Principles of Database Systems (CS307)

Lecturer's Cut: Internal Mechanism of Databases

Yuxin Ma

Department of Computer Science and Engineering
Southern University of Science and Technology

- Most contents are from slides made by Stéphane Faroult and the authors of Database System Concepts (7th Edition).
- Their original slides have been modified to adapt to the schedule of CS307 at SUSTech.

Physical Storage System

Physical Storage System

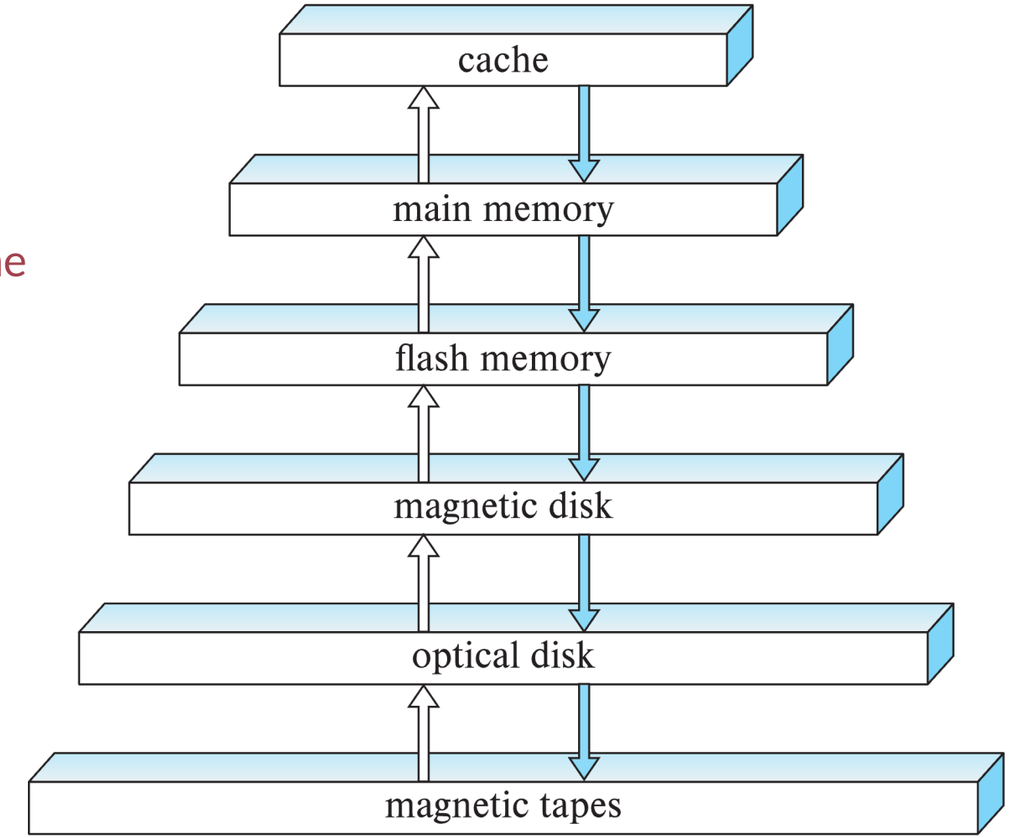
- The hardware where data is recorded

Classification of Physical Storage Media

- 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
- Factors affecting choice of storage media include
  - Speed with which data can be accessed
  - Cost per unit of data
  - Reliability

Storage Hierarchy

- 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 and used for archival storage
  - E.g., magnetic tape, optical storage
    - Magnetic tape
      - Sequential access, 1 to 12 TB capacity
      - A few drives with many tapes
      - Juke boxes with petabytes (1000's of TB) of storage

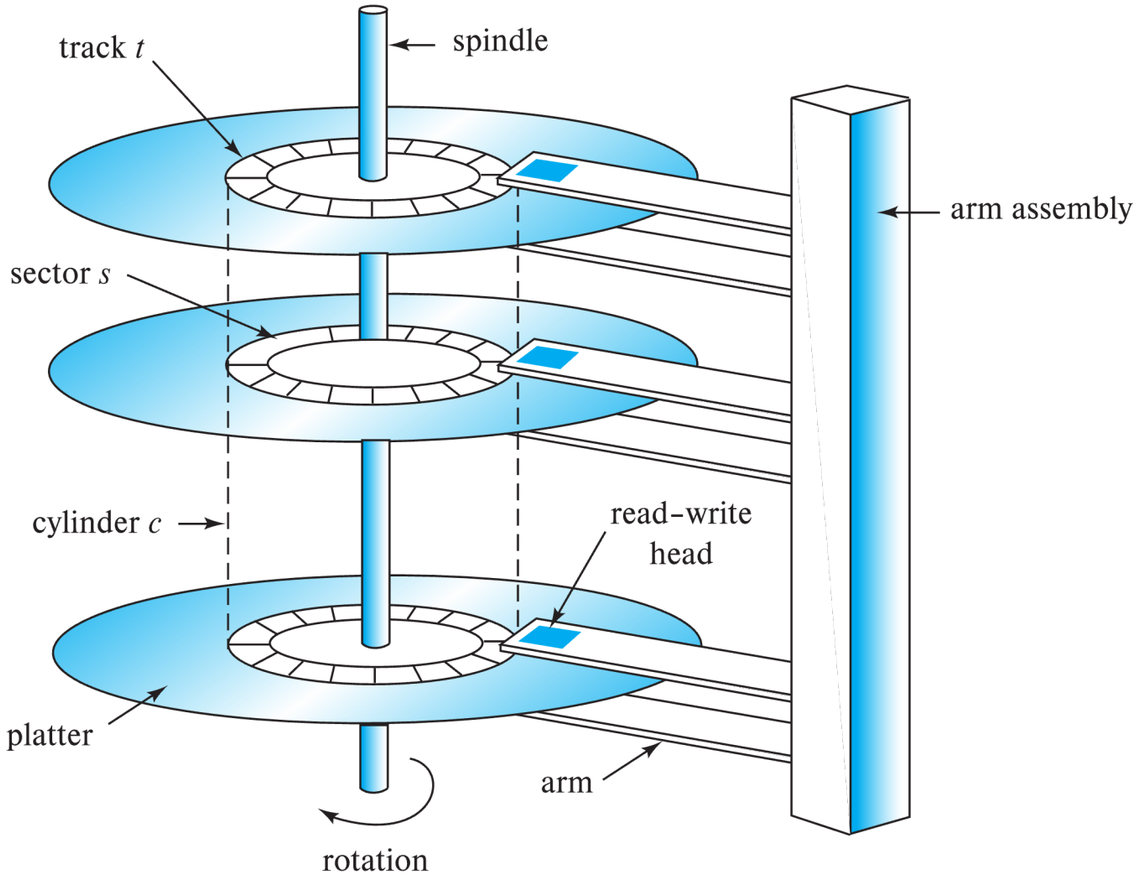


Storage Interfaces

- Disk interface standards families
  - SATA (Serial ATA)
    - SATA 3 supports data transfer speeds of up to 6 gigabits/sec
  - SAS (Serial Attached SCSI)
    - SAS Version 3 supports 12 gigabits/sec
  - NVMe (Non-Volatile Memory Express) interface
    - Works with PCIe connectors to support lower latency and higher transfer rates
    - Supports data transfer rates of up to 24 gigabits/sec

- Disks usually connected directly to computer system, however…
  - In Storage Area Networks (SAN), a large number of disks are connected by a high-speed network to a number of servers
  - In Network Attached Storage (NAS) networked storage provides a file system interface using networked file system protocol, instead of providing a disk system interface

Magnetic Hard Disk Mechanism



Schematic diagram of magnetic disk drive

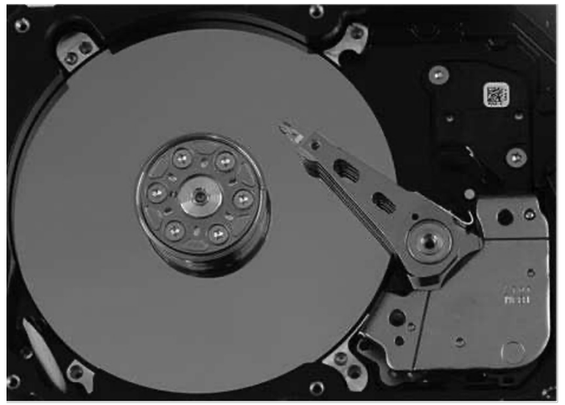
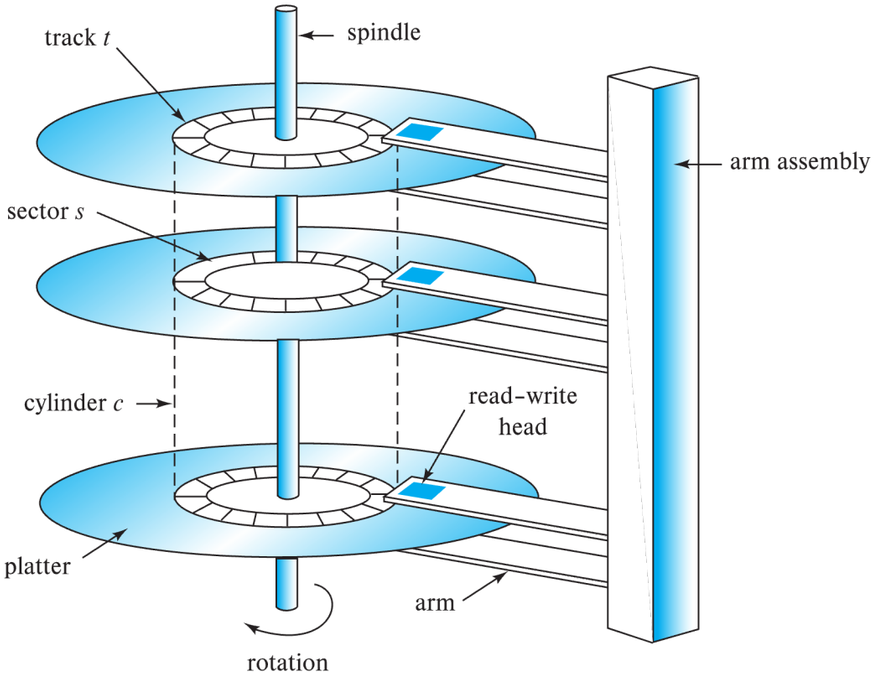


Photo of a magnetic disk drive

Magnetic Hard Disk Mechanism



- Read-write head
- 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 (modern OS requires 4KB)
  - 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 ith track of all the platters

Magnetic Hard Disk Mechanism

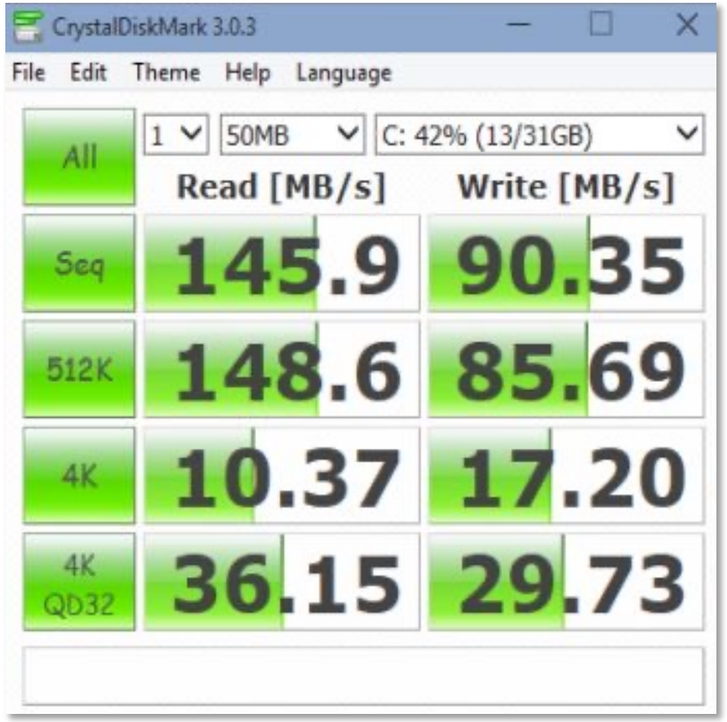
- Disk controller: An interface between the computer system and the disk drive hardware
  - Accept high-level commands to read or write a sector
  - Initiate actions such as moving the disk arm to the right track and reading or writing the data
  - Compute and attach checksums to each sector to verify that data is read back correctly
    - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
  - Ensure successful writing by reading back sector after writing it
  - Perform remapping of bad sectors

Performance Measures of Disks

- Access time: The time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - Seek time – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - Rotational latency – time it takes for the sector to be accessed to appear under the head.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - Average latency is 1/2 of the above latency.
  - Overall latency is 5 to 20 msec depending on disk model
- Data-transfer rate: The rate at which data can be retrieved from or stored to the disk.
  - 25 to 200 MB per second max rate, lower for inner tracks

Performance Measures of Disks

- Disk block is a logical unit for storage allocation and retrieval
  - 4 to 16 kilobytes typically
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
- Sequential access pattern
  - Successive requests are for successive disk blocks
  - Disk seek required only for first block
- Random access pattern
  - Successive requests are for blocks that can be anywhere on disk
  - Each access requires a seek
  - Transfer rates are low since a lot of time is wasted in seeks
- I/O operations per second (IOPS)
  - Number of random block reads that a disk can support per second
  - 50 to 200 IOPS on current generation magnetic disks



