using redundancy to avoid data loss are critical with large numbers of disks
- Originally a cost-effective alternative to large, expensive disks
 - I in RAID originally stood for ``inexpensive''
 - Today RAIDs are used for their higher reliability and bandwidth.
 - ▶ The “I” is interpreted as independent



Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
 - Duplicate every disk. Logical disk consists of two physical disks.
 - Every write is carried out on both disks
 - ▶ Reads can take place from either disk
 - If one disk in a pair fails, data still available in the other
 - ▶ Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
 - Probability of combined event is very small
 - » Except for dependent failure modes such as fire or building collapse or electrical power surges



Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
 1. Load balance multiple small accesses to increase throughput
 2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
 - In an array of eight disks, write bit i of each byte to disk i .
 - Each access can read data at eight times the rate of a single disk.
 - But seek/access time worse than for a single disk
 - ▶ Bit level striping is not used much any more
- **Block-level striping** – with n disks, block i of a file goes to disk $(i \bmod n) + 1$
 - 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

- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
 - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
- **RAID Level 0:** Block striping; non-redundant.
 - Used in high-performance applications where data loss is not critical.
- **RAID Level 1:** Mirrored disks with block striping
 - Offers best write performance.
 - Popular for applications such as storing log files in a database system.



(a) RAID 0: nonredundant striping

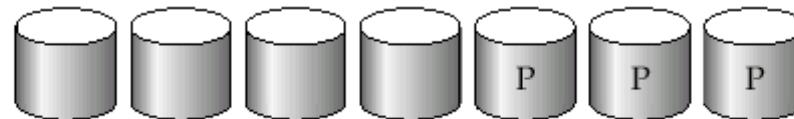


(b) RAID 1: mirrored disks

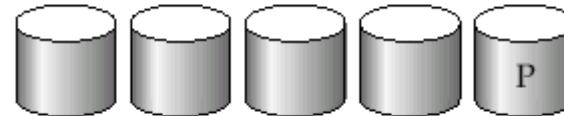


RAID Levels (Cont.)

- **RAID Level 2:** Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- **RAID Level 3:** Bit-Interleaved Parity
 - a single parity bit is enough for error correction, not just detection, since we know which disk has failed
 - ▶ When writing data, corresponding parity bits must also be computed and written to a parity bit disk
 - ▶ To recover data in a damaged disk, compute XOR of bits from other disks (including parity bit disk)



(c) RAID 2: memory-style error-correcting codes



(d) RAID 3: bit-interleaved parity



RAID Levels (Cont.)

- RAID Level 3 (Cont.)

- Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.
 - Subsumes Level 2 (provides all its benefits, at lower cost).

- RAID Level 4: Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for corresponding blocks from N other disks.

- When writing data block, corresponding block of parity bits must also be computed and written to parity disk
 - To find value of a damaged block, compute XOR of bits from corresponding blocks (including parity block) from other disks.



(e) RAID 4: block-interleaved parity



RAID Levels (Cont.)

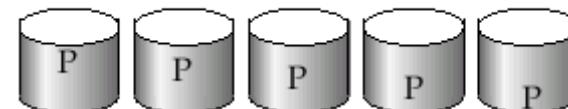
□ RAID Level 4 (Cont.)

- Provides higher I/O rates for independent block reads than Level 3
 - ▶ block read goes to a single disk, so blocks stored on different disks can be read in parallel
- Provides high transfer rates for reads of multiple blocks than no-striping
- Before writing a block, parity data must be computed
 - ▶ Can be done by using old parity block, old value of current block and new value of current block (2 block reads + 2 block writes)
 - ▶ Or by recomputing the parity value using the new values of blocks corresponding to the parity block
 - More efficient for writing large amounts of data sequentially
- Parity block becomes a bottleneck for independent block writes since every block write also writes to parity disk



RAID Levels (Cont.)

- **RAID Level 5:** Block-Interleaved Distributed Parity; partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - 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.



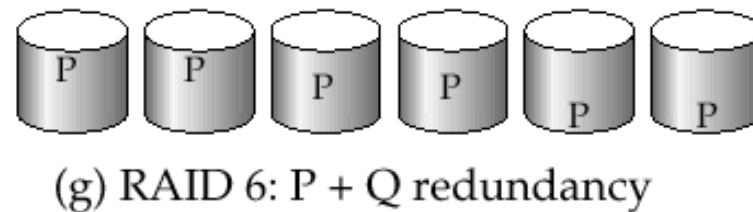
(f) RAID 5: block-interleaved distributed parity

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 Levels (Cont.)

- RAID Level 5 (Cont.)
 - Higher I/O rates than Level 4.
 - ▶ Block writes occur in parallel if the blocks and their parity blocks are on different disks.
 - Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.
- RAID Level 6: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
 - Better reliability than Level 5 at a higher cost; not used as widely.



(g) RAID 6: P + Q redundancy



Choice of RAID Level

- Factors in choosing RAID level
 - Monetary cost
 - Performance: Number of I/O operations per second, and bandwidth during normal operation
 - Performance during failure
 - Performance during rebuild of failed disk
 - ▶ Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
 - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (level 5) avoids
- Level 6 is rarely used since levels 1 and 5 offer adequate safety for most applications



Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
 - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
 - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
 - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
 - I/O requirements have increased greatly, e.g. for Web servers
 - When enough disks have been bought to satisfy required rate of I/O, they often have spare storage capacity
 - ▶ so there is often no extra monetary cost for Level 1!
- Level 5 is preferred for applications with low update rate, and large amounts of data
- Level 1 is preferred for all other applications



Optical Disks

- Compact disk-read only memory (CD-ROM)
 - Removable disks, 640 MB per disk
 - Seek time about 100 msec (optical read head is heavier and slower)
 - Higher latency (3000 RPM) and lower data-transfer rates (3-6 MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
 - DVD-5 holds 4.7 GB , and DVD-9 holds 8.5 GB
 - DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB and 17 GB
 - Blu-ray DVD: 27 GB (54 GB for double sided disk)
 - Slow seek time, for same reasons as CD-ROM
- Record once versions (CD-R and DVD-R) are popular
 - data can only be written once, and cannot be erased.
 - high capacity and long lifetime; used for archival storage
 - Multi-write versions (CD-RW, DVD-RW, DVD+RW and DVD-RAM) also available



Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
 - Few GB for DAT (Digital Audio Tape) format, 10-40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
 - Transfer rates from few to 10s of MB/s
- Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic and optical disks
 - limited to sequential access.
 - Some formats (Accelis) provide faster seek (10s of seconds) at cost of lower capacity
- Used mainly for backup, for storage of infrequently used information, and as an off-line medium for transferring information from one system to another.
- Tape jukeboxes used for very large capacity storage
 - Multiple petabytes (10^{15} bytes)



File Organization, Record Organization and Storage Access



File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
 - 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.



