DBMS Internals
How does it all work?

Agenda

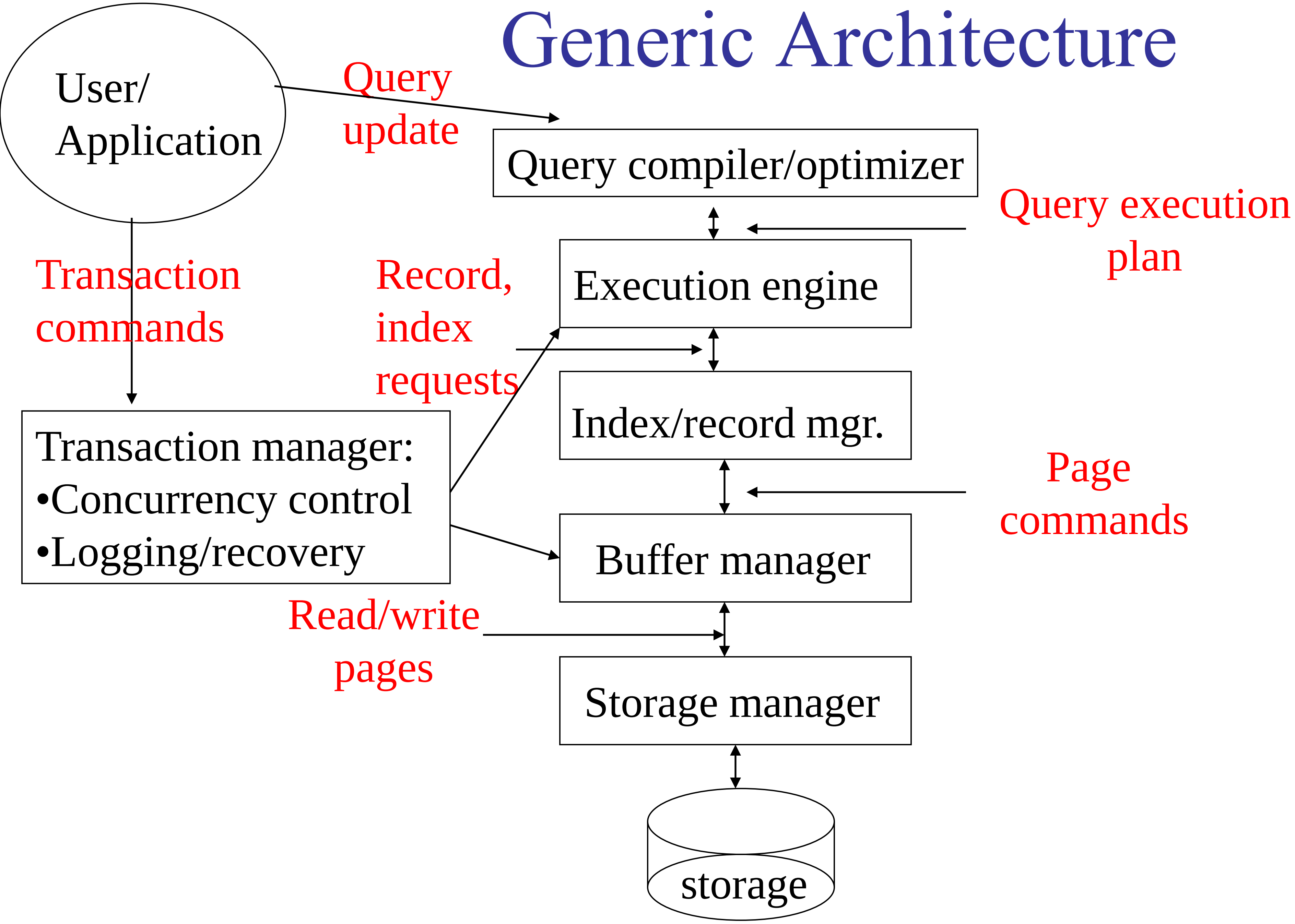
- Today: DBMS internals part 1 --
  - File organization
  - Indexing

What Should a DBMS Do?

- Store large amounts of data
- Process queries efficiently
- Allow multiple users to access the database concurrently and safely.
- Provide durability of the data.

- How will we do all this??

Generic Architecture



Main Points to Take Away

- I/O model of computation
  - We only count accesses to disk.
- Indexing:
  - Basic techniques: B+-tree, hash indexes
  - Secondary indexes.
- Efficient operator implementations: join
- Optimization: from *what* to *how*.

