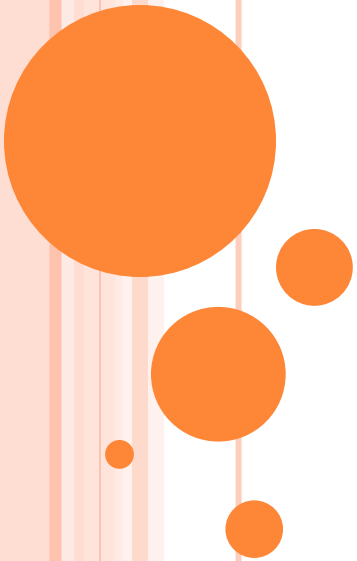


CS354: Database

Storage, File Organization, Indexing & Hashing

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.

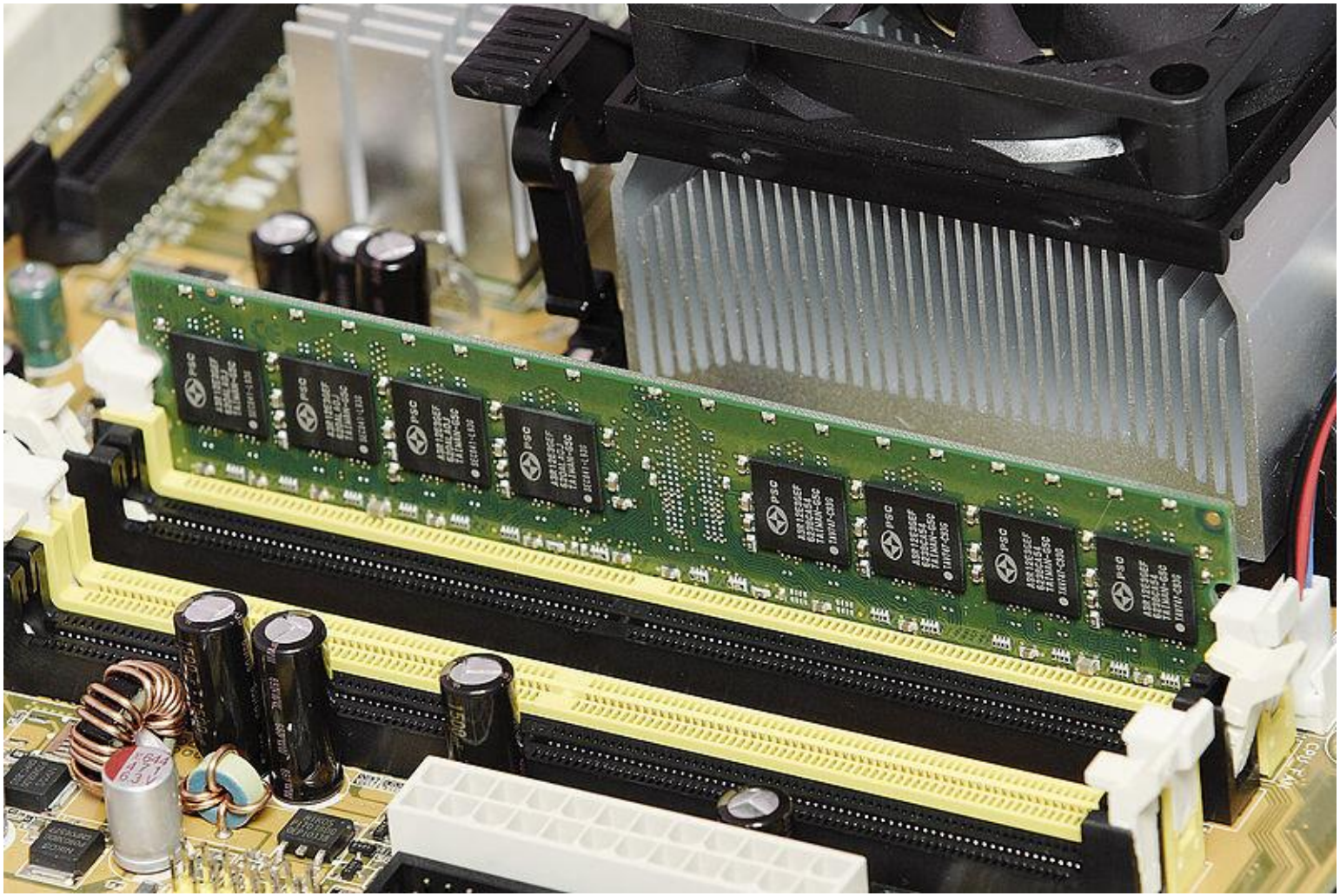
Cache

- Fastest and most costly form of storage
- Volatile in nature
- Managed by the computer system hardware
- This is to compensate the speed difference between the main memory access time and processor logic

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.

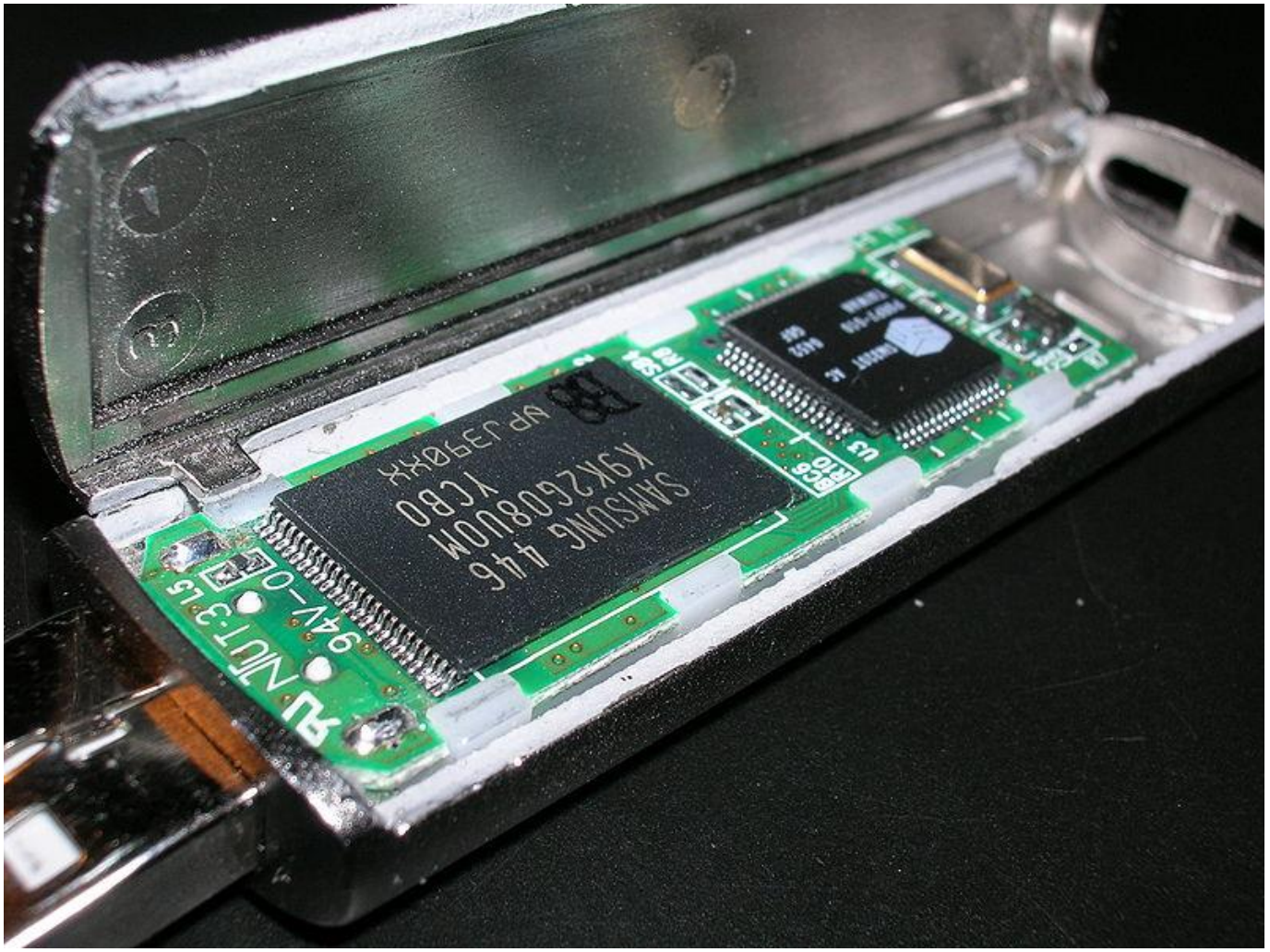
Main Memory



Flash Memory

- Data survives power failure
- Data can be electrically erased and reprogrammed
- Can support only a limited number (around1M) of write/erase cycles.
- 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

Flash Memory



Magnetic Disk

- Much slower access than main memory
- 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
  - **direct-access** – possible to read data on disk in any order, unlike magnetic tape
  - Capacities range up to roughly few TBs
    - Much larger capacity and cost/byte is less 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

Magnetic Disk

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

Optical Storage

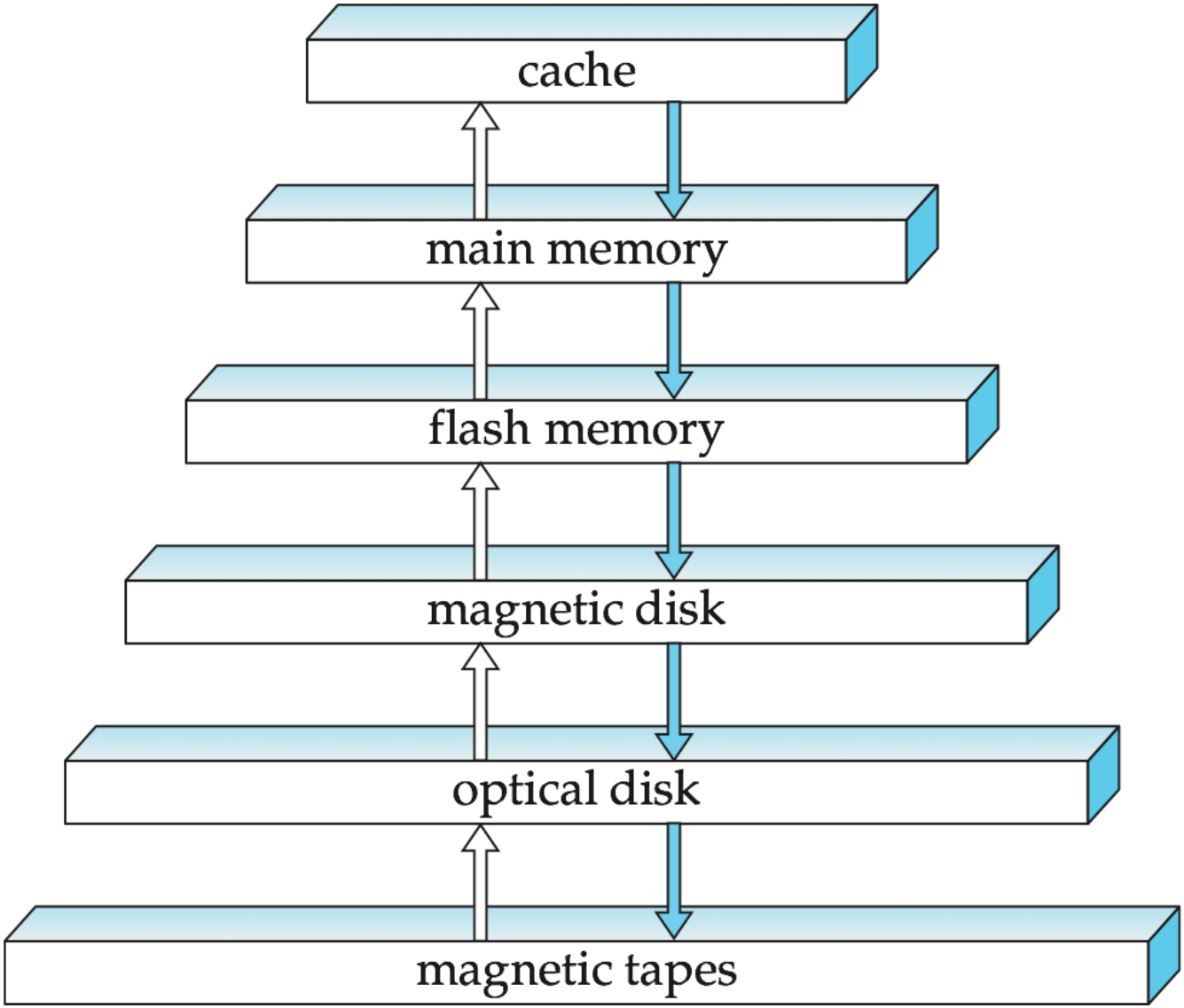
Tape Storage

- 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 ⇒ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
- hundreds of terabytes (1 terabyte = 1000 gigabytes) to even multiple **petabytes** (1 petabyte = 1000 terabytes)

Tape Storage

From Computer Desktop Encyclopedia
© 1999 The Computer Language Co. Inc.

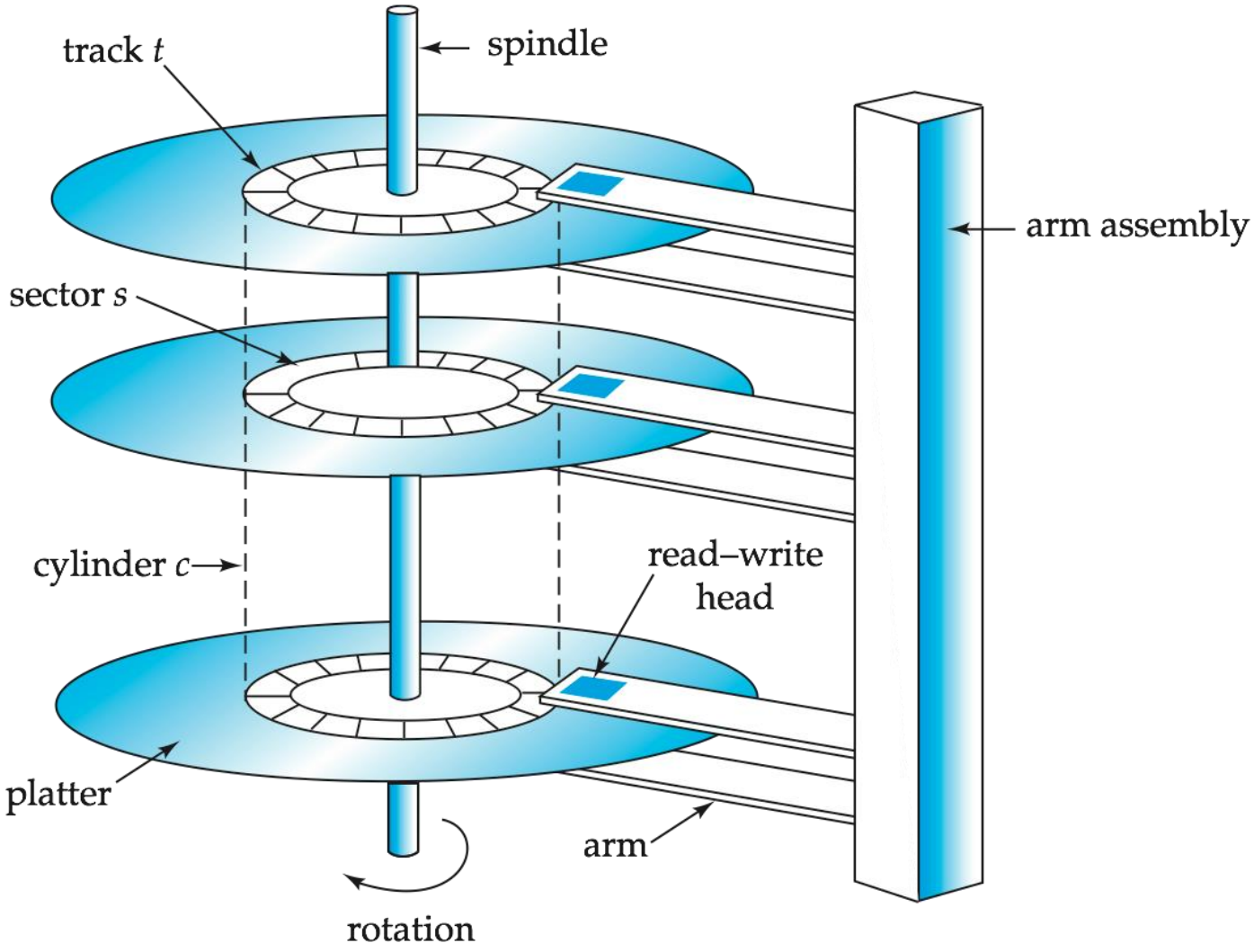
Storage Hierarchy



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
- Multiple disks connected to a computer system through a controller

Disk block access

- Requests for disk I/O are generated both by the file system and by the virtual memory manager
- Each request specifies the address on the disk to be referenced
  - This address is in the form of *block number*
- A **block** is a logical unit consisting of a fixed number of contiguous sectors
  - It may range from 512 bytes to several kilobytes
- Data are transferred between disk and main memory in units of blocks

Performance Measures of Disks

- The main measures of the qualities of a disk are
  - Capacity
  - Access time
  - Data transfer rate
  - Reliability

Access Time

- It is the time when a read/write request is issued to when data transfer begins
- It is also sum of *seek time* and *rotational latency time*
- **Seek time:** the time for repositioning the arm under correct track
  - Typically ranges from 2 to 30 miliseconds
- **Rotational latency time:** the time spent waiting for the sector to be appeared under read/write head
  - Typically ranges from 4 to 11.1 miliseconds per rotation
  - On an average half rotation is required for the beginning of the desired sector to appear under the head

Data Transfer Rate

- The rate at which the data can be retrieved from or stored to the disk
- Data transfer begins when the first sector of the data to be accessed has come under head
- Current disk systems claim to support maximum transfer rate of 25 to 40 megabytes per second

MTTF: Mean Time To Failure

- A measure of the reliability of the disk
- It is the amount of time that, on average, the system runs continuously without any failure
- According to vendors, the MTTF of disks ranges from 30,000 hrs to 1,200,000 hrs i.e., about 3.4 to 136 years

Redundant Array of Independent Disks (RAID)

- 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

- 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 performance, rather than for economic reasons.
    - 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 is 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

- **Mean time to data loss** depends on **mean time to failure**, and **mean time to repair**
- Power failure, natural disasters such as earthquake, fire, floods may result in the damage of both disks at the same time

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
  - Block level striping

Bit-level striping

- **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

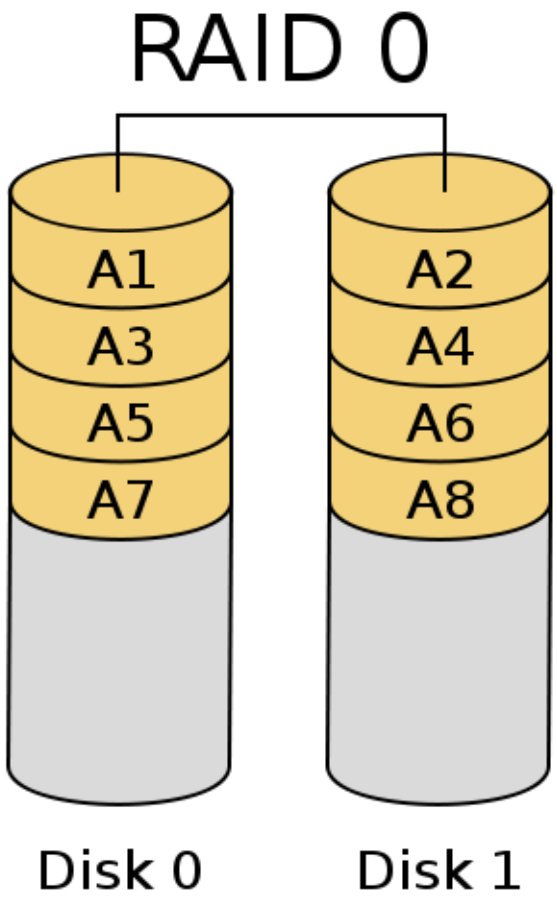
- **Block-level striping** – with n disks, block i of a file goes to disk (i mod n) + 1
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

RAID Levels

- Mirroring provides high reliability but it is expensive
- Striping provides high data transfer rates, but does not improve reliability
- 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

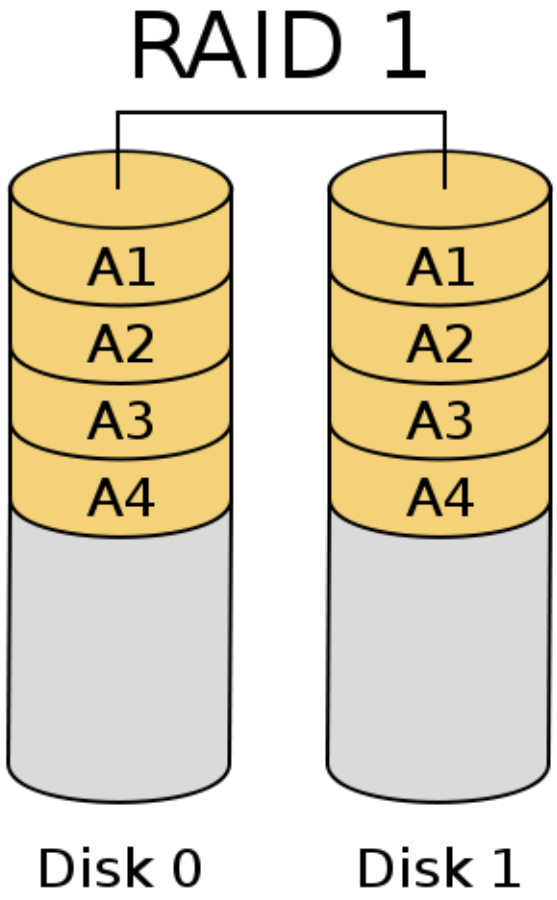
RAID 0

- **RAID Level 0**: Block striping; non-redundant.
  - Data are split up in blocks that get written across all the drives in the array
  - It is ideal for non-critical storage of data that have to be read/written at a high speed



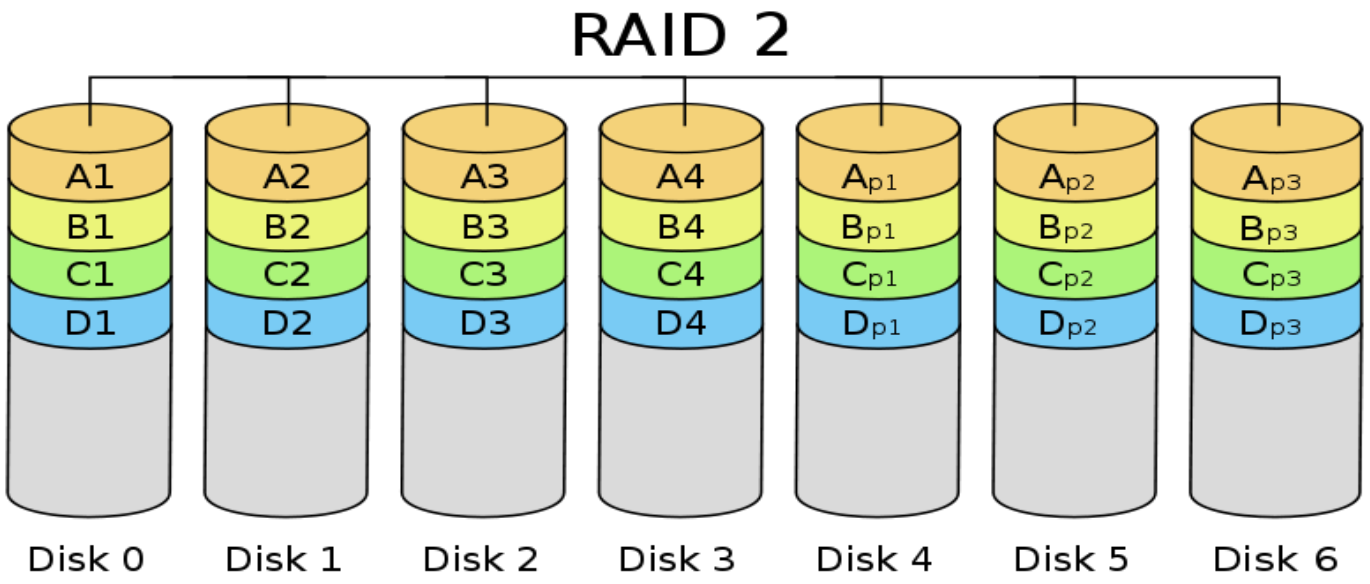
RAID 1

- **RAID Level 1**: Mirrored disks
- Data are stored twice by writing them to the both the data disk(s) and a mirror disk(s)
- **RAID Level 1+0 or 10:** if RAID 1 is combined with RAID 0 to improve performance
- RAID-1 is ideal for mission critical storage, for instance for accounting systems. It is also suitable for small servers in which only two disks will be used.
- Popular for applications such as storing log files in a database system.



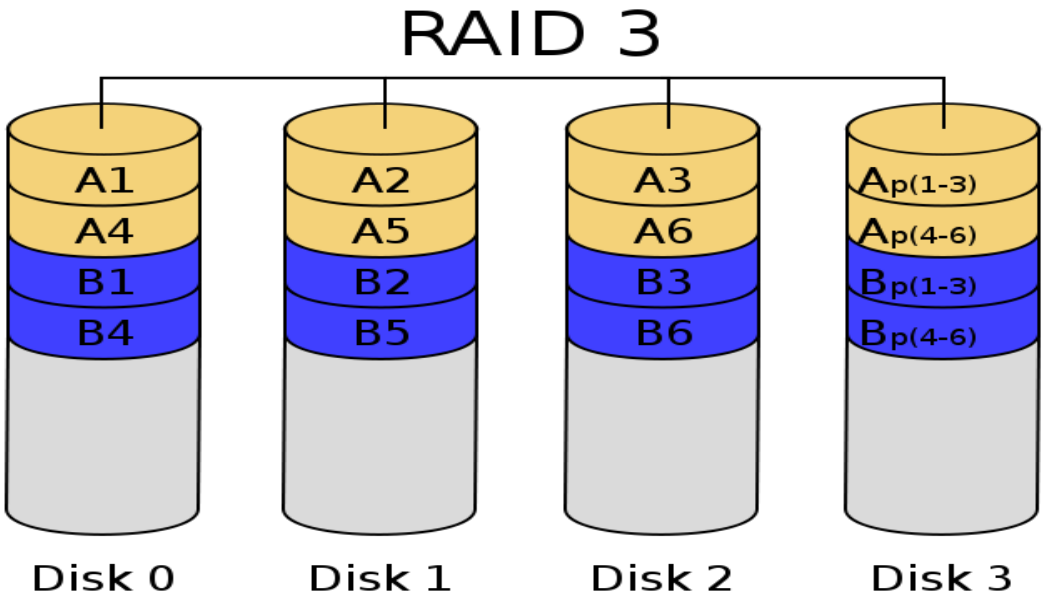
RAID 2

- **RAID Level 2**: Memory-Style Error-Correcting-Codes (ECC) with bit striping.