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
 - ▶ Modification: do not allow records to cross block boundaries

- Deletion of record i : alternatives:
 - move records $i + 1, \dots, n$ to $i, \dots, n - 1$
 - move record n to i
 - do not move records, but link all free records on 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



Deleting record 3 and compacting

record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
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



Deleting record 3 and moving last record

record 0
record 1
record 2
record 11
record 4
record 5
record 6
record 7
record 8
record 9
record 10

	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
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



Free Lists

- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.
- More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

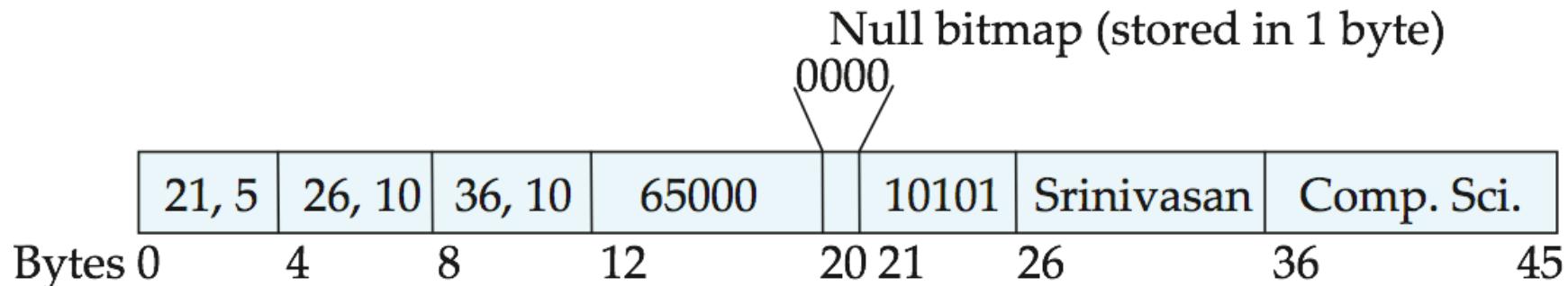
header			
record 0	10101	Srinivasan	Comp. Sci. 65000
record 1			
record 2	15151	Mozart	Music 40000
record 3	22222	Einstein	Physics 95000
record 4			
record 5	33456	Gold	Physics 87000
record 6			
record 7	58583	Califieri	History 62000
record 8	76543	Singh	Finance 80000
record 9	76766	Crick	Biology 72000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

The diagram illustrates the use of free lists. It shows a table with 12 records. Records 0, 2, 3, 5, 7, 8, 9, 10, and 11 contain data. Records 1, 4, and 6 are empty. Arrows point from the fourth column of records 1, 4, and 6 to the first column of record 5, indicating that record 5 is used to store pointers to the locations of deleted records.



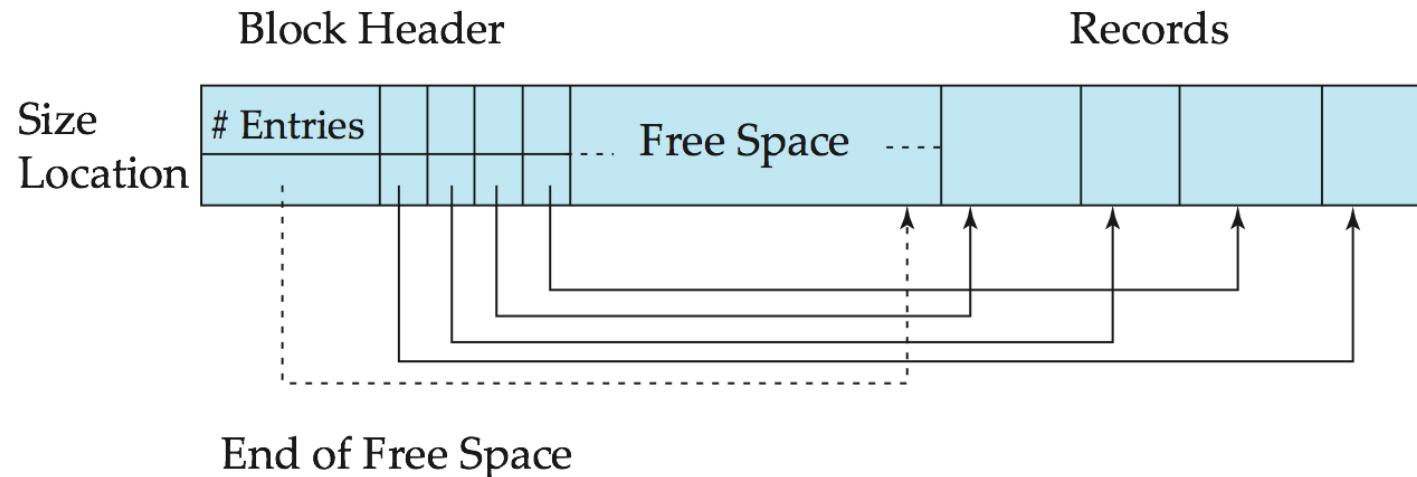
Variable-Length Records

- 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 strings (**varchar**)
 - Record types that allow repeating fields (used in some older data models).
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value 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 with no empty space between them; entry in the header must be updated.
 - Pointers should not point directly to 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
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
 - Motivation: store related records on the same block to minimize I/O



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**

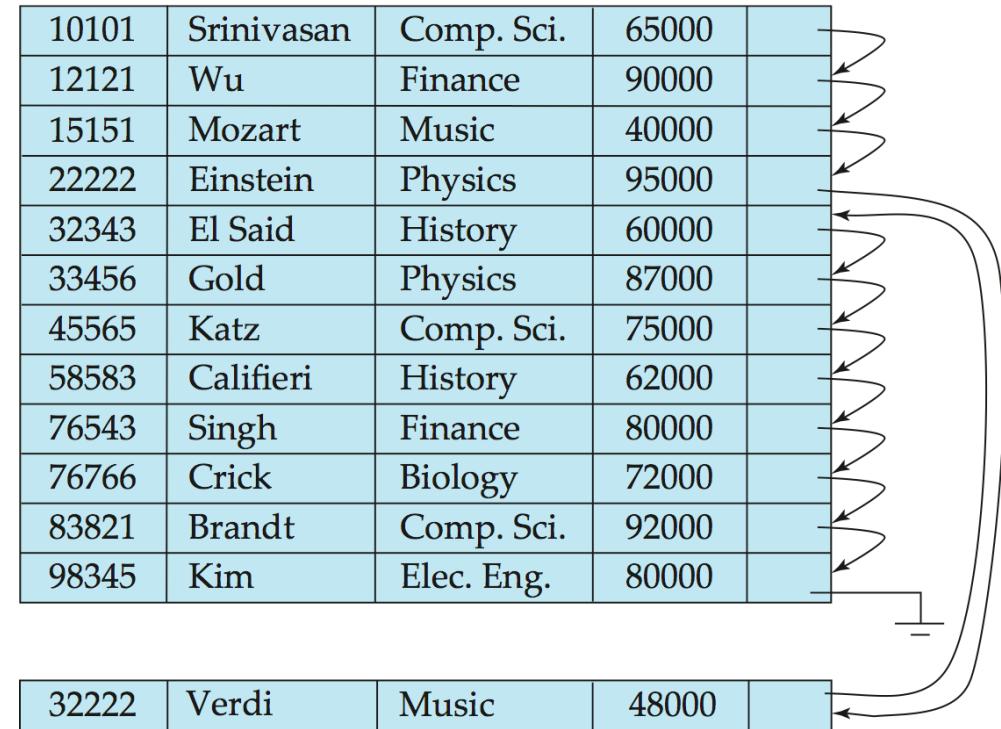
10101	Srinivasan	Comp. Sci.	65000	
12121	Wu	Finance	90000	
15151	Mozart	Music	40000	
22222	Einstein	Physics	95000	
32343	El Said	History	60000	
33456	Gold	Physics	87000	
45565	Katz	Comp. Sci.	75000	
58583	Califieri	History	62000	
76543	Singh	Finance	80000	
76766	Crick	Biology	72000	
83821	Brandt	Comp.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

A diagram illustrating the sequential organization of the data. It shows a vertical stack of ten records from the table above. Each record is represented by a light blue rectangular box containing four fields: a student ID, a name, a major, and a salary. A curved arrow originates from the right edge of each record box and points downwards towards a single horizontal line at the bottom of the slide, representing the sequential reading of the file.