The Memory Hierarchy

Main Memory

- Volatile
- limited address spaces
- expensive
- **average access time: 10-100 nanoseconds**

Cache:

access time 10 nano's

Disk

- 5-10 MB/S transmission rates
- Gigs of storage
- **average time to access a block: 10-15 msecs.**
- Need to consider seek, rotation, transfer times.
- Keep records "close" to each other.

Tape

- 1.5 MB/S transfer rate
- 280 GB typical capacity
- **Only sequential access**
- Not for operational data

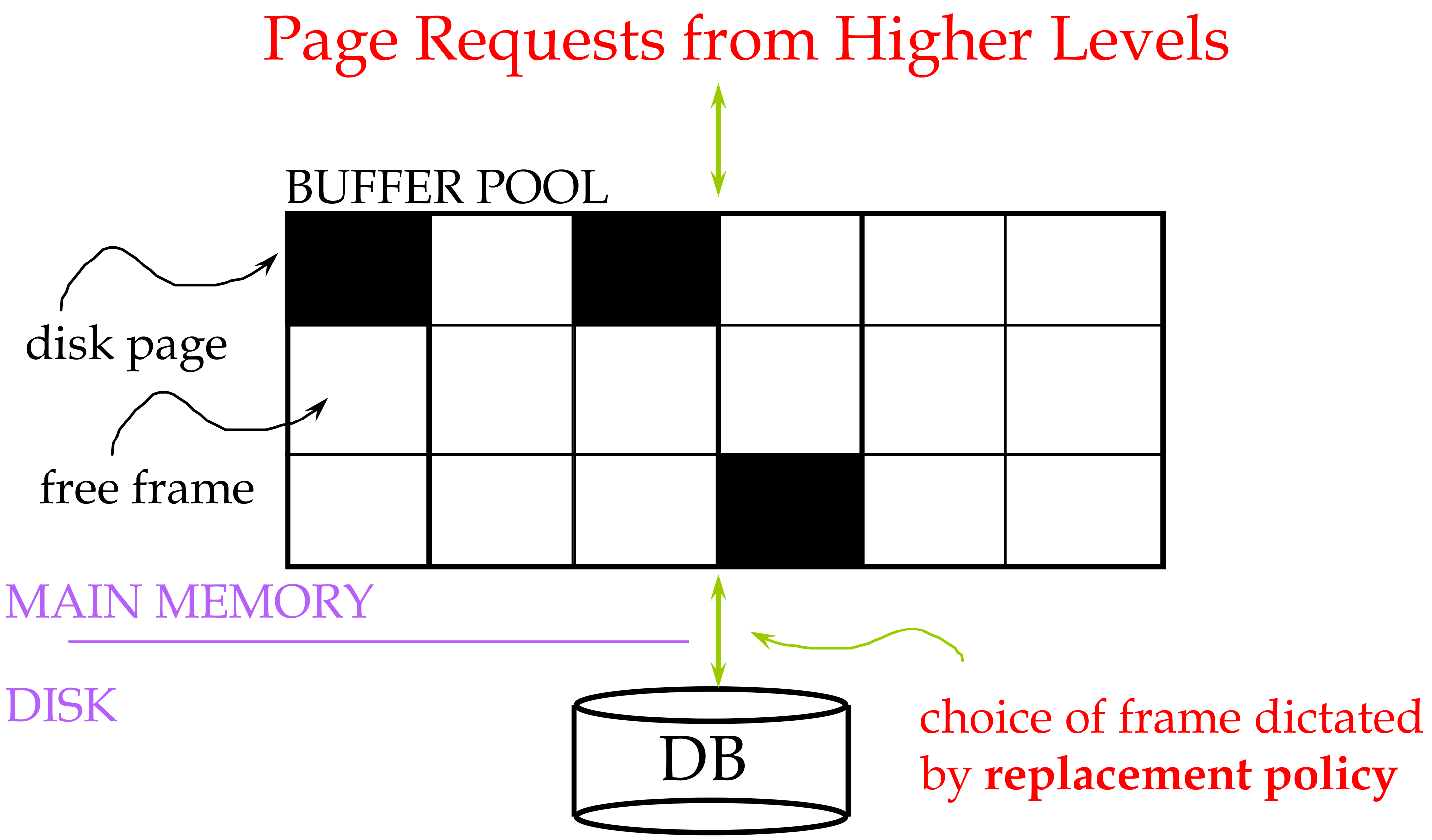
Main Memory

- Fastest, most expensive
- Today: 4GB-8GB are common on PCs
- Many databases could fit in memory
  - New industry trend: Main Memory Database
  - E.g TimesTen
- Main issue is volatility

Secondary Storage

- Disks
- Slower, cheaper than main memory
- Persistent !!!
- Used with a main memory buffer

Buffer Management in a DBMS


Page Requests from Higher Levels
BUFFER POOL
disk page
free frame
MAIN MEMORY
DISK
DB
choice of frame dictated by **replacement policy**

- *Data must be in RAM for DBMS to operate on it!*
- *Table of <frame#, pageid> pairs is maintained.*
- *LRU is not always good.*

Buffer Manager

Manages buffer pool: the pool provides space for a limited number of pages from disk.

