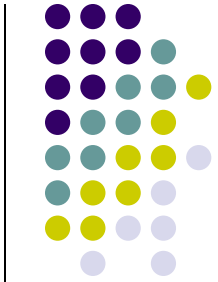
CMSC424: Database Design

Module: Database Implementation

Instructor: Amol Deshpande
amol@cs.umd.edu

Database Implementation



- Shifting into discussing the internals of a DBMS
  - How data stored? How queries/transactions executed?

- Some of the Topics:
  - Storage: How is data stored? Important features of the storage devices (RAM, Disks, SSDs, etc)
  - Tuple Organization: How are tuples laid out
  - Indexes: How to quickly find specific tuples of interest (e.g., all ‘friends’ of ‘user0’)
  - Query processing: How to execute different relational operations? How to combine them to execute an SQL query? How to do this in a parallel setting?
  - Query optimization: How to choose the best way to execute a query?
  - Transactions: How to support the ACID properties? In a distributed setting?

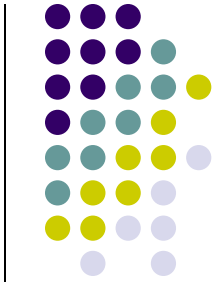
CMSC424: Database Design

Module: Database Implementation

Computing Hardware

Instructor: Amol Deshpande
amol@cs.umd.edu

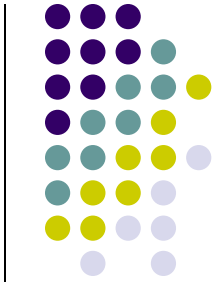
Computing Hardware



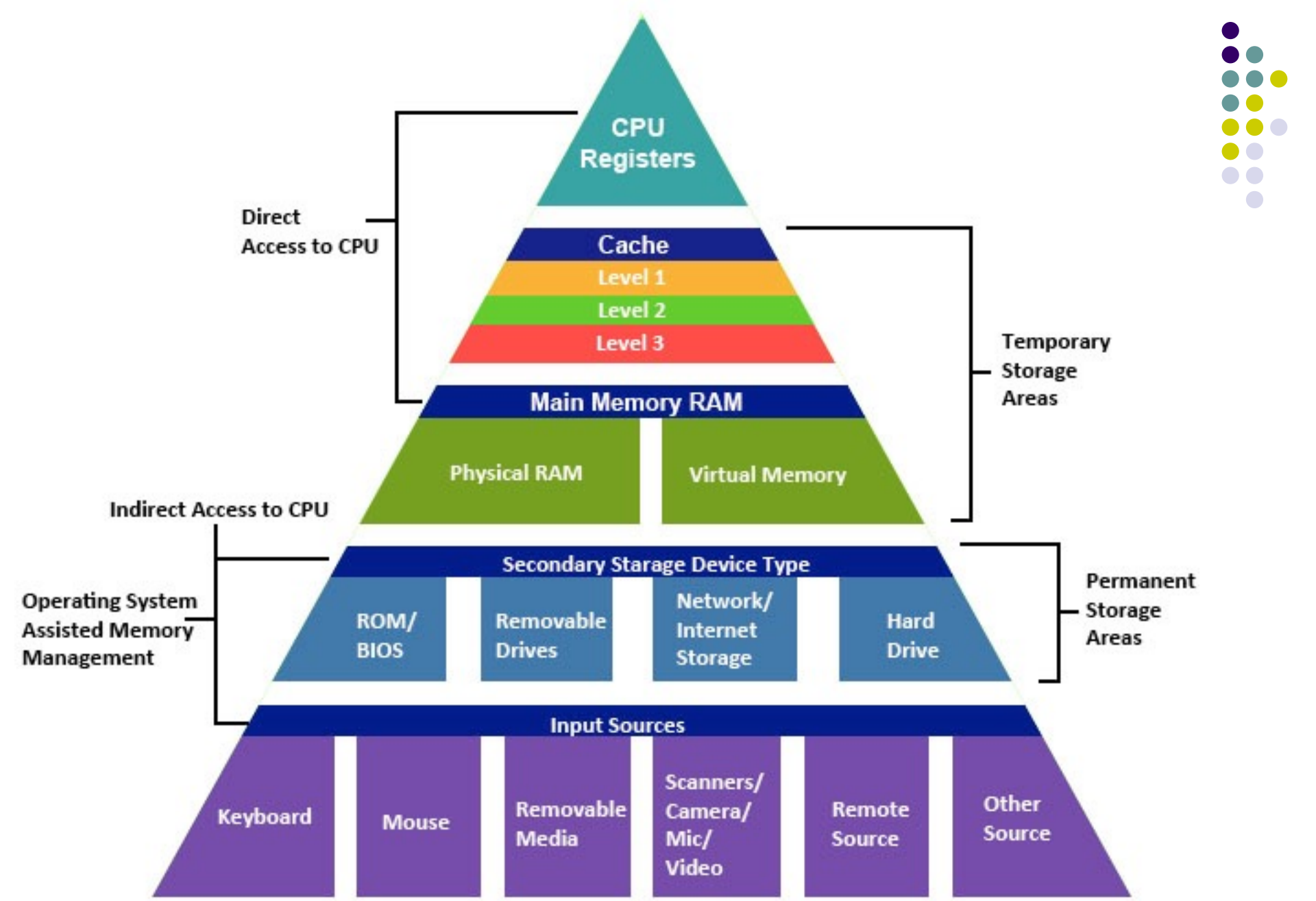
- Book Chapters
  - 10, 20 (and some other online resources)

- Key topics:
  - Differences between storage media
  - Storage hierarchy, Caches, SSDs
  - Parallel vs Distributed

Storage Options

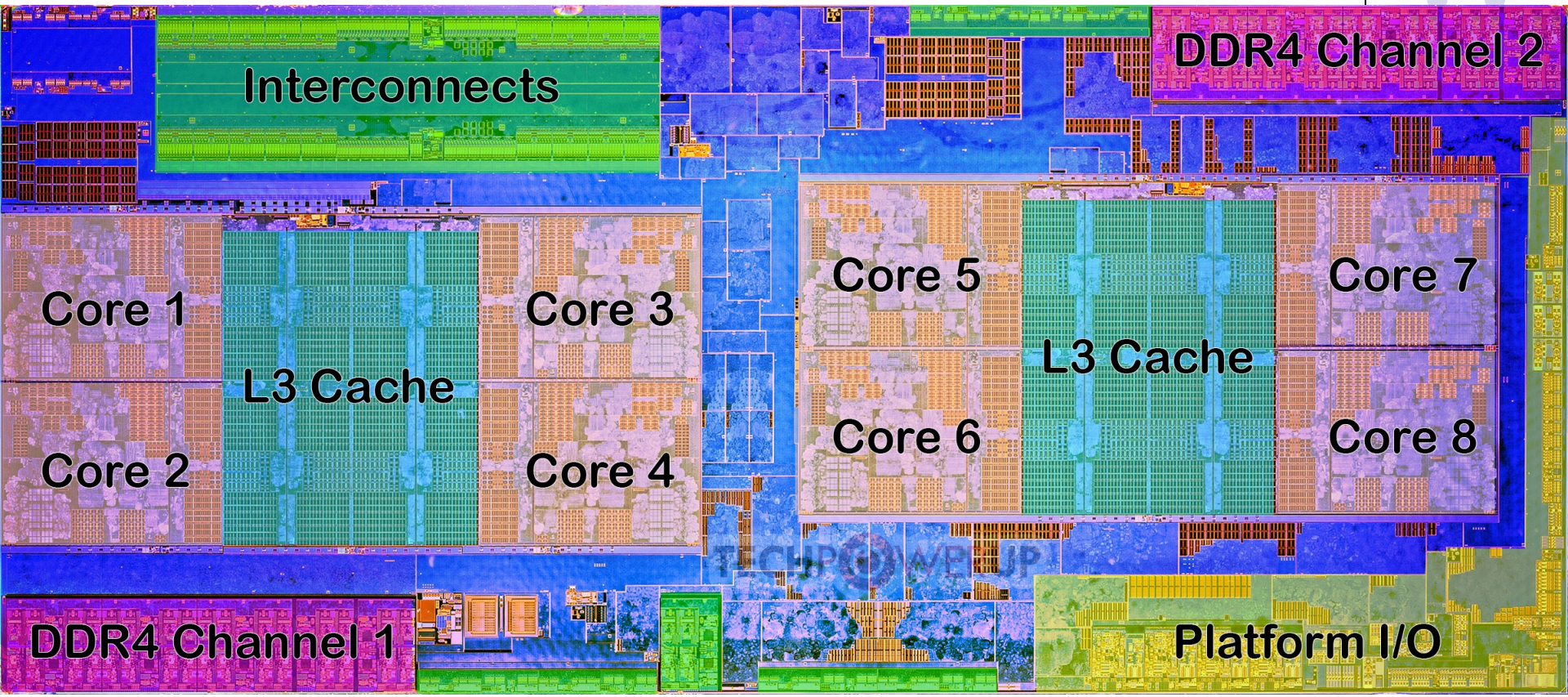


- At various points, data stored in different storage hardware
  - Memory, Disks, SSDs, Tapes, Cache
  - Tradeoffs between speed and cost of access
  - CPU needs the data in memory and cache to operate on it
- Volatile vs nonvolatile
  - Volatile: Loses contents when power switched off
- Sequential vs random access
  - Sequential: read the data contiguously
    - select * from employee
  - Random: read the data from anywhere at any time
    - select * from employee where name like '__a__b'



source: http://cse1.net/recaps/4-memory.html

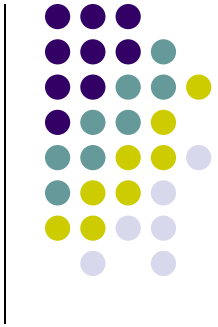
AMD Ryzen CPU Architecture



Die shot overlaid with functional units

https://www.techpowerup.com/review/amd-ryzen-5-1600/3.html

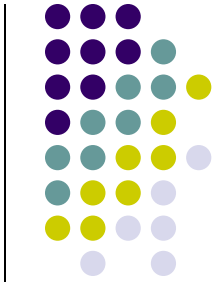
Storage Hierarchy



Storage type	Access time	Relative access time
L1 cache	0.5 ns	Blink of an eye
L2 cache	7 ns	4 seconds
1MB from RAM	0.25 ms	5 days
1MB from SSD	1 ms	23 days
HDD seek	10 ms	231 days
1MB from HDD	20 ms	1.25 years

source: http://cse1.net/recaps/4-memory.html

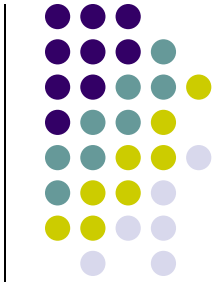
Storage Options



- Primary
  - e.g. Main memory, cache; typically volatile, fast
- Secondary
  - e.g. Disks; Solid State Drives (SSD); non-volatile
- Tertiary
  - e.g. Tapes; Non-volatile, super cheap, slow

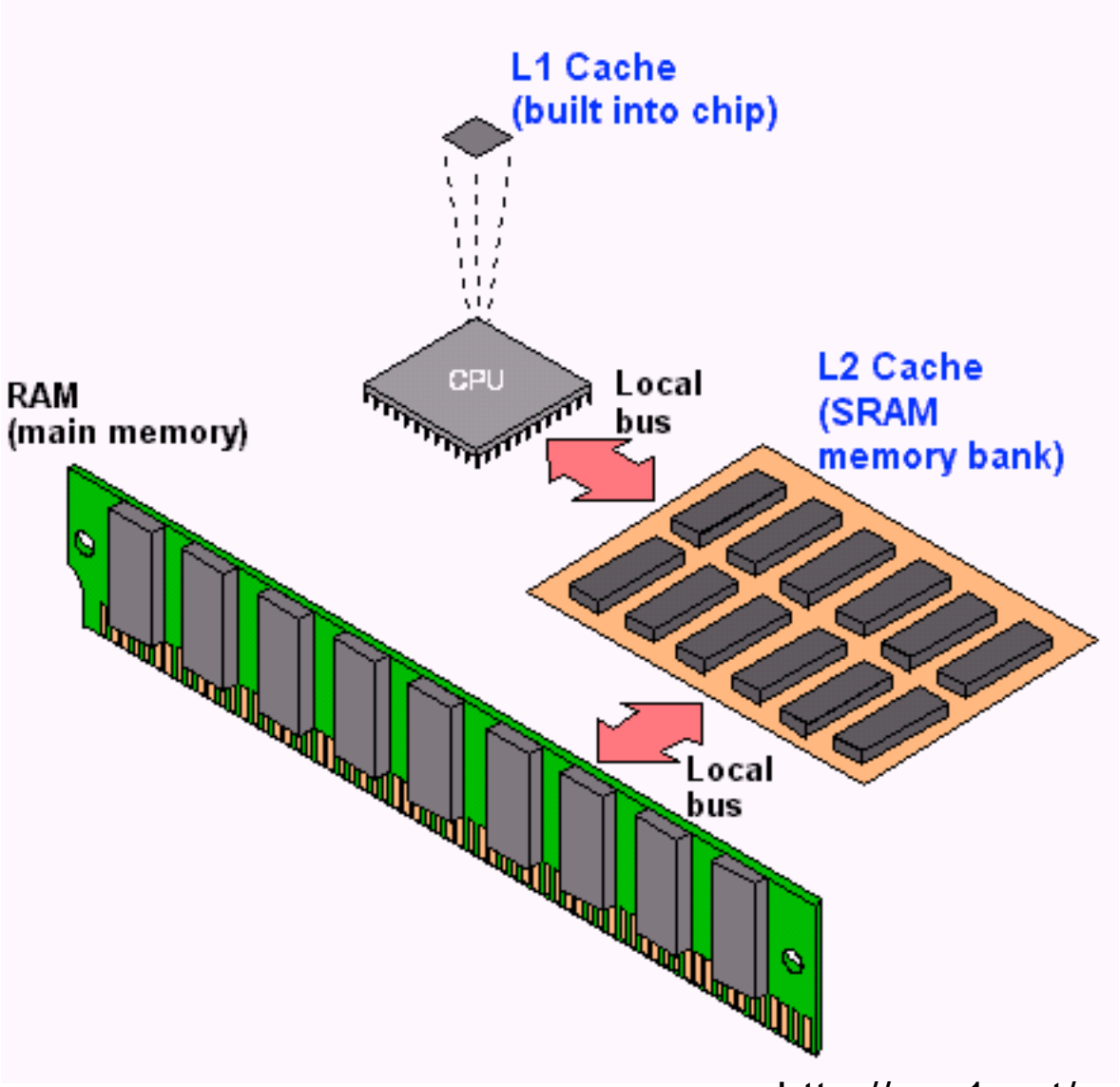
- Each storage media has different performance characteristics
  - Important to understand in order to write systems or optimize queries or tasks

Storage Hierarchy: Cache



- Cache
  - Super fast; volatile; Typically on chip
  - L1 vs L2 vs L3 caches ???
    - L1 about 64KB or so; L2 about 1MB; L3 8MB (on chip) to 256MB (off chip)
    - Huge L3 caches available now-a-days
  - Becoming more and more important to care about this
    - Cache misses are expensive
  - Similar tradeoffs as were seen between main memory and disks

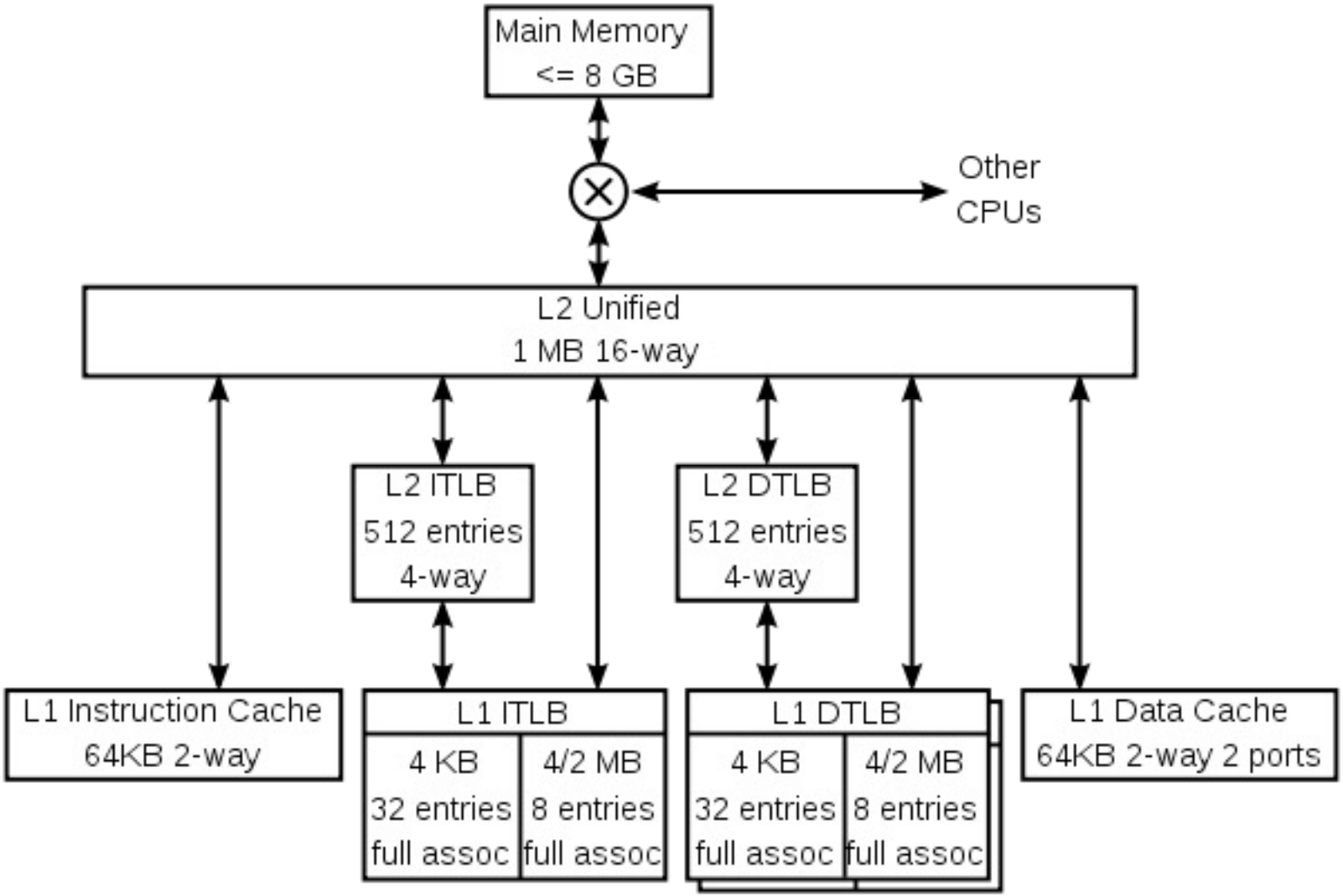
Storage Hierarchy: Cache



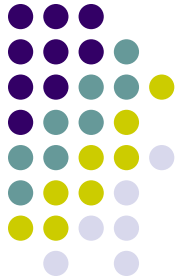
source: http://cse1.net/recaps/4-memory.html

Storage Hierarchy: Cache

K8 core in the AMD Athlon 64 CPU



Storage Hierarchy: Main Memory



- Data must be brought from disks/SSDs into Memory (and then into Caches) for the CPU to access it
  - CPU has no “direct” connection to the disks/SSDs

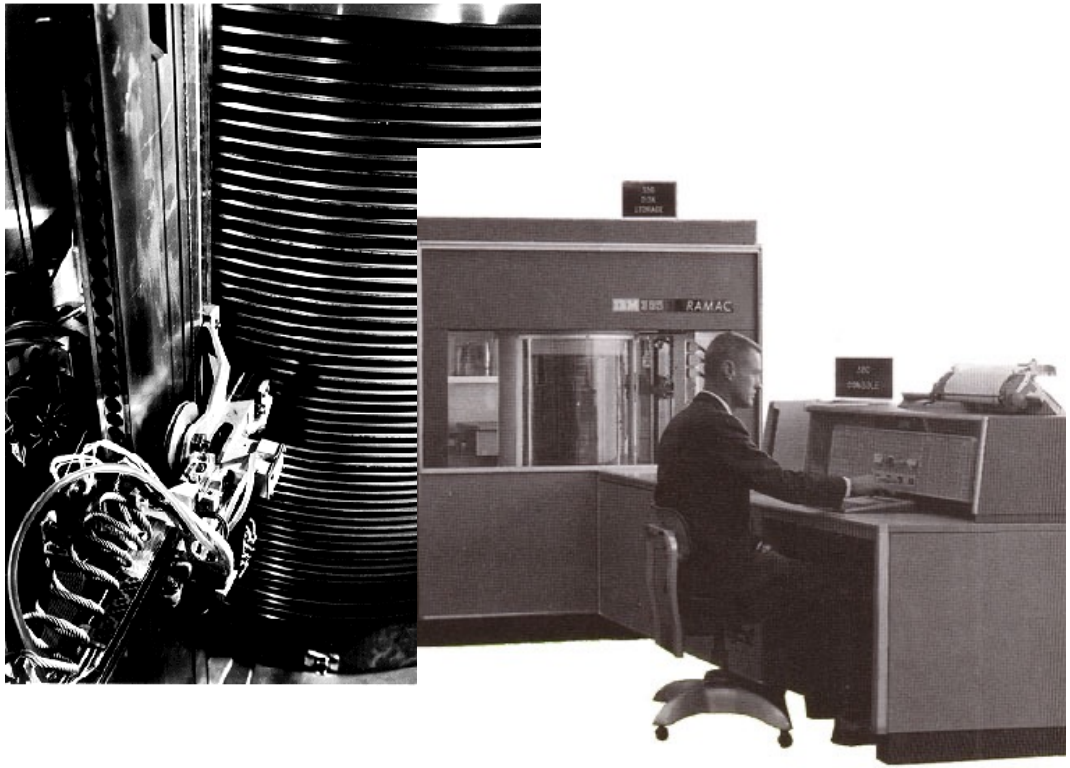
- 10s or 100s of ns; Volatile (so will not survive a power failure)
- Pretty cheap and dropping: 1GByte < $2-3 today

- Main memory databases very common now-a-days
  - Dramatically changes the tradeoffs
  - Don’t need to worry about the disks or SSDs as much

Storage Hierarchy: Magnetic/Hard Disks

- Primary ”non-volatile” storage mechanism

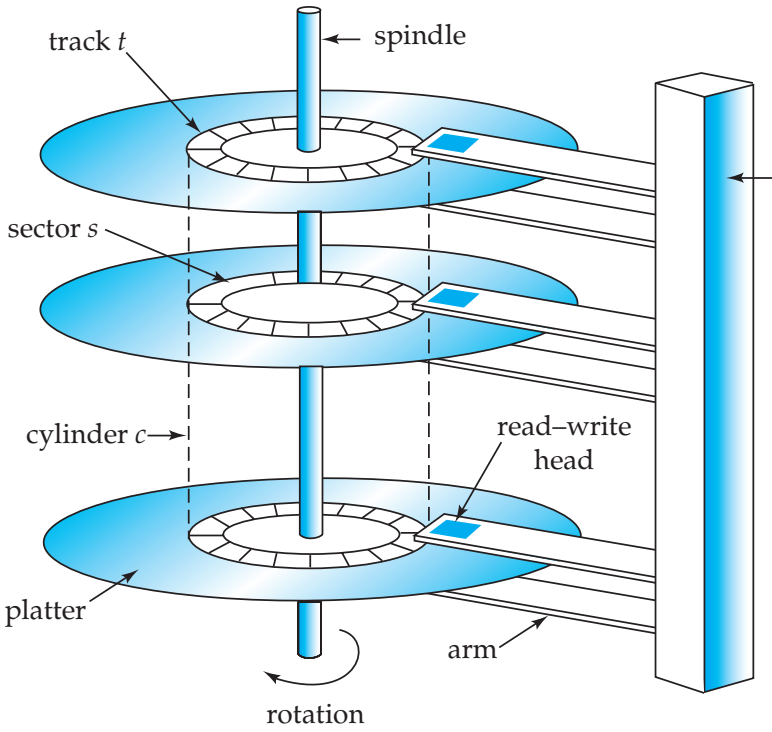
1998
SEAGATE
47GB



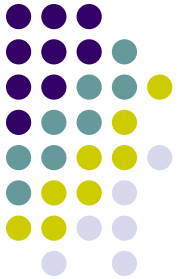
1956
IBM RAMAC
24” platters
100,000 characters each
5 million characters

Storage Hierarchy: Magnetic/Hard Disks

- Primary ”non-volatile” storage mechanism
- Accessing a sector
  - Time to *seek* to the track (seek time)
    - average 4 to 10ms
  - + Waiting for the sector to get under the head (rotational latency)
    - average 4 to 11ms
  - + Time to transfer the data (transfer time)
    - very low
  - About 10ms per access
    - So if randomly accessed blocks, can only do 100 block transfers
    - 100 x 512bytes = 50 KB/s
- Data transfer rates
  - Rate at which data can be transferred (w/o any seeks)
  - 30-50MB/s to up to 200MB/s (Compare to above)
    - Seeks are bad !



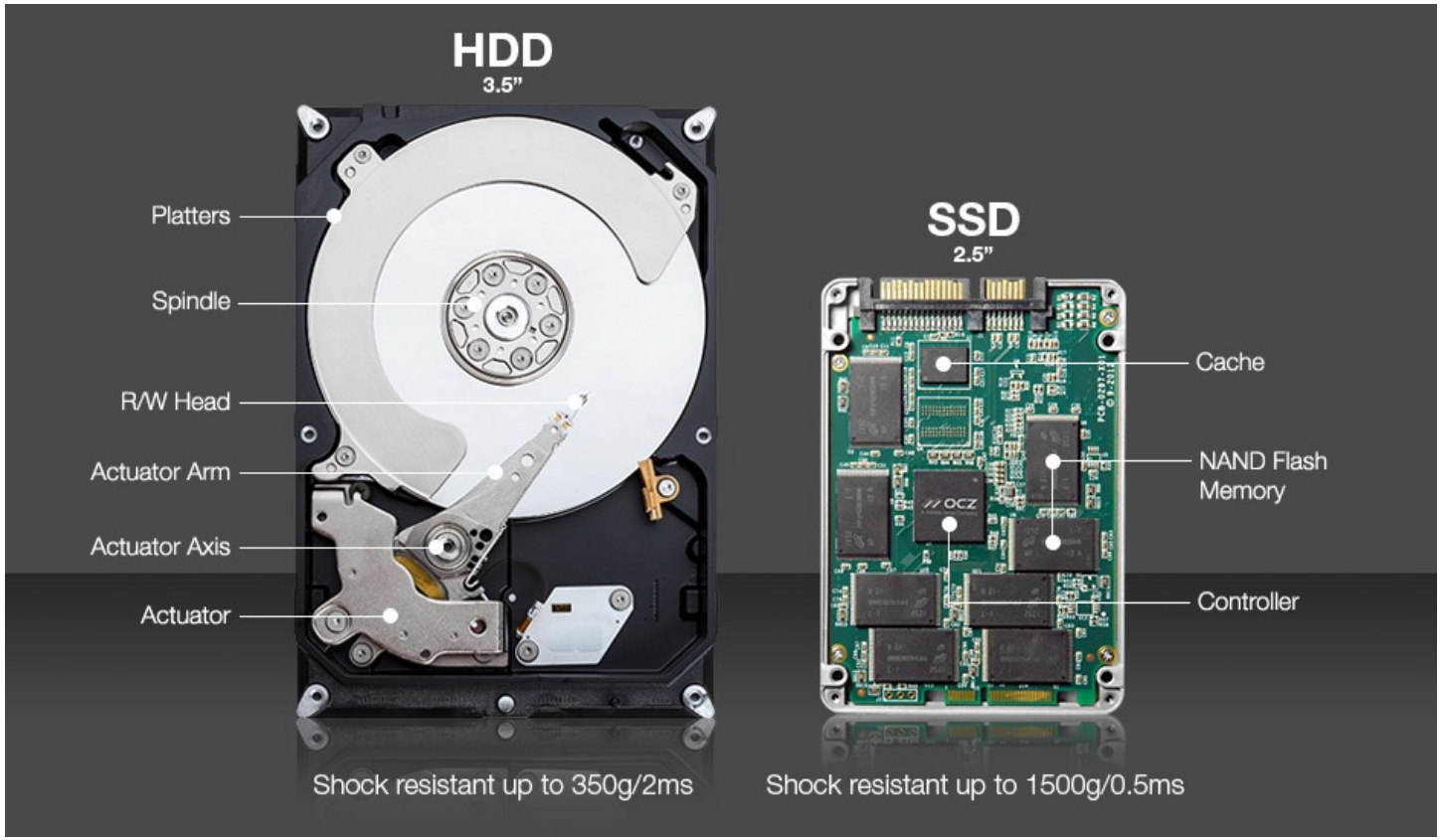
Hard Disks: Reliability



- Mean time to/between failure (MTTF/MTBF):
  - 57 to 136 years
- Consider:
  - 1000 new disks
  - 1,200,000 hours of MTTF each
  - On average, one will fail 1200 hours = 50 days !
- Need to assume disk failures are common
  - Handled today through keeping data in duplicate, or triplicate
  - If a disk fails, replace with a new disk and copy data over

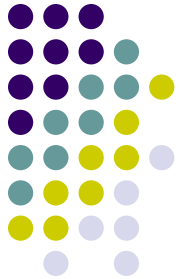
Solid State Drives

- Essentially flash that emulates hard disk interfaces



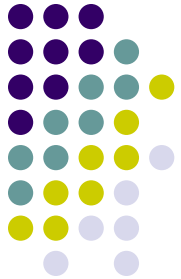
https://blogs.umass.edu/Techbytes/2018/02/23/types-of-ssds-and-which-ones-to-buy/

Solid State Drives



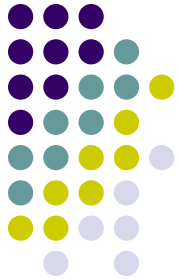
- Still support the same “block-oriented” interface
  - So reads/writes happen in units of blocks
- No seeks → Much better random reads performance
- Writes are more complicated
  - Must write an entire block at a time, after first “erasing” it
  - Limit on how many times you can erase a block
- Wear leveling
  - Distributes writes across the SSD for uniform wearing out
- Flash Translation Layer (FTL) takes care of these issues
- About a factor of 5-10 more expensive right now

Storage Hierarchy



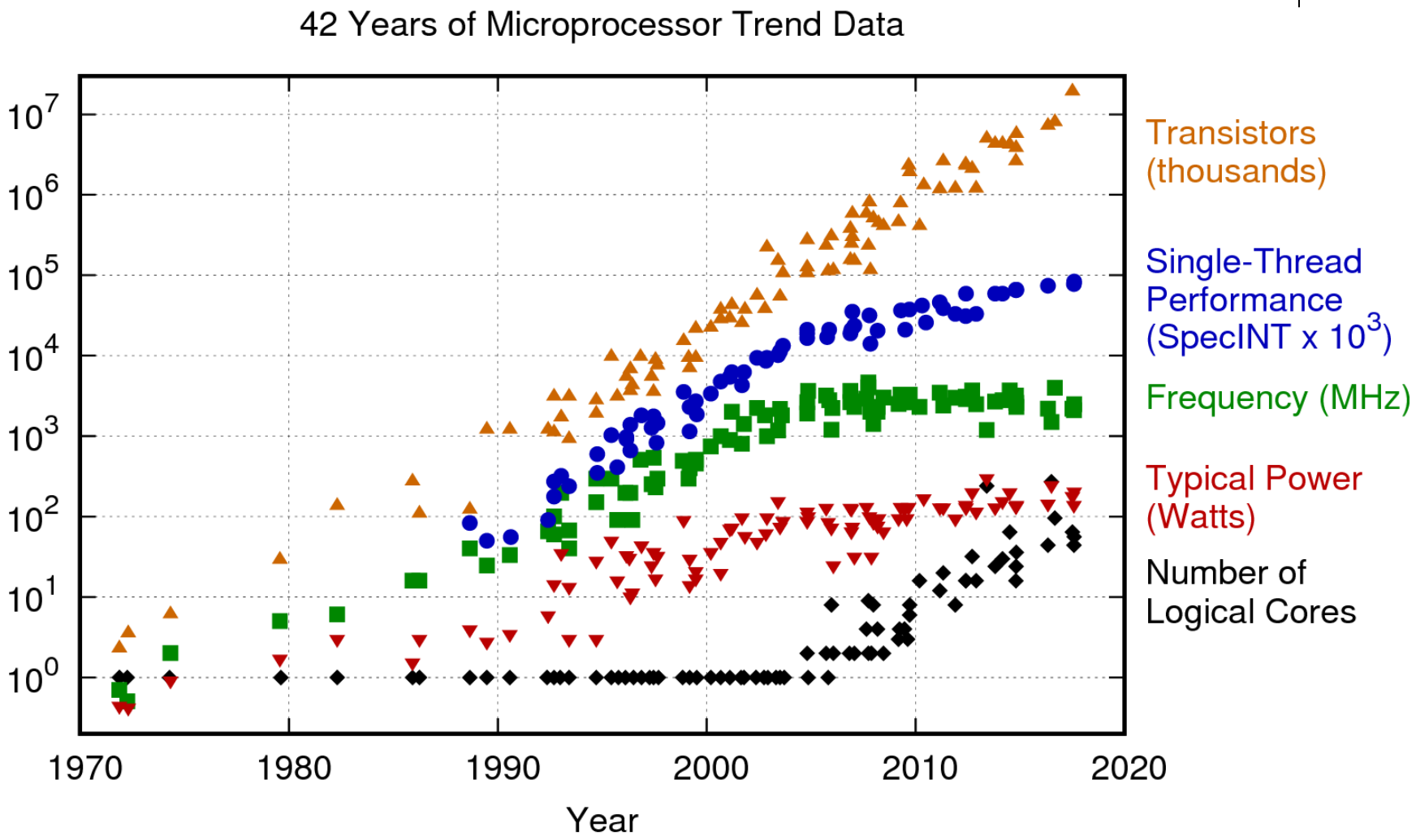
- Optical Storage - CDs/DVDs; Jukeboxes
  - Used more as backups… Why ?
  - Very slow to write (if possible at all)
- Tape storage
  - Backups; super-cheap; painful to access
  - IBM just released a secure tape drive storage solution

Data Storage Options



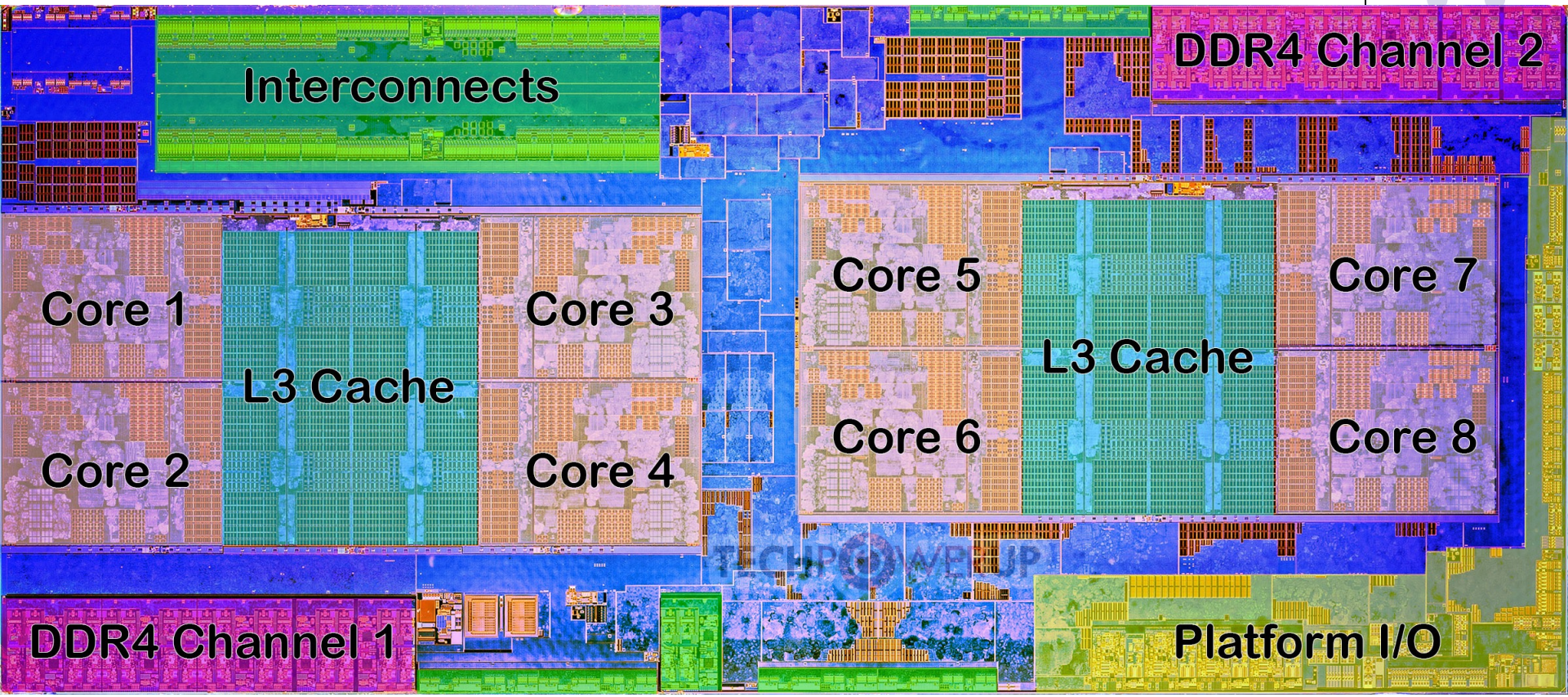
- Hard disks dominant form of storage for a long time
  - About 10ms per random access ➔ at most 100 random reads per second
  - vs up to 500 MB/s sequential I/O
- Many traditional database design decisions driven by:
  - Huge volumes of data on disks + Low amounts of memory + Low-speed networks
  - ➔ Communication between disks and memory the main bottleneck
- Solid state drives much more common today
  - No seeks → Much better random reads
  - Writes require erasing an entire block, and rewriting it
  - SSDs provide a similar interface of “blocks”