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 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* \bowtie *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

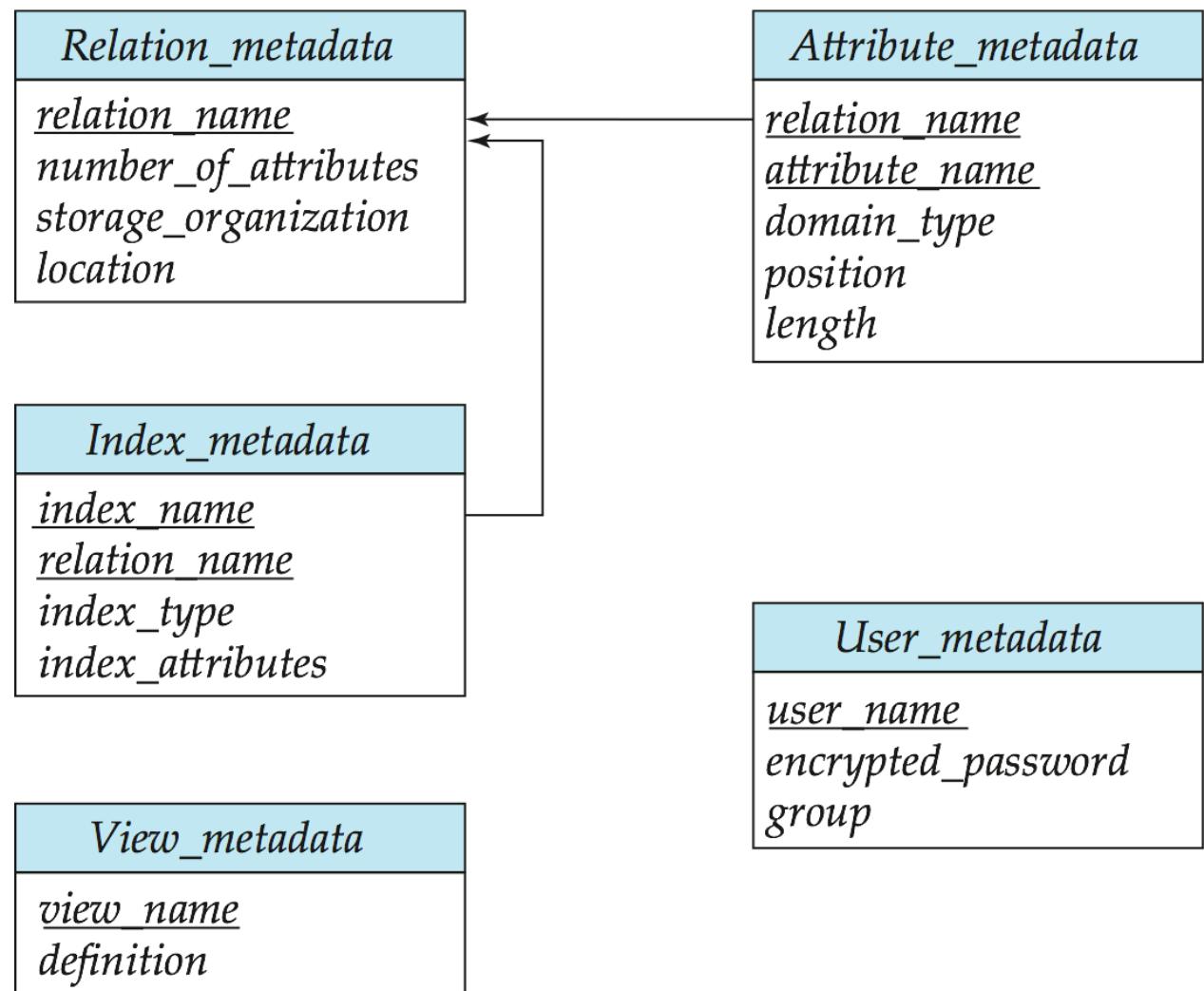
The **Data dictionary** (also called **system catalog**) stores **metadata**; that is, data about data, such as

- 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 relation is stored (sequential/hash/...)
 - Physical location of relation
- Information about indices



Relational Representation of System Metadata

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





Storage Access

- A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.
- Database system seeks 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** – portion of main memory available to store copies of disk blocks.
- **Buffer manager** – subsystem responsible for allocating buffer space in main memory.



Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
 1. If the block is already in the buffer, buffer manager returns the address of the block in main memory
 2. If the block is not in the buffer, the buffer manager
 1. Allocates space in the buffer for the block
 1. Replacing (throwing out) some other block, if required, to make space for the new block.
 2. Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.
 2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.



Buffer-Replacement Policies

- Most operating systems replace the block **least recently used (LRU strategy)**
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
 - LRU can be a bad strategy for certain access patterns involving repeated scans of data
 - ▶ For example: when computing the join of 2 relations r and s by a nested loops
 - for each tuple tr of r do
 - for each tuple ts of s do
 - if the tuples tr and ts match ...
 - Mixed strategy with hints on replacement strategy provided by the query optimizer is preferable



Buffer-Replacement Policies (Cont.)

- **Pinned block** – memory block that is not allowed to be written back to disk.
- **Toss-immediate** strategy – frees the space occupied by a block as soon as the final tuple of that block has been processed
- **Most recently used (MRU) strategy** – system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.
- Buffer manager can use statistical information regarding the probability that a request will reference a particular relation
 - E.g., the data dictionary is frequently accessed. Heuristic: keep data-dictionary blocks in main memory buffer
- Buffer managers also support **forced output** of blocks for the purpose of recovery



Introduction to Indexing and Hashing



Indexing: Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
 - E.g., author catalog in library
- **Search Key** - attribute or set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

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

- Index files are typically much smaller than the original file
- Two basic kinds of indices:
 - **Ordered indices:** search keys are stored in sorted order
 - **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.