Performance Measures of Disks

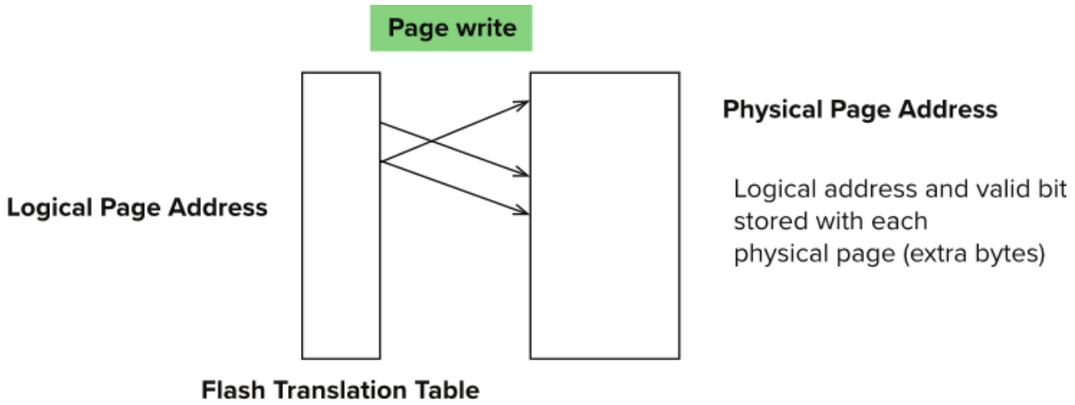
- Mean time to failure (MTTF) – the average time the disk is expected to run continuously without any failure.
  - Typically, 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a "theoretical MTTF" of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000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

Flash Storage

- NOR flash vs NAND flash
- NAND flash
  - Used widely for storage, cheaper than NOR flash
  - Requires page-at-a-time read (page: 512 bytes to 4 KB)
    - 20 to 100 microseconds for a page read
    - Not much difference between sequential and random read
  - Page can only be written once
    - Must be erased to allow rewrite
- Solid state disks (SSD)
  - Use standard block-oriented disk interfaces, but store data on multiple flash storage devices internally
  - Transfer rate of up to 500 MB/sec using SATA, and up to 3 GB/sec using NVMe PCIe

Flash Storage

- Erase happens in units of erase block
  - Takes 2 to 5 millisecs
  - Erase block typically 256 KB to 1 MB (128 to 256 pages)
- Remapping of logical page addresses to physical page addresses avoids waiting for erase
- Flash translation table tracks mapping
  - Also stored in a label field of flash page
  - Remapping carried out by flash translation layer



- After 100,000 to 1,000,000 erases, erase block becomes unreliable and cannot be used
  - Wear leveling

SSD Performance Metrics

- Random reads/writes per second
  - Typical 4KB reads: 10,000 reads per second (10,000 IOPS)
  - Typical 4KB writes: 40,000 IOPS
  - SSDs support parallel reads
    - Typical 4KB reads:
      - 100,000 IOPS with 32 requests in parallel (QD-32) on SATA
      - 350,000 IOPS with QD-32 on NVMe PCIe
    - Typical 4KB writes:
      - 100,000 IOPS with QD-32, even higher on some models
- Data transfer rate for sequential reads/writes
  - 400 MB/sec for SATA3, 2 to 3 GB/sec using NVMe PCIe

- Hybrid disks: Combine small amount of flash cache with larger magnetic disk

Storage Class Memory

- 3D-XPoint memory technology pioneered by Intel
- Available as Intel Optane
  - SSD interface shipped from 2017
    - Allows lower latency than flash SSDs
  - Non-volatile memory interface announced in 2018
    - Supports direct access to words, at speeds comparable to main-memory speeds

Magnetic Tapes

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

RAID

- RAID: Redundant Arrays of Independent Disks
  - Disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
    - high capacity and high speed by using multiple disks in parallel
    - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks

Improvement of Reliability via Redundancy

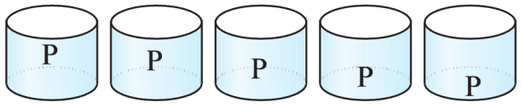
- Redundancy – Store extra information that can be used to rebuild information lost in a disk failure
  - E.g., Mirroring (or shadowing)
    - Duplicate every disk. Logical disk consists of two physical disks.
    - Every write is carried out on both disks
      - Reads can take place from either disk
    - If one disk in a pair fails, data still available in the other
      - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
      - Except for dependent failure modes such as fire or building collapse or electrical power surges
  - Mean time to data loss depends on mean time to failure, and mean time to repair
    - E.g., MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of 500*106 hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)

RAID Levels

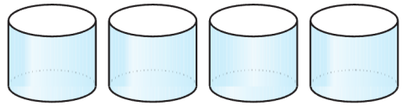
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics

  - RAID 0: Block striping; non-redundant
  - RAID 1: Mirrored disks with block striping
  - RAID 10: Combination of striping and mirroring

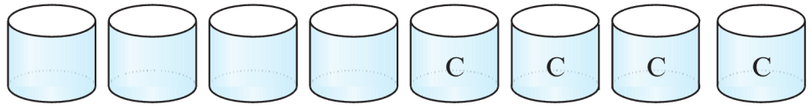
  - RAID 5: Block-interleaved distributed parity
  - RAID 6: P+Q Redundancy scheme



(c) RAID 5: block-interleaved distributed parity



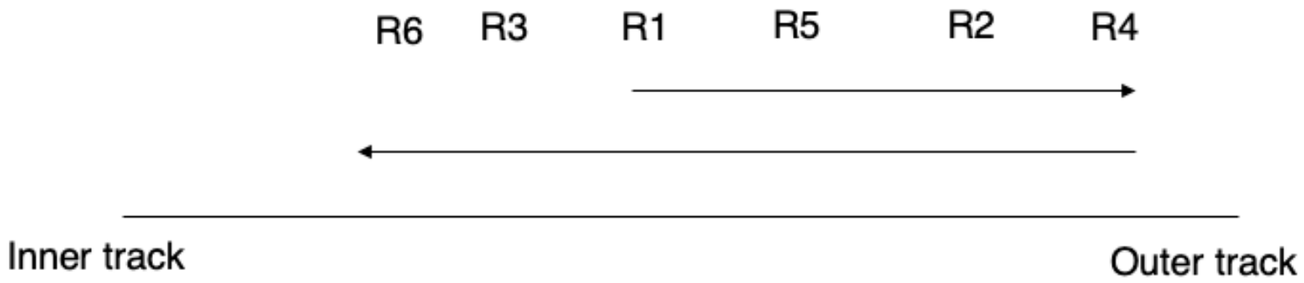
(a) RAID 0: nonredundant striping



(b) RAID 1: mirrored disks

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
  - E.g., elevator algorithm



Optimization of Disk-Block Access

- File organization
  - Allocate blocks of a file in as contiguous a manner as possible
  - Allocation in units of extents
  - Files may get fragmented
    - E.g., if free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
    - Some systems have utilities to defragment the file system, in order to speed up file access
- Non-volatile write buffers
  - Temporarily store the written data
    - … and immediately notifies the OS that writing is completed without errors
  - Write data into the disk when idle
    - … with some optimizations

(Logical) Data Storage Structure

File Organization

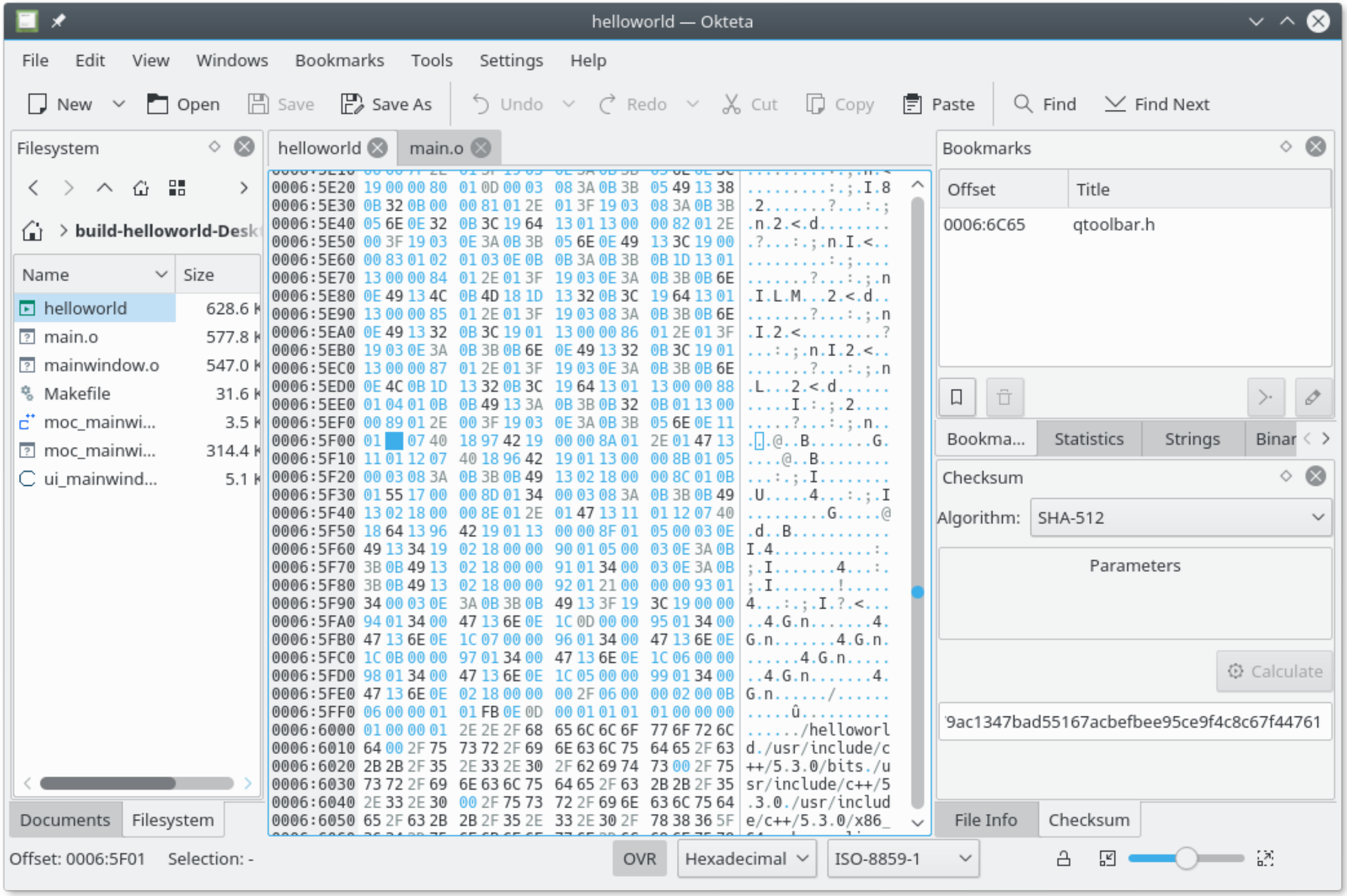
- The database is stored as a collection of files
  - Each file is a sequence of records
  - A record is a sequence of fields.
- One approach
  - Assume record size is fixed
  - Each file has records of one particular type only
  - Different files are used for different relations

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

- We assume that records are smaller than a disk block

File Organization

- Bitmap of a file



https://apps.kde.org/okteta/

File Organization

- Goals: Time and Space
  - Support CURD operations as fast as possible
  - Save storage space as much as possible

  - Also, to some extent, maintain data integrity

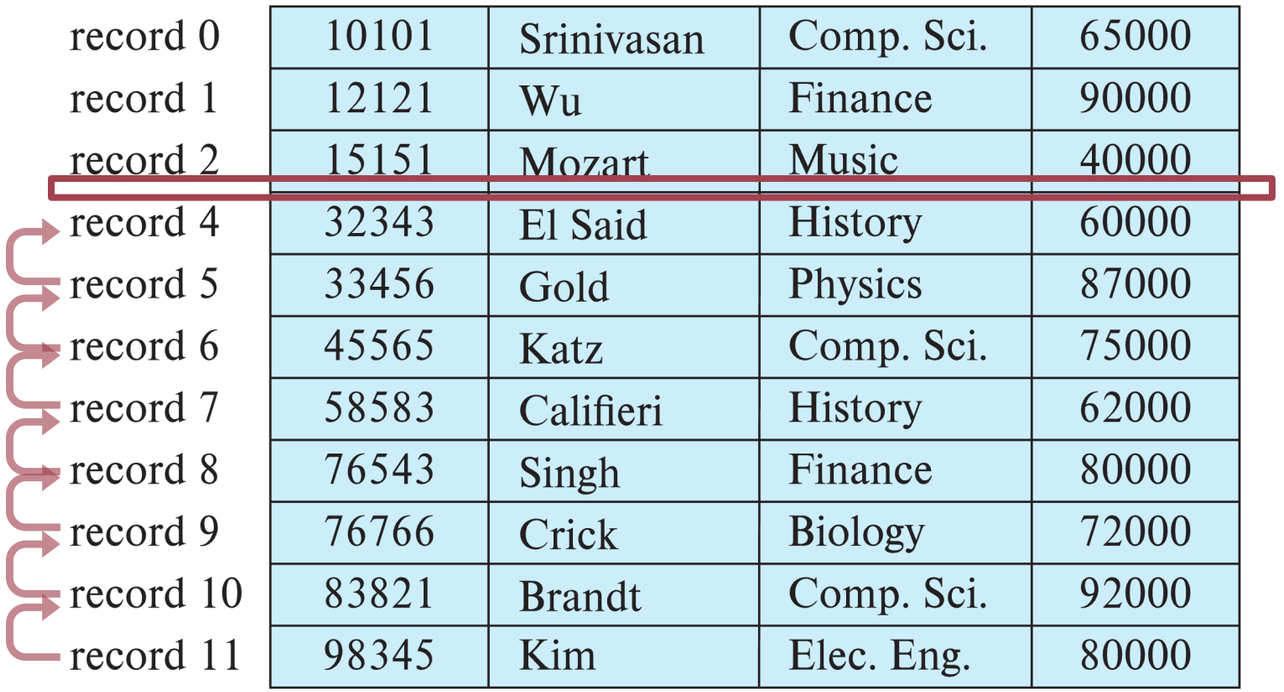
Fixed-Length Records

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

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

Fixed-Length Records

- Deletion of record i
  - Way #1: 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