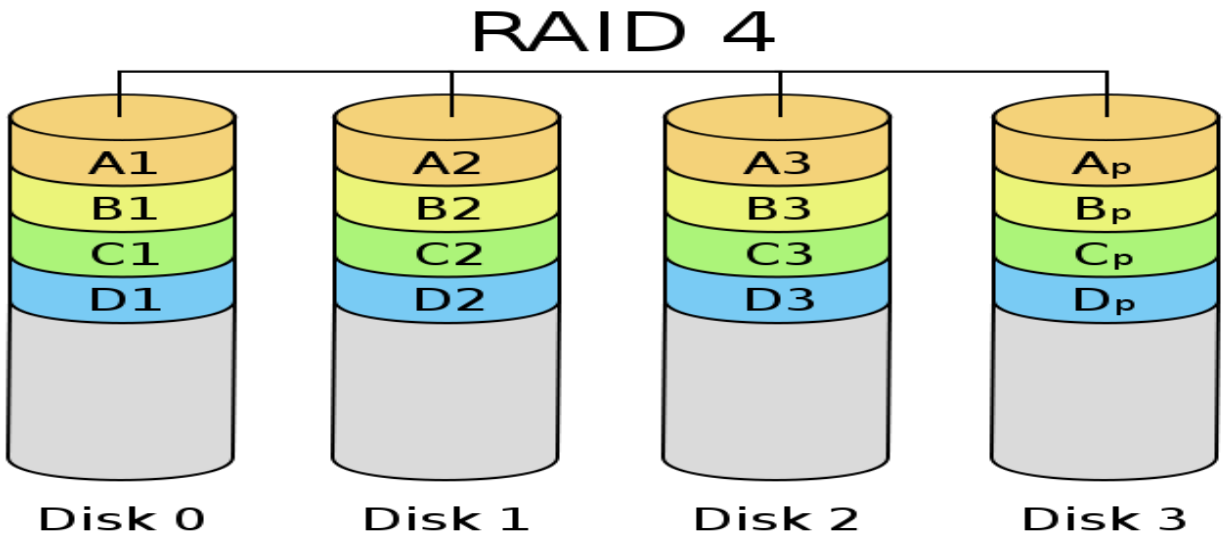
RAID 3

- **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)



RAID 4

- **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.

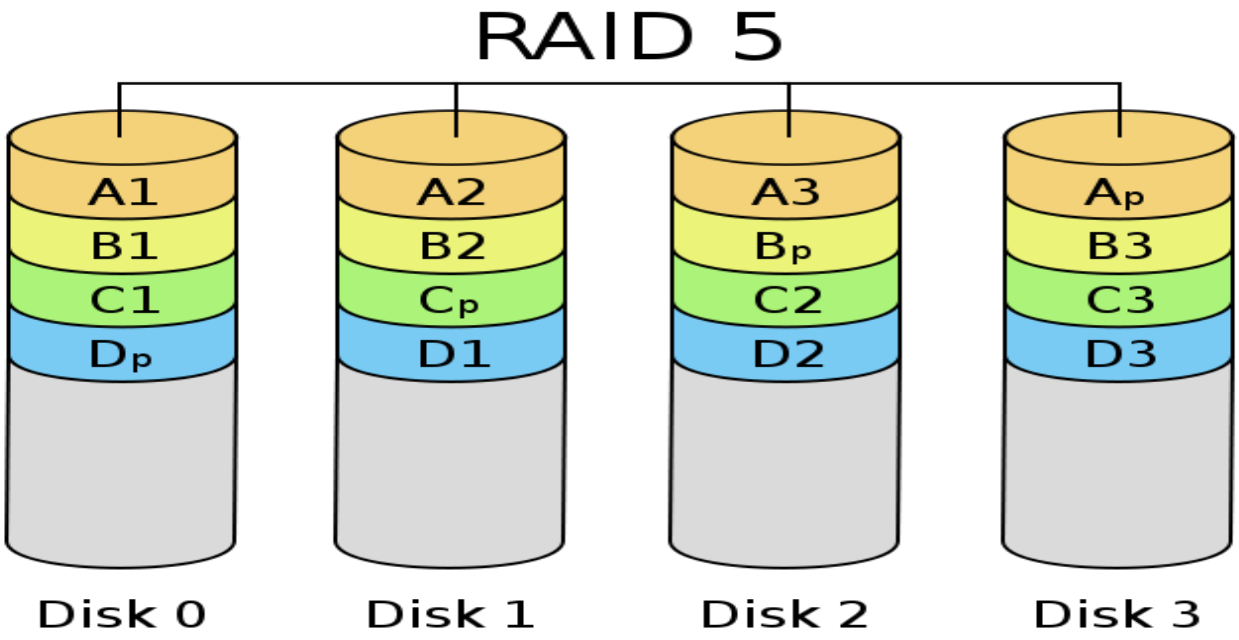


RAID Levels (Cont.)

- **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 5

- **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.

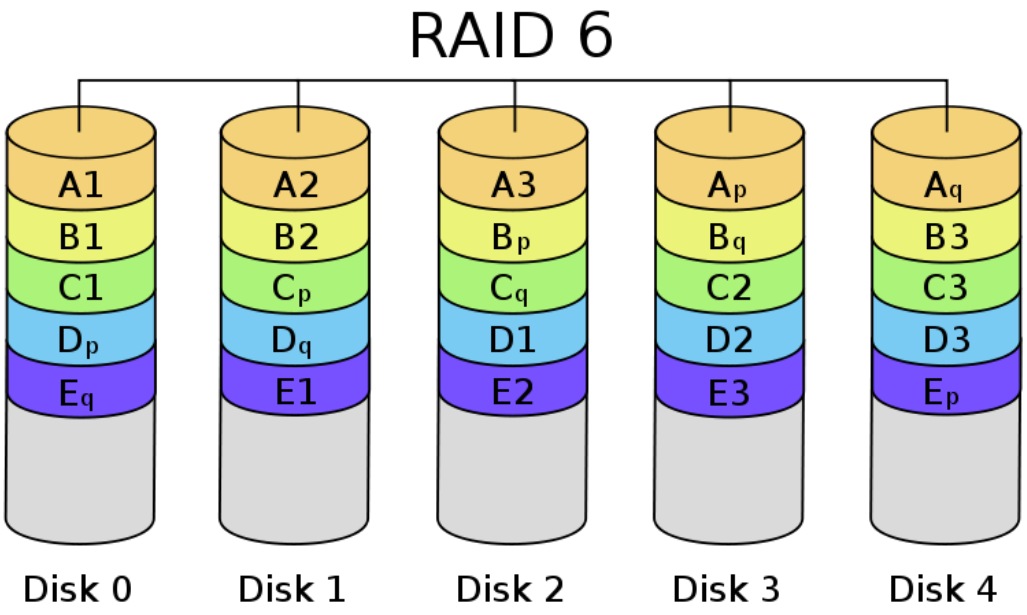


RAID 5 (CONTD.)

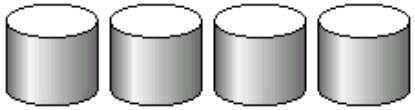
- Higher I/O rates than Level 4.
  - Block writes occur in parallel if the blocks and their parity blocks are on different disks.
- Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.

RAID 6

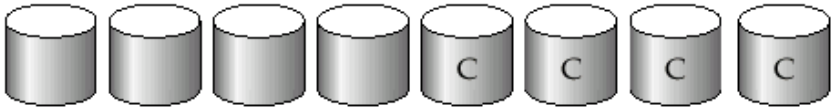
- **RAID Level 6**: P+Q Redundancy scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.
  - Better reliability than Level 5 at a higher cost; not used as widely.



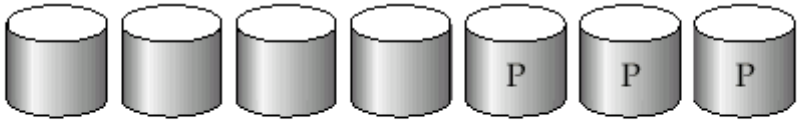
RAID Levels at a glance



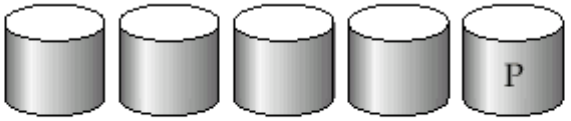
(a) RAID 0: nonredundant striping



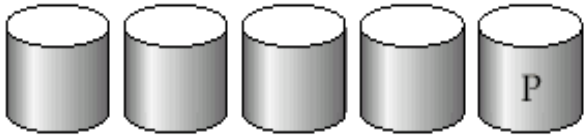
(b) RAID 1: mirrored disks



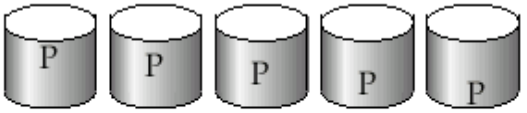
(c) RAID 2: memory-style error-correcting codes



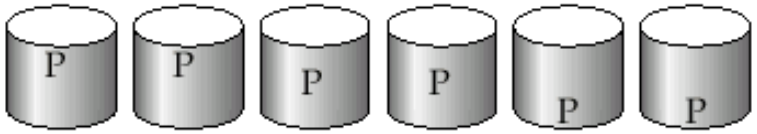
(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(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
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk

Optimization of Disk-Block Access

- **Buffering:** in-memory buffer to cache disk blocks
- **Read-ahead:** Read extra blocks from a track in anticipation that they will be requested soon
- **Disk-arm-scheduling** algorithms re-order block requests so that disk arm movement is minimized
  - **elevator algorithm**

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*.
- The records are mapped to disk blocks
- A block may contain several records
- Blocks are unit of storage allocation and data transfer
- Generally, most of the records are smaller than a block
  - Exceptions are image, video, etc.
- Each record is entirely contained in a single block
  - No record is partly in one block and partly in another block

- In relational database, tuples of distinct relations are generally of different sizes
- 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

- Let's consider a file of instructor records for University database system
  - ***type*** *instructor* = ***record***

    ID varchar(5);

    name varchar(20);

    dept_name varchar(20);

    salary numeric(8,2);

    end*;*
- Each character occupies 1 byte and numeric occupies 8 byte
- Thus each instructor record is 53 bytes long

Fixed-Length Records

- Simple approach:
  - Use first n bytes for first record and next n bytes for next record 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:*
  What are the options?

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

move records $i + 1, \ldots, n$ to $i, \ldots, n - 1$

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

All these records shifted one place up

Deleting record 3 and moving last record

move record n to i

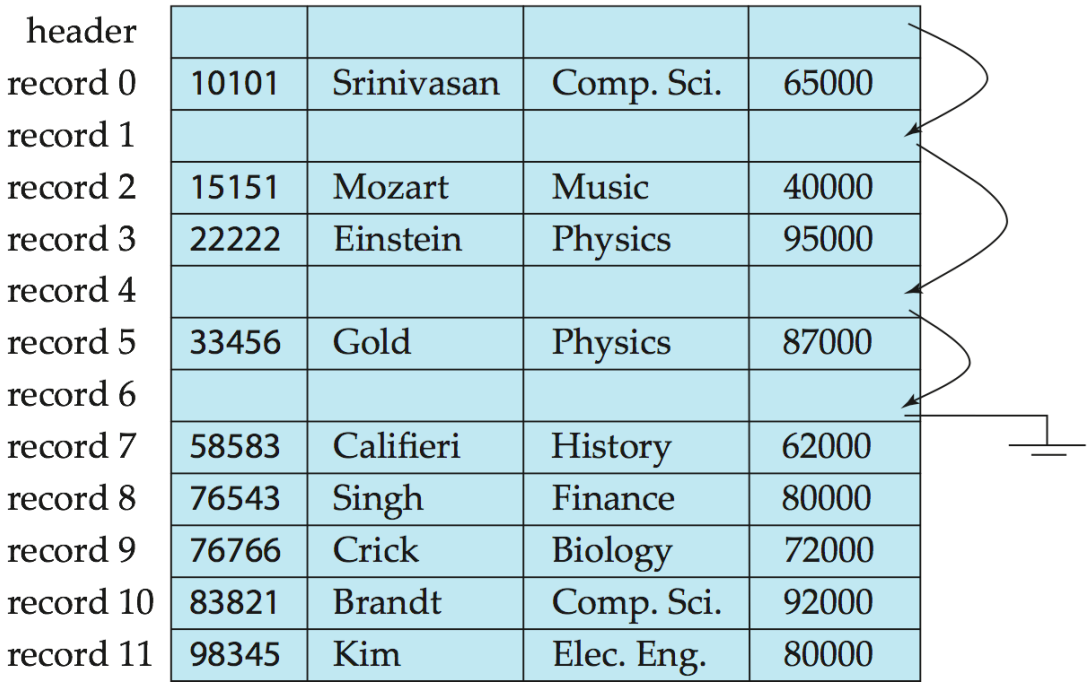
record 0	10101	Srinivasan	Comp. Sci.	65000
record 1	12121	Wu	Finance	90000
record 2	15151	Mozart	Music	40000
record 11	98345	Kim	Elec. Eng.	80000
record 4	32343	El Said	History	60000
record 5	33456	Gold	Physics	87000
record 6	45565	Katz	Comp. Sci.	75000
record 7	58583	Califieri	History	62000
record 8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

Record 11 is only moved up to fill the empty slot

Free Lists

do not move records, but link all free records on a *free list*

- 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

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).
- Variable length representation
- Fixed length representation

Byte String Representation

- A simple variable length representation method
- Attaches a special *end of record* (⊥) symbol to the end of each record
- Each record is stored as a string of consecutive bytes
- Let's consider the following account information
  - **type** account = **record**

```
Branch_name:char(22)
Account_info: array[1..infinity] of
                record;
                        account_no: char(10);
                        balance: real;
                end
end
```

Byte String Representation

Perryridge	A-102	400	A-201	900	A-218	700	⊥
Round Hill	A-305	350	⊥				
Mianus	A-215	700	⊥				
Downtown	A-101	500	A-110	600	⊥		
Redwood	A-222	700	⊥				
Brighton	A-217	750	⊥				

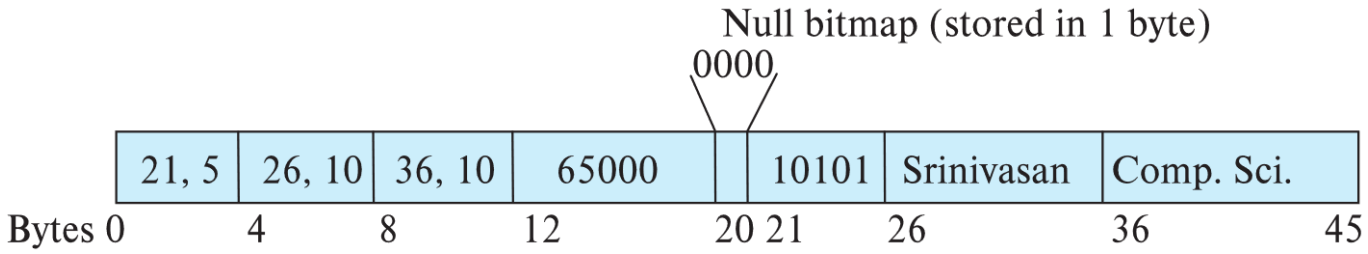
Problems

Not easy to reuse space occupied by the deleted record

No space for records to grow longer

Variable-Length Records (Offset and length)

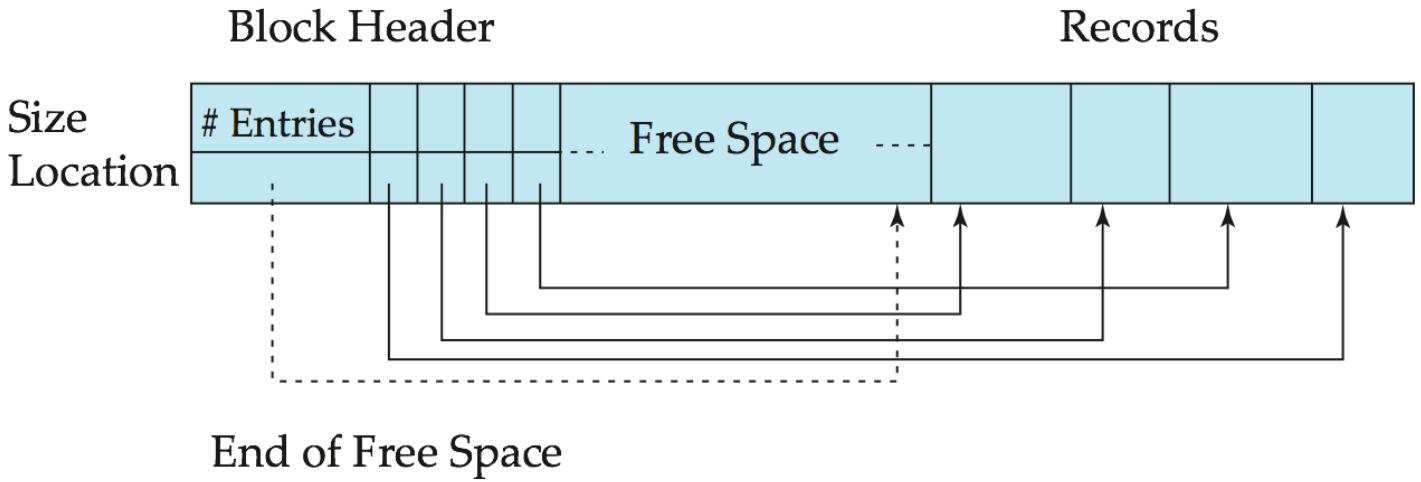
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Slotted Page Structure

- *Slotted page structure* is commonly used for organizing variable length records within a block

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.

Fixed Length Representation

Using reserved space method

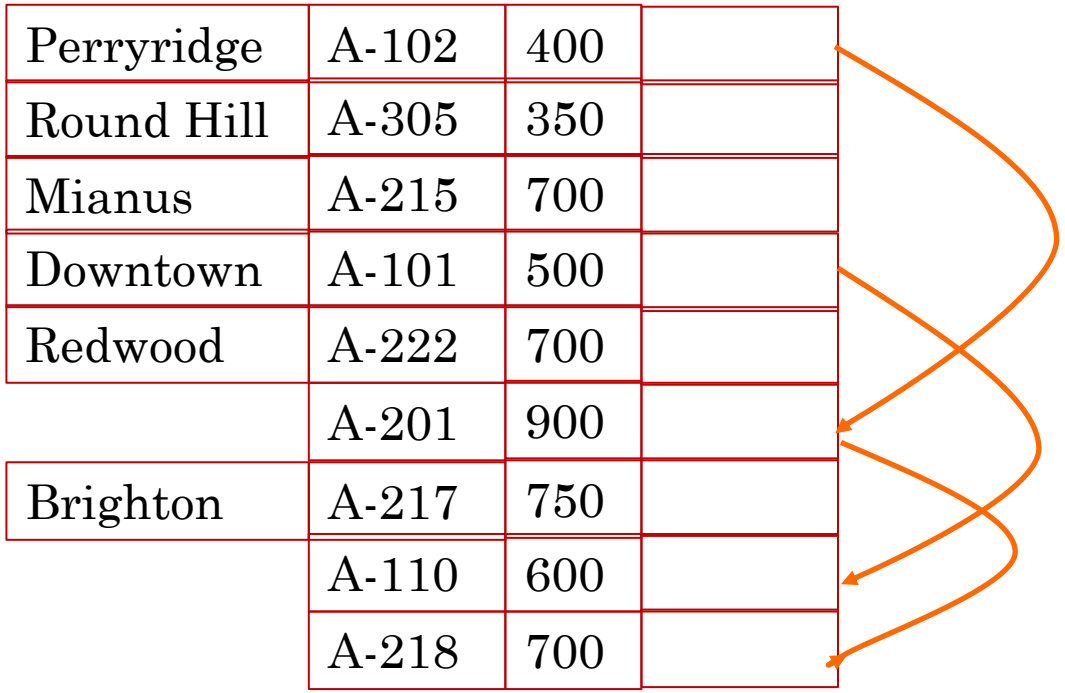
Perryridge	A-102	400	A-201	900	A-218	700
Round Hill	A-305	350	⊥	⊥	⊥	⊥
Mianus	A-215	700	⊥	⊥	⊥	⊥
Downtown	A-101	500	A-110	600	⊥	⊥
Redwood	A-222	700	⊥	⊥	⊥	⊥
Brighton	A-217	750	⊥	⊥	⊥	⊥

If there is a maximum record length that is never exceeded, then this scheme can be used

Unused space shorter than the maximum length is filled with a special null or end of character

Another Fixed Length Representation

Using list representation



Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
	A-201	900	
Brighton	A-217	750	
	A-110	600	
	A-218	700	

The records are chained together by pointers

Wastage of space except the first record in the chain

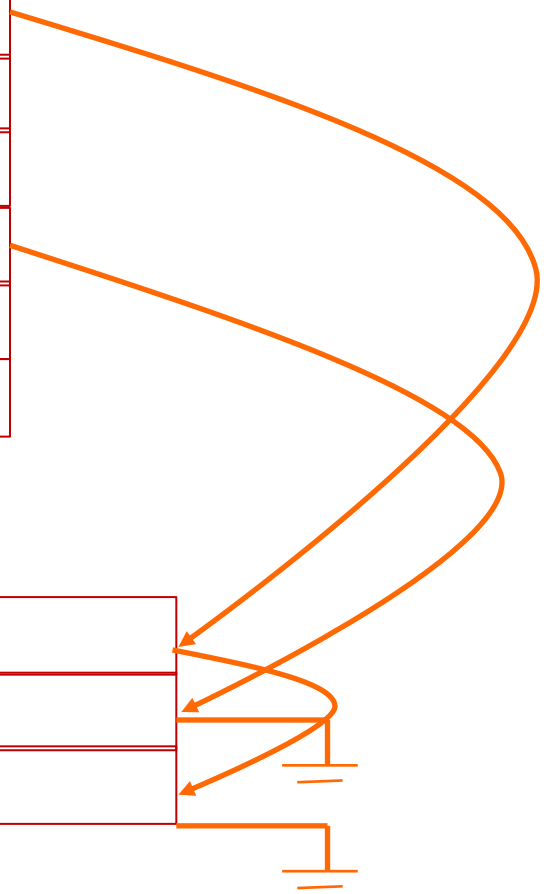
Alternate list Representation

Anchor block

Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

Overflow block

A-201	900	
A-110	600	
A-218	700	



- We have seen how records are represented in a file structure
- A relation is a set of records
- How to organize them in a file?

Organization of Records in Files

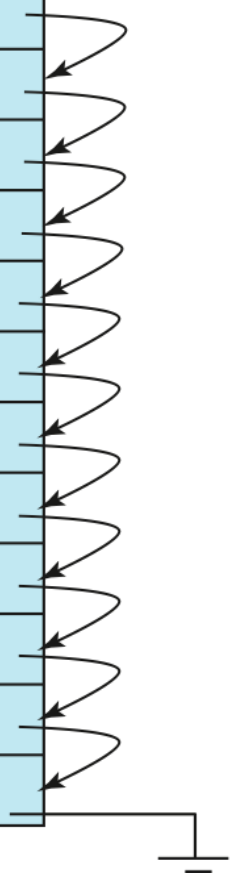
- **Heap** – a record can be placed anywhere in the file where there is space
- **Sequential** – store records in sequential order, based on the value of the search key of each record
- **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- Generally, records of each relation are stored in a separate file. However, 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

- Designed for efficient processing of records in sorted order based on some search key
- **Search key** is
  - any attribute or set of attributes
  - It need not be the primary key or superkey
  - Used for fast retrieval of records in search key order
- The records are linked together by pointers
- The pointer in each record points to the next record in search key order
- To minimize the no. of block accesses the records are stored physically in search key order or as close to search key order as possible

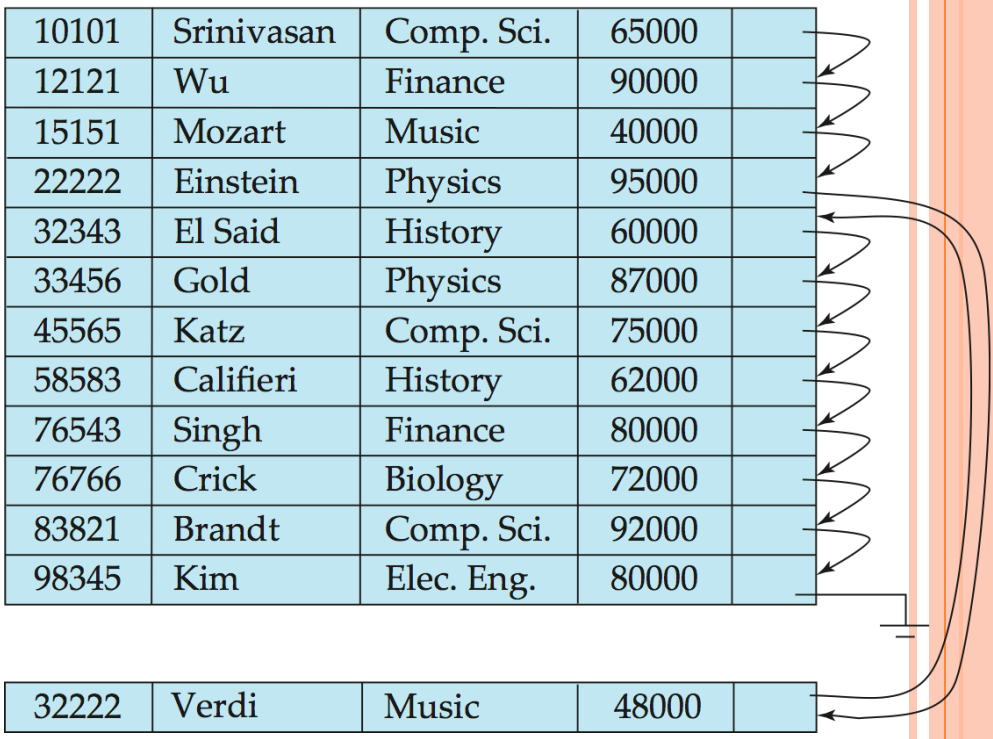
EXAMPLE

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 sequential order



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	

32222	Verdi	Music	48000	

Multitable Clustering File Organization

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

department

dept_name	*building*	*budget*
Comp. Sci.	Taylor	100000
Physics	Watson	70000

instructor

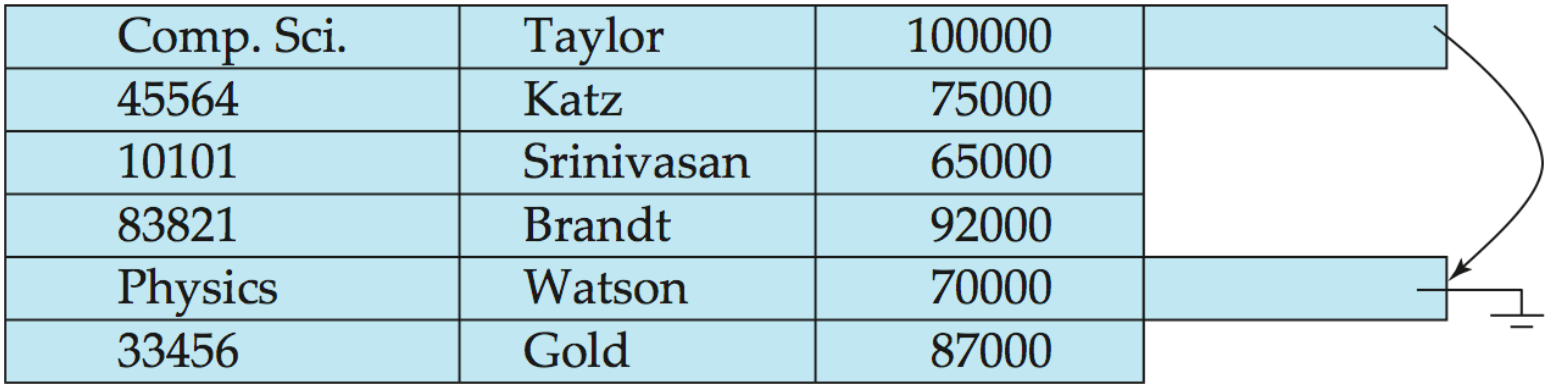
ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000