Index Evaluation Metrics

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



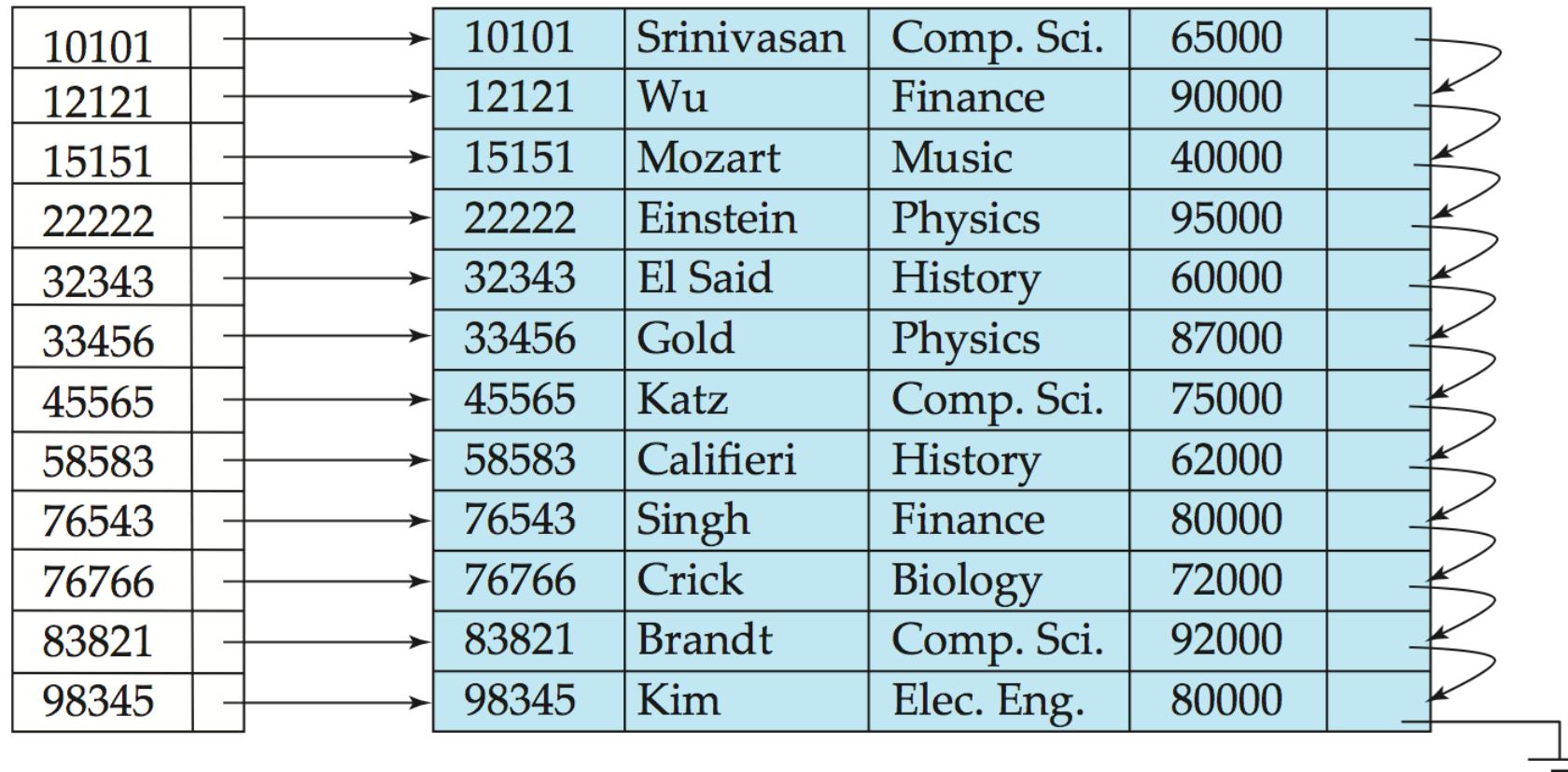
Ordered Indices

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



Dense Index Files

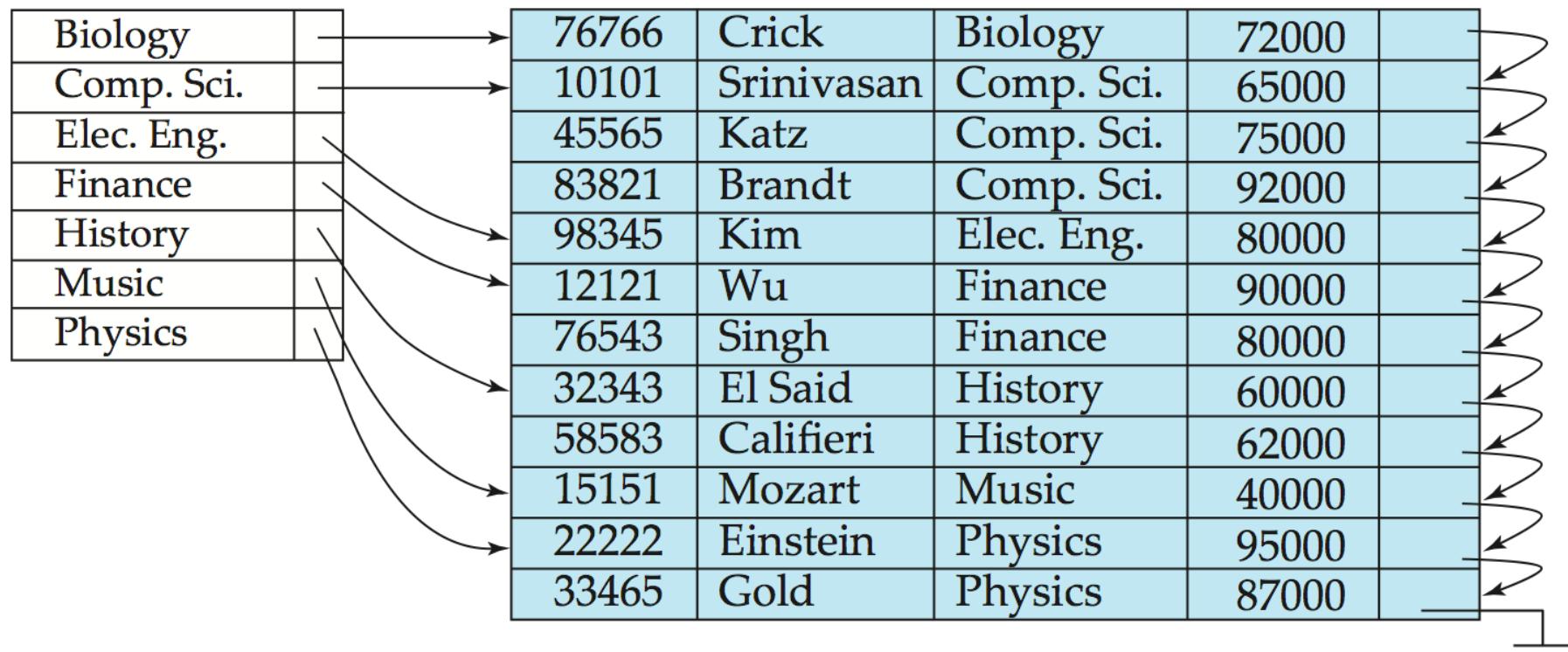
- **Dense index** — Index record appears for every search-key value in the file.
- E.g. index on *ID* attribute of *instructor* relation





Dense Index Files (Cont.)

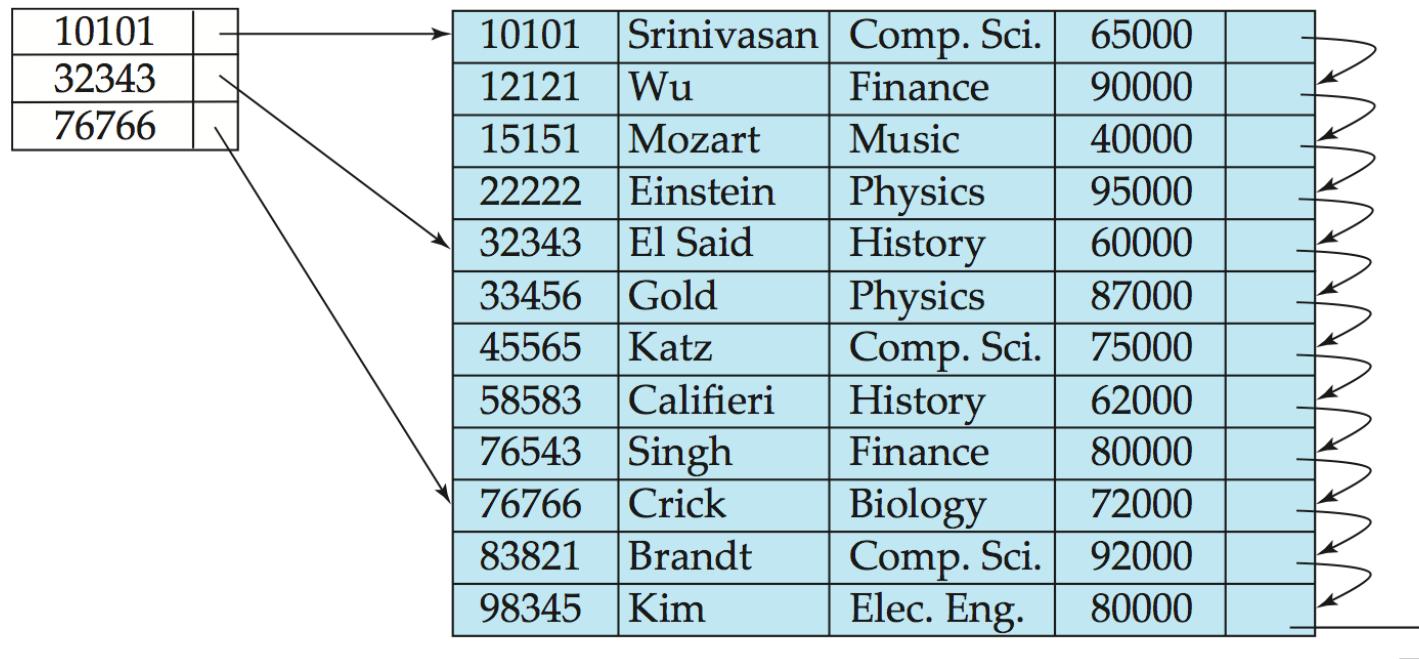
- Dense index on *dept_name*, with *instructor* file sorted on *dept_name*