Fixed-Length Records

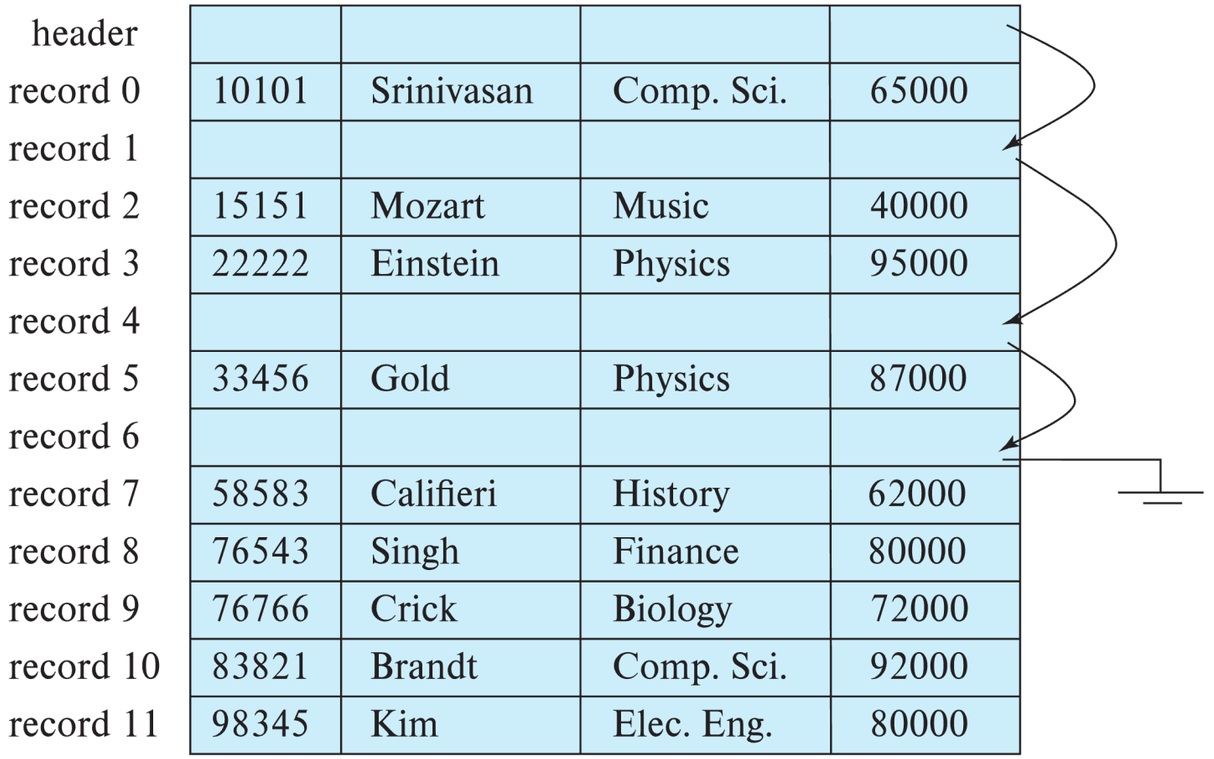
- Deletion of record i
  - Way #2: move record n to i
    - Record 3 is removed and replaced by record 11

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	98345	Kim	Elec. Eng.	80000

Fixed-Length Records

- Deletion of record i
  - Way #3: Do not move records, but link all free records on a *free list*



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

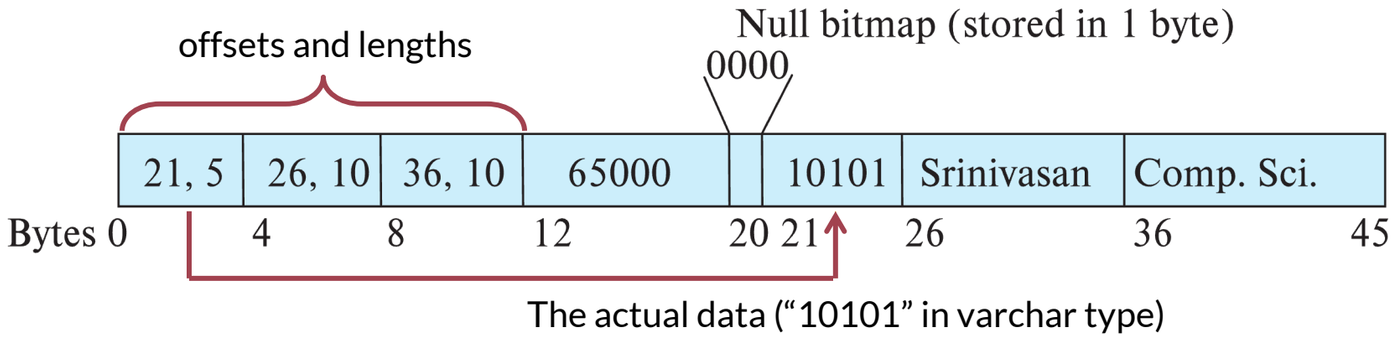
Variable-Length Records

- 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).

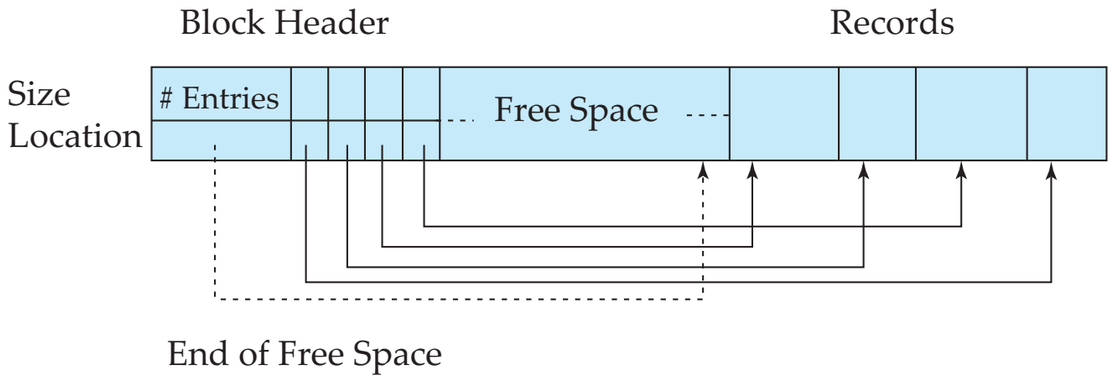
- Problem with variable-length records
  - How can we retrieve the data in an easy way without wasting too much space
    - `varchar(1000)`: do we really need to allocate 1000 bytes for this field, even if most of the actual data items only costs less than 10 bytes?

Variable-Length Records

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



Variable-Length Records



- Slotted page 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 records — instead, they should point to the entry for the record in header
- Page size is usually aligned with the disk block size (4KB-8KB)

Storing Large Objects

- E.g., BLOB/CLOB types
  - BLOB: Binary Large OBject
  - CLOB: Character Large OBject
- Records must be smaller than pages
- Alternatives:
  - Store as files in file systems
  - Store as files managed by databases
  - Break into pieces and store in multiple tuples in separate relation
    - PostgreSQL TOAST

Organization of Records in Files

- Heap – records 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
- 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
- B+-tree file organization
  - Ordered storage even with inserts/deletes
- Hashing – a hash function computed on search key; the result specifies in which block of the file the record should be placed

Heap File Organization

- Records can be placed anywhere in the file where there is free space
  - Records usually do not move once allocated

- Important to be able to efficiently find free space within file
- Free-space map
  - Array with 1 entry per block. Each entry is a few bits to a byte, and records fraction of block that is free
  - In example below, 3 bits per block, value divided by 8 indicates fraction of block that is free

4	2	1	4	7	3	6	5	1	2	0	1	1	0	5	6

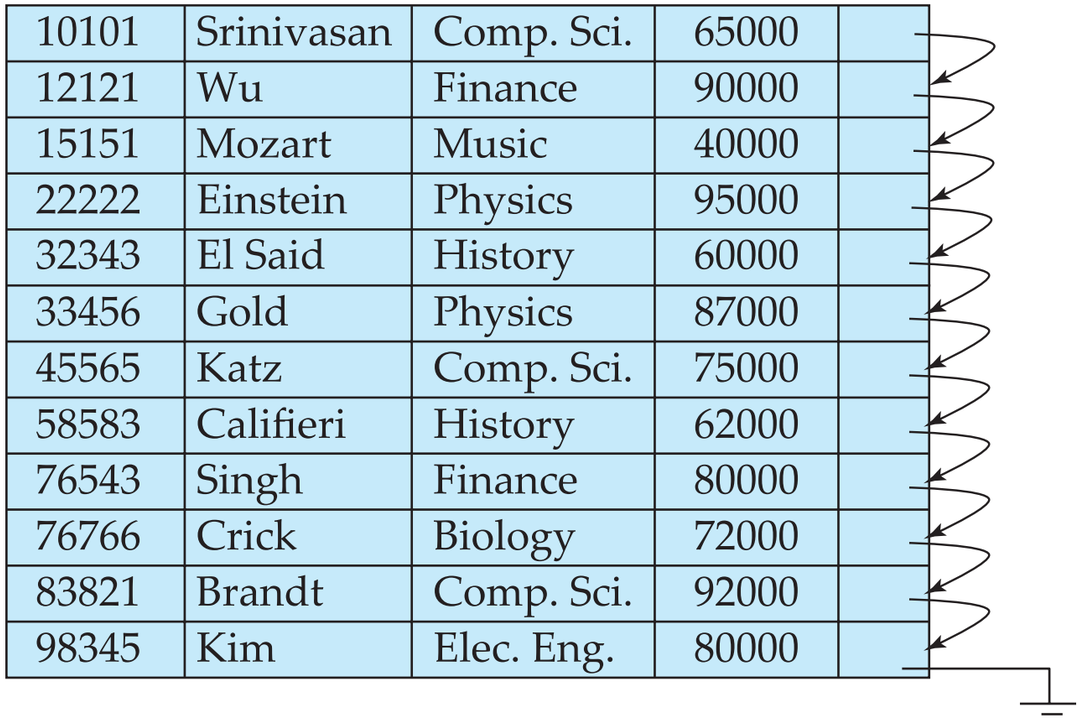
  - Can have second-level free-space map
    - In example below, each entry stores maximum from 4 entries of first-level free-space map

4	7	2	6

- Free space map written to disk periodically, OK to have wrong (old) values for some entries (will be detected and fixed)

Sequential File Organization

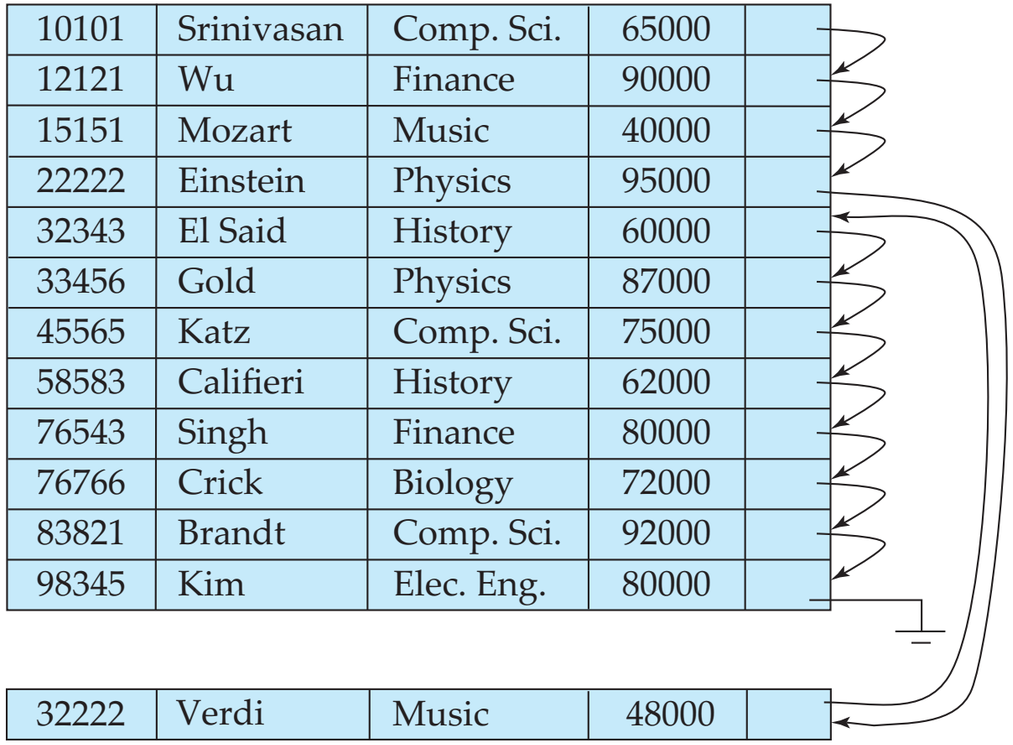
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key



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

Sequential File Organization

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



Multitable Clustering File Organization

- Store several relations in one file using a multitable clustering file organization
  - Good for queries involving:
    - *department* ⋈ *instructor*
    - or: one single department and its instructors (one-to-many correspondense)
  - Bad for queries involving only *department*
  - Results in variable size records
    - Can add pointer chains to link records of a particular relation
  - Bad for large databases
    - where other operations than joins are required

