



Principles of Database Systems

Course Number: CSGY-6083

Section Number: B

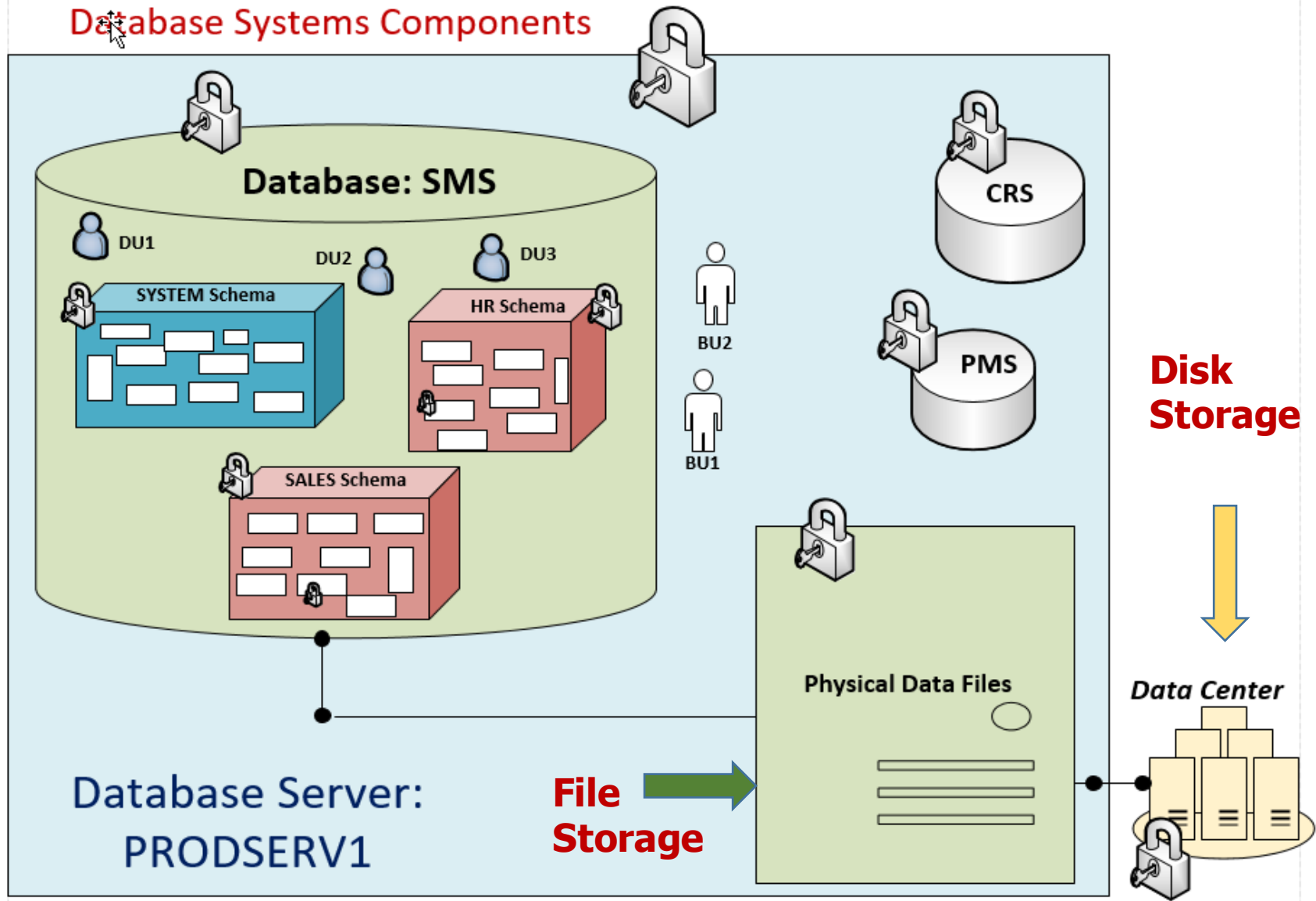
Module 4

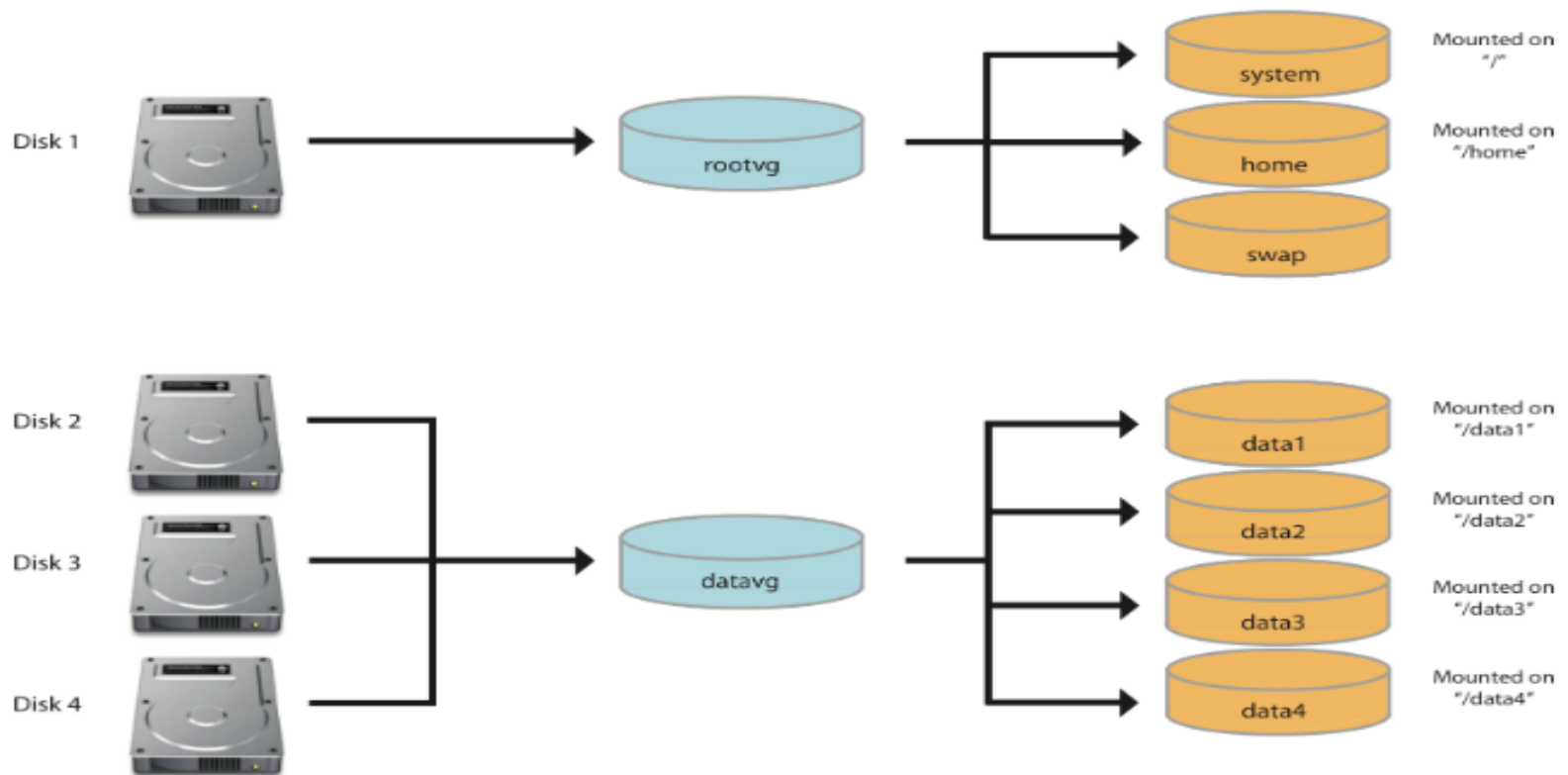
Instructor : Amit Patel

MS, PMP®, OCP®

Email: asp13@nyu.edu
patelamitnyu@gmail.com

Database Systems Components





Physical volumes (Hard Disks)

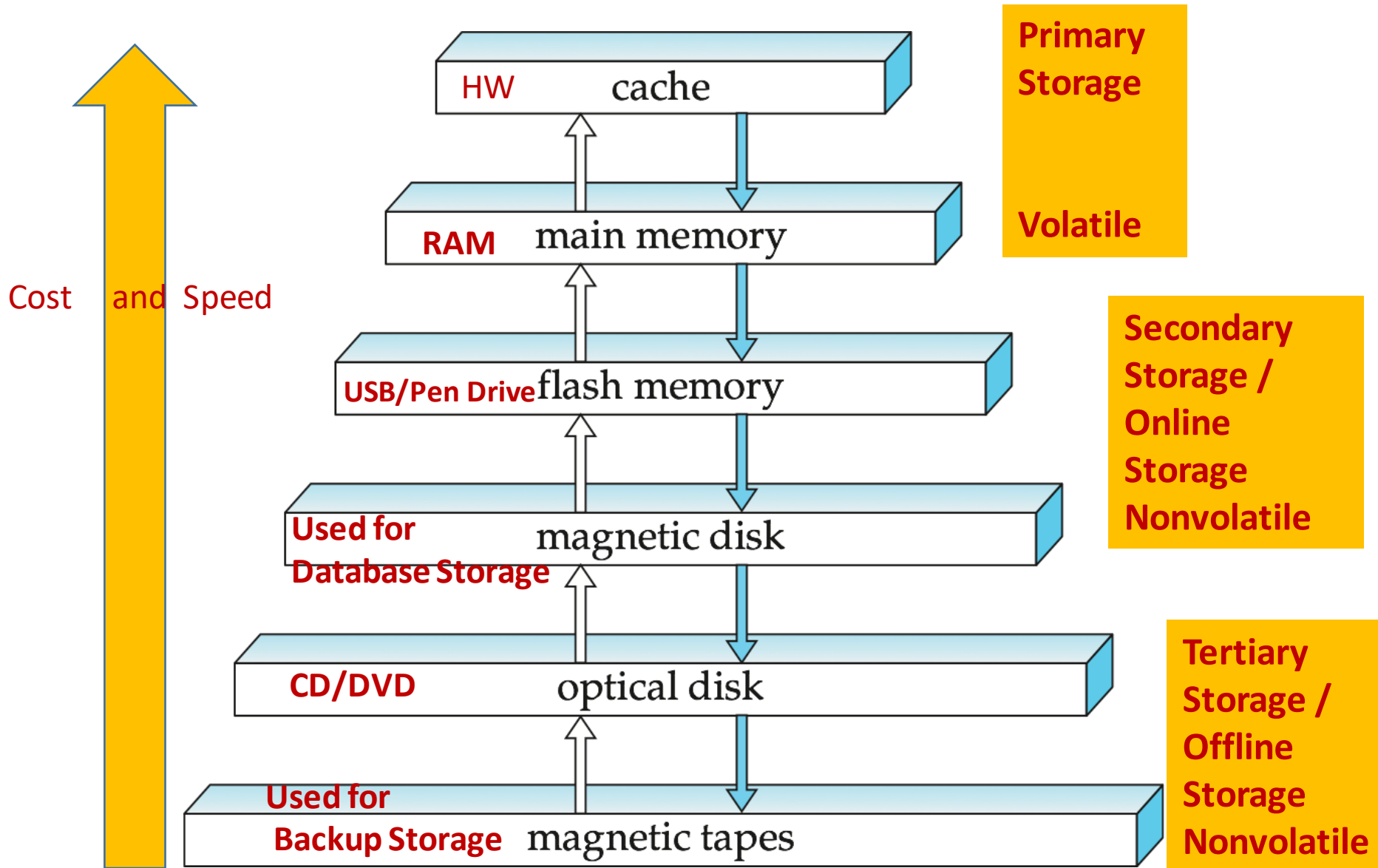
Volume Groups



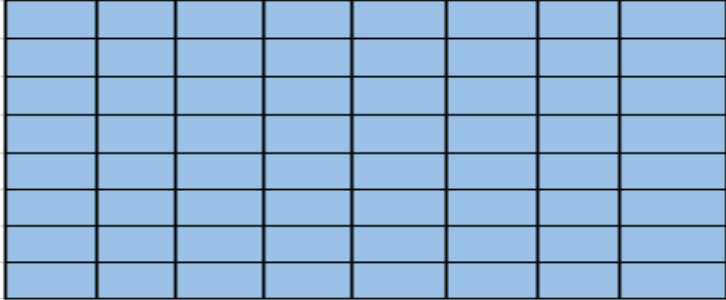
Logical volumes (file systems)