Sparse Index Files

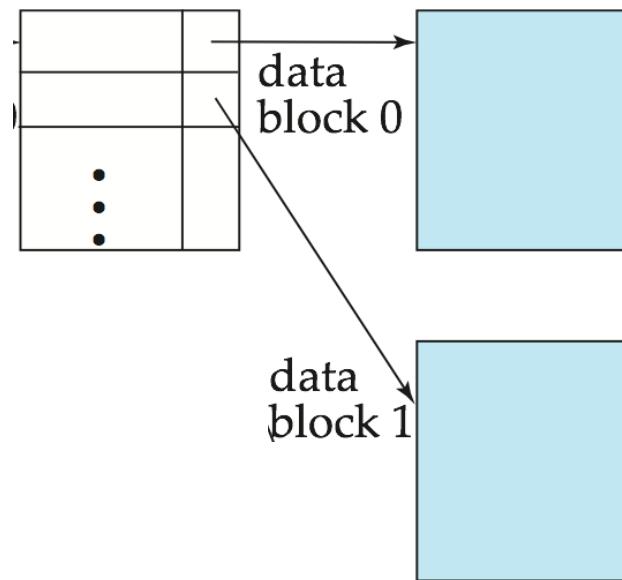
- **Sparse Index:** contains index records for only some search-key values.
 - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value K we:
 - Find index record with largest search-key value $< K$
 - Search file sequentially starting at the record to which the index record points





Sparse Index Files (Cont.)

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



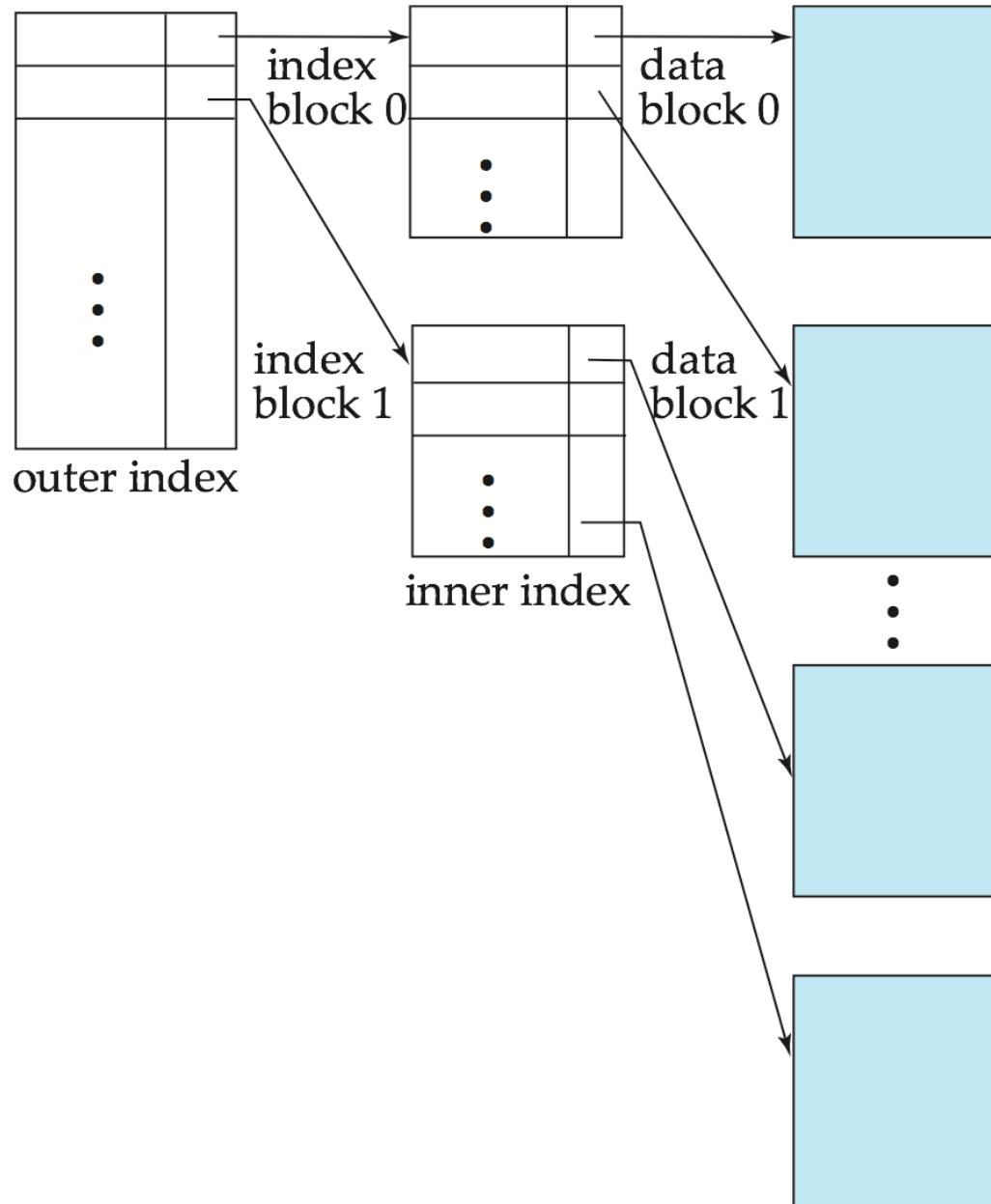


Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
 - outer index – a sparse index of primary index
 - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.



Multilevel Index (Cont.)