Microprocessors Evolution



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

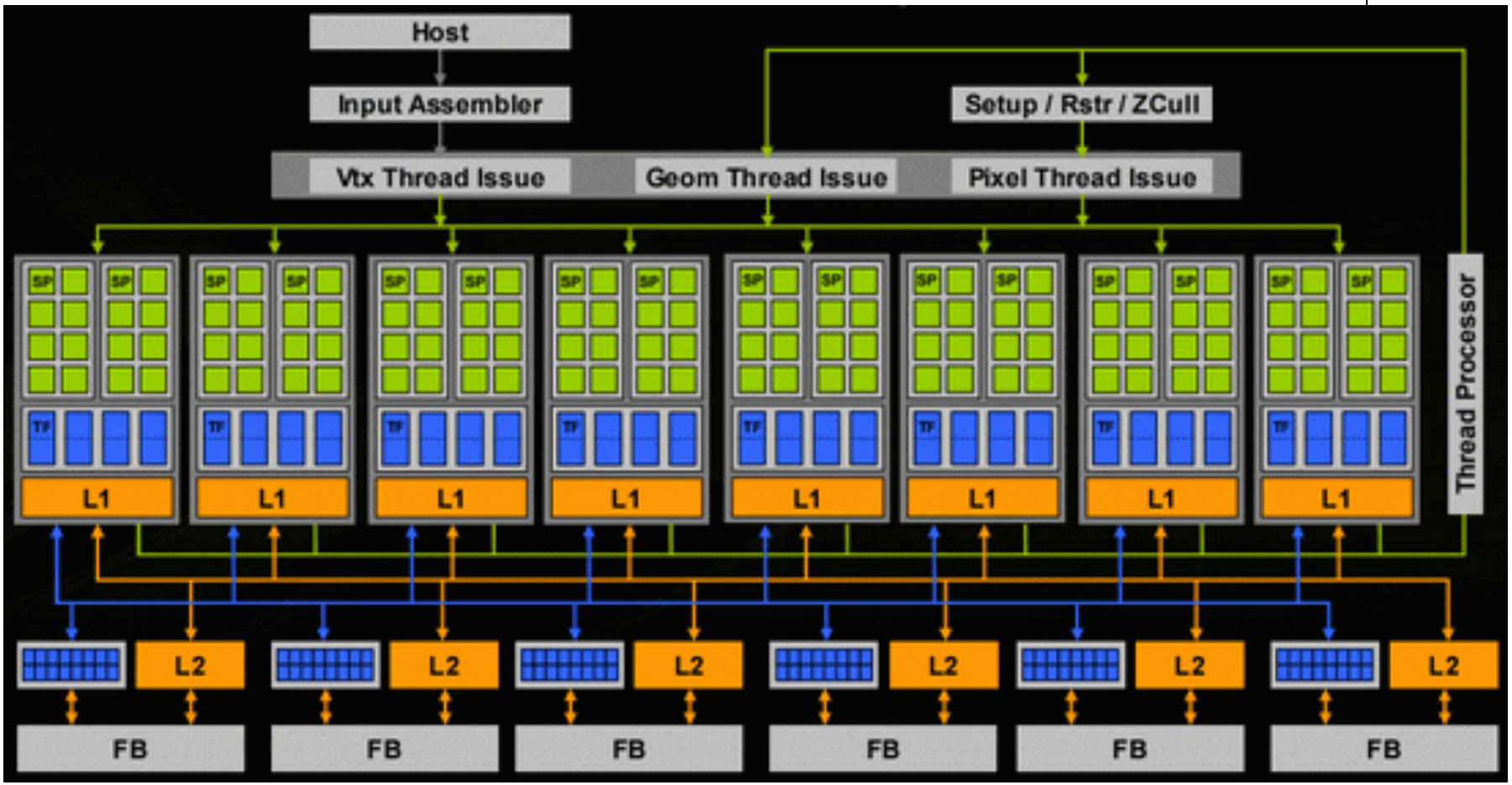
AMD Ryzen CPU Architecture



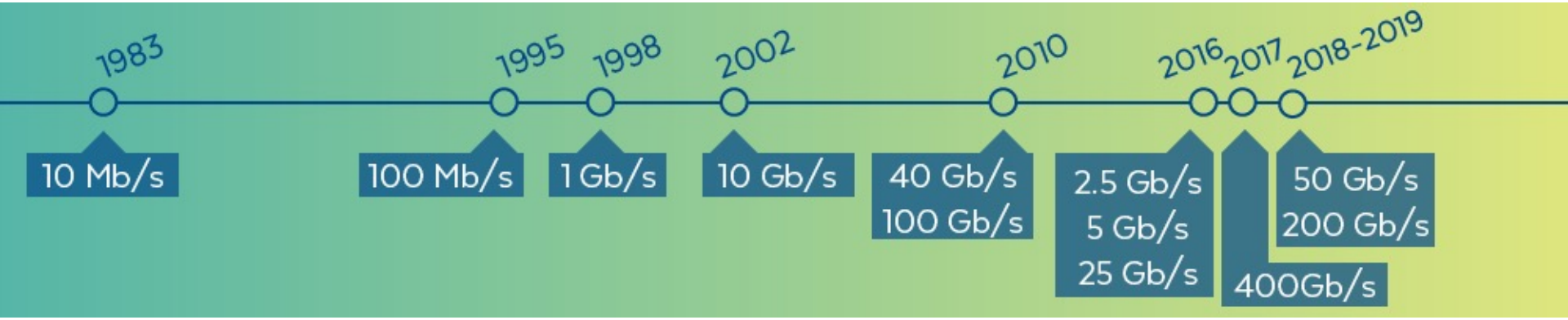
Die shot overlaid with functional units

https://www.techpowerup.com/review/amd-ryzen-5-1600/3.html

Graphics Processing Units (GPUs)

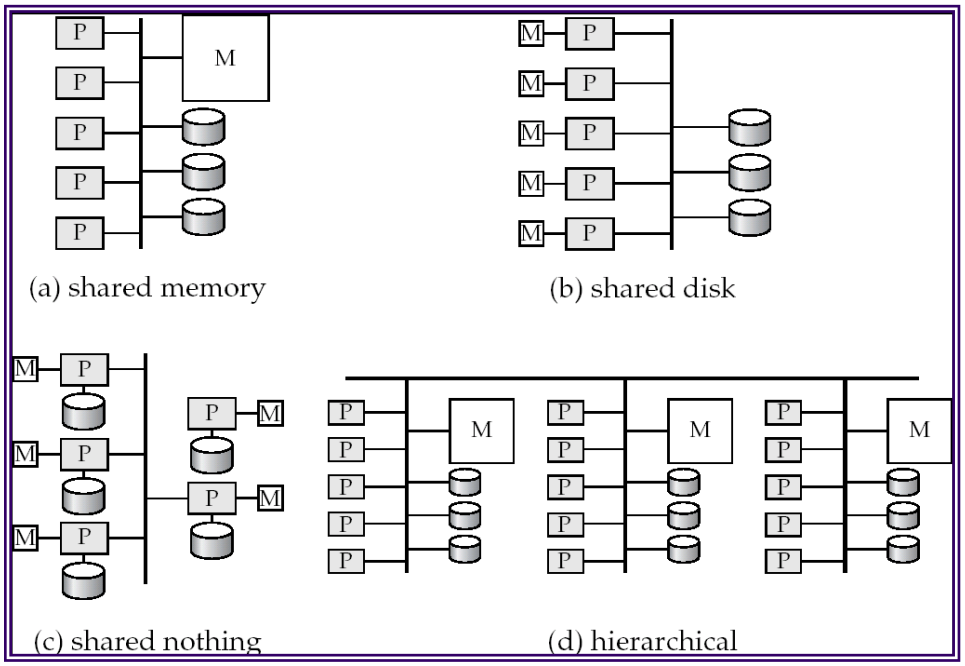


Networks



Faster now to access another computer's memory (in a data center) than your own disk.

Parallel Architectures



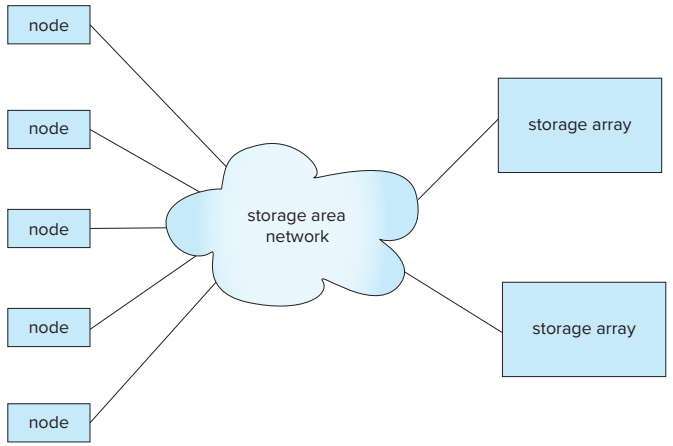


Figure 20.8 Storage-area network.

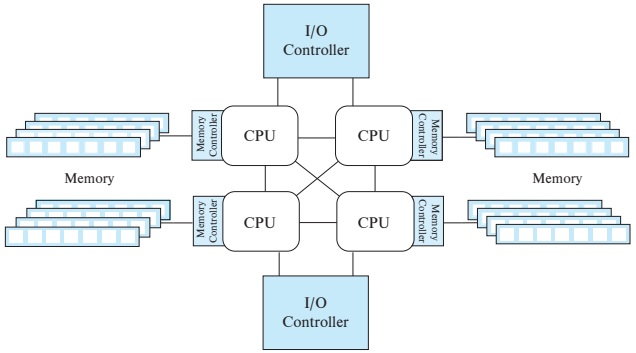


Figure 20.6 Architecture of a modern shared-memory system.

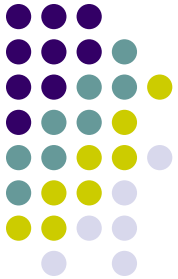
CMSC424: Database Design

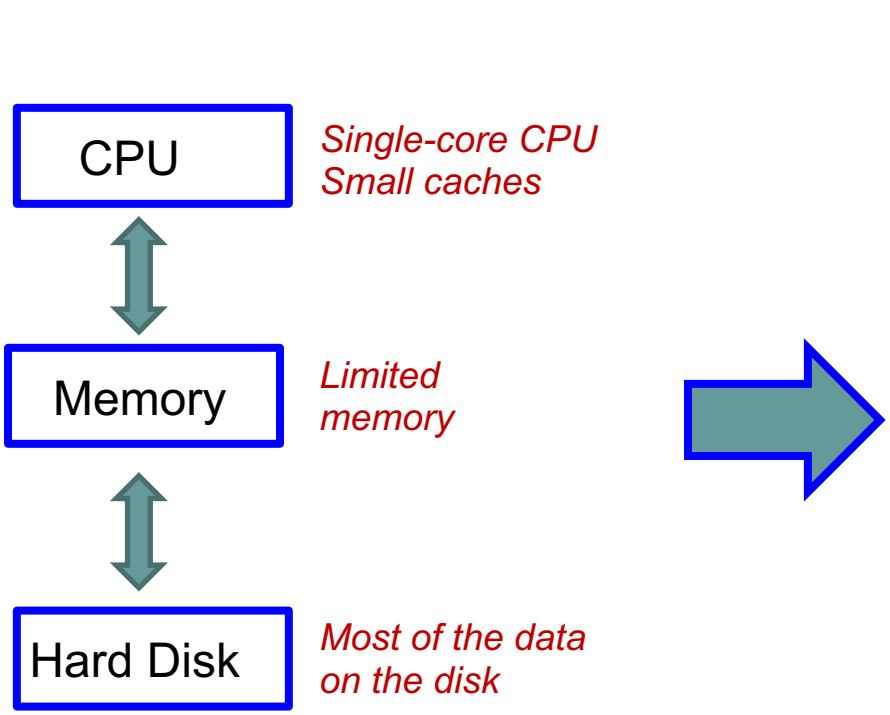
Module: Database Implementation

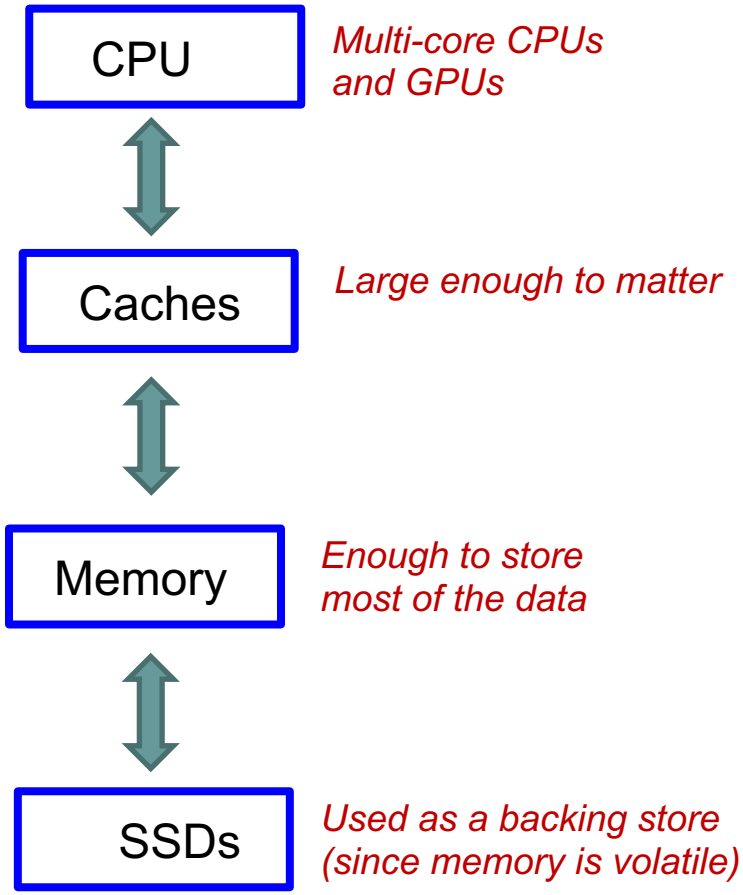
Architectures and Tradeoffs

Instructor: Amol Deshpande
amol@umd.edu

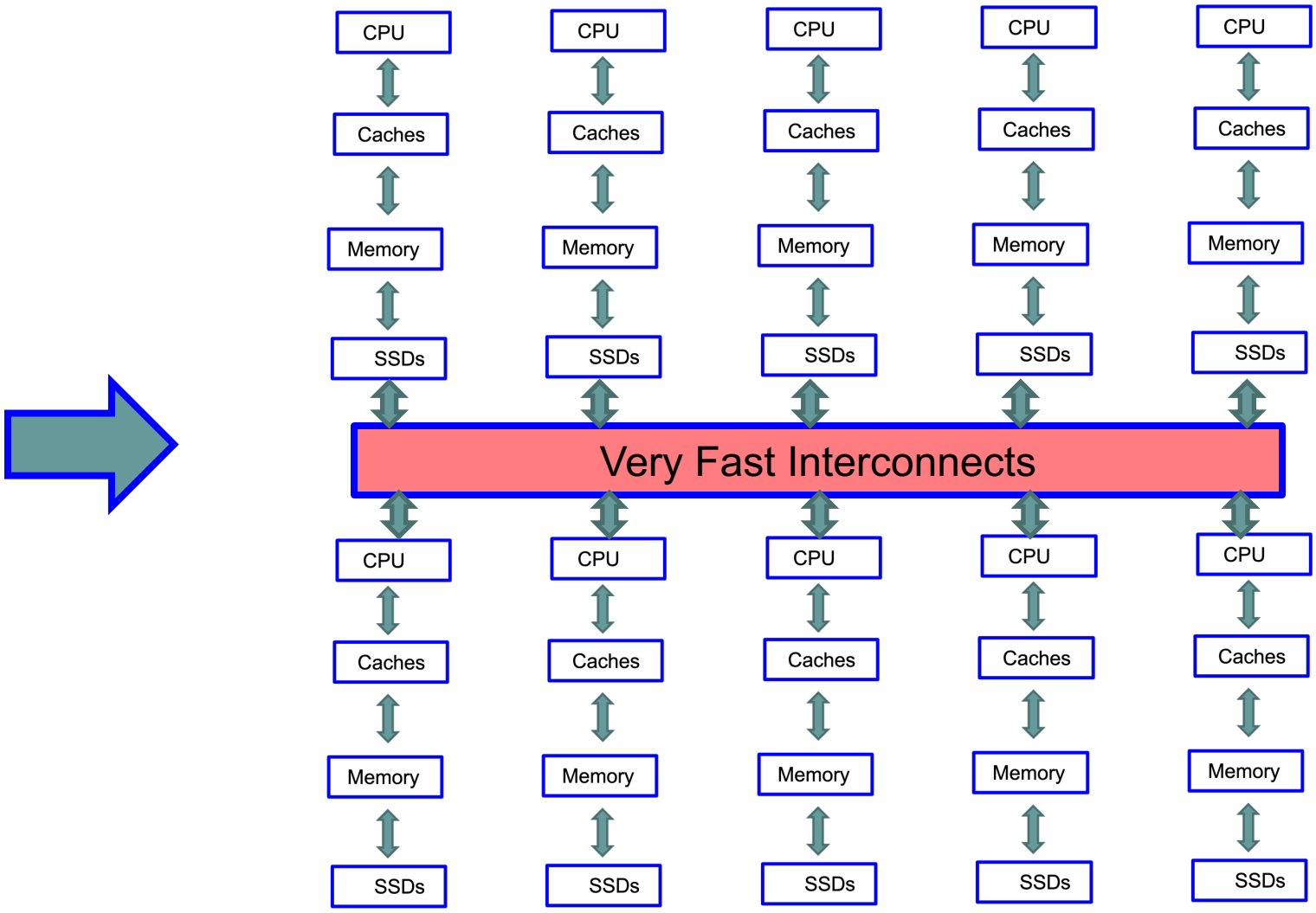
Database Architectures



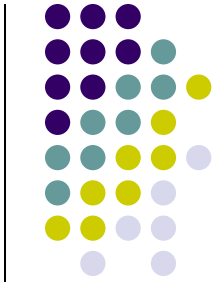




Database Architectures



What we will cover



- Hard to cover all different compute configurations
- We will cover some of the key techniques developed for different environments
  - Storage formats that are used in different settings
  - Indexing data structures for disk-resident data, as well as memory
  - etc..
- We will try keep as much of the discussion general as possible
  - Query processing and optimization
  - We have already covered some of the parallel techniques
  - Transaction processing discussion relatively abstract

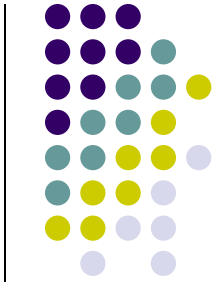
CMSC424: Database Design

Module: Database Implementation

Storage Representations

Instructor: Amol Deshpande
amol@umd.edu

Storage Representations



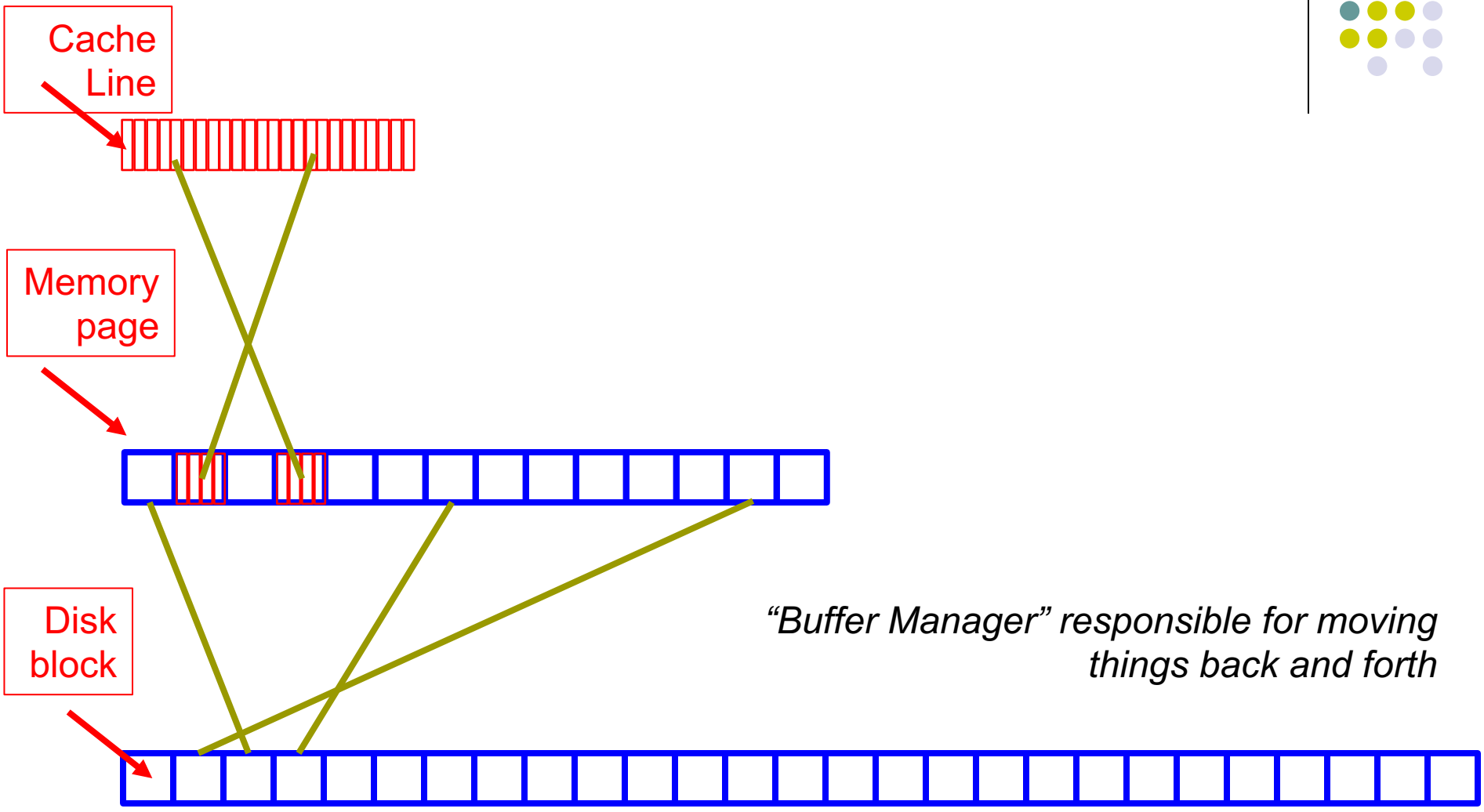
- Book Chapters
  - 10.5, 10.6
  - 13.6 (column-oriented storage)

- Key topics:
  - Different ways the tuples are laid out in disk/memory
  - Pros and cons of the different approaches

Storage Representations

- Hard disks/SSDs/Memory are all typically divided into “blocks” or “pages”
  - Traditionally 4K or 8K
- Data movement usually in units of blocks/pages
  - i.e., we usually move one or more blocks from disk to memory
  - Can’t move just a small part of a block
- Memory-to-Cache is in smaller units
  - Called “cache lines” -- 64 bytes or 128 bytes
  - Usually each memory page is divided into multiple cache lines

Storage Representations



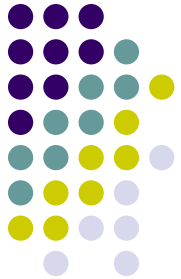
"Buffer Manager" responsible for moving things back and forth

Storage Representations

- Hard disks/SSDs/Memory are all typically divided into “blocks” or “pages”
  - Traditionally 4K or 8K
- Data movement usually in units of blocks/pages
  - i.e., we usually move one or more blocks from disk to memory
  - Can’t move just a small part of a block
- Memory-to-Cache is in smaller units
  - Called “cache lines” -- 64 bytes or 128 bytes
  - Usually each memory page is divided into multiple cache lines

- ➔ Important to allocate tuples to blocks carefully
  - Want to keep data that is accessed together in the same block
  - Fetching 4KB of data, and only using 100 bytes not a good idea
- ➔ Even within a block, how data is laid out matters for cache performance

Additional Considerations



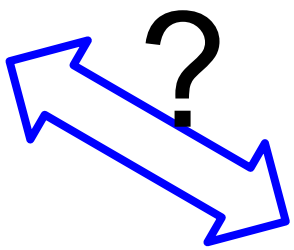
- Need to support efficient search/updates
  - e.g., may wish to keep the data sorted → inserts/deletes are tricky
  - May wish to build indexes/search trees on top of the blocks
- Want to avoid serialization/deserialization overhead → ideal to use the same format in memory as on disk
  - So after loading a block from disk to memory, the execution engine can immediately start reading/writing it
- For hard disks: sequential reads are much much better than random block reads
  - So prefer to arrange data so we do few random reads
- For SSDs, “writes” needs to be done carefully -- updating a single byte on a page usually not a good idea

Mapping Tuples to Blocks

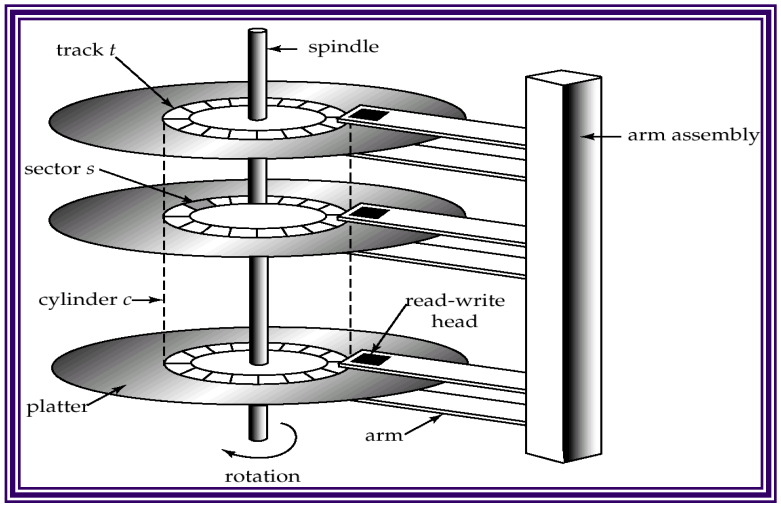
ID	name	salary	dept_name	building	budget
22222	Einstein	95000	Physics	Watson	70000
12121	Wu	90000	Finance	Painter	120000
32343	El Said	60000	History	Painter	50000
45565	Katz	75000	Comp. Sci.	Taylor	100000
98345	Kim	80000	Elec. Eng.	Taylor	85000
76766	Crick	72000	Biology	Watson	90000
10101	Srinivasan	65000	Comp. Sci.	Taylor	100000
58583	Califieri	62000	History	Painter	50000
838				aylor	100000
151	Mozart	40000	Music	Packard	80000
334	Gold	87000	Physics	Watson	70000
765	Singh	80000	Finance	Painter	120000

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

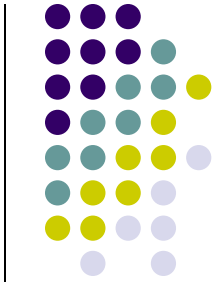
dept_name	building	budget
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000



- Very important implications on performance
- Quite a few different ways to do this
- Similar issues even if not using disks as the primary storage

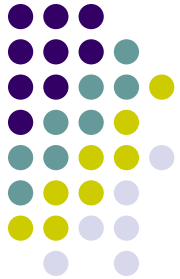


Mapping Tuples to Blocks



- Requirements and Performance Goals:
  - Allow insertion/deletions of tuples/records in relations
  - Fetch a particular record (specified by record id)
  - Find all tuples that match a condition (say SSN = 123) ?
  - Fetch all tuples from a specific relation (scans)
    - Faster if they are all sequential/in contiguous blocks
  - Allow building of “indexes”
    - Auxiliary data structures maintained on disks and in memory for faster retrieval
  - And so on…

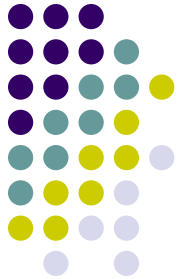
Aside: File System or Not



- Option 1: Use OS File System
  - File systems are a standard abstraction provided by Operating Systems (OS) for managing data
  - Major Con: Databases don't have as much control over the physical placement anymore --- OS controls that
    - E.g., Say DBMS maps a relation to a "file"
    - No guarantee that the file will be "contiguous" on the disk
    - OS may spread it across the disk, and won't even tell the DBMS

- Option 2: DBMS directly works with the disk or uses a lightweight/custom OS
  - Increasingly uncommon – most DBMSs today run on top of OSes (e.g., PostgreSQL on your laptop, or on linux VMs in the cloud, or on a distributed HDFS)

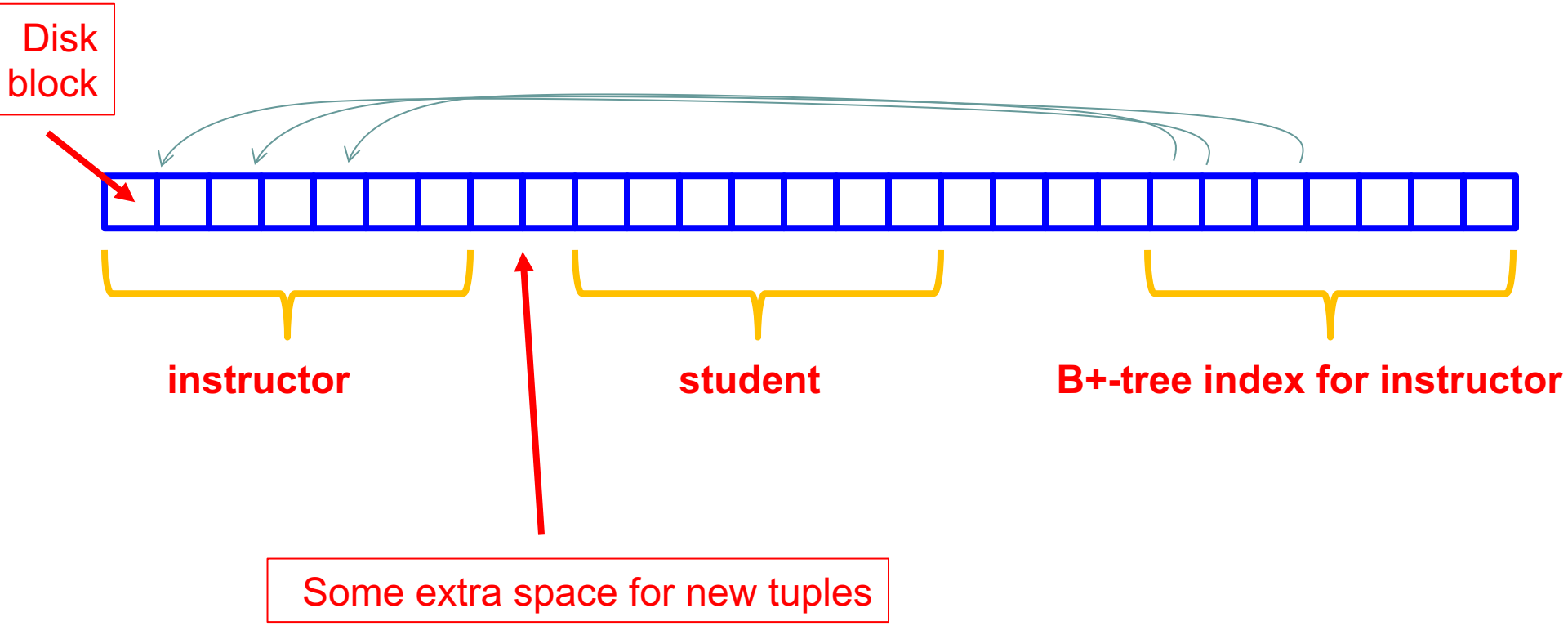
Through a File System



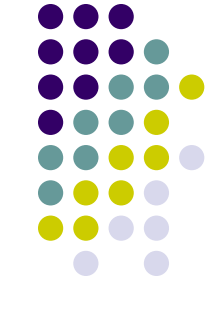
- Option 1: Allocate a single “file” on the disk, and treat it as a contiguous sequence of blocks
  - This is what PostgreSQL does
  - The blocks may not actually be contiguous on disk
- Option 2: A different file per relation
  - Some of the simpler DBMS use this approach
- Either way: we have a set of relations mapped to a set of blocks on disk

Assumptions for Now

- Each relation stored separately on a separate set of blocks
  - Assumed to be contiguous
- Each "index" maintained in a separate set of blocks
  - Assumed to be contiguous



Within block: Fixed Length Records

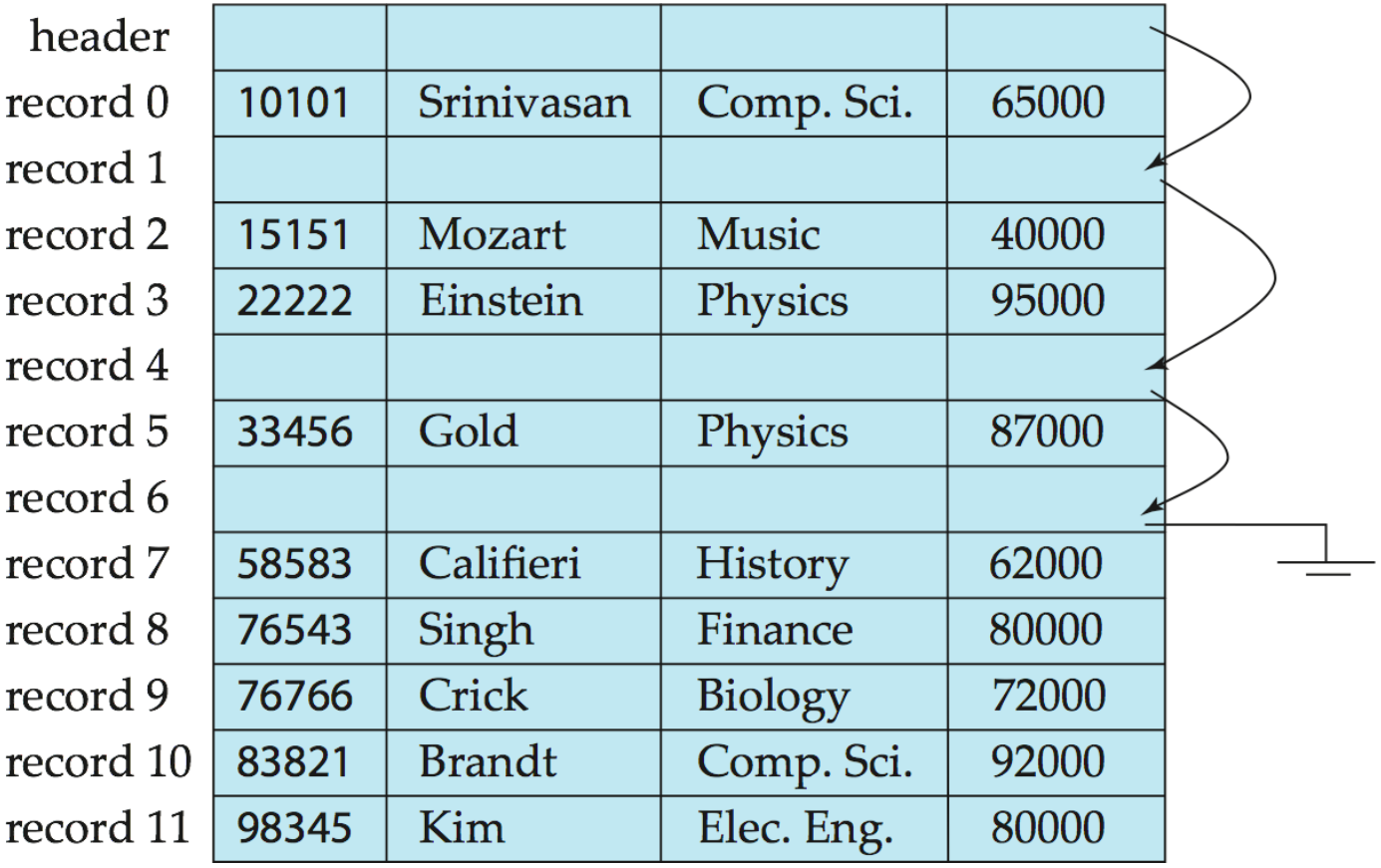


- n = number of bytes per record
- Store record *i* at position:
  - n * (i – 1)
- Records may cross blocks
  - Not desirable
  - Stagger so that that doesn't happen
- Inserting a tuple ?
  - Depends on the policy used
  - One option: Simply append at the end of the record
- Deletions ?
  - Option 1: Rearrange
  - Option 2: Keep a *free list* and use for next insert