Classification of Physical Storage Media

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

Storage Hierarchy



Bit (Binary value 1 or 0)							
							
							
Byte= 8 Bit							
Byte is the smallest Unit of Storage							
							
Block = 8KB							
8KB = 8 KiloBytes							
= 8*1024 Bytes							
=8192 Bytes							
Block is typically baic unit of database storage							
SIZE UNITS							
1KB = 1024 Bytes							
1MB = 1024 KB							
1GB = 1024 MG							
1TB = 1024 GB							
1PB = 1024 TB							
Number blocks in 8GB datafile:							
= 8 GB/8 KB							
= 8*1024 MB/8 KB							
=8*1024*1024 KB/8 KB							
=1,048, 576							

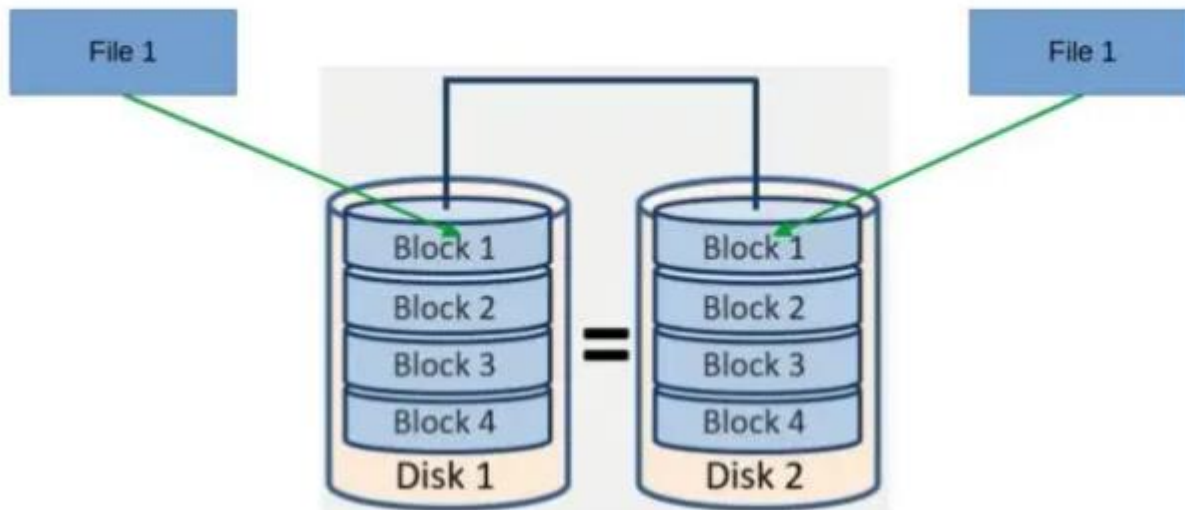
RAID – for Performance and Reliability

- **RAID: Redundant Arrays of Independent Disks**
 - **disk organization techniques that manage a large numbers of disks, providing a view of a single volume.**
 - **high capacity and high speed** by using multiple disks in parallel (**performance gain**) == **throughput of the disks, transfer rate of data**
 - **high reliability** by storing data redundantly, so that data can be recovered even if a disk fails (**reliability gain**)
- **The chance that some disk out of a set of N disks will fail is much higher than the chance that a specific single disk will fail.**

Improvement of Reliability via Redundancy

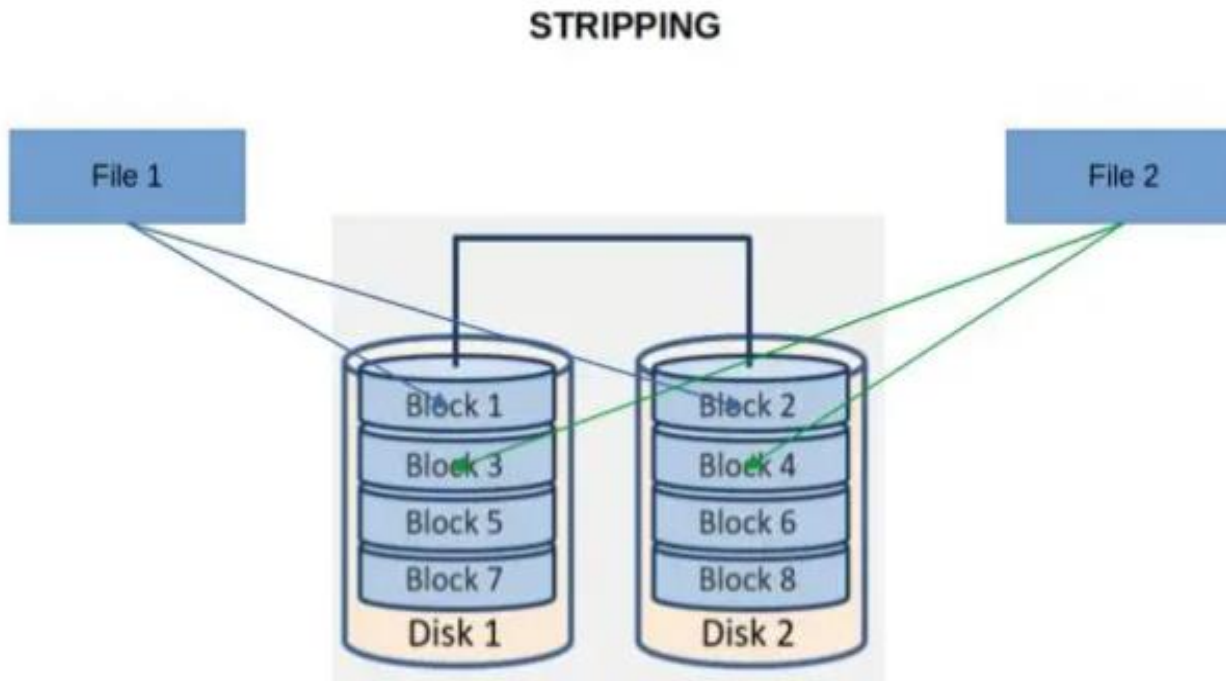
- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or shadowing)
 - Duplicate every disk. Logical disk consists of two physical disks.
 - Every write is carried out on both disks
 - Reads can take place from either disk
 - If one disk in a pair fails, data still available in the other

Mirroring



Improvement in Performance via Parallelism

- **Two main goals of parallelism in a disk system:**
 1. **Load balance multiple small accesses to increase throughput**
 2. **Parallelize large accesses to reduce response time.**
- **Improve transfer rate by striping data across multiple disks.**



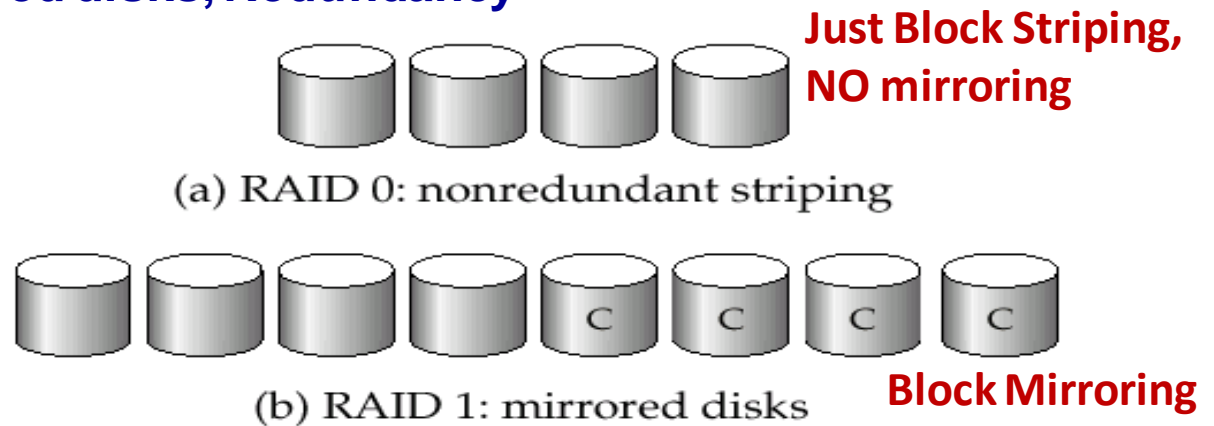
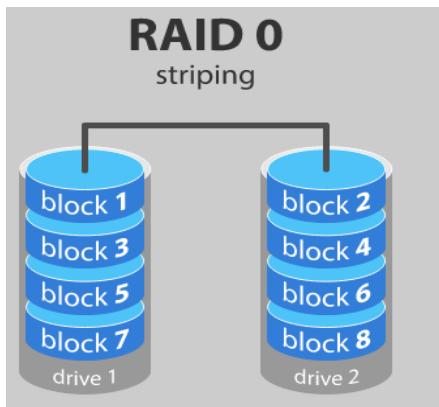
RAID Levels

- **RAID Level** : Different schemes (techniques) of disk organization to achieve reliability (via Redundancy/Mirroring) and Performance (via Parallelism / Stripping)
- Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics

□ RAID Level 0: Block striping; non-redundant.

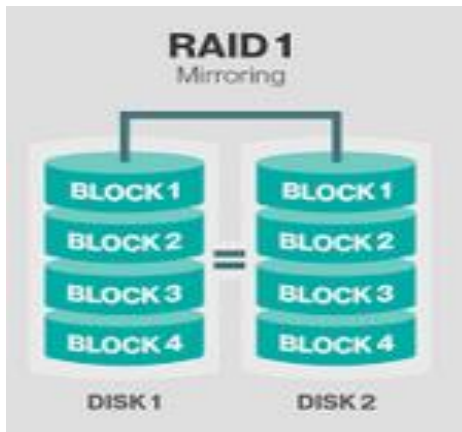
- Used in high-performance applications where data loss is not critical.