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	
10101	Srinivasan	Comp. Sci.	65000
45565	Katz	Comp. Sci.	75000
83821	Brandt	Comp. Sci.	92000
Physics	Watson	70000	
33456	Gold	Physics	87000

Partitioning

- Table partitioning: Records in a relation can be partitioned into smaller relations that are stored separately
  - E.g., `transaction` relation may be partitioned into `transaction_2018`, `transaction_2019`, etc.
- Queries written on transaction must access records in all partitions
  - Unless query has a selection such as year=2019, in which case only one partition in needed
- Partitioning
  - Reduces costs of some operations such as free space management
  - Allows different partitions to be stored on different storage devices
    - E.g., transaction partition for current year on SSD, for older years on magnetic disk

Column-Oriented Storage

- (Also known as columnar representation)

- Store each attribute of a relation separately

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

Column-Oriented Storage

- Benefits:
  - Reduced IO if only some attributes are accessed
  - Improved CPU cache performance
  - Improved compression
    - Data in the same type can be compressed more efficiently
  - Vector processing on modern CPU architectures
- Drawbacks
  - Cost of tuple reconstruction from columnar representation
  - Cost of tuple deletion and update
  - Cost of decompression

Column-Oriented Storage

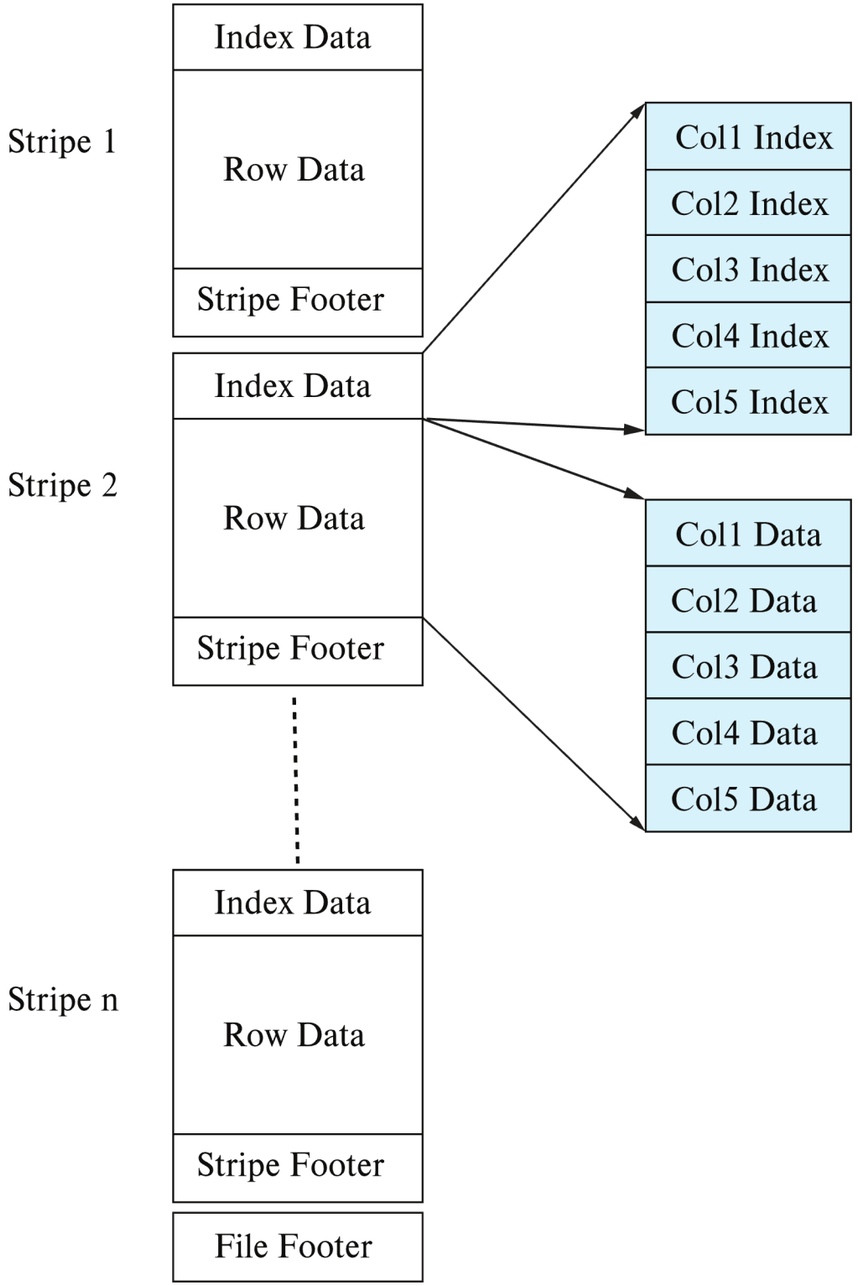
- Columnar representation found to be more efficient for decision support than row-oriented representation
- Traditional row-oriented representation preferable for transaction processing
- Some databases support both representations
  - Called hybrid row/column stores

Column-Oriented Storage

- ORC (Optimized Row Columnar) File Format
  - File format with columnar storage inside file

- Details if you are interested in big data processing:

`https://cwiki.apache.org/confluence/display/hive/languagemanual+orc`



Data Dictionary Storage

- The Data dictionary (also called system catalog) stores metadata
  - … that is, data about data – “meta” means “of”
- It includes
  - 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

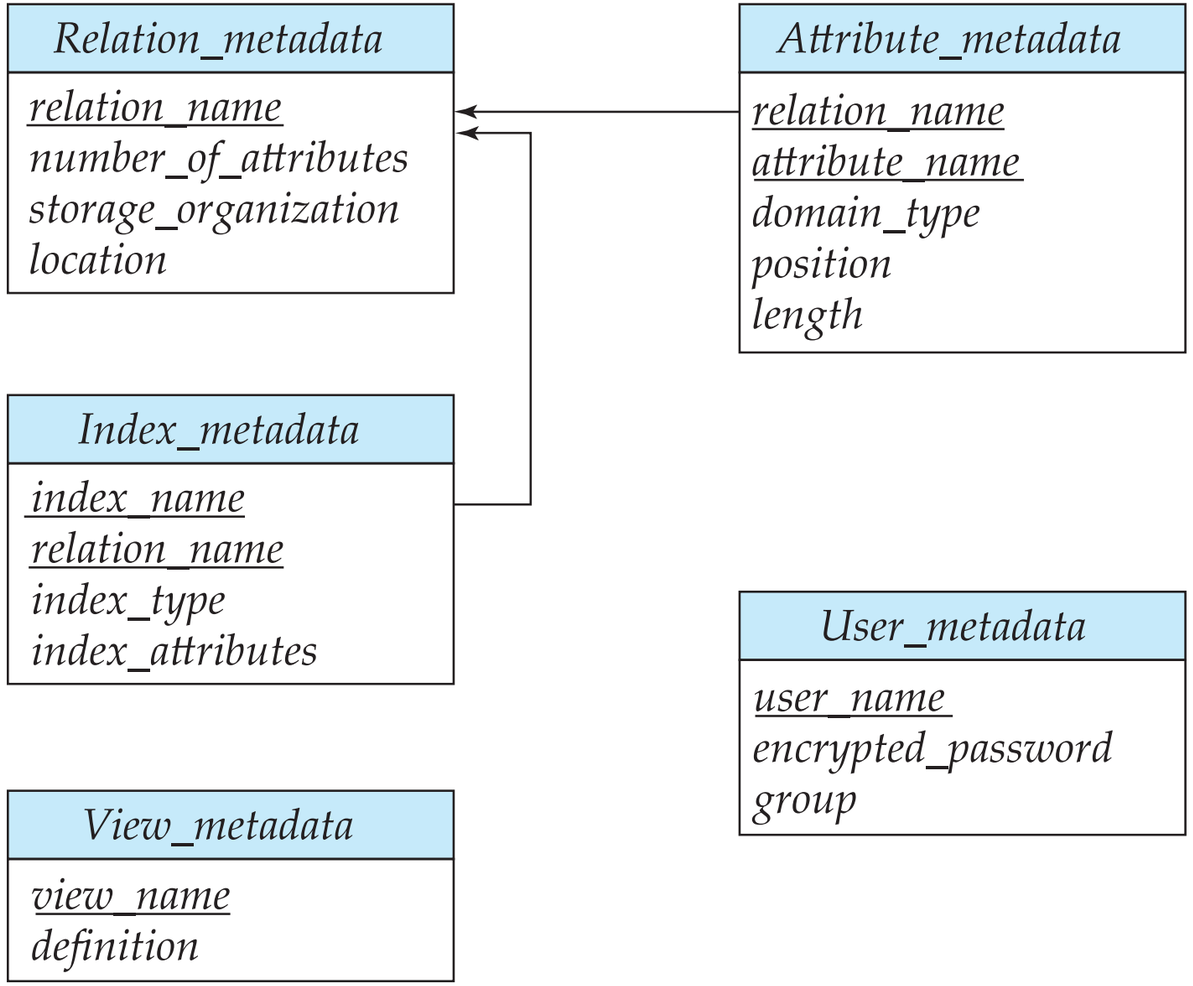
Data Dictionary Storage

- It includes (continue)
  - Physical file organization information
    - How relation is stored (sequential/hash/…)
    - Physical location of relation
  - Information about indices/indexes

(we will learn it later)

Relational Representation of System Metadata

- An example of the relational representation of the metadata
  - Relational representation on disk
  - Specialized data structures designed for efficient access, in memory
  - Different DBMS may have their own implementation



Storage Access

- Blocks are units of both storage allocation and data transfer
- Database system seeks to minimize the number of block transfers between the disk and memory
  - We can reduce the number of disk accesses by keeping as many blocks as possible in main memory

  - Buffer: Portion of main memory available to store copies of disk blocks
  - Buffer Manager: Subsystem responsible for allocating buffer space in main memory

Self Study

- Database System Concepts , 7th Edition
  - Chapter 13.5 “Database Buffer”

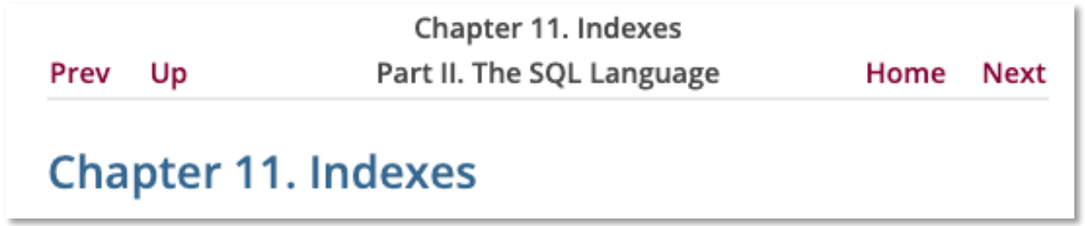
Indexing

Motivation

- Think about an example in a library:
  - How can we find a book?
    - Books are on the shelves in a sequential order
    - We had drawers where you could look for books by author, title or sometimes subject that were telling you what were the "***coordinates***" of a book.

Terminology

- Plural of index: indices, or indexes?
  - Both are correct in English
    - indices (Latin): Often used in scientific and mathematical context representing the places of an element in an array, vector, matrix, etc.
    - indexes (American English): Used in publishing for the books
  - What about database?
    - A good way: Follow the naming convention of the project or the DBMS



https://www.postgresql.org/docs/current/indexes.html

Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key

```
1   1,12 stulyev,ru,1971,161
2   2,Al-mummia test,eg,1969,102
3   3,"Ali Zaoua, prince de la rue",ma,2000,90
4   4,Apariencias,ar,2000,94
5   5,Ardh Satya,in,1983,130
6   6,Armaan,in,2003,159
7   7,Armaan,pk,1966,
8   8,Babettes gæstebud,dk,1987,102
9   9,Banshun,jp,1949,108
10  10,Bidaya wa Nihaya,eg,1960,
11  11,Variety,us,2008,106
12  12,"Bon Cop, Bad Cop",ca,2006,
13  13,Brilliantovaja ruka,ru,1969,100
14  14,C'est arrivé près de chez vous,be,1992,95
15  15,Carlota Joaquina - Princesa do Brasil,br,1995,
16  16,Cicak-man,my,2006,107
```

Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key
    - E.g., Find movies with IDs larger than 100 and smaller than 200

In the current storage structure, the records are sorted by `movieid`

- So, it will be easy to find a specific `movieid` with binary search

Searching for Record

- Remember searching algorithms in Data Structure?
  - Linear search
    - Scan all records from top to bottom
  - Binary search
    - Divide and conquer
    - Assumption: Records are **sorted** by the search key
  - However, how can we find data based on the non-sorted columns?
    - E.g., find all Chinese movies

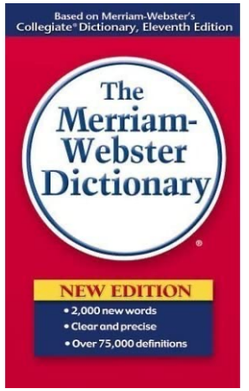
```
1   1,12 stulyev,ru,1971,161
2   2,Al-mummia test,eg,1969,102
3   3,"Ali Zaoua, prince de la rue",ma,2000,90
4   4,Apariencias,ar,2000,94
5   5,Ardh Satya,in,1983,130
6   6,Armaan,in,2003,159
7   7,Armaan,pk,1966,
8   8,Babettes gæstebud,dk,1987,102
9   9,Banshun,jp,1949,108
10  10,Bidaya wa Nihaya,eg,1960,
11  11,Variety,us,2008,106
12  12,"Bon Cop, Bad Cop",ca,2006,
13  13,Brilliantovaja ruka,ru,1969,100
14  14,C'est arrivé près de chez vous,be,1992,95
15  15,Carlota Joaquina - Princesa do Brasil,br,1995,
16  16,Cicak-man,my,2006,107
```