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Within block: Fixed Length Records

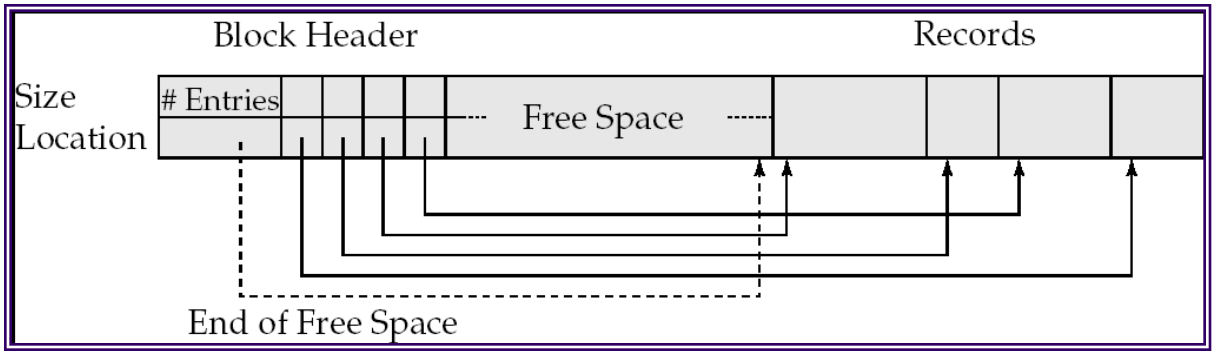
- Deleting: using “free lists”



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

Within block: Variable-length Records

Slotted page/block structure



- *Indirection:*
  - The records may move inside the page, but the outside world is oblivious to it
  - Why ?
    - The headers are used as a indirection mechanism
    - *Record ID 1000 is the 5th entry in the page number X*

Within block: Cache-optimized Structure

Row-wise storage (called NSM here) not optimized for caches

select *name*
from *R*
where *age* < 40;

Most of the data brought into cache not useful
➔ Lot of "cache misses"

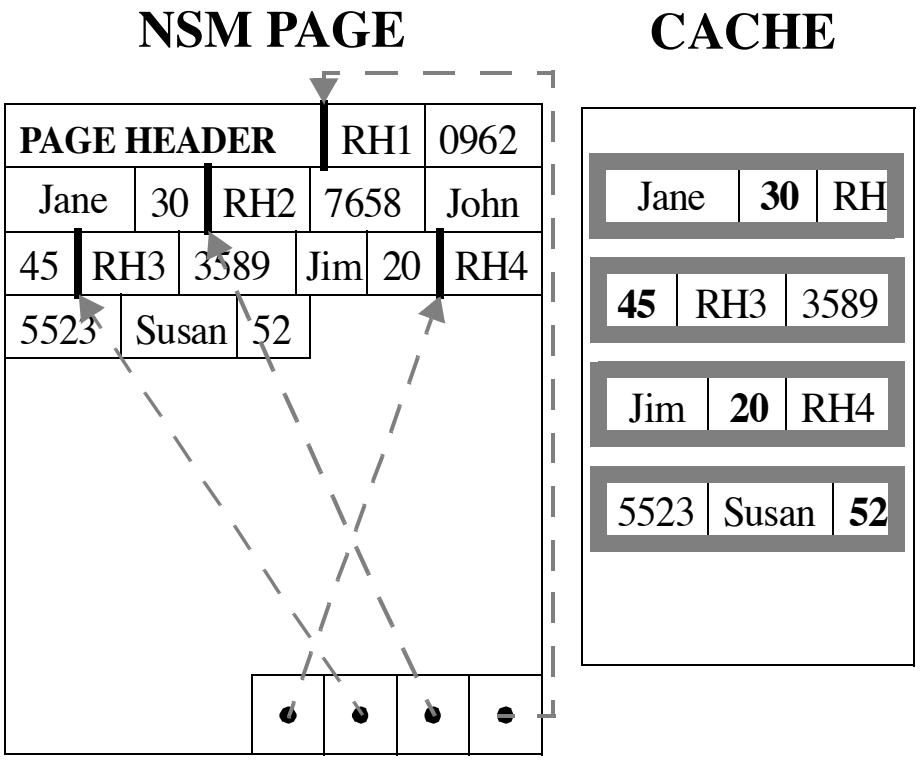
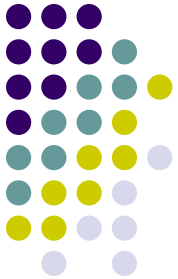


FIGURE 1: *The cache behavior of NSM.*

Within block: Cache-optimized Structure



Instead, lay out the data column-wise inside a block/page (called PAX below)

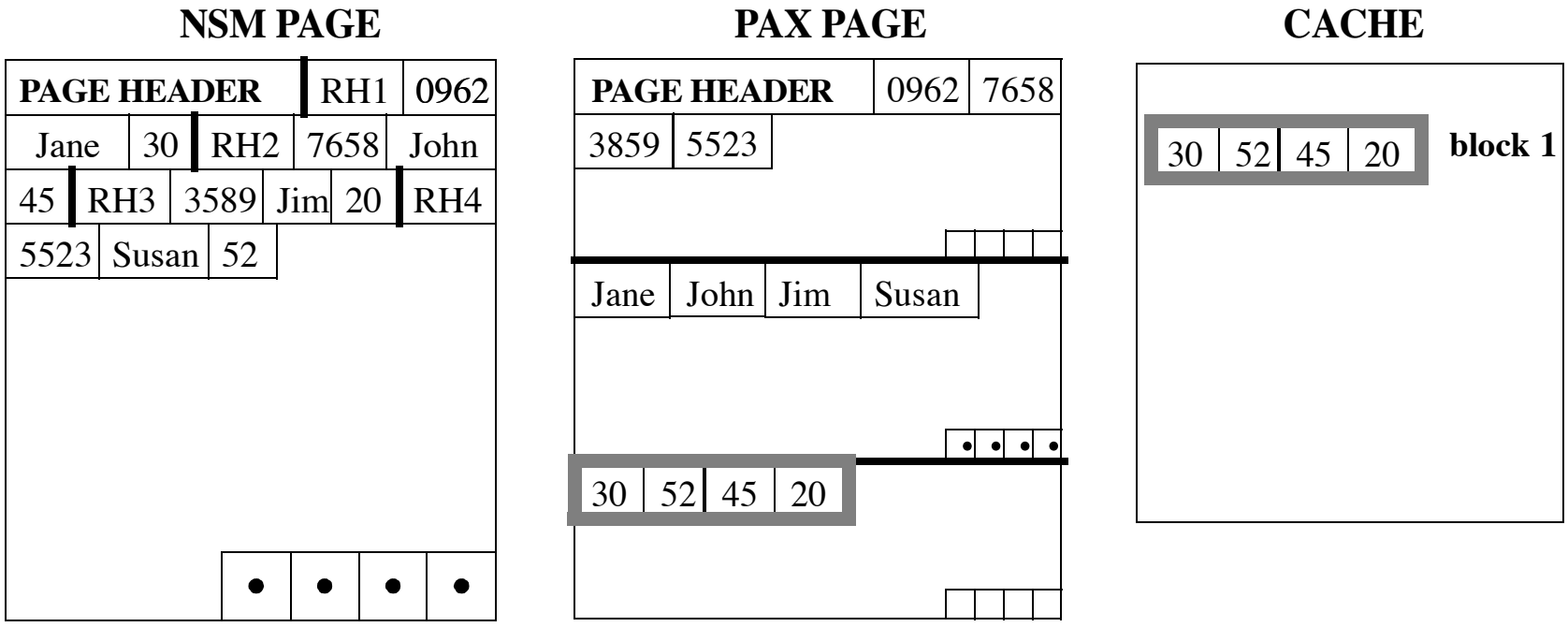
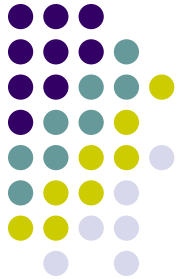


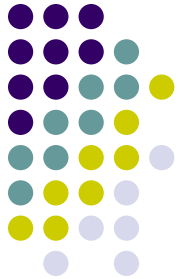
FIGURE 3: Partition Attributes Across (PAX), and its cache behavior. *PAX partitions records into minipages within each page. As we scan R to read attribute* age,*values are much more efficiently mapped onto cache blocks, and the cache space is now fully utilized.*

Across Blocks of a Relation



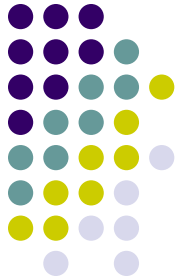
- Which block should a record go to ?
  - Anywhere ?
    - How to search for “SSN = 123” ?
    - Called “heap” organization
  - Sorted by SSN ?
    - Called “sequential” organization
    - Keeping it sorted would be painful
    - How would you search ?
  - Based on a “hash” key
    - Called “hashing” organization
    - Store the record with SSN = x in the block number x%1000
    - Why ?

Across Blocks: Sequential File Organization



- Keep sorted by some search key
- Insertion
  - Find the block in which the tuple should be
  - If there is free space, insert it
  - Otherwise, must create overflow pages
- Deletions
  - Delete and keep the free space
  - Databases tend to be insert heavy, so free space gets used fast
- Can become *fragmented*
  - Must reorganize once in a while

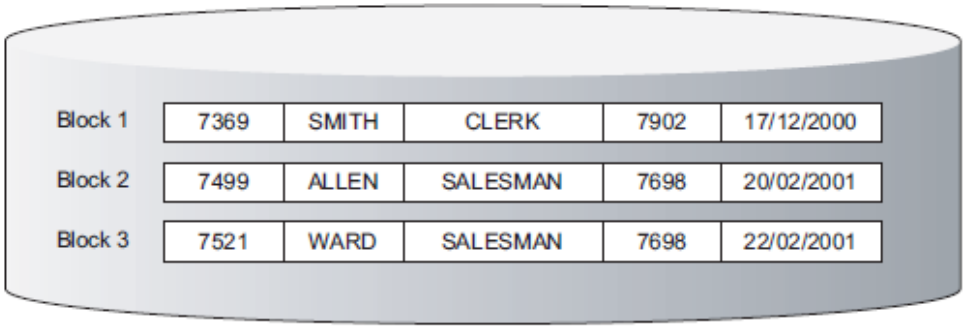
Across Blocks: Sequential File Organization



- What if I want to find a particular record by value ?
  - *Account info for SSN = 123*
- Binary search
  - Takes *log(n)* number of disk accesses
    - Random accesses
  - Too much
    - n = 1,000,000,000 -- log(n) = 30
    - Recall each random access approx 10 ms
    - 300 ms to find just one account information
    - < 4 requests satisfied per second

Indexes – later

Column-oriented Storage

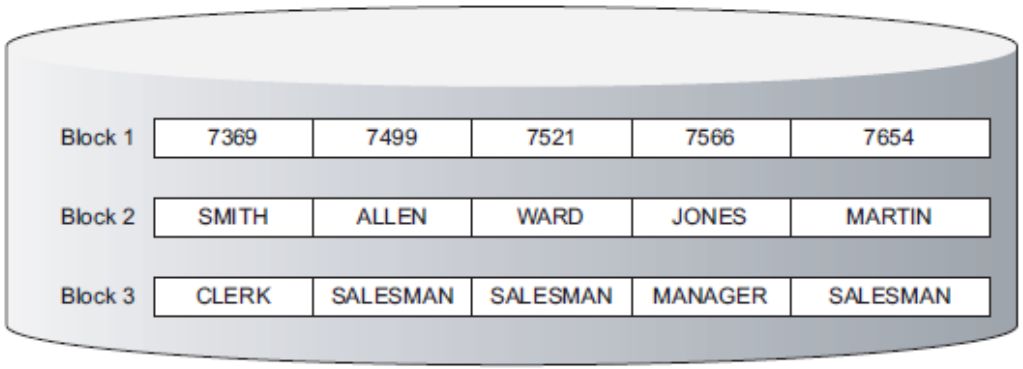


Row Database stores row values together

Row-Store Physical Layout

EmpNo	EName	Job	Mgr	HireDate
7369	SMITH	CLERK	7902	17/12/1980
7499	ALLEN	SALESMAN	7698	20/02/1981
7521	WARD	SALESMAN	7698	22/02/1981
7566	JONES	MANAGER	7839	2/04/1981
7654	MARTIN	SALESMAN	7698	28/09/1981
7698	BLAKE	MANAGER	7839	1/05/1981
7782	CLARK	MANAGER	7839	9/06/1981

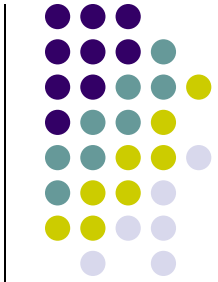
Logical Schema



Column Database stores column values together

Column Store physical layout

Column-oriented Storage



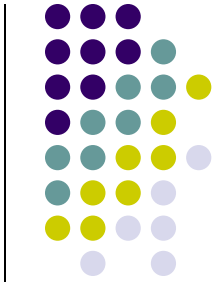
- Benefits
  - Reduced I/O -- only fetch the relevant queries
    - Especially useful when a table has say 50 columns and the query only wants 2-3 of those
  - Improved CPU Cache Performance
  - Improved compression
    - Each block contains similar data -- so more easily compressed
  - Vector processing
    - Can better utilize modern CPUs

Column-oriented Storage

- Drawbacks
  - Extra cost to reconstruct full tuples
    - To support queries like: “select * from R”
    - A single tuple is spread out across many blocks
  - Tuple deletions/updates much harder
  - Cost of decompression (if compression being used)

- Widely used today in data warehouses
  - Ideal use case -- fewer/batched updates, and very large and wide tables
  - Similar ideas also used in systems like pandas

Columnar "file" representations



- Scenario: "CSV"-like files stored in data lakes or file-systems
  - Also JSON collections (e.g., "prize.json" from Assignment 4) not being stored in a database like MongoDB
  - Modern data lakes have thousands to millions of such files, and want to support SQL over them (e.g., through Spark)

- Keeping as "text" files not efficient
  - Storage overhead -- binary compressed representations much smaller
  - Parsing text is expensive
  - Same row-vs-column representation issues
    - Since these are basically relations, often only need to read a few columns

Columnar "file" representations

Several formats developed over the years

- Parquet probably most popular right now
- ORC another one

Mostly "columnar" and designed to be primarily read-only

Usually compressed formats

- Each "chunk" of column data compressed

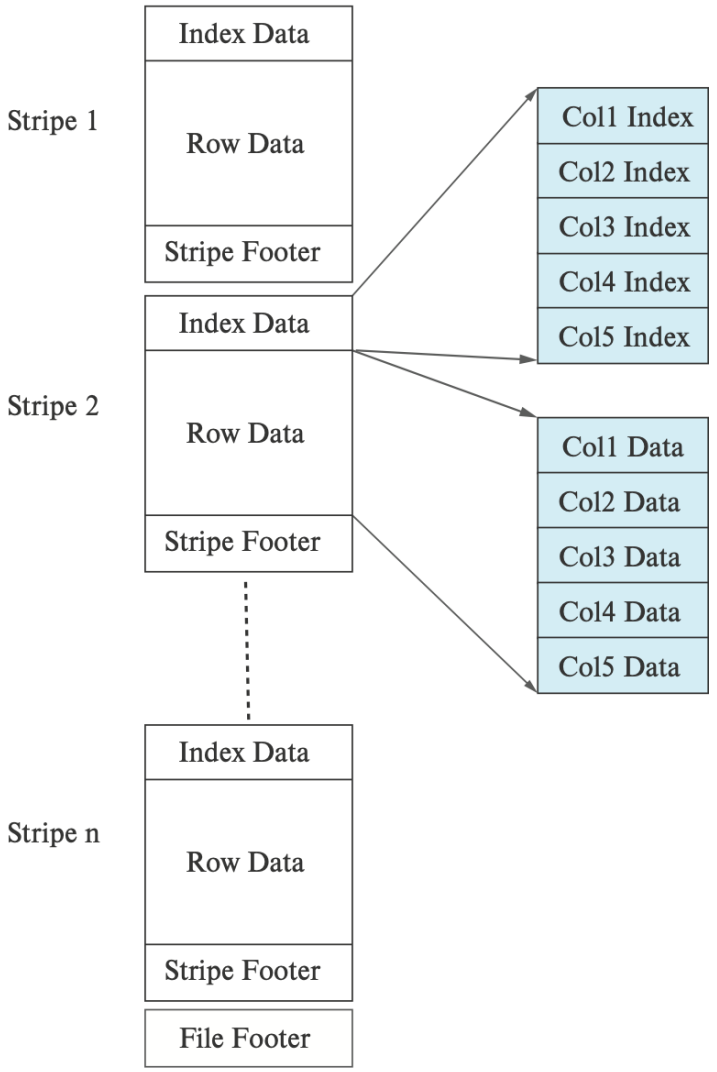
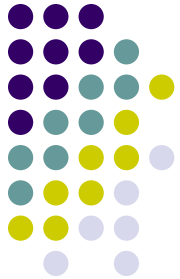


Figure 13.15 Columnar data representation in the ORC file format.

Summary



- Storage layouts an important consideration
  - Given the data sizes we see in practice
- Primary decision: row vs column representation
  - Row representations better for scenarios where updates more common
  - Column representations much better for data warehouses or data lakes (large data sizes + batched updates.+ complex queries)
  - Hybrid representations tried, but haven't really worked out
- Many other secondary considerations
  - E.g, sort orders/assignment of tuples to blocks
  - How to handle inserts/deletions
  - Cache performance and vectorized execution
- Similar issues for non-relational data (e.g., document stores)

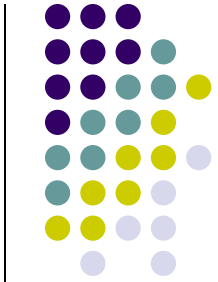
CMSC424: Database Design

Module: Database Implementation

Indexes; B+-Trees

Instructor: Amol Deshpande
amol@cs.umd.edu

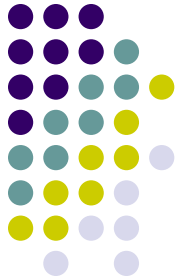
Indexes Overview



- Book Chapters
  - 11.1, 11.2, 11.3 (B+-tree)

- Key topics:
  - How an “index” helps efficiently find tuples that satisfy a condition?
  - What are key characteristics of indexes?
  - B+-Tree Indexes

Index

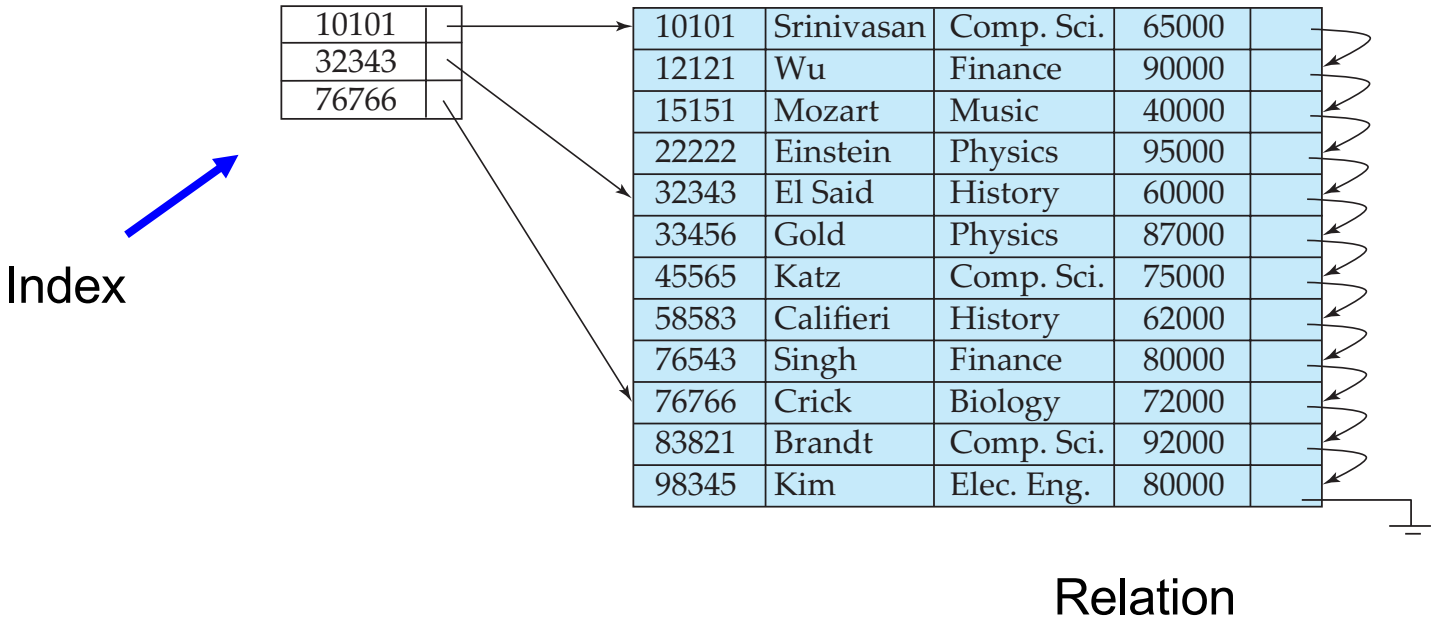


- A data structure for efficient search through large databases
- Two key ideas:
  - The records are mapped to the disk blocks in specific ways
    - Sorted, or hash-based
  - Auxiliary data structures are maintained that allow quick search
- Search key:
  - Attribute or set of attributes used to look up records
  - E.g. SSN for a persons table
- Two types of indexes
  - Ordered indexes
  - Hash-based indexes
- Think library index/catalogue


EXIT
FACTS ON

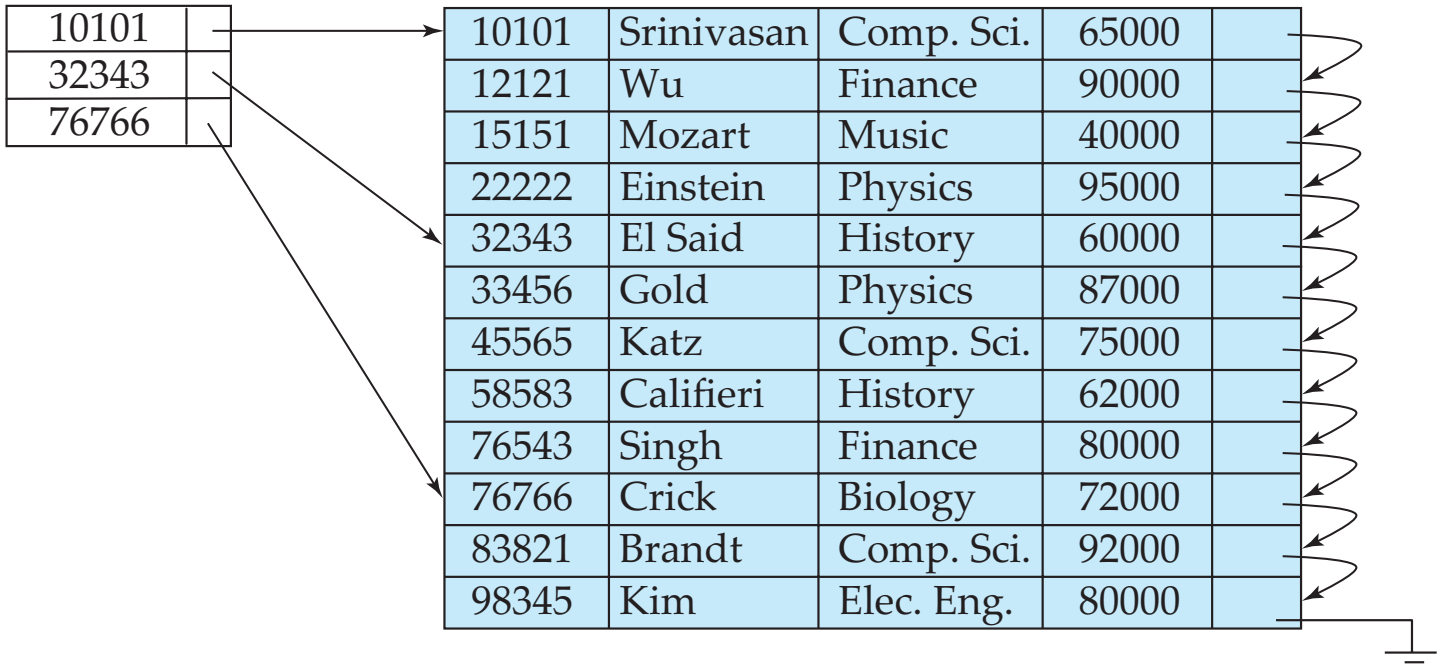
Ordered Indexes

- Primary index
  - The relation is sorted on the search key of the index
- Secondary index
  - It is not
- Can have only one primary index on a relation



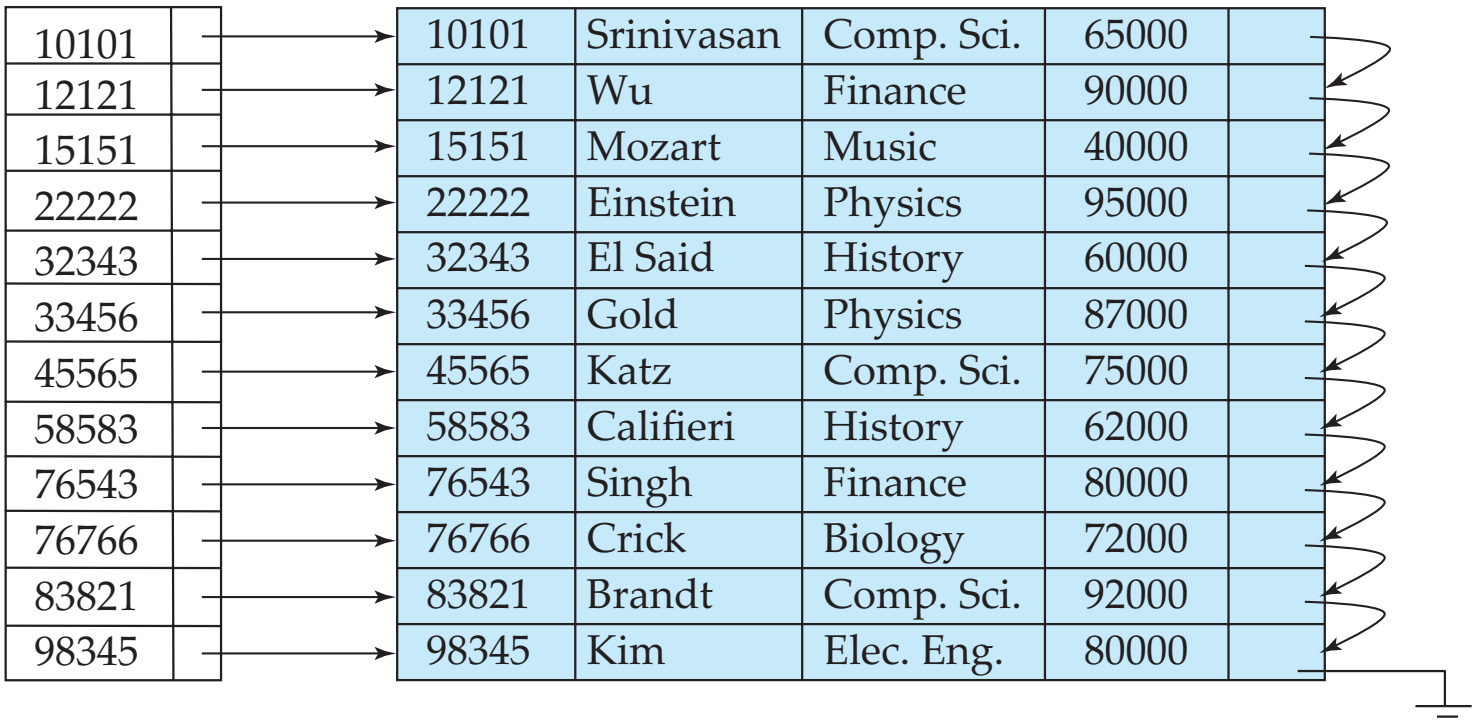
Primary *Sparse* Index

- Every key doesn't have to appear in the index
- Allows for very small indexes
  - Better chance of fitting in memory
  - Tradeoff: Must access the relation file even if the record is not present



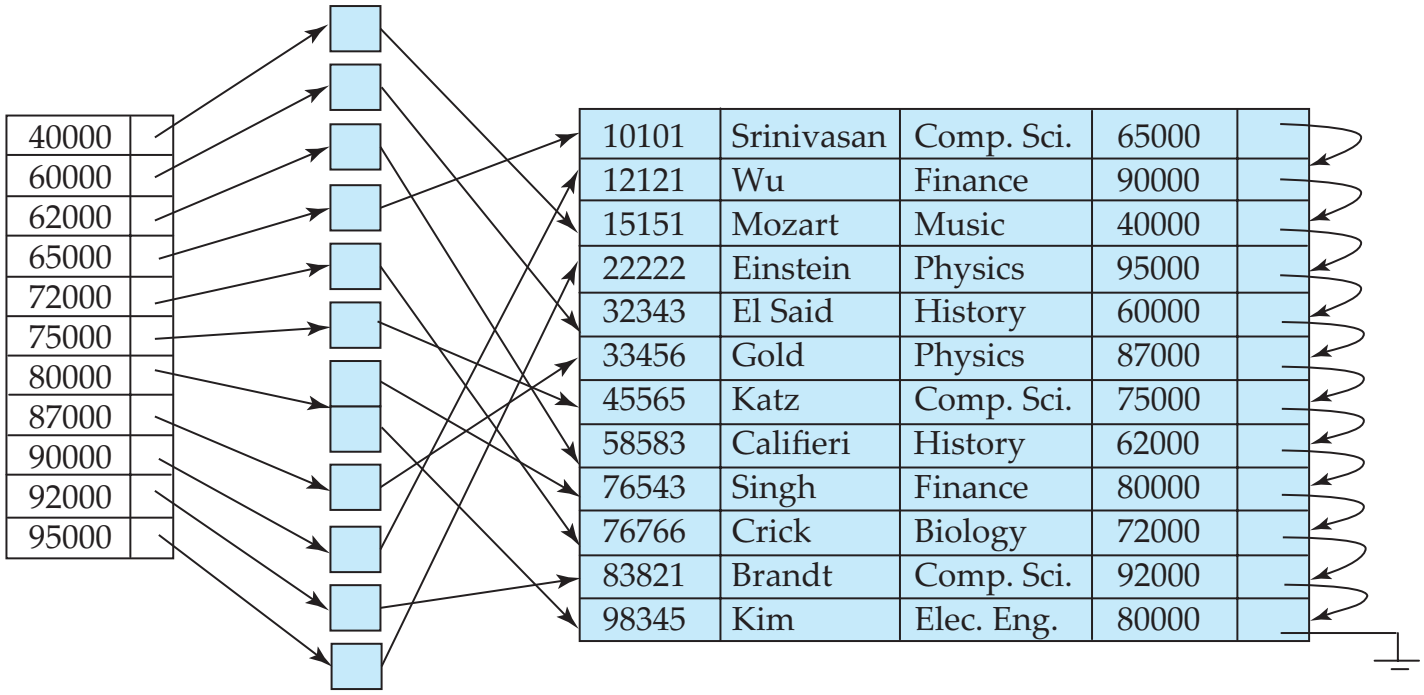
Primary *Dense* Index

- Every key must appear in the index
- Index becomes pretty large, but can often avoid having to go to the relation
  - E.g., select * from instructor where ID = 10000
    - Not found in the index, so can return immediately



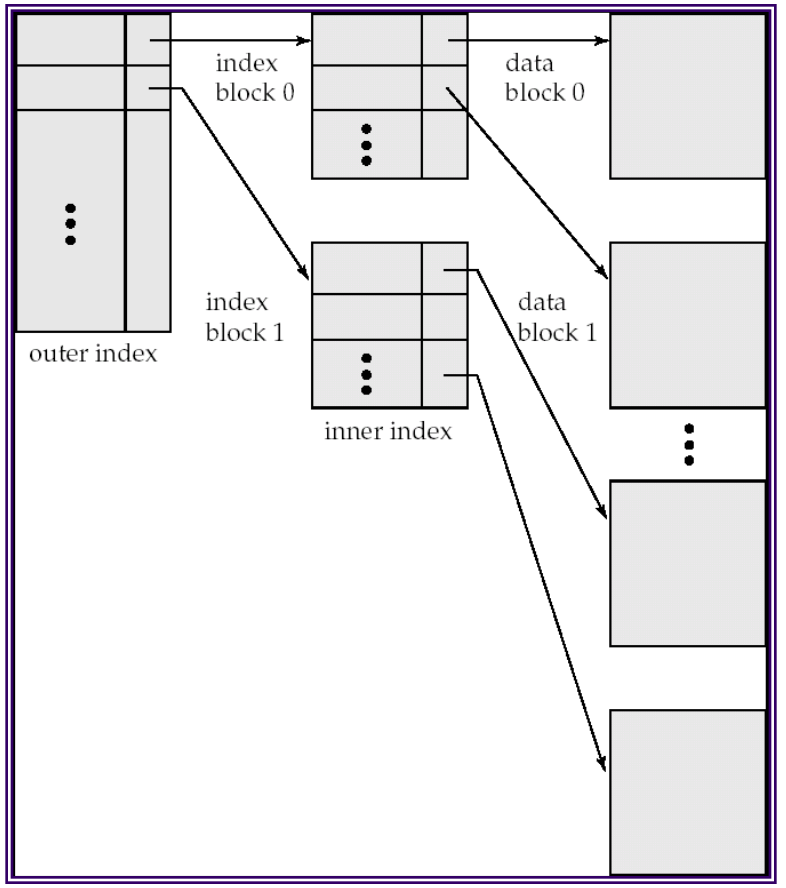
Secondary Index

- Relation sorted on *ID*
- But we want an index on *salary*
- Must be dense
  - Every search key must appear in the index

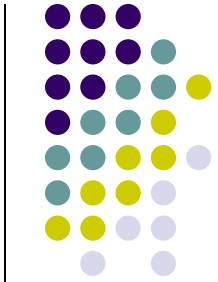


Multi-level Indexes

- What if the index itself is too big for memory ?
- Relation size = n = 1,000,000,000
- Block size = 100 tuples per block
- So, number of pages = 10,000,000
- Keeping one entry per page takes too much space
- Solution
  - Build an index on the index itself