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	

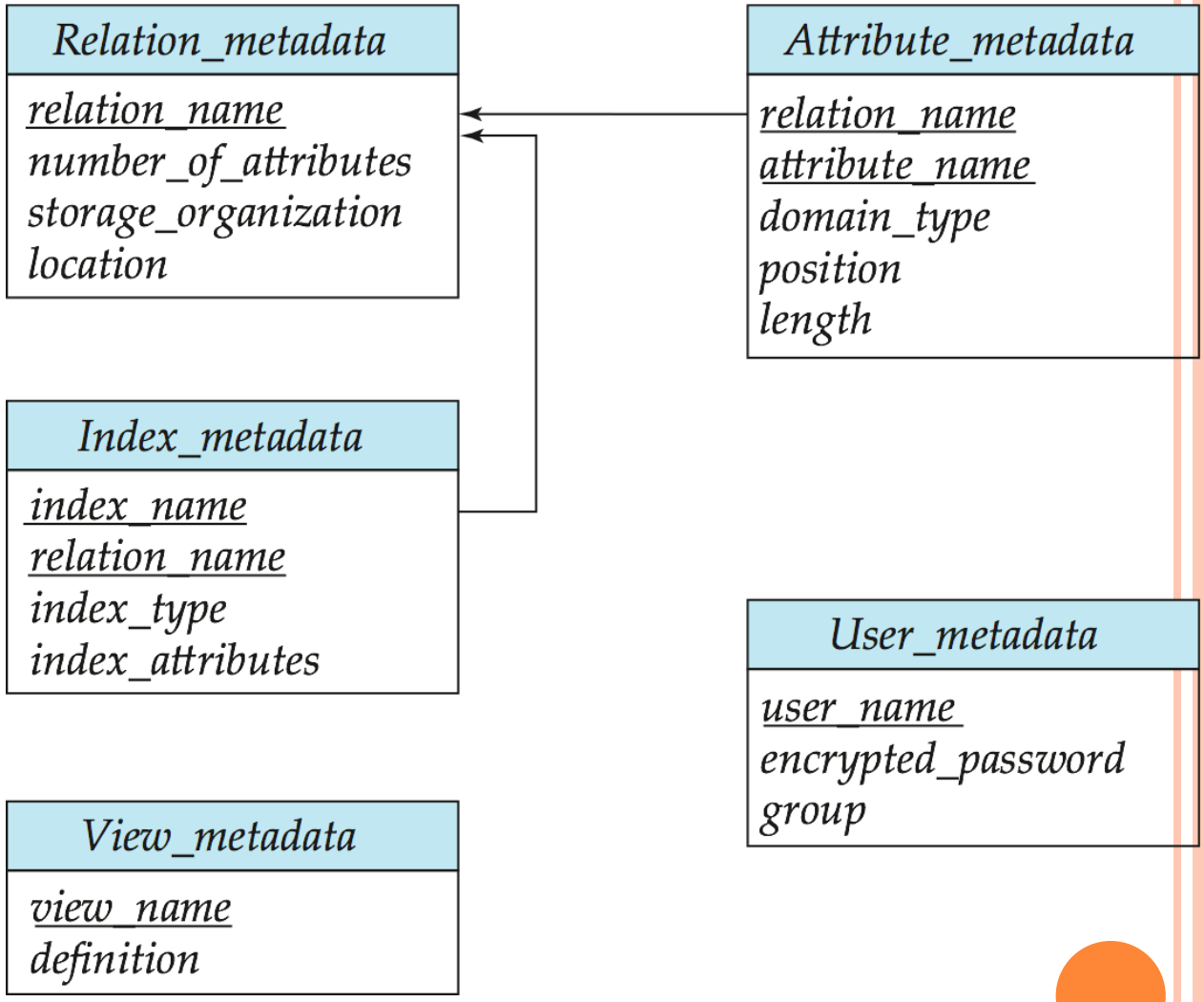
Data Dictionary Storage

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

Relational Representation of System Metadata

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



Indexing and Hashing: 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

- 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

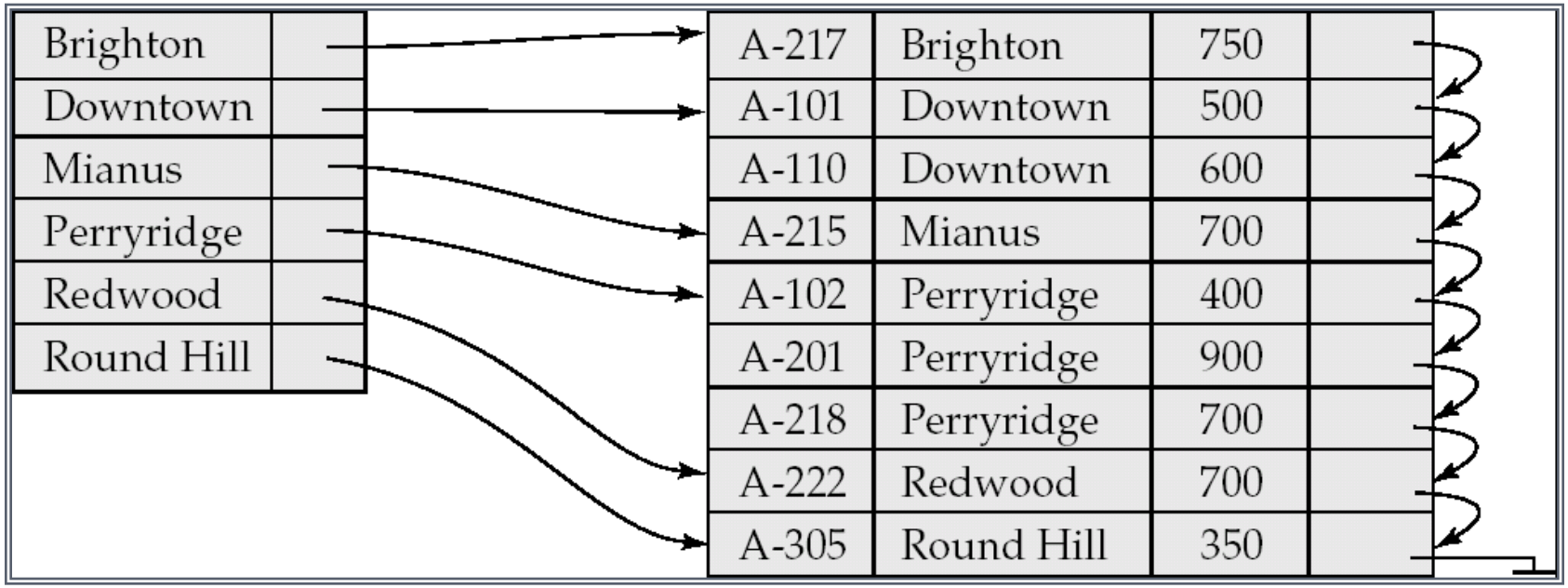
- Access types supported efficiently
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values (e.g. 10000 < *salary* < 40000)
- Access time
  - The time it takes to find a particular data item, or set of items
- Insertion time includes
  - The time it takes to find the correct place to insert new data item
  - The time it takes to update the index structure
- Deletion time includes
  - The time it takes to find the item to be deleted
  - The time to update the index structure
- Space overhead
  - The additional space occupied by an index structure
  - Usually worthwhile to sacrifice some space to achieve improved performance

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.
- Types-
  - Dense
  - Sparse

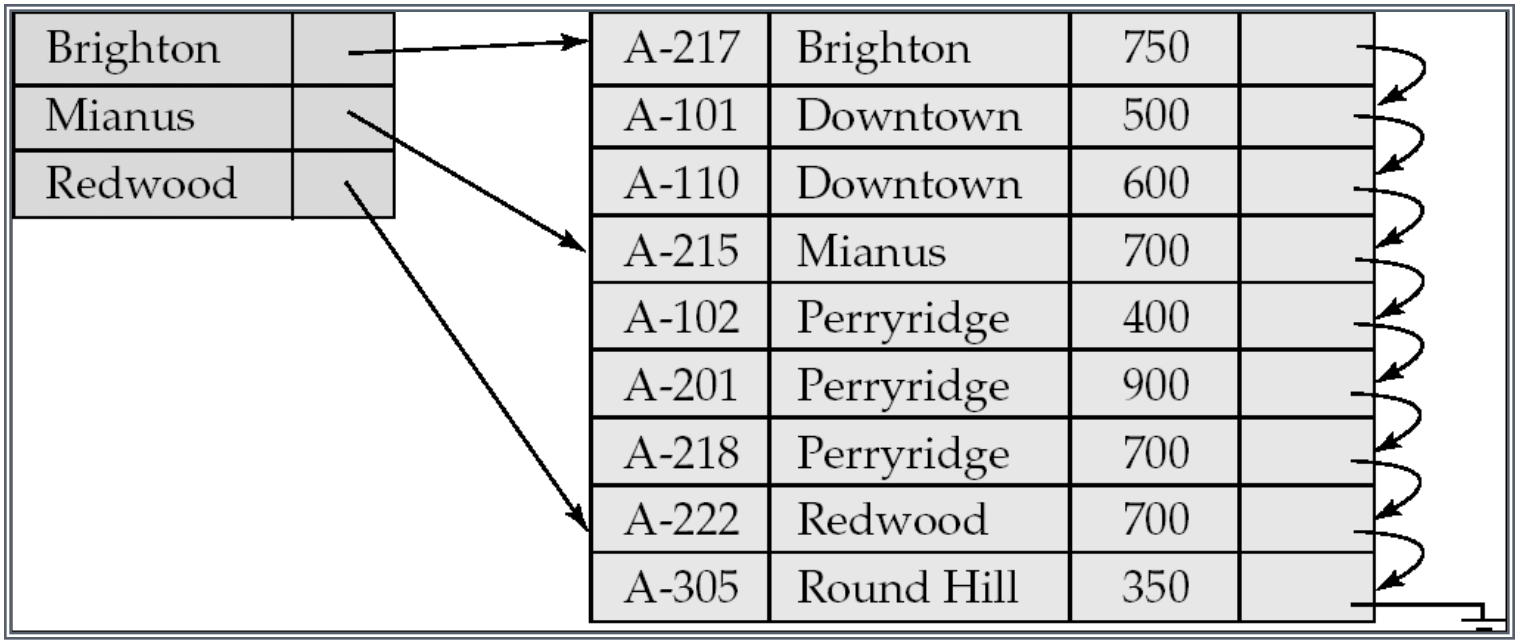
Dense Index Files

- Dense index — Index record appears for every search-key value in the file.



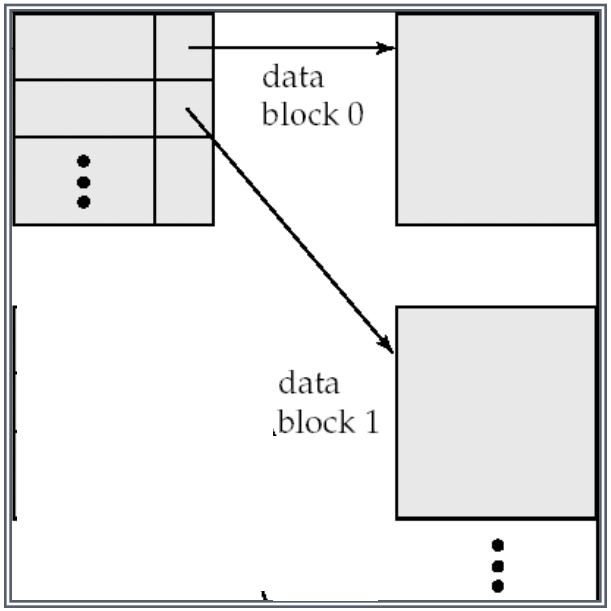
Sparse Index Files

- **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.)

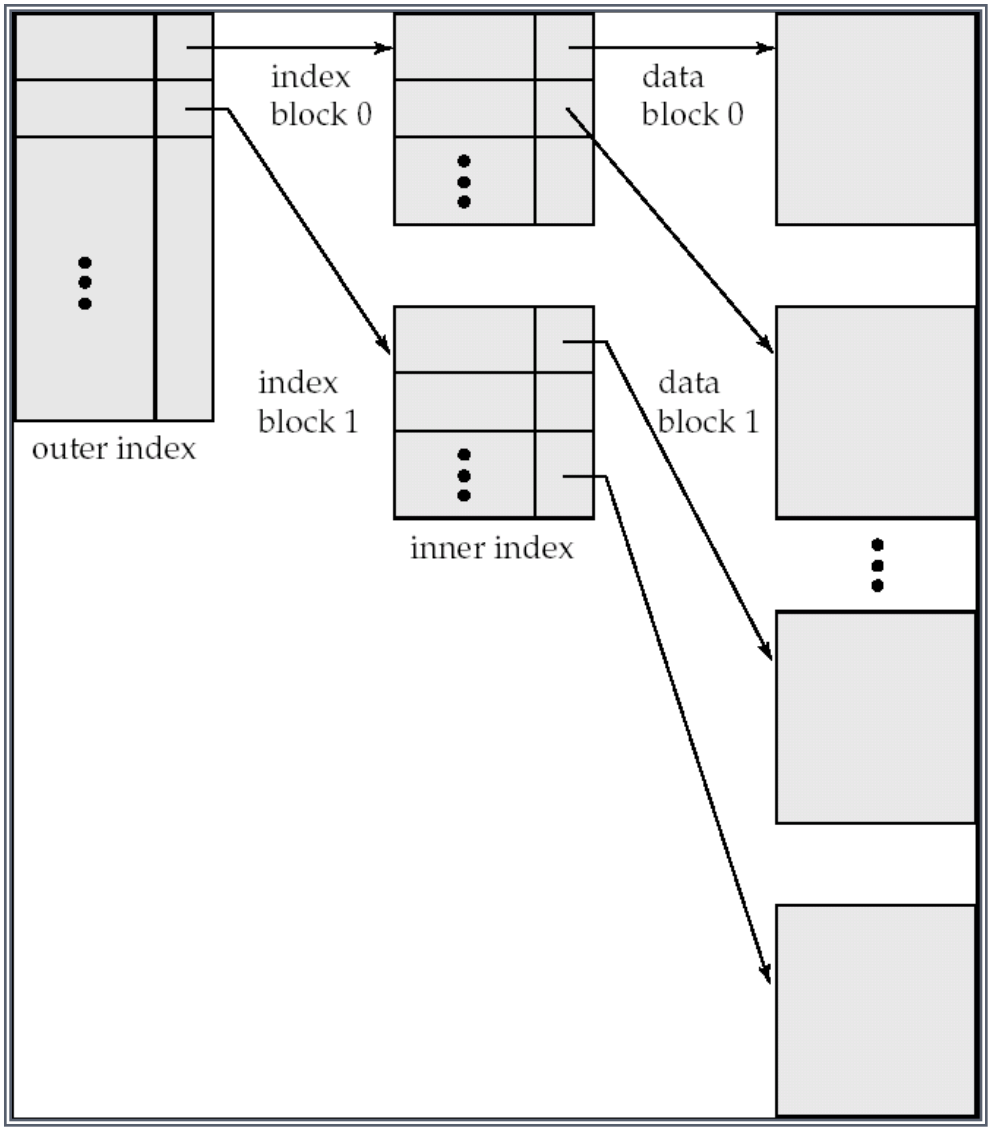
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff**: sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



Multilevel Index

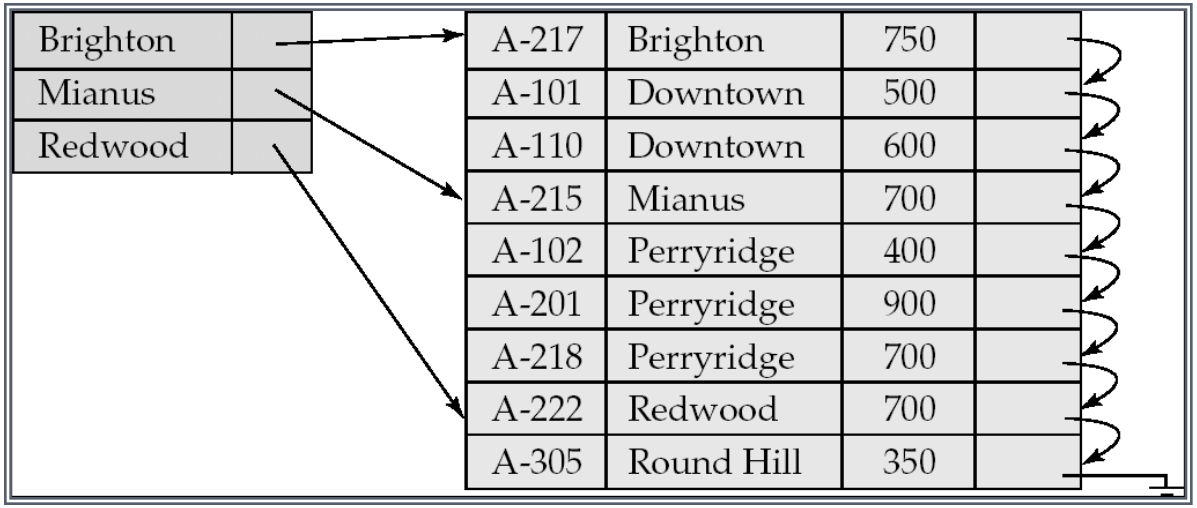
- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit 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.)



Index Update: Record Deletion

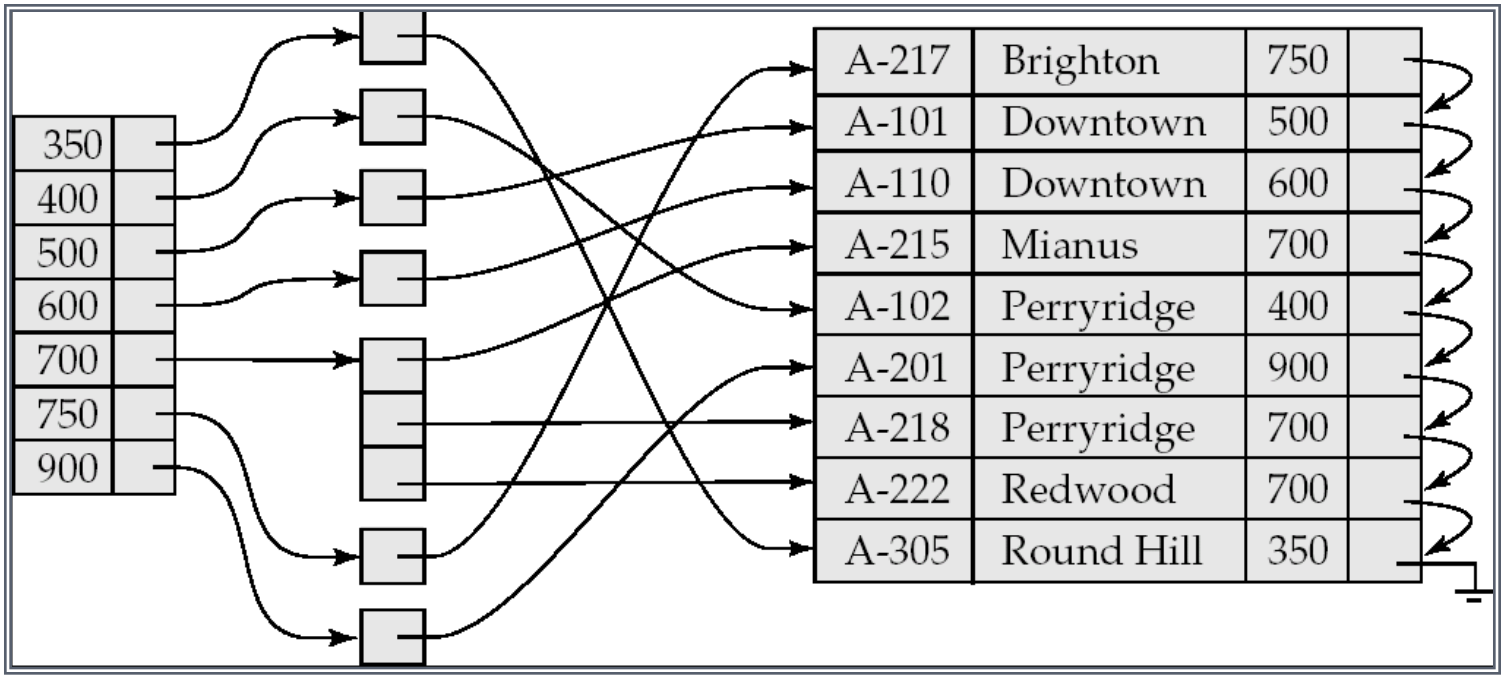
- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - **Dense indices** – deletion of search-key: similar to file record deletion.
  - **Sparse indices** –
    - if deleted key value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



Index Update: Record Insertion

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

Secondary Indices Example



Secondary index on *balance* field of *account*

- 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

Indices on Multiple Keys

- A search key containing more than one attribute is referred as a composite search key
- If the index attributes are A1, …, An then the tuple of values can be represented of the form (a1,…,an)
- The ordering of search key is lexicographic ordering
- For example: for the case of two attributes search keys, (a1,a2)<(b1,b2) if either a1<b1 or a1=b1 and a2<b2

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 micro seconds, versus about 100 nanoseconds for memory access

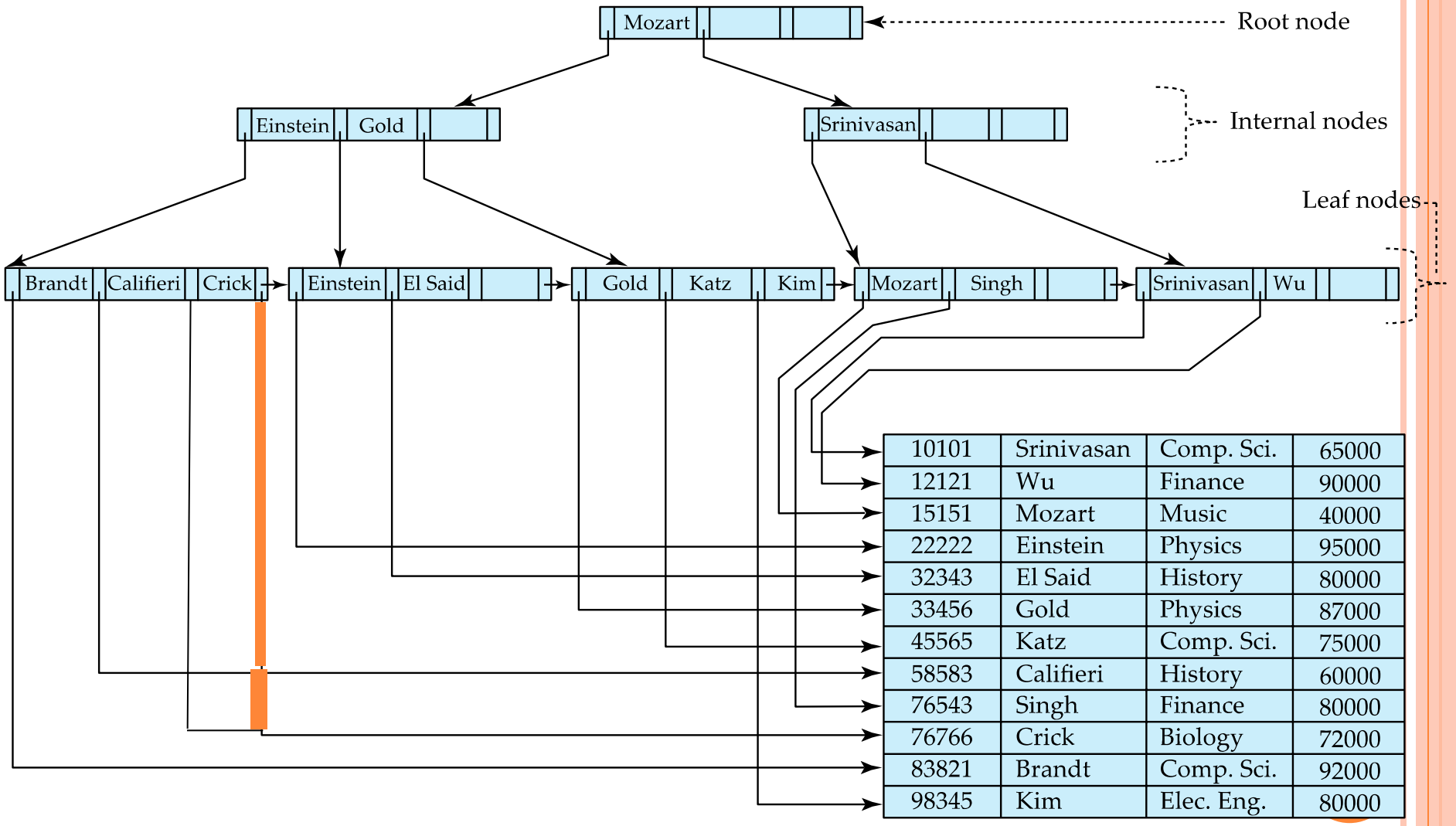
Indexed Sequential File

- Disadvantage of indexed-sequential files
  - Performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.

B^+-Tree Index 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



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

B+-Tree Index Files (Cont.)

A B+-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
- A leaf node has between $\lceil (n–1)/2 \rceil$ and $n–1$ values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and $(n–1)$ values.