Needs to decide on page replacement policy.

Enables the higher levels of the DBMS to assume that the needed data is in main memory.

Why not use the Operating System for the task??

- DBMS may be able to anticipate access patterns
- Hence, may also be able to perform prefetching
- DBMS needs the ability to force pages to disk.

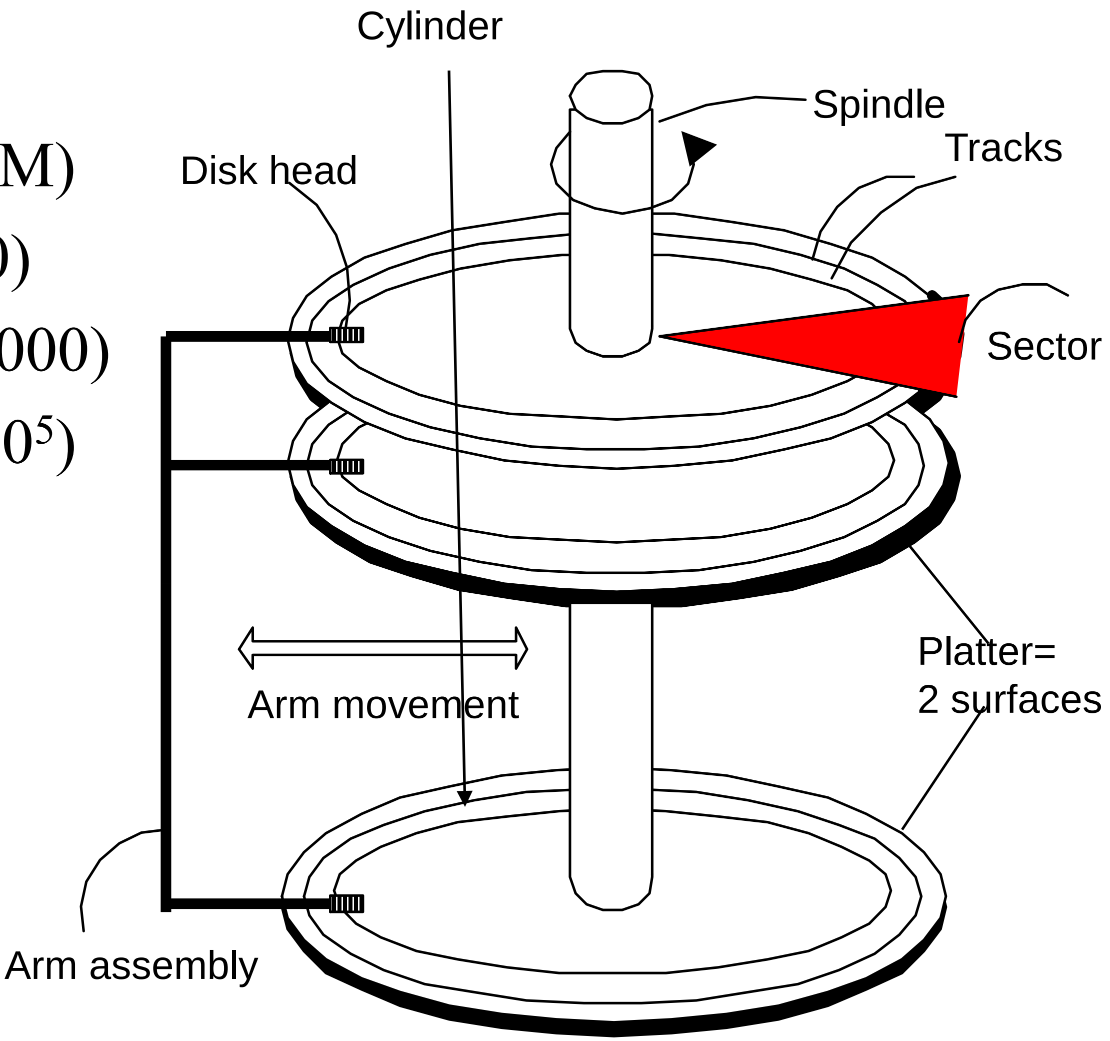
Tertiary Storage

- Tapes or optical disks
- Extremely slow: used for long term archiving only

The Mechanics of Disk

Mechanical characteristics:

- Rotation speed (5400RPM)
- Number of platters (1-30)
- Number of tracks (<=10000)
- Number of bytes/track(10^5)



Disk Access Characteristics

- Disk latency = time between when command is issued and when data is in memory
- Is not following Moore’s Law!