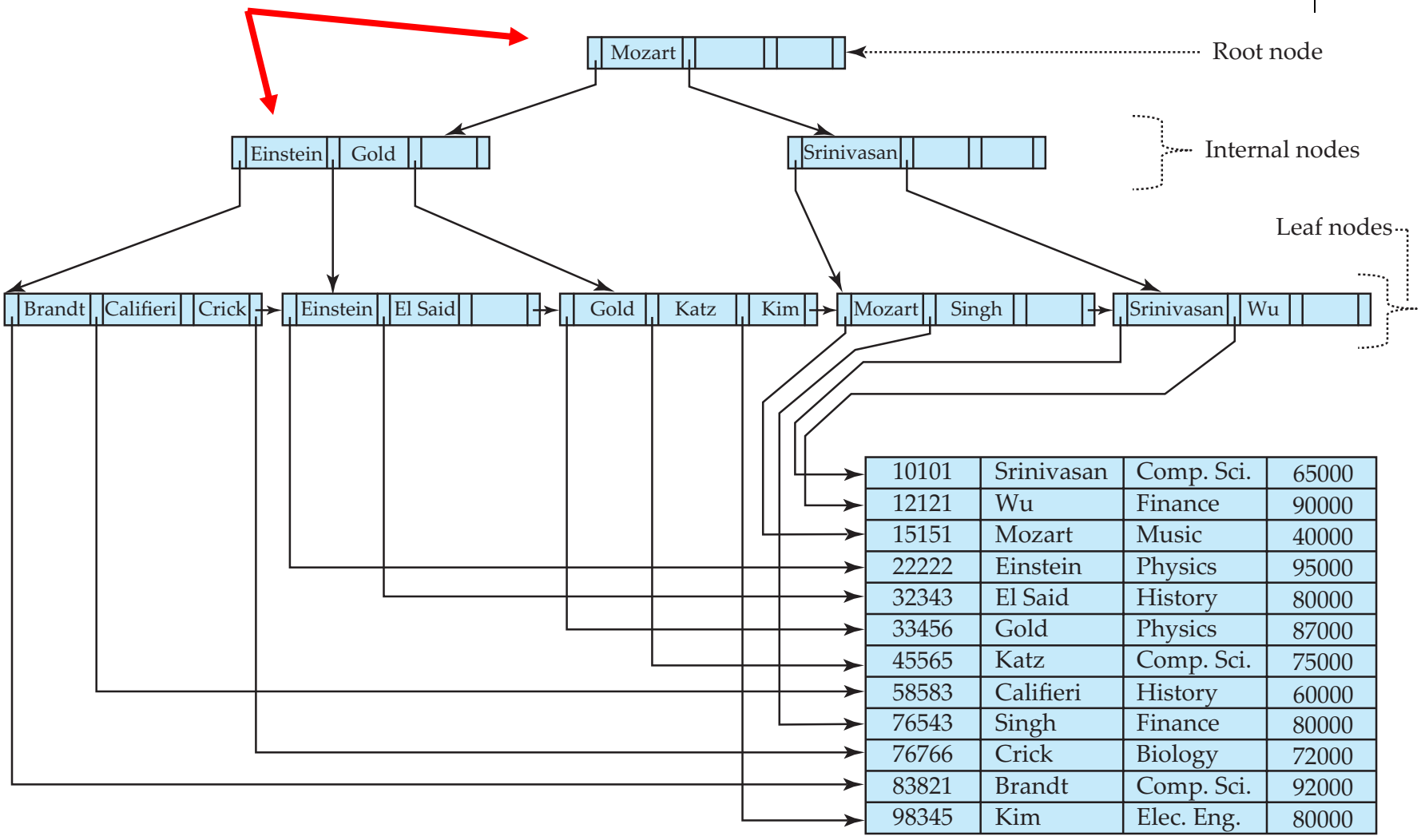
B+-Tree Indexes



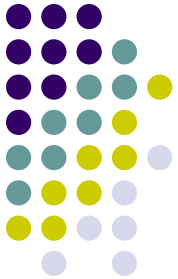
- Very widely used in relational database systems
  - Optimized for hard disks
  - Not the first choice today
- Ordered indexes
  - Allow efficient “range” searches
- Balanced and small height

B+-Tree Index

Index Disk Blocks



B+-Tree Node Structure

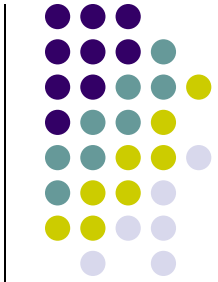


- Typical node

  - 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

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

Properties of B+-Trees

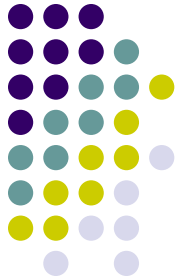


- It is balanced
  - Every path from the root to a leaf is same length

- Leaf nodes (at the bottom)
  - *P1* contains the pointers to tuple(s) with key *K1*
  - …
  - *Pn* is a pointer to the *next* leaf node
  - Must contain at least n/2 entries

P_1	K_1	P_2	. . .	P_{n-1}	K_{n-1}	P_n

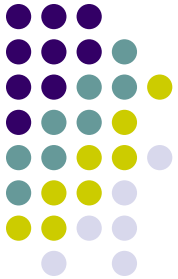
Properties



- Interior nodes

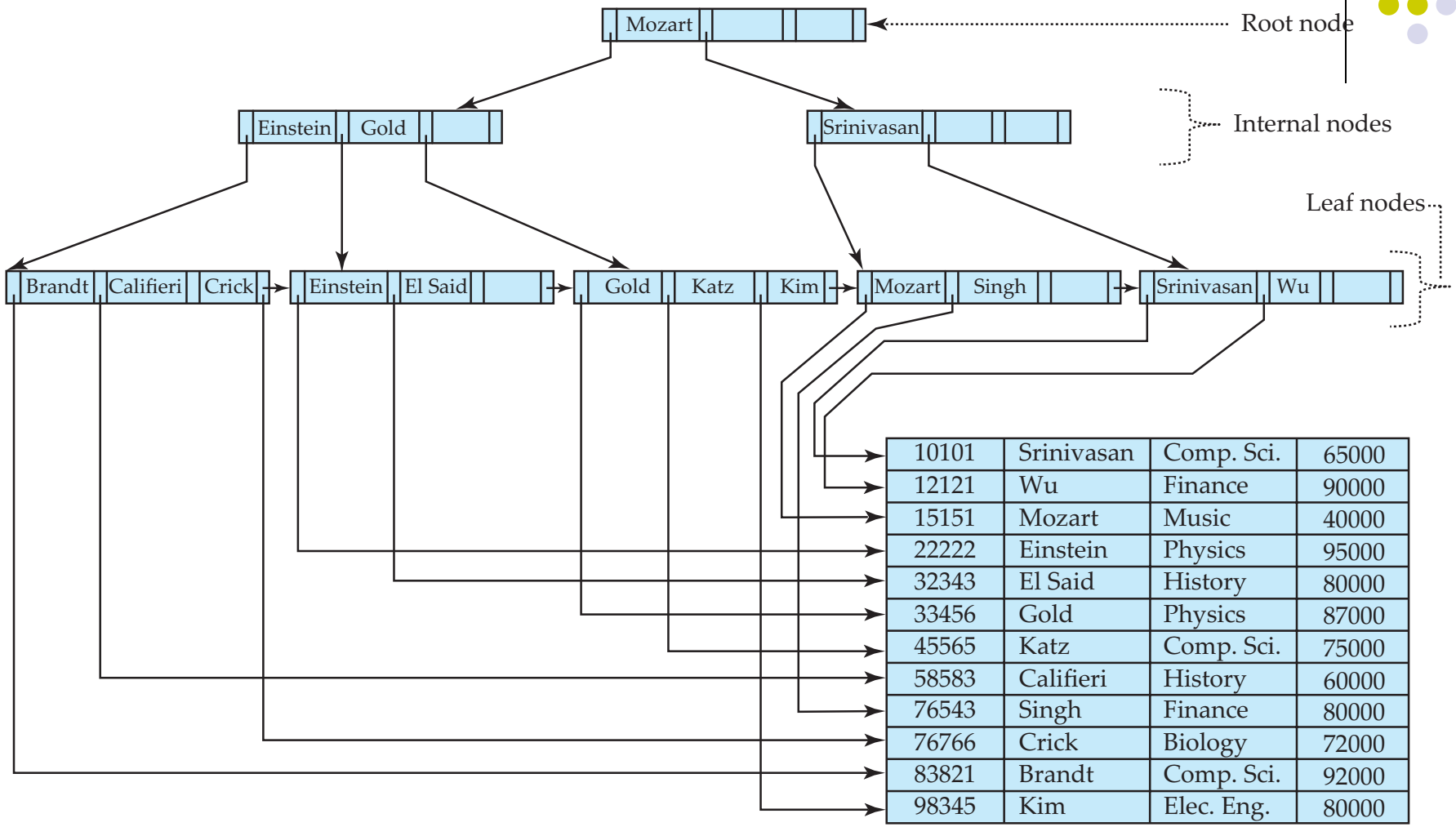
  - All tuples in the subtree pointed to by *P1,* have search key < *K1*
  - To find a tuple with key *K1'* < *K1,* follow *P1*
  - …
  - Finally, search keys in the tuples contained in the subtree pointed to by *Pn,* are all larger than *Kn-1*
  - Must contain at least n/2 entries (unless root)

B+-Trees - Searching



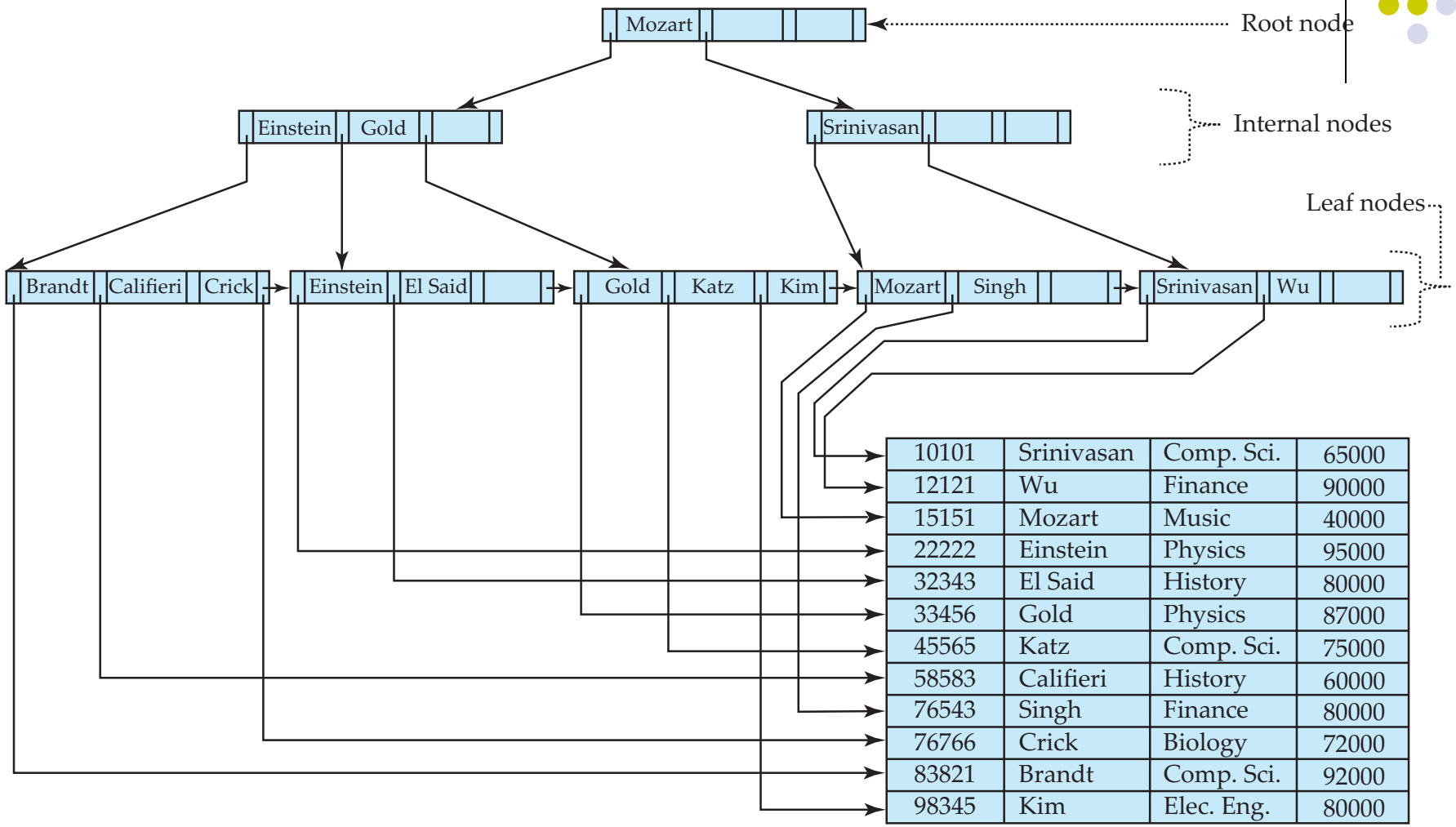
- How to search ?
  - Follow the pointers
- Logarithmic
  - $log_{B/2}(N)$, where B = *Number of entries per block*
  - B is also called the order of the B+-Tree Index
    - Typically 100 or so
- If a relation contains1,000,000,000 entries, takes only 4 random accesses
- The top levels are typically in memory
  - So only requires 1 or 2 random accesses per request

Example B+-Tree Index



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

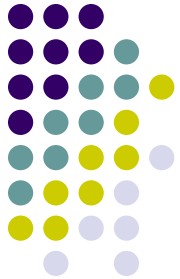
Example B+-Tree Index



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

If this were a “primary” index, then not all ”keys” are present in the index

B+-Trees - Updates



- When inserting or deleting, two properties need to be maintained
  - Each block remains at least half-full (except the root which may be less than half-full)
  - The tree must remain balanced
- When inserting, this may result in a new root being created
- When deleting, this may result in reducing the height of the tree by one

B+-Trees: Insertion

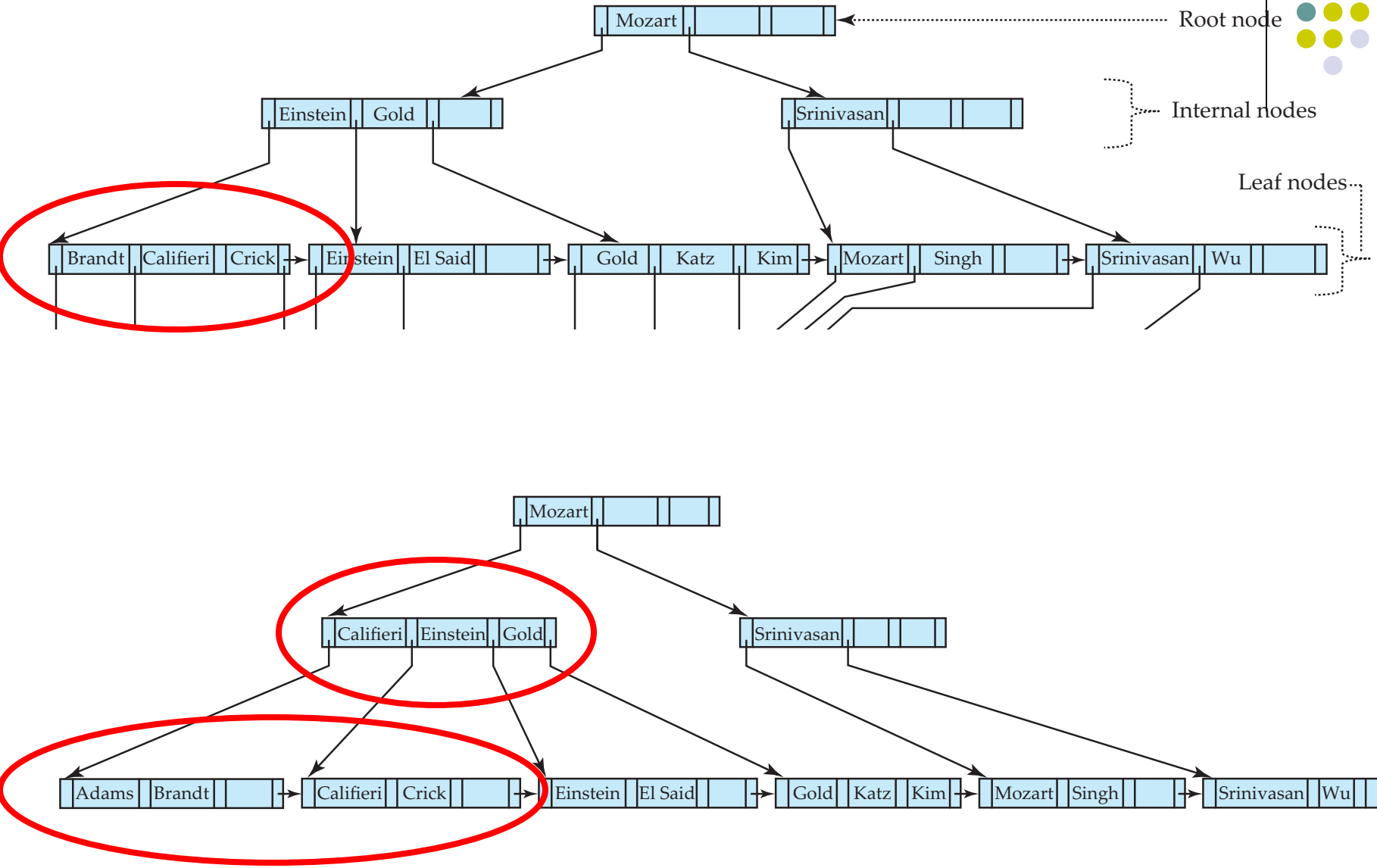


Figure 11.13 Insertion of "Adams" into the B+-tree of Figure 11.9.

B+-Trees: Insertion

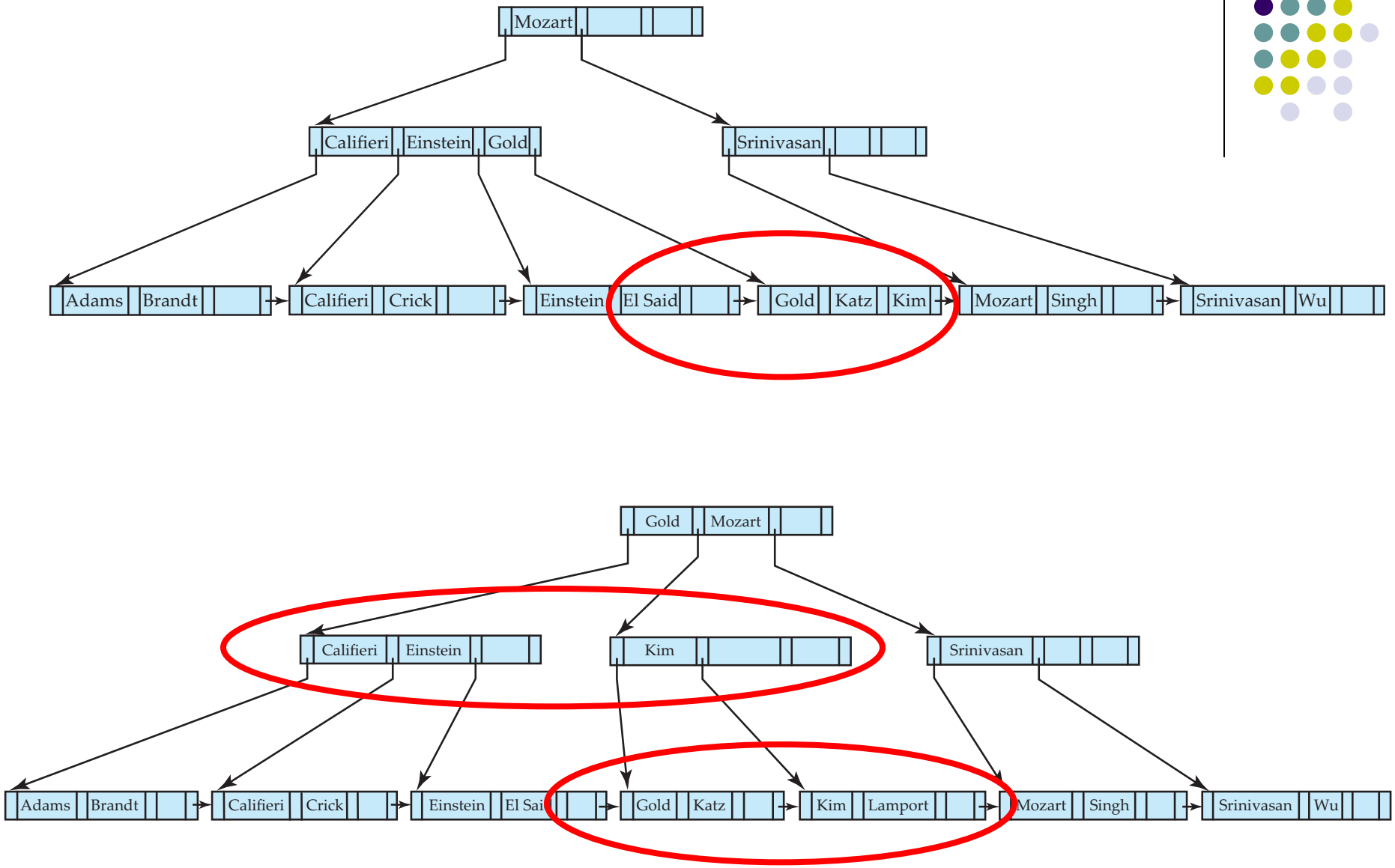
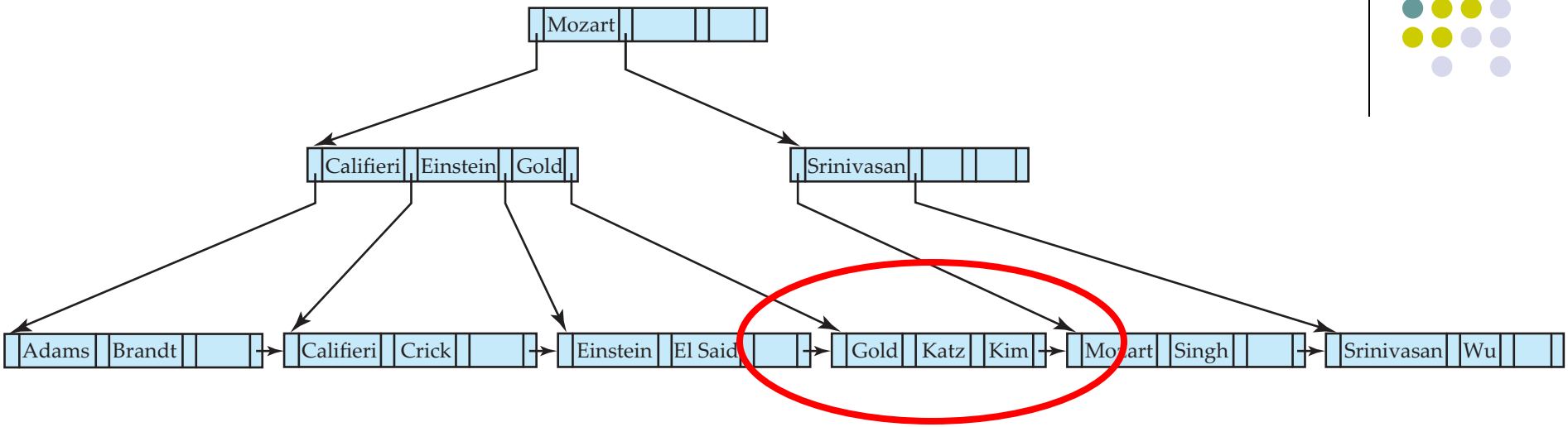


Figure 11.14 Insertion of "Lamport" into the B^+-tree of Figure 11.13.

Examples of B+-Tree Deletion



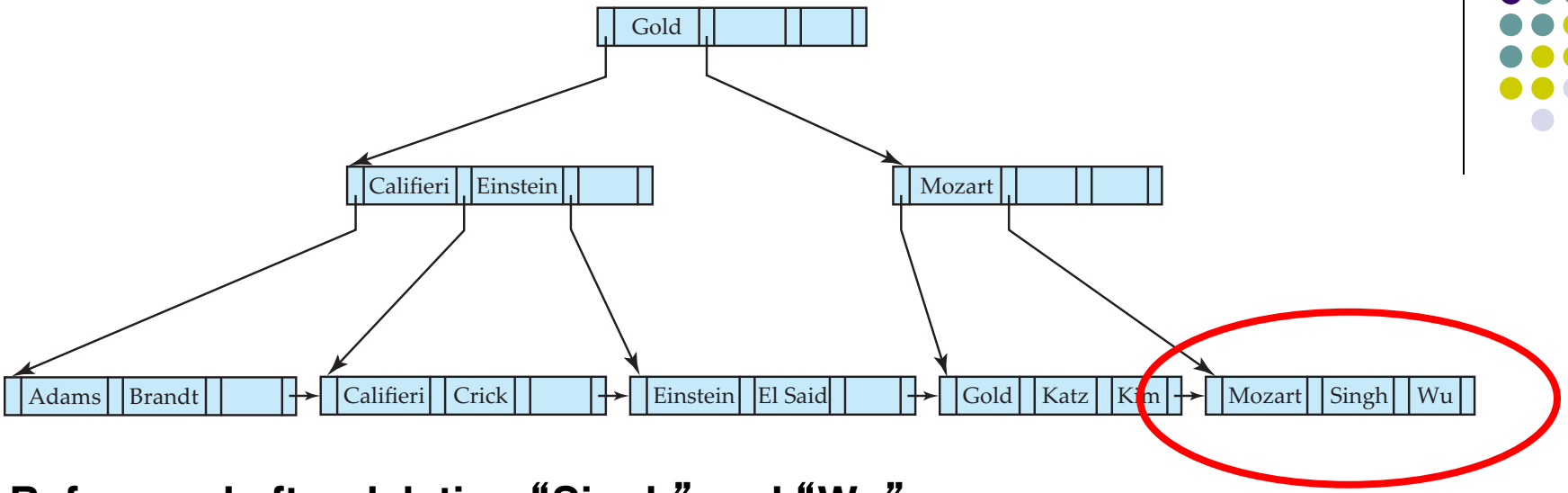
Deleting “Katz” – No issues

Deleting “Gold” – Just delete from the leaf

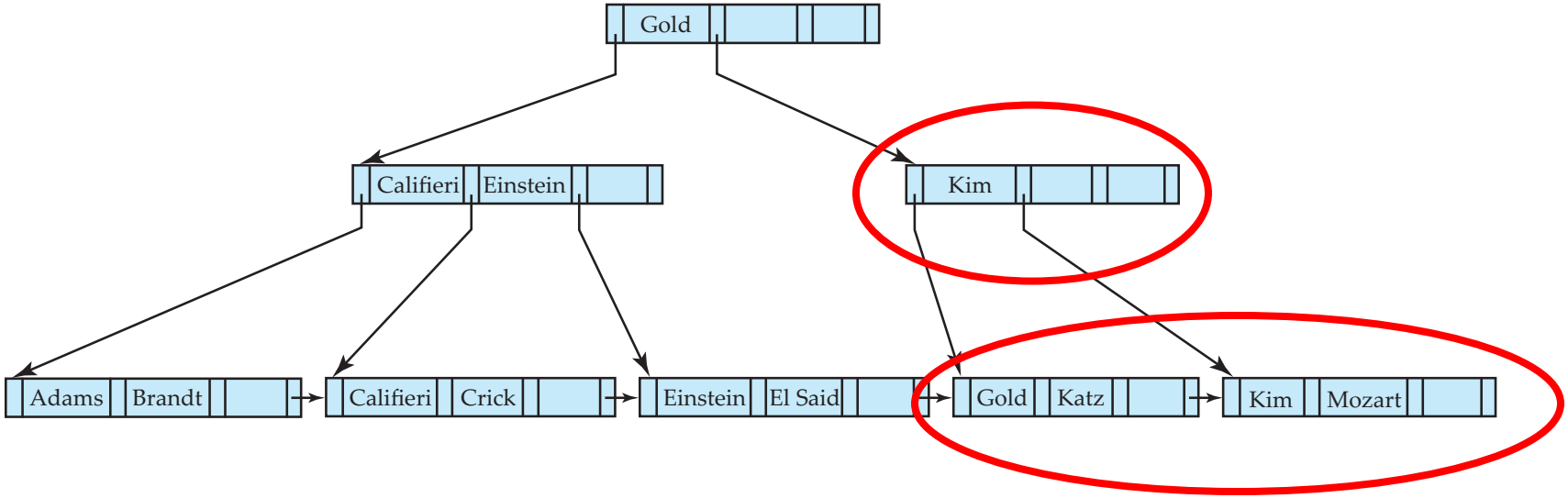
Gold can stay in the “interior” node – no need to delete it

The purpose of the search keys in the interior nodes is to “direct” searches

Examples of B+-Tree Deletion

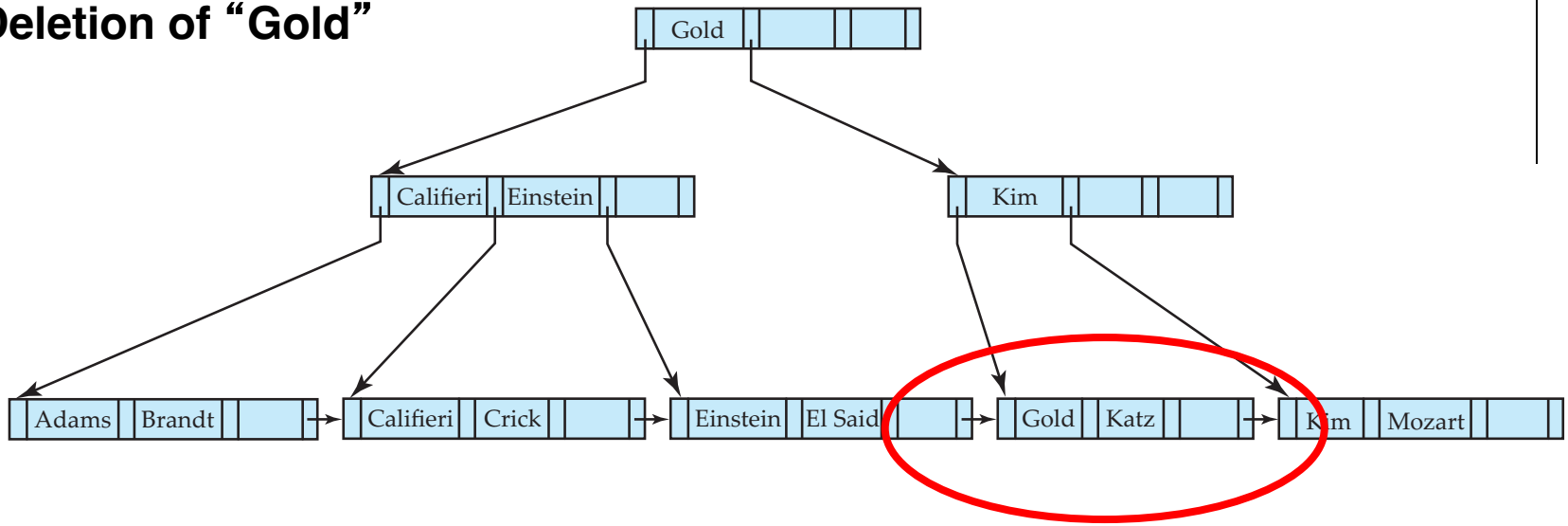


Before and after deleting "Singh" and "Wu"

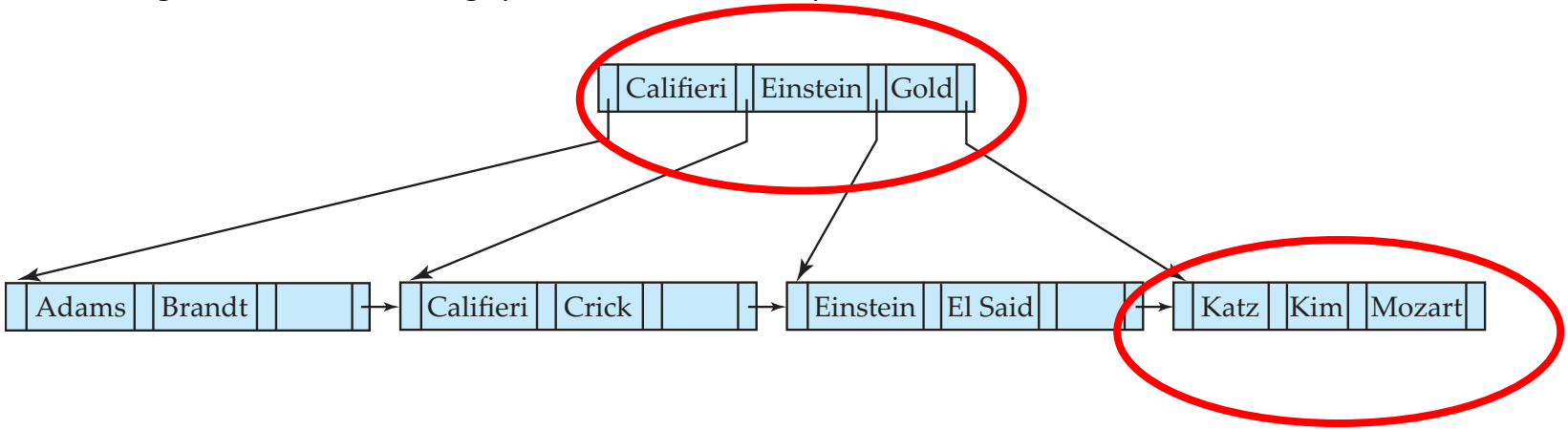


Examples of B+-Tree Deletion

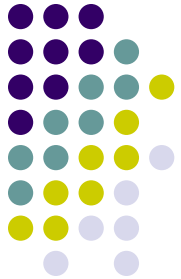
Deletion of "Gold"



- Rightmost two leaves merged into a single one: (Katz, Kim, Mozart)
- Need to remove a pointer from parent node (Kim), which also becomes underful and merged with its sibling (Califieri, Einstein) ➔ New root



B+-Trees: Summary



- Main benefit: reduce the number of “block” accesses on a disk when searching for one or few tuples
  - Consider a relation with 1,000,000 blocks (= 4GB assuming 4k blocks)
  - Sequential scan to find a tuple would take: 4GB/200MB/s = 20s
    - Assuming we are not doing any seeks
  - Using a B+-tree would take 4-5 random accesses = 40-50 ms

- Today, LSM Trees (Log-structured-merge trees) are considered a better idea in most use cases
  - More compact, and better write performance

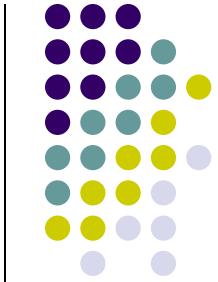
CMSC424: Database Design

Module: File Organization and Indexes

Hash Indexes; Miscellaneous

Instructor: Amol Deshpande
amol@cs.umd.edu

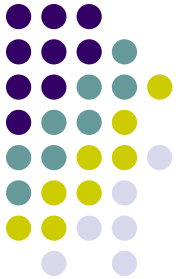
Hash Indexes



- Book Chapters
  - 11.6, 11.7 (at a high level), 11.4.1, 11.4.5, 11.5, 11.9 (briefly)

- Key topics:
  - Hash-based file organization
  - Static hashing-based indexes
  - Handling of bucket overflows
  - Log-structured Merge Trees
  - Multi-key indexes, Bitmap indexes, R-Trees

Hash-based File Organization



Store record with search key *k*
in block number *h(k)*

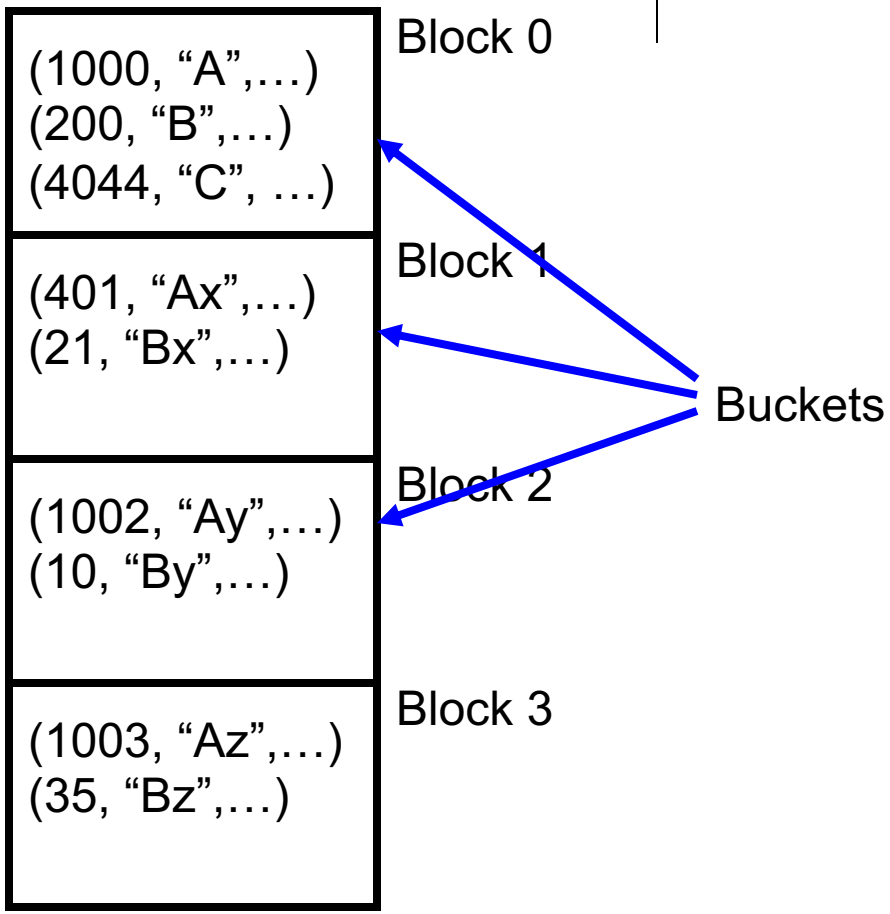
e.g. for a person file,
h(SSN) = SSN % 4