B+-Tree Node Structure

- Typical node

  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

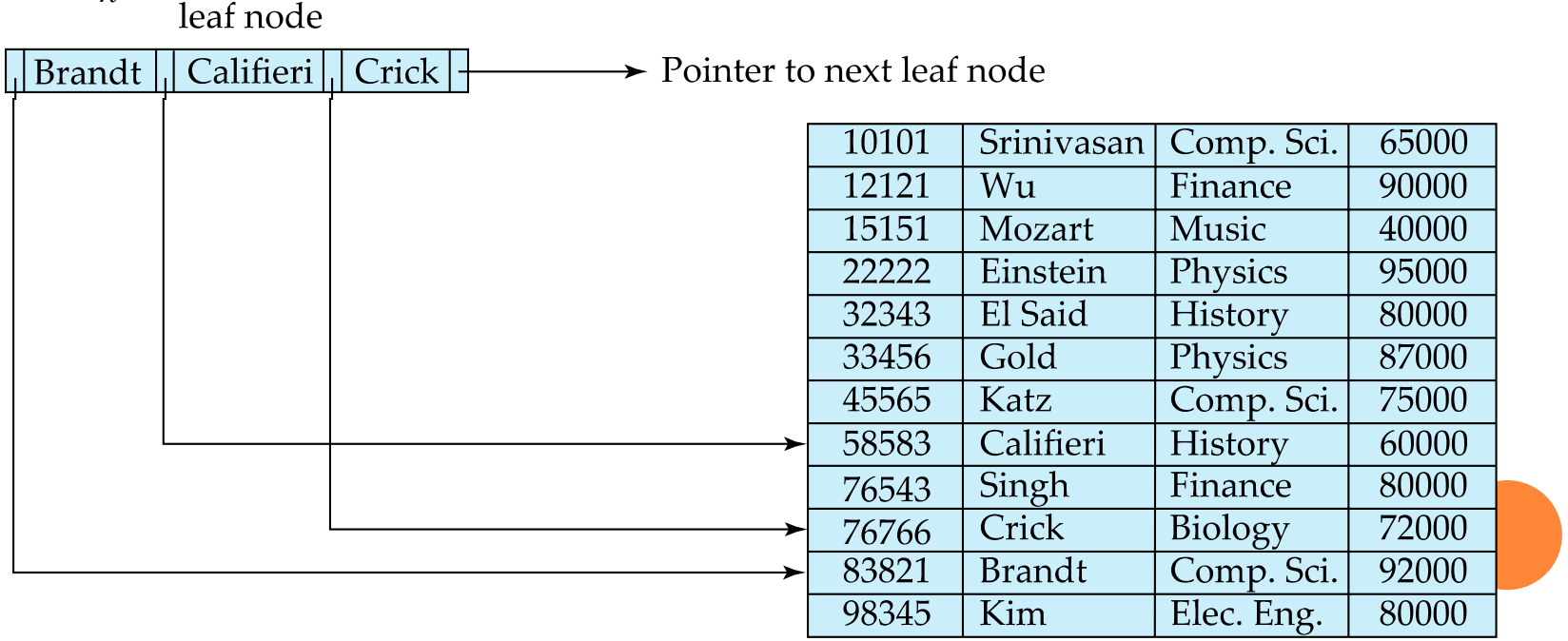
$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

(Initially assume no duplicate keys, address duplicates later)

Leaf Nodes in B+-Trees

Properties of a leaf node:

- For $i = 1, 2, \ldots, n-1$, pointer P_i points to a file record with search-key value K_i,
- If L_i, L_j are leaf nodes and $i < j$, L_i's search-key values are less than or equal to L_j's search-key values
- P_n points to next leaf node in search-key order



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

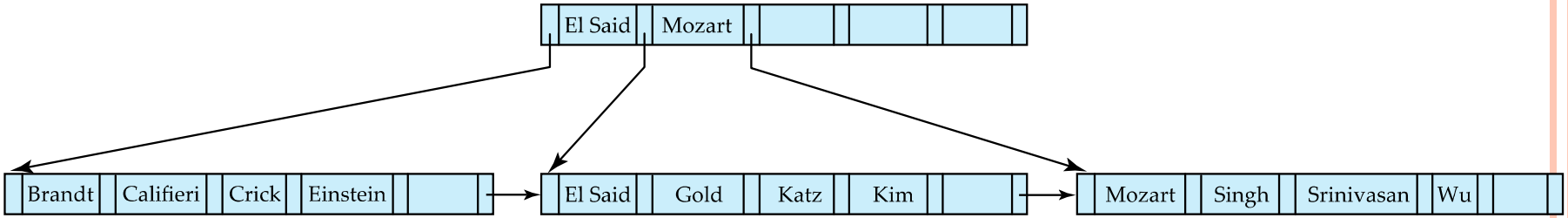
Non-Leaf Nodes in B+-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with m pointers:
  - All the search-keys in the subtree to which P_1 points are less than K_1
  - For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_i
  - All the search-keys in the subtree to which P_n points have values greater than or equal to K_{n-1}
  - General structure

P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

Example of B+-tree

- B+-tree for *instructor* file ($n = 6$)



- Leaf nodes must have between 3 and 5 values ($\lceil (n–1)/2 \rceil$ and n –1, with $n = 6$).
- Non-leaf nodes other than root must have between 3 and 6 children ($\lceil (n/2 \rceil$ and n with $n =6$).
- Root must have at least 2 children.

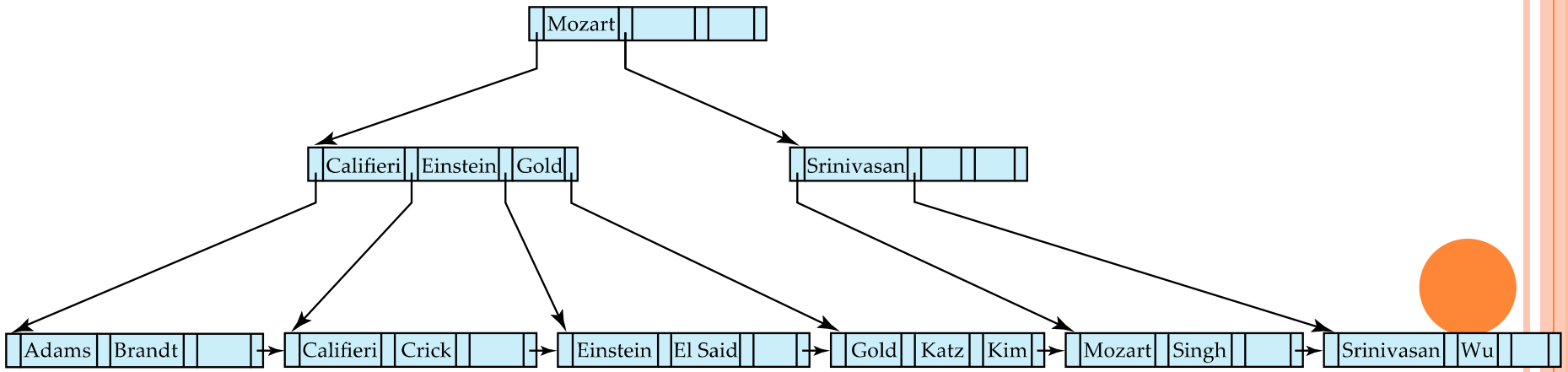
Observations about B^+-trees

- Since the inter-node connections are done by pointers, "logically" close blocks need not be "physically" close.
- The non-leaf levels of the B^+-tree form a hierarchy of sparse indices.
- The B^+-tree contains a relatively small number of levels
  - If there are K search-key values in the file, the tree height is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

Queries on B+-Trees

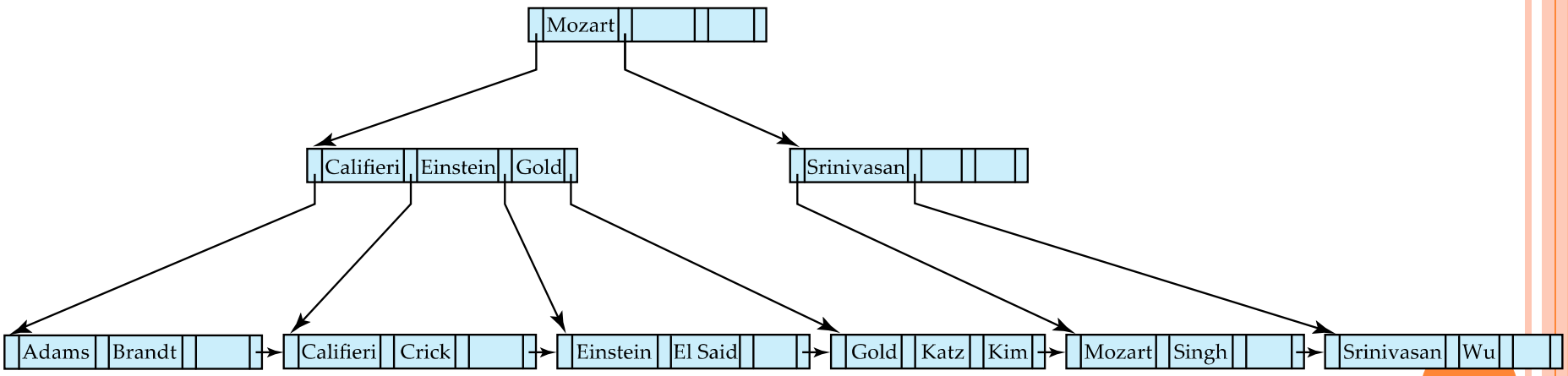
function *find*(*v*)

1. *C=root*
2. **while** (C is not a leaf node)
   1. Let *i* be least number s.t. $V \leq K_i$.
   2. **if** there is no such number *i then*
   3. Set *C* = *last non-null pointer in C*
   4. **else if** (v = C.K_i) Set C = P_{i+1}
   5. **else set** *C* = C.P_i
3. **if** for some *i*, $K_i = V$ **then** return C.P_i
4. **else** return null /* no record with search-key value *v* exists. */



Queries on B+-Trees (Cont.)

- **Range queries** find all records with search key values in a given range
  - See book for details of **function** *findRange*(*lb*, *ub*) which returns set of all such records
  - Real implementations usually provide an iterator interface to fetch matching records one at a time, using a *next*() function



Queries on B+-Trees (Cont.)

- If there are K search-key values in the file, the height of the tree is no more than $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and n is typically around 200 (with search key as 12 bytes and pointer size as 8 bytes).
- With 1 million search key values and $n = 100$
  - at most $log_{50}(1{,}000{,}000) = 4$ nodes are accessed in a lookup traversal from root to leaf.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

Non-Unique Keys

- If a search key a_i is not unique, create instead an index on a composite key (a_i , A_p), which is unique
  - A_p could be a primary key, record ID, or any other attribute that guarantees uniqueness
- Search for $a_i = v$ can be implemented by a range search on composite key, with range $(v, -\infty)$ to $(v, +\infty)$
- But more I/O operations are needed to fetch the actual records
  - If the index is clustering, all accesses are sequential
  - If the index is non-clustering, each record access may need an I/O operation

Updates on B+-Trees: Insertion

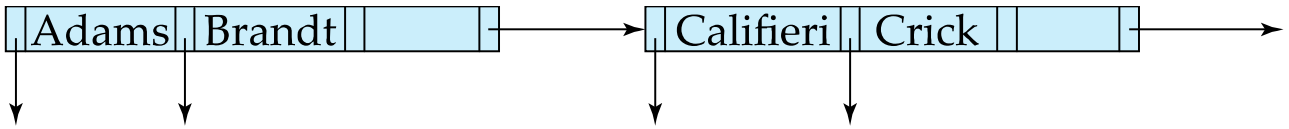
Assume record already added to the file. Let

- *pr* be pointer to the record, and let
- v be the search key value of the record

1. Find the leaf node in which the search-key value would appear
   1. If there is room in the leaf node, insert (v, *pr*) pair in the leaf node
   2. Otherwise, split the node (along with the new (*v*, *pr*) entry) as discussed in the next slide, and propagate updates to parent nodes.

Updates on B+-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the n (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first $\lceil n/2 \rceil$ in the original node, and the rest in a new node.
  - let the new node be p, and let k be the least key value in p. Insert (k,p) in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

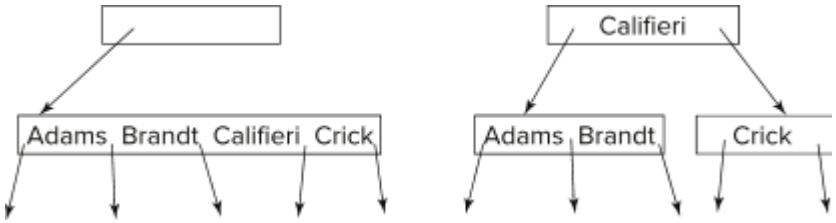


Result of splitting node containing Brandt, Califieri and Crick on inserting Adams

Next step: insert entry with (Califieri, pointer-to-new-node) into parent

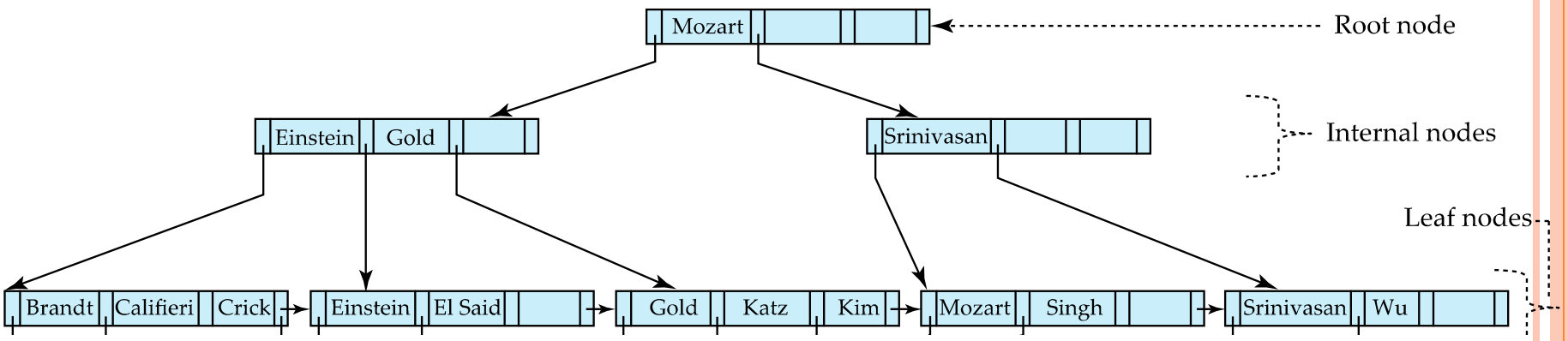
Insertion in B+-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ from M back into node N
  - Copy $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ from M into newly allocated node N'
  - Insert $(K_{\lceil n/2 \rceil}, N')$ into parent N
- Example

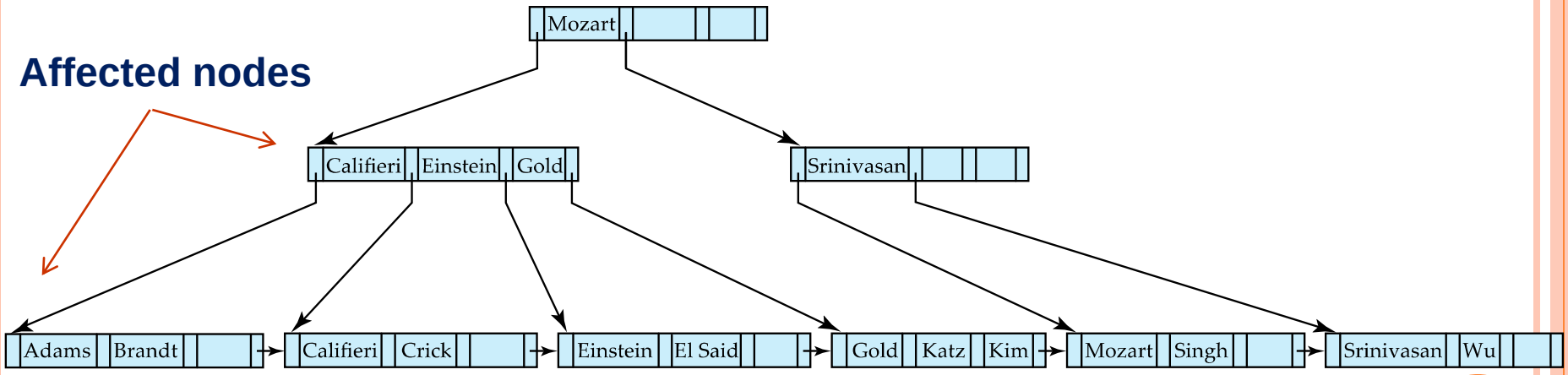


- **Read pseudocode in book!**

B+-Tree Insertion

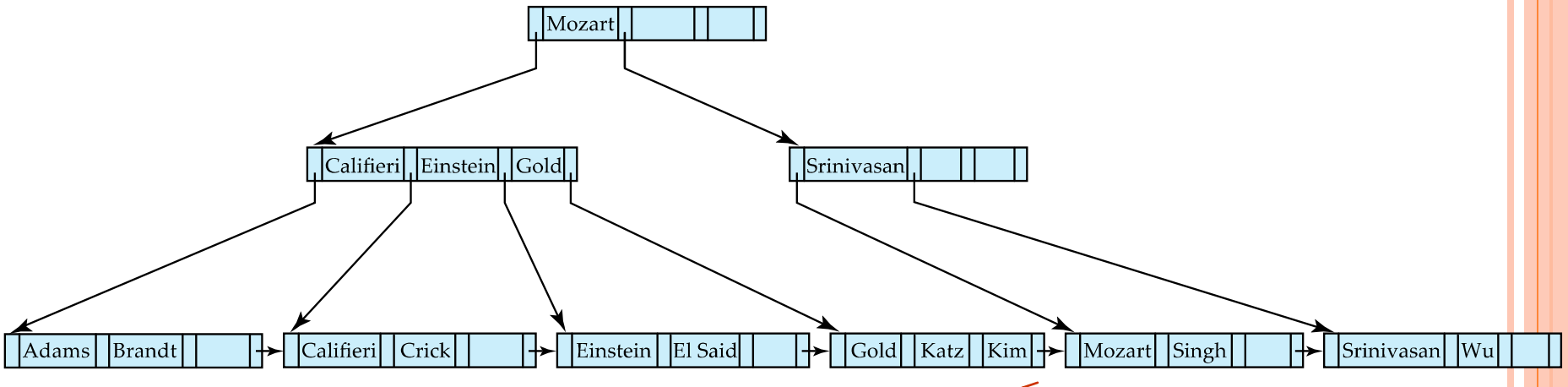


B+-Tree before insertion of "Adams"



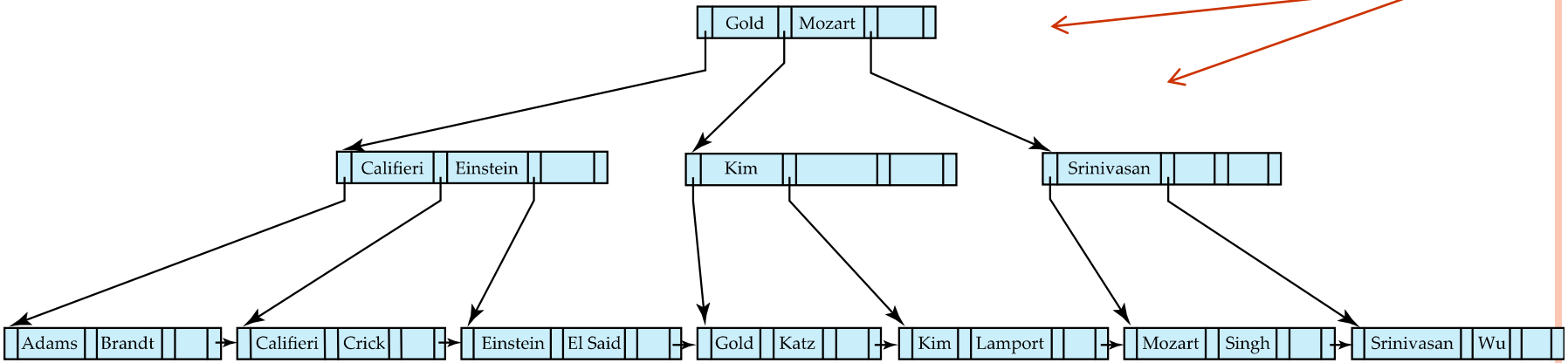
B+-Tree after insertion of "Adams"

B+-Tree Insertion



B+-Tree before insertion of "Lamport"

Affected nodes



Affected nodes

B+-Tree after insertion of "Lamport"

Updates on B+-Trees: Deletion

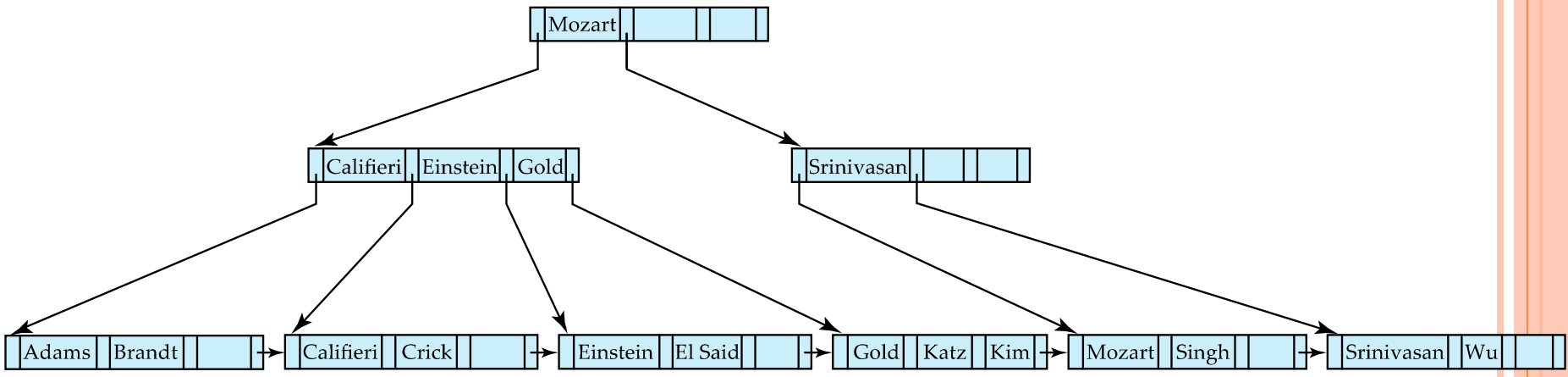
Assume record already deleted from file. Let *V* be the search key value of the record, and *Pr* be the pointer to the record.

- Remove (*Pr*, *V*) from the leaf node
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings***:
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair (K_{i-1}, P_i), where P_i is the pointer to the deleted node, from its parent, recursively using the above procedure.

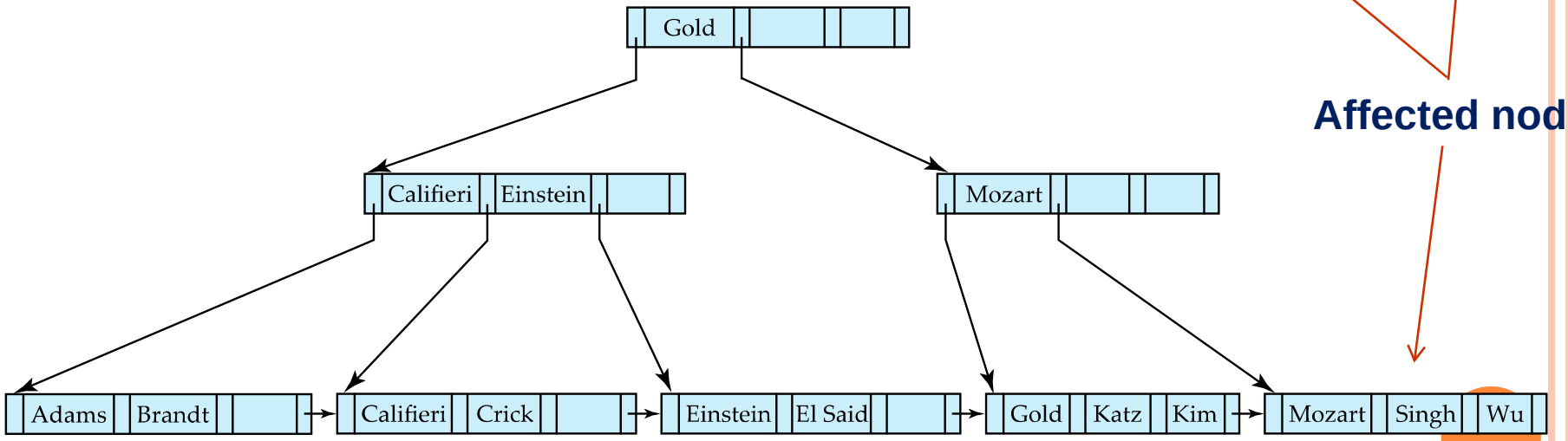
Updates on B+-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has $\lceil n/2 \rceil$ or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

Examples of B+-Tree Deletion



Before deleting "Srinivasan"

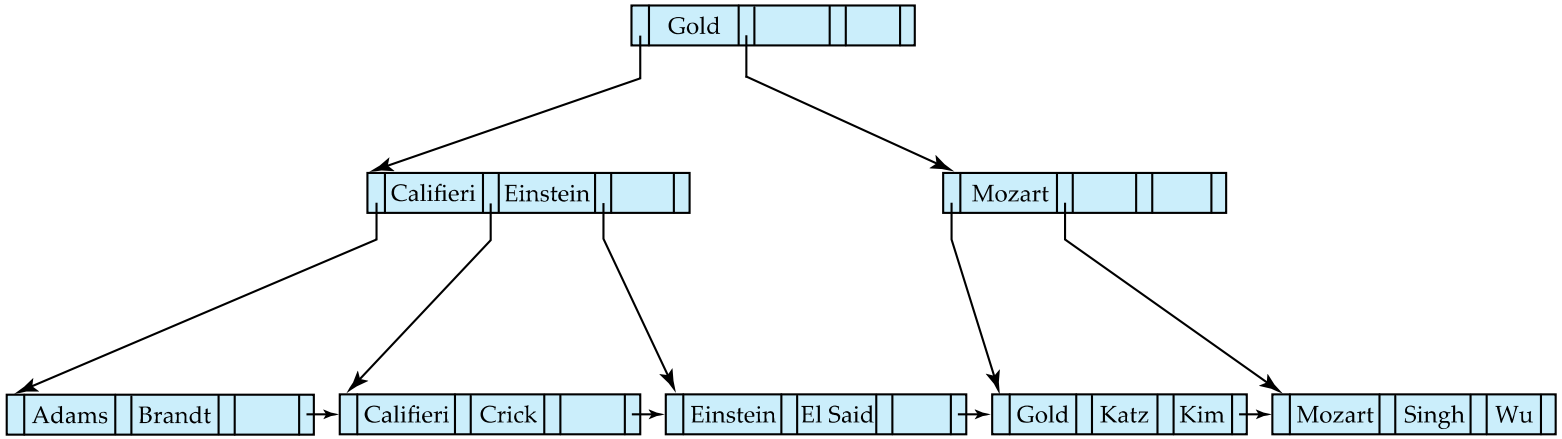


Affected nodes

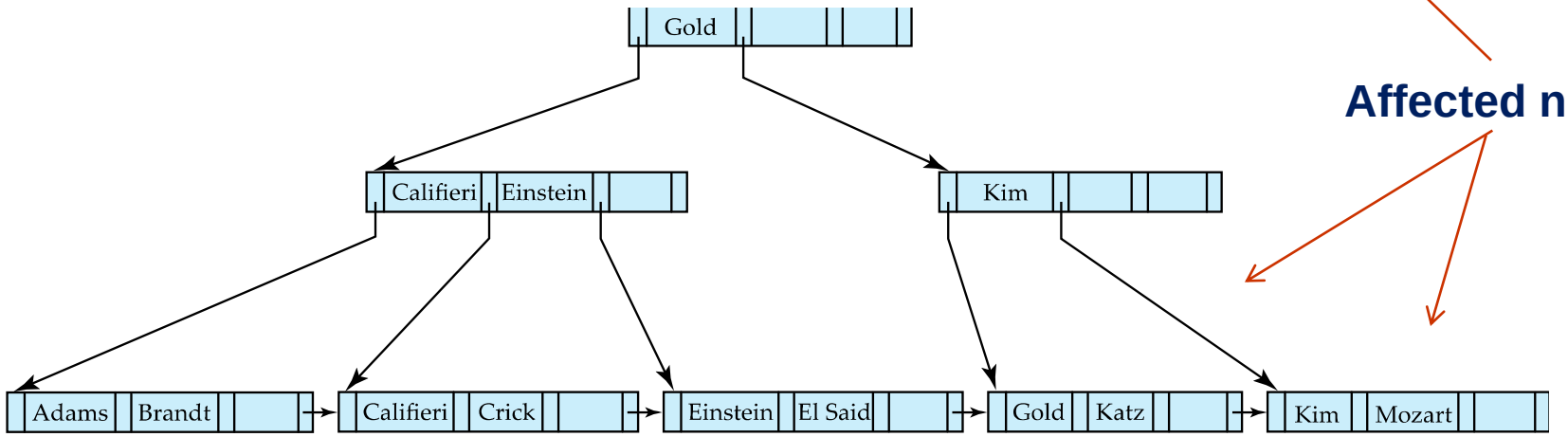
- Deleting "Srinivasan" causes **merging** of under-full leaves

After deleting "Srinivasan"

Examples of B+-Tree Deletion (Cont.)