摩尔定律是由英特尔（Intel）创始人之一戈登·摩尔（Gordon Moore）提出来的。其内容为：当价格不变时，集成电路上可容纳的元器件的数目，约每隔18-24个月便会增加一倍，性能也将提升一倍。换言之，每一美元所能买到的电脑性能，将每隔18-24个月翻一倍以上。这一定律揭示了信息技术进步的速度。

Disk Access Characteristics

Disk latency = seek time
+ rotational latency
+ transfer time

- Seek time = time for the head to reach cylinder containing the track on which the block is located
  - 10ms – 40ms
- Rotational latency = time for the first sector of the block moves under the head
  - Rotation time = 10ms
  - Average latency = 10ms/2

- Transfer time = all the sectors and gaps between them pass under the head, while the disk controller reads or writes data in these sectors, typically 10MB/s

- Disks read/write one block at a time (typically 4kB)

The I/O Model of Computation

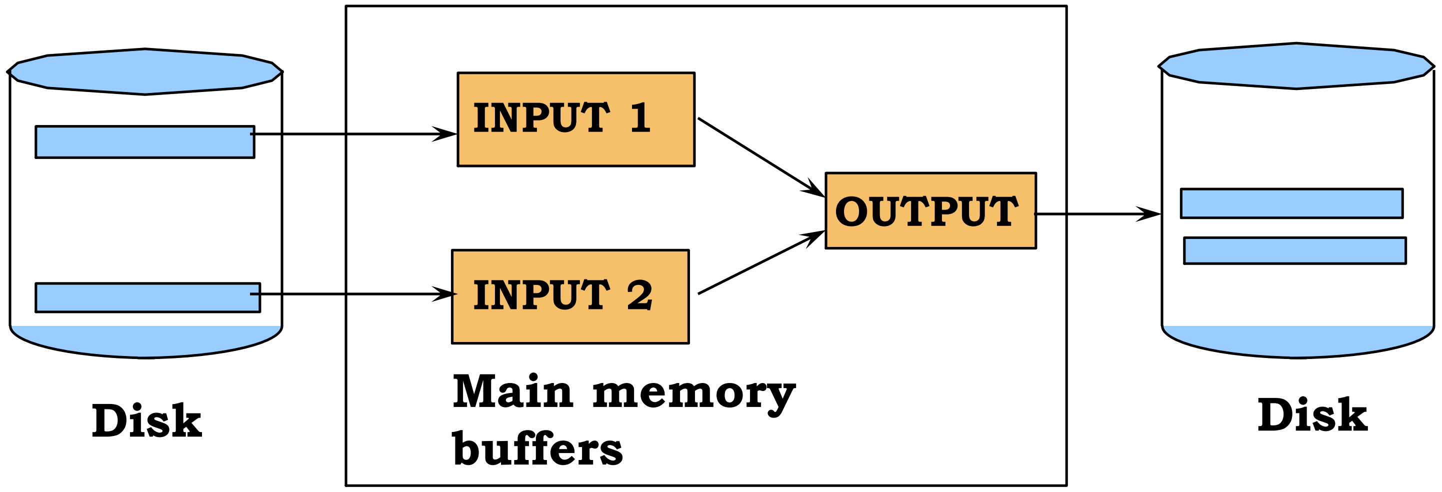
- In main memory algorithms we care about CPU time
- In databases time is dominated by I/O cost
- Assumption: cost is given only by I/O
- Consequence: need to redesign certain algorithms
- Will illustrate here with sorting

Sorting

- Illustrates the difference in algorithm design when your data is not in main memory:
  - Problem: sort 1Gb of data with 1Mb of RAM.
- Arises in many places in database systems:
  - Data requested in sorted order (ORDER BY)
  - Needed for grouping operations
  - First step in sort-merge join algorithm
  - Duplicate removal
  - Bulk loading of B+-tree indexes.

2-Way Merge-sort: Requires 3 Buffers

- Pass 1: Read a page, sort it, write it.
  - only one buffer page is used
- Pass 2, 3, …, etc.:
  - three buffer pages used.