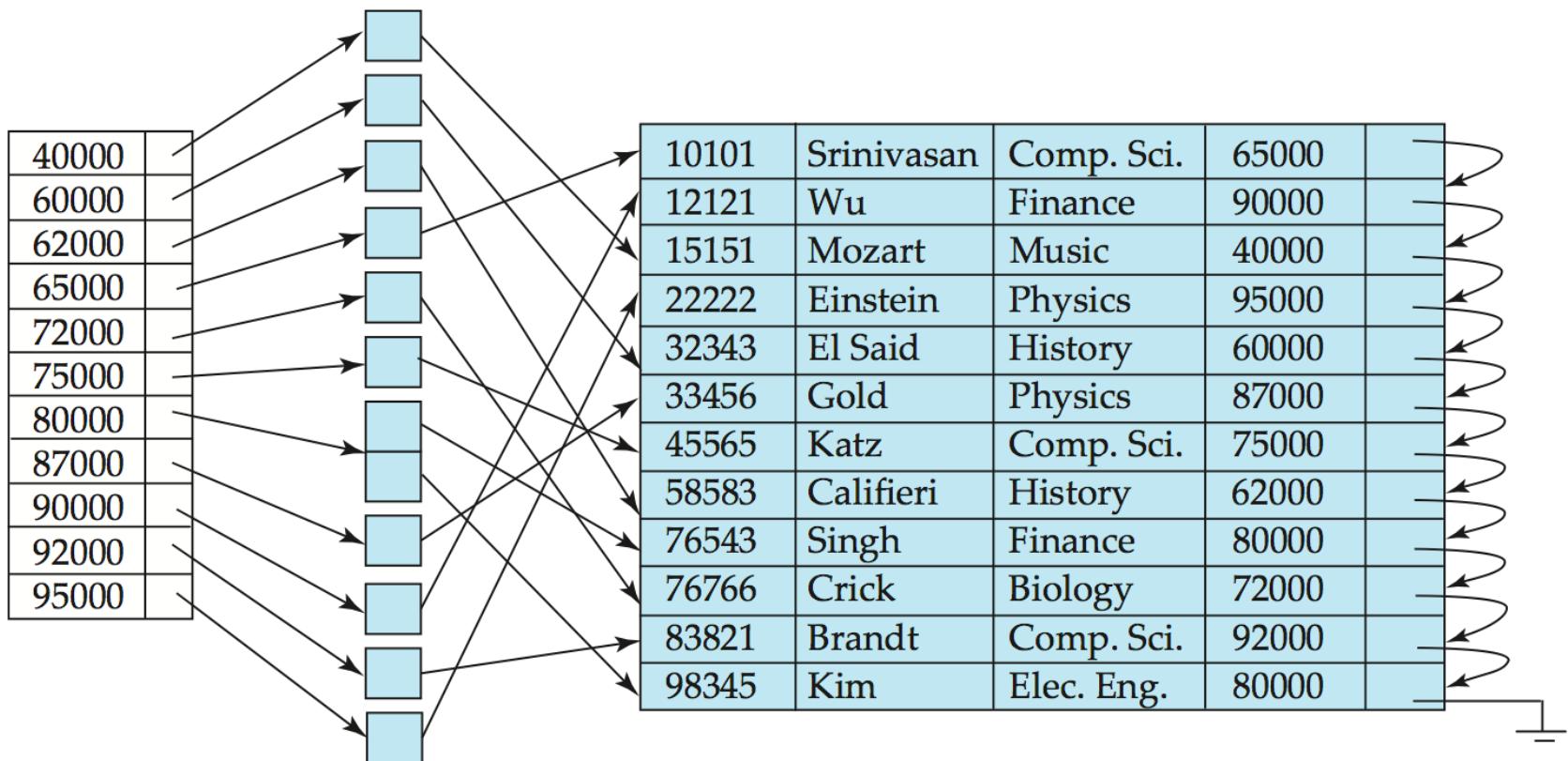


Secondary Indices

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



Secondary Indices Example



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



Primary and Secondary Indices

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



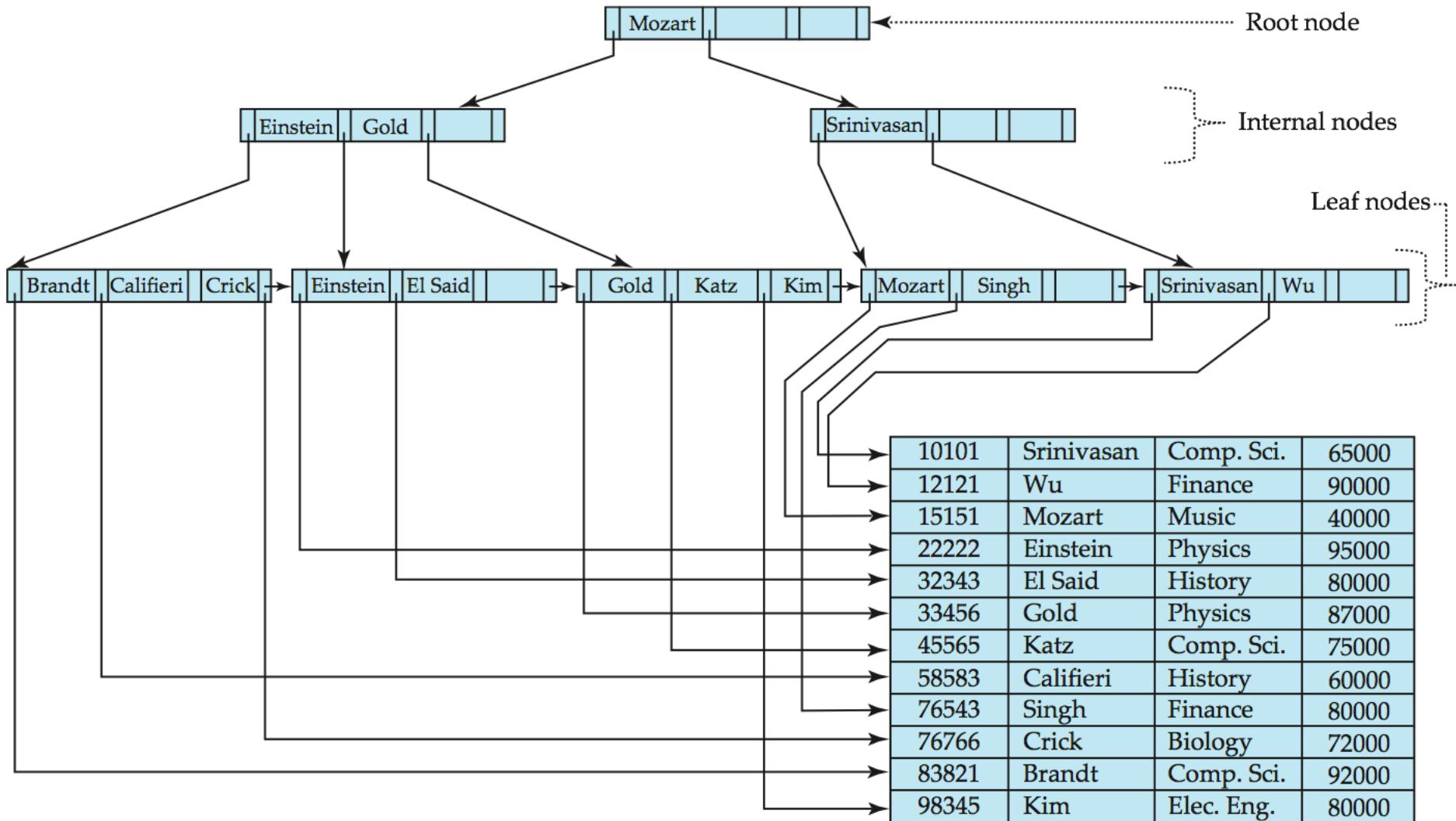
B⁺-Tree Index Files

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

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire file is required.
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
 - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively



Example of B+-Tree





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. $h(\text{Music}) = 1 \quad h(\text{History}) = 2$
 $h(\text{Physics}) = 3 \quad h(\text{Elec. Eng.}) = 3$



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).



Hash Functions

- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
 - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .



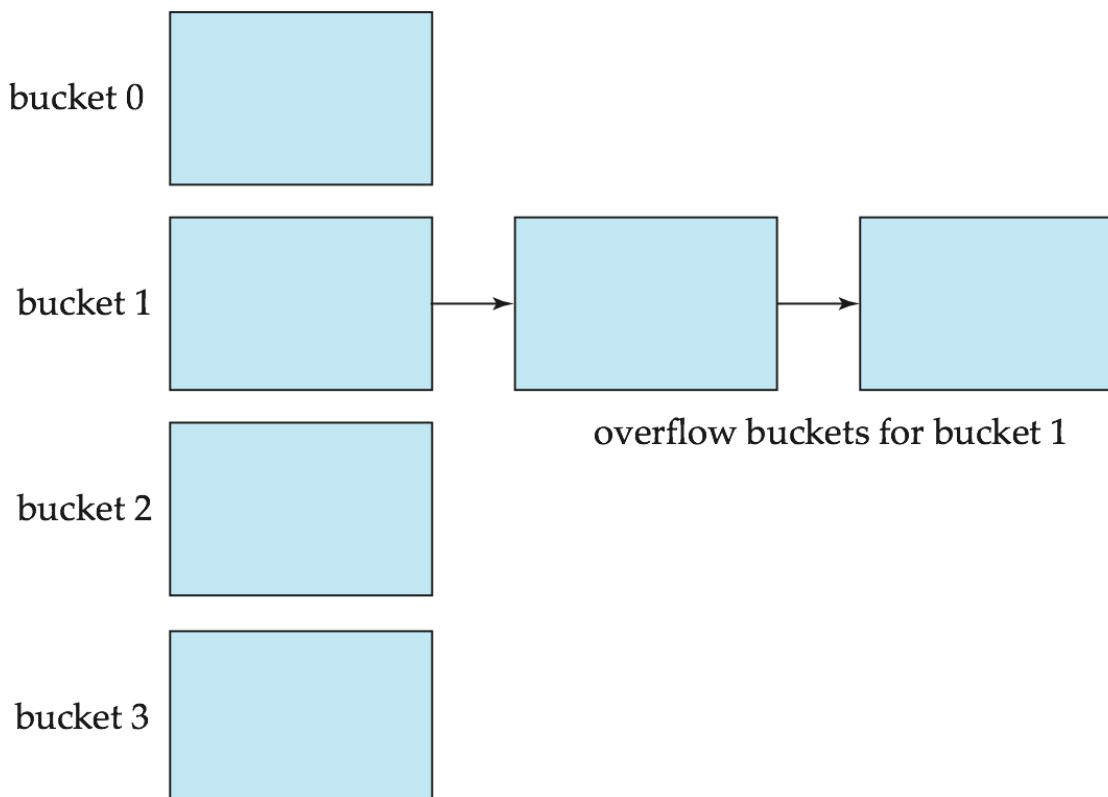
Handling of Bucket Overflows

- Bucket overflow can occur because of
 - Insufficient buckets
 - Skew in distribution of records. This can occur due to two reasons:
 - ▶ multiple records have same search-key value
 - ▶ chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.



Handling of Bucket Overflows (Cont.)

- **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called **closed hashing**.
 - An alternative, called **open hashing**, which does not use overflow buckets, is not suitable for database applications.





Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
 - Given a number n it must be easy to retrieve record n
 - ▶ Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
 - E.g. gender, country, state, ...
 - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits



Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
 - Bitmap has as many bits as records
 - In a bitmap for value v , the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	<i>ID</i>	<i>gender</i>	<i>income_level</i>
0	76766	m	L1
1	22222	f	L2
2	12121	f	L1
3	15151	m	L4
4	58583	f	L3

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

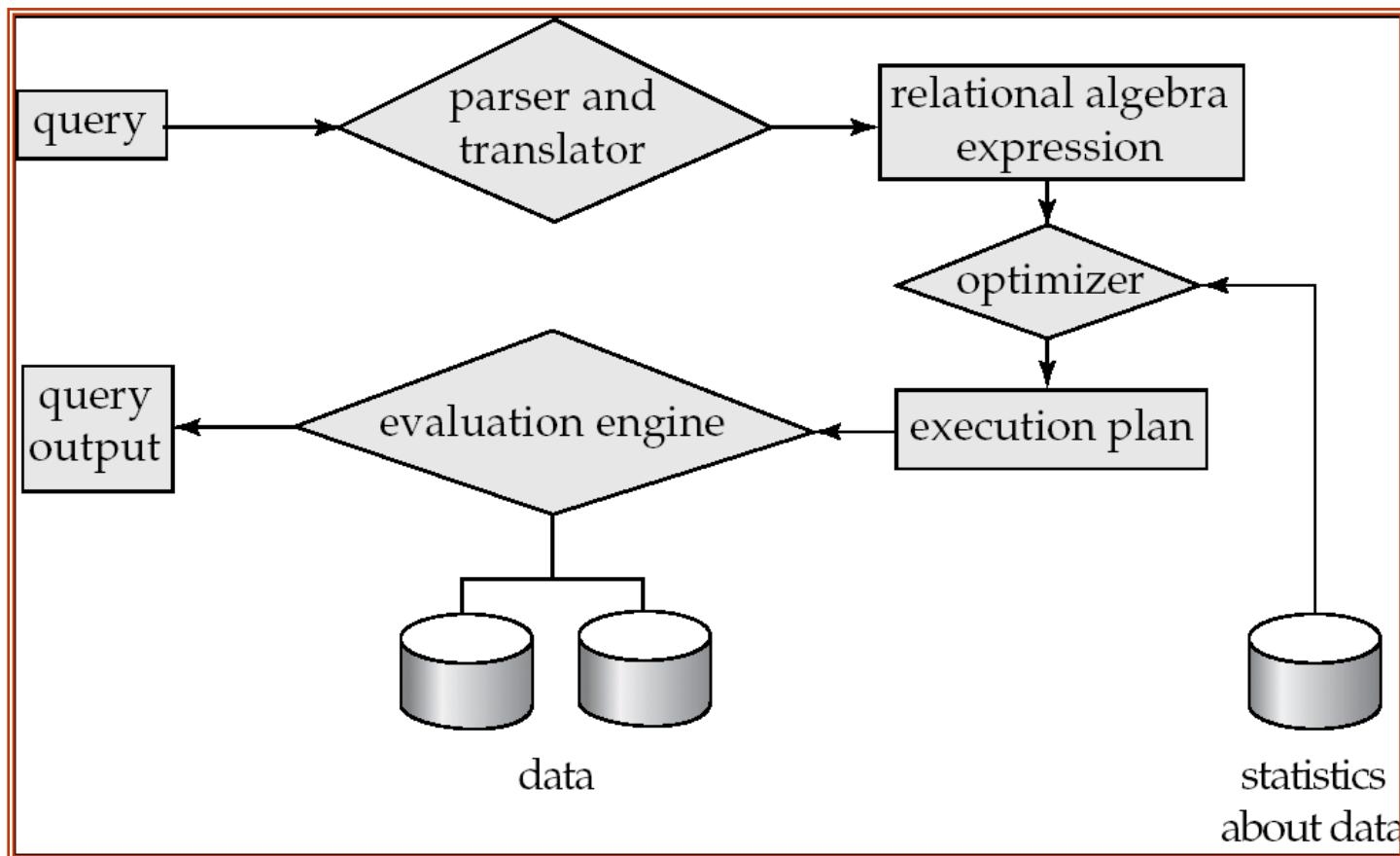


Introduction to Query Processing and Optimization



Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation





Basic Steps in Query Processing (Cont.)

- Parsing and translation
 - translate the query into its internal form. This is then translated into relational algebra.
 - Parser checks syntax, verifies relations
- Evaluation
 - The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.



Basic Steps in Query Processing : Optimization (example)

```
select balance  
from account  
where balance < 2500
```

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{balance < 2500}(\Pi_{balance}(account))$ is equivalent to
$$\Pi_{balance}(\sigma_{balance < 2500}(account))$$
- Each relational algebra operation can be evaluated using one of several different algorithms
 - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
 - E.g., can use an index on *balance* to find accounts with $balance < 2500$,
 - or can perform complete relation scan and discard accounts with $balance \geq 2500$



Basic Steps: Optimization (Cont.)

- **Query Optimization:** Amongst all equivalent evaluation plans choose the one with lowest cost.
 - Cost is estimated using statistical information from the database catalog
 - ▶ e.g. number of tuples in each relation, size of tuples, etc.