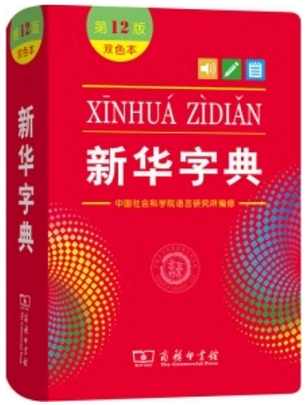
country

Find the rows `where country = 'cn'`

- The country codes are not sorted in the current storage structure, so the binary search algorithm cannot be used

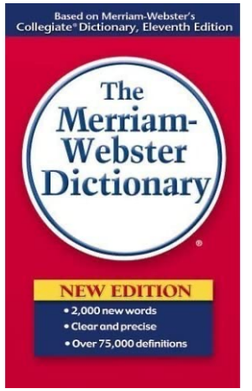
Searching for Record

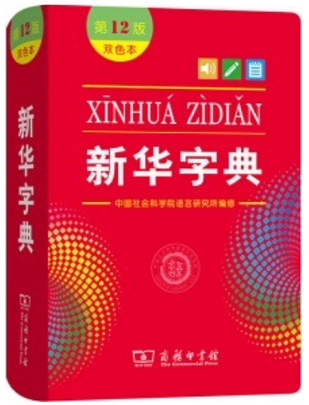




- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are usually sorted in the alphabetical order of Pinyin

Searching for Record





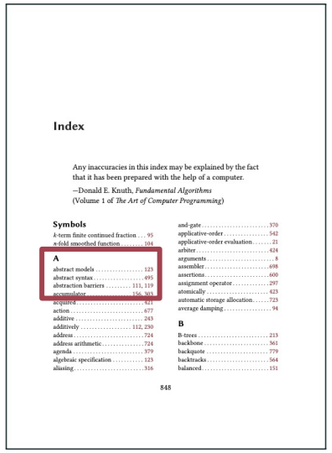
- This happens in real life too
  - English dictionary
    - The words are sorted in an alphabetical order
  - Chinese dictionary
    - The characters are sorted in the alphabetical order of Pinyin
    - However, we have other ways of looking up a character
      - Radicals (偏旁部首)
      - Number of strokes (数笔画)
      - Four-corner method (四角号码)

Index in Databases

- Concept
  - An **index** is a data structure which improves the efficiency of retrieving data with specific values from a database
  - Usually, indexes locate a row by a series of location indicators
    - E.g., (filename, block number, offset)

Index in Databases

- Concept
  - An **index** is a data structure which improves the efficiency of retrieving data with specific values from a database
  - Usually, indexes locate a row by a series of location indicators
    - E.g., (filename, block number, offset)
  - It is like indexes in books
    - Location indicator: (page, row)


Index

Any inaccuracies in this index may be explained by the fact that it has been prepared with the help of a computer.

—Donald E. Knuth, *Fundamental Algorithms* (Volume 1 of *The Art of Computer Programming*)

Symbols

4-term finite continued fraction . . . 95
n-fold smoothed function . . . 104

A

abstract models . . . 123
abstract syntax . . . 495
abstraction barriers . . . 111, 119
accumulator . . . 156, 303
acquired . . . 617
action . . . 677
additive . . . 243
additively . . . 112, 230
address . . . 724
address arithmetic . . . 724
agenda . . . 379
algebraic specification . . . 123
aliasing . . . 316
and-gate . . . 370
applicative-order . . . 542
applicative-order evaluation . . . 21
arbiter . . . 424
arguments . . . 8
assembler . . . 698
assertions . . . 600
assignment operator . . . 297
atomically . . . 423
automatic storage allocation . . . 723
average damping . . . 94

B

B-trees . . . 233
backbone . . . 361
backquote . . . 779
backtracks . . . 564
balanced . . . 151

848

A

abstract models 123
abstract syntax 495
abstraction barriers 111, 119
accumulator 156, 303

Index in Databases

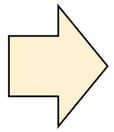
- Actually, we have been benefited from indexes off-the-shelf

  - In PostgreSQL, indexes are built automatically on columns with `primary key` or `unique` constraints

Experiment on Using Indexes

- Duplicate a table with no index

```
create table movies_no_index as select * from movies;
```



```
-- auto-generated definition
create table movies_no_index
(
    movieid       integer,
    title         varchar(100),
    country       char(2),
    year_released integer,
    runtime       integer,
    user_name     varchar(20)
);
```

Experiment on Using Indexes

- Check the performance on retrieving data
  - Significant difference between queries on the two tables

```sql
-- Query 1
explain analyze
    select *
    from movies
    where movieid > 100 and movieid < 300;

-- Query 2
explain analyze
    select *
    from movies_no_index
    where movieid > 100 and movieid < 300;
```

Query 1
(on `movies`)

```
QUERY PLAN
Bitmap Heap Scan on movies  (cost=10.32..136.35 rows=199 width=40) (actual time=0.162..0.440 rows=199 loops=1)
  Recheck Cond: ((movieid > 100) AND (movieid < 300))
  Heap Blocks: exact=6
  ->  Bitmap Index Scan on movies_pkey  (cost=0.00..10.28 rows=199 width=0) (actual time=0.136..0.136 rows=199 loops=1)
        Index Cond: ((movieid > 100) AND (movieid < 300))
Planning Time: 0.413 ms
Execution Time: 0.507 ms
```

Query 2
(on `movies_no_index`)

```
QUERY PLAN
Seq Scan on movies_no_index  (cost=0.00..217.06 rows=199 width=40) (actual time=0.039..5.075 rows=199 loops=1)
  Filter: ((movieid > 100) AND (movieid < 300))
  Rows Removed by Filter: 9005
Planning Time: 0.444 ms
Execution Time: 5.156 ms
```

Experiment on Using Indexes

- If there is no index on a column (or several columns), we can create one manually

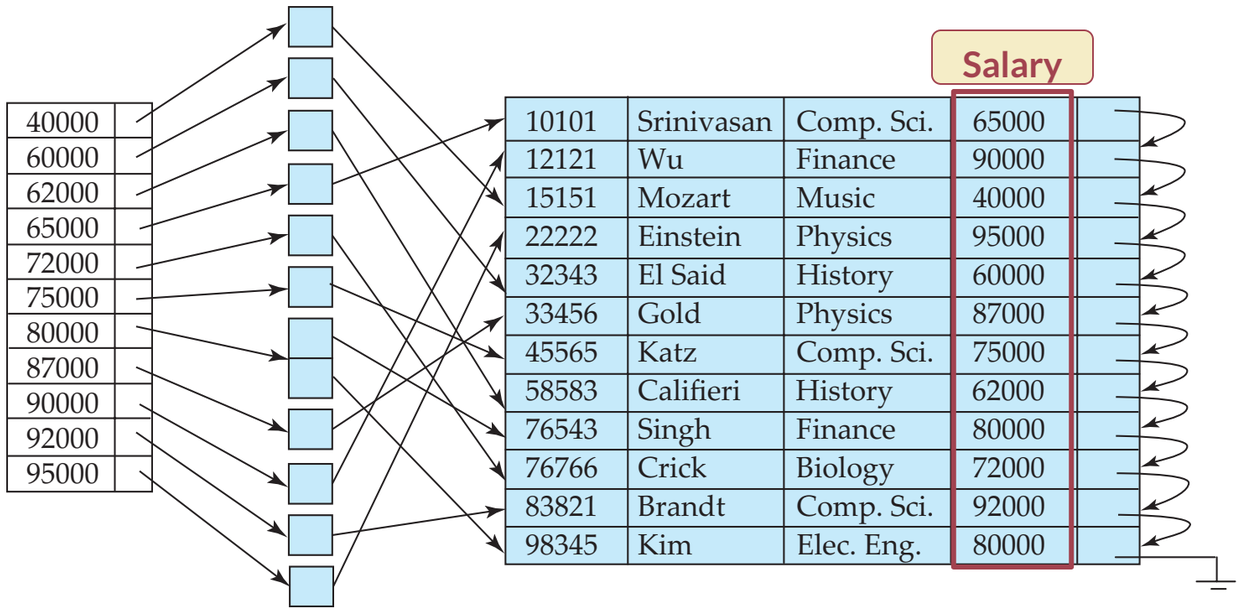
```
-- SQL Syntax for creating indexes
create index index_name
on table_name (column_name [, ...]);
```

Index Taxonomy

- 1) In terms of storage structure, is the index completely separated with the data records?
  - No ⇒ **Integrated index**
    - PK index in a MySQL InnoDB database
    - PK index in a SQL Server database
  - Yes ⇒ **External index**
    - Indexes in a PostgreSQL database
    - Indexes in a MySQL MyISAM database

Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)



A secondary index on the column "salary"

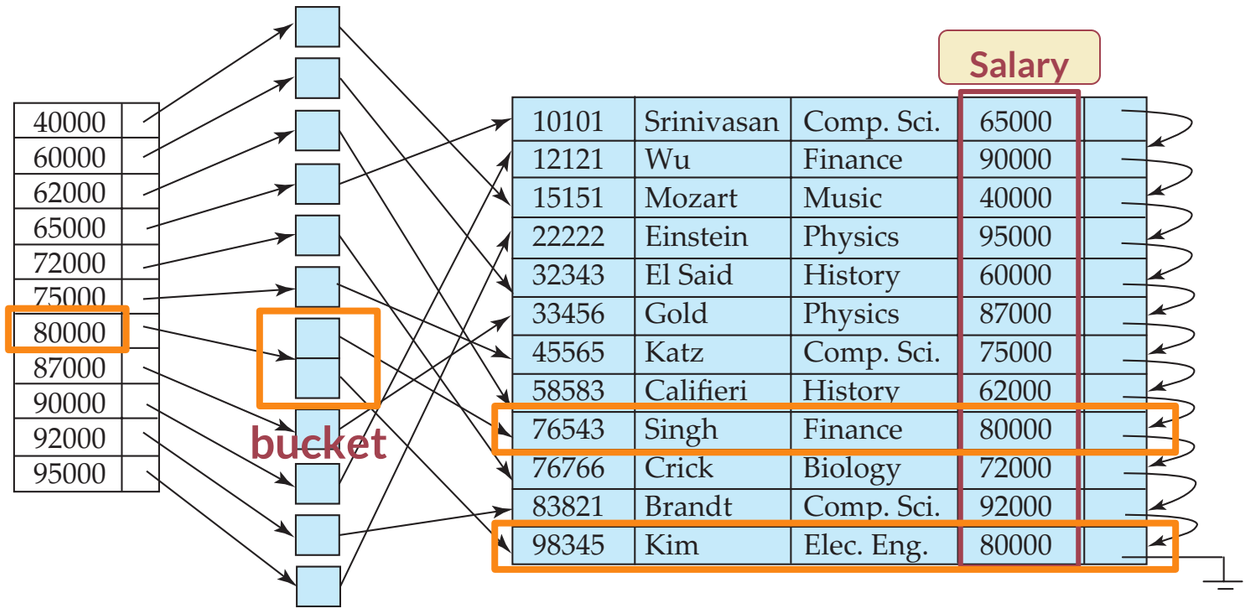
- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value
- Secondary indices have to be dense

Index Taxonomy

- 2) Does the index specify the order in which records are stored in the data file?
  - Yes ⇒ **Clustered index** (a.k.a. primary index)
  - No ⇒ **Non-clustered index** (a.k.a. secondary index)

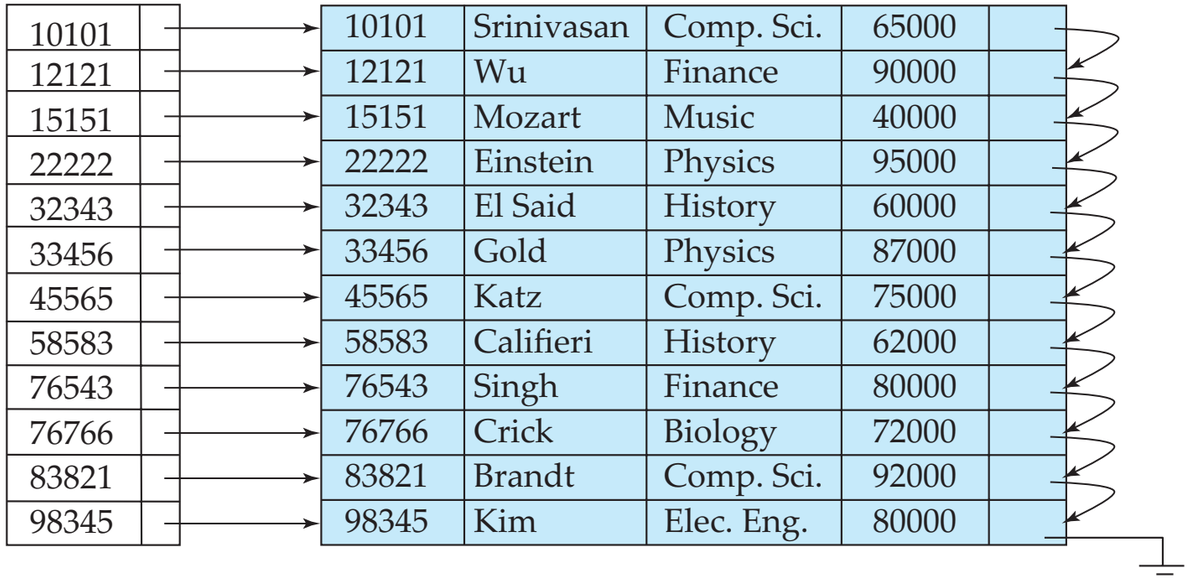
A secondary index on the column "salary"