Two-Way External Merge Sort

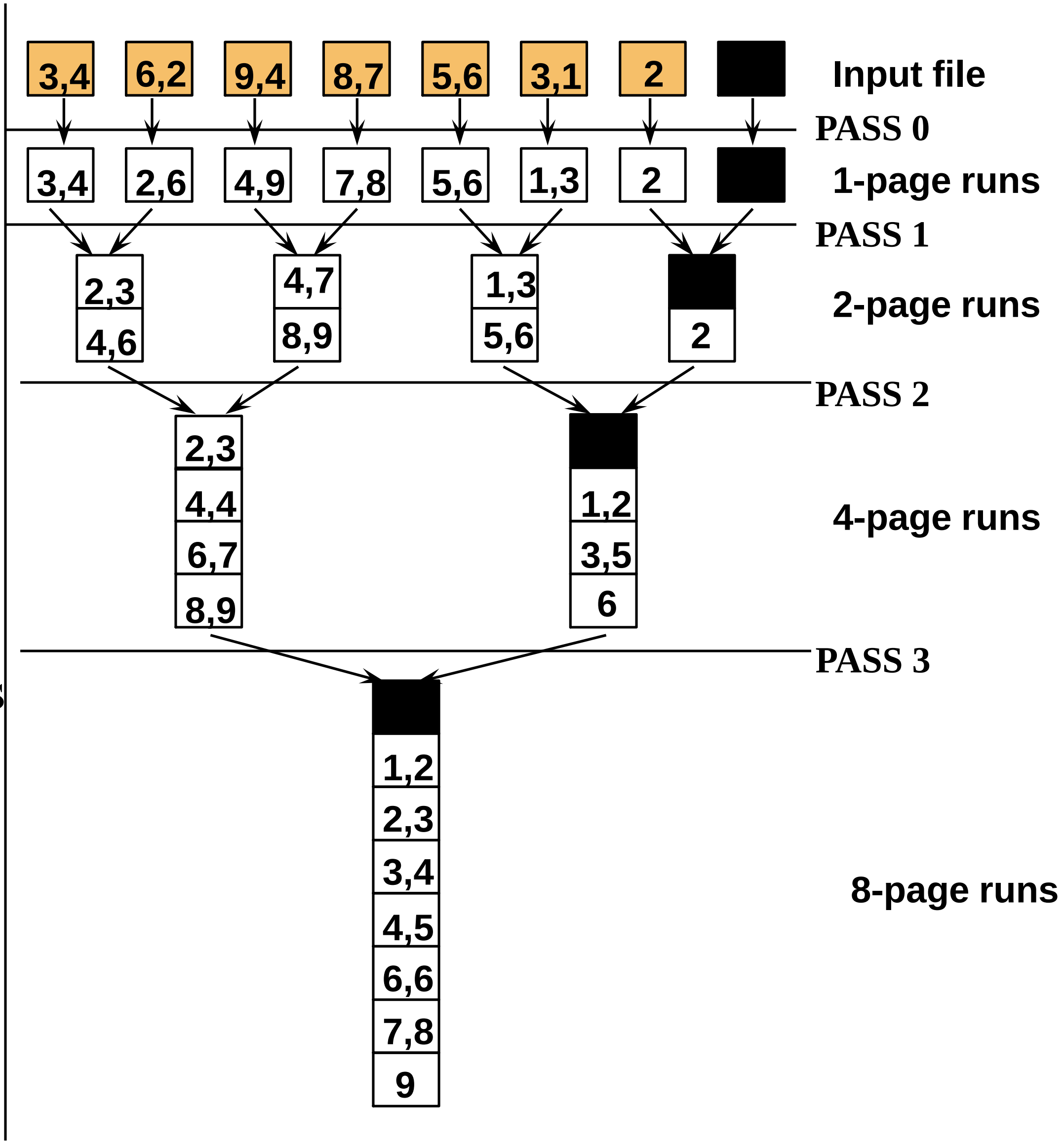
- Each pass we read + write each page in file.
- N pages in the file => the number of passes

$$=\lceil \log_2 N \rceil + 1$$

- So total cost is:

$$2N\left(\lceil \log_2 N \rceil + 1\right)$$

- Improvement: start with larger runs
- Sort 1GB with 1MB memory in 10 passes



Can We Do Better ?