□ RAID Level 1: Mirrored disks, Redundancy



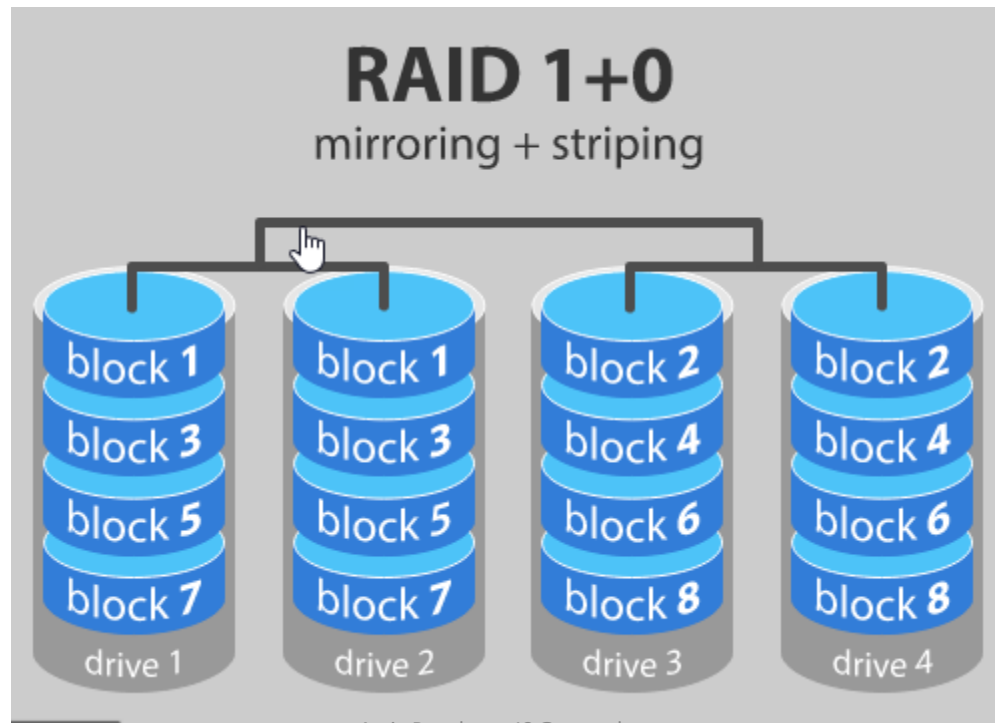
Striping provides high data transfer rate, but not improved reliability

Mirroring provides high reliability but it is expensive



RAID Levels (Cont.)

- ❑ Some Vendor says RAID Level 10 or RAID Level 1+0
[Striping and Mirroring]
- ❑ Offers best write performance.
- ❑ Popular for applications such as storing achieve log files in a database system



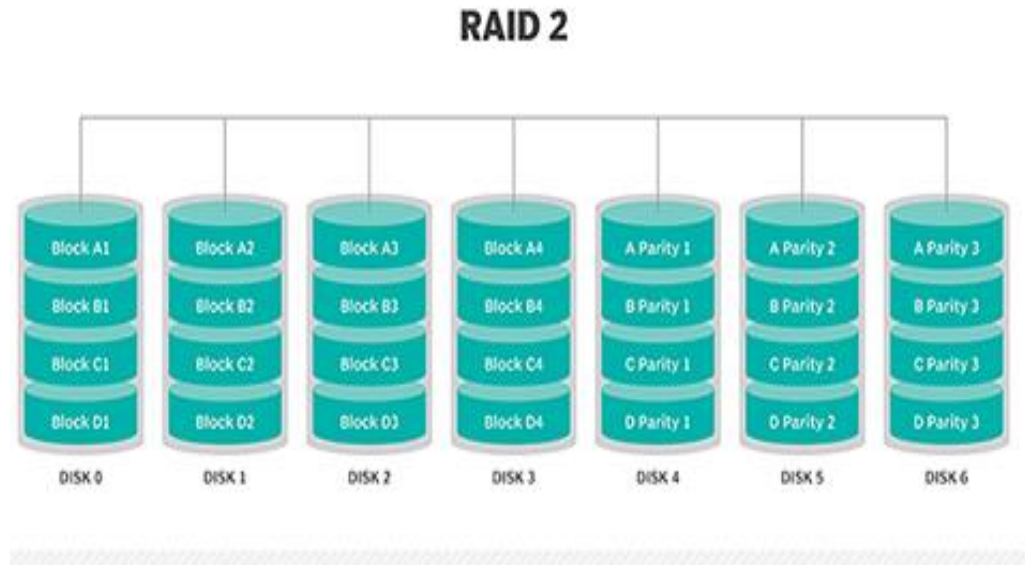
RAID Levels (Cont.)

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

Parity Bit for Error detection and correction: Each byte in memory structure may have a bit associated with it that provides whether the number of bits in bytes that are set to even (parity 0) or odd (parity 1). If one of the bits in byte get damaged (either 1 becomes 0 or 1 becomes 1), the parity of byte changes and thus do not match the stored parity. Similarly, if stored parity get damaged, it will not match the computed parity. Thus, error will be detected by memory system. Error correction scheme stores 2 or more extra bits and can construct the data.



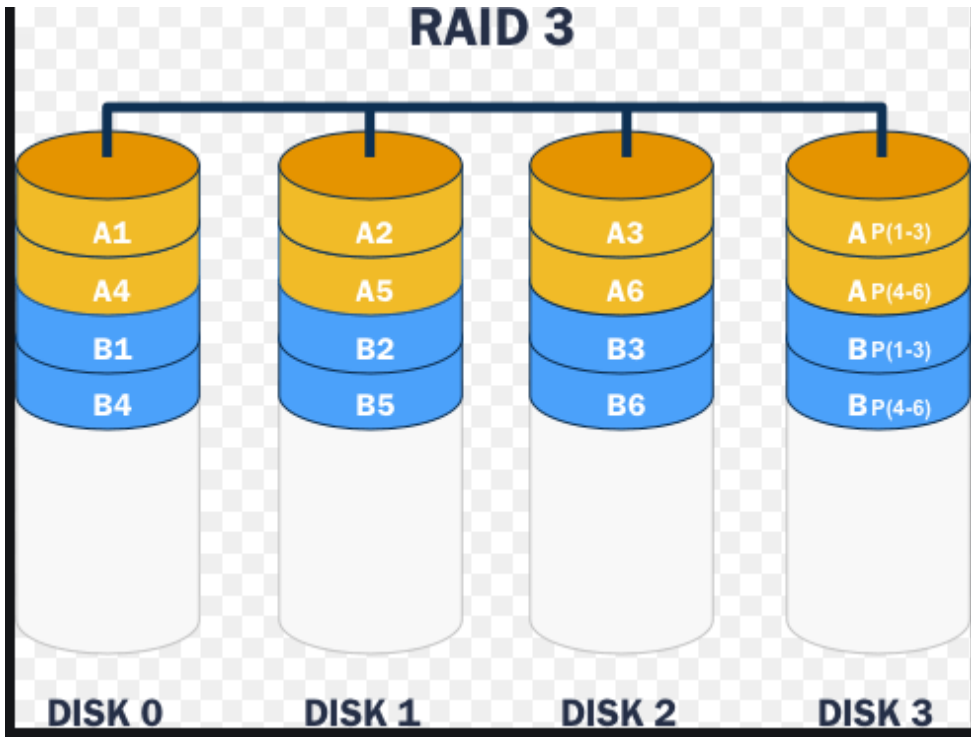
(c) RAID 2: memory-style error-correcting codes



RAID Levels (Cont.)

- **RAID Level 3: Bit-Interleaved Parity**

- a single parity bit is enough for error correction, not just detection, since we know which disk has failed
 - When writing data, corresponding parity bits must also be computed and written to a parity bit disk



(d) RAID 3: bit-interleaved parity

Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O

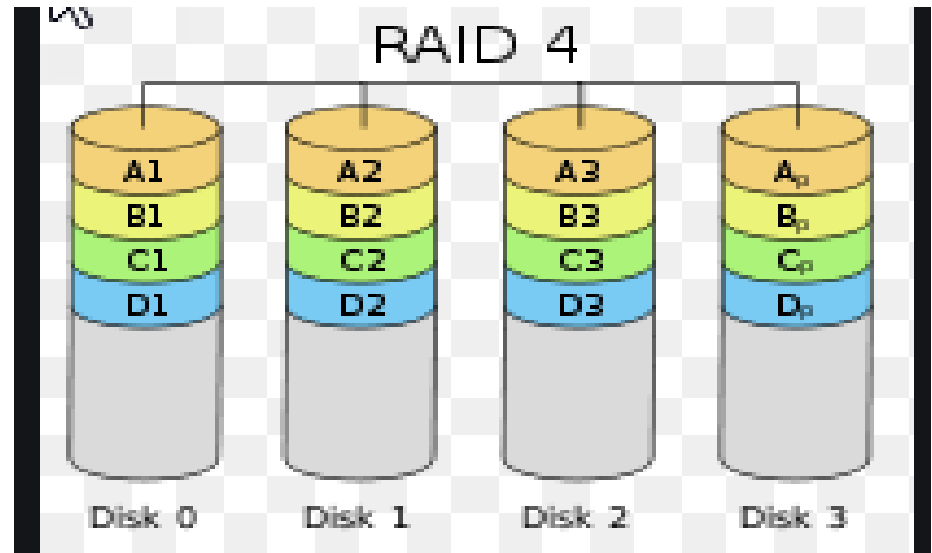
Subsumes Level 2 provides all its benefits, at lower cost)

RAID Levels (Cont.)

- **RAID Level 4: Block-Interleaved Parity**; uses block-level striping and keeps a parity block on a separate disk for corresponding blocks from N other disks.
 - When writing data block, corresponding block of parity bits must also be computed and written to parity disk



(e) RAID 4: block-interleaved parity



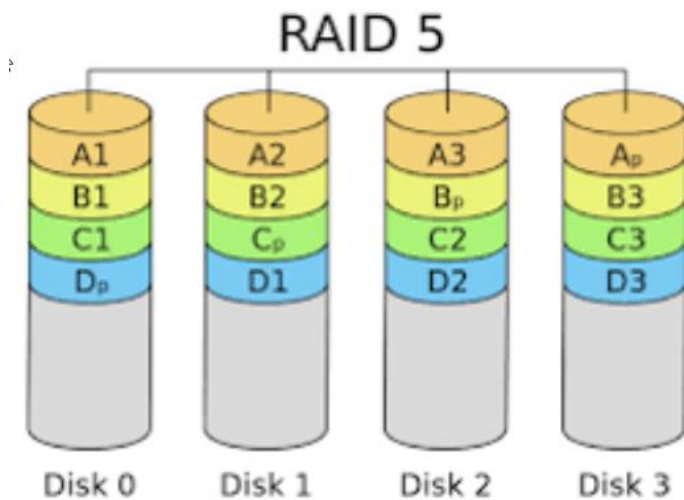
RAID Levels (Cont.)