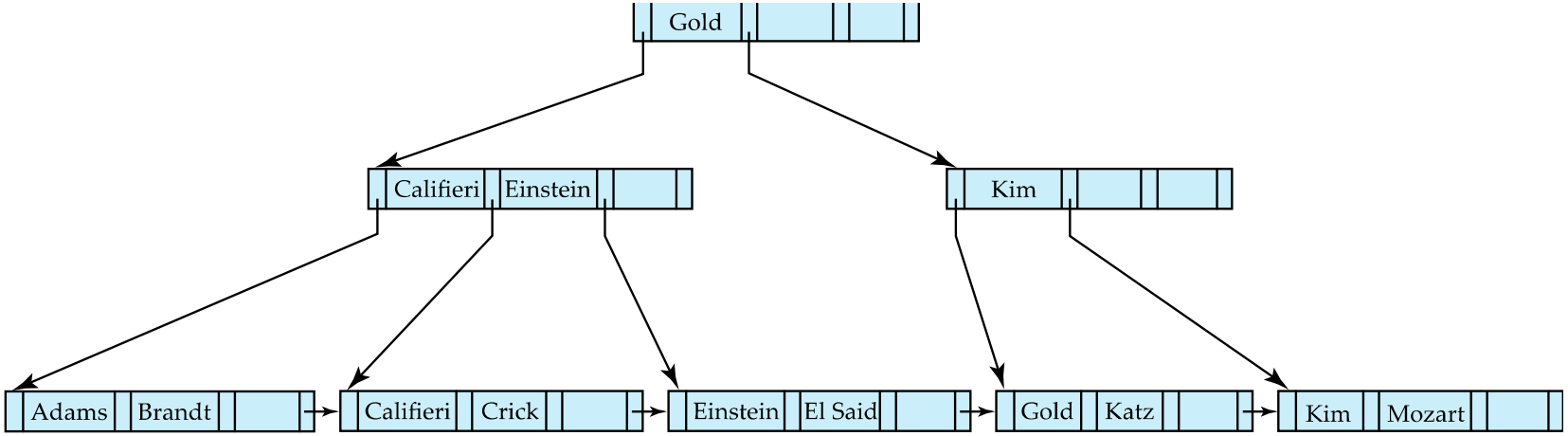
Before deleting "Singh" and "Wu"



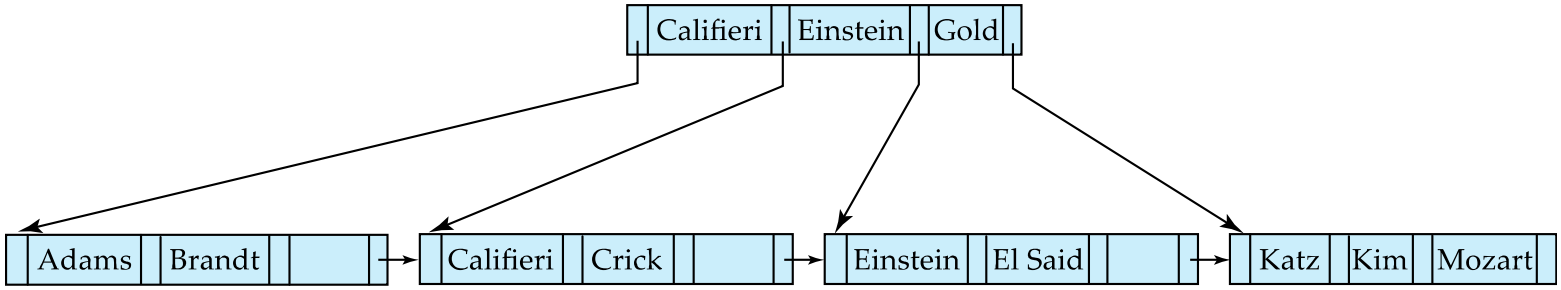
- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

After deleting "Singh" and "Wu"

Example of B+-tree Deletion (Cont.)



Before deletion of "Gold"



- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
  - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

After deletion of "Gold"

Complexity of Updates

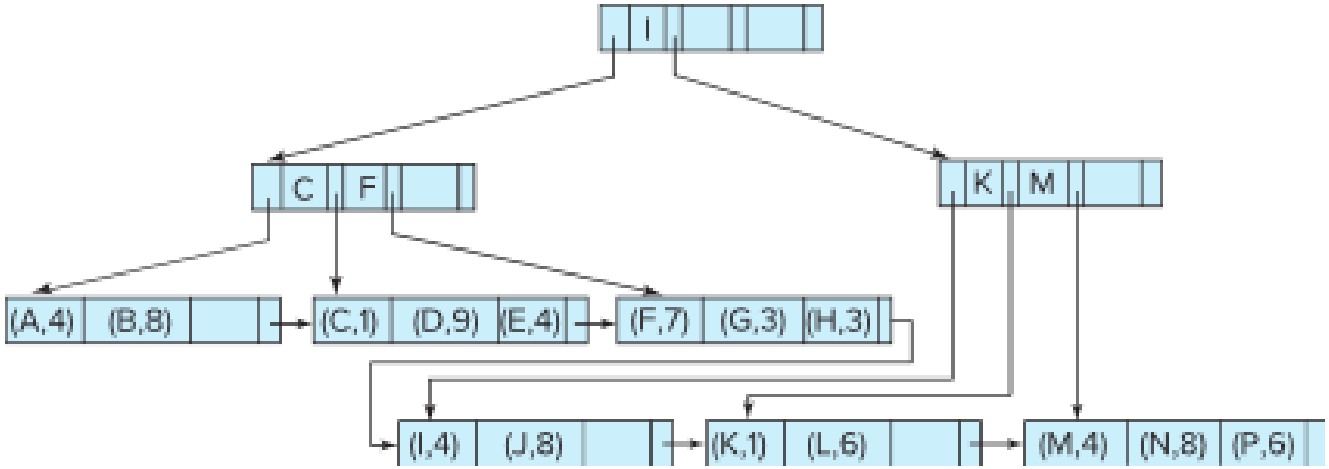
- Cost (in terms of number of I/O operations) of insertion and deletion of a single entry proportional to height of the tree
  - With K entries and maximum fanout of n, worst case complexity of insert/delete of an entry is $O(\log_{\lceil n/2 \rceil}(K))$
- In practice, number of I/O operations is less:
  - Internal nodes tend to be in buffer
  - Splits/merges are rare, most insert/delete operations only affect a leaf node
- Average node occupancy depends on insertion order
  - 2/3rds with random, ½ with insertion in sorted order

B^+-Tree File Organization

- B^+-Tree File Organization:
  - Leaf nodes in a B^+-tree file organization store records, instead of pointers
  - Helps keep data records clustered even when there are insertions/deletions/updates
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B^+-tree index.

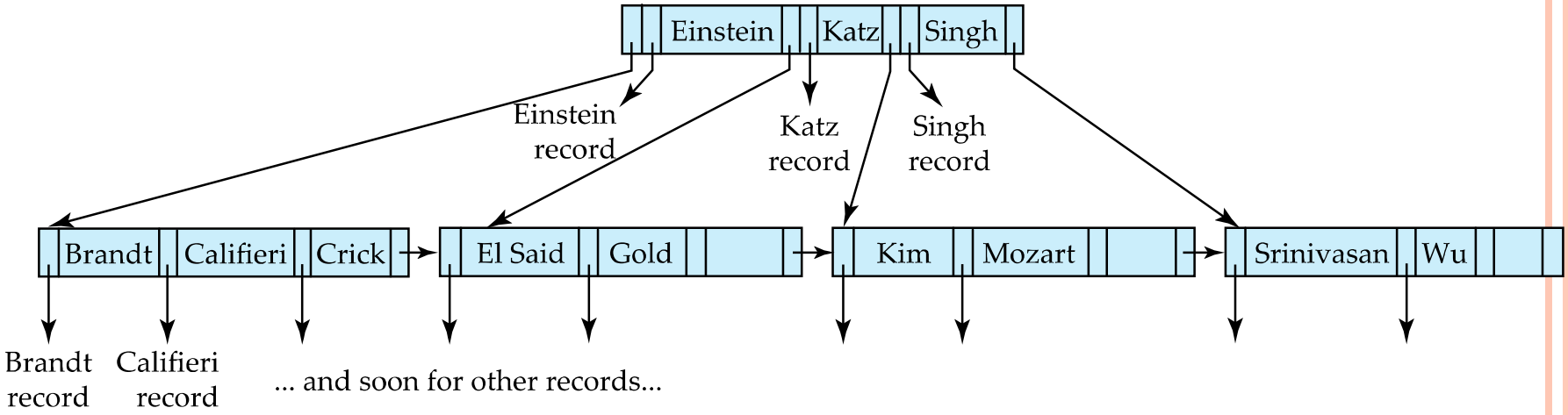
B+-Tree File Organization (Cont.)

- Example of B+-tree File Organization

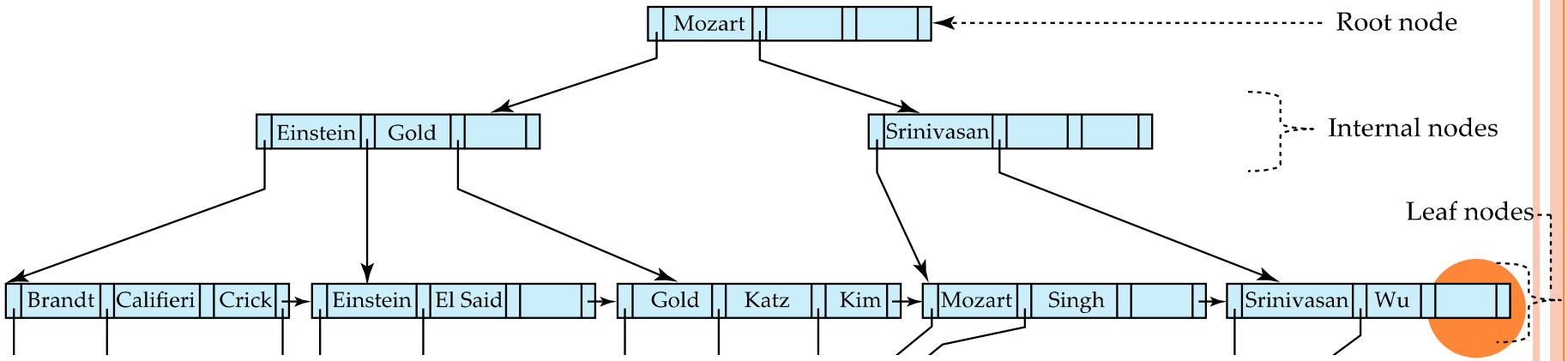


- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least $\lfloor 2n/3 \rfloor$ entries

B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



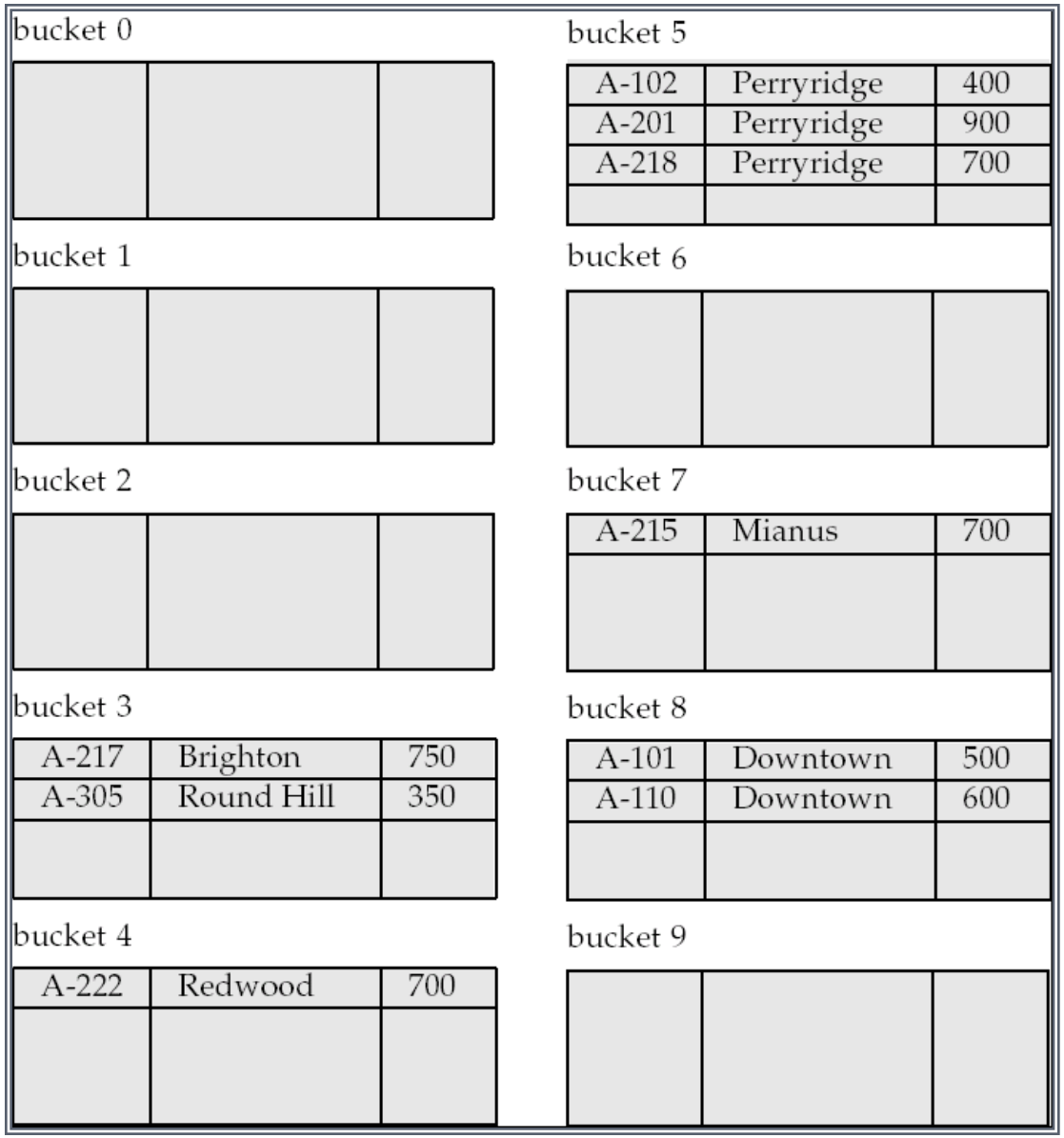
Static Hashing

- 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

- Let's consider the hash file organization of *account* file, using *branch_name* as search key
- 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(Perryridge) = 5 h(Round Hill) = 3
    h(Brighton) = 3

Example of Hash File Organization



Hash file organization of *account* file, using *branch_name* as key

Hash Functions

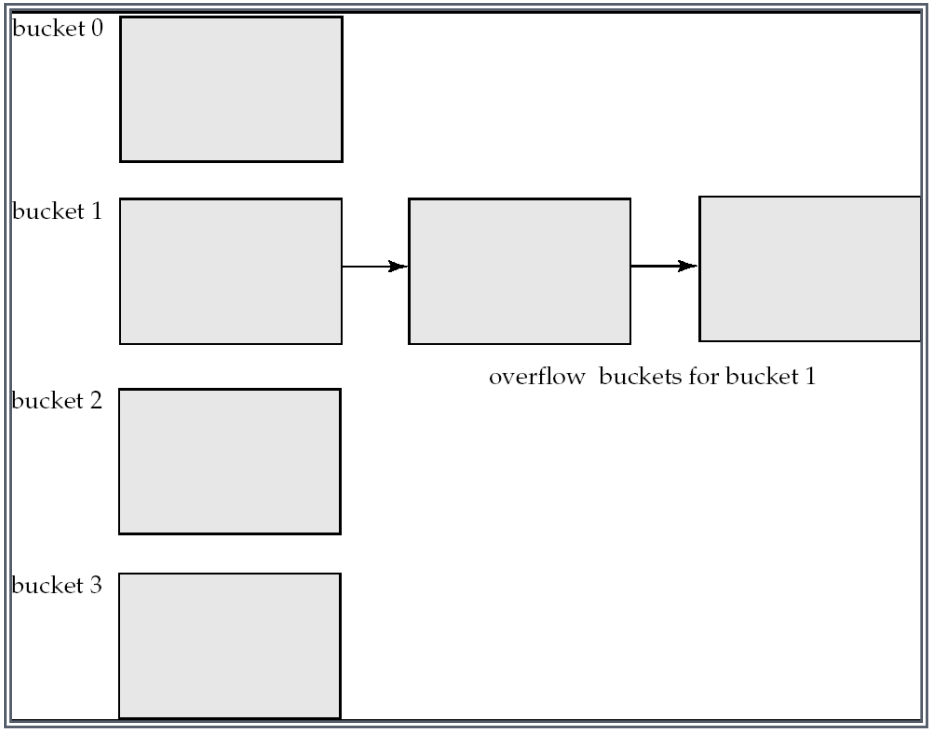
- Worst hash function maps all search-key values to the same bucket;
  - this makes access time proportional to the number of search-key values in the file.
- Hash function should be-
  - Uniform:
    - the hash function should distribute the search keys uniformly over all the available buckets
  - Random:
    - Typical hash value should not be correlated to any externally visible ordering on the search key value
    - Like alphabet ordering, length of the search keys, etc.

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.)

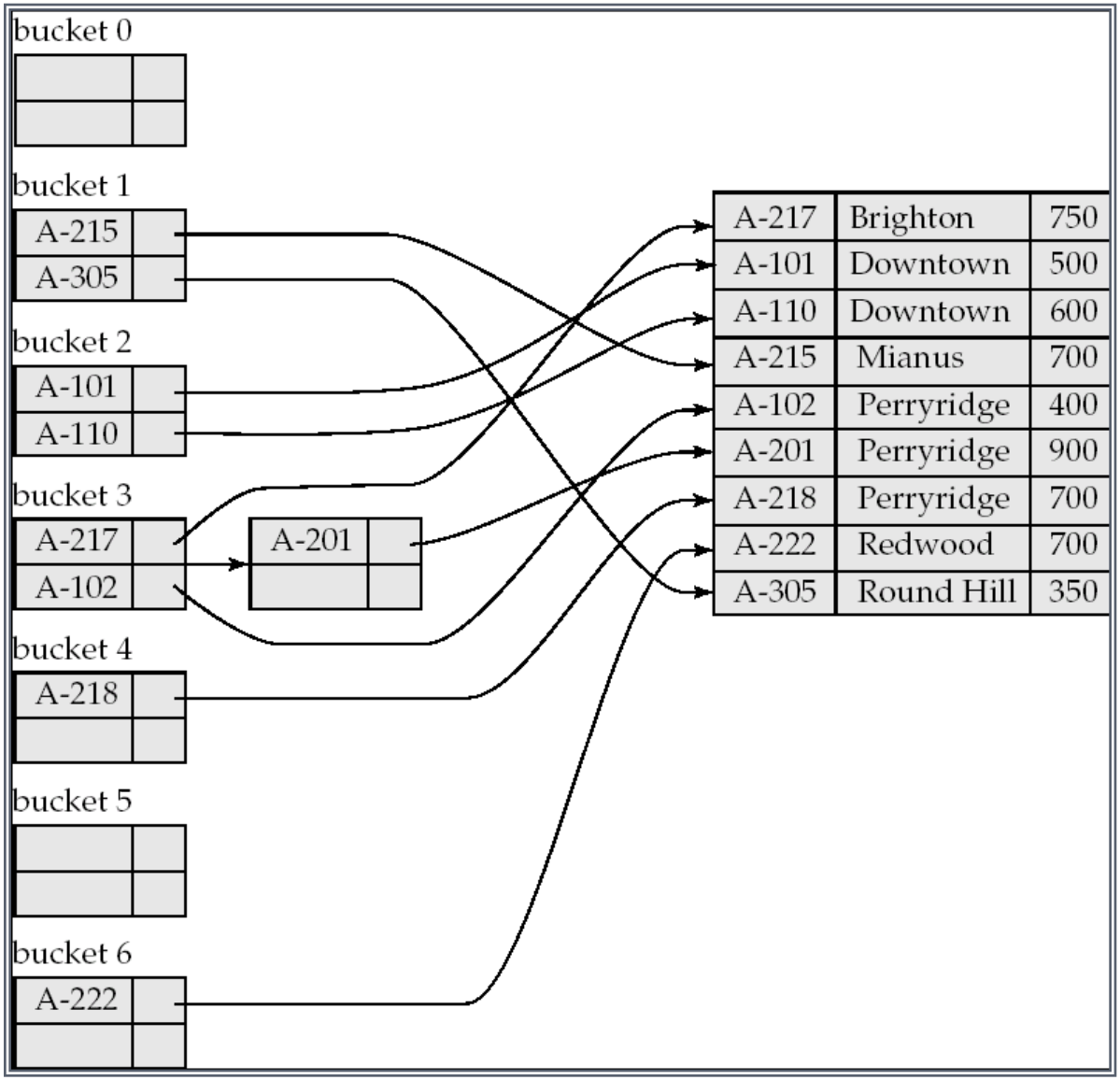
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.



Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



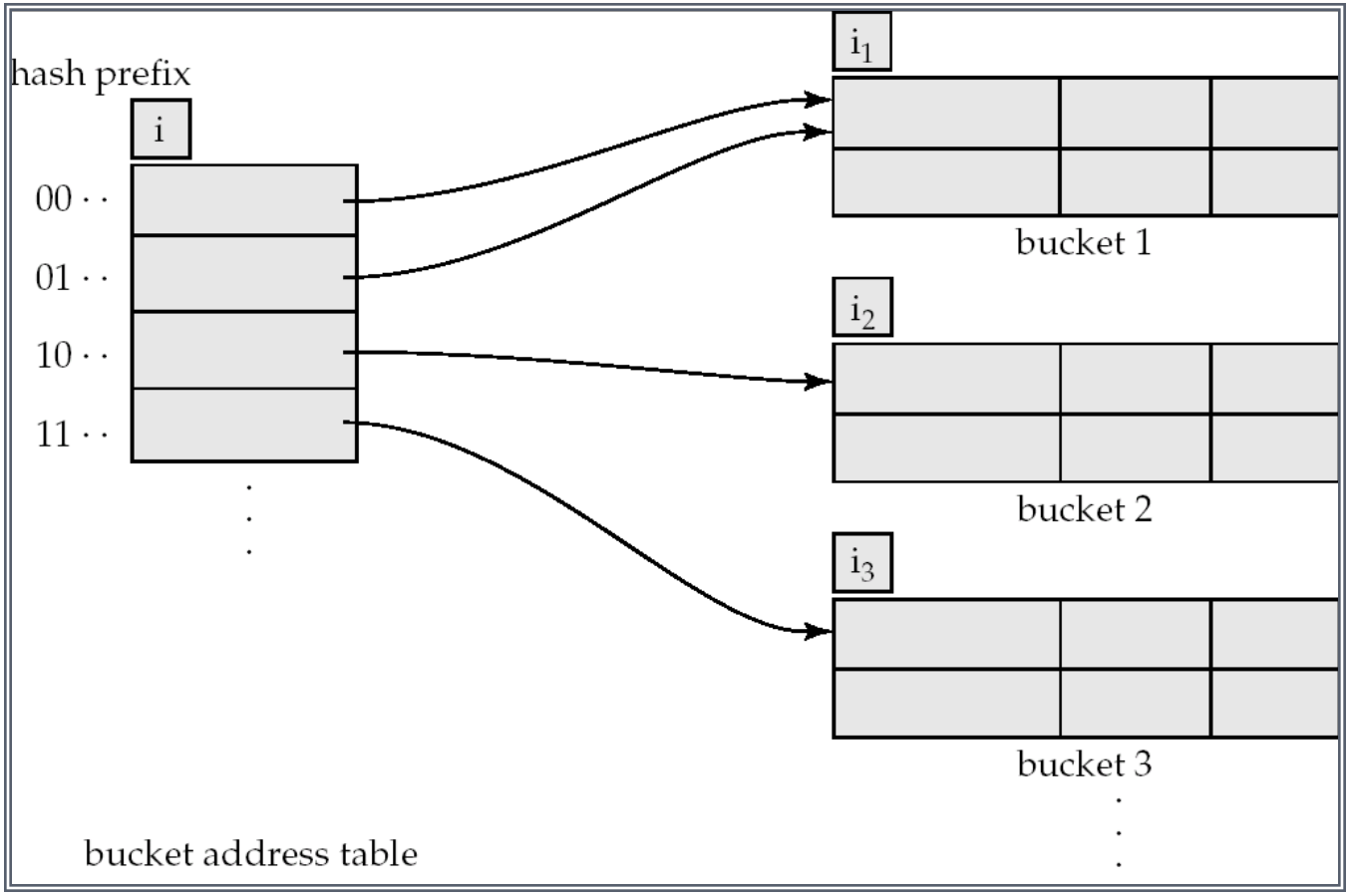
Deficiencies of Static Hashing

- In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- 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, $0 \le i \le 32$.
    - Bucket address table size = 2^i. Initially $i = 0$
    - Value of i grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket
  - Thus, actual number of buckets is typically $< 2^i$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$

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 a displacement into 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
    - Overflow buckets used instead in some cases

Insertion in Extendable Hash Structure (Cont)

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

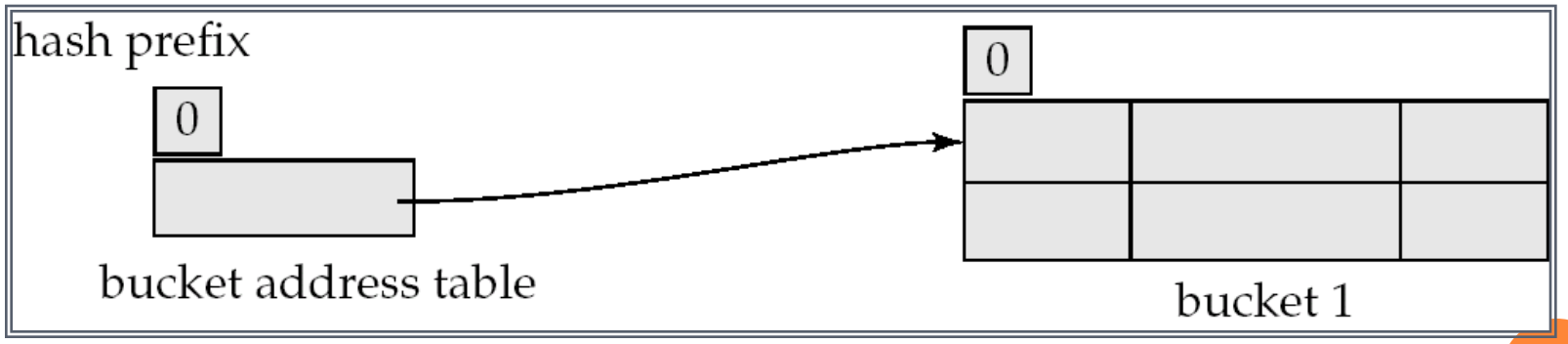
- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - recompute new bucket for 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.
    - recompute 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).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

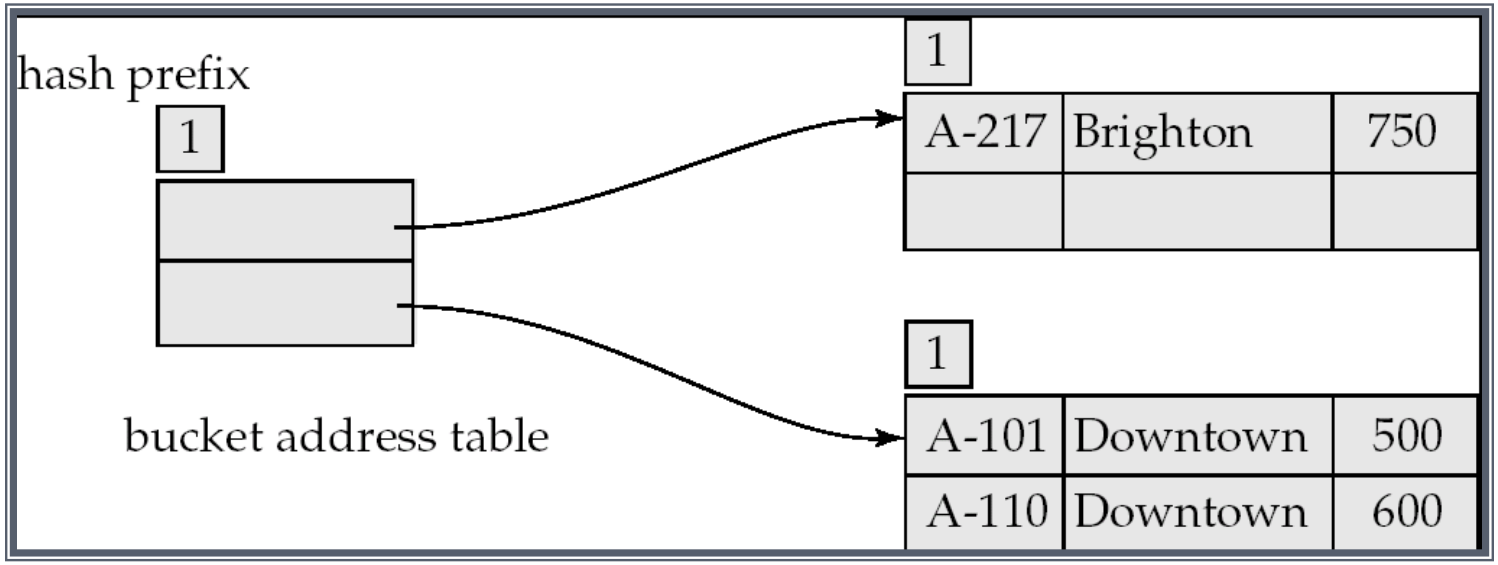
branch_name	h(*branch_name*)
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

Example (Cont.)