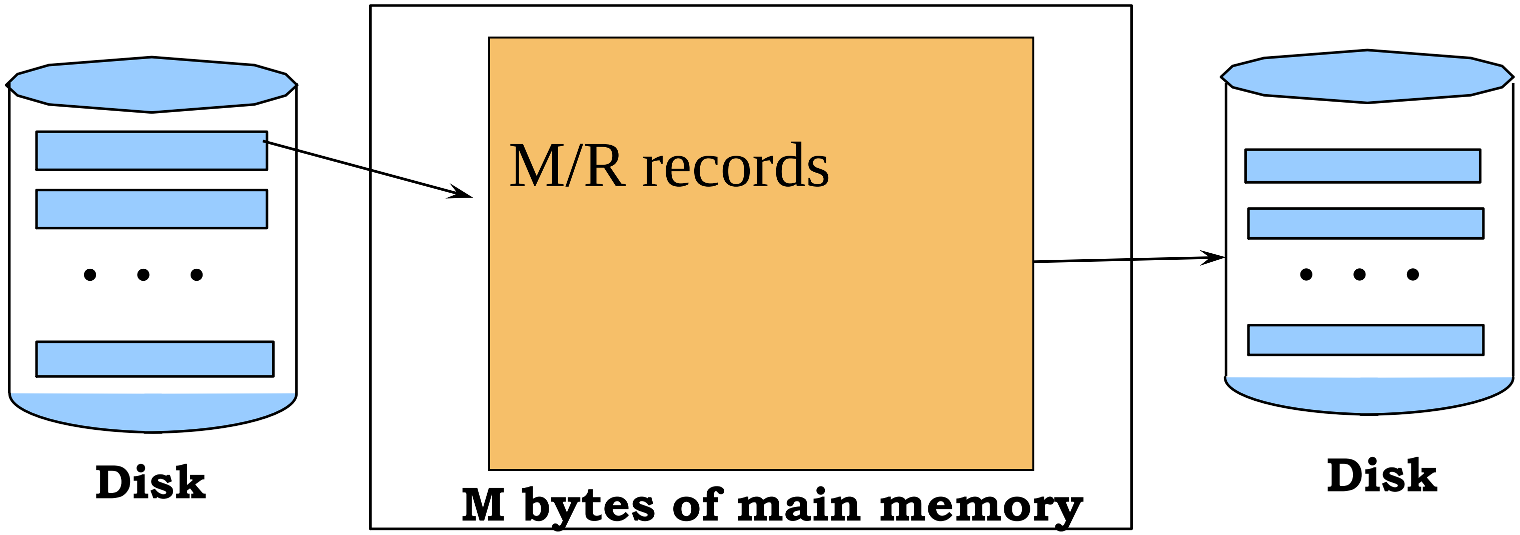
- We have more main memory
- Should use it to improve performance

Cost Model for Our Analysis

- **B:** Block size
- **M:** Size of main memory
- **N:** Number of records in the file
- **R:** Size of one record