Blocks called "*buckets*"

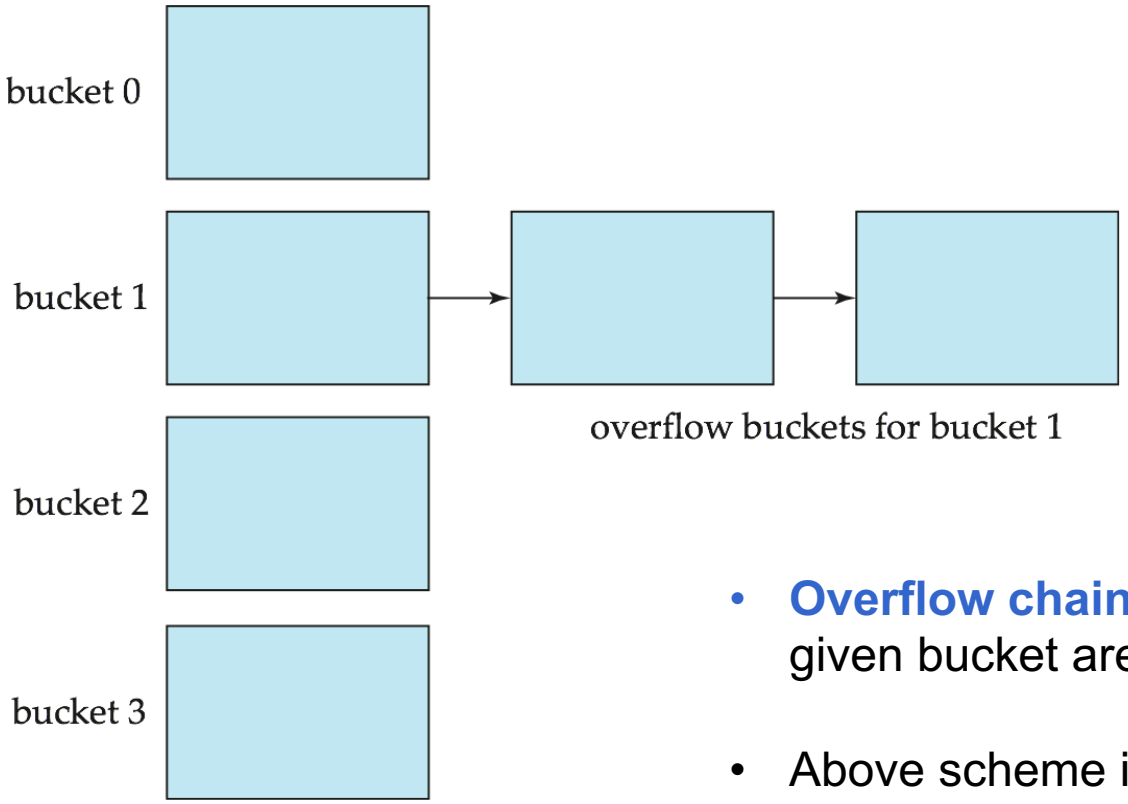
What if the block becomes full ?
Overflow pages

Uniformity property:
Don't want all tuples to map to
the same bucket
h(SSN) = SSN % 2 would be bad

Hash functions should also be **random**
Should handle different real datasets

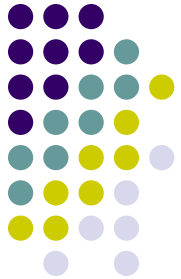


Overflow Pages



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

Hash-based File Organization



Hashed on “branch-name”

Hash function:

a = 1, b = 2, .., z = 26

h(abz)

= (1 + 2 + 26) % 10

= 9

bucket 0

bucket 1

15151	Mozart	Music	40000

bucket 2

32343	El Said	History	80000
58583	Califieri	History	60000

bucket 3

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

bucket 4

12121	Wu	Finance	90000
76543	Singh	Finance	80000

bucket 5

76766	Crick	Biology	72000

bucket 6

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

bucket 7

Hash Indexes

Extends the basic idea

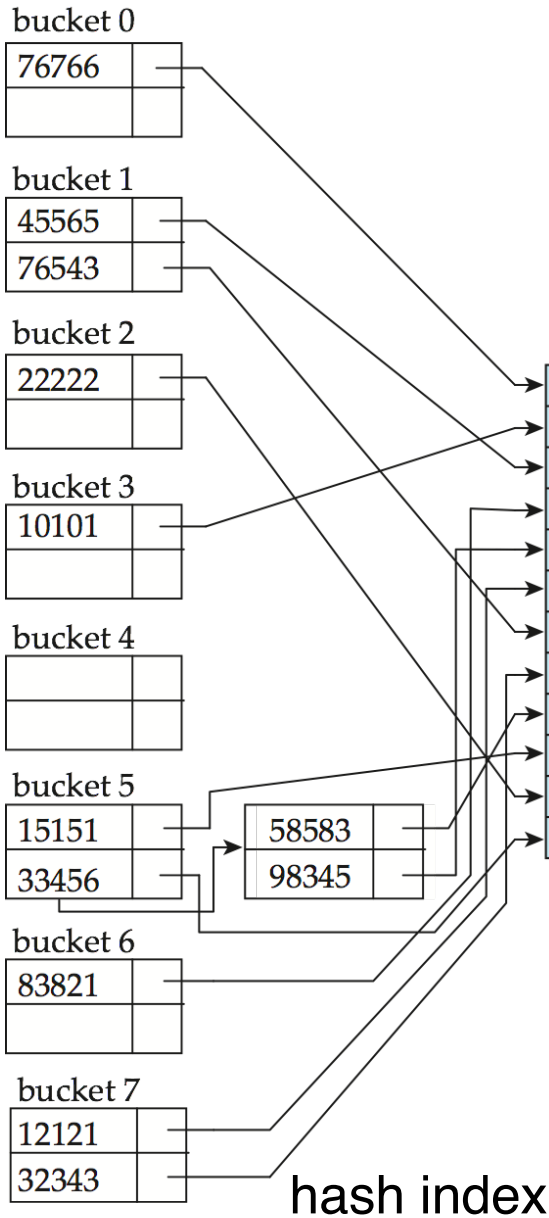
Search:

Find the block with search key

Follow the pointer

Range search ?

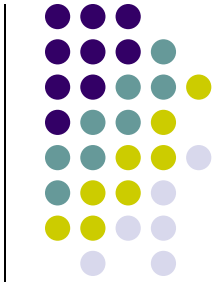
a < X < b ?



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

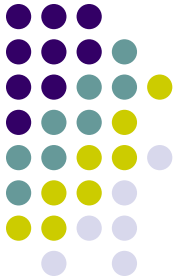
hash index on *instructor,* on attribute *ID*

Hash Indexes



- Very fast search on equality
- Can't search for "ranges" at all
  - Must scan the file
- Inserts/Deletes
  - Overflow pages can degrade the performance
  - Can do periodic reorganization (by modifying hash functions)
- A better approach is to use "dynamic hashing"
  - Allow use of a hash function that can be modified
  - e.g., Extendable Hashing, or Linear Hashing

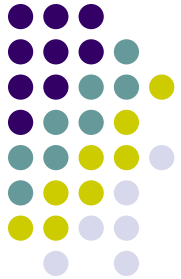
Comparison of Ordered Indexing and Hashing



- 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

- Hashing very common in distributed settings (e.g., in key-value stores)

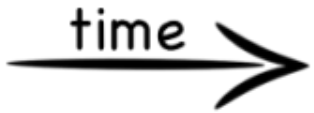
Log-structured Merge Trees



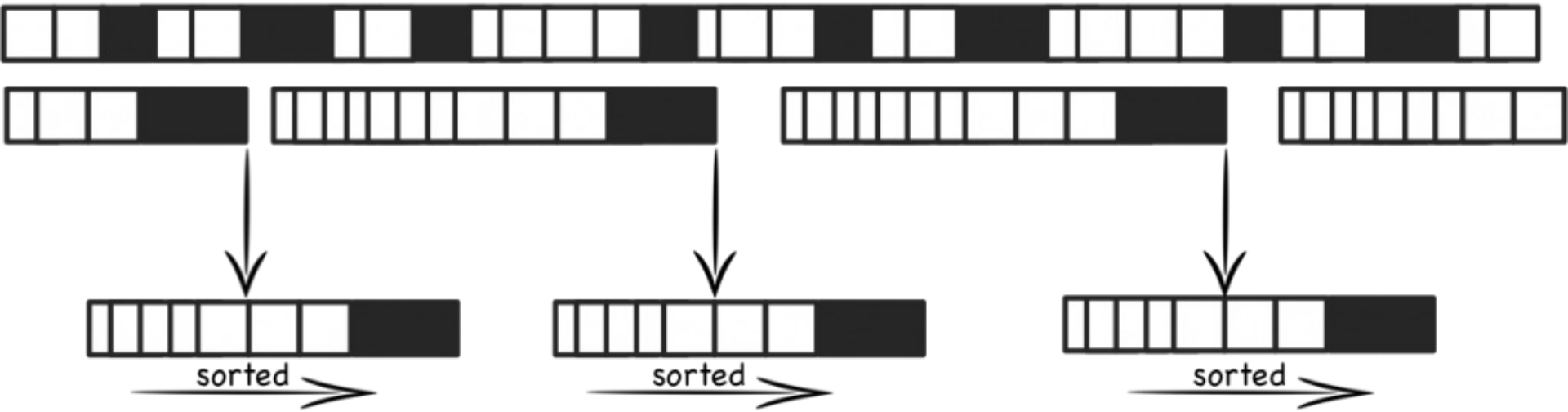
- Very widely used today, in most modern systems
  - RocksDB, Cassandra, LevelDB, InfluxDB, Bigtable, …

- Key insight:
  - B+-trees/Hash index etc., require ”in-place” random updates
    - See the reorganizations that we had to do in the earlier examples
  - Not a good idea as the gap between random and sequential increases
  - Also, not a good idea for SSDs which don’t like small writes
  - Instead
    - Keep all the data sorted in memory and build red/black tree or binary tree on it
    - As you run out of memory, write out the sorted “run” to disk and never modify it again (except see below)
    - When “searching”, you have to search all of ”runs” -- so periodically compact them to reduce the number of runs

Log-structured Merge Trees



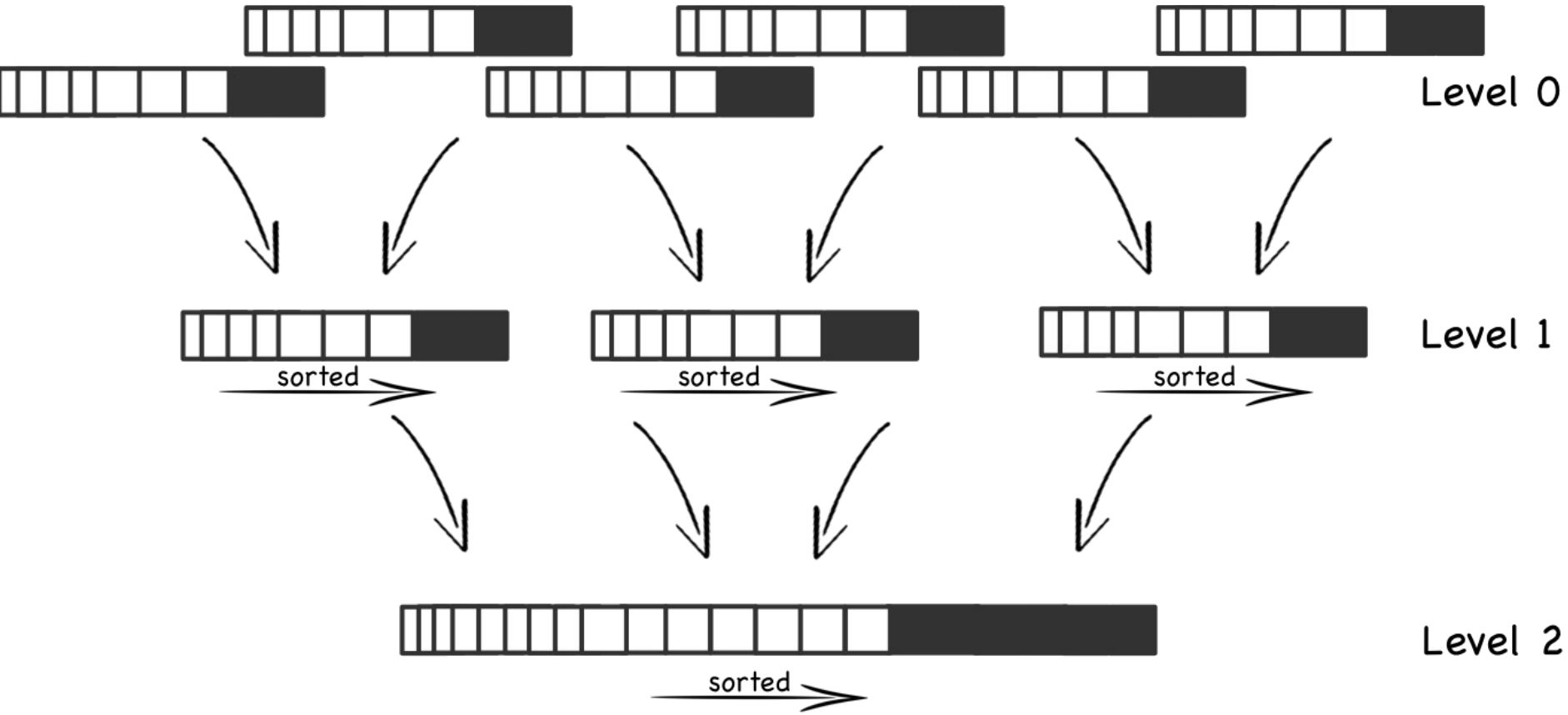
Data stream of k-v pairs ...are buffered in sorted memtables



and periodically flushed to disk...forming a set of small, sorted files.

http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

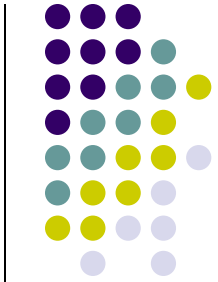
Log-structured Merge Trees



Compaction continues creating fewer, larger and larger files

http://www.benstopford.com/2015/02/14/log-structured-merge-trees/

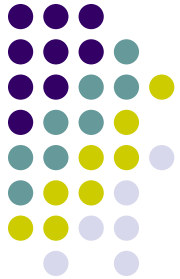
Log-structured Merge Trees



- Benefits:
  - Large sequential writes
  - Compaction is done in a batched fashion and exploits the sorted nature → very fast and sequential writes as well

- Drawbacks:
  - Searching is expensive
  - Same “key” is present in many different files
  - Can use “bloom filters” to reduce the number of files touched
    - Read up if interested

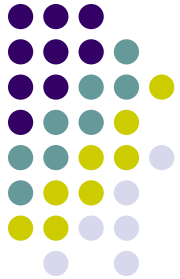
Multiple-Key Access



select *ID*
from *instructor*
where *dept_name* = “Finance” **and** *salary* = 80000

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

Indices on Multiple Keys



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

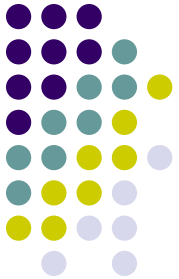
- Ideal for something like:

  where *dept_name* = “Finance” **and** *salary* = 80000
- Can also efficiently handle

  where *dept_name* = “Finance” **and** *salary* < 80000
- But cannot efficiently handle

  where *dept_name* < “Finance” **and** *balance* = 80000

Bitmap Indices



- Specialized indexes used in data warehouses
- Assume records numbered sequentially from 0
  - Given a number *n* it must be easy to retrieve record *n*
    - Particularly easy if records are of fixed size
- Best for attributes that with a small domain
  - E.g., gender, country, state, …
  - E.g., income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000- infinity)
- A bitmap is simply an array of bits

Bitmap Indices (Cont.)

- Bitmap index on an attribute has one bitmap **for each value of the attribute**
  - Bitmap has as many bits as records
  - Keeps track of whether a record has that value for the attr

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

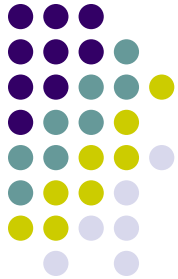
Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

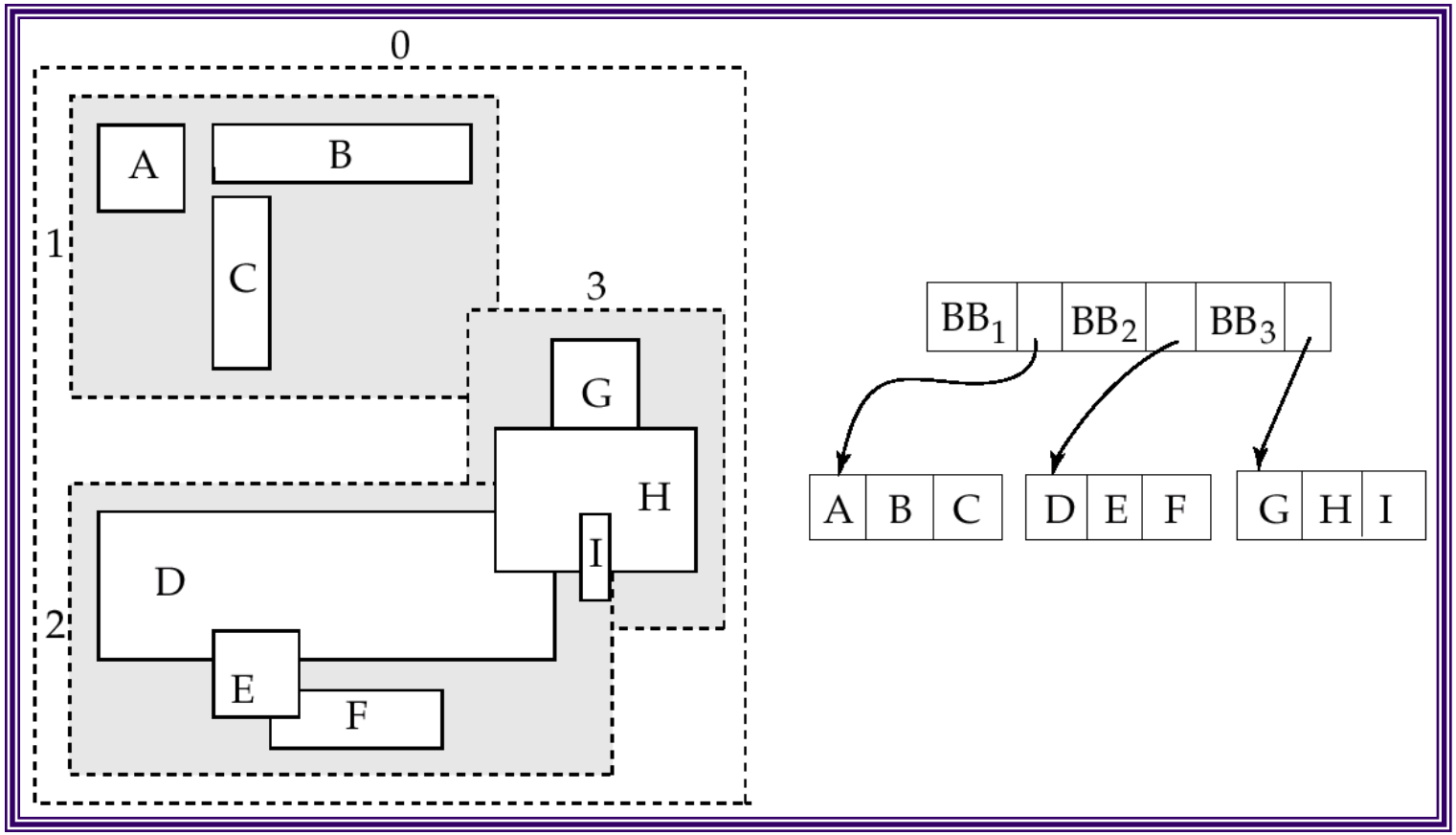
Bitmap Indices (Cont.)



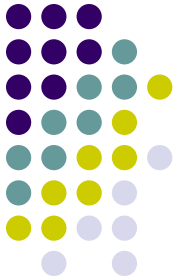
- Not particularly useful for single attribute queries
- But consider a query: gender = m and income_level = L1
  - Retrieve individual bitmaps for those two
  - Do an AND to find all records that satisfy both conditions
  - Retrieve only those records
- Can also be used for gender = m or income_level = L1
- Really useful when queries have many predicates, and relations are large (i.e., a data warehouse)
- Updating bitmap indexes is very expensive

R-Trees

For spatial data (e.g. maps, rectangles, GPS data etc)



Conclusions



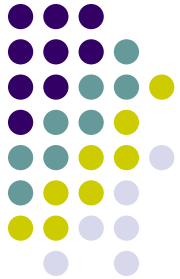
- Indexing Goal: “Quickly find the tuples that match certain conditions”
- Equality and range queries most common
  - Hence B+-Trees the predominant structure for on-disk representation
  - Hashing is used more commonly for in-memory operations
- Many many more types of indexing structures exist
  - For different types of data
  - For different types of queries
    - E.g. “nearest-neighbor” queries

CMSC424: Database Design

Module: Database Implementation

Instructor: Amol Deshpande
amol@cs.umd.edu

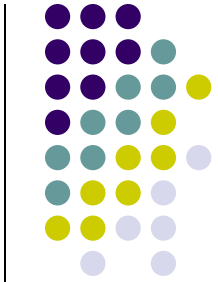
Database Implementation



- Shifting into discussing the internals of a DBMS
  - How data stored? How queries/transactions executed?

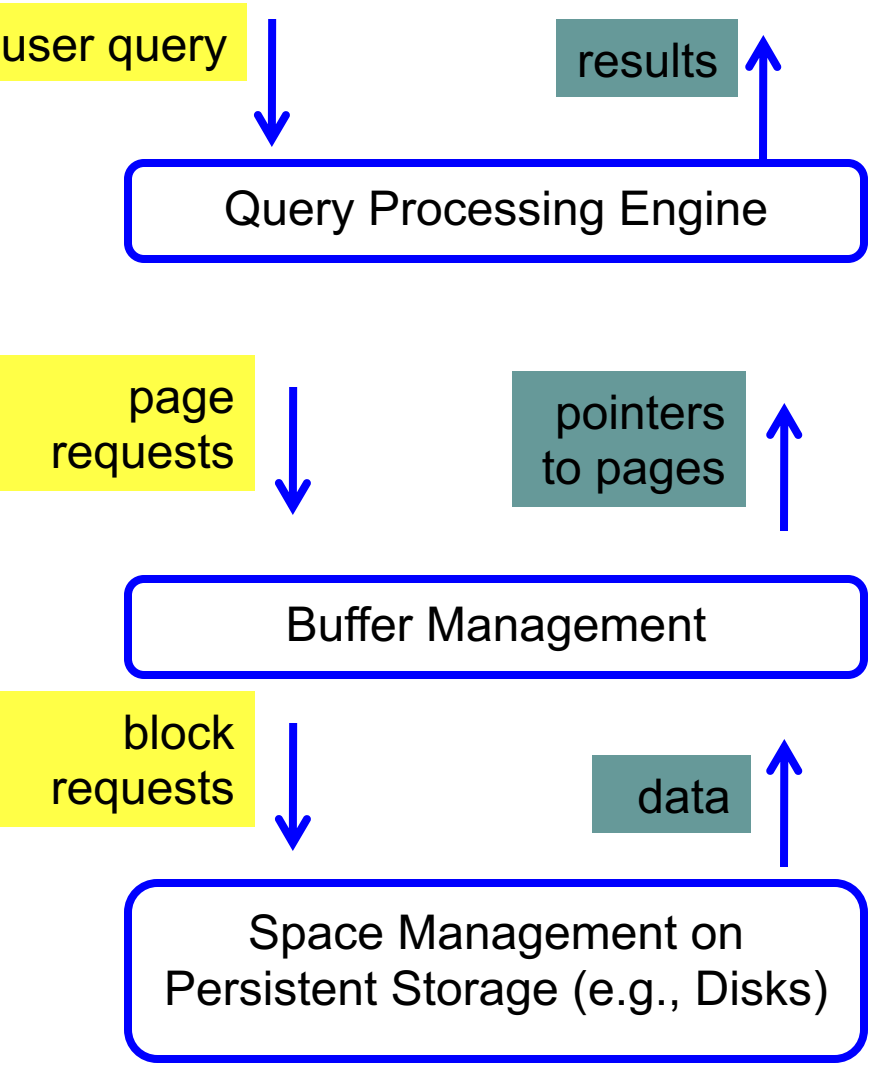
- Topics:
  - Storage: How is data stored? Important features of the storage devices (RAM, Disks, SSDs, etc)
  - File Organization: How are tuples mapped to blocks
  - Indexes: How to quickly find specific tuples of interest (e.g., all ‘friends’ of ‘user0’)
  - Query processing: How to execute different relational operations? How to combine them to execute an SQL query?
  - Query optimization: How to choose the best way to execute a query?

Overview and Cost Measures



- Book Chapters
  - 12.1, 12.2, 12.3, 12.4, 12.5, 12.6
- Key topics:
  - Main steps in Query Processing
  - How to measure the ”cost” of an operation so we can compare alternatives?
  - Different types of operations

Query Processing/Storage

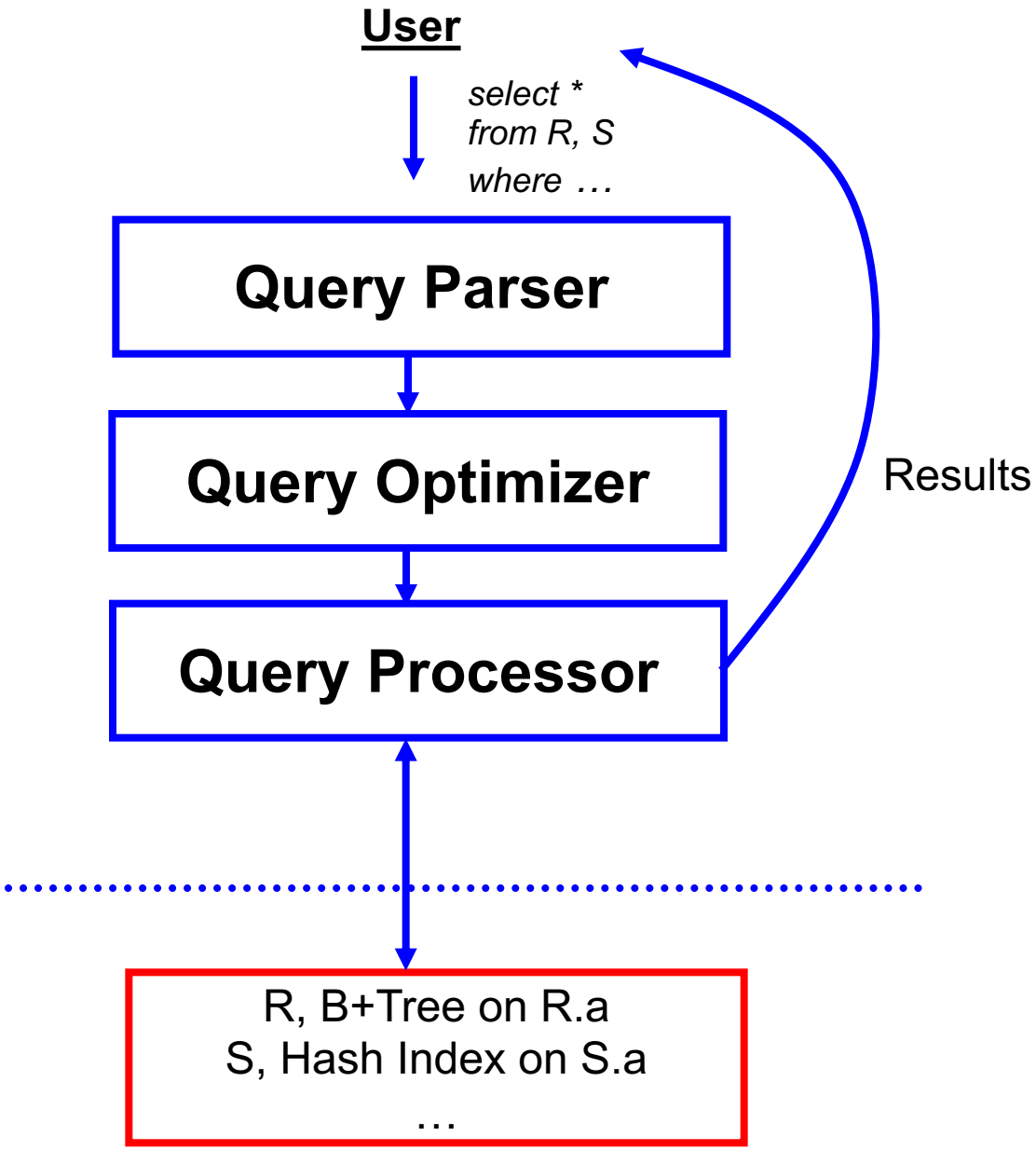


- Given an input user query, decide how to "execute" it
- Specify sequence of pages to be brought in memory
- Operate upon the tuples to produce results

- Bringing pages from disk to memory
- Managing the limited memory

- Storage hierarchy
- How are relations mapped to files?
- How are tuples mapped to disk blocks?

Getting Deeper into Query Processing

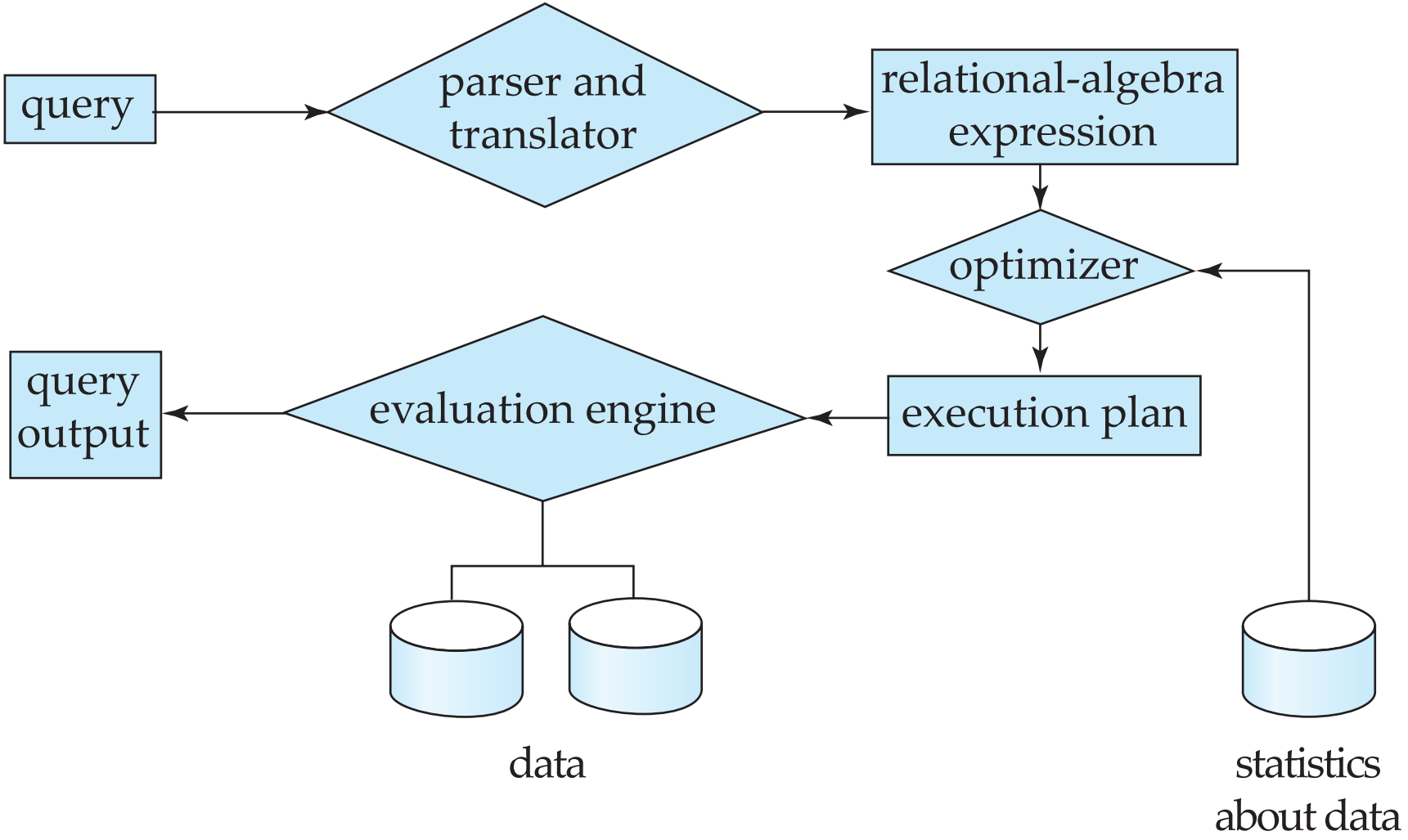


Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

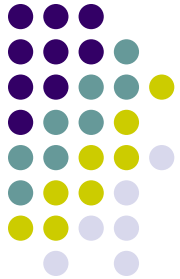
Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

Read the data from the files
Do the query processing
joins, selections, aggregates
…

Getting Deeper into Query Processing

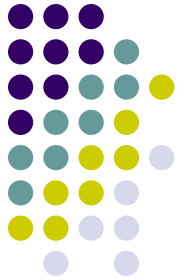


“Cost”



- Complicated to compute, but very important to decide early on
  - Need to know what you are “optimizing” for
- Many competing factors in today’s computing environment
  - CPU Instructions
  - Disk I/Os
  - Network Usage – either peak or average (for distributed settings)
  - Memory Usage
  - Cache Misses
  - … and so on
- Want to pick the one (or combination) that’s actually a bottleneck
  - No sense in optimizing for “memory usage” if you have a TB of memory and a single disk
  - Can do combinations by doing a weighted sum: e.g., 10 * Memory + 50 * Disk I/Os

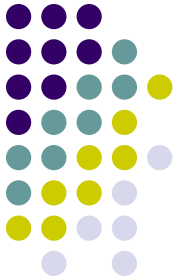
“Cost”



- We will focus on counting the amount of data processed (in terms of disk blocks), and the CPU costs if appropriate
  - Somewhat of a simplification, but works okay
  - In particular, it doesn’t account for random vs sequential access, or the tuple layouts (mostly)

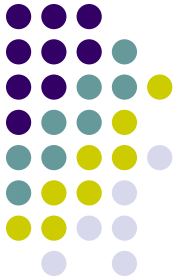
- Textbook uses a cost function that considers random vs sequential disk I/Os

Selection Operation



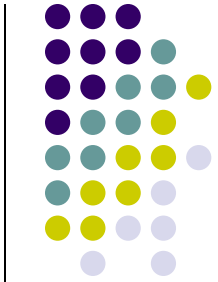
- select * from person where SSN = “123”
- Option 1: Sequential Scan
  - Read the relation start to end and look for “123”
    - Can always be used (not true for the other options)
  - Cost ?
    - *Data transferred = b_{person} = Number of relation blocks*
    - *CPU cost = O(N), if N is the number of tuples*

Selection Operation



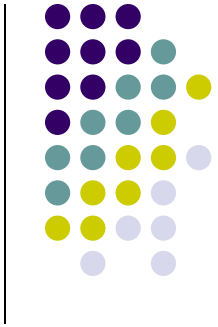
- select * from person where SSN = “123”
- Option 2 : Use a B+-tree Index
  - Pre-condition:
    - *An appropriate index must exist*
  - Cost:
    - n = the number of disk blocks containing data of interest
      - For a secondary index, the data of interest spread out all over the relation
    - h = height of the B+-tree index
    - Data transferred: h + n
    - CPU Cost: harder to calculate but comparatively lower

Selection Operation



- Selections involving ranges
  - *select * from accounts where balance > 100000*
  - *select * from matches where matchdate between '10/20/06' and '10/30/06'*
  - Option 1: Sequential scan
  - Option 2: Using an appropriate index
    - Can't use hash indexes for this purpose

Join



- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"
- *select * from R, S where |R.a – S.a | < 0.5*
  - *Not an "equi-join"*

- *Goal: For each tuple r in R, find all "matching" tuples in S (or vice versa)*

- Simplest Algorithm ("nested loops" join)

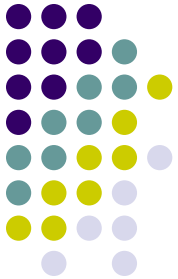
  for each tuple r in R

  for each tuple s in S

  check if r.a = s.a (or whether |r.a – s.a| < 0.5)

- Complexity too high -- O(N_r N_s)
  - e.g., imagine if |R| and |S| both in millions of tuples

Index Nested-loops Join



- *select * from R, S where R.a = S.a*
  - Called an "*equi-join*"

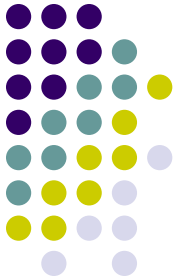
- Let's say there is an "index" on S.a

  for each tuple r in R

  use the index to find S tuples with S.a = r.a

- Works pretty well if the index on S is a "primary" index
  - Otherwise the disk I/Os may become too much

Index Nested-loops Join



- Restricted applicability
  - An appropriate index must exist
  - What about $|R.a - S.a| < 5$?