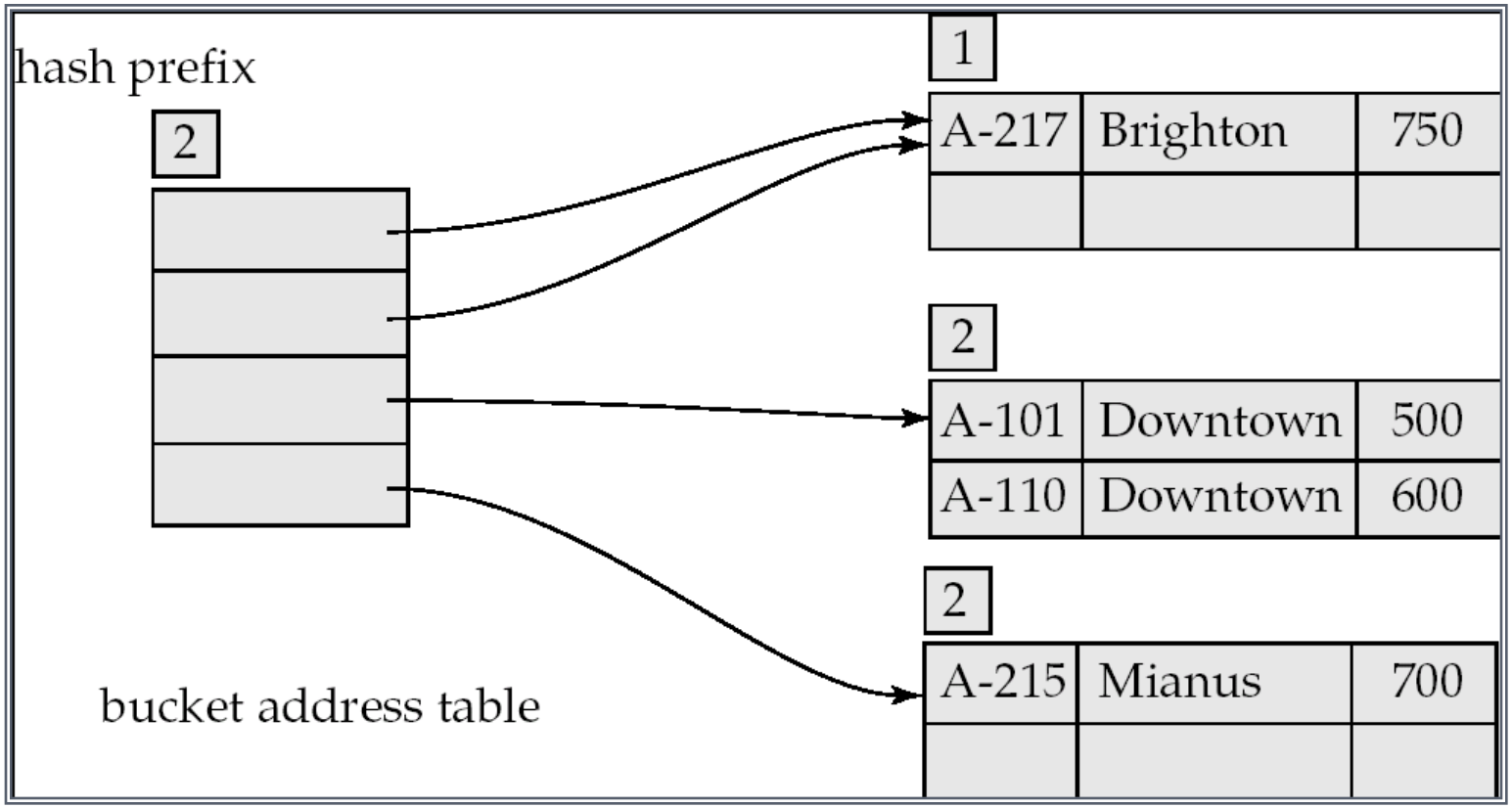
- Hash structure after insertion of one Brighton and two Downtown records



Now Insert Mianus record

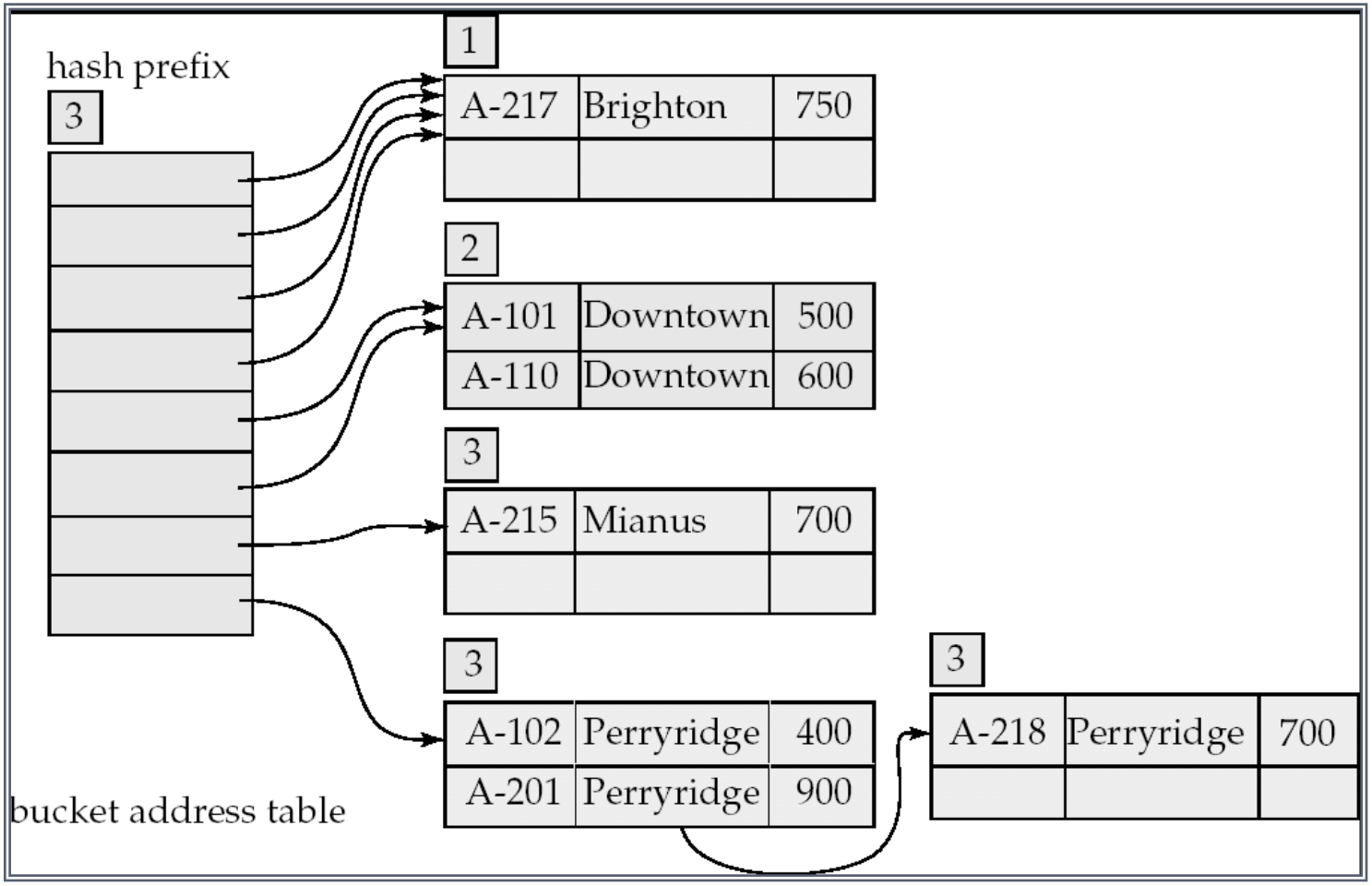
Example (Cont.)

Hash structure after insertion of Mianus record



Now insert Perryridge record three times

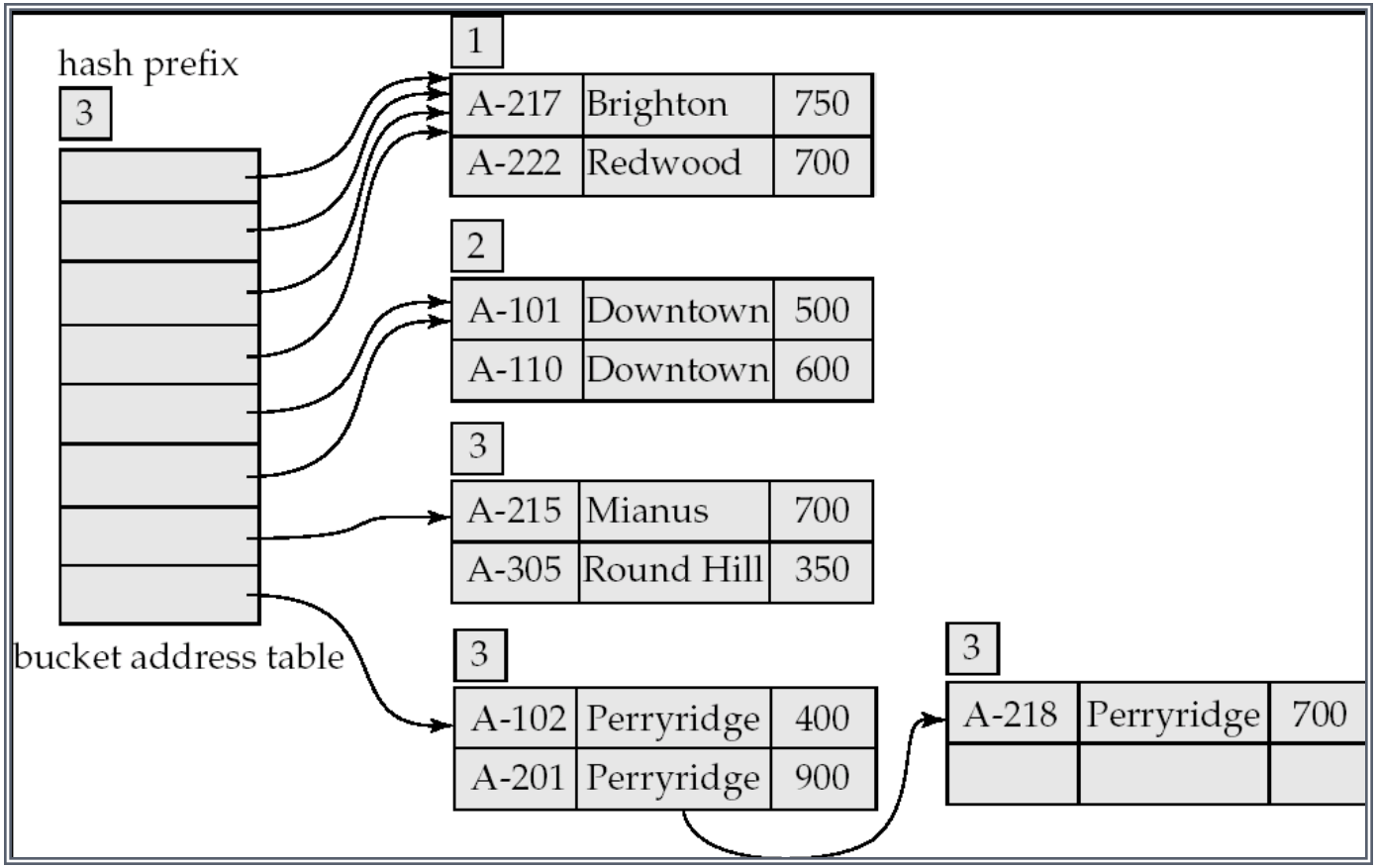
EXAMPLE (CONT.)



Hash structure after insertion of three Perryridge records

Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B^+-tree file organization to store bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

Linear Hashing

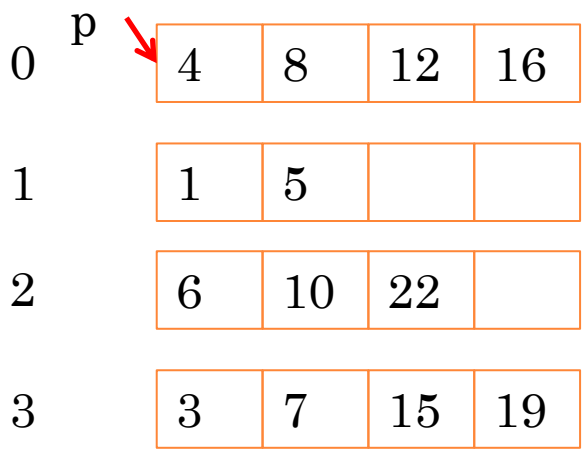
- Another dynamic hashing technique
- It grows and shrinks one bucket at a time
- Unlike extendable hashing, it does not use a bucket address table
- When an overflow occurs it does not always split the overflow bucket
- The no. of buckets grows and shrinks in a linear fashion
- Overflow are handled by creating a chain of buckets
- Hashing function changes dynamically
- Atmost two hashing functions can be used at any given instant

Linear Hashing: Initial Layout

- The linear hashing has m initial buckets labeled 0 through m-1
- An initial hashing function $h_0(k) = f(k) \% m$
- For simplicity, we assume that $h_0(k) = k \% m$
- A pointer p which points to the bucket to be split next
- The bucket split occurs whenever an overflow bucket is generated

Bucket#

Overflow buckets

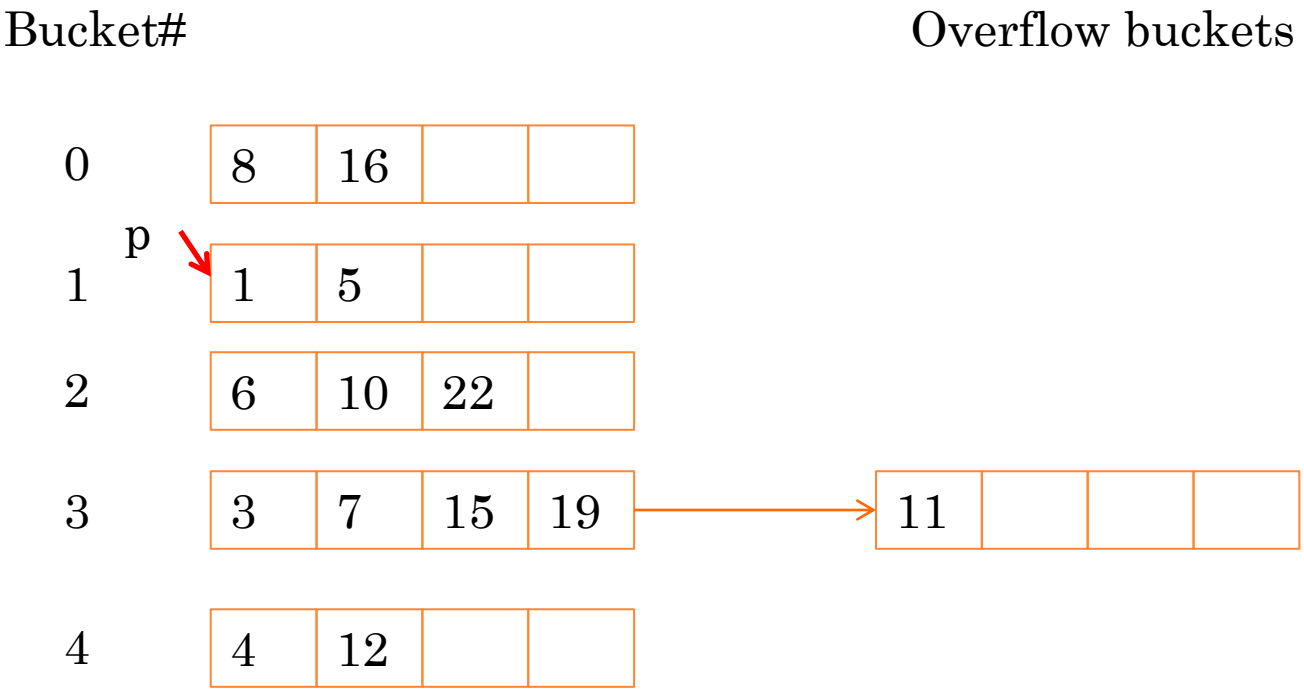


Here m=4, p=0, $h_0(k)=k \%4$

Linear Hashing: Bucket Split

- When the first overflow occurs (it may occur in any bucket)
  - Bucket 0 which is pointed by p is split into two buckets (original bucket# 0 and a new bucket bucket# m)
  - A new empty bucket is also added in the overflown bucket to accommodate the overflow
  - The search values originally mapped into bucket 0 (using function h_0) are now distributed between buckets 0 and m using a new hashing function h_1

Now let's try to insert a new record with key 11



Here p=1, $h_0(k)=k \%4$, $h_1(k)=k \%8$

- In case of insertion and overflow condition,
  - If ($h_0(k)<p$) then use $h_1(k)$
  - a new split that will attach a new bucket *m+1* and the contents of bucket *1* will be distributed using h_1 between buckets *1* and *m+1*
- For every split, *p* is increased by *1*
- The necessary property for linear hashing to work-
  - The search values that were originally mapped by h_0 to some bucket *j* must be remapped using h_1 to bucket *j* or *j+m*
  - An example of such hashing function would be $h_1(k) = k \% 2m$

Linear Hashing: Round and Hash Function advancement

- After enough overflows, all original *m* buckets will be split
- This marks the end of splitting round 0
- During round 0, p went subsequently from 0 to m-1
- At the end of round 0, the linear hashing scheme has a total of 2m buckets
- Hashing function h_0 is no longer needed as all 2m buckets can be addressed by hashing function h_1
- Variable p is reset to 0 and a new round (namely splitting round 1) starts
- A new hash function h_2 will start to be used

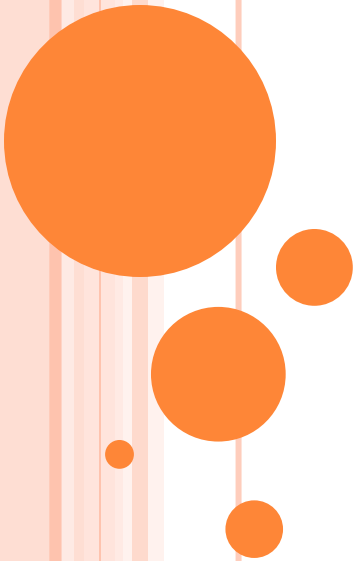
- In general, linear hashing involves a family of hash functions h_0, h_1, h_2, and so on
- Let the initial hash function is $h_0(k) = f(k) \% m$
- Then any later hash function can be defined as $h_i(k) = f(k) \% 2^i m$
- This will guarantee that if h_i hashes a key to bucket j
  - then h_{i+1} will hash the same key to either j or bucket $j+2^i m$

Linear Hashing: Searching

- A search scheme is needed to map a key k to a bucket
- It works as follows-
  - If $h_i(k) >= p$ choose bucket $h_i(k)$ since the bucket has not been split yet
  - If $h_i(k) < p$ choose bucket $h_{i+1}(k)$ which can either be $h_i(k)$ or its split image $h_i(k) + 2^i m$

Comparison of Ordered Indexing and Hashing

- The following issues must be considered
  - Cost of periodic re-organization
  - Relative frequency of insertions and deletions
  - Is it desirable to optimize average access time at the expense of worst-case access time?
  - Expected type of queries:
    - Hashing is generally better at retrieving records having a specified value of the key.
    - If range queries are common, ordered indices are to be preferred

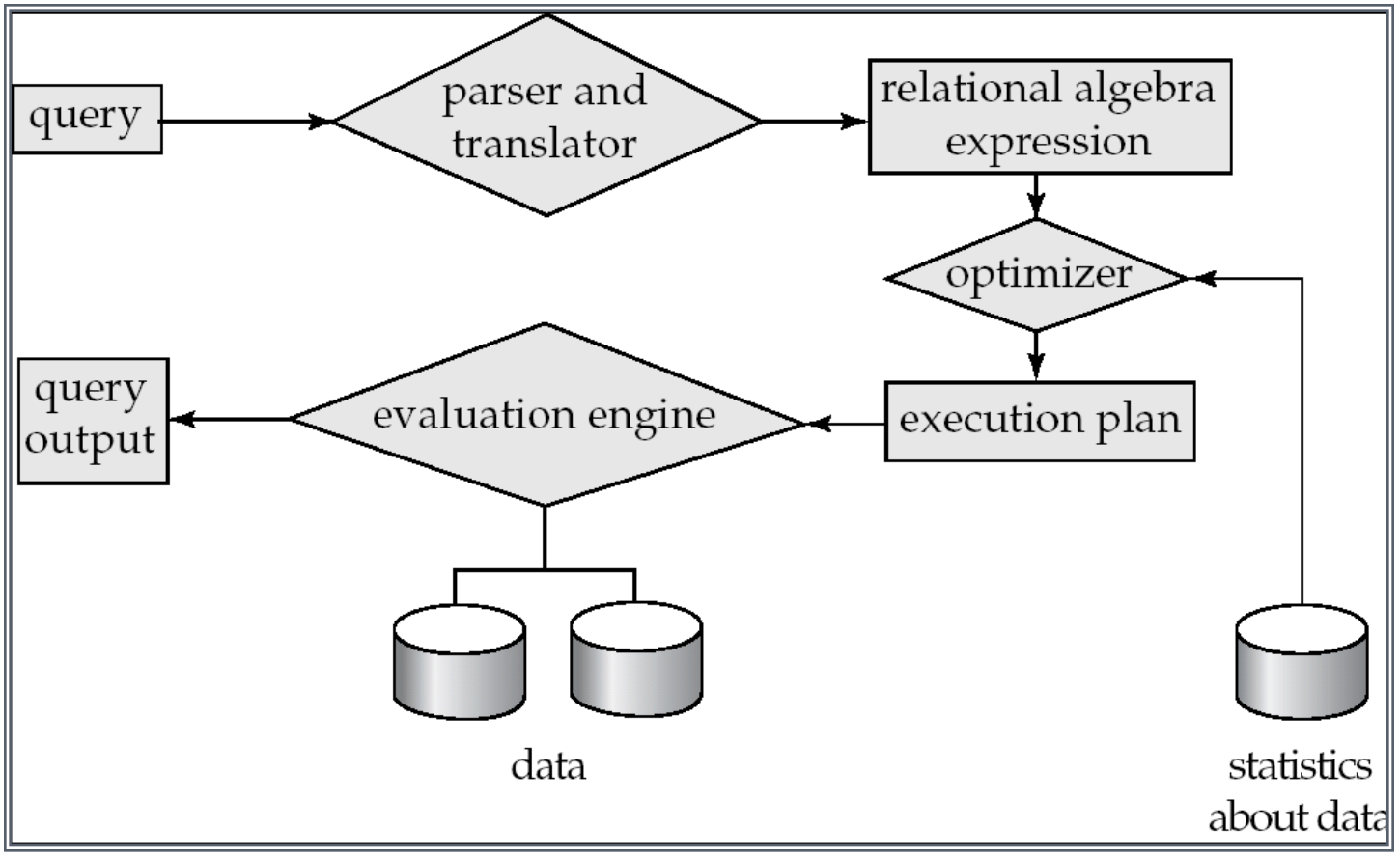


CS354: Query Processing & Optimization

Database

Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- **Parsing and translation**
  - Parser checks syntax, verifies relations
  - This is then translated into parse tree representation and then into relational algebra expression
- **Optimization**
  - A query can be evaluated in several ways
  - Even relational algebra expression specifies partially how to evaluate a query
  - A sequence of primitive operations that can be used to evaluate a query is known as *query evaluation plan*
- **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

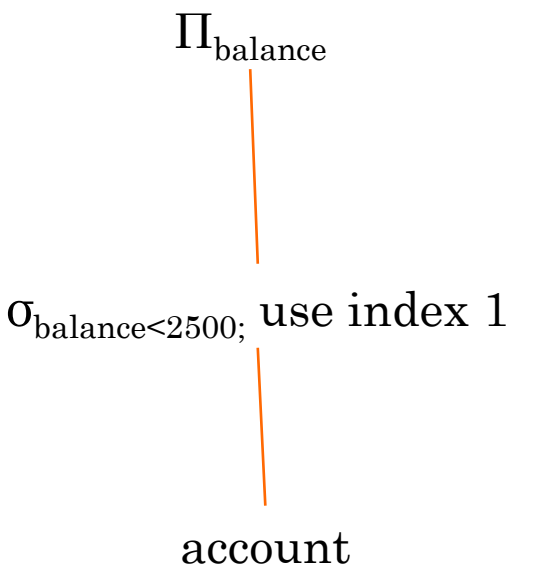
- Consider the following query:

```
SELECT balance
FROM account
WHERE balance < 2500
```

- The query can be expressed in relational algebra expressions:
  a. $\sigma_{balance<2500}(\Pi_{balance}(account))$
  b. $\Pi_{balance}(\sigma_{balance<2500}(account))$

Basic Steps in Query Processing : Optimization

- Each relational algebra operation can be evaluated using one of several different algorithms
- 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 ≥ 2500



Query-evaluation plan

A relational algebra operation annotated with instructions on how to evaluate it is called an evaluation primitive

A sequence of primitive operations that can be used to evaluate a query is a query execution plan or query evaluation plan

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.
- Next we will see
  - How to measure query costs
  - Algorithms for evaluating relational algebra operations
  - How to combine algorithms for individual operations in order to evaluate a complete expression

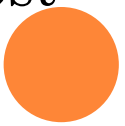
Measures of Query Cost

- 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
  - Assumption: single disk
    - Can modify formulae for multiple disks/RAID arrays
    - Or just use single-disk formulae, but interpret them as measuring **resource consumption** instead of time

Measures of Query Cost (Cont.)