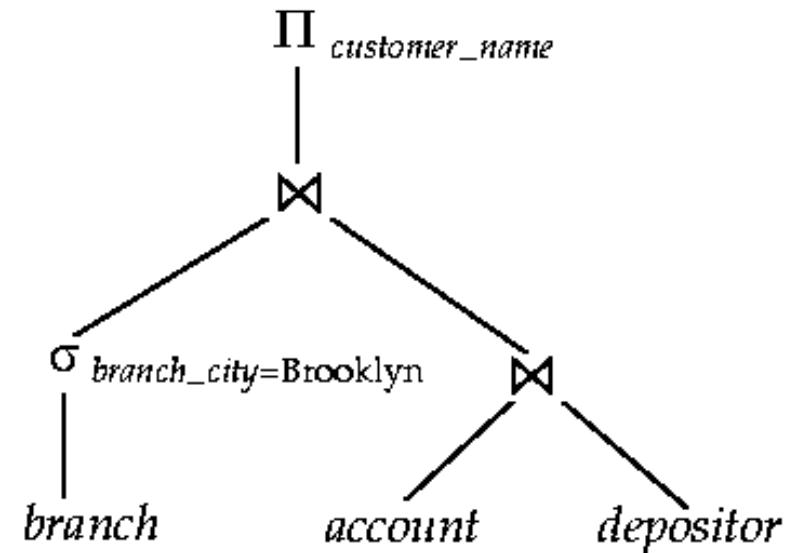
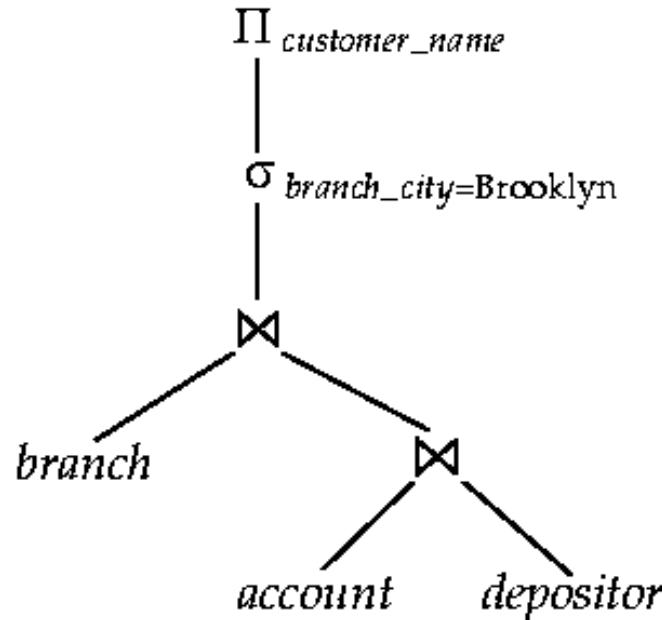
Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
 - Many factors contribute to time cost
 - ▶ disk accesses, CPU, or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
 - Number of seeks * average-seek-cost
 - Number of blocks read * average-block-read-cost
 - Number of blocks written * average-block-write-cost
 - ▶ Cost to write a block is greater than cost to read a block
 - data is read back after being written to ensure that the write was successful



Alternative Query Expressions (ex.)

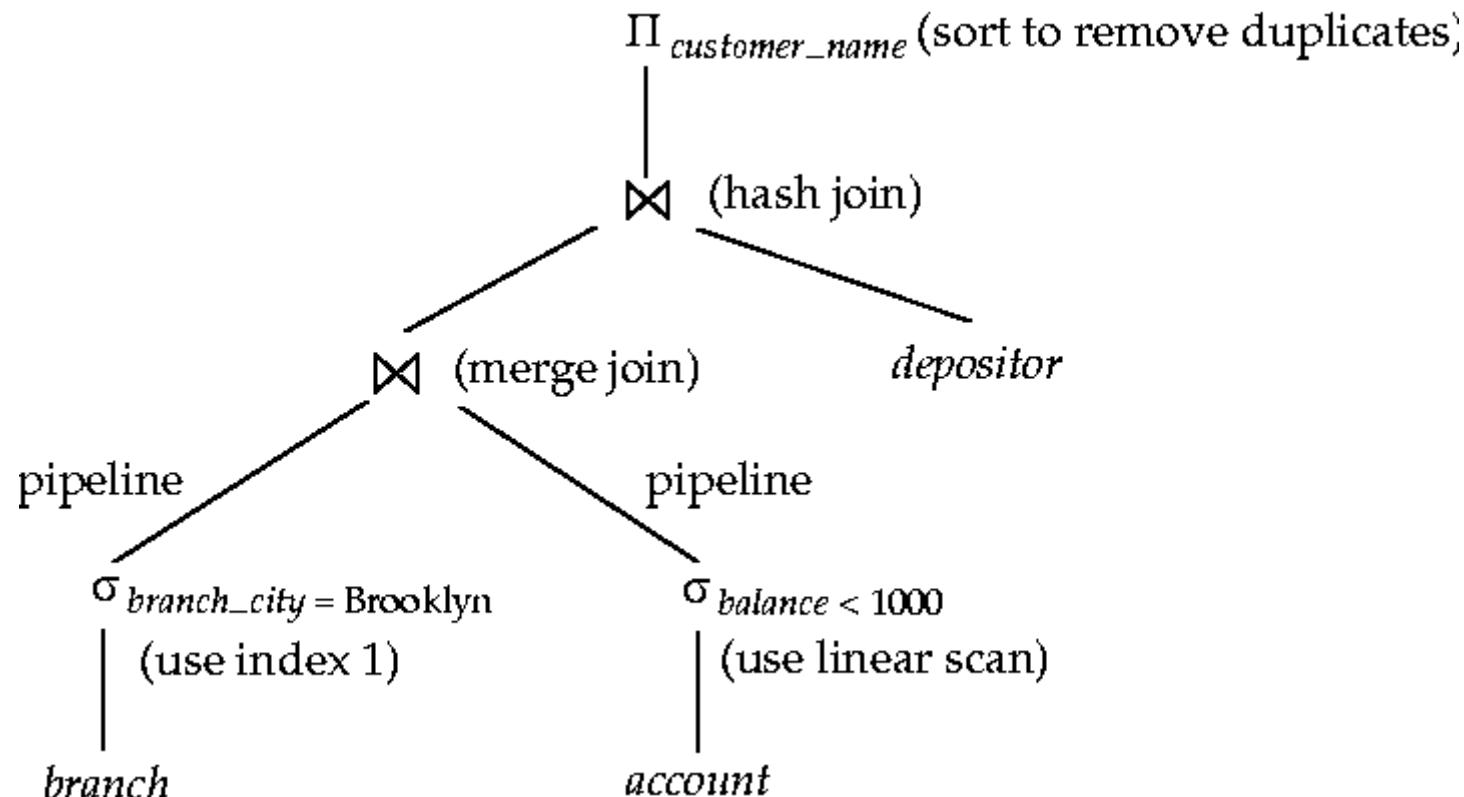
- Alternative ways of evaluating a given query
 - Equivalent expressions
 - Different algorithms for each operation





Query Evaluation Plan (example)

- An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.





Introduction (Cont.)

- Cost difference between evaluation plans for a query can be enormous
 - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
 1. Generate logically equivalent expressions using **equivalence rules**
 2. Annotate resultant expressions to get alternative query plans
 3. Choose the cheapest plan based on **estimated cost**
- Estimate of plan cost based on:
 - Statistical information about relations. e.g. -
 - ▶ number of tuples, number of distinct values for an attribute
 - Statistics estimation for intermediate results
 - ▶ to compute cost of complex expressions
 - Cost formulae for algorithms, computed using statistics



Transactions



Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
 - Failures of various kinds, such as hardware failures and system crashes
 - Concurrent execution of multiple transactions



ACID Properties of Transactions

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
 - That is, for every pair of transactions T_i and T_j , it appears to T_i that either T_j finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.



Example of Fund Transfer (Transaction)

- Transaction to transfer \$50 from account A to account B:
 1. **read(A)**
 2. $A := A - 50$
 3. **write(A)**
 4. **read(B)**
 5. $B := B + 50$
 6. **write(B)**
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.



Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
 - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
 - However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist despite failures.



Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** -- after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:
 - restart the transaction; can be done only if no internal logical error
 - kill the transaction
- **Committed** – after successful completion.



Transaction State (Cont.)