- Great for queries with joins and selections

  *select **

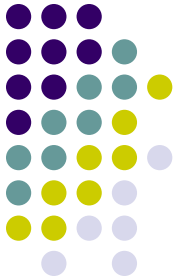
  from accounts, customers

  where accounts.customer-SSN = customers.customer-SSN and

  accounts.acct-number = "A-101"

- Only need to access one SSN from the other relation

Hash Join



- Assume S is the smaller relation and fits in memory

  read S in memory and build a hash index (hash table) on it

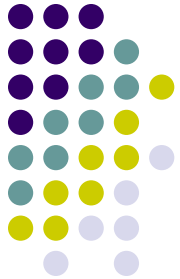
  for each tuple r in R

  use the hash index on S to find tuples such that S.a = r.a

- Cost: O(N_r + N_s + *O*) -- *where O is the number of output tuples*

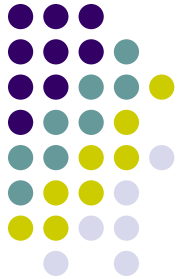
- Why good ?
  - CPU cost is much better

Hash Join



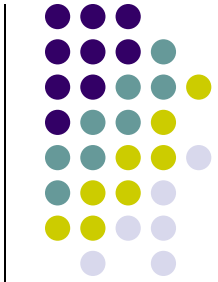
- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 1:
  - Read the relation *R* block by block and partition it using a hash function, *h1(a)*
    - Create one partition for each possible value of *h1(a)*
  - Write the partitions to disk
    - R gets partitioned into R1, R2, …, Rk
  - Similarly, read and partition S, and write partitions S1, S2, …, Sk to disk
  - Only requirement:
    - Each S partition fits in memory

Hash Join



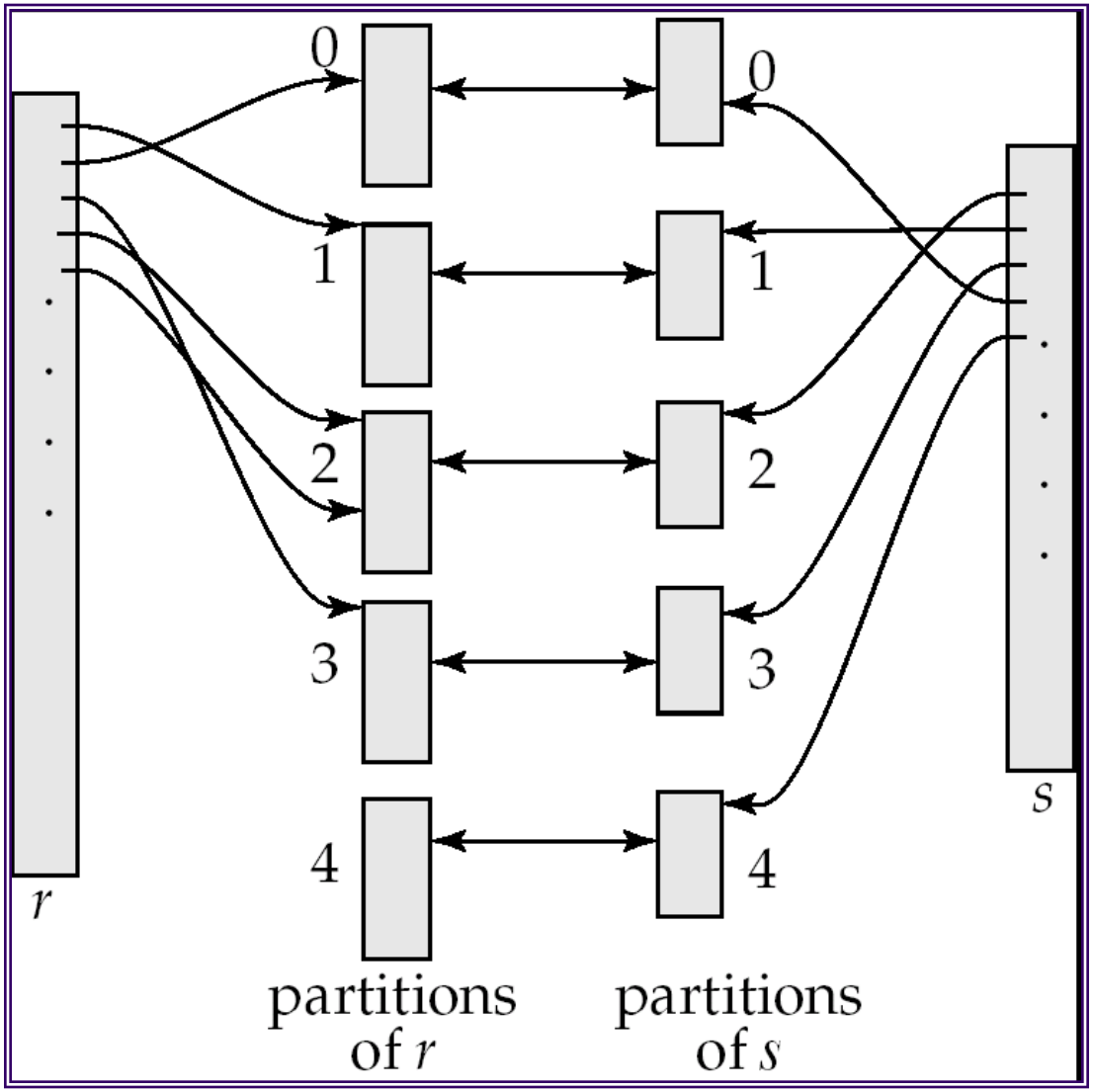
- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases"
- Phase 2:
  - Read S1 into memory, and bulid a hash index on it (S1 fits in memory)
    - Using a different hash function, $h_2(a)$
  - Read R1 block by block, and use the hash index to find matches.
  - Repeat for S2, R2, and so on.

Hash Join

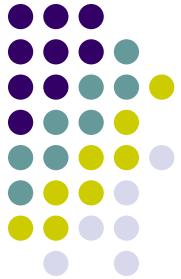


- Case 2: Smaller relation *(S)* doesn't fit in memory
- Two "phases":
- Phase 1:
  - Partition the relations using one hash function, $h_1(a)$
- Phase 2:
  - Read S_i into memory, and bulid a hash index on it (S_i fits in memory)
  - Read R_i block by block, and use the hash index to find matches.
- Cost ?
  - We read and write the two relations thrice (instead of once)

Hash Join

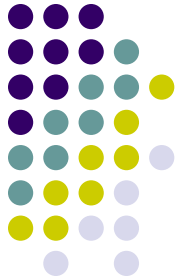


Hash Join: Issues



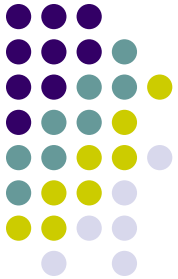
- How to guarantee that the partitions of *S* all fit in memory ?
  - Say S = 10000 blocks, Memory = M = 100 blocks
  - Use a hash function that hashes to 100 different values ?
    - Eg. *h1(a) = a % 100 ?*
  - Problem: Impossible to guarantee uniform split
    - Some partitions will be larger than 100 blocks, some will be smaller
  - Use a hash function that hashes to *100*f* different values
    - *f* is called fudge factor, typically around 1.2
    - So we may consider *h1(a) = a % 120.*
    - This is okay IF *a* is uniformly distributed
  - What if the hash function turns out to be bad ?
    - Repartition using a different hash function (at run time)

Sorting



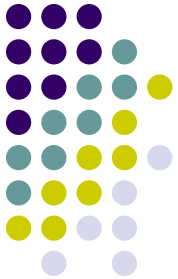
- Commonly required for many operations
  - Duplicate elimination, group by's, sort-merge join
  - Queries may have ASC or DSC in the query
- One option:
  - Read the lowest level of the index
    - May be enough in many cases
  - But if relation not sorted, this leads to too many random accesses
- If relation small enough…
  - Read in memory, use quick sort (qsort() in C)
- What if relation too large to fit in memory ?
  - External sort-merge

External sort-merge



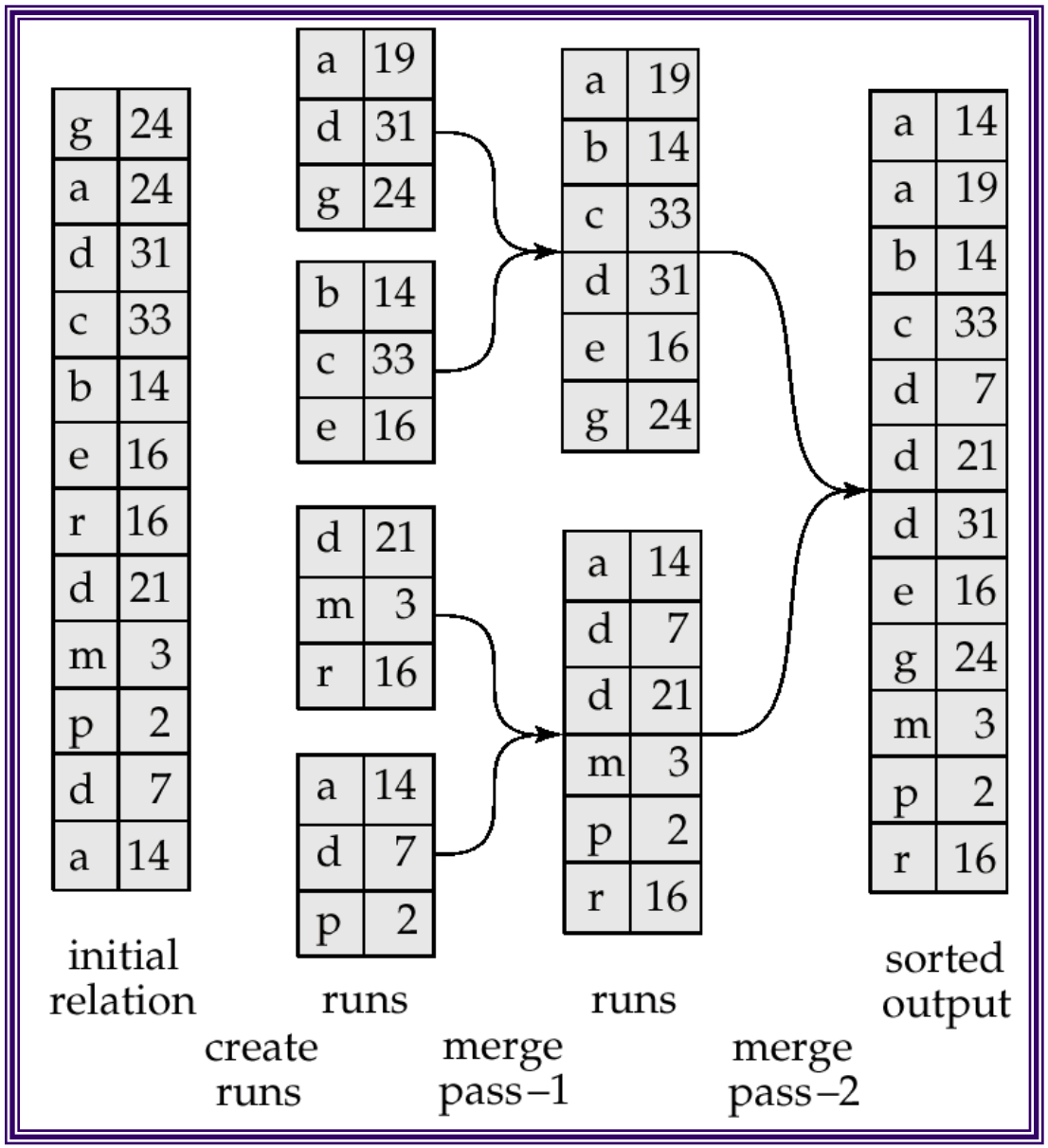
- Divide and Conquer !!
- Let *M* denote the memory size (in blocks)
- Phase 1:
  - Read first M blocks of relation, sort, and write it to disk
  - Read the next M blocks, sort, and write to disk …
  - Say we have to do this “N” times
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*

External sort-merge

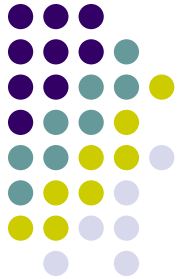


- Phase 1:
  - Create *sorted runs of size M* each
  - Result: *N* sorted runs of size *M* blocks each
- Phase 2:
  - Merge the *N* runs (*N-way merge)*
  - Can do it in one shot if *N* < *M*
- What if *N* > *M ?*
  - Do it recursively
  - Not expected to happen
  - If *M* = 1000 blocks = 4MB (assuming blocks of 4KB each)
    - Can sort: 4000MB = 4GB of data

Example: External Sorting Using Sort-Merge



Merge-Join (Sort-merge join)

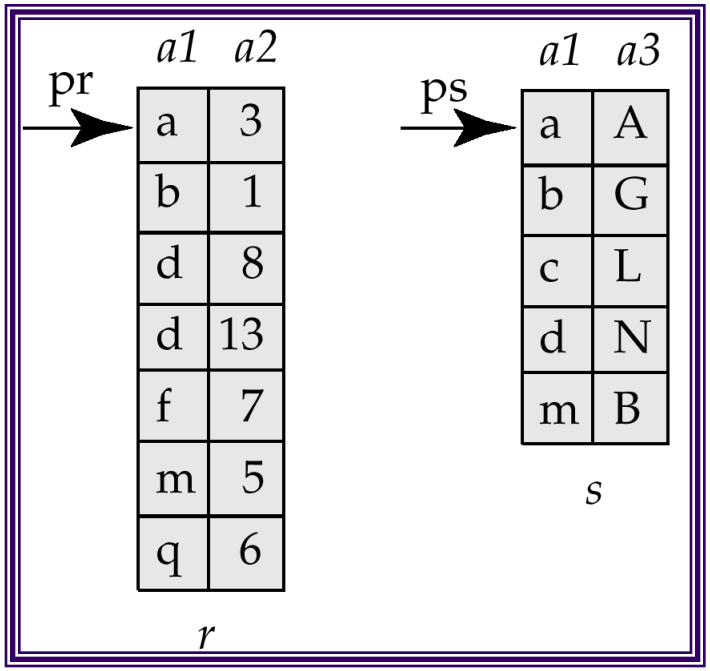


- Pre-condition:
  - The relations must be sorted by the join attribute
  - If not sorted, can sort first, and then use this algorithms
- Called “sort-merge join” sometimes

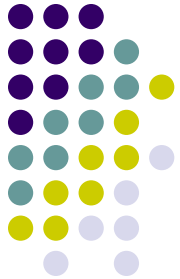
*select **
from r, s
where r.a1 = s.a1

Step:

1. *Compare the tuples at pr and ps*
2. *Move pointers down the list*
   - *Depending on the join condition*
3. *Repeat*

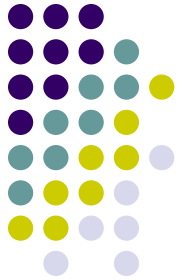


Merge-Join (Sort-merge join)



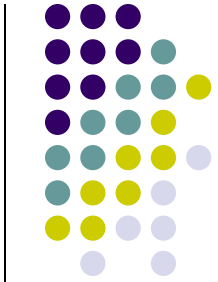
- Cost:
  - If the relations sorted, then just
    - $b_r + b_s$ block transfers, some seeks depending on memory size
  - What if not sorted ?
    - Then sort the relations first
    - In many cases, still very good performance
    - Typically comparable to hash join
- Observation:
  - The final join result will also be sorted on *a1*
  - This might make further operations easier to do
    - E.g. duplicate elimination

Joins: Summary



- Nested-loops join
  - Can always be applied irrespective of the join condition
- Index Nested-loops join
  - Only applies if an appropriate index exists
- Hash joins – only for equi-joins
  - Join algorithm of choice when the relations are large
- Sort-merge join
  - Very commonly used – especially since relations are typically sorted
  - Sorted results commonly desired at the output
    - To answer group by queries, for duplicate elimination, because of ASC/DSC

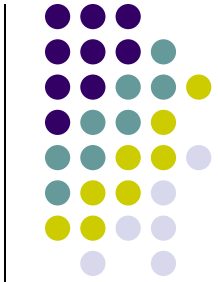
Group By and Aggregation



select a, count(b)
from R
group by a;

- Hash-based algorithm
- Steps:
  - Create a hash table on *a,* and keep the *count(b) so far*
  - Read *R* tuples one by one
  - For a new *R tuple, "r"*
    - Check if *r.a* exists in the hash table
    - If yes, increment the count
    - If not, insert a new value

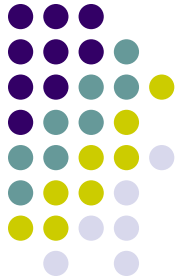
Group By and Aggregation



select a, count(b)
from R
group by a;

- Sort-based algorithm
- Steps:
  - Sort *R* on *a*
  - Now all tuples in a single group are contigous
  - Read tuples of *R (sorted)* one by one and compute the aggregates

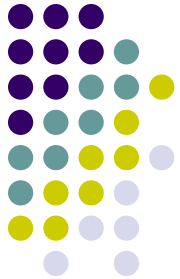
Group By and Aggregation



select a, AGGR(b) from R group by a;

- sum(), count(), min(), max(): only need to maintain one value per group
  - Called “distributive”
- average() : need to maintain the “sum” and “count” per group
  - Called “algebraic”
- stddev(): algebraic, but need to maintain some more state
- median(): can do efficiently with sort, but need two passes (called “holistic”)
  - First to find the number of tuples in each group, and then to find the median tuple in each group
- count(distinct b): must do duplicate elimination before the count

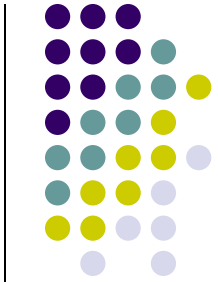
Duplicate Elimination



select distinct a
from R ;

- Best done using sorting – Can also be done using hashing
- Steps:
  - Sort the relation *R*
  - Read tuples of *R* in sorted order
  - prev = null;
  - for each tuple *r* in *R (sorted)*
    - if *r != prev* then
      - Output *r*
      - *prev = r*
    - else
      - Skip *r*

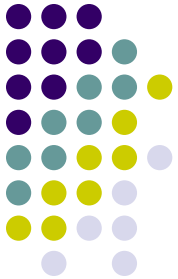
Set operations



*(select * from R) union (select * from S) ;*
*(select * from R) intersect (select * from S) ;*
*(select * from R) union all (select * from S) ;*
*(select * from R) intersect all (select * from S) ;*

- Remember the rules about duplicates
- "union all": just append the tuples of *R and S*
- *"*union": append the tuples of *R and S, and do duplicate elimination*
- *"intersection"*: similar to joins
  - Find tuples of R and S that are identical on all attributes
  - Can use *hash-based or sort-based algorithm*

Outer Joins



- Say: R FULL OUTER JOIN S, on R.a = S.a
- Need to keep track of which tuples of R "do not match" any tuples from S, and vice versa

- Hash-based, with a hash index on S:
  - For a tuple r in R, if the probe returns NULL, output r padded with NULLs
  - For each tuple *s* in *S*, maintain a Boolean variable (in the hash table) to track whether *s* was returned for any probes
  - At the end, go through the hash table, and look for *S* tuples that did not match anything

- Merge join can also be adapted in a similar way

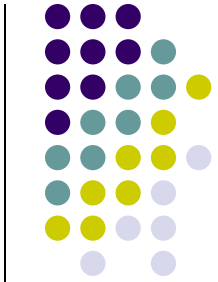
CMSC424: Database Design

Module: Query Processing

Putting it All Together

Instructor: Amol Deshpande
amol@cs.umd.edu

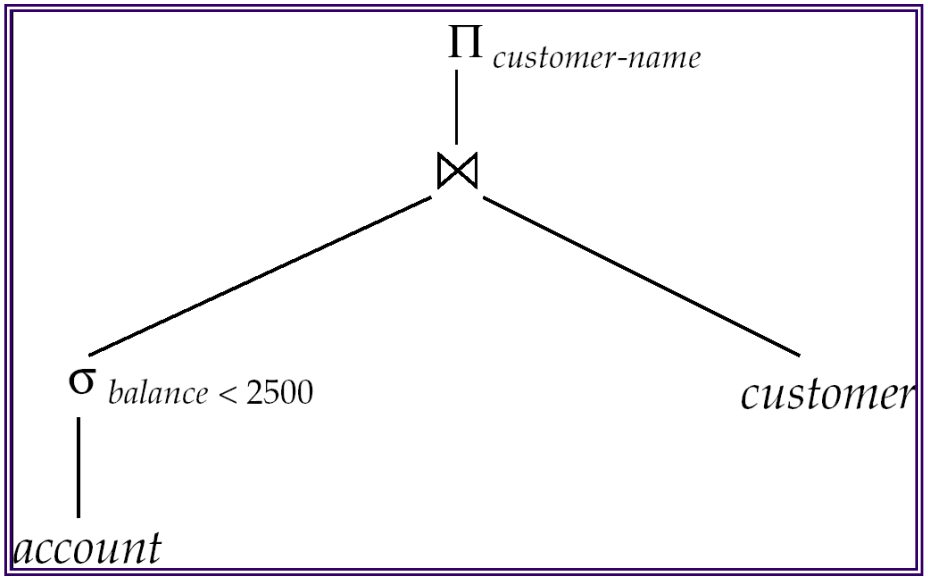
Putting it all together



- Book Chapters
  - 12.7
- Key topics:
  - How to put it all together in a query plan
  - Pipelining vs Materialization
  - Iterator Interface

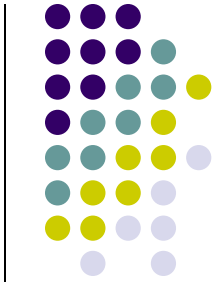
Evaluation of Expressions

select customer-name
from account a, customer c
where a.SSN = c.SSN and
a.balance < 2500



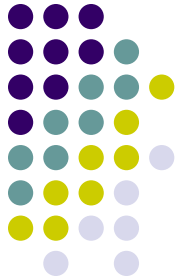
- Two options:
  - Materialization
  - Pipelining

Evaluation of Expressions



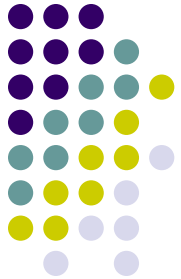
- Materialization
  - Evaluate each expression separately
    - Store its result on disk in *temporary relations*
    - Read it for next operation
- Pipelining
  - Evaluate multiple operators simultaneously
  - Skip the step of going to disk
  - Usually faster, but requires more memory
  - Also not always possible..
    - E.g. Sort-Merge Join
  - Harder to reason about

Materialization



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk, while the other is getting filled
  - Allows overlap of disk writes with computation and reduces execution time

Pipelining

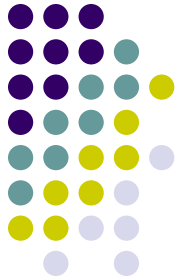


- Evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of

$$\sigma_{balance<2500}(account)$$

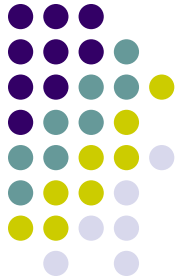
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper: no need to store a temporary relation to disk.
- Requires higher amount of memory
  - All operations are executing at the same time (say as processes)
- Somewhat limited applicability
- A "blocking" operation: An operation that has to consume entire input before it starts producing output tuples

Pipelining



- Need operators that generate output tuples while receiving tuples from their inputs
  - Selection: Usually yes.
  - Sort: NO. The sort operation is blocking
  - Sort-merge join: The final (merge) phase can be pipelined
  - Hash join: The partitioning phase is blocking; the second phase can be pipelined
  - Aggregates: Typically no. Need to wait for the entire input before producing output
    - However, there are tricks you can play here
  - Duplicate elimination: Since it requires sort, the final merge phase could be pipelined
  - Set operations: see duplicate elimination

Recap: Query Processing



- Many, many ways to implement the relational operations
  - Numerous more used in practice
  - Especially in data warehouses which handle TBs (even PBs) of data
- However, consider how complex SQL is and how much you can do
  - Compared to that, this isn't much
- Most of it is very nicely modular
  - Can plug in new operators quite easily
  - PostgreSQL query processing codebase easy to read and modify
- Having so many operators does complicate the codebase and the query optimizer though
  - But needed for performance

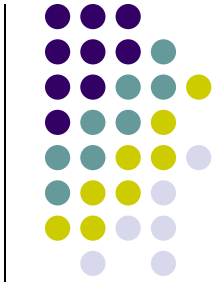
CMSC424: Database Design

Module: Query Processing

Query Optimization

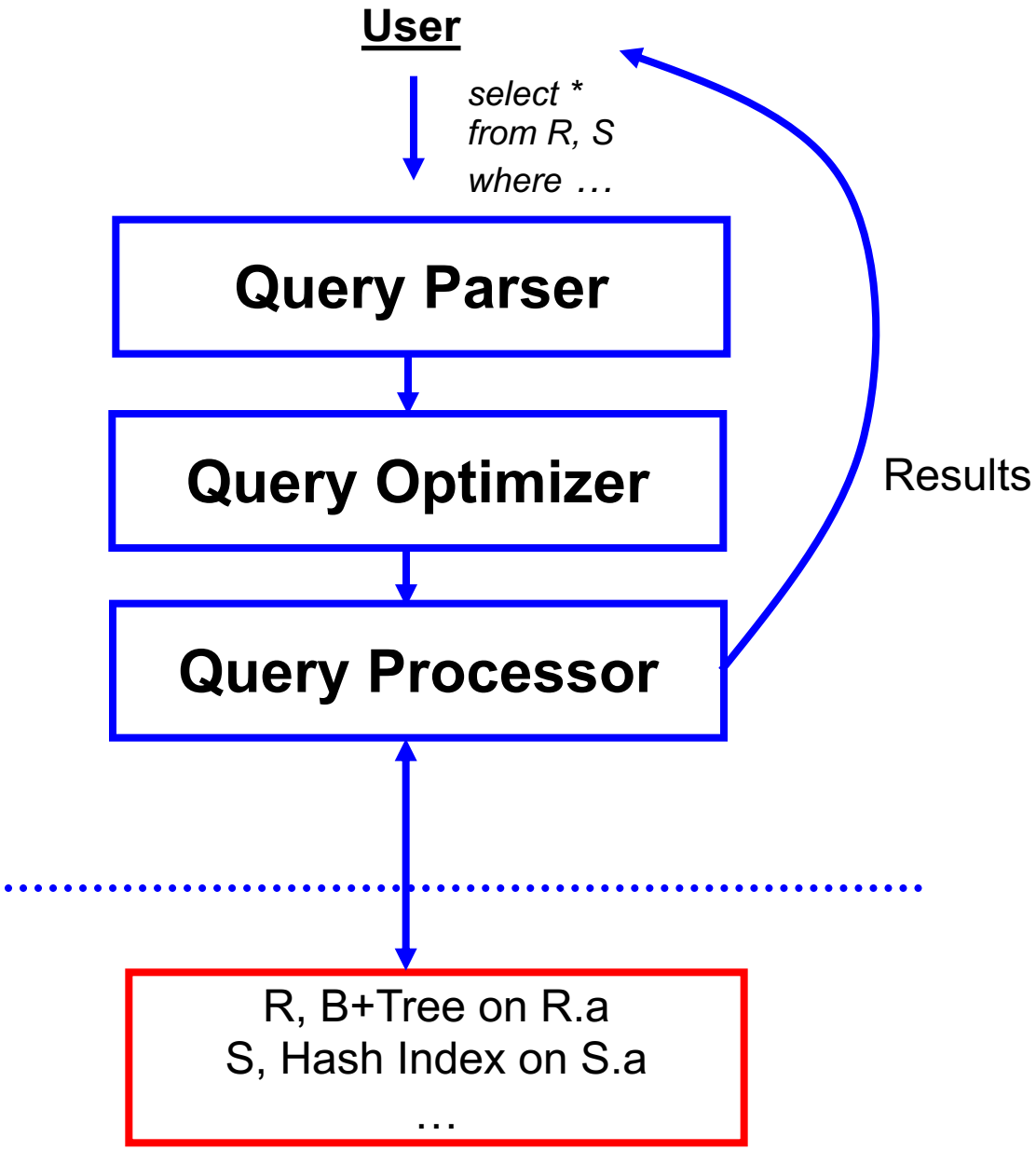
Instructor: Amol Deshpande
amol@umd.edu

Query Optimization: Overview



- Key topics:
  - Why query optimization is so important?
  - Key steps in query optimization
  - High-level concepts

Getting Deeper into Query Processing

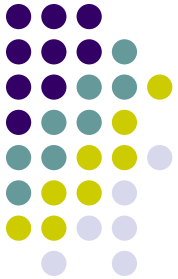


Resolve the references,
Syntax errors etc.
Converts the query to an internal format
relational algebra like

Find the *best* way to evaluate the query
Which index to use ?
What join method to use ?
…

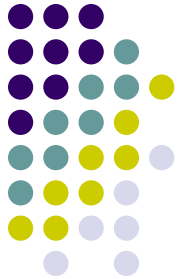
Read the data from the files
Do the query processing
joins, selections, aggregates
…

Query Optimization



- Why ?
  - Many different ways of executing a given query
  - Huge differences in cost
- Example:
  - select * from person where ssn = “123”
  - Size of *person* = 1GB
  - Sequential Scan:
    - Takes 1GB / (20MB/s) = 50s
  - Use an index on SSN (assuming one exists):
    - Approx 4 Random I/Os = 40ms

Query Optimization



- Many choices
  - Using indexes or not, which join method (hash, vs merge, vs NL)
  - What join order ?
    - Given a join query on R, S, T, should I join R with S first, or S with T first ?
- This is an optimization problem
  - Similar to say *traveling salesman problem*
  - Number of different choices is very very large
  - Step 1: Figuring out the *solution space*
  - Step 2: Finding algorithms/heuristics to search through the solution space

Query Optimization: Goal

- Find the best (or a good enough) execution plan
- Execution plans = Evaluation expressions annotated with the methods used

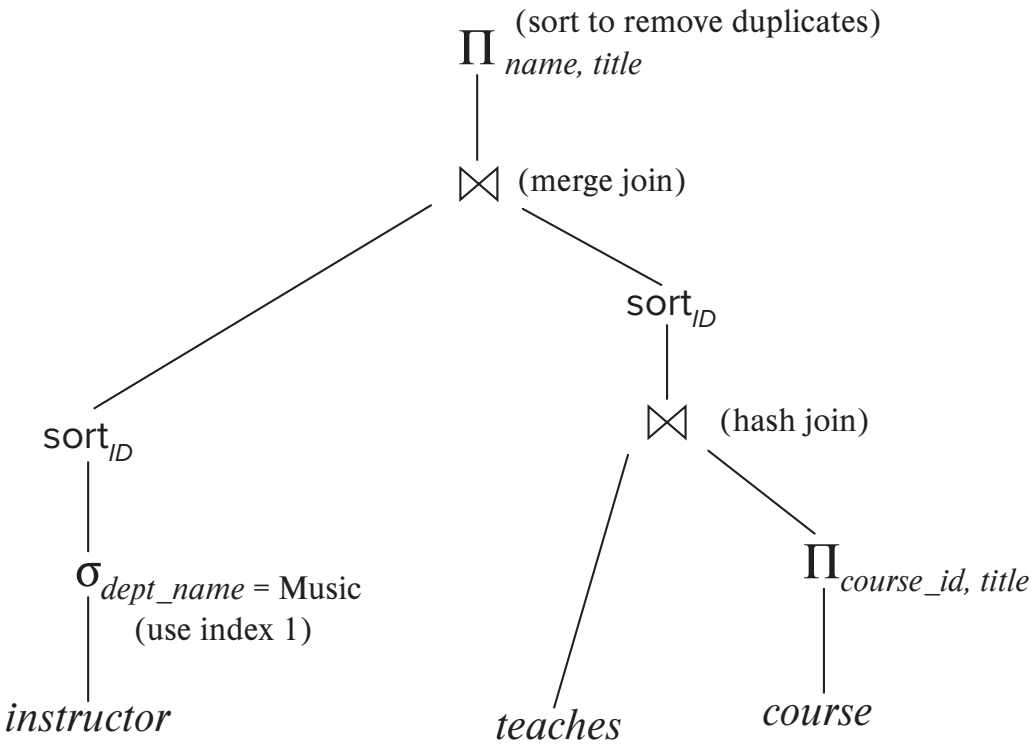
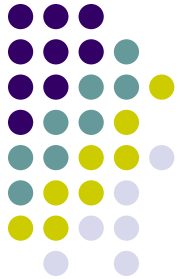


Figure 16.2 An evaluation plan.

Query Optimization

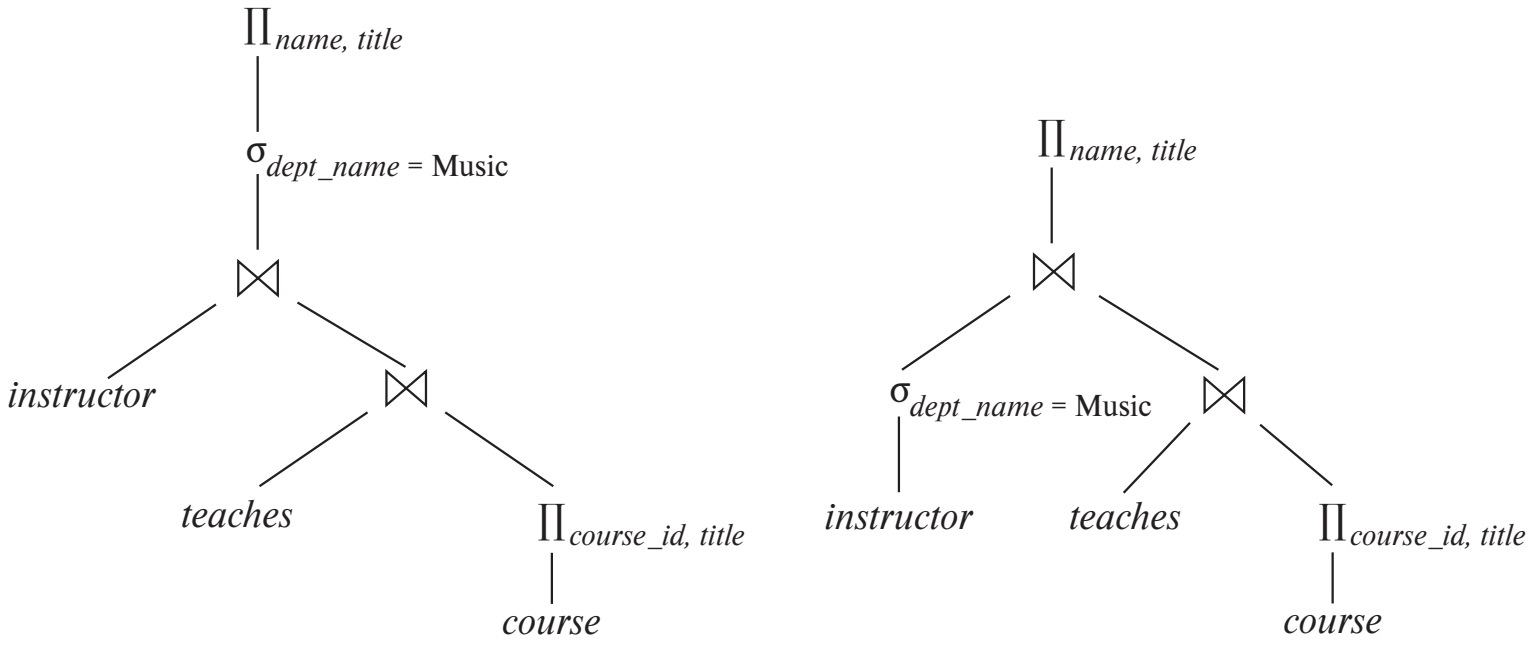


- Steps:
  - Generate all possible execution plans for the query
  - Figure out the cost for each of them
  - Choose the best

- Not done exactly as listed above
  - Too many different execution plans for that
  - Typically interleave all of these into a single efficient search algorithm

Equivalence of Expressions

- Equivalent relational expressions
  - Drawn as a tree
  - List the operations and the order

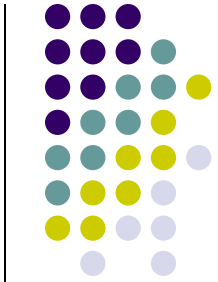


(a) Initial expression tree

(b) Transformed expression tree

Figure 16.1 Equivalent expressions.