- 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

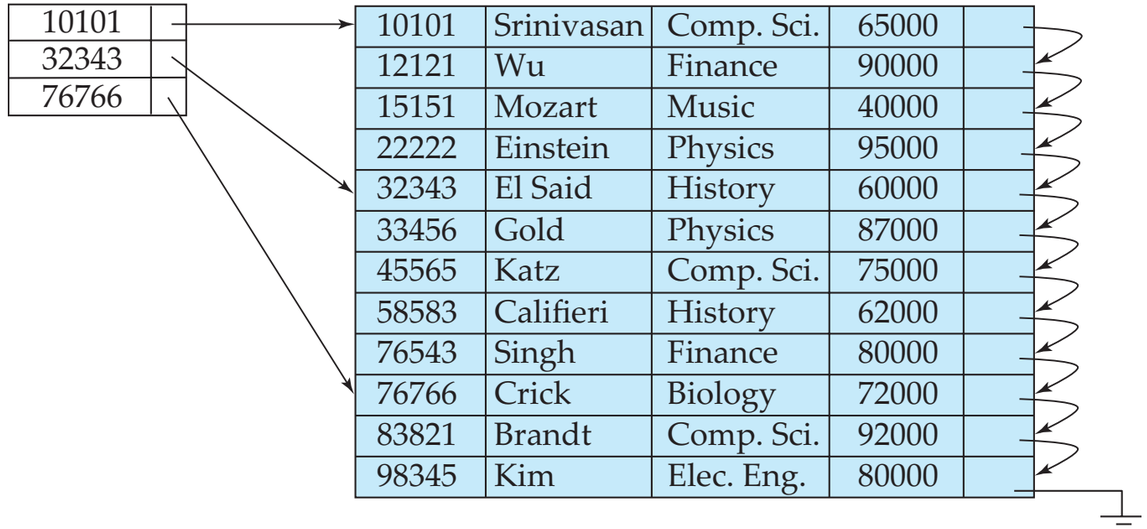


Index Taxonomy

- 3) Does every search key in the data file correspond to an index entry?
  - Yes ⇒ **Dense Index**
  - No ⇒ **Sparse Index**



Dense Index



Sparse Index

Index Taxonomy

- 4) Does the search key contain more than one attribute?
  - Yes ⇒ **Multi-key index** (Multi-column index)
  - No ⇒ **Single-key index** (Single-column index)
    - *We mainly focus on single-key index for now*

Index Implementation

- Data Structures for Indexes
  - B-tree, B+-tree
    - Very famous data structures for building indexes
  - Hash table

B-tree

- A B-tree of order ***m*** satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$
    - … except that a root node may have less than $\lceil m/2 \rceil$ children
  - (**Always balanced**) All leaves appear on the same level

B-tree

- A B-tree of order m satisfies that
  - For every node, # of children = # of keys + 1

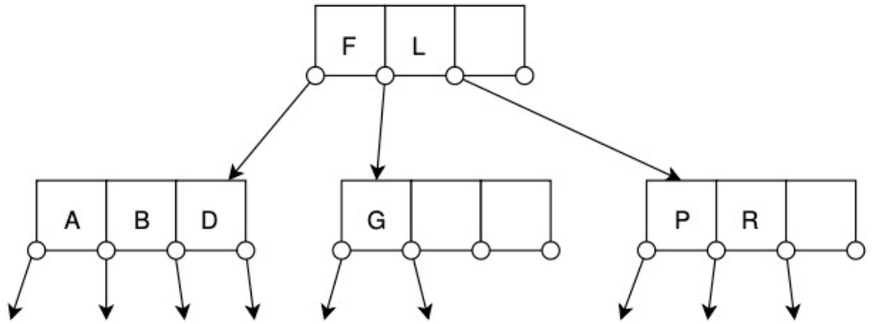
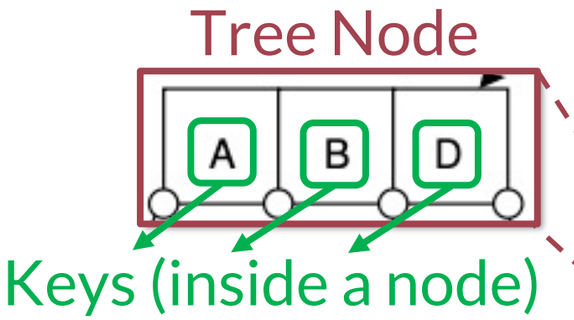


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$
    - … except that a root node may have less than $\lceil m/2 \rceil$ children
  - (**Always balanced**) All leaves appear on the same level

B-tree



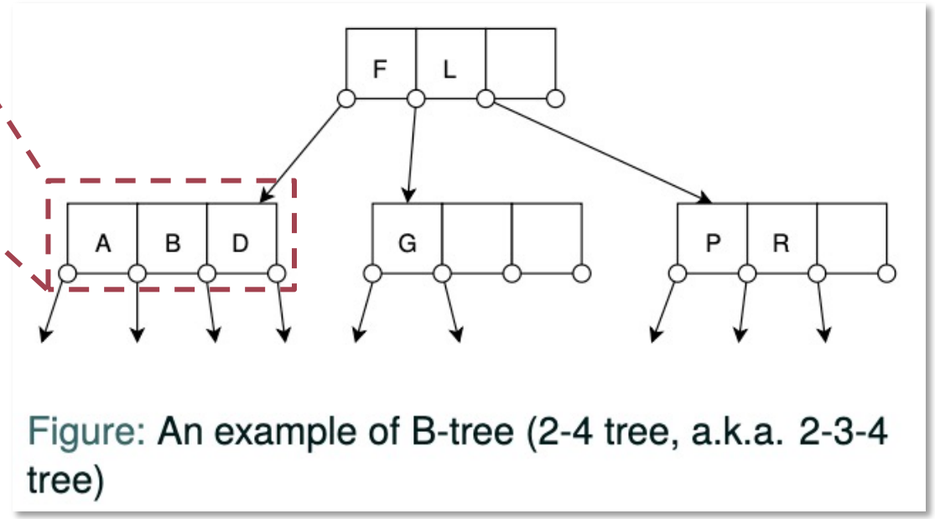


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

- A B-tree of order m satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$
    - … except that a root node may have less than $\lceil m/2 \rceil$ children
  - (**Always balanced**) All leaves appear on the same level

B-tree

- A B-tree of order m satisfies that
  - For every node, # of children = # of keys + 1
  - (**Ordered**) For a node containing n keys ($K_1 < K_2 < K_3 < \cdots < K_n$) with n+1 children (pointed by $P_0, P_1, P_2, \ldots, P_n$), any key $k_{sub\ i}$ in the sub-tree pointed by P_i satisfies that $K_i < k_{sub\ i} < K_{i+1}$
  - (**Multiway**) For an internal node, $\lceil m/2 \rceil \leq$ # of children $\leq m$
    - … except that a root node may have less than $\lceil m/2 \rceil$ children
  - (**Always balanced**) All leaves appear on the same level

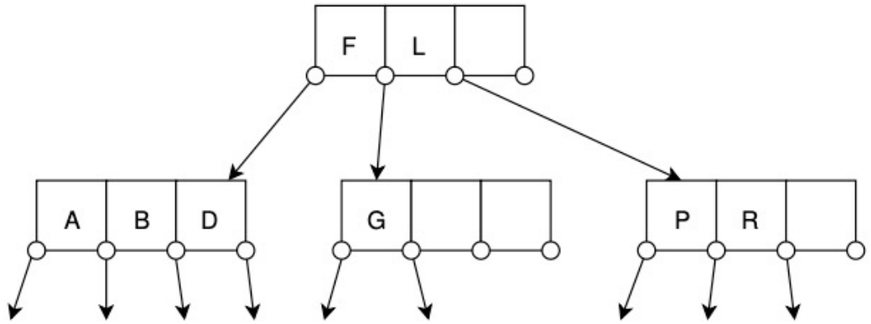


Figure: An example of B-tree (2-4 tree, a.k.a. 2-3-4 tree)

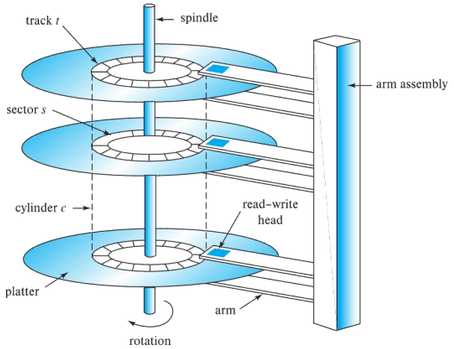
- $\lceil m/2 \rceil$ is the called the minimum branching factor (a.k.a. minimum degree) of the tree
- A B-tree of order m is usually called a "$\lceil m/2 \rceil$-m tree", like 2-3 tree, 2-4 tree, 3-5 tree, 3-6 tree, . . .
  - In practice, the order m is much larger (~100)

B-tree

- Height of a *B*-tree: $h \leq 1 + \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right)$
  - If we take an 50-100 tree with 1M records:
    - $h \leq 1+\log_{100/2}(1000000/2) = 4.354$ (i.e., 4 levels)

B-tree

- Height of a *B*-tree: $h \leq 1 + \log_{\lceil m/2 \rceil}\left(\frac{n+1}{2}\right)$
  - If we take an 50-100 tree with 1M records:
    - h ≤ 1+$\log_{100/2}$(1000000/2) = 4.354 (i.e., 4 levels)
  - Why do we use B-trees?
    - We can set the size of a B-tree node as the disk page size
      - i.e., m can be chosen with consideration on the page size
    - The height of the tree -> Number of disk I/Os
      - The number of disk I/Os can be relatively small



Access time: 5-20ms

1ns = 10^{-6}ms

Access time: 50-70ns

Seconds:

100000

Hours:

27.7777777778

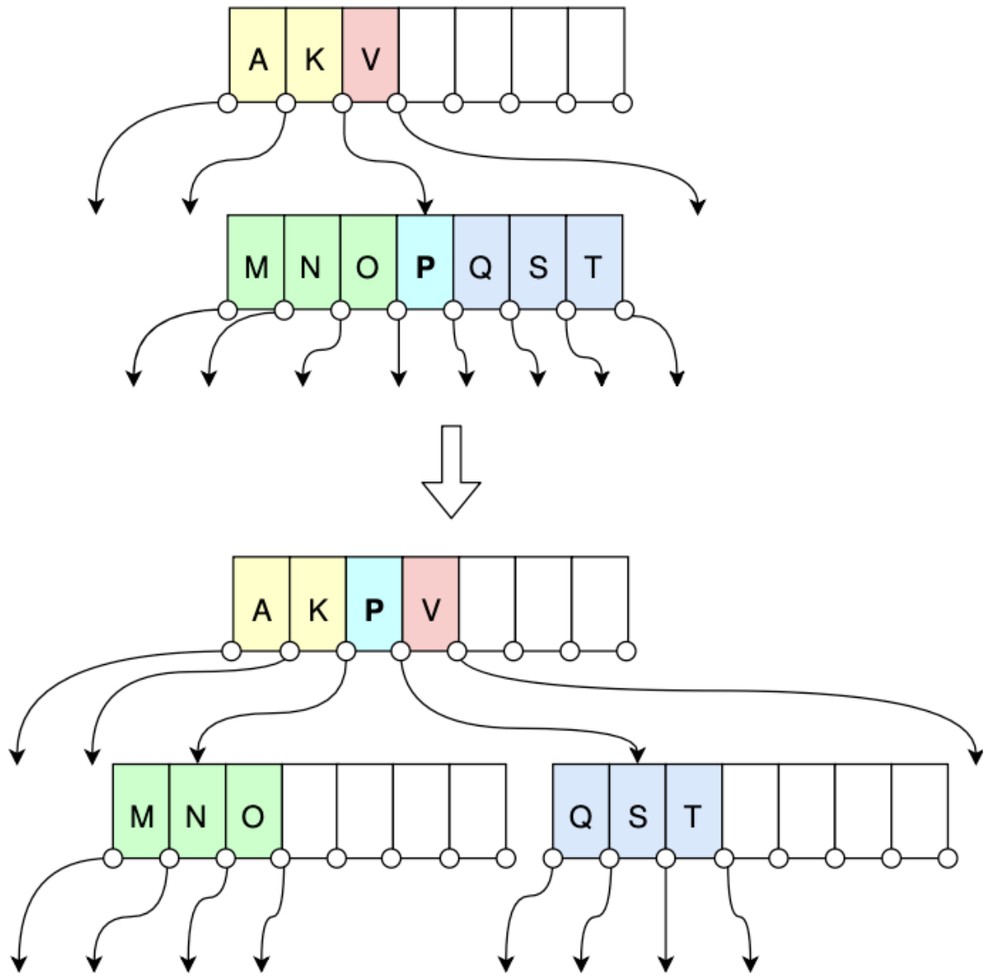
B-tree

- Tree operations:
  - Search, Insert, Delete
  - Update (Delete + Insert)