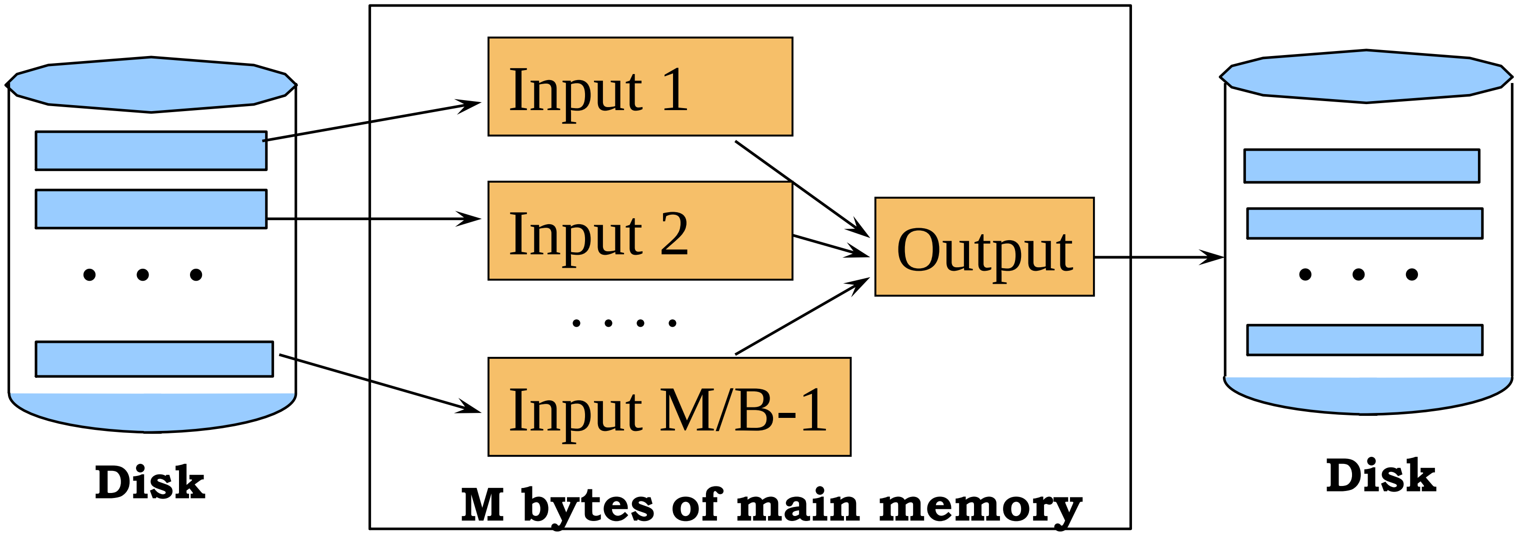
External Merge-Sort

- Phase one: load M bytes in memory, sort
  - Result: runs of length M/R records (only by one pass)



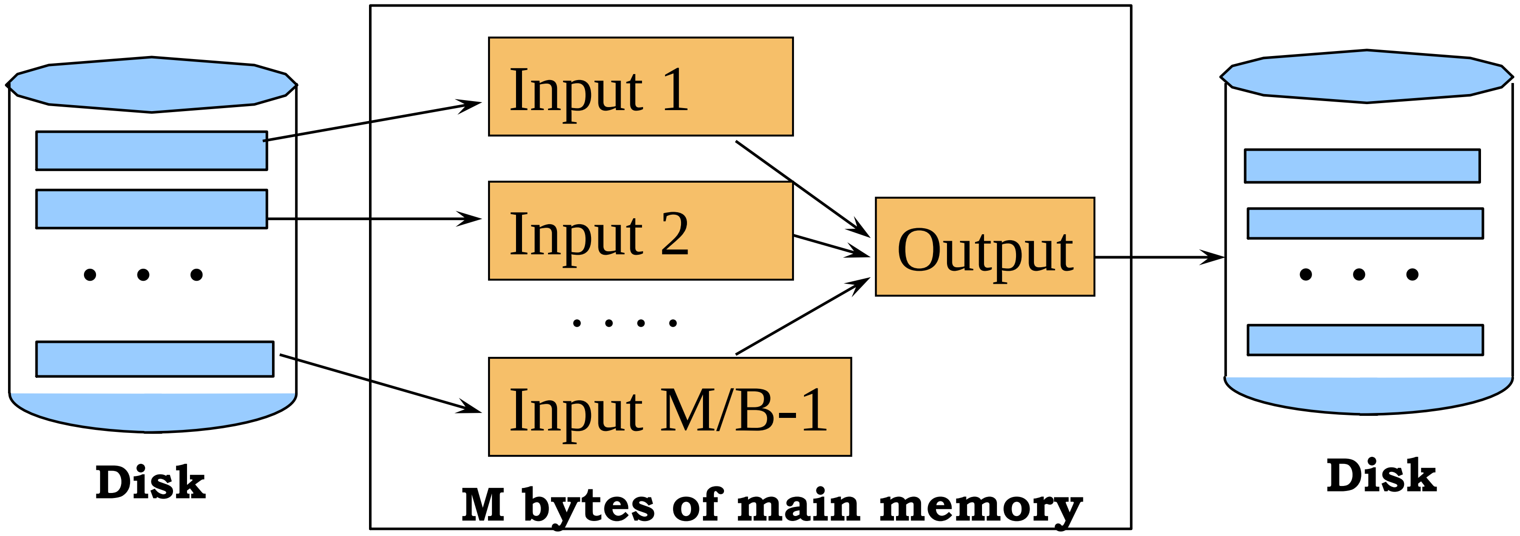
Phase Two

- Merge M/B – 1 runs into a new run
- Result: runs have now M/R (M/B – 1) records (by two passes)



Phase Three

- Merge M/B – 1 runs into a new run
- Result: runs have now M/R $(M/B – 1)^2$ records



Cost of External Merge Sort

- Number of passes: $1+\lceil \log_{M/B-1} \lceil NR/M \rceil \rceil$
- Think differently
  - Given B = 4KB, M = 64MB, R = 0.1KB
  - Pass 1: runs of length M/R = 640000
    - Have now sorted runs of 640000 records
  - Pass 2: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10,240,000,000 = 10^{10} records
  - Pass 3: runs increase by a factor of M/B – 1 = 16000
    - Have now sorted runs of 10^{14} records
    - Nobody has so much data !
- Can sort everything in 2 or 3 passes !

Number of Passes of External Sort

P	F=3	F=5	F=9	F=17	F=129	F=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

F=M/B: number of frames in the buffer pool;
P=NR/B : number of pages in relation.