Equivalence of Expressions



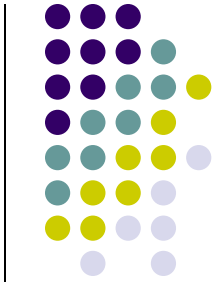
- Two relational expressions equivalent iff:
  - Their result is identical on all legal databases
- Equivalence rules:
  - Allow replacing one expression with another
- Examples:

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

2. Selections are commutative

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

Equivalence Rules



- Examples:

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

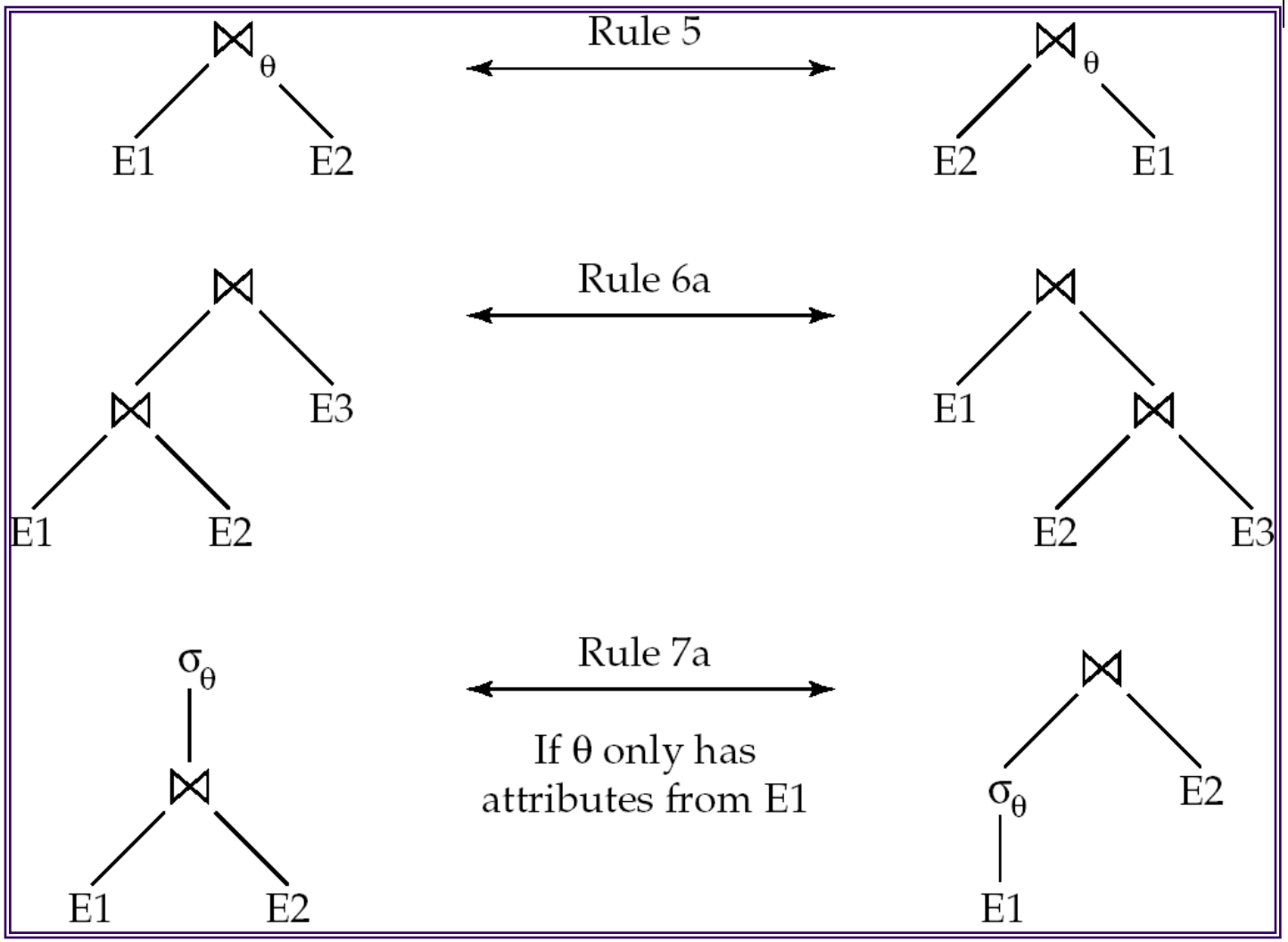
  5. $E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$

  7(a). If θ_0 *only involves attributes from* E_1

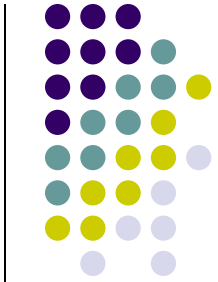
  $$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

- And so on…
  - Many rules of this type

Pictorial Depiction

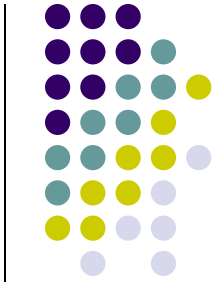


Equivalence of Expressions



- The rules give us a way to enumerate all equivalent expressions
  - Note that the expressions don't contain physical access methods, join methods etc…
- Simple Algorithm:
  - Start with the original expression
  - Apply all possible applicable rules to get a new set of expressions
  - Repeat with this new set of expressions
  - Till no new expressions are generated

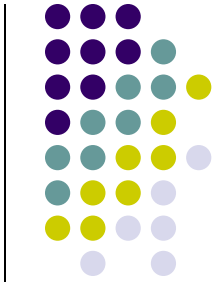
Equivalence of Expressions



- Works, but is not feasible
- Consider a simple case:
  - *R1* ⋈ *(R2* ⋈ *(R3* ⋈ *(…* ⋈ *Rn)))….)*

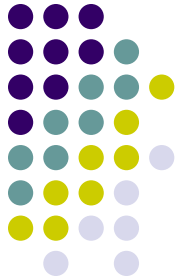
- Just join commutativity and associativity will give us:
  - At least:
    - n^2 * 2^n
  - At worst:
    - n! * 2^n
- Typically the process of enumeration is combined with the search process

Evaluation Plans



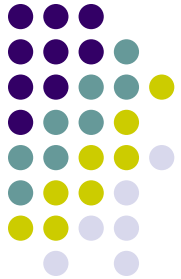
- We still need to choose the join methods etc..
  - Option 1: Choose for each operation separately
    - Usually okay, but sometimes the operators interact
    - Consider joining three relations on the same attribute:
      - $R1 \bowtie_a (R2 \bowtie_a R3)$
    - Best option for R2 join R3 might be hash-join
      - But if *R1* is sorted on *a,* then *sort-merge join* is preferable
      - Because it produces the result in sorted order by *a*
- Also, we need to decide whether to use pipelining or materialization
- Such issues are typically taken into account when doing the optimization

Query Optimization



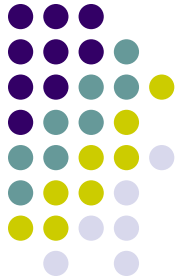
- Steps:
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples for, say, $\sigma_{balance<2500}(account)$
    - Add them !
  - Choose the best

Cost estimation



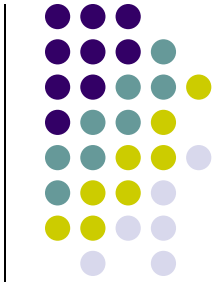
- Computing operator costs requires information like:
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
  - How many tuples match a predicate like “age > 40” ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
    - E.g. (R JOIN S) is input to another join operation – need to know if it fits in memory
  - And so on…

Cost estimation



- Some information is static and is maintained in the metadata
  - Primary key ?
  - Sorted or not, which attribute
    - So we can decide whether need to sort again
  - How many tuples in the relation, how many blocks ?
- Typically kept in some tables in the database
  - “all_tab_columns” in Oracle
- Most systems have commands for updating them

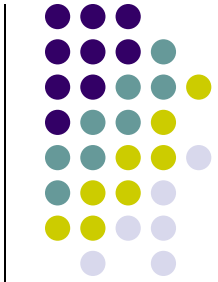
Cost estimation



- However, others need to be estimated somehow
  - How many tuples match a predicate like “age > 40” ?
    - E.g. Need to know how many index pages need to be read
  - Intermediate result sizes
- The problem variously called:
  - “intermediate result size estimation”
  - “selectivity estimation”

- Very important to estimate reasonably well
  - e.g. consider “select * from R where zipcode = 20742”
  - We estimate that there are 10 matches, and choose to use a secondary index (remember: random I/Os)
  - Turns out there are 10000 matches
  - Using a secondary index very bad idea
  - Optimizer also often choose Nested-loop joins if one relation very small… underestimation can result in very bad

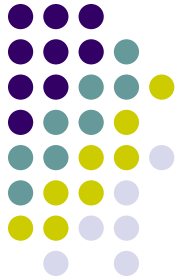
Selectivity Estimation



- Basic idea:
  - Maintain some information about the tables
    - More information → more accurate estimation
    - More information → higher storage cost, higher update cost
  - Make uniformity and randomness assumptions to fill in the gaps

- Example:
  - For a relation “people”, we keep:
    - Total number of tuples = 100,000
    - Distinct “zipcode” values that appear in it = 100
  - Given a query: “zipcode = 20742”
    - We estimated the number of matching tuples as: 100,000/100 = 1000
  - What if I wanted more accurate information ?
    - Keep better statistics/summaries…

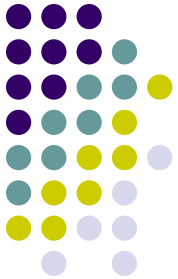
Examples



- Consider a range query: $x < R.a < y$
  - *Let Max(a, R) = maximum value of a in R*
  - *Let Min(a, R) = minimum value of a in R*
  - *Then: fraction of tuples that satisfy = (y – x) / (Max – Min)*
    - *Assuming all tuples are distributed uniformly and randomly*
    - *If y > Max or x < Min → adjust accordingly*

  - Better summary statistics (like histograms) can help with refining these estimates

Example: Joins



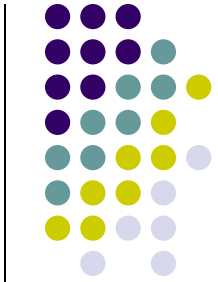
- R JOIN S: R.a = S.a
  - |R| = 10,000; |S| = 5000

- CASE 1: *a* is key for S
  - *Each tuple of R joins with exactly one tuple of S*
  - So: |R JOIN S| = |R| = 10,000
  - Assumption: Referential integrity holds

  - What if there is a selection on R or S
    - Adjust accordingly
    - Say: S.b = 100, with selectivity 0.1
    - THEN: |R JOIN S| = |R| * 0.1 = 100

- CASE 2: *a* is key for R
  - Similar

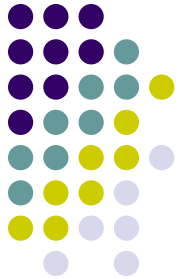
Joins



- R JOIN S: R.a = S.a
  - |R| = 10,000; |S| = 5000

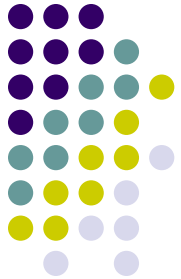
- CASE 3: *a* is not a key for either
  - Reason with the distributions on *a*
  - Say: the domain of *a: V(A, R)* = 100 (the number of distinct values *a* can take)
  - THEN, *assuming uniformity*
    - For each value of *a*
      - We have 10,000/100 = 100 tuples of R with that value of a
      - We have 5000/100 = 50 tuples of S with that value of a
      - All of these will join with each other, and produce 100 *50 = 5000
    - So total number of results in the join:
      - 5000 * 100 = 500000
  - We can improve the accuracy if we know the distributions on *a* better
    - Say using a histogram

Query Optimization



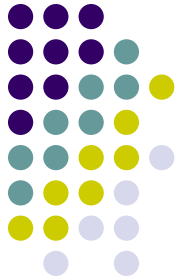
- Steps:
  - Generate all possible execution plans for the query
    - First generate all equivalent expressions
    - Then consider all annotations for the operations
  - Figure out the cost for each of them
    - Compute cost for each operation
      - Using the formulas discussed before
      - One problem: How do we know the number of result tuples for, say, $\sigma_{balance<2500}(account)$
    - Add them !
  - Choose the best

Optimization Algorithms



- Two types:
  - Exhaustive: That attempt to find the best plan
  - Heuristical: That are simpler, but are not guaranteed to find the optimal plan
- Consider a simple case
  - Join of the relations *R1, …, Rn*
  - No selections, no projections
- Still very large plan space

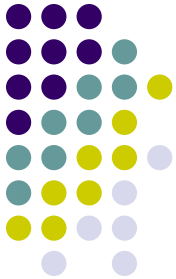
Searching for the best plan



- Option 1:
  - Enumerate all equivalent expressions for the original query expression
    - Using the rules outlined earlier
  - Estimate cost for each and choose the lowest

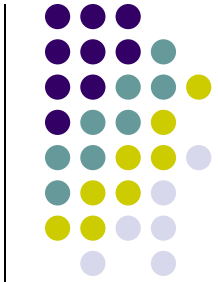
- Too expensive !
  - Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
  - There are $(2(n-1))!/(n-1)!$ different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!

Searching for the best plan



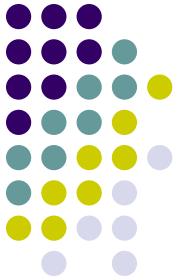
- Option 2:
  - Dynamic programming
    - There is too much commonality between the plans
    - Also, costs are additive
      - Caveat: Sort orders (also called “interesting orders”)
  - Reduces the cost down to O(n3^n) or O(n2^n) in most cases
    - Interesting orders increase this a little bit
  - Considered acceptable
    - Typically n < 10.
  - Switch to heuristic if not acceptable

Heuristic Optimization



- Dynamic programming is expensive
- Use *heuristics* to reduce the number of choices
- Typically rule-based:
  - Perform selection early (reduces the number of tuples)
  - Perform projection early (reduces the number of attributes)
  - Perform most restrictive selection and join operations before other similar operations.
- Some systems use only heuristics, others combine heuristics with partial cost-based optimization.

Summary



- Integral component of query processing
  - Why ?
- One of the most complex pieces of code in a database system
- Active area of research
  - E.g. MongoDB Query Optimization ?
  - What if you don’t know anything about the statistics
  - Better statistics
  - Etc …

CMSC424: Database Design

Module: Implementation

Transactions Summary

Instructor: Amol Deshpande
amol@umd.edu

Overview

- *Transaction*: A sequence of database actions enclosed within special tags
- Properties:
  - ***Atomicity***: Entire transaction or nothing
  - ***Consistency***: Transaction, executed completely, takes database from one consistent state to another
  - ***Isolation***: Concurrent transactions *appear* to run in isolation
  - ***Durability***: Effects of committed transactions are not lost

- Consistency: Transaction programmer needs to guarantee that
  - DBMS can do a few things, e.g., enforce constraints on the data

- Rest: DBMS guarantees

Examples of Transactions

insert into students values (…)

update instructor
set salary = salary * 1.03

enrolled = select count(*)
from takes
where (course_info) = (CMSC 424, 201, Fall 2022)
if enrolled < capacity for the room:
insert new student into takes for that course

(Modify prerequisites)
delete (CMSC422, CMSC351)
insert (CMSC422, CMSC320)

(Add a new section for a course for a given room and instructor)
if no section currently in that room:
insert a tuple into “sections” with that room
insert a tuple into “teaches”

(Remove a section)
delete from takes for that section
delete from teaches for that section
delete tuple from section

(Switch the advisor for a student)
delete old tuple with that s_id
add new tuple with that s_id and new advisor

Examples of Transactions

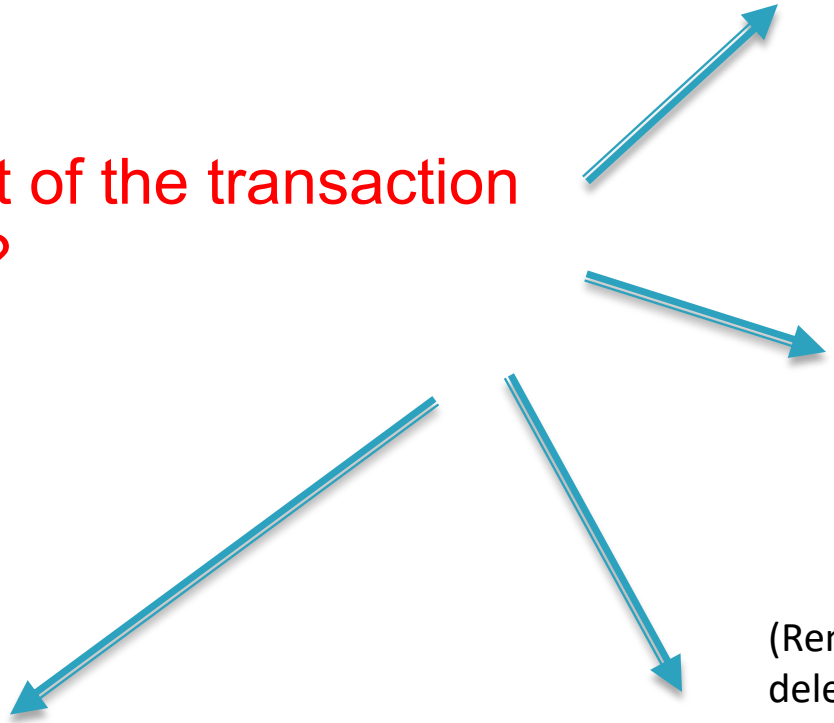
```
enrolled = select count(*)
           from takes
           where (course_info) = (CMSC 424, 201, Fall 2022)
if enrolled < capacity for the room:
    insert student A into takes for that course
```

```
enrolled = select count(*)
           from takes
           where (course_info) = (CMSC 424, 201, Fall 2022
if enrolled < capacity for the room:
    insert student B into takes for that course
```

What if two different students tried to enroll at the same time?

Concurrency Control

Examples of Transactions



update instructor
set salary = salary * 1.03

What if only part of the transaction was completed?

(Modify prerequisites)
delete (CMSC422, CMSC351)
insert (CMSC422, CMSC320)

(Remove a section)
delete from takes for that section
delete from teaches for that section
delete tuple from section

(Switch the advisor for a student)
delete old tuple with that s_id
add new tuple with that s_id and new advisor

Recovery

A Schedule

Transactions:

T1: transfers $50 from A to B

T2: transfers 10% of A to B

Database constraint: A + B is constant (*checking+saving accts)*

T1	T2
read(A)	
A = A -50	
write(A)	
read(B)	
B=B+50	
write(B)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	45
B	50	105

Each transaction obeys the constraint.

This schedule does too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B + tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Consistent.
So this schedule is okay too.

Another schedule

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
read(B)	
B=B+50	
write(B)	
	read(B)
	B = B+ tmp
	write(B)

Is this schedule okay ?

Lets look at the final effect…

Effect:	Before	After
A	100	45
B	50	105

Further, the effect same as the serial schedule 1.

Called *serializable*

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A)	
A = A -50	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	read(B)
write(A)	
read(B)	
B=B+50	
write(B)	
	B = B+ tmp
	write(B)

Effect:	Before	After
A	100	50
B	50	60

Not consistent

CMSC424: Database Design

Module: Implementation

Concurrency Control: Locking

Instructor: Amol Deshpande
amol@umd.edu

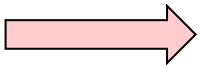
Locking - 1

- Book Chapters
  - 15.1.1-15.1.4, 15.2
- Key topics:
  - Using locking to guarantee concurrency
  - 2-Phase Locking (2PL)
  - Implementation of locking
  - Deadlocks

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
  - *Shared* (S) locks (also called *read locks)*
    - Obtained if we want to only read an item – lock-S() instruction
  - *Exclusive* (X) locks (also called *write locks)*
    - Obtained for updating a data item – lock-X() instruction

T1	T2
read(B)	read(A)
B ←B-50	read(B)
write(B)	display(A+B)
read(A)	
A ←A + 50	
write(A)	



T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
B ←B-50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
lock-X(A)	unlock(B)
read(A)	display(A+B)
A ←A + 50	
write(A)	
unlock(A)	

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
  - Depends

T2 lock type	T1 lock type	Should allow ?
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible,* grant the lock, otherwise T1 waits in a *queue.*

Lock-based Protocols

- How do we actually use this to guarantee serializability/recoverability ?
  - ★ Not enough just to take locks when you need to read/write something

T1

```
lock-X(B)
read(B)
B ←B-50
write(B)
unlock(B)
```

```
lock-X(A), lock-X(B)
TMP = (A + B) * 0.1
A = A - TMP
B = B + TMP
unlock(A), unlock(B)
```

```
lock-X(A)
read(A)
A ←A + 50
write(A)
unlock(A)
```

NOT SERIALIZABLE

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking phase
  - Transaction may only release locks

- Can be shown that this achieves *serializability*
  - *lock-point*: the time at which a transaction acquired last lock
  - if *lock-point*(T1) < *lock-point*(T2), there is no way for T1 to read any update of T2

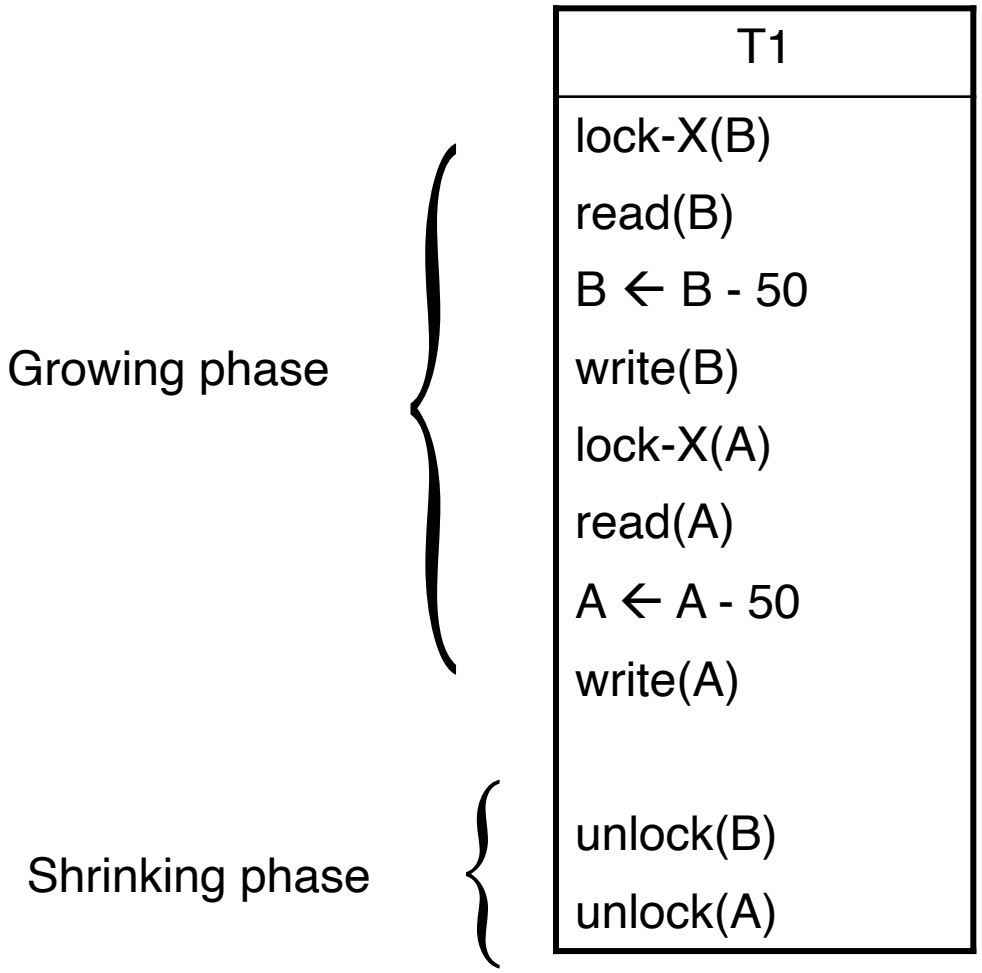
T1

lock-X(B)
read(B)
B ←B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ←A + 50
write(A)
unlock(A)

2 Phase Locking

- Example: T1 in 2PL



2 Phase Locking

- Guarantees *serializability*, but not “cascade-less recoverability”

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B)		
	lock-X(A) read(A) write(A) unlock(A)	
	Commit	lock-S(A) read(A) Commit
<xction fails>		

2 Phase Locking

- Guarantees *serializability*, but not "cascade-less recoverability"

- Guaranteeing just recoverability:
  - If T2 reads a dirty data of T1 (ie, T1 has not committed), then T2 can't commit unless T1 either commits or aborts
  - If T1 commits, T2 can proceed with committing
  - If T1 aborts, T2 must abort
    - So cascades still happen

Strict 2PL

- Release *exclusive* locks only at the very end, just after commit or abort

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B)		
	lock-X(A) read(A) write(A) unlock(A)	
	Commit	lock-S(A) read(A) Commit
<xction fails>		

Strict 2PL will not allow that

Works. Guarantees cascade-less and recoverable schedules.

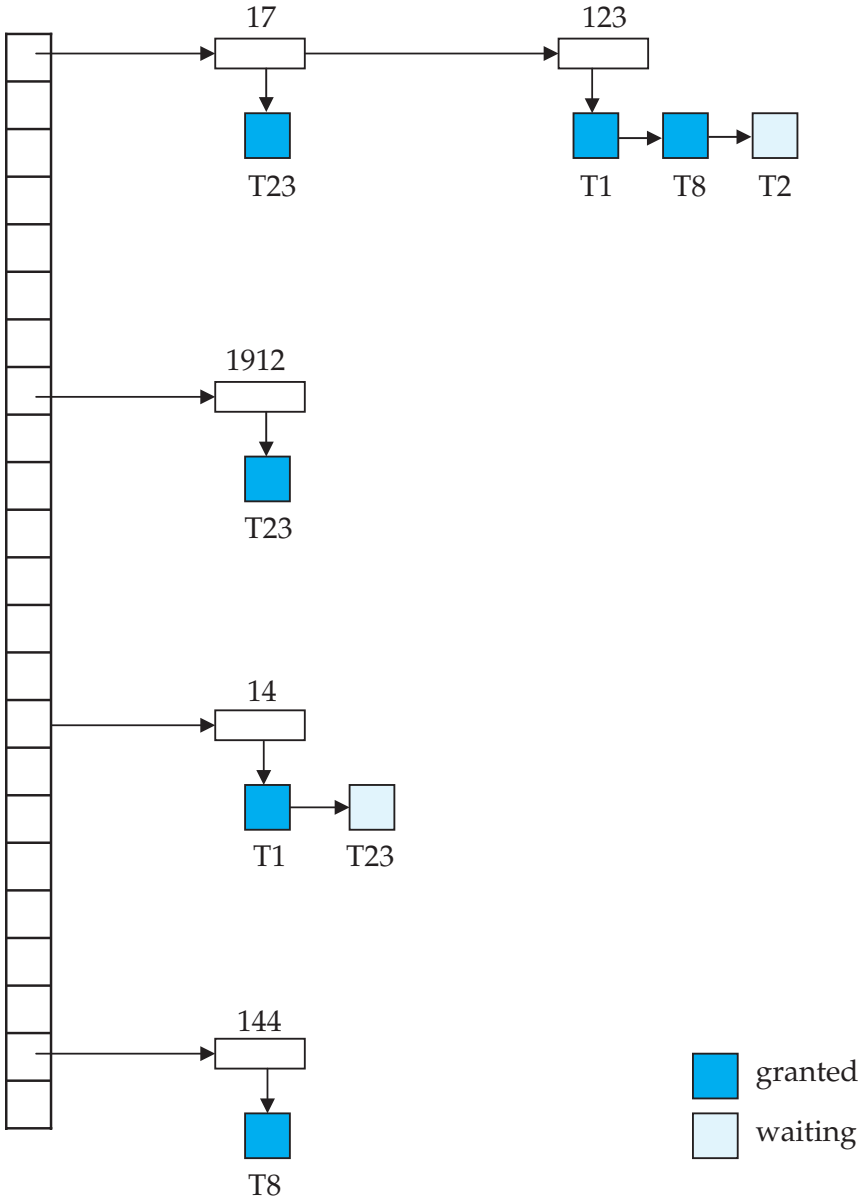
Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are not important
- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
  - The serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system
- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn't change any of the other properties of the protocol

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks

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

More Locking Issues: Deadlocks

- No xction proceeds:

Deadlock

- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

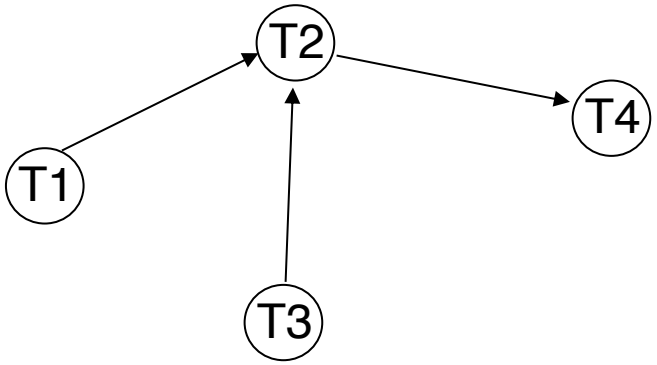
Rollback transactions
Can be costly...

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- 2PL does not prevent deadlock
  - ★ Strict doesn't either

Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen
- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - Ti waiting for Tj



T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	

Suppose T4 requests lock-S(Z)....

Dealing with Deadlocks

- Deadlock detected, now what ?
  - Will need to abort some transaction
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continueing

Preventing deadlocks

- Solution 1: A transaction must acquire all locks before it begins
  - Not acceptable in most cases
  - Still need some way to deal with deadlocks during lock acquisition

- Solution 2: A transaction must acquire locks in a particular order over the data items
  - Also called *graph-based protocols*
  - The particular order used doesn't matter (e.g., based on the value of some unique attribute)
  - Guarantees that there can never be a cycle in the precedence graph

Preventing deadlocks

- Solution 3: Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
    - In the precedence graph, there can be an edge from old transaction to a new transaction, but never the other way
    - So there cannot be a cycle in precedence graph
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.
    - Similar to above: edges only from newer transactions to older transactions
  - May abort more transactions that needed

- Solution 4: Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn't get it by then
  - As above, may lead to unnecessary restarts, but very simple to implement

CMSC424: Database Design

Module: Transactions and ACID Properties

Concurrency Control: Other Schemes

Instructor: Amol Deshpande
amol@umd.edu

1. Time-stamp Based

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - So if T1 entered before T2, then T1 should be before T2 in the serializability order
  - Say *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

1. Time-stamp Based

- Maintain for each data Q, two timestamps:
  - W-timestamp(Q): largest time-stamp of any transaction that executed Write(Q) successfully
  - R-timestamp(Q): largest time-stamp of any transaction that executed Read(Q) successfully

- Suppose Ti wants to read(Q):
  - If TS(T_i) < W-Timestamp(Q): Reject the operation and roll back T_i
  - Otherwise, allow the operation and modify:
    - R-timestamp(Q) = max(R-timestamp(Q)), TS(T_i))

1. Time-stamp Based

- Maintain for each data Q, two timestamps:
  - W-timestamp(Q): largest time-stamp of any transaction that executed Write(Q) successfully
  - R-timestamp(Q): largest time-stamp of any transaction that executed Read(Q) successfully

- Suppose Ti wants to write(Q):
  1. If TS(T_i) < R-timestamp(Q): reject the write and roll back T_i
  2. If TS(T_i) < W-timestamp(Q), then T_i is attempting to write an obsolete value of Q.
     - Hence, this **write** operation is rejected, and T_i is rolled back.
  3. Otherwise, execute **write**, and W-timestamp(Q) is set to TS(T_i).

1. Example of Schedule Under TSO

- Is this schedule valid under TSO?

  Assume that initially:
  R-TS(A) = W-TS(A) = 0
  R-TS(B) = W-TS(B) = 0
  Assume TS(T_{25}) = 25 and
  TS(T_{26}) = 26

T_{25}	T_{26}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

- How about this one, where initially
  R-TS(Q)=W-TS(Q)=0

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

1. Another Example

- Example

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

2. Optimistic Concurrency Control

- Optimistic concurrency control
  - Also called validation-based
  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart
  - Optimistic: The hope is that there won't be too many problems/aborts

2. Optimistic Concurrency Control

- Each transaction T_i has 3 timestamps
  - Start(T_i) : the time when T_i started its execution
  - Validation(T_i): the time when T_i entered its validation phase
  - Finish(T_i) : the time when T_i finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency.
  - Thus TS(T_i) is given the value of Validation(T_i).
- This protocol is useful and gives greater degree of concurrency if probability of conflicts is low.
  - because the serializability order is not pre-decided, and
  - relatively few transactions will have to be rolled back.

2. Optimistic Concurrency Control

- If for all T_i with TS (T_i) < TS (T_j) either one of the following condition holds:
  - **finish**(T_i) < **start**(T_j)
  - **start**(T_j) < **finish**(T_i) < **validation**(T_j) **and** the set of data items written by T_i does not intersect with the set of data items read by T_j.

  then validation succeeds and T_j can be committed. Otherwise, validation fails and T_j is aborted.
- *Justification*: Either the first condition is satisfied, and there is no overlapped execution, or the second condition is satisfied and
  - the writes of T_j do not affect reads of T_i since they occur after T_i has finished its reads.
  - the writes of T_i do not affect reads of T_j since T_j does not read any item written by T_i.

2. Optimistic Concurrency Control

- Example of schedule produced using validation

T_{25}	T_{26}
read (B)	
	read (B)
	$B := B$ 50
	read (A)
	$A := A + 50$
read (A)	
⟨ *validate* ⟩	
display ($A + B$)	
	⟨ *validate* ⟩
	write (B)
	write (A)

3. Snapshot Isolation

- Very popular scheme, used as the primary scheme by many systems including Oracle, PostgreSQL etc…
  - Several others support this in addition to locking-based protocol
- A type of “multi-version concurrency control”
  - Also similar to optimistic concurrency control in many ways
- Key idea:
  - For each object, maintain past “versions” of the data along with timestamps
    - Every update to an object causes a new version to be generated

3. Snapshot Isolation

- Read queries:
  - Let “t” be the “time-stamp” of the query, i.e., the time at which it entered the system
  - When the query asks for a data item, provide a version of the data item that was latest as of “t”
    - Even if the data changed in between, provide an old version
  - No locks needed, no waiting for any other transactions or queries
  - The query executes on a consistent snapshot of the database
- Update queries (transactions):
  - Reads processed as above on a snapshot
  - Writes are done in private storage
  - At commit time, for each object that was written, check if some other transaction updated the data item since this transaction started
    - If yes, then abort and restart
    - If no, make all the writes public simultaneously (by making new versions)

3. Snapshot Isolation

- A transaction T1 executing with Snapshot Isolation
  - takes snapshot of committed data at start
  - always reads/modifies data in its own snapshot
  - updates of concurrent transactions are not visible to T1
  - writes of T1 complete when it commits
  - **First-committer-wins rule**:
    - Commits only if no other concurrent transaction has already written data that T1 intends to write.

T1	T2	T3
W(Y := 1) Commit		
	Start R(X) → 0 R(Y)→ 1	
		W(X:=2) W(Z:=3) Commit
	R(Z) → 0 R(Y) → 1 W(X:=3) Commit-Req Abort	

Concurrent updates not visible
Own updates are visible
Not first-committer of X
Serialization error, T2 is rolled back

3. Snapshot Isolation

- Advantages:
  - Read query don't block at all, and run very fast
  - As long as conflicts are rare, update transactions don't abort either
  - Overall better performance than locking-based protocols

- Major disadvantage:
  - Not serializable
  - Inconsistencies may be introduced
  - See the wikipedia article for more details and an example
    - http://en.wikipedia.org/wiki/Snapshot_isolation

3. Snapshot Isolation

- Example of problem with SI
  - T1: x:=y
  - T2: y:= x
  - Initially x = 3 and y = 17
    - Serial execution: x = ??, y = ??
    - if both transactions start at the same time, with snapshot isolation: x = ?? , y = ??
- Called **skew write**
- Skew also occurs with inserts
  - E.g:
    - Find max order number among all orders
    - Create a new order with order number = previous max + 1

3. SI In Oracle and PostgreSQL

- **Warning**: SI used when isolation level is set to serializable, by Oracle, and PostgreSQL versions prior to 9.1
  - PostgreSQL's implementation of SI (versions prior to 9.1) described in Section 26.4.1.3
  - Oracle implements "first updater wins" rule (variant of "first committer wins")
    - concurrent writer check is done at time of write, not at commit time
    - Allows transactions to be rolled back earlier
    - Oracle and PostgreSQL < 9.1 do not support true serializable execution
  - PostgreSQL 9.1 introduced new protocol called "Serializable Snapshot Isolation" (SSI)
    - Which guarantees true serializabilty including handling predicate reads (coming up)

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Overview; Terminology; Steal and Force

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Recovery

- Book Chapters
  - 16.1, 16.2, 16.3.2
- Key topics:
  - Challenges in guaranteeing Atomicity and Durability
  - Basics of how disks and memory interact
  - New operations: Output() and Input()
  - STEAL and NO FORCE: Why those are desirable
  - Terminology used in the book: Immediate vs Deferred Modifications

Context

- ACID properties:
  - We have talked about Isolation and Consistency
  - How do we guarantee Atomicity and Durability ?
    - Atomicity: Two problems
      - Part of the transaction is done, but we want to cancel it
        - ABORT/ROLLBACK
      - System crashes during the transaction. Some changes made it to the disk, some didn't.
    - Durability:
      - Transaction finished. User notified. But changes not sent to disk yet (for performance reasons). System crashed.
- Essentially similar solutions

Reasons for crashes

- Transaction failures
  - **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
  - Power failures, operating system bugs etc
  - **Fail-stop assumption**: non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data
- Disk failure
  - Head crashes; ***for now we will assume***
    - ***STABLE STORAGE:* Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data**

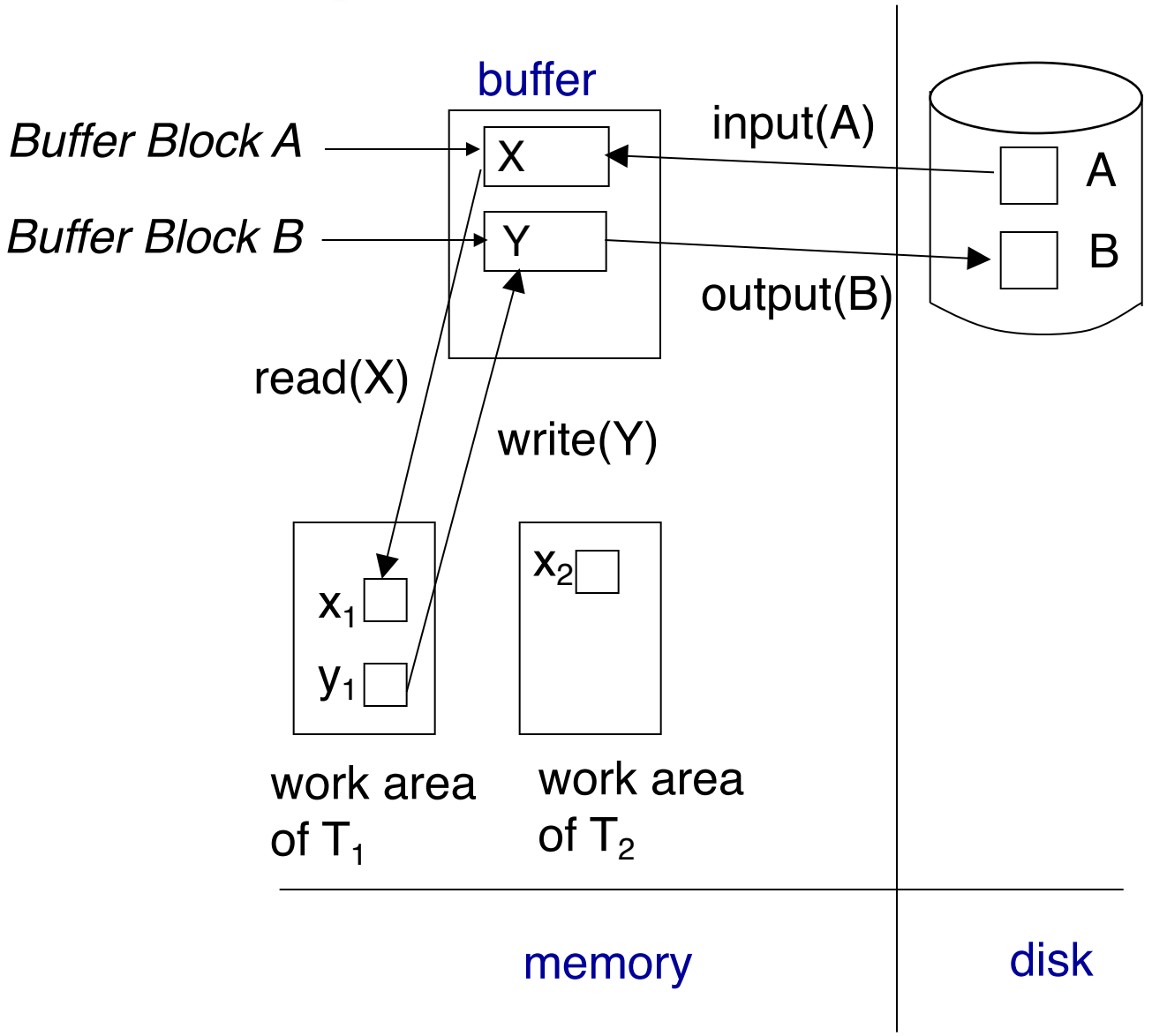
Approach, Assumptions etc..

- Approach:
  - Guarantee A and D:
    - by controlling how the disk and memory interact,
    - by storing enough information during normal processing to recover from failures
    - by developing algorithms to recover the database state
- Assumptions:
  - System may crash, but the *disk is durable*
  - The only *atomicity* guarantee is that *a disk block write* is *atomic*
- Once again, obvious naïve solutions exist that work, but that are too expensive.
  - E.g. The shadow copy solution
    - Make a copy of the database; do the changes on the copy; do an atomic switch of the *dbpointer* at commit time
  - Goal is to do this as efficiently as possible

Data Access

- **Physical blocks** are those blocks residing on the disk.
- **Buffer blocks** are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input**(*B*) transfers the physical block *B* to main memory.
  - **output**(*B*) transfers the buffer block *B* to the disk, and replaces the appropriate physical block there.
- We assume, for simplicity, that each data item fits in, and is stored inside, a single block.

Example of Data Access



Data Access (Cont.)

- Each transaction T_i has its private work-area in which local copies of all data items accessed and updated by it are kept.
  - T_i's local copy of a data item X is called x_i.
- Transferring data items between system buffer blocks and its private work-area done by:
  - **read**(X) assigns the value of data item X to the local variable x_i.
  - **write**(X) assigns the value of local variable x_i to data item {X} in the buffer block.
  - **Note: output**(B_X) need not immediately follow **write**(X). System can perform the **output** operation when it deems fit.
- Transactions
  - Must perform **read**(X) before accessing X for the first time (subsequent reads can be from local copy)
  - **write**(X) can be executed at any time before the transaction commits

STEAL vs NO STEAL, FORCE vs NO FORCE

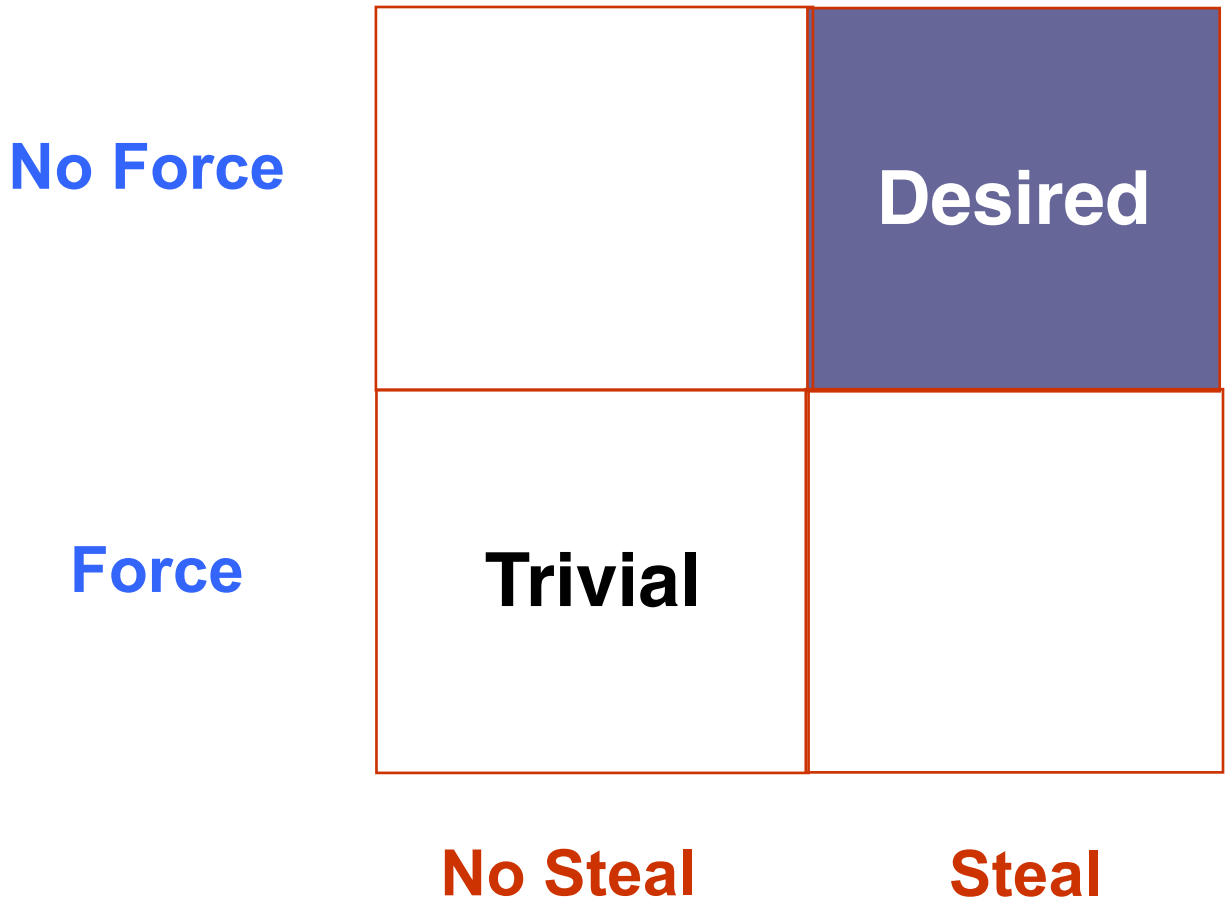
- STEAL:
  - The buffer manager *can steal* a (memory) page from the database
    - ie., it can write an arbitrary page to the disk and use that page for something else from the disk
    - In other words, the database system doesn't control the buffer replacement policy
  - Why a problem ?
    - The page might contain *dirty writes,* ie., writes/updates by a transaction that hasn't committed
  - But, we must allow *steal* for performance reasons.

- NO STEAL:
  - Not allowed. More control, but less flexibility for the buffer manager.

STEAL vs NO STEAL, FORCE vs NO FORCE

- FORCE:
  - The database system *forces* all the updates of a transaction to disk before committing
  - Why ?
    - To make its updates permanent before committing
  - Why a problem ?
    - Most probably random I/Os, so poor response time and throughput
    - Interferes with the disk controlling policies
- NO FORCE:
  - Don't do the above. Desired.
  - Problem:
    - Guaranteeing durability becomes hard
  - We might still have to *force* some pages to disk, but minimal.

STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications



STEAL vs NO STEAL, FORCE vs NO FORCE: Recovery implications

- How to implement A and D when No Steal and Force ?
  - Only updates from committed transaction are written to disk (since no steal)
  - Updates from a transaction are forced to disk before commit (since force)
    - A minor problem: how do you guarantee that all updates from a transaction make it to the disk atomically ?
      - Remember we are only guaranteed an atomic *block write*
      - What if some updates make it to disk, and other don't ?
    - Can use something like shadow copying/shadow paging
  - No atomicity/durability problem arise.

Terminology

- Deferred Database Modification:
  - Similar to NO STEAL, NO FORCE
    - Not identical
  - Only need *redos, no undos*
  - We won't cover this in detail

- Immediate Database Modification:
  - Similar to STEAL, NO FORCE
  - Need both *redos, and undos*

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Basics of Logging and UNDO

Instructor: Amol Deshpande
amol@umd.edu

Transactions: Recovery

- Book Chapters
  - 16.3.1, 16.3.5
- Key topics:
  - Generating log records
  - Using log records to support UNDO/Rollback

Log-based Recovery

- Most commonly used recovery method
- Intuitively, a log is a record of everything the database system does
- For every operation done by the database, a *log record* is generated and stored *typically on a different (log) disk*
- *<T1, START>*
- *<T2, COMMIT>*
- *<T2, ABORT>*
- *<T1, A, 100, 200>*
  - T1 modified A; old value = 100, new value = 200

Log

- 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)

- Log:

(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>

Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
  - ★ The first assumption can be relaxed
  - ★ As a special case, a transaction is considered *committed* only after the *<T1, COMMIT> has been pushed to the disk*
- But, this seems like exactly what we are trying to avoid ??
  - ★ Log writes are *sequential*
  - ★ They are also typically on a different disk
- Aside: LFS == log-structured file system

Log-based Recovery

- Assumptions:
  1. Log records are immediately pushed to the disk as soon as they are generated
  2. Log records are written to disk in the order generated
  3. A log record is generated *before* the actual data value is updated
  4. *Strict two-phase locking*
  - ★ The first assumption can be relaxed
  - ★ As a special case, a transaction is considered *committed* only after the *<T1, COMMIT> has been pushed to the disk*
- NOTE: As a result of assumptions 1 and 2, if *data item A* is updated, the log record corresponding to the update is always forced to the disk before *data item A* is written to the disk
  - ★ This is actually the only property we need; assumption 1 can be relaxed to just guarantee this (called *write-ahead logging)*

Using the log to *abort/rollback*

- STEAL is allowed, so changes of a transaction may have made it to the disk

- UNDO(T1):
  - Procedure executed to *rollback/undo* the effects of a transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - T1 decides to abort
  - Any of the changes might have made it to the disk

Using the log to *abort/rollback*

- UNDO(T1):
  - Go *backwards* in the *log* looking for log records belonging to T1
  - Restore the values to the old values
  - NOTE: Going backwards is important.
    - *A* was updated twice
  - In the example, we simply:
    - Restore A to 300
    - Restore B to 400
    - Restore A to 200
  - Write a log record $<T_i , X_j, V_1>$
    - such log records are called **compensation log records**
    - **<T1, A, 300>, <T1, B, 400>, <T1, A, 200>**
  - Note: No other transaction better have changed A or B in the meantime
    - *Strict two-phase locking*

CMSC424: Database Design

Module: Transactions and ACID Properties

Recovery: Log-based Restart Recovery

Instructor: Amol Deshpande
amol@umd.edu

Using Logs for Recovery

- Book Chapters
  - 16.4
- Key topics:
  - How to use logs for REDO
  - Idempotency of log records
  - Restart recovery after a failure

Using the log to *recover*

- We don't require FORCE, so a change made by the committed transaction may not have made it to the disk before the system crashed
  - BUT, the log record did (recall our assumptions)
- REDO(T1):
  - Procedure executed to recover a committed transaction
  - E.g.
    - *<T1, START>*
    - *<T1, A, 200, 300>*
    - *<T1, B, 400, 300>*
    - *<T1, A, 300, 200>* *[[note: second update of A]]*
    - *<T1, COMMIT>*
  - By our assumptions, all the log records made it to the disk (since the transaction committed)
  - But any or none of the changes to A or B might have made it to disk

Using the log to *recover*

- REDO(T1):
  - Go *forwards* in the *log* looking for log records belonging to T1
  - Set the values to the new values
  - NOTE: Going forwards is important.
  - In the example, we simply:
    - Set A to 300
    - Set B to 300
    - Set A to 200

Idempotency

- Both redo and undo are required to *idempotent*
  - *F is idempotent, if F(x) = F(F(x)) = F(F(F(F(…F(x)))))*
- Multiple applications shouldn't change the effect
  - This is important because we don't know exactly what made it to the disk, and we can't keep track of that
  - E.g. consider a log record of the type
    - <T1, A, *incremented by 100*>
    - Old value was 200, and so new value was 300
  - But the on disk value might be 200 or 300 (since we have no control over the buffer manager)
  - So we have no idea whether to apply this log record or not
  - Hence, *value based logging* is used (also called *physical)*, not operation based (also called *logical)*

Log-based recovery

- Log is maintained
- If during the normal processing, a transaction needs to abort
  - UNDO() is used for that purpose
- If the system crashes, then we need to do recovery using both UNDO() and REDO()
  - Some transactions that were going on at the time of crash may not have completed, and must be *aborted/undone*
  - Some transaction may have committed, but their changes didn't make it to disk, so they must be *redone*
  - Called *restart recovery*

Recovery Algorithm (Cont.)

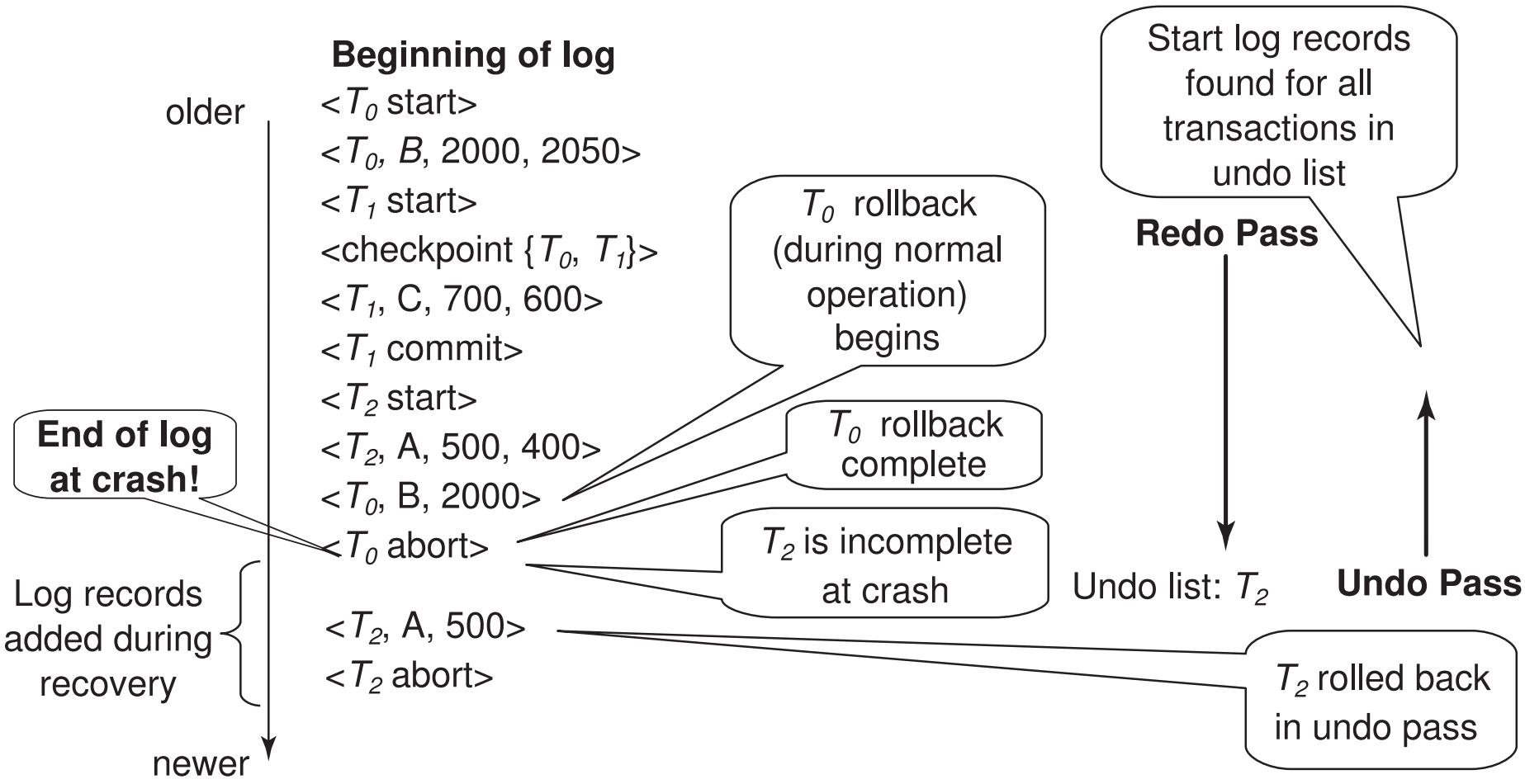
- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions

- **Redo phase**:
  1. Set undo-list to *{} (empty)*.
  2. Scan forward from first log record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**$>$ is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**$>$ *or* $<T_i$ **abort**$>$ is found, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**
  1. Scan log backwards from end
     1. Whenever a log record $<T_i, X_j, V_1, V_2>$ is found where T_i is in undo-list perform same actions as for transaction rollback:
        1. perform undo by writing V_1 to X_j.
        2. write a log record $<T_i, X_j, V_1>$
     2. Whenever a log record $<T_i$ **start**$>$ is found where T_i is in undo-list,
        1. Write a log record $<T_i$ **abort**$>$
        2. Remove T_i from undo-list
     3. Stop when undo-list is empty
        - i.e. $<T_i$ **start**$>$ has been found for every transaction in undo-list
- After undo phase completes, normal transaction processing can commence

Example of Recovery



CMSC424: Database Design

Module: Transactions and ACID Properties

Checkpointing; Write-ahead Logging; Recap

Instructor: Amol Deshpande
amol@umd.edu

Recovery: Recap

- Book Chapters
  - 16.3.6, 16.5
- Key topics:
  - Checkpointing
  - Write-ahead logging
  - Recap

Checkpointing

- How far should we go back in the log while constructing redo and undo lists ??
  - It is possible that a transaction made an update at the very beginning of the system, and that update never made it to disk
    - very very unlikely, but possible (because we don't do force)
  - For correctness, we have to go back all the way to the beginning of the log
  - Bad idea !!

- Checkpointing is a mechanism to reduce this

Checkpointing

- Periodically, the database system writes out everything in the memory to disk
  - Goal is to get the database in a state that we know (not necessarily consistent state)
- Steps:
  - Stop all other activity in the database system
  - Write out the entire contents of the memory to the disk
    - Only need to write updated pages, so not so bad
    - Entire === all updates, whether committed or not
  - Write out all the log records to the disk
  - Write out a special log record to disk
    - *<CHECKPOINT LIST_OF_ACTIVE_TRANSACTIONS>*
    - The second component is the list of all active transactions in the system right now
  - Continue with the transactions again

Recovery Algorithm (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of **all** transactions, whether they committed, aborted, or are incomplete
  - **Undo phase**: undo all incomplete transactions

- **Redo phase** (No difference for Undo phase):
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.

     - If no checkpoint record, start at the beginning
  2. Scan forward from above <**checkpoint** *L*> record
     1. Whenever a record $<T_i, X_j, V_1, V_2>$ is found, redo it by writing V_2 to X_j
     2. Whenever a log record $<T_i$ **start**> is found, add T_i to undo-list
     3. Whenever a log record $<T_i$ **commit**> *or* $<T_i$ **abort**> is found, remove T_i from undo-list

Recap so far …

- Log-based recovery
  - Uses a *log* to aid during recovery
- UNDO()
  - Used for normal transaction abort/rollback, as well as during restart recovery
- REDO()
  - Used during restart recovery
- Checkpoints
  - Used to reduce the restart recovery time

Write-ahead logging

- We assumed that log records are written to disk as soon as generated
  - Too restrictive
- Write-ahead logging:
  - Before an update on a data item (say A) makes it to disk, the log records referring to the update must be forced to disk
  - How ?
    - Each log record has a log sequence number (LSN)
      - Monotonically increasing
    - For each page in the memory, we maintain the LSN of the *last log record* that updated a record on this page
      - *pageLSN*
    - If a page *P* is to be written to disk, all the log records till *pageLSN(P)* are forced to disk

Write-ahead logging

- Write-ahead logging (WAL) is sufficient for all our purposes
  - All the algorithms discussed before work

- Note the special case:
  - A transaction is not considered committed, unless the <T, commit> record is on disk

Other issues

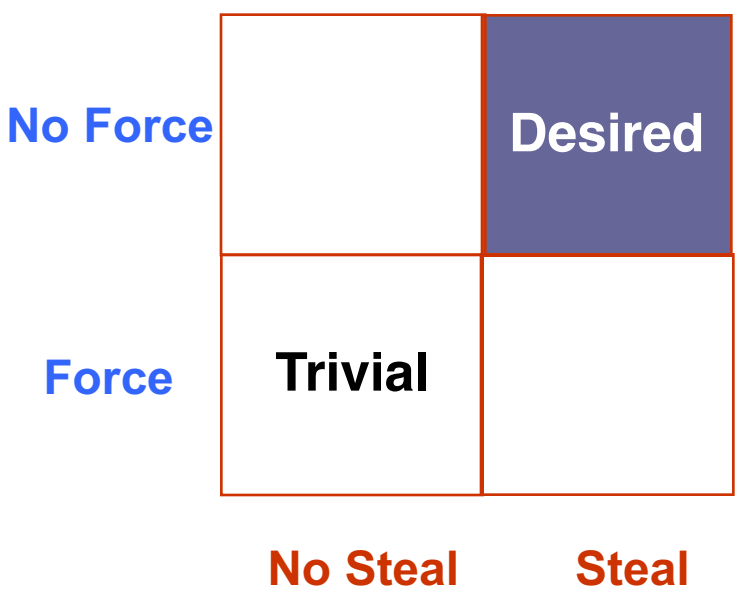
- The system halts during checkpointing
  - Not acceptable
  - Advanced recovery techniques allow the system to continue processing while checkpointing is going on
- System may crash during recovery
  - Our simple protocol is actually fine
  - In general, this can be painful to handle
- B+-Tree and other indexing techniques
  - Strict 2PL is typically not followed (we didn't cover this)
  - So physical logging is not sufficient; must have logical logging

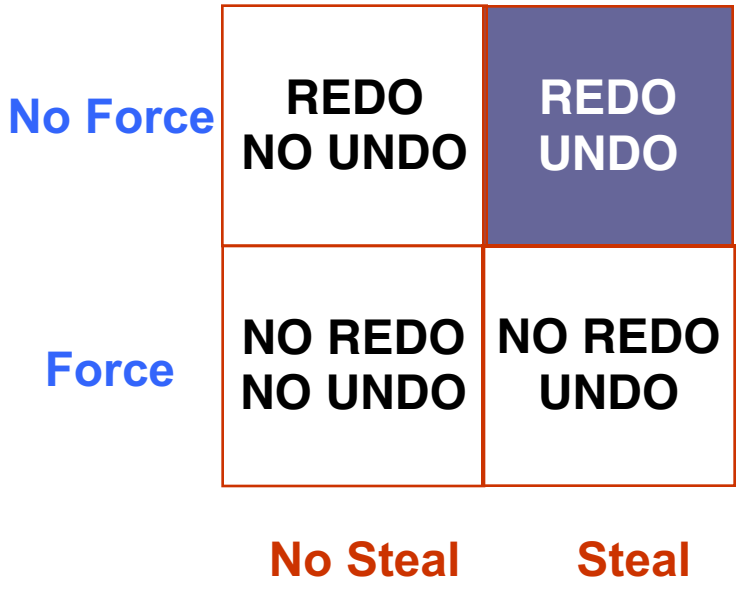
Other issues

- ARIES: Considered *the canonical description of log-based recovery*
  - Used in most systems
  - Has many other types of log records that simplify recovery significantly
- Loss of disk:
  - Can use a scheme similar to checkpoining to periodically dump the database onto *tapes* or *optical storage*
  - Techniques exist for doing this while the transactions are executing (called *fuzzy dumps)*
- Shadow paging:
  - Read up

Recap

- STEAL vs NO STEAL, FORCE vs NO FORCE
  - ★ We studied how to do STEAL and NO FORCE through log-based recovery scheme





Recap

- ACID Properties
  - Atomicity and Durability :
    - Logs, undo(), redo(), WAL etc
  - Consistency and Isolation:
    - Concurrency schemes
  - Strong interactions:
    - We had to assume Strict 2PL for proving correctness of recovery

CMSC424: Database Design

Module: Transactions and ACID Properties

Distributed Transactions

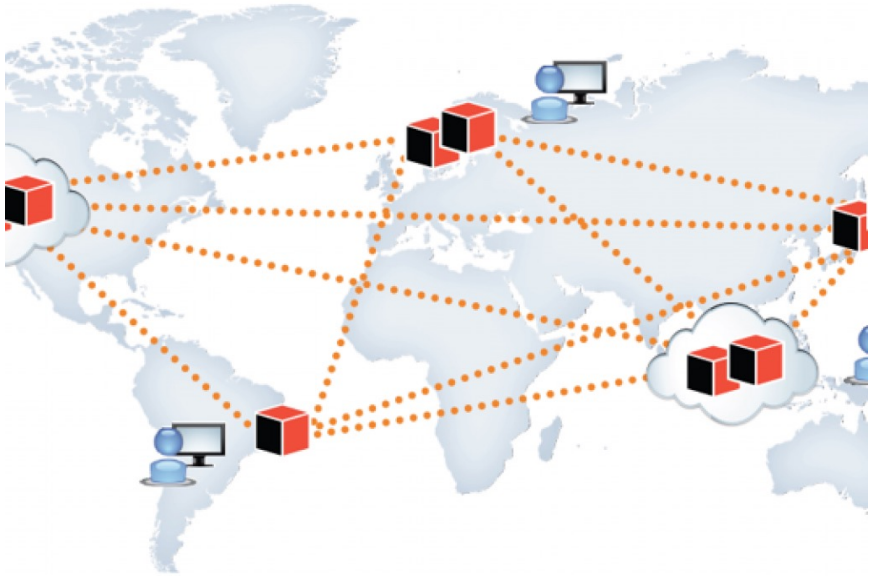
Instructor: Amol Deshpande
amol@umd.edu

Distributed Transactions

- Book Chapters
  - 19.1-19.4, 19.6: at a fairly high level
- Key topics:
  - Distributed databases and replication
  - Transaction processing in distributed databases
  - 2-Phase Commit
  - Brief discussion of other protocols including Paxos

Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
  - Or not – lot of variations here
- Transactions may access data at one or more sites
  - Because of replication, even updating a single data item involves a "distributed transaction" (to keep all replicas up to date)

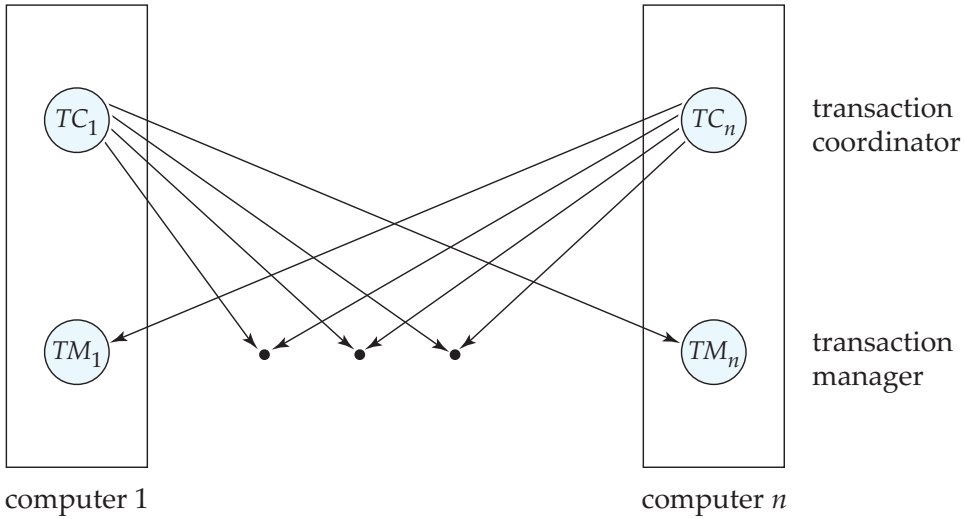


Data Replication

- A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites
- Advantages:
  - **Availability**: failures can be handled through replicas
  - **Parallelism**: queries can be run on any replica
  - **Reduced data transfer**: queries can go to the "closest" replica
- Disadvantages:
  - **Increased cost of updates:** both computation as well as latency
  - **Increased complexity of concurrency control:** need to update all copies of a data item/tuple
- **Typically we use the term "data items", which may be tuples or relations or relation partitions**

Distributed Transactions

- Transaction may access data at several sites
  - As noted, single data item update is also a distributed transaction
- **Each site has a local transaction manager** responsible for:
  - Maintaining a log for recovery purposes
  - Coordinating the concurrent execution of the transactions
- **Each site has a transaction coordinator,** which is responsible for:
  - Starting the execution of transactions that originate at the site.
  - Distributing sub-transactions at appropriate sites for execution.
  - Coordinating the termination of each transaction that originates at the site -- transaction may commit at all sites or abort at all sites.



System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of massages
    - Handled by network transmission control protocols such as TCP-IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - **Network partition**
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node
- Network partitioning and site failures are generally indistinguishable.

Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- **Two-phase *commit* (2PC)** protocol is widely used
- **Three-phase commit (3PC)** protocol
  - Handles some situations that 2PC doesn't
  - Not widely used
- **Paxos**
  - Robust alternative to 2PC that handles more situations as well
  - Was considered too expensive at one point, but widely used today
- **RAFT**: Alternative to Paxos

Two Phase Commit Protocol (2PC)

- Assumes **fail-stop** model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i

Two Phase Commit Protocol (2PC)

Coordinator Log	Messages	Subordinate Log
	PREPARE →	
		prepare*/abort*
	← VOTE YES/NO	
commit*/abort*		
	COMMIT/ABORT→	
		commit*/abort*
	← ACK	
end		

Goal: Make sure all ”sites” commit or abort

Assumption: Some log records can be “forced” (denote * above)

Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction T_i.
  - C_i adds the records <**prepare** *T*> to the log and forces log to stable storage
  - sends **prepare** *T* messages to all sites at which *T* executed

- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <**no** *T*> to the log and send **abort** *T* message to C_i
  - if the transaction can be committed, then:
  - add the record <**ready** *T*> to the log
  - force *all records* for *T* to stable storage
  - send **ready** *T* message to C_i

Phase 2: Recording the Decision

- *T* can be committed of C_i received a **ready** *T* message from all the participating sites: otherwise *T* must be aborted.
- Coordinator adds a decision record, <**commit** *T*> or <a**bort** *T*>, to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)
- Participants take appropriate action locally.

Handling of Failures - Site Failure

When site S_i recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contain <**commit** *T*> record: txn had completed, nothing to be done
- Log contains <**abort** *T*> record: txn had completed, nothing to be done
- Log contains <**ready** *T*> record: site must consult C_i to determine the fate of *T*.
  - If *T* committed, **redo** (*T*); write <**commit** *T*> record
  - If *T* aborted, **undo** (*T*)
- The log contains no log records concerning *T*:
  - Implies that S_k failed before responding to the **prepare** *T* message from C_i
  - since the failure of S_k precludes the sending of such a response, coordinator C_1 must abort *T*
  - S_k must execute **undo** (*T*)

Handling of Failures- Coordinator Failure

- If coordinator fails while the commit protocol for T is executing then participating sites must decide on T's fate:
  1. If an active site contains a <**commit** T> record in its log, then T must be committed.
  2. If an active site contains an <**abort** T> record in its log, then T must be aborted.
  3. If some active participating site does not contain a <**ready** T> record in its log, then the failed coordinator C_i cannot have decided to commit T.
     - Can therefore abort T; however, such a site must reject any subsequent <**prepare** T> message from C_i
  4. If none of the above cases holds, then all active sites must have a <**ready** T> record in their logs, but no additional control records (such as <**abort** T> of <**commit** T>).
     - In this case active sites must wait for C_i to recover, to find decision.
- **Blocking problem**: active sites may have to wait for failed coordinator to recover.

Handling of Failures - Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to wait for decision from coordinator.
- The coordinator and the sites are in the same partition as the coordinator think that the sites in the other partition have failed, and follow the usual commit protocol.
    - Again, no harm results

More…

- Three-phase Commit
  - 2PC can't handle failure of a coordinator well – everything halts waiting for the coordinator to come back up
  - Three-phase commit handles that through another phase

- Paxos and RAFT
  - Solutions for the "consensus problem": get a collection of distributed entities to "choose" a single value
    - In case of transaction, you are choosing abort/commit
  - Fairly complex, but well-understood today
  - Widely used in most distributed systems today
  - See the Wikipedia pages
  - A nice recent paper: **Paxos vs Raft: Have we reached consensus on distributed consensus? – Heidi Howard, 2020**

More…

- Bitcoin (and other cryptocurrencies)
  - Fundamental problem is the same one, of obtaining “consensus”
    - But need to support a large number of entities, 1000s or more
    - Can’t assume full one-to-one communication
  - Instead:
    - Choose a “leader” based on ”proof of work”
      - Whoever solves a hard puzzle first becomes the “leader”
    - The ”leader” chooses the next “block” in the blockchain
      - A block is basically a list of transactions to accept
    - Reward the puzzle solvers with money (“bitcoins”)
      - So they have an incentive to keep solving puzzles
  - Blockchain?
    - Blockchain is a small part of bitcoin
    - A cryptographically designed chain of blocks that are immutable