- **RAID Level 5: Block-Interleaved Distributed Parity;** partitions data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in 1 disk.
 - E.g., with 5 disks, parity block for n th set of parity blocks is stored on disk $(n \bmod 5) + 1$, with the data blocks stored on the other 4 disks (assume that block number start with 0)

Parity Block cannot store Parity for Blocks in same disk since failure of disk result in loss of data and parity block, cannot be recovered.



(f) RAID 5: block-interleaved distributed parity



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

RAID Levels (Cont.)

- **RAID Level 5 (Cont.)**

- **Higher I/O rates than Level 4.**
- **Subsumes Level 4: provides same benefits but avoids bottleneck of parity disk.**

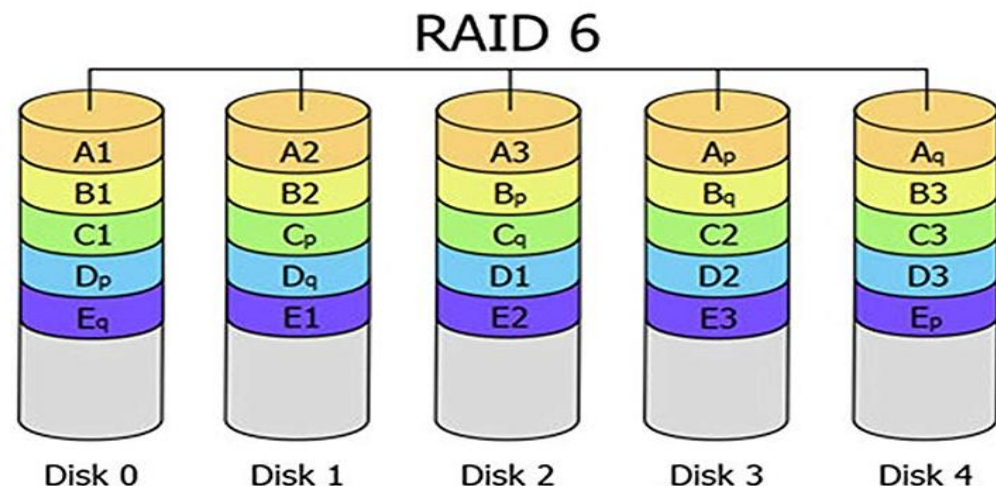
- **RAID Level 6: P+Q Redundancy** scheme; similar to Level 5, but stores extra redundant information to guard against multiple disk failures.

- **Better reliability than Level 5 at a higher cost; used widely.**

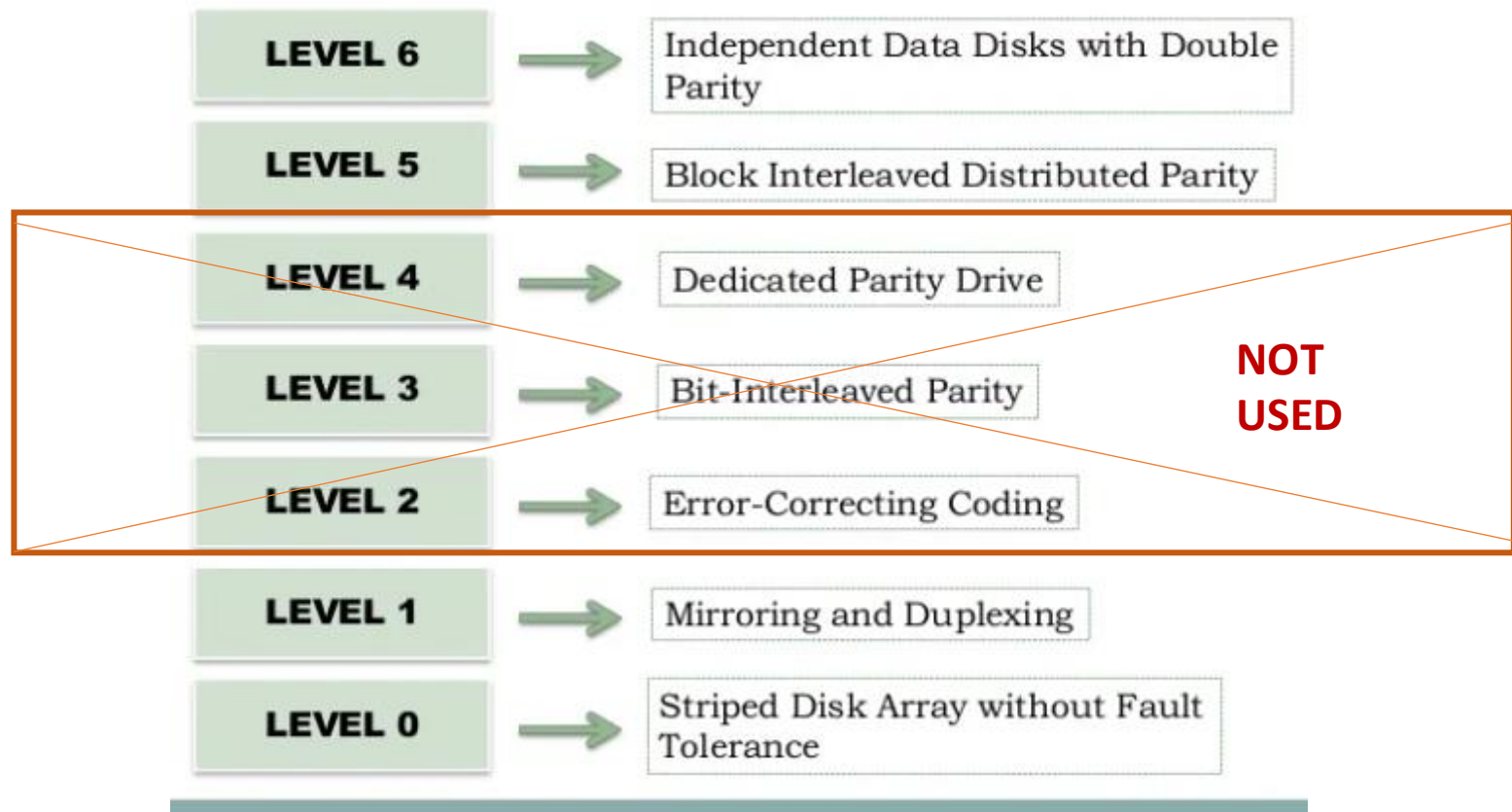


(g) RAID 6: P + Q redundancy

**Guards against multiple disk failure.
2 bit of redundant data stored for
every 4 bit of data, unlike 1 parity bit
in RAID 5.**



Summary of RAID levels



Choice of RAID Level

- Factors in choosing RAID level
 - Monetary cost
 - Performance: Number of I/O operations per second
 - Performance during failure
 - Performance during rebuild of failed disk, Including time taken to rebuild failed disk
- Level 0 is used only when data safety is not important
 - E.g. data can be recovered quickly from other sources
- Level 1 is preferred for applications with low update rate, and large amounts of data
- Level 2 and Level 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement, which block striping (Level 5) avoids
- Level 6 is often used when risk tolerance of organization is low and can afford more cost, else Level 5 offer adequate safety for most applications.
- Level 10 (Level 0 + Level 1) is mostly used for high writing needs, such as for Archive Log files of databases.

File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
- One approach:
 - assume record size is fixed
 - each file has records of one particular type only
 - different files are used for different relationsThis case is easiest to implement

Database → File → Block → Record (of Multiple Table) → Columns

Fixed-Length Records

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

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

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

Deleting record 3 and compacting

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

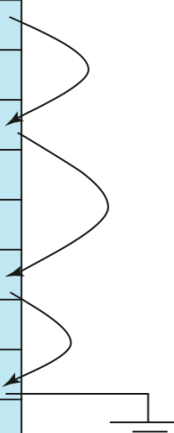
Deleting record 3 and moving last record

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

Free Lists

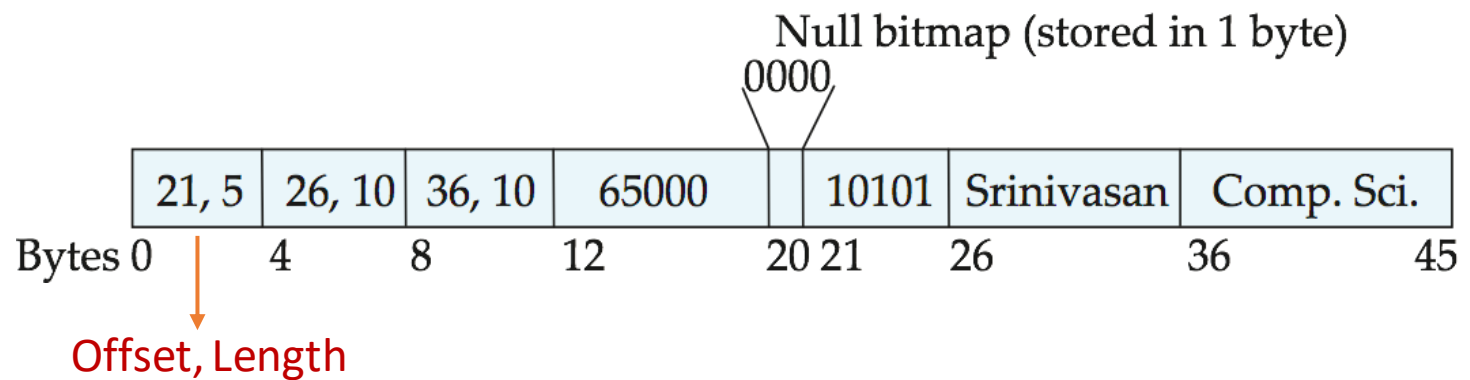
- Store the address of the first deleted record in the file header.
- Use this first record to store the address of the second deleted record, and so on
- Can think of these stored addresses as **pointers** since they “point” to the location of a record.
- More space efficient representation: reuse space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

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



Variable-Length Records

- Variable-length records arise in database systems in several ways:
 - Record types that allow variable lengths for one or more fields such as strings (varchar)
 - Storage of multiple record types in file
- Two different challenges must be solved
 - ✓ How to represent single record in such a way that individual attribute can be easily extracted
 - ✓ How to store variable length record within a block such that record in block can be easily extracted
- Attributes are stored in order
- Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
- Null values represented by null-value bitmap



Offset: Where data for attribute begins

Length: Length in bytes for variable length attributes

Offset and Length are stored two bytes each, stored in total 4 bytes

Numbers and Date type attributes are considered fixed length attribute

Salary is assumed to be stored in 8 bytes

Null bitmap indicates which attribute is null. e.g. If salary is null 4th bit on Null Bitmap will 1.

One byte of Null Bitmap can handle up-to 8 null bitmap, so up-to 8 attributes. If we have up-to 16 attributes, we will need two bytes to store Null Bitmap, and if we have up-to 24 attributes, we will need three bytes to store Null Bitmap and so on.

INDEX: Concept of Table of Contents

CONTENTS	
TABLES.....	iii
FIGURES.....	iv
1.0 INTRODUCTION.....	1
2.0 DESIGN PROJECT BACKGROUND	2
2.1 Problem Definition.....	2
2.2 Needs Analysis.....	3
2.3 Key Performance and Environmental Requirements	4
2.4 Deliverables	5
2.4.1 Firmware Code.....	5
2.4.2 Hardware.....	5
2.4.3 Report	5
3.0 SYSTEM DESIGN.....	6
3.1 On- & Off-Board Design	6
3.2 Functional Block Descriptions	8
3.3 Hardware Implementation.....	8
3.4 Linear Testing	10
3.4.1 Gain Error	10
3.4.2 Offset Error	11
3.4.3 Differential Non-Linearity.....	12
3.4.4 Integral Non-Linearity.....	13
3.4.5 Total Unadjusted Error.....	14
3.4.6 Missing Codes	15
3.5 Dynamic Testing.....	15
3.5.1 The FFT	16
3.5.2 Total Harmonic Distortion	18
3.5.3 Signal-To-Noise Ratio	18
3.5.4 Spurious Free Dynamic Range	19
3.5.5 Two Tone Intermodulation Distortion	20
3.5.6 Effective Number of Bits	20
4.0 USER INTERFACE	20
5.0 RISK REDUCTION	21
6.0 PROTOTYPE COST	21
7.0 TEST PLAN	22
8.0 PROJECT SCHEDULE.....	22
9.0 CONCLUSION	23
REFERENCES.....	25
APPENDIX A – RELEVANT STANDARDS	A-1
APPENDIX B – BILL OF MATERIALS	B-1
APPENDIX C – GANTT CHART	C-1

- A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage.
- An index helps in rapid access of database records by storing pointers to their actual disk location (address of record).
- Index is created on column or combination of columns
- Table can have more than one index

Index **PK_EMPNO**, EMP (EMPNO)

Search Key EMPNO	ROWID
7369	AF85OtADDAABuvAAA
7499	AF85OtADDAABuvAAB
7521	AF85OtADDAABuvAAC
7566	AF85OtADDAABuvAAD
7654	AF85OtADDAABuvAAE
7698	AF85OtADDAABuvAAF
7782	AF85OtADDAABuvAAG
7788	AF85OtADDAABuvAAH
7838	AF85OtADDAABuvAAI
7844	AF85OtADDAABuvAAJ
7896	AF85OtADDAABuvAAK
7900	AF85OtADDAABuvAAL
7902	AF85OtADDAABuvAAM
7934	AF85OtADDAABuvAAN

SEARCH KEY: Column or Combination of columns on which Index is created.

ROWID: Unique physical location (address) of the row in data file. It is stored in **hexadecimal format**.

AP_EMP
EMPNO
7698
7369
7654
7902
7788
7499
7999
7876
7782
7900
7839
7521
7934
7566
7844

DF1																			
DF2																			
DF3																			
DF4																			
DF5																			

For Index, search key columns are stored in sequential order.

Table records are not stored in sequential order (order of records are irreverent)

INDEX: Basic Concepts

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

search-key	Pointer (address/ROWID)
------------	-------------------------

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

Ordered Indices

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

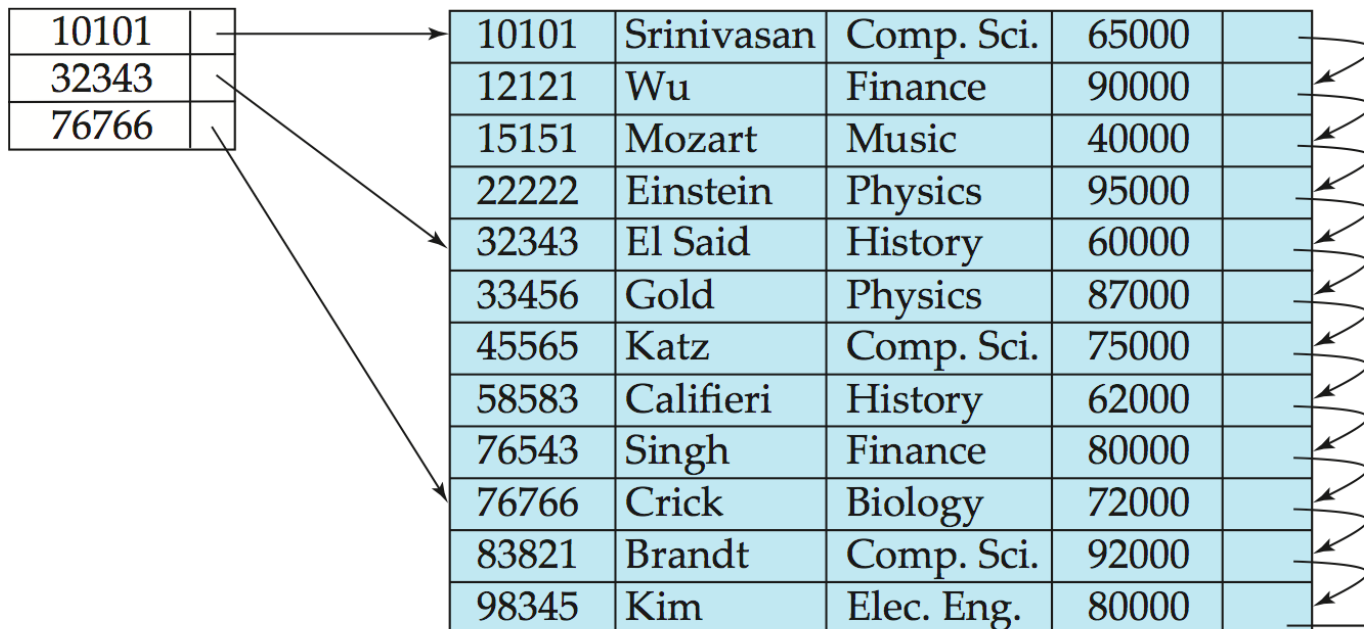
Dense Index Files

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

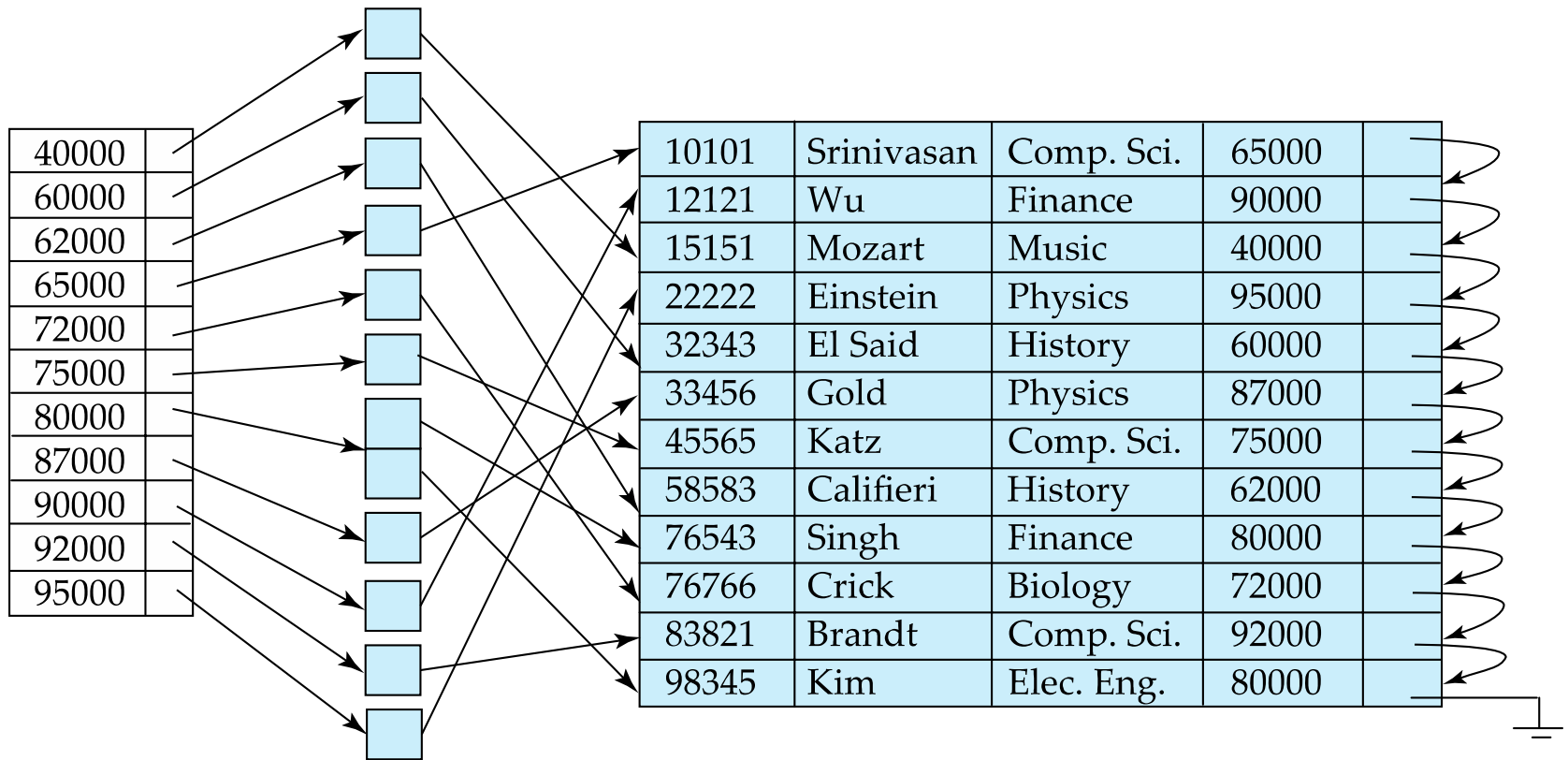
10101	→	10101	Srinivasan	Comp. Sci.	65000	↙
12121	→	12121	Wu	Finance	90000	↙
15151	→	15151	Mozart	Music	40000	↙
22222	→	22222	Einstein	Physics	95000	↙
32343	→	32343	El Said	History	60000	↙
33456	→	33456	Gold	Physics	87000	↙
45565	→	45565	Katz	Comp. Sci.	75000	↙
58583	→	58583	Califieri	History	62000	↙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

Sparse Index Files

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



Secondary Indices Example



Secondary index on *salary* field of *instructor*

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

Secondary Indices

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

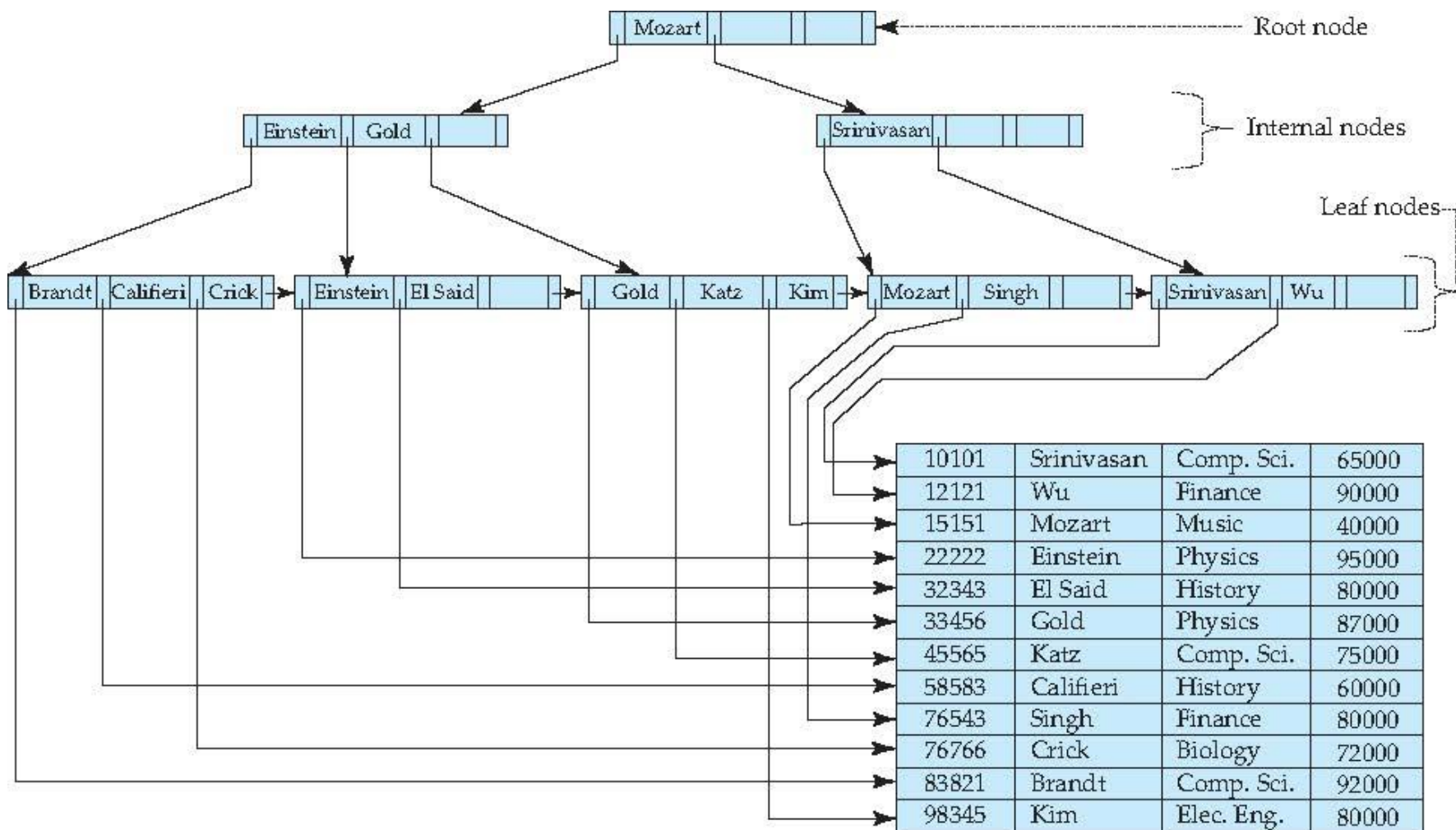
We can have a secondary index with an index record for each search-key value

B⁺ Tree Index Files (Balanced Tree Index)

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

- Disadvantage of indexed-sequential files
 - performance degrades as file grows, since many overflow blocks get created.
 - Periodic reorganization of entire index file is required.
- Advantage of B⁺-tree index files:
 - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
- (Minor) disadvantage of B⁺-trees:
 - extra insertion and deletion overhead, space overhead.
- Advantages of B⁺-trees outweigh disadvantages
 - B⁺-trees are used extensively

Example of B⁺-Tree – Equi Distance Search



B⁺-Tree Index Files (Cont..)

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

- Each node that is not a root or a leaf, called intermediate node and has between $\lceil n-1/2 \rceil$ and n **children**, where n is number of branches from the root.
- A leaf node has between $\lceil n/2 \rceil$ and $n-1$ **values**

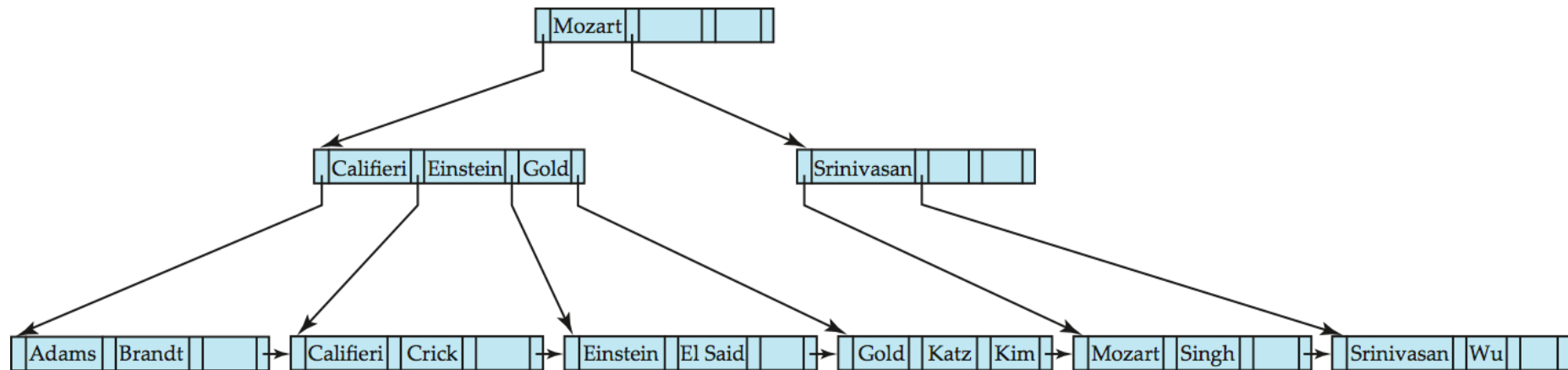
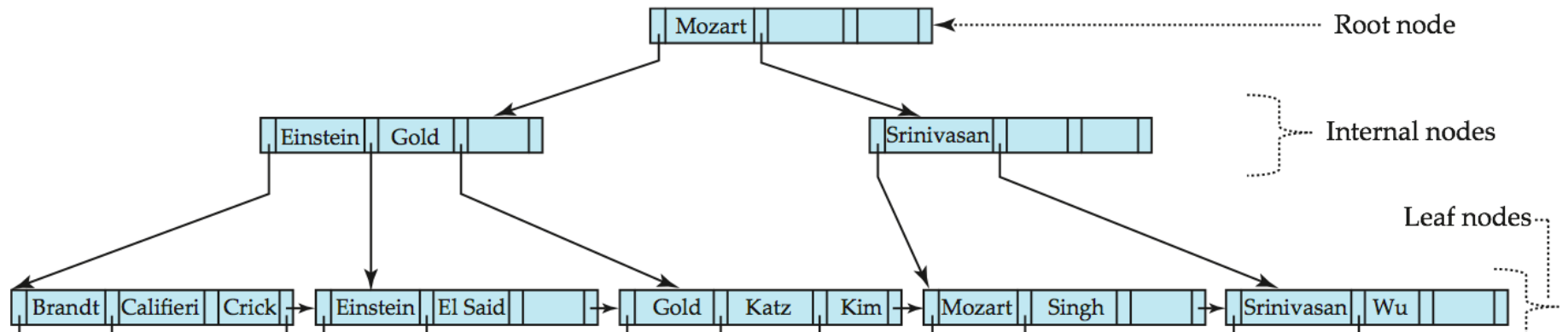
Nodes	Min	Max	With $n=7$	
			Min	Max
Root	1	1	1	1
Intermediate nodes	$\lceil n-1/2 \rceil$	n	3	7
Leaf node	$\lceil n/2 \rceil$	$n-1$	4	6

The main goals of B⁺ tree is:

- ✓ Sorted intermediate and leaf nodes
- ✓ Fast traversal and quick search

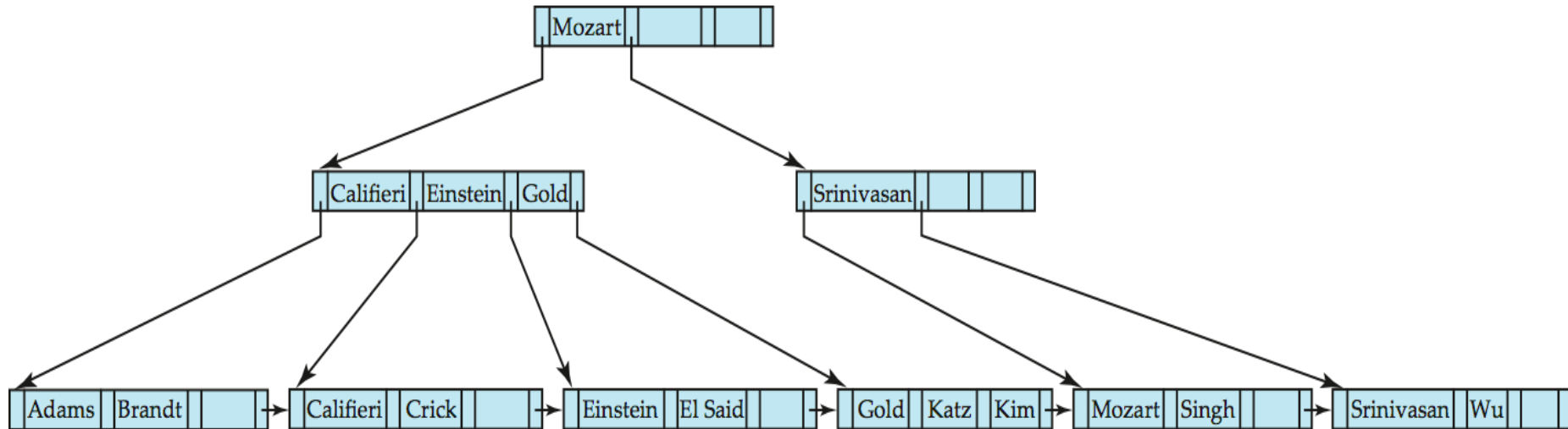
If we have to search for any record, they are all found at leaf node. Hence searching any record will take same time because of equidistance of the leaf nodes. Also they are all sorted. Hence searching a record is like a sequential search and does not take much time.

B⁺-Tree Insertion

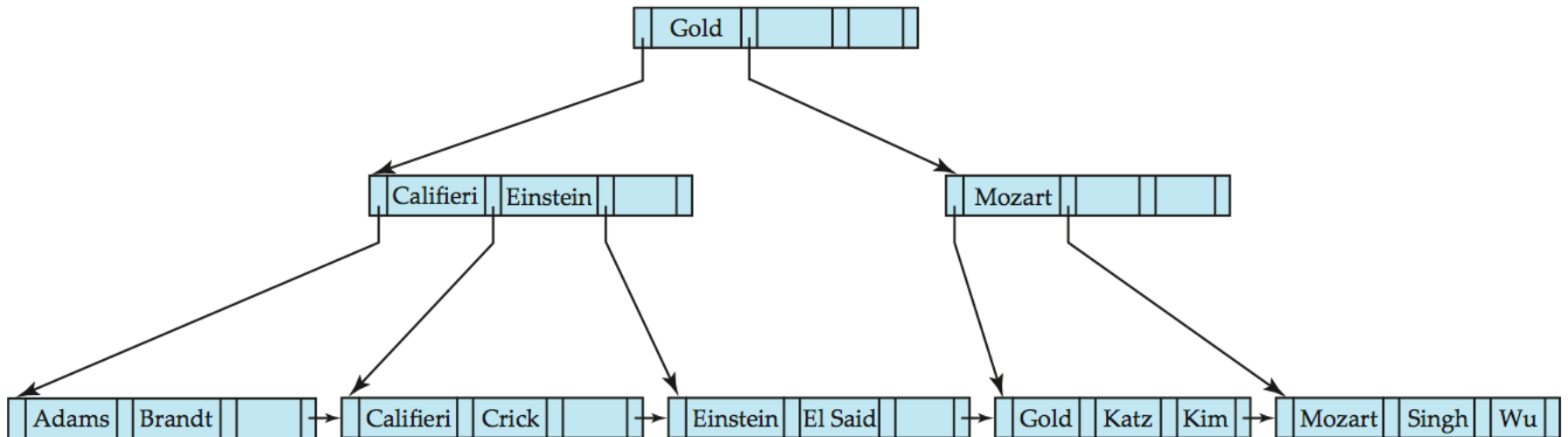


B⁺-Tree before and after insertion of “Adams”

Examples of B⁺-Tree Deletion

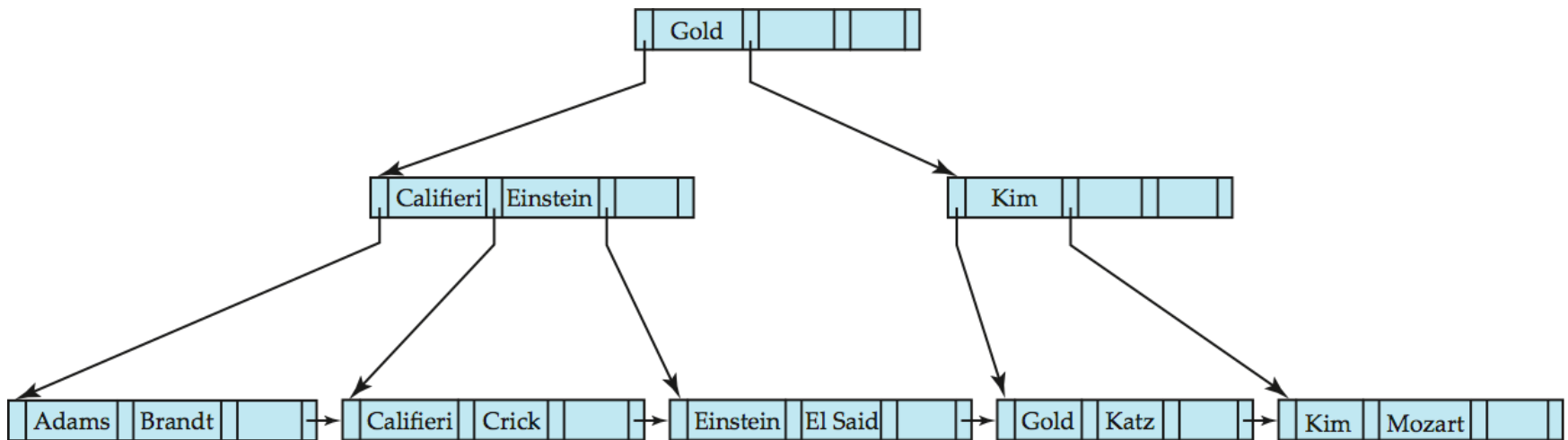


Before and after deleting “Srinivasan”



- Deleting “Srinivasan” causes merging of under-full leaves

Examples of B⁺-Tree Deletion (Cont.)



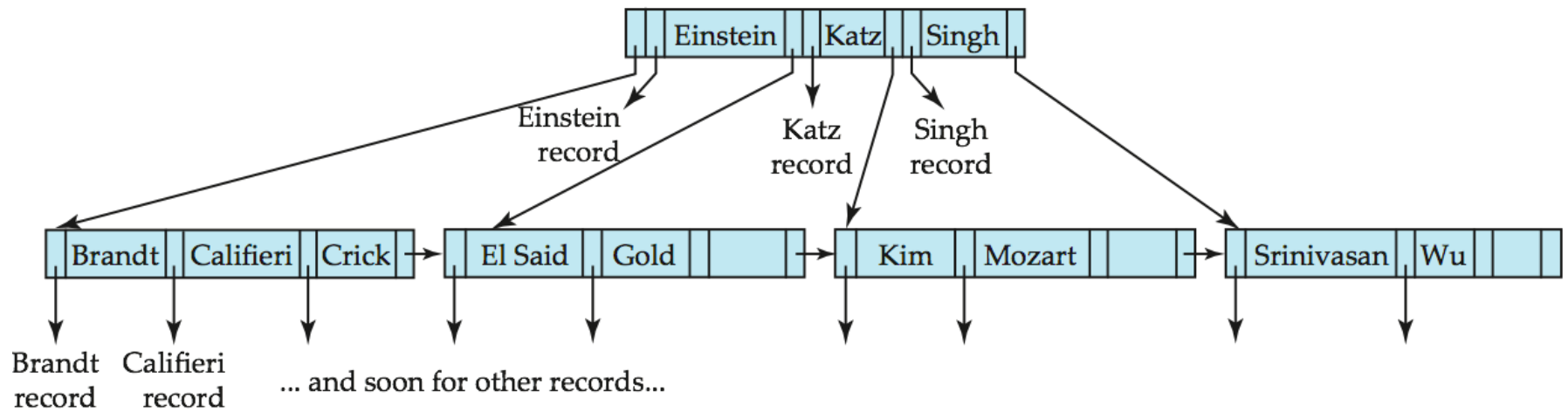
Deletion of “Singh” and “Wu” from result of previous example

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

B-Tree Index Files

- **Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.**
- **Search keys in non-leaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a non-leaf node must be included.**

B-Tree Index File Example



B-Tree Index Files (Cont.)

- **Advantages of B-Tree indices:**
 - May use less tree nodes than a corresponding B⁺-Tree.
 - Sometimes possible to find search-key value before reaching leaf node.
- **Disadvantages of B-Tree indices:**
 - Only small fraction of all search-key values are found early
 - Insertion and deletion more complicated than in B⁺-Trees
 - Implementation is harder than B⁺-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

Multiple-Key Access

- Use multiple indices for certain types of queries.

- Example:

select *ID*

from *instructor*

where *dept_name* = “Finance” and *salary* = 80000

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

Bitmap Indices

- **Bitmap indices are a special type of index designed for efficient querying on multiple keys**
- **Applicable on attributes that take on a relatively small number of distinct values (**Low Cardinality Column**)**
 - **E.g.**
Gender
Country
State,
Product_category
Student_major etc..
- **A bitmap is simply an array of bits**

Bitmap Indices (Cont.)

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

record number	<i>ID</i>	<i>gender</i>	<i>income_level</i>	Bitmaps for <i>gender</i>		Bitmaps for <i>income_level</i>	
				m	10010		
				f	01101	L1	10100
0	76766	m	L1			L2	01000
1	22222	f	L2			L3	00001
2	12121	f	L1			L4	00010
3	15151	m	L4			L5	00000
4	58583	f	L3				

Bitmap Indices

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

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

Bitmaps for *gender*

m	10010
f	01101

Bitmaps for *income_level*

L1	10100
L2	01000
L3	00001
L4	00010
L5	00000

Bitmap Indices (Cont.)

- Queries are answered using bitmap operations
 - Intersection (and)
 - Union (or)
 - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap

- Males with income level L1: 10010
AND 10100
= 10000

In AND condition, only 1 AND 1 is 1, else 0

- Male or income level L1: 10010
OR 10100
= 10110

In OR condition, only 0 OR 0 is 0, else 1

B-TREE AND BITMAP INDEX REPRESENTATION

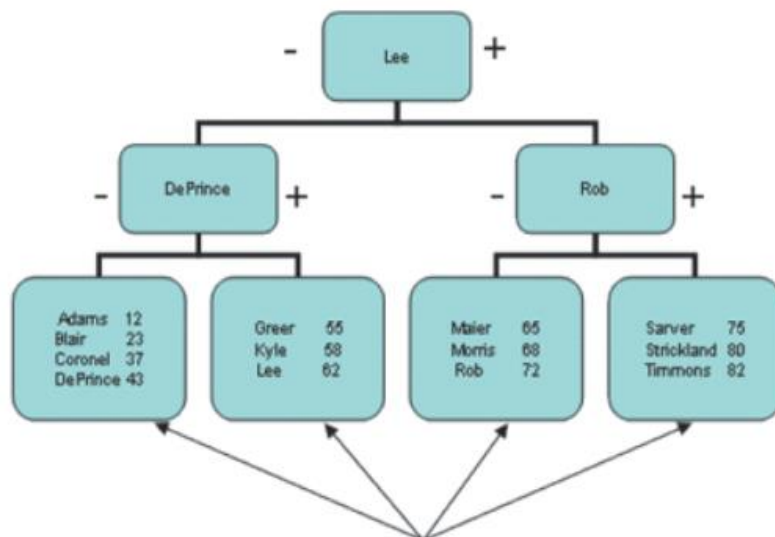
B-Tree index is used in columns with high data sparsity—that is, columns with many different values relative to the total number of rows.

CUSTOMER TABLE

CUS_ID	CUS_LNAME	CUS_FNAME	CUS_PHONE	REGION_CODE
12	Adams	Charlie	4533	NW
23	Blair	Robert	5426	SE
37	Coronel	Carlos	2358	SW
43	DePrince	Albert	6543	NE
55	Greer	Tim	2764	SE
58	Kyle	Ruben	2453	SW
62	Lee	John	7895	NE
65	Maier	Jerry	7689	NW
68	Morris	Steve	4568	NW
72	Rob	Pete	8123	NE
75	Sarver	Lee	8193	SE
80	Strickland	Tomas	3129	SW
82	Timmons	Douglas	3499	NE

Bitmap index is used in columns with low data sparsity—that is, columns with few different values relative to the total number of rows.

B-tree Index
On CUS_LNAME



Leaf objects contain index key and pointers to rows in table. Access to any row using the index will take the same number of I/O accesses. In this example, it would take four I/O accesses to access any given table row using the index: One for each index tree level (root, branch, leaf object) plus access to data row using the pointer.

Bitmap Index
On REGION_CODE

Region	NE	NW	SE	SW		
Bit 1	Bit 2	Bit 3	Bit 4	Bit ...	Bit ...	
0	1	0	0			
0	0	1	0			
0	0	0	1			
1	0	0	0			
0	0	1	0			
0	0	0	1			
1	0	0	0			
0	1	0	0			
0	1	0	0			
1	0	0	0			
0	0	1	0			
0	0	0	1			
1	0	0	0			

←One byte

In the bitmap index, each bit represents one region code. In the first row, bit number two is turned on, thus indicating that the first row region code value is NW.

REGION_CODE = 'NW'

Each byte in the bitmap index represents one row of the table data. Bitmap indexes are very efficient with searches. For example, to find all customers in the NW region, the DBMS will return all rows with bit number two turned on.

INDEX CONSIDERATION

- **Primary Key columns are by default generate UNIQUE INDEX**
- **Columns which have unique values (e.g. Email address, Phone number, SSN), suitable for UNIQUE INDEX**
- **Columns that are used for Table Joins, e.g. Foreign Keys**
- **Columns that are used frequently in WHERE clause**
- **Column that are used in ORDER BY clause**
- **Don't create index on column(s) that are frequently updated**
- **Don't crate index on small tables**

- **UNIQUE INDEX:** Suitable for columns which have unique values, e.g. EMAIL

```
CREATE UNIQUE INDEX ind_emp_email ON AP_EMPLOYEE (EMAIL);
```

- **NON-UNIQUE INDEX:** Also, called **secondary index**, column values are not unique, but a wide range of values (high cardinality/sparsity/ high distinct values)

```
SELECT * FROM AP_EMPLOYEE  
WHERE LAST_NAME='Grant';
```

```
CREATE INDEX idx_last_name ON AP_EMPLOYEE(LAST_NAME);
```

COMPOSITE INDEX : Index created on multiple columns.

```
SELECT * FROM AP_EMPLOYEE  
WHERE JOB_ID ='IT_PROG' and DEPARTMENT_ID =60;
```

```
CREATE INDEX idx_job_department ON AP_EMPLOYEE(JOB_ID,  
DEPARTMENT_ID);
```

```
DROP INDEX idx_job_department;
```

```
ALTER INDEX idx_job_department REBUILD ONLINE;
```

- **FUNCTION BASED INDEX:** Used when filter condition is frequently used with column expression or function

```
SELECT * FROM AP_EMPLOYEE  
WHERE UPPER(LAST_NAME)='GRANT';
```

```
CREATE INDEX IDX_UPPER_LAST_NAME ON AP_EMPLOYEE(UPPER(LAST_NAME));
```

BITMAP INDEX: Used when column has few distinct values in comparison to total number of records in table (e.g. Gender, Martial Status, State)

```
SELECT * FROM AP_EMPLOYEE  
WHERE JOB_ID ='IT_PROG' and DEPARTMENT_ID=60;
```

```
SELECT DISTINCT JOB_ID FROM AP_EMPLOYEE;
```

```
CREATE BITMAP INDEX IDX_JOB_ID ON AP_EMPLOYEE(JOB_ID );
```

```
SELECT DISTINCT DEPARTMENT_ID FROM AP_EMPLOYEE;
```

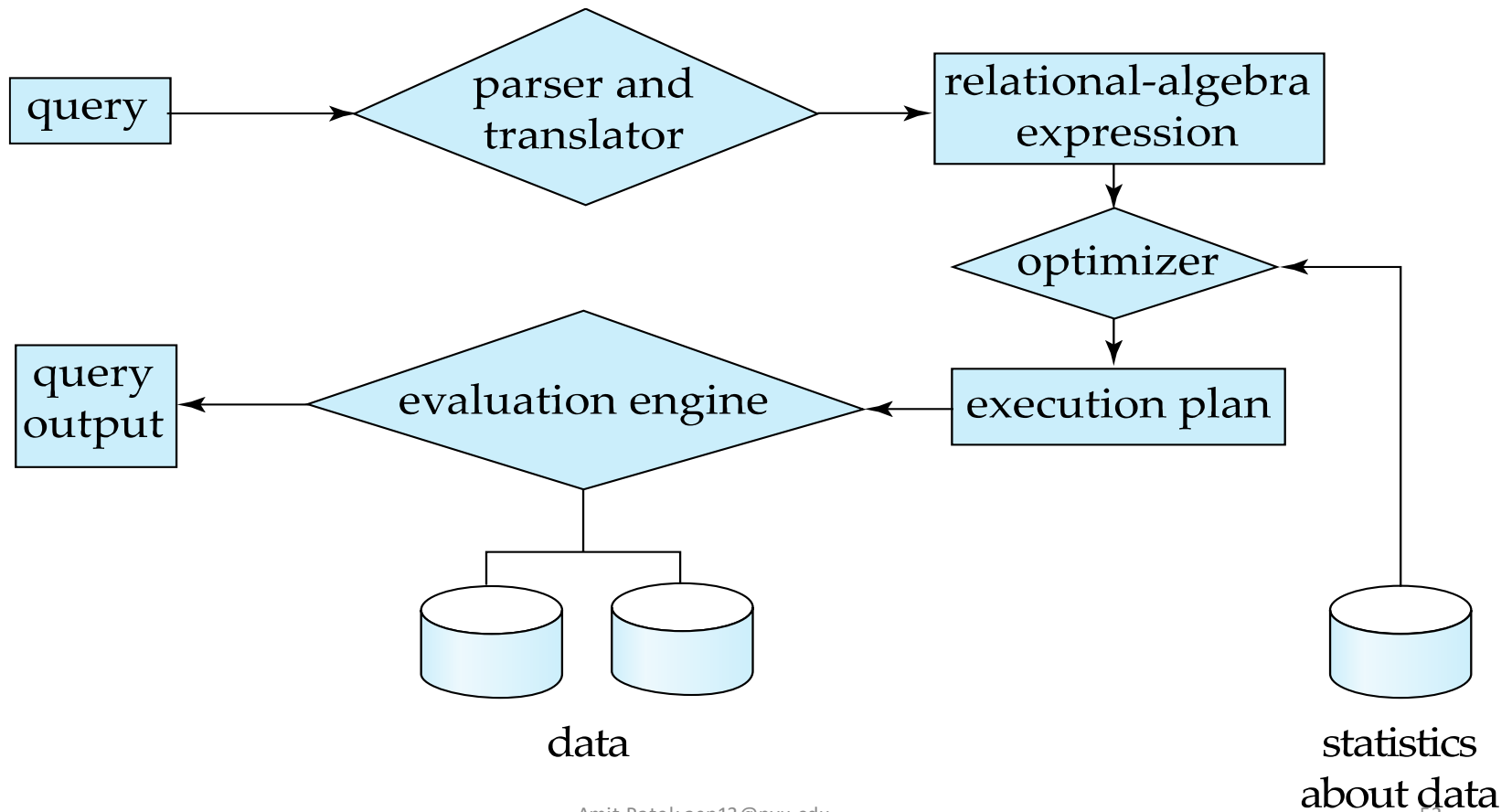
```
CREATE BITMAP INDEX IDX_DEPARTMENT_ID ON AP_EMPLOYEE(DEPARTMENT_ID );
```

-- Dictionary Query to find list of indexes created for a table.

```
SELECT * FROM USER_INDEXES  
WHERE TABLE_NAME='AP_EMPLOYEE';
```

Basic Steps in Query Processing and Optimization

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing (Cont.)

- **Parsing and translation**
 - Parser checks syntax, verifies relations, access to database object
 - Translate the query into its internal form. This is then translated into relational algebra/calculus.
- **Optimization**
 - Create and choose best optimal execution plan based upon data and their statistics
- **Evaluation**
 - The query-execution engine takes a query-execution plan, executes that plan, and returns the answers to the query.

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
 - E.g., $\sigma_{salary < 75000}(\pi_{salary}(instructor))$ is equivalent to $\pi_{salary}(\sigma_{salary < 75000}(instructor))$

*SELECT salary FROM INSTRUCTOR
WHERE salary < 75000;*

- Each relational algebra operation can be evaluated using one of several different algorithms
 - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an *evaluation-plan/execution-plan*.
 - E.g., can use an index on *salary* to find instructors with salary < 75000,
 - or can perform complete relation scan and discard instructors with salary ≥ 75000 [assumption, most of instructor's salary is < 75000]

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost (in terms of query elapsed time)

Cost is estimated based upon:

Filter condition

Sorting columns and their orders,

Table joins,

Subqueries

Indexes on table and type of indexes

Optimizer Statistics:

- Total number of records in table
- Average size of records in table ,
- Min and Max value of column data
- Distinct values of key in search (filter),

Optimizer Statistics : statistical metadata of data that highly influence the performance of the query.

Statistics simply are a form of dynamic metadata that assists the query optimizer in making better decisions. For example, if there are only a dozen rows in a table, then there's no point going to an index to do a lookup; you will always be better off doing a full table scan.

Table and Index Statistics Collection

-- sampling statistics of table for 50% of records

ANALYZE TABLE ap_employee ESTIMATE STATISTICS SAMPLE 50 PERCENT;

-- sampling statistics of table for 20 records

ANALYZE TABLE ap_employee ESTIMATE statistics SAMPLE 20 ROWS;

-- collecting statistics of entire table

ANALYZE TABLE ap_employee COMPUTE STATISTICS;

-- collecting statistics of an index

ANALYZE INDEX idx_last_name COMPUTE STATISTICS;

-- collecting statistics of table and all associated indexes of the table

ANALYZE TABLE ap_employee COMPUTE STATISTICS FOR ALL INDEXES;

Table and Index Statistics Collection

begin

```
DBMS_STATS.GATHER_TABLE_STATS ( ownname => 'APATEL',  tabname =>
'AP_EMPLOYEE', estimate_percent => 1);
```

end;

/

```
create index idx_last_name on ap_employee(last_name);
```

begin

```
DBMS_STATS.GATHER_INDEX_STATS(ownname => 'APATEL', indname =>
'IDX_LAST_NAME');
```

end;

/

begin

```
DBMS_STATS.GATHER_TABLE_STATS ( ownname => 'APATEL',  tabname =>
'AP_EMPLOYEE', estimate_percent => 1, cascade=>true );
```

end;

/

DATA DICTIONARY

- One of the most important parts of an **Oracle database** is its **data dictionary**, which is a read-only set of system tables that provides information about the **database**.

List of tables

```
select * from user_tables where table_name like 'AP%';
```

List of columns

```
select * from user_tab_columns where table_name='AP_EMPLOYEE';
```

List of columns constraints

```
select * from user_constraints where table_name='AP_EMPLOYEE';
```

- **Table statistics**

```
select * from user_tab_statistics;
```

- **Column statistics**

```
select * from user_tab_col_statistics  
where table_name='AP_EMPLOYEE';
```

- **Index statistics**

```
select * from user_ind_statistics  
where table_name='AP_EMPLOYEE';
```

Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
 - Note: order of tuples is irrelevant
 - we don't care if they generate different results on databases that violate integrity constraints

a)

```
SELECT ename, job, deptno  
FROM emp  
WHERE deptno in (10,30);
```

```
c) select ename, job, deptno  
from emp  
where deptno=10 or deptno=30;
```

b)

```
SELECT ename, job, deptno  
FROM emp where deptno 10  
UNION  
SELECT ename, job, deptno  
FROM emp where deptno 30;
```

Cost-Based Optimization

- Consider finding the best join-order for $r_1 \ r_2 \ \dots r_n$.
- 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!
- No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2, \dots r_n\}$ is computed only once and stored for future use.