- For simplicity we just use the *number of block transfers from disk and the number of seeks* as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for ***b*** block transfers plus ***S*** seeks

    $$b * t_T + S * t_S$$
- We ignore CPU costs for simplicity
  - Real systems do take CPU cost into account
- We do not include cost to writing output to disk in our cost formulae



Measures of Query Cost (Cont.)

- Several algorithms can reduce disk I/O by using extra buffer space
  - Amount of real memory available to buffer depends on other concurrent queries and OS processes, known only during execution
    - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available

Selection Operation

- **File scan** – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm **A1** (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - **Cost estimate = b_r block transfers + 1 seek**
    - b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - **Cost for avg. cases = (b_r/2) block transfers + 1 seek**
  - Linear search can be applied regardless of
    - selection condition or
    - ordering of records in the file, or
    - availability of indices

Selection Operation (Cont.)

- **A2** *(binary search).* Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - cost of locating the first tuple by a binary search on the blocks = $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
    - If there are multiple records satisfying selection
      - *Add transfer cost of the* number of blocks containing records that satisfy selection condition

Selections Using Indices

- **Index scan** – search algorithms that use an index
  - selection condition must be on search-key of index.
- **A3** (*primary B+ tree index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - ***Cost* = $(h_i + 1) * (t_T + t_S)$**
- **A4** (*primary B+ tree index on nonkey, equality)* Retrieve multiple records.
  - Records will be on consecutive blocks
    - Let b = number of blocks containing matching records
  - ***Cost* = $h_i * (t_T + t_S) + t_S + t_T * b$**
- **A5** (*equality on search-key of secondary index).*
  - Retrieve a single record if the search-key is a candidate key
    - ***Cost* = $(h_i + 1) * (t_T + t_S)$**
  - Retrieve multiple records if search-key is not a candidate key
    - each of *n* matching records may be on a different block
    - **Cost = $(h_i + n) * (t_T + t_S)$**
      - Can be very expensive!

Selections Involving Comparisons

- Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- **A6** (*primary index, comparison).* (Relation is sorted on A)
  - For $\sigma_{A \ge V}(r)$ use index to find first tuple $\ge v$ and scan relation sequentially from there
  - For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple $> v$; do not use index
- **A7** (*secondary index, comparison*).
  - For $\sigma_{A \ge V}(r)$ use index to find first index entry $\ge v$ and scan index sequentially from there, to find pointers to records.
  - For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry $> v$
  - In either case, retrieve records that are pointed to
    - may require an I/O for each record
    - Linear file scan may be cheaper

Implementation of Complex Selections

- **Conjunction:** $\sigma_{\theta 1 \wedge \theta 2 \wedge \ldots \theta n}(r)$
- **A8** (*conjunctive selection using one index).*
  - Select a combination of θ_i and algorithms A1 through A7 that results in the least cost for $\sigma_{\theta i}$ (*r).*
  - Test other conditions on tuple after fetching it into memory buffer.
- **A9** (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- **A10** (*conjunctive selection by intersection of identifiers).*
  - Requires indices with record pointers.
  - Use corresponding index for each condition, and take intersection of all the obtained sets of record pointers.
  - Then fetch records from file
  - If some conditions do not have appropriate indices, apply test in memory.

Algorithms for Complex Selections

- **Disjunction:** $\sigma_{\theta 1 \vee \theta 2 \vee \ldots \theta n}(r)$.
- **A11** (*disjunctive selection by union of identifiers).*
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- **Negation:** $\sigma_{\neg\theta}(r)$
  - Use linear scan on file
  - If very few records satisfy ¬θ, and an index is applicable to θ
    - Find satisfying records using index and fetch from file

Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Example: use the following information
  - Number of records-
    - *customer*: 10,000 and *depositor*: 5000
  - Number of blocks-
    - *customer*: 400 and *depositor*: 100

Nested-Loop Join

- To compute the theta join $r \bowtie_\theta s$

  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 they satisfy the join condition θ
      if they do, add $t_r \bullet t_s$ to the result.
    end
  end

- r is called the **outer relation** and s the **inner relation** of the join.
- Requires no indices and can be used with any kind of join condition.
- Expensive since it examines every pair of tuples in the two relations.

Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  - Block transfers $n_r * b_s + b_r$ and
  - Seeks $n_r + b_r$
- Example: Assuming worst case memory availability
- Cost estimate is
  - with *depositor* as outer relation:
    - **5000 * 400 + 100 = 2,000,100** block transfers,
  - with *customer* as the outer relation
    - **10000 * 100 + 400 = 1,000,400** block transfers

Nested-Loop Join (Cont.)

- If the smaller relation fits entirely in memory, use that as the inner relation.
- What would be the cost?
- The cost becomes
  - block transfers $b_r + b_s$ and
  - seeks **2**
- Example: If smaller relation (*depositor)* fits entirely in memory, the cost estimate will be **(100+400)=500** block transfers.

Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of 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
                Check if (t_r,t_s) satisfy the join condition
                    if they do, add t_r • t_s to the result.
            end
        end
    end
end
```

Block Nested-Loop Join (Cont.)

- Each block in the inner relation *s* is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation
- Worst case estimate:
  - block transfers $b_r * b_s + b_r$ and
  - Seeks $2b_r$
- Clearly it is efficient to use the smaller relation as the outer relation
- Best case: $b_r + b_s$ block transfers and **2** seeks
- If we use depositor as the outer relation
  - The worst case: **100*400+100=40,100** block accesses required
  - The best case: **100+400=500** remains same

Performance Improvement Strategies of Nested and Block Nested Loop Join

- If equi-join attribute forms a key on inner relation,
  - stop inner loop on first match
- Scan inner loop forward and backward alternately, to reduce the no. of disk accesses
- In block nested-loop, use M — 2 disk blocks as blocking unit for outer relations, where M = memory size in blocks; use remaining two blocks to buffer inner relation and output
  - The number of scans of inner relation reduces from b_r to $\lceil b_r / (M-2) \rceil$
  - Cost = $\lceil b_r / (M-2) \rceil * b_s + b_r$ block transfers and $2\lceil b_r / (M-2) \rceil$ seeks
- Use index on inner relation if available

Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple t_r in the outer relation r, use the index to look up tuples in s that satisfy the join condition with tuple t_r.
- Worst case: buffer has space for only one block of r, and one block of index
- For each tuple in r, we perform an index lookup on s.
- Cost of the join: $b_r\,(t_T + t_S) + n_r * c$
  - Where c is the cost of traversing index and fetching all matching s tuples for one tuple or r
  - c can be estimated as cost of a single selection on s using the join condition.
- If indices are available on join attributes of both r and s, use the relation with fewer tuples as the outer relation.

Example of Nested-Loop Join Costs

- Compute *depositor* ⋈ *customer*, with *depositor* as the outer relation.
- Let *customer* relation has a primary B^+-tree index on the join attribute *customer-name*, which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- *depositor* has 5000 tuples
- Cost of block nested loops join
  - **400*100 + 100 = 40,100** block transfers
    - assuming worst case memory
    - may be significantly less with more memory
- Cost of indexed nested loops join
  - **100 + 5000 * 5 = 25,100** block transfers
  - CPU cost likely to be less than that for block nested loops join

Query Optimization

- Process of selecting most efficient query evaluation plan
- Users may not write the query efficiently
- However, the system has to construct a query evaluation plan that minimizes the cost of query evaluation
- Different aspects of query optimizations
  - Equivalent expressions at the relational algebra level
  - Different algorithms for each operation

- 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**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - 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

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent**
  - if the two expressions generate the same set of tuples on every legal database instance
- An **equivalence rule** says that expressions of two forms are equivalent
  - can replace expression of first form by second, or vice versa

Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{L_1}(\Pi_{L_2}(\ldots(\Pi_{Ln}(E))\ldots)) = \Pi_{L_1}(E)$$

4. Selections can be combined with Cartesian products and theta joins.
   a. $\sigma_{\theta}(E_1 \text{ X } E_2) = E_1 \bowtie_{\theta} E_2$
   b. $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

Equivalence Rules (Cont.)

5. Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6. (a) Natural join operations are associative:

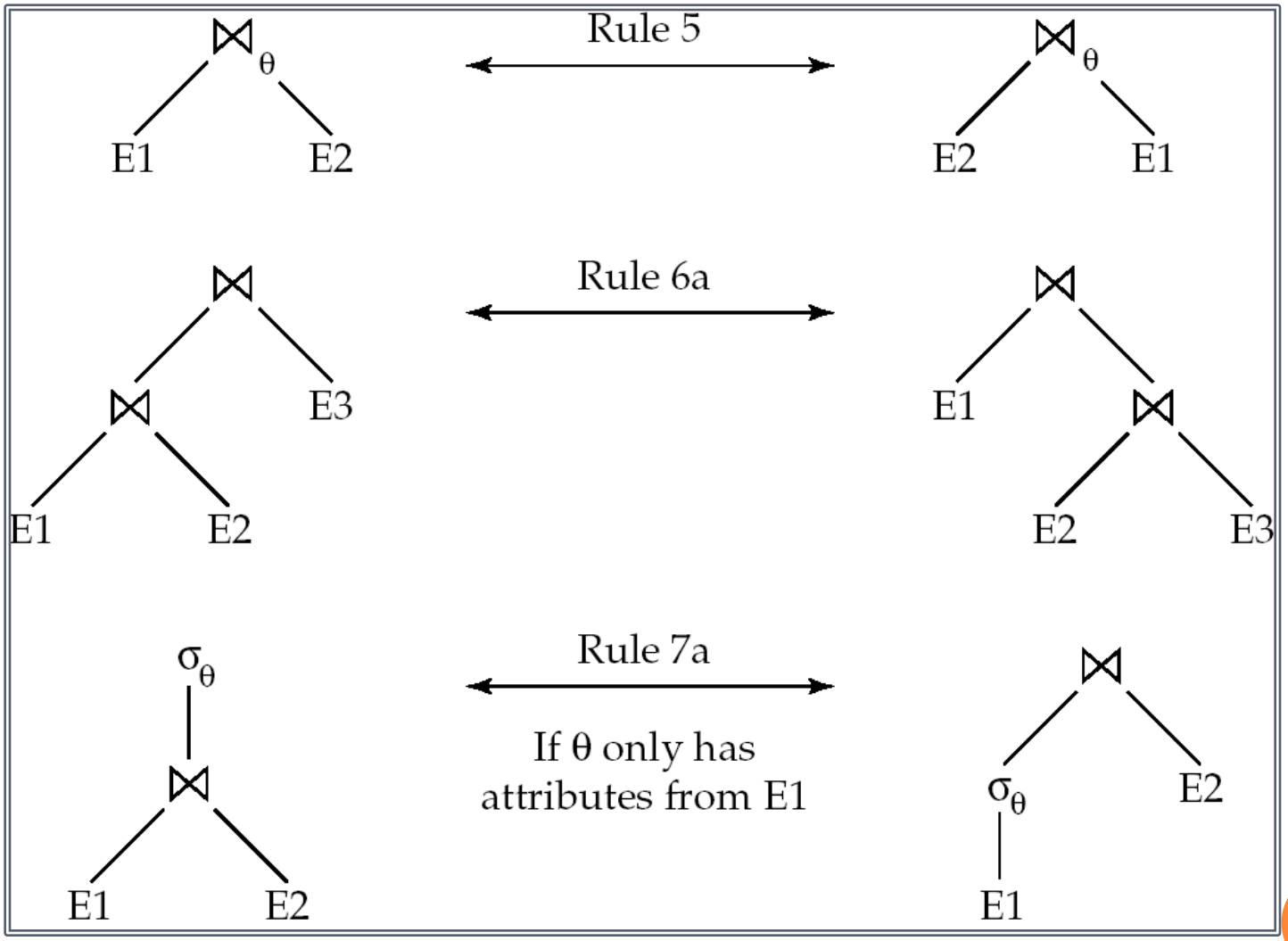
$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 1 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

where θ_2 involves attributes from only E_2 and E_3.

Pictorial Depiction of Equivalence Rules



Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
   (a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_{\theta} E_2$$

   (b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2.

$$\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_{\theta} (\sigma_{\theta 2}(E_2))$$

Equivalence Rules (Cont.)

8. The projection operation distributes over the theta join operation as follows:

   (a) if θ involves only attributes from $L_1 \cup L_2$; where L_1 and L_2 be attributes of E1 and E2 respectively

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = (\Pi_{L_1}(E_1)) \bowtie_\theta (\Pi_{L_2}(E_2))$$

   (b) Consider a join $E_1 \bowtie_\theta E_2$.

   - Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
   - Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
   - Let L_4 be attributes of E_2 that are involved in join condition θ, but are not in $L_1 \cup L_2$.

$$\Pi_{L_1 \cup L_2}(E_1 \bowtie_\theta E_2) = \Pi_{L_1 \cup L_2}((\Pi_{L_1 \cup L_3}(E_1)) \bowtie_\theta (\Pi_{L_2 \cup L_4}(E_2)))$$

Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$E_1 \cup E_2 = E_2 \cup E_1$$
$$E_1 \cap E_2 = E_2 \cap E_1$$

   - (set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$
$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Can we write $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$?

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Transformation Example: Pushing Selections

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

$$\Pi_{customer\_name}(\sigma_{branch\_city = \text{“Brooklyn”}}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using rule 7a.

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch)) \bowtie (account \bowtie depositor))$$

- Performing the selection as early as possible reduces the size of the relation to be joined.

Example with Multiple Transformations

- Query: Find the names of all customers with an account at a Brooklyn branch whose account balance is over $1000.

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie (account \bowtie depositor)))$$

- Transformation using join associatively (Rule 6a):

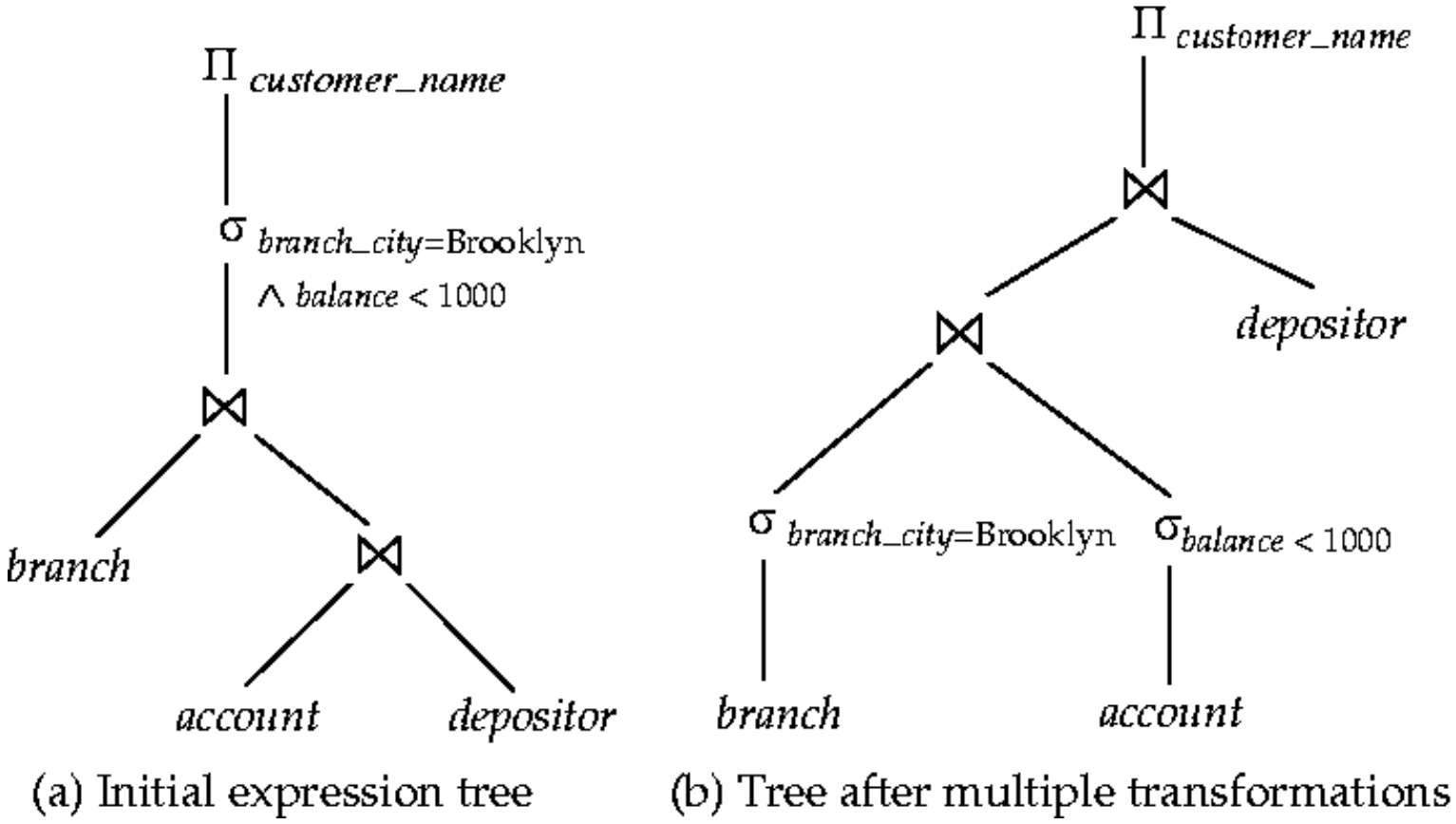
$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”} \wedge balance > 1000}(branch \bowtie account)) \bowtie depositor)$$

- Second form provides an opportunity to apply the “perform selections early” rule, resulting in the subexpression

$$\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie \sigma_{balance > 1000}(account)$$

- Thus a sequence of transformations can be useful

Multiple Transformations (Cont.)



(a) Initial expression tree

(b) Tree after multiple transformations

Transformation Example: Pushing Projections

$$\Pi_{customer\_name}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account) \bowtie depositor)$$

- When we compute

$$(\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account)$$

  we obtain a relation whose schema is:
  (*branch_name, branch_city, assets, account_number, balance)*
- Push projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$$\Pi_{customer\_name}((\Pi_{account\_number}((\sigma_{branch\_city = \text{“Brooklyn”}}(branch) \bowtie account)) \bowtie depositor))$$

- Performing the projection as early as possible reduces the size of the relation to be joined.

Join Ordering Example

- For all relations r_1, r_2, and r_3,

$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$

(Join Associativity)

- If $r_2 \bowtie r_3$ is quite large and $r_1 \bowtie r_2$ is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{customer\_name}\ ((\sigma_{branch\_city = \text{“Brooklyn”}}\ (branch)) \bowtie (\text{account} \bowtie \text{depositor}))$$

- Could compute $account \bowtie depositor$ first, and join result with

$$\sigma_{branch\_city = \text{“Brooklyn”}}\ (branch)$$

but $account \bowtie depositor$ is likely to be a large relation.

- Only a small fraction of the bank's customers are likely to have accounts in branches located in Brooklyn
  - it is better to compute

$$\sigma_{branch\_city = \text{“Brooklyn”}}\ (branch) \bowtie account$$

first.

CS354: Database

Transaction Management, Concurrency Control and Recovery Control

Transaction

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g. 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*)

- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

- To preserve the integrity of data the database system must ensure:
  - **A**tomicity
  - **C**onsistency
  - **I**solation
  - **D**urability

Atomicity

- **Atomicity requirement**
  - Either all operations of the transaction are properly reflected in the database or none are.
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database

Consistency

- Execution of a transaction in isolation preserves the consistency of the database.
- 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*)
- **Consistency requirement** in above example:
  - the sum of A and B is unchanged by the execution of the transaction

Consistency (contd.)

- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

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.

Isolation (contd.)

- **Isolation requirement** - if between steps 3 and 6, another transaction T2 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).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B)	

Isolation (contd.)

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits

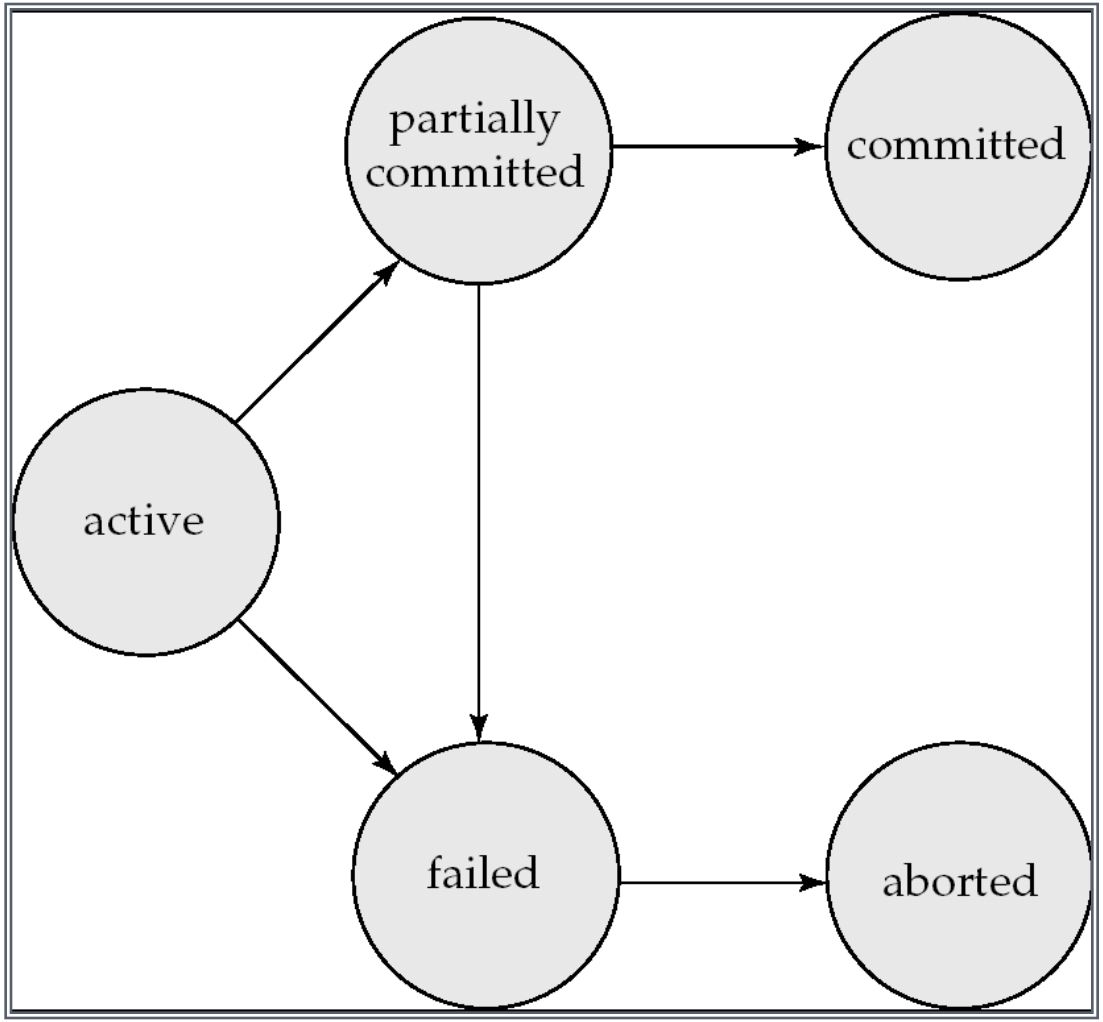
Durability

- **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 even if there are software or hardware failures.

Transaction State

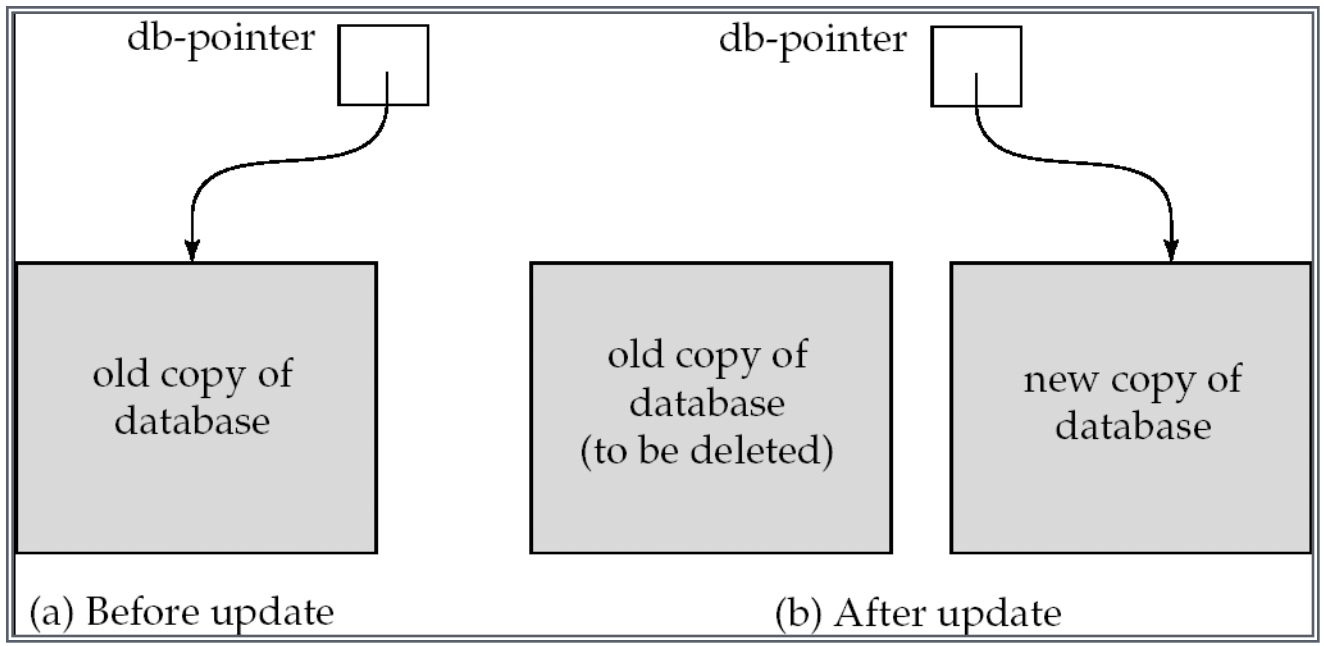
- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** – after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back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

Transaction state



Implementation of Atomicity and Durability

- The **recovery-management** component of a database system implements the support for atomicity and durability.
- E.g. the ***shadow-database*** scheme:
  - all updates are made on a *shadow copy* of the database
    - **db_pointer** is made to point to the updated shadow copy after
      - the transaction reaches partial commit and
      - all updated pages have been flushed to disk.



Implementation of Atomicity and Durability (Cont.)

- db_pointer always points to the current consistent copy of the database.
  - In case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.
- The shadow-database scheme:
  - Assumes that only one transaction is active at a time.
  - Assumes disks do not fail
  - Useful for text editors, but
    - extremely inefficient for large databases
  - Does not handle concurrent transactions

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Schedule

- A sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a *commit* instruction as the last statement
  - by default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an *abort* instruction as the last statement

Schedule 1

- Let T_1 transfers \$50 from A to B, and T_2 transfers 10% of the balance from A to B.
- A *serial* schedule in which T_1 is followed by T_2 :

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

SCHEDULE 2

- A serial schedule where T_2 is followed by T_1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

SCHEDULE 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In Schedules 1, 2 and 3, the sum A + B is preserved.

SCHEDULE 4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

- The above concurrent schedule does not preserve the value of $(A + B)$.

SERIALIZABILITY

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule.
- Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**
- *Simplified view of transactions*
  - We ignore operations other than **read** and **write** instructions
  - We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
  - Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

- Instructions l_i and l_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions wrote Q.
  1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
  3. l_i = **write**(Q), l_j = **read**(Q). They conflict
  4. l_i = **write**(Q), l_j = **write**(Q). They conflict
- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
  - If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

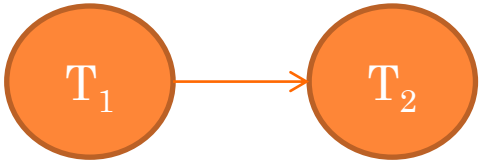
T_3	T_4
read(Q)	
	write(Q)
write(Q)	

- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n
- **Precedence graph** — a directed graph where the vertices are the transactions participating in the schedule
- We draw an edge from T_i to T_j if the two transaction conflict, i.e.,
  - T_i executes write(Q) before T_j executes read(Q)
  - T_i executes read(Q) before T_j executes write(Q)
  - T_i executes write(Q) before T_j executes write(Q)

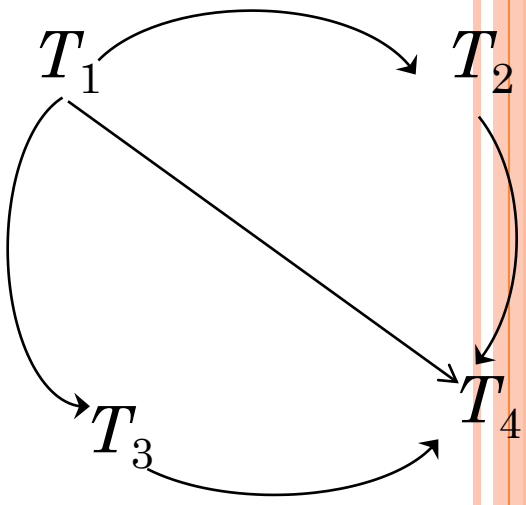
- We may label the arc by the item that was accessed.
- **Example 1**



- Now if an edge $T_1 \rightarrow T_2$ exists in the precedence graph, then, in any serial schedule S′ equivalent to S, T_1 must appear before T_2

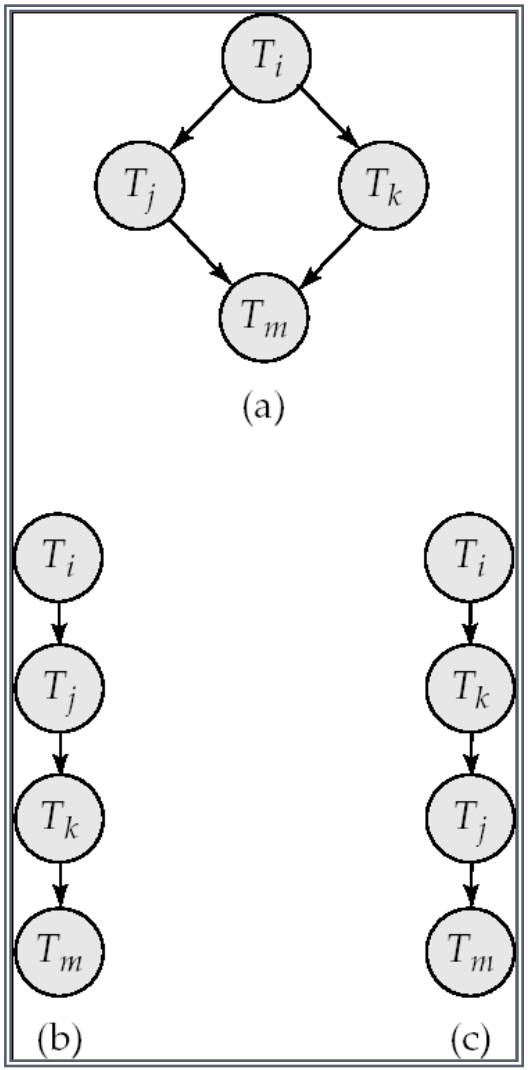
Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  - This is a linear order consistent with the partial order of the graph.
  - For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$
    - Are there others?



View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met, for each data item Q,
  1. If in schedule S, transaction T_i **reads** the initial value of Q, then in schedule S' also transaction T_i must **read** the initial value of Q.
  2. If in schedule S transaction T_i executes **read**(Q), and that value was produced by transaction T_j (if any), then in schedule S' also transaction T_i must read the value of Q that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(Q) operation in schedule S must also perform the final **write**(Q) operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also a view serializable but there are view-serializable schedules that are not conflict serializable.

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- The above schedule is view equivalent to serial schedule $<T_3, T_4, T_6>$
- Every view serializable schedule that is not conflict serializable has **blind writes**

View Serializability

- View serializability provides weaker conditions then conflict serializability
  - Still this will ensure serializability
- The key difference between view and conflict serializability appears when a transaction writes a value without reading it
- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.

Other Notions of Serializability

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- The schedule shown produces same outcome as the serial schedule $< T_1, T_5 >$
  - yet is not conflict equivalent or view equivalent to it.

Recoverability

- So far we have seen whether a schedule is acceptable from the viewpoint of consistency of the database
  - Implicit assumption- no transaction failures
- However if a transaction fails then we need to undo the effect of this transaction to ensure the atomicity property
- If concurrent execution is allowed then a transaction may depend on some other transaction
  - So in the case of a failure, if a transaction is aborted then the dependent transaction may also be aborted

Recoverable Schedules

- Let's consider the following schedule (Schedule 11)

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- It is not recoverable if T_9 commits immediately after the read
- If T_8 aborts, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.
- **Recoverable schedule** — if a transaction T_j reads a data item previously written by a transaction T_i, then the commit operation of T_i appears before the commit operation of T_j.

Cascading Rollbacks

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks.
- Consider the following schedule where none of the transactions have yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work
- How to avoid cascading rollbacks?

Cascadeless Schedules

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - *either conflict or view serializable, and*
  - *are recoverable and preferably cascadeless*
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

  1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.

  2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager
- Transaction can proceed only after request is granted.

Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is ***compatible*** with locks already held on the item by other transactions
- Any number of transactions can hold *shared locks* on an item,
  - but if any transaction holds an *exclusive lock* on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made *to wait till all incompatible locks held by other transactions have been released*
  - the lock is then granted.

Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

T_1: **lock-X**(*B*);
read (*B*);
B:=B-50;
write (*B*)
unlock(*B*);
lock-X(*A*);
read (*A*);
A:=A+50;
write (*A*);
unlock(*A*);

Lock-Based Protocols (Cont.)

- Example of another transaction performing locking:

  T_2: **lock-S**(*A*);
  read (*A*);
  unlock(*A*);
  lock-S(*B*);
  read (*B*);
  unlock(*B*);
  display(*A*+*B*)

- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- A **locking protocol** is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

Pitfalls of Lock-Based Protocols

- Consider the partial schedule

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing **lock-S**(*B*) causes T_4 to wait for T_3 to release its lock on *B*, while executing **lock-X**(*A*) causes T_3 to wait for T_4 to release its lock on *A*.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.

Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols.
- Deadlocks are a necessary evil
  - Preferable to inconsistent states

Starvation

- **Starvation** is also possible if concurrency control manager is badly designed.
- For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.

Granting of locks

- Concurrency control manager can be designed to prevent *starvation*
- When a transaction T_i requests a lock on data item Q in a particular mode M, the concurrency control manager grants the lock provided that –
  - There is no other transaction holding a lock on Q in a mode that conflicts with M
  - There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i
- Thus a lock request will never get blocked by a lock request that is made later

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedule.
- This protocol issues lock and unlock requests in two phases:
- **Phase 1: Growing Phase**
  - A transaction may obtain locks, but may not release locks
- **Phase 2: Shrinking Phase**
  - A transaction may release locks but may not obtain any new locks
- The transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).

The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Two phase locking protocol (contd.)

- Cascading roll-back is also possible under two-phase locking.

T5	T6	T7
lock-X(A) read (A) lock-S(B) read (B) write (A) unlock (A)		
	lock-X(A) read (A) write (A) unlock (A)	
		lock-S(A) read (A)

- Cascading rollbacks can be avoided by
  - Following a modified protocol called **strict two-phase locking**. Here a transaction must hold all its *exclusive locks* till it commits/aborts.
  - **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort.

Lock Conversions

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)
- This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.

- Consider the following two transactions-

T_8

Read(A_1)
Read(A_2)
…
Read(A_n)
Write(A_1)

T_9

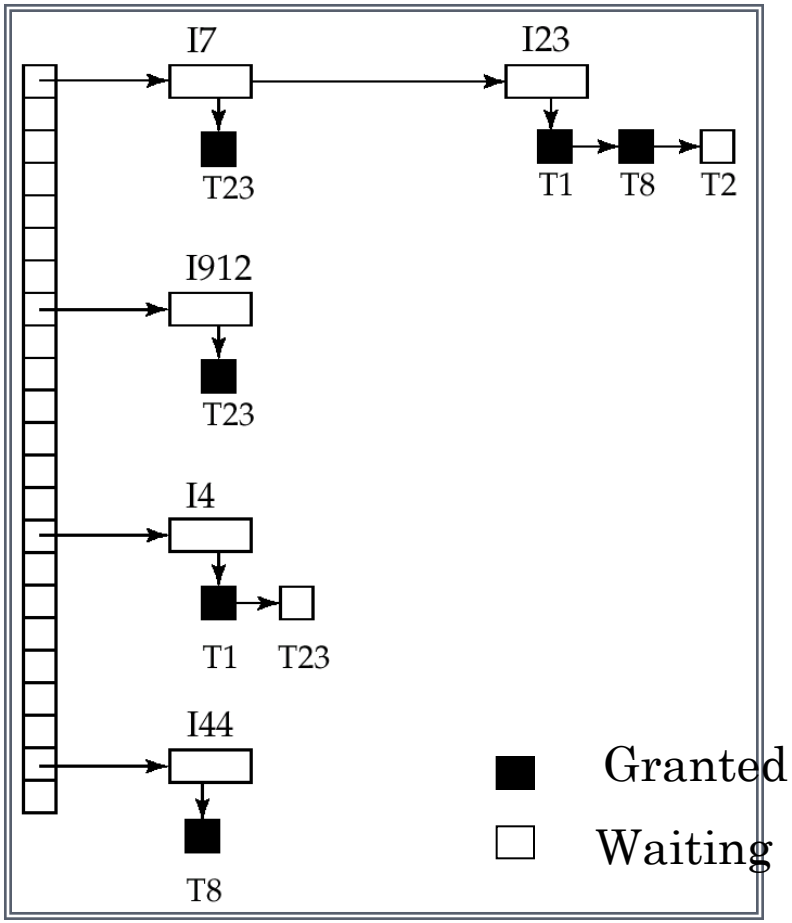
Read(A_1)
Read(A_2)
display(A_1+ A_2)

If two phase locking protocol is employed, then T_8 must lock A_1 in exclusive mode
However, T_8 needs the exclusive mode lock only at the end
So initially it can lock A_1 in shared mode and later upgrade to exclusive mode
This will allow more concurrency

Implementation of Locking

- A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered
- The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests
- The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked

Lock Table



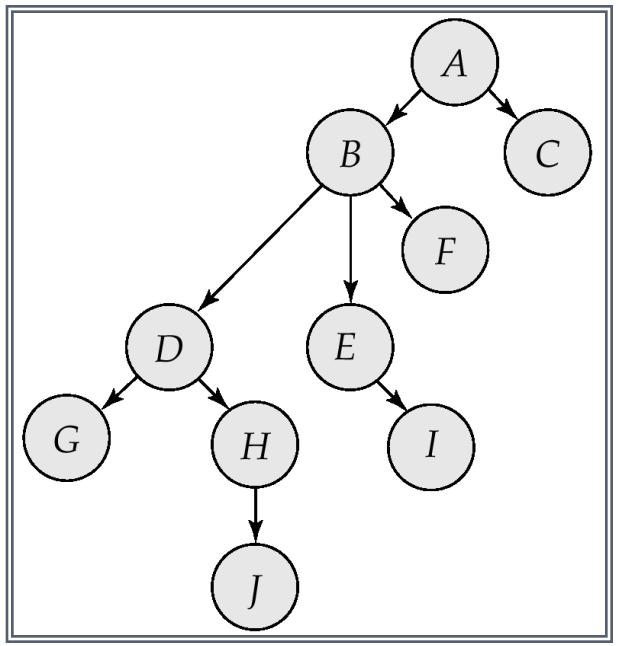
- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering $\rightarrow$ on the set **D** = $\{d_1, d_2, \ldots, d_h\}$ of all data items.
  - If $d_i \rightarrow d_j$ then any transaction accessing both d_i and d_j must access d_i before accessing d_j.
  - Implies that the set **D** may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol.

Tree Protocol

1. Only exclusive locks are allowed.
2. The first lock by T_i may be on any data item. Subsequently, a data Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i



SCHEDULE USING THE TREE PROTOCOL

T_{10}	T_{11}	T_{12}	T_{13}
Lock-X(B)			
	Lock-X(D)		
	Lock-X(H)		
	Unlock (D)		
Lock-X(E)			
Lock-X(D)			
Unlock(B)			
Unlock (E)			
		Lock-X(B)	
		Lock-X(E)	
	Unlock (H)		
Lock-X(G)			
Unlock (D)			
			Lock-X(D)
			Lock-X(H)
			Unlock (D)
			Unlock (H)
		Unlock (E)	
		Unlock (B)	
Unlock (G)			

Graph-Based Protocols (Cont.)

- **Advantages:**
  - The tree protocol ensures conflict serializability
  - No rollback is required as it is free from deadlock
  - Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
    - shorter waiting times, and increase in concurrency
- **Disadvantages:**
  - Protocol does not guarantee recoverability or cascade freedom
    - Need to introduce commit dependencies to ensure recoverability
  - Transactions may have to lock data items that they do not access.
    - increased locking overhead, and additional waiting time
    - potential decrease in concurrency

Deadlock Handling

- Consider the following two transactions:

  T_1: write (A) T_2: write(B)

  write(B) write(A)

- Schedule with deadlock

T_1	T_2
lock-X on A	
write (A)	
	lock-X on B
	write (B)
	wait for **lock-X** on A
wait for **lock-X** on B	

Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- ***Deadlock prevention*** protocols ensure that the system will *never* enter into a deadlock state.
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

More Deadlock Prevention Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- **wait-die** scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- **wound-wait** scheme — preemptive
  - older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
  - may be fewer rollbacks than *wait-die* scheme.

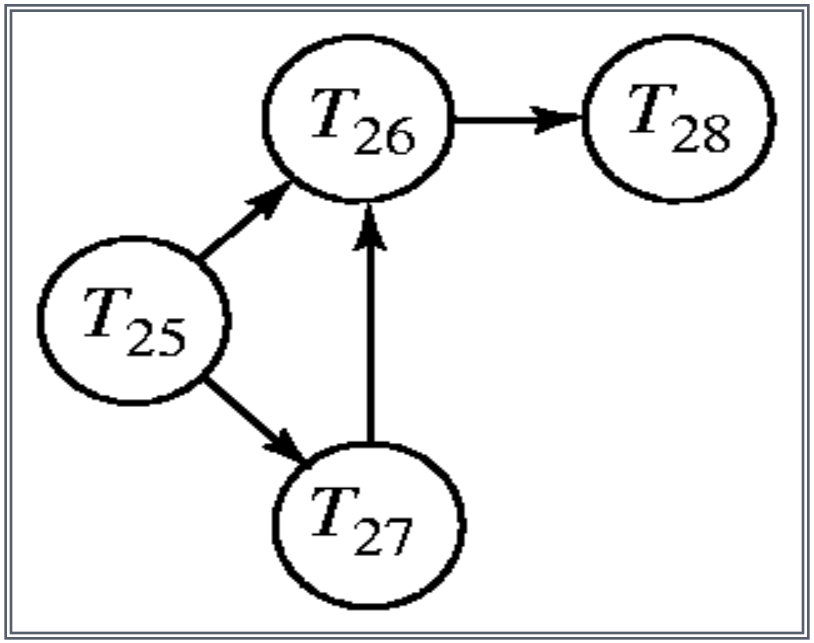
Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes
  - A rolled back transaction is restarted with its original timestamp.
  - Older transactions thus have precedence over newer ones, and starvation is hence avoided.
- Timeout-Based Schemes :
  - A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  - Thus deadlocks are not possible
  - Simple to implement; but starvation is possible.
  - Also difficult to determine good value of the timeout interval.

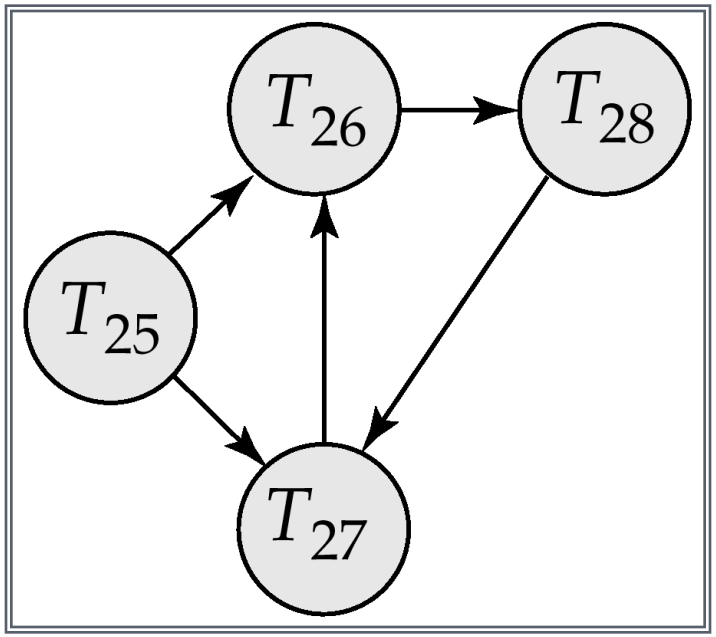
Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle. Must invoke a deadlock-detection algorithm periodically to look for cycles.

Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

Deadlock Recovery

- When deadlock is detected
  - The system must recover from the deadlock state
  - The most common solution is to roll back one or more transactions
- Actions to be taken
  - Selection of a victim
  - Rollback
  - Starvation

Selection of a victim

- Some transactions will have to be rolled back (made a victim) to break deadlock
- Select that transaction as victim that will incur minimum cost while rolling back
- Depends on
  - How long the transaction has computed, and how much longer the transaction will compute before it completes it designated task
  - How many data items the transaction has used
  - How many more data items the transactions needs for it to complete
  - How many transactions will be involved in the rollback

Rollback

- How far a particular transaction should be rolled back
  - Total rollback: abort the transaction and restarts it
  - Partial rollback: rollback the transaction only as far as necessary to break deadlock
    - The system must maintain some additional information – sequence of lock requests/grants, updates performed by the transaction, etc.

Starvation

- The selection of a victim is based on cost factor
- Starvation happens if same transaction is always chosen as victim
- As a result this transaction never completes its designated task
- A transaction can be picked as a victim only a finite no. of times
- A possible solution:
  - Include the number of rollbacks in the cost factor to avoid starvation

Recovery System

- A computer system is subject to failure from a variety of causes
- In any failure data may be lost
- So a good database system must take actions in advance to ensure that the atomicity and durability properties are preserved
- Recovery system is an integral part of a database system
- The recovery system does the followings-
  - restore the database to a consistent state
  - provide high availability

Failure Classification

- **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption**: non-volatile storage contents are assumed not to be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use checksums to detect failures

Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing
     - to ensure enough information exists to recover from failures
  2. Actions taken after a failure
     - to recover the database contents to a state that ensures atomicity, consistency and durability

Recovery and Atomicity

- To ensure atomicity despite failures
  - we first output information describing the modifications to stable storage without modifying the database itself.
- We study the following approach:
  - **log-based recovery**
- For simplicity, we assume that transactions run serially

Log-Based Recovery

- A **log** is kept on stable storage
  - The log is a sequence of records to keep track of all the update activities in the database
- The fields of a log record
  - Transaction identifier
  - Data item identifier
  - Old value
  - New value

Log-Based Recovery (contd.)

- Update log record:
  - $<T_i, X_j, V_1, V_2>$
    - Transaction T_i has performed a write on data item X_j. X_j had old value V_1 before the write, and will have value V_2 after the write.
- Special log records to record significant events in transaction processing:
  - $<T_i$ **start**$>$
    - Transaction T_i has started
  - $<T_i$ **commit**$>$
    - Transaction T_i has committed
  - $<T_i$ **abort**$>$
    - Transaction T_i has aborted

- We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **writes** to after partial commit.
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A write(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
  - The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.

Deferred Database Modification (Cont.)

- During recovery after a crash, a transaction needs to be redone if and only if both $<T_i$ **start**> and $<T_i$ **commit**> are there in the log.
- Redoing a transaction T_i (**redo** T_i) sets the value of all data items updated by the transaction to the new values.
- **redo** operation must be idempotent
  - Executing it several times must be equivalent to executing it once
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken

- Example transactions T_0 and T_1 (T_0 executes before T_1):

T_0: **read** (A)
$A := A - 50$
write (A)
read (B)
$B := B + 50$
write (B)

T_1 : **read** (C)
$C := C - 100$
write (C)

Deferred Database Modification (Cont.)

- Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
$<T_0$ start>	$<T_0$ start>	$<T_0$ start>
$<T_0, A, 950>$	$<T_0, A, 950>$	$<T_0, A, 950>$
$<T_0, B, 2050>$	$<T_0, B, 2050>$	$<T_0, B, 2050>$
	$<T_0$ commit>	$<T_0$ commit>
	$<T_1$ start>	$<T_1$ start>
	$<T_1, C, 600>$	$<T_1, C, 600>$
		$<T_1$ commit>

- If log on stable storage at time of crash is as in case:
  - (a) No redo actions need to be taken
  - (b) redo(T_0) must be performed since $<T_0$ **commit**> is present
  - (c) **redo**(T_0) must be performed followed by redo(T_1) since $<T_0$ **commit**> and $<T_1$ commit> are present

Immediate Database Modification

- The **immediate database modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(*B*) operation for a data block B, all log records corresponding to items *B* must be flushed to stable storage
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Immediate Database Modification Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	$A = 950$	
	$B = 2050$	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	$C = 600$	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

Immediate Database Modification (Cont.)

- Recovery procedure has two operations instead of one:
  - **undo(T_i)** restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **redo(T_i)** sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i

- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery

Immediate Database Modification (Cont.)

- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record $<T_i$ **start**$>$, but does not contain the record $<T_i$ **commit**$>$.
  - Transaction T_i needs to be redone if the log contains both the record $<T_i$ **start**$>$ and the record $<T_i$ **commit**$>$.

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000.

(b) undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then C is set to 600

Common Problems in Recovery Procedure

- Searching the entire log is time-consuming
- Unnecessarily redo transactions which have already output their updates to the database

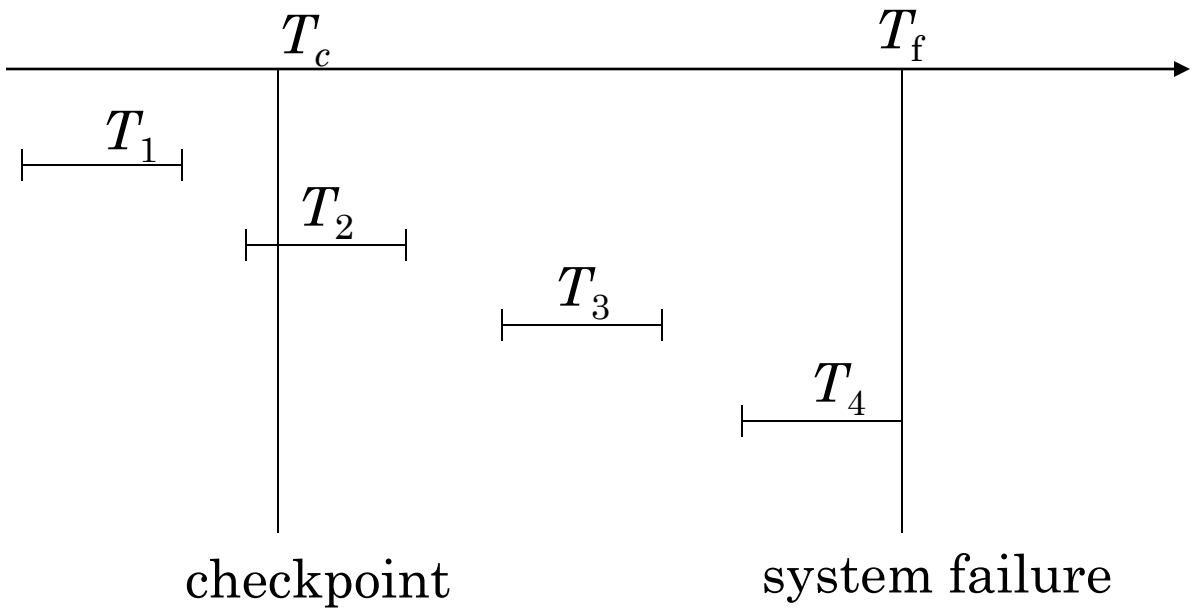
CHECKPOINTING

- Streamline recovery procedure by periodically performing **checkpointing**
- It performs the following sequence of operations-
  1. Output all log records currently residing in main memory onto stable storage.
  2. Output all modified buffer blocks to the disk.
  3. Write a log record < **checkpoint**> onto stable storage
- Transactions are not allowed to perform any update actions such as writing to buffer block or writing a log record, while a checkpoint is in progress

Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent **<checkpoint>** record
  2. Continue scanning backwards till a record $<T_i$ **start>** is found.
  3. Need only to consider the part of log following above **start** record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  4. For all transactions (starting from T_i or later) with no $<T_i$ **commit>**, execute **undo(***T_i***)**. (Done only in case of immediate modification.)
  5. Scanning forward in the log, for all transactions starting from T_i or later with a $<T_i$ **commit>**, execute **redo(***T_i***)**

Example of Checkpoints



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone