Introduction to Database System

2019

Sophie Ammann

Lecture 1

1.1 Terminology :

- **Data** : facts, basis for reasoning, useful or irrelevant (only 10% of data is useful). Must be *processed* to be meaningful. “Everything that can be mathematicaly defined is data”
- **Information** : meaning, relevant to the problem
- **Database (DB)** : large, integrated, structured collection of data
- **Database Management System (DBMS)** : software system designed to store, manage and facilitate access to databases (connected bridge btw user and database)
- **Data model** : collection of concepts for describing data (relational, hierarchical, graph,…)
- **Relational data model** : set of records represented by a table.

1.2 Relational data model

- **Relation** : table with row and columns
- **Schema** : Describes the structure (columns) of a relation

1.3 Logical and physical data independence

Data independence is the ability to change the schema at one level of the database system without changing the schema at the next higher level

- **Logical data independence** : capacity to change the conceptual schema without changing the user views
- **Physical data independence** : capacity to change the internal schema without having to change the conceptual schema or user views

---

Lecture 2 : ER model

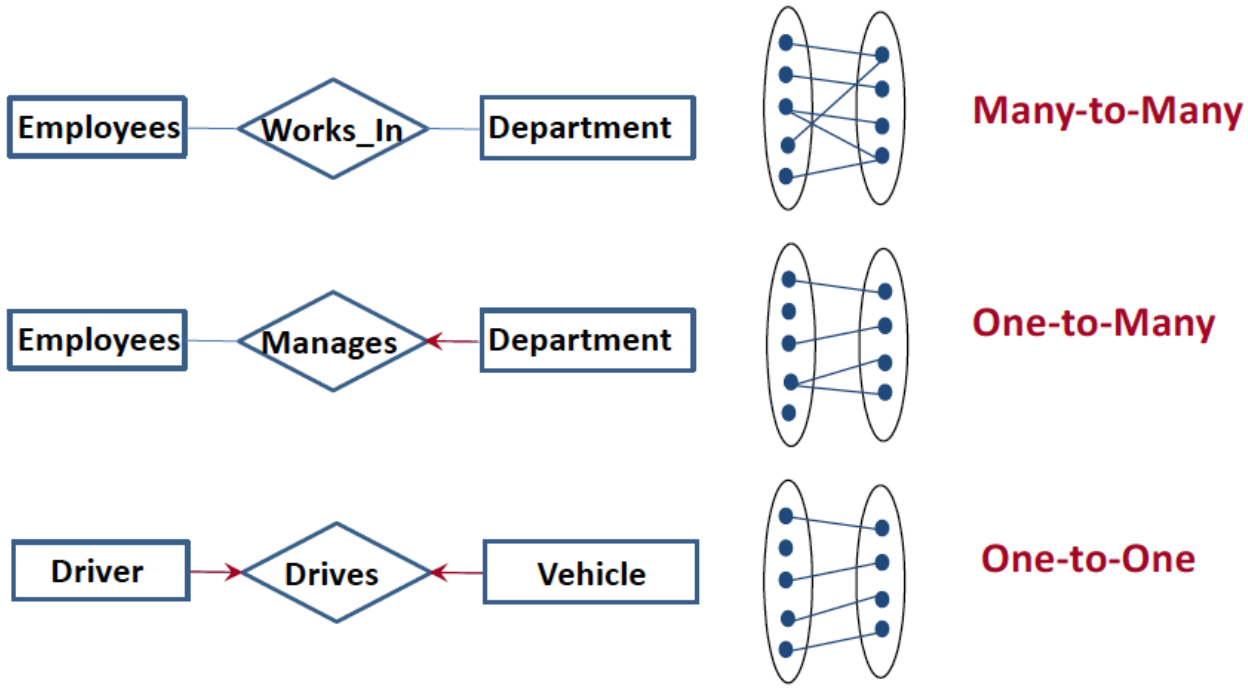
2.1 Conceptual design

ER model = entity-relationship model

- **Entity** : real-world object, distinguishable from other objects.
  Attributes are used to describe an entity. (defined in a domain)
- **Entity set** : A collection of similar entities. E.g., all employees
  Key : each entity set has a key
- **Relationship** : association between entities, can have their own attributes.

2.2 Constraints

2.2.1 Key constraints



- Many-to-many :
  an employee can work in many departments; a department can have many employees

- One-to-many :
  each department has at most one manager

- One-to-one :
  each driver can drive at most one vehicle and each vehicle will have at most one driver.

2.2.2 Participation constraints

Total Participation

Means: “exactly one”

Partial Participation

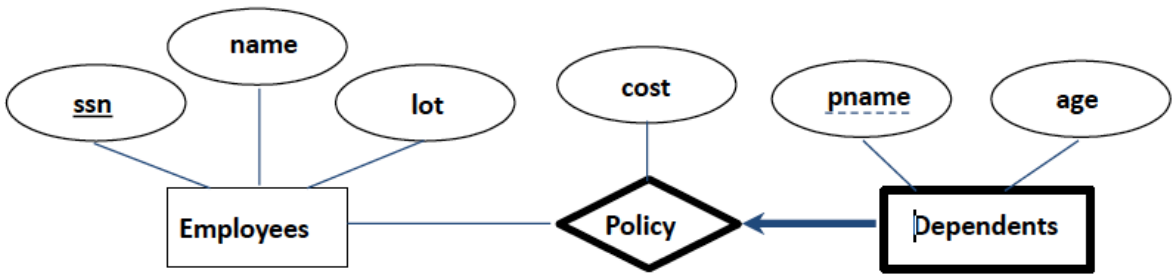
- Total participation :
  Every employee should work in at least one department.
  Every department should have at least one employee.

- Participation + key constraint :
  There could be some employees who are not managers.
  Every department should have at least one manager.

- Partial participation :
  There could be some customers who do not buy any products.
  There could be some products which are not bought by any customers.

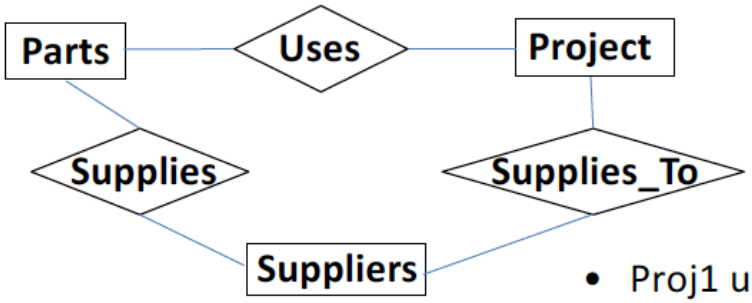
2.3 Weak entities

Entity that can be identified uniquely only by considering the primary key of another entity (owner).



There has to be a one-to-many relationship (one owner, many weak entities).
The weak entity set must have total participation

2.4 Ternary relationships

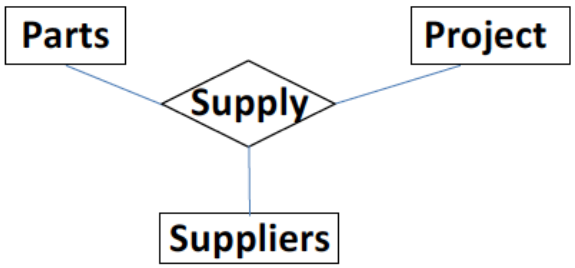


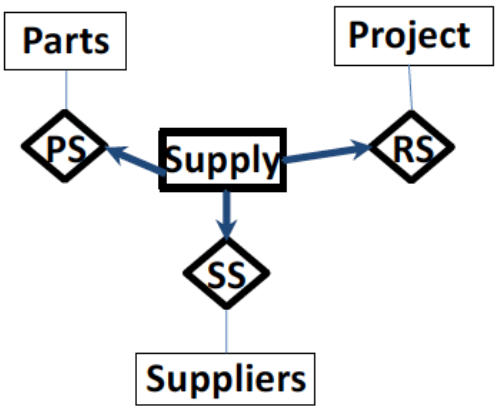
- Proj1 uses Part1
- Supp1 supplies to Proj1
- Supp1 supplies Part1

does this imply

- Proj1 uses Part1 supplied by Supp1

No

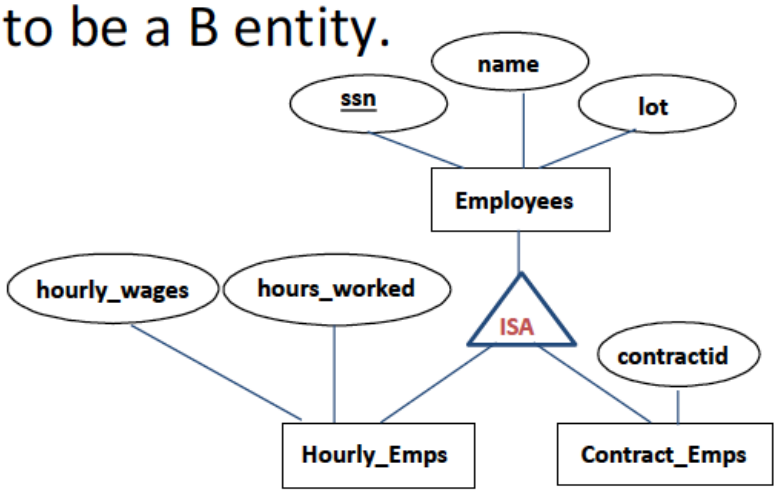




2.5 ISA ('is a') hierarchies

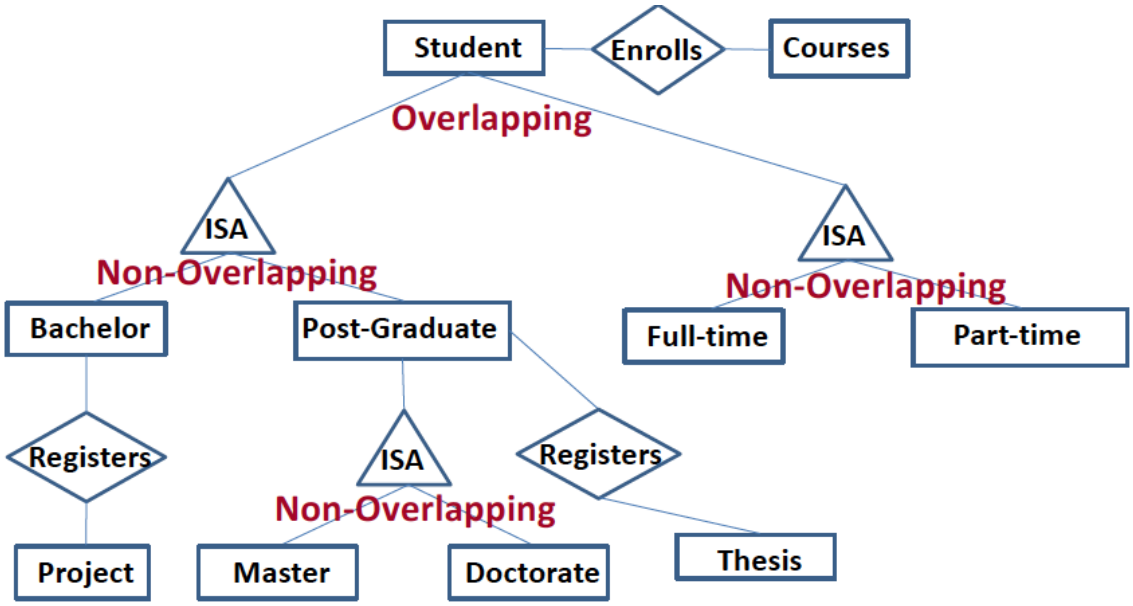
Attributes inherited

- If we declare A **ISA** B, every A entity is also considered to be a B entity.



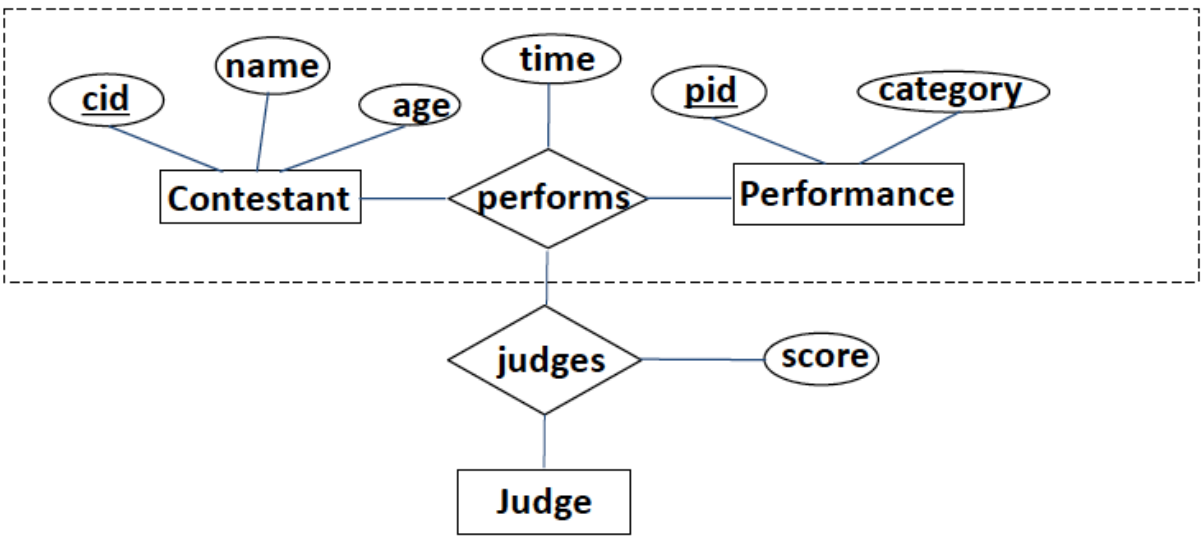
2.5.1 Constraints :

- **Overlap cosntraints** : *Can a student be a master as well as a doctorate entity? (Allowed/Disallowed)*
- **Covering constraints** : *Does every Employees entity also have to be an Hourly_Emps or a Contract_Emps entity? (Yes/No)*



2.6 Aggregation :

Can treat a relationship set as an entity set.



Lecture 3 : Data model

SQL = Structured Query Language

3.1 Creating relations in SQL

- `CREATE TABLE <name> ( <field> <domain>, … )`

- ```
  INSERT INTO <name> (<field names>)
       VALUES (<field values>)
  ```

- ```
  DELETE FROM <name>
         WHERE <condition>
  ```

- ```
  UPDATE <name>
     SET <field name> = <value>
   WHERE <condition>
  ```

- ```
  SELECT <fields>
    FROM <name>
   WHERE <condition>
  ```

3.2 Key

- **superkey** :
  Set of attributes for which no two distinct tuples can have same values in all key fields . Can be all the attributes, or just a few.
- **key** :
  minimal superkey (no subset of the fields is a superkey)
- **candidate key** :
  if there are multiple keys, then each of them is referred to as candidate key
- **primary key** :
  one of the candidate key is chosen

```
CREATE TABLE Students
        (sid CHAR(20),
         name CHAR(20),
         login CHAR(10),
         age INTEGER,
         gpa FLOAT,
          primary key(sid))

CREATE TABLE Person
        (ssn CHAR(9),
         name CHAR(20),
         licence# CHAR(10),
         primary key(ssn),
         unique(licence#))
```

Example :

- *UNIQUE* keyword indicates a candidate key that is not the primary key.
- *PRIMARY* keyword indicates the primary key.

3.3 Integrity constraints (ICs)

- **IC** = condition that must be true for any instance of the database (the domain constraints)
- **legal instance** : satisfies all the specified ICs.
  //TODO …

Lecture 4 : Relational algebra

4.1 Introduction

relation algebra = operational, useful for representing execution plans

- query is applied to *relation instances*, the result is also a *relation instance*.
- Schema of the input relations for a query is **fixed** (but query will run over any legal instance)
- Schema of output (result) of a given query is also **fixed**

4.2 Basic operations

- **selection** σ :
  selects *rows* from a relation (horizontal)

↔ *WHERE* in SQL

example :

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$$\sigma_{rating<9 \wedge age>50}(S2)$$

Output

sid	sname	rating	age
31	Lubber	8	55.5

- **projection** π :
  retains only wanted *columns* from a relation (vertical)
  ↔ *SELECT* in SQL

  example :

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

$$\pi_{sname,rating}(S2)$$

Output

sname	rating
yuppy	9
Lubber	8
guppy	5
Rusty	10

- **cross-product** $\times$:
  combines two relations

  example :

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

$$S1 \times R1$$

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

$$\rho_{1 \to sid1, 5 \to sid2}(S1 \times R1)$$

sid1	sname	rating	age	sid2	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

- **set-difference** $-$:
  tuples in R_1 but not in R_2

  R_1 and R_2 must be *union compatible* (same number of fields and fields of same type)

  example :

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

S1 − S2

sid	sname	rating	age
22	Dustin	7	45.0

S2 − S1

sid	sname	rating	age
28	yuppy	9	35.0
44	guppy	5	35.0

- **union** $\cup$:
  tuples in R_1 and/or in R_2

  R_1 and R_2 must be *union compatible* (same number of fields and fields of same type)

  example :

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

S2

sid	sname	rating	age
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

S1 ∪ S2

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

4.3 Renaming operator ρ

renames the list of attributes :
$< oldname > \longrightarrow < newname >$
or
$< position > \longrightarrow < newname >$
, where *position* starts at 1!

example :

Boats

bid	bname	color
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$$\rho_{bname \to boatname, color \to boatcolor}(Boats)$$

bid	boatname	boatcolor
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

$$\rho_{2 \to boatname, 3 \to boatcolor}(Boats)$$

4.4 Compound operators

4.4.1 Natural join ⋈

idea :

- compute $R \times S$
- select rows where attributes that appear in both relations have equal values
- project all unique attributes and one copy of the common ones

example :

$$\pi_{S1.sid, sname, \ldots}(\sigma_{S1.sid = R1.sid}(S1 \times R1))$$

S1

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

R1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

sid	sname	rating	age	sid	bid	day
22	Dustin	7	45.0	22	101	10/10/96
~~22~~	~~Dustin~~	~~7~~	~~45.0~~	~~58~~	~~103~~	~~11/12/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~22~~	~~101~~	~~10/10/96~~
~~31~~	~~Lubber~~	~~8~~	~~55.5~~	~~58~~	~~103~~	~~11/12/96~~
~~58~~	~~Rusty~~	~~10~~	~~35.0~~	~~22~~	~~101~~	~~10/10/96~~
58	Rusty	10	35.0	58	103	11/12/96

$S1 \bowtie R1$

sid	sname	rating	age	bid	day
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

4.4.2 Condition join or theta-join $\bowtie_c$

$R \bowtie_c S = \sigma_c(R \times S)$

4.4.3 Equi-join

special case of the theta-join : condition *c* contains only conjunction of equality conditions

example :
good way of finding all pairs of sailors in $S_1 \times S_2$ who have the same age :

$$\sigma_{sid_1 < sid_2}(S_1 \bowtie_{age=age_2} \rho_{age \to age2, sid \to sid2}(S_2))$$

4.4.3 Division

A/B contains all x tuples such that for every tuple in B, there is an (x, y) tuple in A.
(B is a proper subset of A)

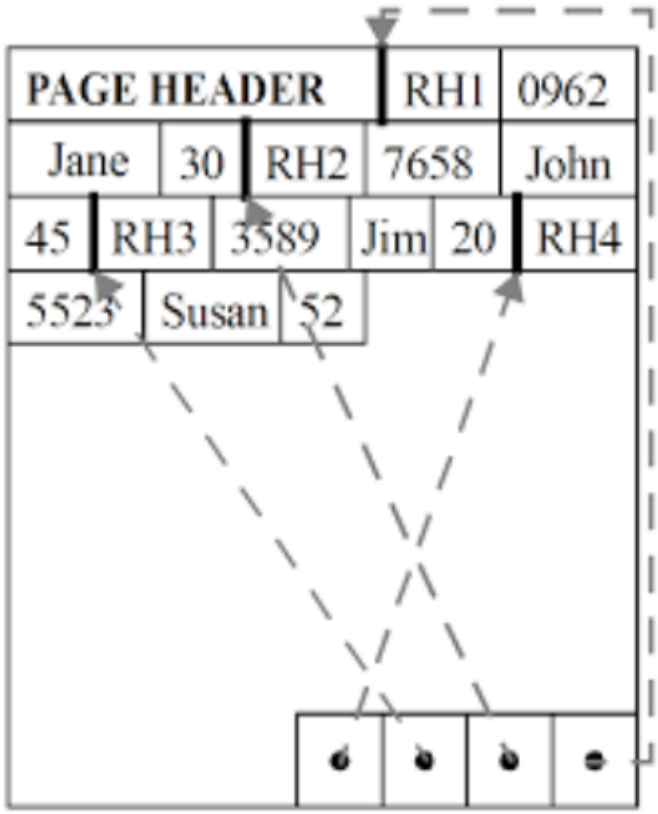
Lecture 5 : Storage, files and indexing

5.1 Introduction

file and access layer :

- retrieve one particular record (using record id) : **point access**
- retrieve a range of records (satisfying some conditions) : **range access**
- retrieve all records : **scan**

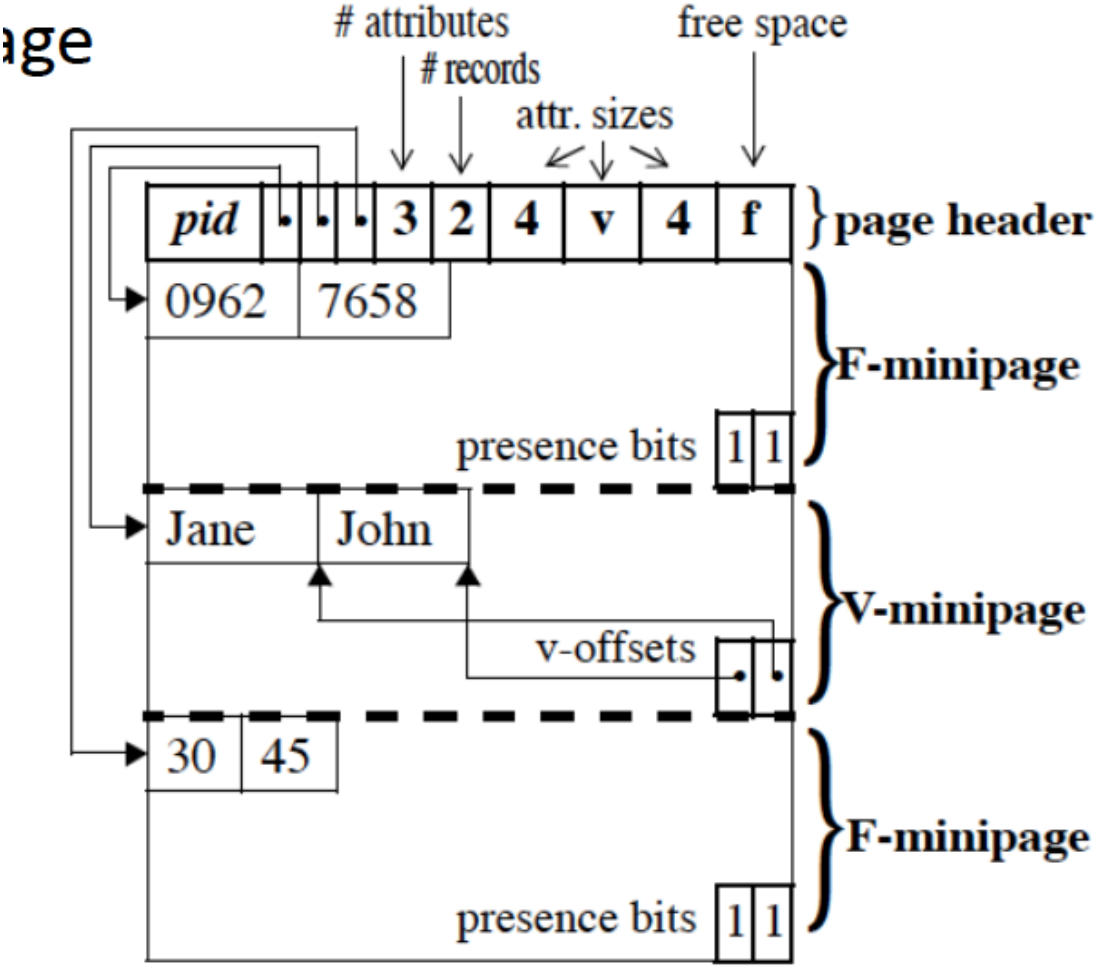
5.2 N-ary storage model (flash page)



- **page** : collection of slots
- **slot** : one record
- **rid** : record id = <page id, slot#>, should be unique

5.3 PAX

PAX = partition attributes across



5.3 Indexing

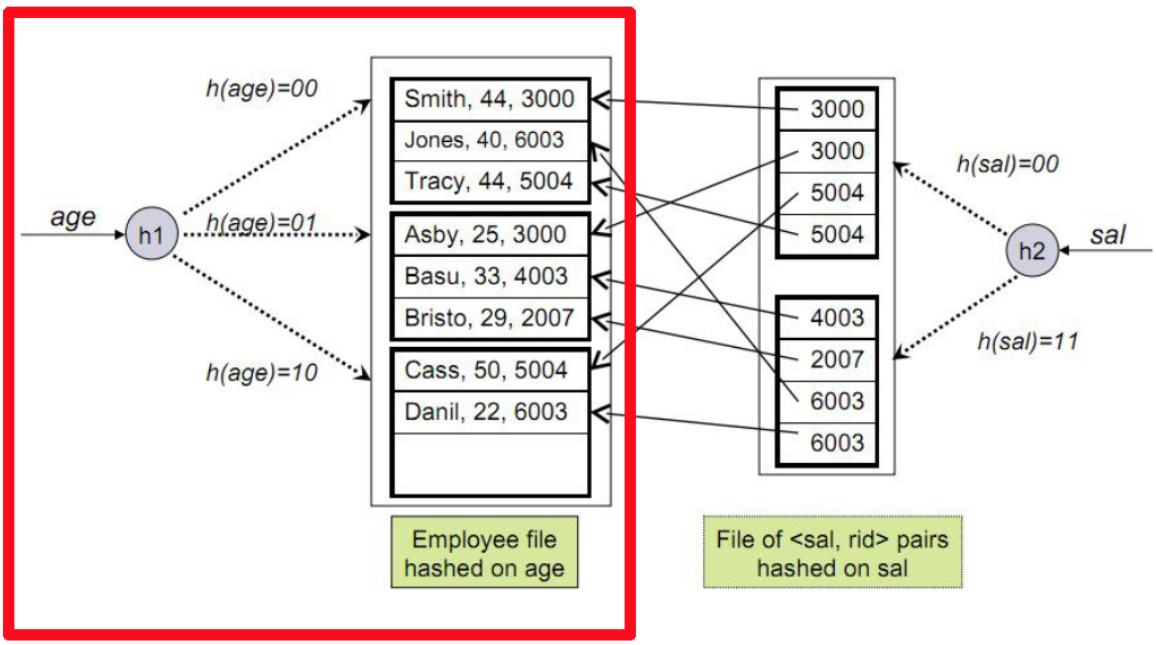
- an **index** :
  An index is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations. An index allows us to efficiently retrieve all records that satisfy search conditions on the search key fields of the index.
- a **key** :
  indexing field
- a **data entry** :
  refers to the records stored in an index file.
  A data entry with search key value k, denoted as $k*$, contains enough information to locate (one or more) data records with search key value k.

5.3.1 Data entry representation

three alternative representations with search key value k :

1. data entry with $k*$ is an actual data record

Alternative 1: Example

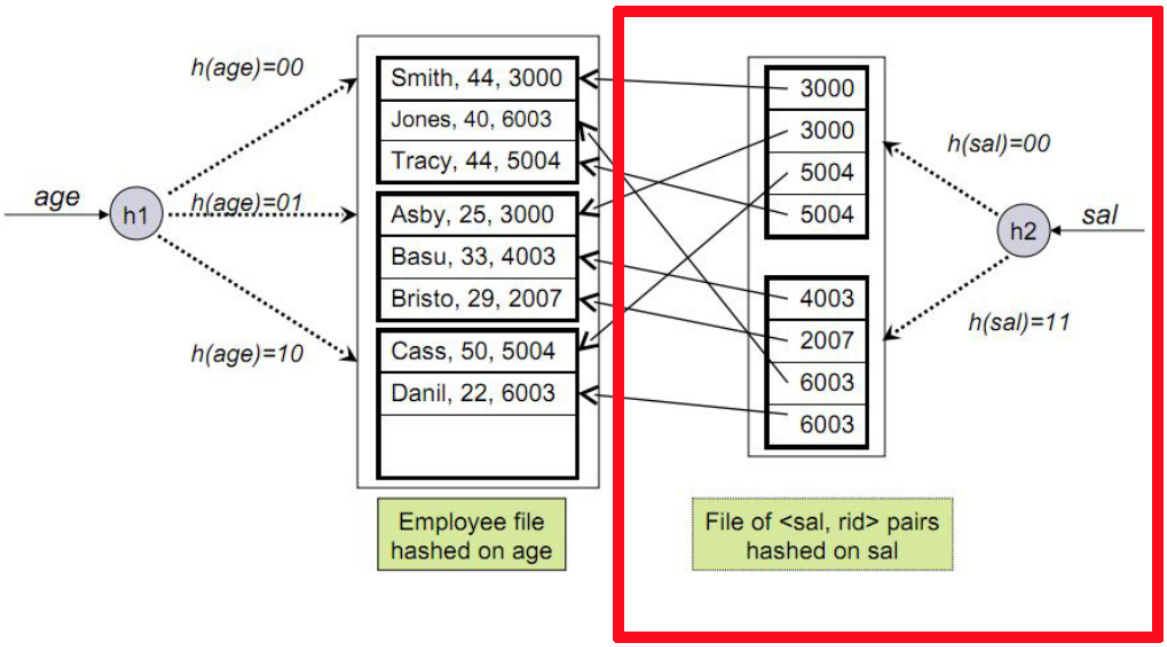


(image : Alt. 1, hash-based indexing)

At most one index can use Alt. 1. Efficient but can be expensive to maintain (insertions and deletion modify the data file)

2. data entry is a (k, rid) pair
3. data entry is a $(k, rid - list)$ pair

Alternative 2: Example



(image : Alt. 2, hash-based indexing)

Notes :

Alt. 2 and Alt. 3, which contain data entries that point to data records, are independent of

the file organization that is used for the indexed file.
Easier to maintain than Alt. 1.

5.3.2 Primary and secondary indexes

- **primary index** : index on a set of fields that includes the primary key
- **secondary index** : all the other indexes
- *Note* : a primary index is guaranteed not to contain duplicates, but an index on other (collections of) fields can contain duplicates.

5.3.3 Clustering

- **clustered index** : index whose data entries are sorted and ordered the same way as the file records. One index entry per distinct value, sparse index
- **unclustered index** : not the same sorting

5.3.4 Dense vs Sparse

- **dense** : at least one entry per key value
  Alt. 1 is a dense indexing

- **sparse** : an entry

- **summary** :

Index Classification: Summary

Type of Index	Indexing Field	File physically sorted on indexing field?	Index Entries	Index Pointers	Sparse or Dense?
Primary	Key	Yes	One per block	Block anchor	Sparse
Clustering	Non-Key	Yes	One per value	Block pointer	Sparse
Secondary Key	Key	No	One per record	Record pointer	Dense
Secondary Non-Key	Non-Key	No	One per record/ value	Record pointer/ Variable length/ indirection	Sparse or Dense

5.3.5 Index data structure

1. hash-based indexing (see **week 8**):
   - hash function:

$$r = record$$
$$h(r.searchKey) = bucket\ for\ record\backslash r$$

    - the records in a file are grouped in **buckets**, where a bucket consists of a **primary page** and, possibly, additional pages linked in a chain.
    - The bucket to which a record belongs can be determined by applying a special function, called a *hash* function, to the search key.

2. tree-based indexing :

    - The data entries are arranged in sorted order by search key value, and a hierarchical search data structure is maintained that directs searches to the correct page of data entries.
    - The **leaf level** (lowest level on the tree) contains the data entries.
    - The average number of children for a non-leaf node is called the **fan-out**
    - A **B+ tree** is a tree where all leafs have equal **height** (path from root to leaf)

5.4 File organisation

5.4.1 heap files

- randomly ordered file
- contains records in no particular order, search based on *rid*
- the file manager must keep track of the pages allocated for the file

5.4.2 sorted files

- sorted file on a certain attribute
- search done on file-ordering attribute

5.4.3 cost

Assumptions :

- IO is the dominating cost

- consider average case

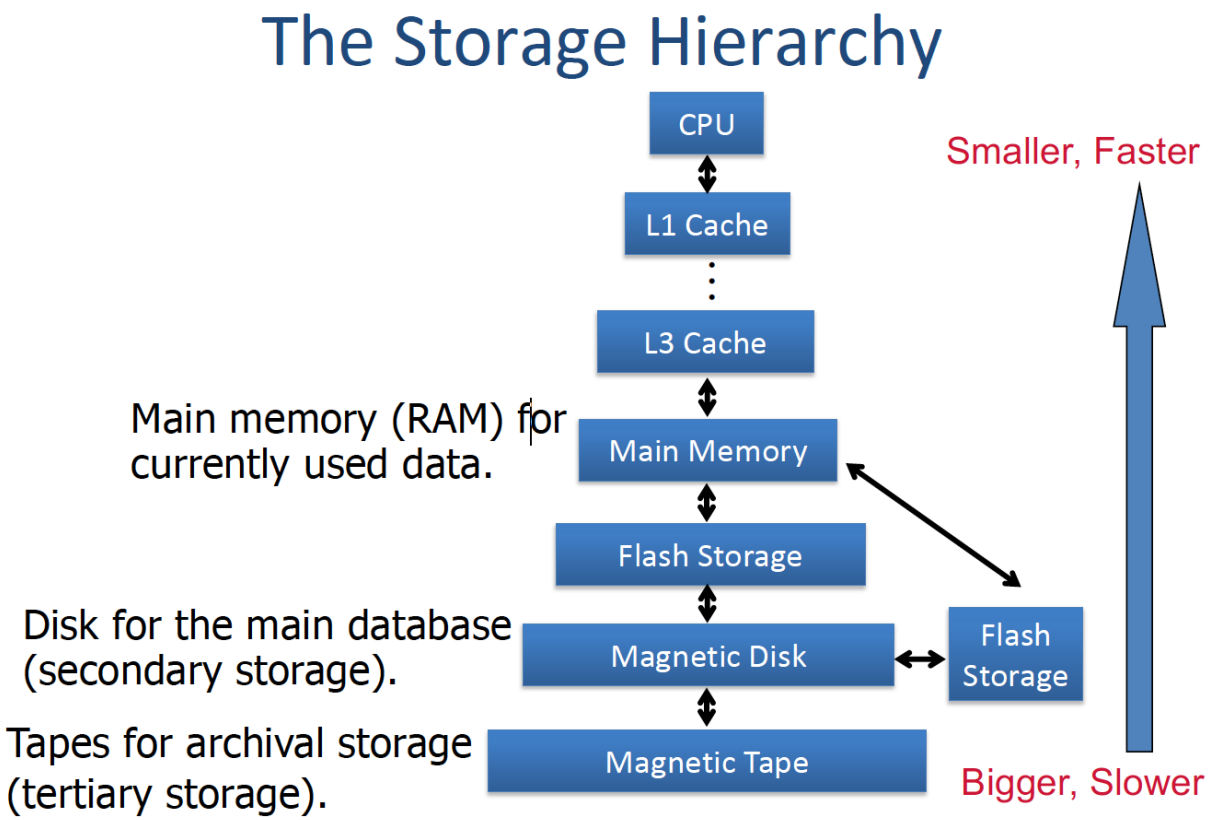
Cost of Operations (in # of I/O's)

B: Number of data pages

	Heap File	Sorted File	notes...
Scan all records	B	B	
Equality Search	0.5B	$\log_2 B$	assumes exactly one match!
Range Search	B	($\log_2 B$) + (#match pages)	
Insert	2	($\log_2 B$) + 2*(B/2)	must R & W
Delete	0.5B + 1	($\log_2 B$) + 2*(B/2)	must R & W

Lecture 6 : Storage layer

6.1 Remainder

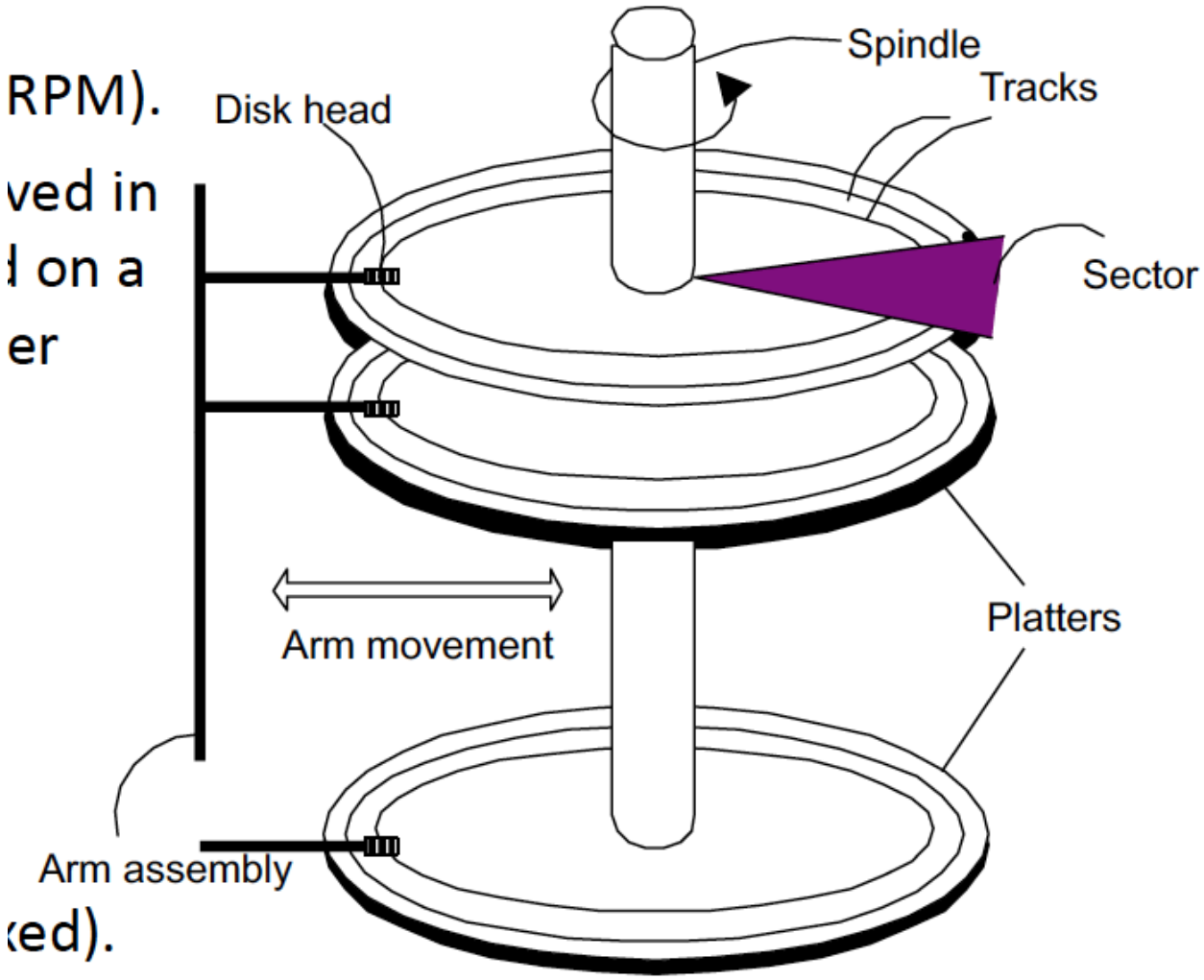


- **DBMS** stores information on disks
- a flash is more expensive than disks
- data is stored in disks

BASIS FOR COMPARISON	MAGNETIC TAPE	MAGNETIC DISK
Basic	Used for backup, and storage of less frequently used information.	Used as a secondary storage.
Physical	Plastic thin, long, narrow strip coated with magnetic material.	Several platters arranged above each other to form a cylinder, each platter has a read-write head.
Use	Idle for sequential access.	Idle for random access.
Access	Slower in data accessing.	Fast in data accessing.
Update	Once data is fed, it can't be updated.	Data can be updated.
Data loss	If the tape is damaged, the data is lost.	In a case of a head crash, the data is lost.
Storage	Typically stores from 20 GB to 200 GB.	From Several hundred GB to Terabytes.
Expense	Magnetic tapes are less expensive.	Magnetic disk is more expensive.

6.2 Disk

6.2.1 Anatomy

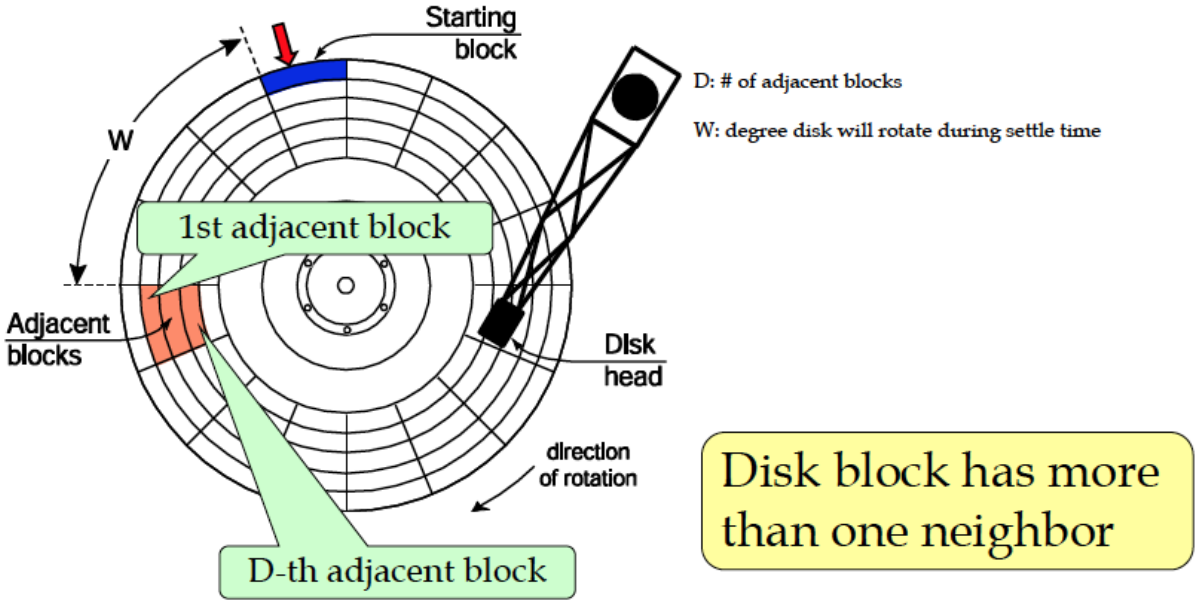


- **Disk head** has a horizontal movement (from the spindle to the side of the platter, arm movement)
- **Platters** spin around the **spindle** (rotation)
- A **track** is a concentric ring on a platter where data is written
- A set of tracks is a called **cylinder**.
- **Block size** : multiple of a **sector size** (fixed)

6.2.2 Access time

- **seek time** : moving arms to position to position disk head on tracks
- **rotational delay** : waiting for block to rotate under head, less than seek time
- **transfer time** : actually moving data to/from disk surface
- **settle time** : part of the seek time, time that the head need to stabilise to the wanted location

6.2.3 Adjacent blocks



Lecture 8 : Hashing

8.1 Pros

Hash-based index are best for equality selections. **Cannot** support range searches.

- Can be beneficial if you have only equality selections
- Very useful in join implementation

8.2 Static hashing

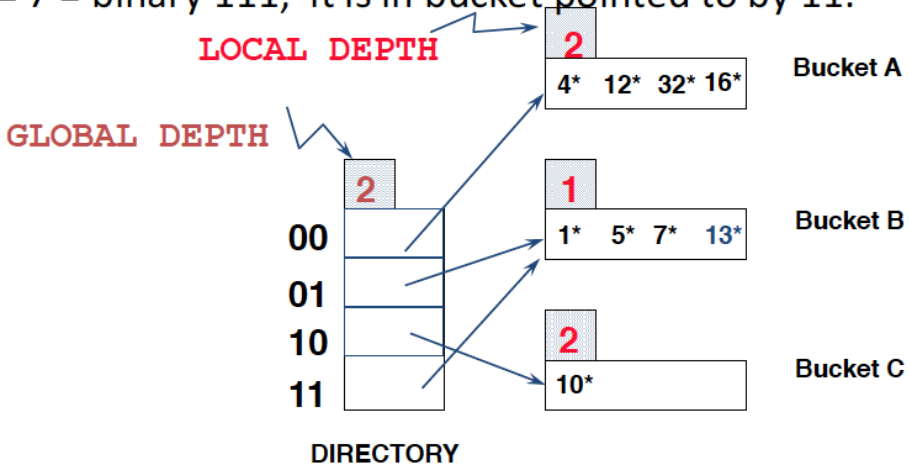
- Hash file is a collection of buckets.
- To search for a data entry, we apply a hash function h to identify the bucket to which it belongs and then search this bucket.
- Let N be the number of buckets, the following hash function works well :
  $h(key) = (a * key + b) \mod N$, where a, b are constants to adjust h.
- Problems :
  Since the number of buckets in a static hashing file is known when the file is created, the primary pages can be stored on successive disk pages. Hence, a search ideally requires just one disk I/O, and insert and delete operations require two I/Os (read and write the page), although the cost could be higher in the presence of overflow pages.

8.3 Extendible hashing

Use a directory of pointers to bucket.s, and double t.he size of the number of buckets by doubling just the directory and splitting only the bucket that overflowed.

- *Example* :

- Bucket for record *r* has entry with index =
  `*global depth*'-least significant bits of h(*r*);
  - If h(*r*) = 5 = binary 101, it is in bucket pointed to by 01.
  - If h(*r*) = 7 = binary 111, it is in bucket pointed to by 11.

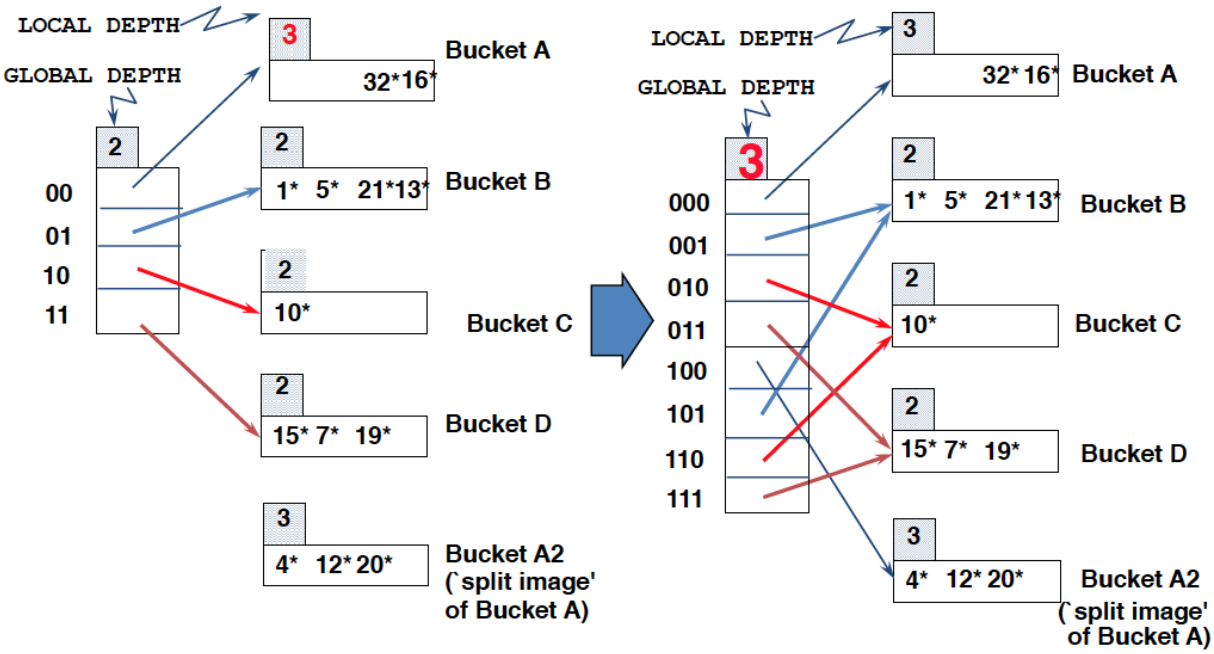


The directory is an array of size 4, where each element points to a bucket.

- **Global depth** (of a directory):
  Max # of bits needed to tell which bucket an entry belongs to.
- **Local depth** (of a bucket):
  The # of bits used to determine if an entry belongs to this bucket.
- **Locate a data entry** :
  Apply the hash function ***h***.
  Pick the LSB to get a number of the size of directory.
  Follow the pointer to the pointed bucket.
- **Insert a data entry** :
  Apply the hash function ***h***.
  Find the pointed bucket with the LSB.
  If the bucket is *non-full*, just insert the new page.
  If the bucket is *full*, allocate a new bucket A2 and redistribute the contents across the old bucket and the new bucket (split image).
  Splitting : look at the bit preceding global depth bit to discriminate between the two buckets.
  Splitting a bucket does not imply to double the directory. It is only the case if local depth > global depth

Insert h(r)=20 (Causes Doubling)

- 20 = 10100



8.4 Linear hashing

8.4.1 Introduction

- no directory
- suppose they are N initial buckets
- many hash functions $h_1, h_2, \ldots$ where $h_i = h(key) mod(2^i N)$.
  $d_0 :=$ number of bits needed to represent N
  $d_i := d_0 + i$
- if h_i maps to M buckets, h_{i+1} maps to $2M$ buckets
- *example :*

$$h_0(key) = h(key) mod(4)$$
$$h_1(key) = h(key) mod(8)$$
$$h_2(key) = h(key) mod(16)$$
$$\ldots$$

8.4.2 Rounds

Current round number : $Level$
number of buckets at the beginning of a round = $N_{level} = N * 2^{Level}$

Linear Hashing (Contd.)

- Algorithm proceeds in `rounds'. Current round number is "*Level*".
- There are N_{Level} (= N * 2^{Level}) buckets at the beginning of a round
- "*Next*" points to next bucket that will be split. When *any* bucket overflows, split the "*Next* " bucket and then increment "*Next* ".
- Buckets 0 to *Next-1* have been split; *Next* to N_{Level} have not been split yet this round.
- Round ends when all initial buckets have been split (i.e. *Next* = N_{Level}).
- To start next round: Level++; Next = 0;

Lecture 8bis : Sorting

8.1 Why sort ?

- If a query needs an answer in sorted order
- First step in *bulk loading* (creation) B+ tree index
- Sort-merge join algorithm involves sorting

8.2 Two-way merge sort

We only need 3 pages in main memory.
The procedure :

- pages are read in one at a times
- the loaded page's records are sorted
- the sorted page can be written out
- in the next passes, pairs of runs (sorted pages) are read and merged to produce runs twice as long.

Let the number of pages be 2^k, then :

pass	number of produced runs	size of a run
0	2^k	one page
1	2^{k-1}	2 pages
2	2^{k-2}	4 pages
…	…	…
k	one	2^k

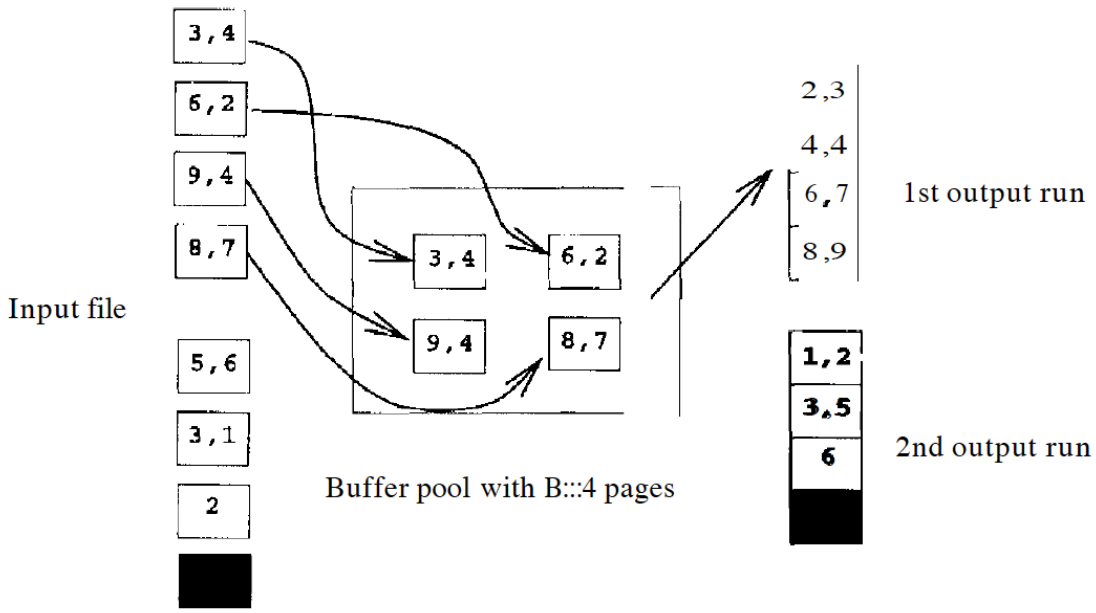
The overall cost for a file of N number of pages :

- for one pass : read the file, process it, write it out $= 2$ I/Os per page
- number of passes : $\lceil \log_2 N \rceil + 1$
- number of pages processed/pass $= N$
- total cost $= 2N(\lceil \log_2 N \rceil + 1)$

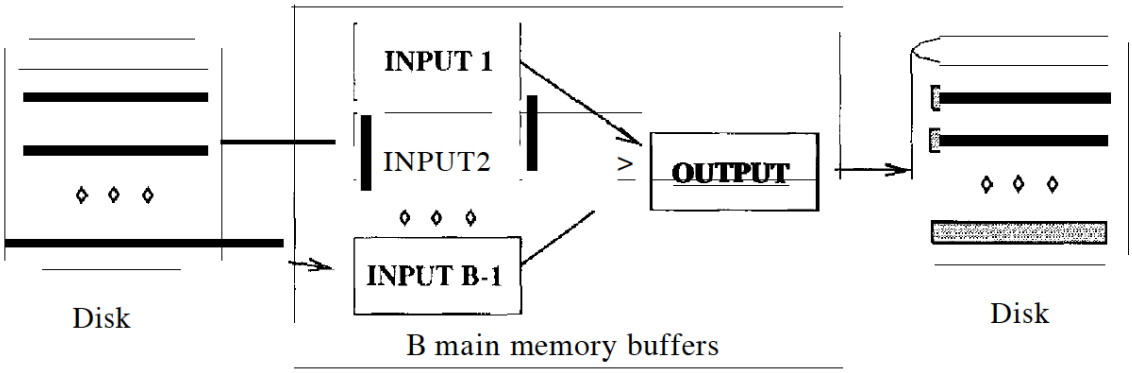
8.3 General external merge sort

Instead of using 3 buffer pages, use B buffer pages.

- pass 0 :
  first sort by groups of B pages. Each produced run is of size B, and they are $\lceil N/B \rceil$ different runs.



- pass $1, 2, \ldots$:
  merge the $B - 1$ runs



Cost/pass $= 2N$
Number of passes $= 1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil$

- **total cost** :

$$2N \cdot (1 + \lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

- *example* :
  $B = 5$, $N = 108$

pass	#sorted runs	pages/run (except last run)	pages/last_run
0	$108/5$ = 22	5	only 3 pages
1	$22/4$ = 6	$5 \cdot 4 = 20$	$22\%4 = 2$ runs left to merge (of size 5 and 3) $5 + 3 = 8$ pages
2	$6/4$ = 2	$5 \cdot 4^2 = 80$	$6\%4 = 2$ runs left to merge (of size 20 and 8) $20 + 8 = 28$ pages
3	sorted file of 108 pages		

Notation abuse : $x/y = \lceil x/y \rceil$