- What is special in B-tree
  - Split and merge nodes

B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

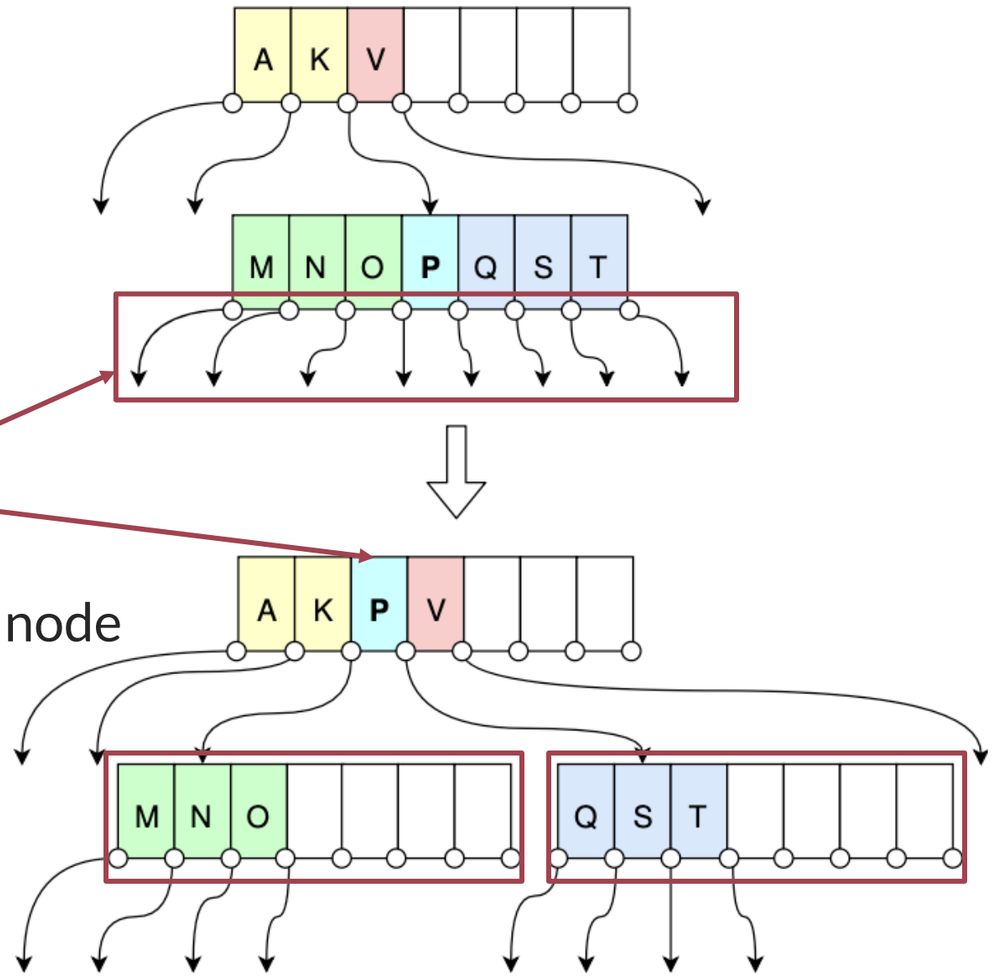


B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

The number of children is larger than m (7)

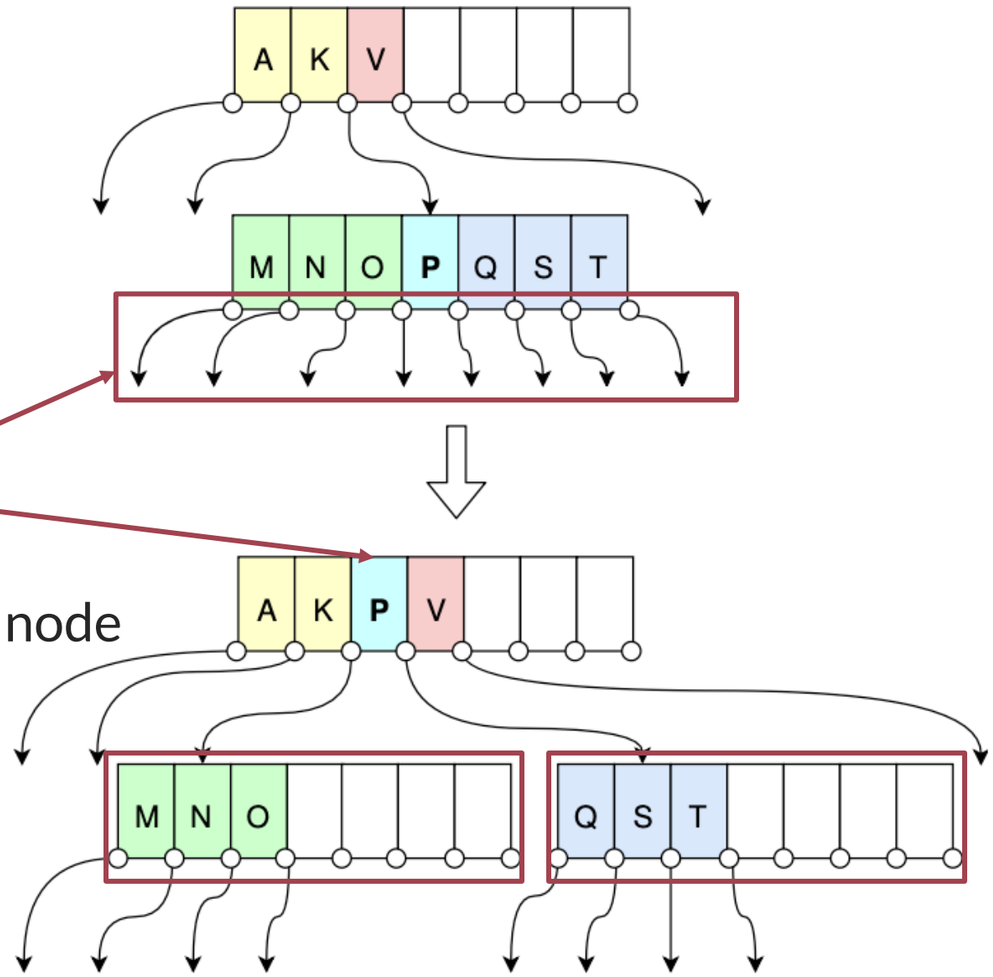
- This node will be split into two nodes
- The pivot key will be elevated into the parent node



B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

The number of children is larger than m (7)
- This node will be split into two nodes
- The pivot key will be elevated into the parent node

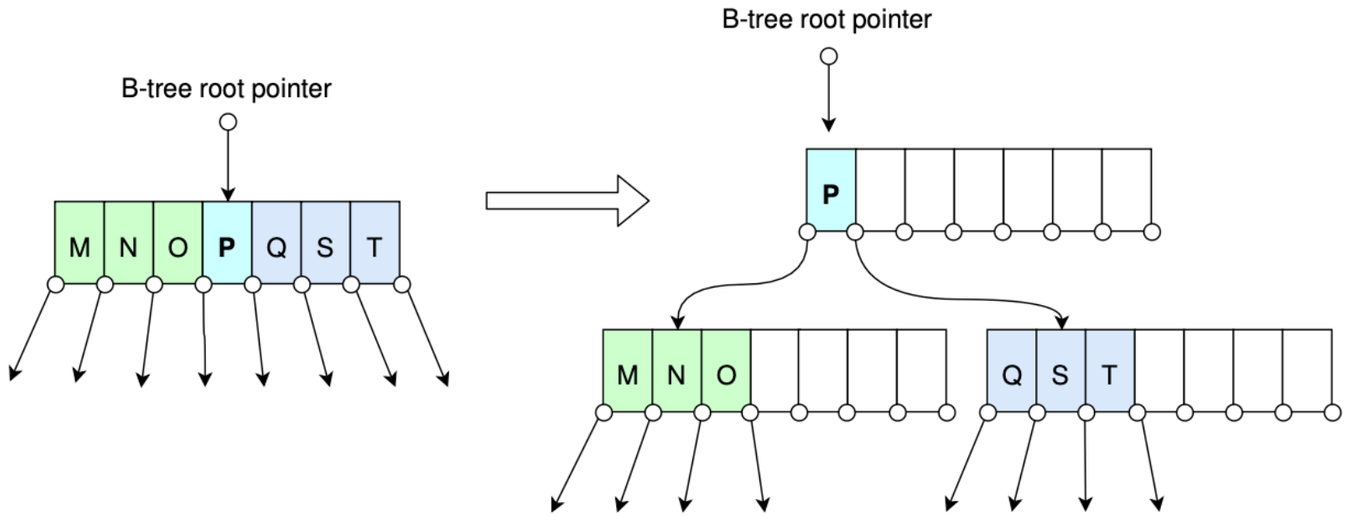


  - What if the parent (or even the root) node is also full?

B-tree

- Split a node in a B-tree
  - Example: when m=7
  - … and we want to insert the record with key="P"

- Split the root node of the B-tree

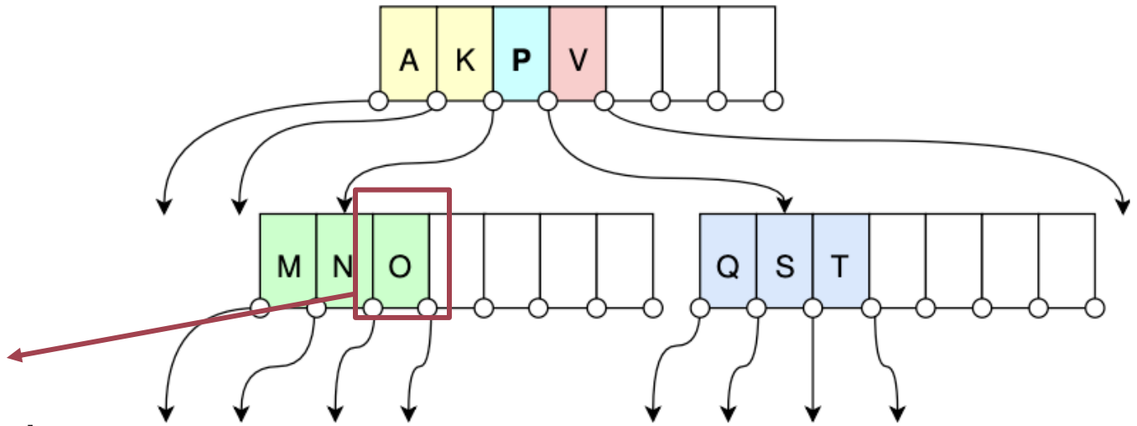


Note that the height of the B-tree is increased by 1

- This is the only way that a B-tree increases its height

B+-tree

- A Problem in B-tree: **Table traversal** when <u>only the B-tree is provided</u>
  - In B-tree, data are stored on all nodes
  - What if we want to traverse all records in the table?
    - `select * from letters`

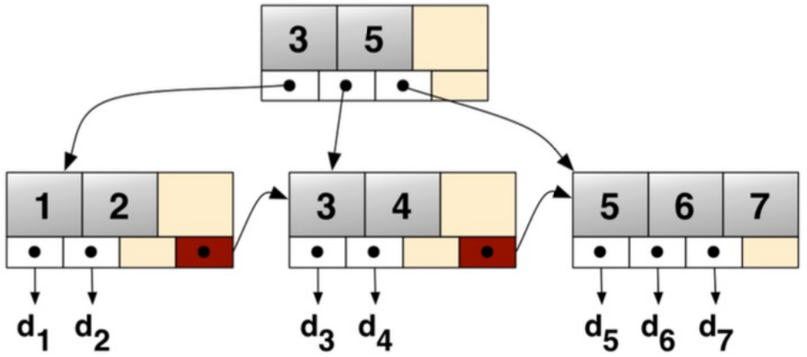


For example, we have accessed the node for letter “O”

- How can we find the next row?
  - We must go back to the parent node to access “P” (extra time cost)

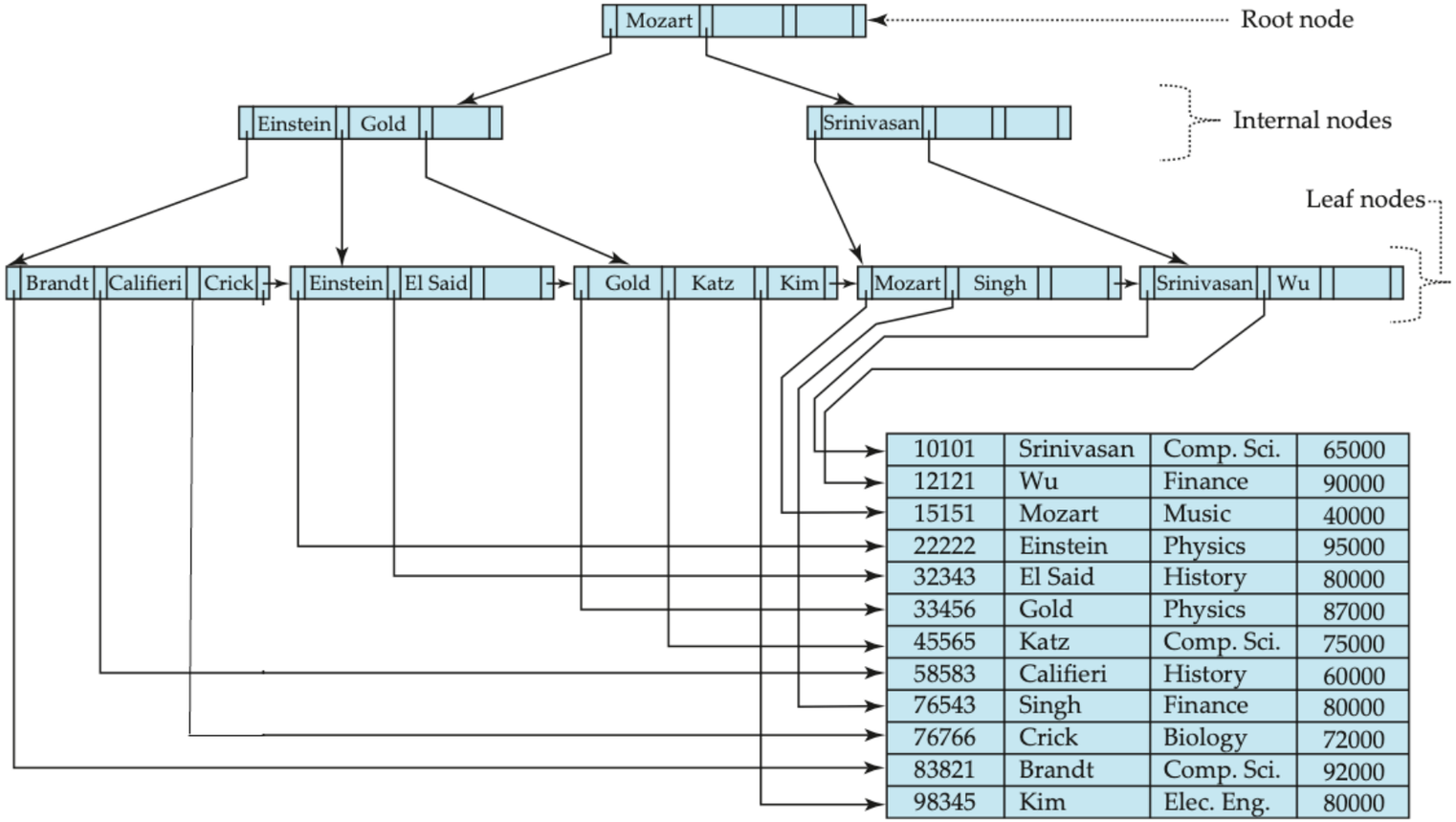
B+-tree

- Features of a B+-tree (compared with B-trees)
  - Data stored only in leaves
  - Leaves are linked sequentially



B+-tree

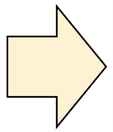
- A complete example of a B+-tree
  - Data stored only in leaves
    - No need to squeeze data into non-leaf nodes
  - Leaves are linked sequentially
    - Faster table traversal from top to bottom
    - Better support for range queries



Index It or Not: Where Indexing May Help

- Check whether the PK / Unique index helps first
- Index those columns frequently appeared as search criteria
  - `=`
  - `<, <=, >, >=, between`
  - `in`
  - `exists`
  - `like (prefix matching)`
- Be cautious when the indexed columns need frequent writing operations
  - Overhead to update indexes in `insert`, `update`, and `delete` opereations
- Functions

```sql
SELECT attr1, attr2
FROM table
WHERE function(column) = search_key
```



```sql
-- Create an index on the return values of the function
-- instead of the original values
create index idx_name ON table1(function(col1));
```

Note: The expression should be deterministic. For detailed usage, please refer to:
`https://www.postgresql.org/docs/14/indexes-expressional.html`

Index It or Not: Where Indexing May Help

- Be cautious when using indexes on a small table
  - Full scan ≠ Bad scheme
  - Index retrieval ≠ Good scheme

Hashing

- A **bucket** is a unit of storage containing one or more entries
  - A bucket is typically a disk block
  - We obtain the bucket of an entry 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 entries for access, insertion as well as deletion.

- Entries with different search-key values may be mapped to the same bucket
  - … thus, the entire bucket must be searched sequentially to locate an entry.

Hashing Index & Hashing File Organization

- In a hash index, buckets store entries with pointers to records

bucket1

```
15151: block 1, offset 0
```

bucket2

```
32343: block 3, offset 0
58583: block 3, offset 24
```

- In a hash file-organization, buckets store records

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

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

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

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

bucket 7

Example: How `join` Works (with the help of indexes)

- Some widely used `join` algorithms
  - Nested-loop join
  - Hash join
  - Sort-merge join

Example: How `join` Works (with the help of indexes)

- Nested (loop) join
  - Straight-forward linking between records from two tables in a nested-loop manner

```
for each row in t1 match C1(t1)
    for each row in t2 match P(t1, t2)
        if C2(t2)
            add t1|t2 to the result
```

Example: How `join` Works (with the help of indexes)

- Hash join
  - Build a set of buckets for a smaller table to speed up the data lookup

- Procedure:
  - 1. Create a hash table for the smaller table `t1` in the memory
  - 2. Scan the larger table `t2`. For each record `r`,
    - 2.1 Compute the hash value of `r.join_attribute`
    - 2.2 Map to corresponding rows in `t1` using the hash table

Example: How `join` Works (with the help of indexes)

- Sort-merge join (a.k.a. merge join)
  - Zipper-like joining

- Procedure:
  - 1. Sort tables t1 and t2 respectively according to the join attributes
  - 2. Perform an interleaved scan of t1 and t2. When encountering a matched value, join the related rows together.

When there are clustered indexes on the join attributes, step 1, the most expensive operation, can be skipped because t1 and t2 are already sorted in this scenario.

Transaction

Transaction in Real Life

- “An exchange of goods for money”
  - A series of steps
  - All or nothing

Flickr:BracketingLife (Clarence)

Transaction in Computer

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥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)
```

An Example of Transactions in PostgreSQL

- BEGIN, COMMIT, ROLLBACK

```
begin;  -- Start a transaction

update people_1 set num_movies = 50000 where peopleid = 1;

select * from people_1 where peopleid = 1;

delete from people_1 where peopleid > 100 and peopleid < 200;

commit;  -- start executing all the queries above
-- or "rollback;", which means to revoke the operationso of all the queries
```

Transaction in Computer

- A transaction is a unit of program execution that accesses and possibly updates various data items
  - A classical example in database: money transfer

E.g., transaction to transfer CNY ¥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

Requirements in Transactions

- Atomicity Requirement
  - 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

E.g., transaction to transfer CNY ¥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)
```

Requirements in Transactions

- 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.

Requirements in Transactions

- Consistency Requirement
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts
      - In the example: The sum of A and B is unchanged by the execution of the transaction

E.g., transaction to transfer CNY ¥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)
```

Requirements in Transactions

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

ACID Properties

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

Atomicity: Either all operations of the transaction are properly reflected in the database, or none are.

Consistency: Execution of a transaction in isolation preserves the consistency of the database.

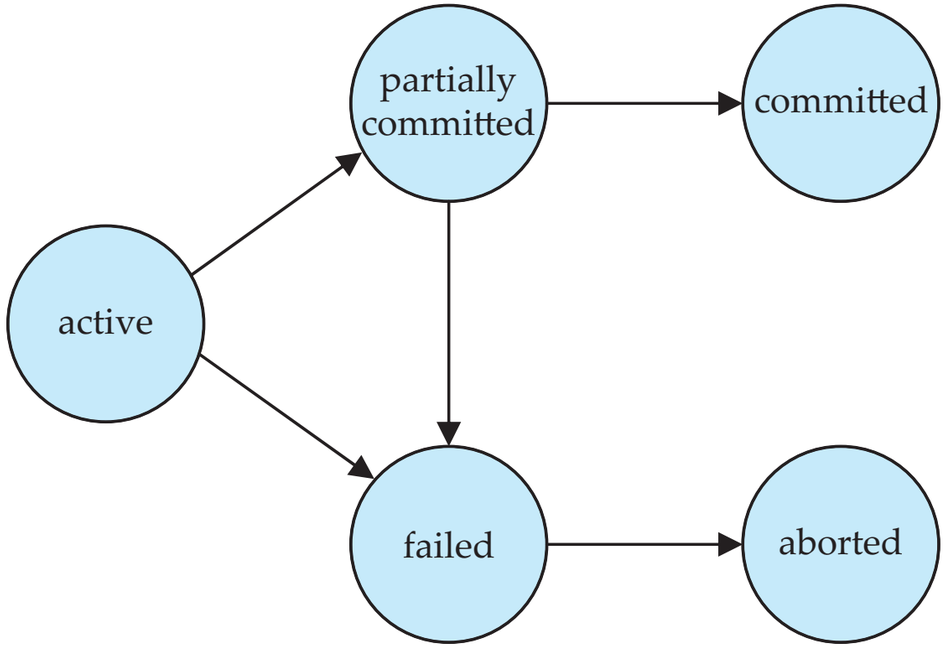
Isolation: Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.

- That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.

Durability: After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

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



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 do not need to 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

Schedules

- **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 instructions 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 transfer CNY ¥50 from A to B, and T_2 transfer 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)	
commit	
	read (A)
	$temp := A * 0.1$
	$A := A - temp$
	write (A)
	read (B)
	$B := B + temp$
	write (B)
	commit

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

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

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)	
commit	
	read (B)
	$B := B + temp$
	write (B)
	commit

Schedule 4

- The following concurrent schedule does not preserve the value of (A + B)

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)	
commit	
	$B := B + temp$
	write (B)
	commit

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 I_i and I_j , of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  - 1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict
  - 2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  - 3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  - 4. I_i = **write**(Q), I_j = **write**(Q). They conflict

- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them.
  - If I_i and I_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

- 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*.

Operations on different data
- … and hence swappable in temporal order

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

- 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 >.

* 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*,
  - 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*.
  - 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 .
  - 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

- A schedule *S* is **view serializable** if it is view-equivalent to a serial schedule
- Every conflict serializable schedule is also view serializable

- Below is a schedule which is view-serializable but *not* conflict serializable

Two "blind writes" in T27 and T28
- Since the written values were not used anywhere else

T_{27}	T_{28}	T_{29}
read (Q)		
write (Q)	write (Q)	
		write (Q)

Overwrites values from T27 and T28
- … and hence, swapping write(Q) in T27 and T28 will not affect the resulting value of Q

- What serial schedule is above equivalent to?
- Every view-serializable schedule that is not conflict serializable has **blind writes**
  - **Blind write:** Write operations without further reading

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, ..., T_n

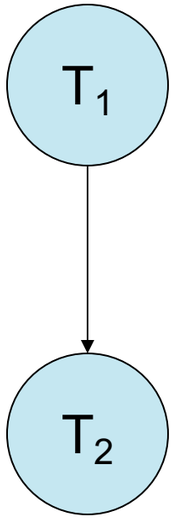
- Precedence graph
  - A directed graph where the vertices are the transactions (names of the transactions)
  - We draw an arc from T_i to T_j if the two transactions **conflict**
    - which means, in the schedule S, T_i **must** appear earlier than T_j
  - We may label the arc by the item that was accessed.

Conflict – At least one of the following situations exists for a data item Q:

- **T_i: write(Q) -> T_j: read(Q)**
- **T_i: read(Q) -> T_j: write(Q)**
- **T_i: write(Q) -> T_j: write(Q)**

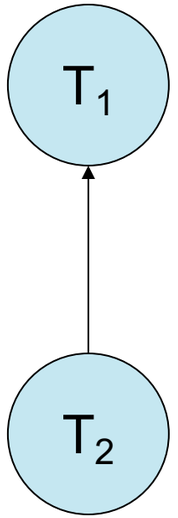
Testing for Serializability

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



Schedule 1

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

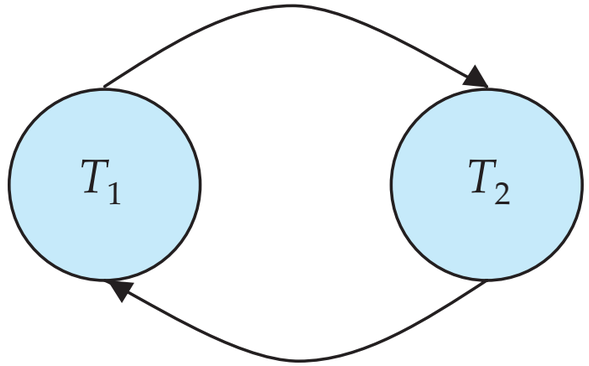


Schedule 2

Testing for Serializability

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)	
commit	
	$B := B + temp$
	write (B)
	commit

Schedule 4

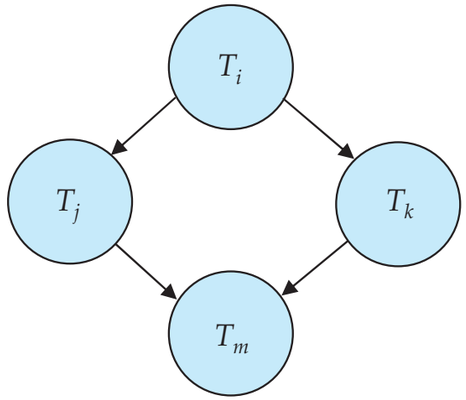


Testing for Serializability

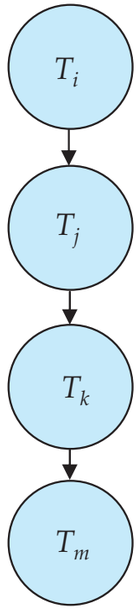
- A schedule is conflict serializable if and only if its precedence graph is acyclic

Cycle-detection: Cycle-detection algorithms exist which take n^2 time, where n is the number of vertices in the graph.

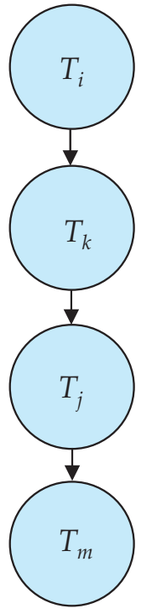
- If the precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph
  - E.g., The topological order of (a) can be (b) and (c)



(a)



(b)



(c)

Recoverable Schedules

- Need to address the effect of transaction failures on concurrently running transactions
- **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.
- The following schedule is not recoverable

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

  - If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
    - Such transactions do not need to be serializable with respect to other transactions
  - Purpose: Trade-off between accuracy and performance

Levels of Consistency (in SQL-92)

- Serializable (Strongest)
  - Default
- Repeatable read — only committed records to be read.
  - Repeated reads of same record must return same value.
  - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- Read committed — only committed records can be read.
  - Successive reads of record may return different (but committed) values.
- Read uncommitted (Weakest) — even uncommitted records may be read.

Levels of Consistency

- Lower degrees of consistency can be useful for gathering approximate information about the database

- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

- Warning 2: All SQL-92 consistency levels infer that dirty writes are prohibited
  - Dirty write - when one transaction overwrites a value that has previously been written by another still in-flight transaction