Data Storage and Indexing

Representing Data Elements

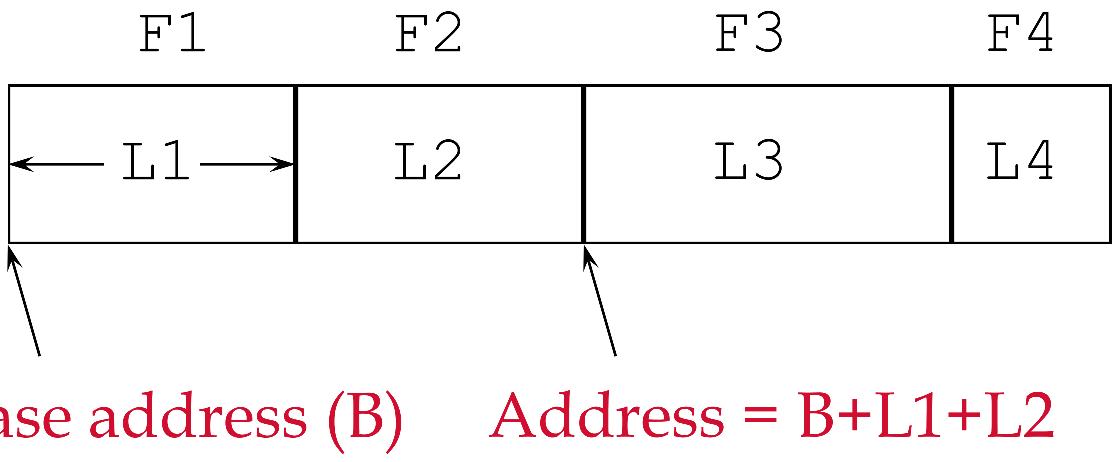
- Relational database elements:

```
CREATE TABLE Product (

        pid INT PRIMARY KEY,
        name CHAR(20),
        description VARCHAR(200),
        maker CHAR(10) REFERENCES Company(name)
)
```

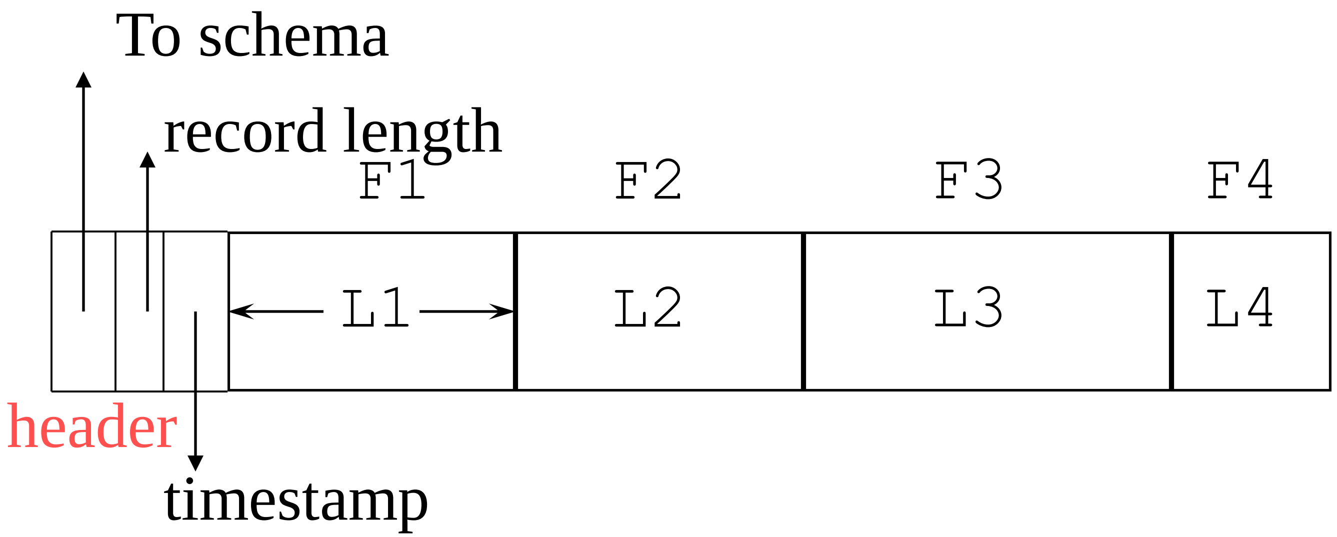
- A tuple is represented as a record

Record Formats: Fixed Length



- Information about field types same for all records in a file; stored in *system catalogs.*
- Finding *i'th* field requires scan of record.
- **Note the importance of schema information!**

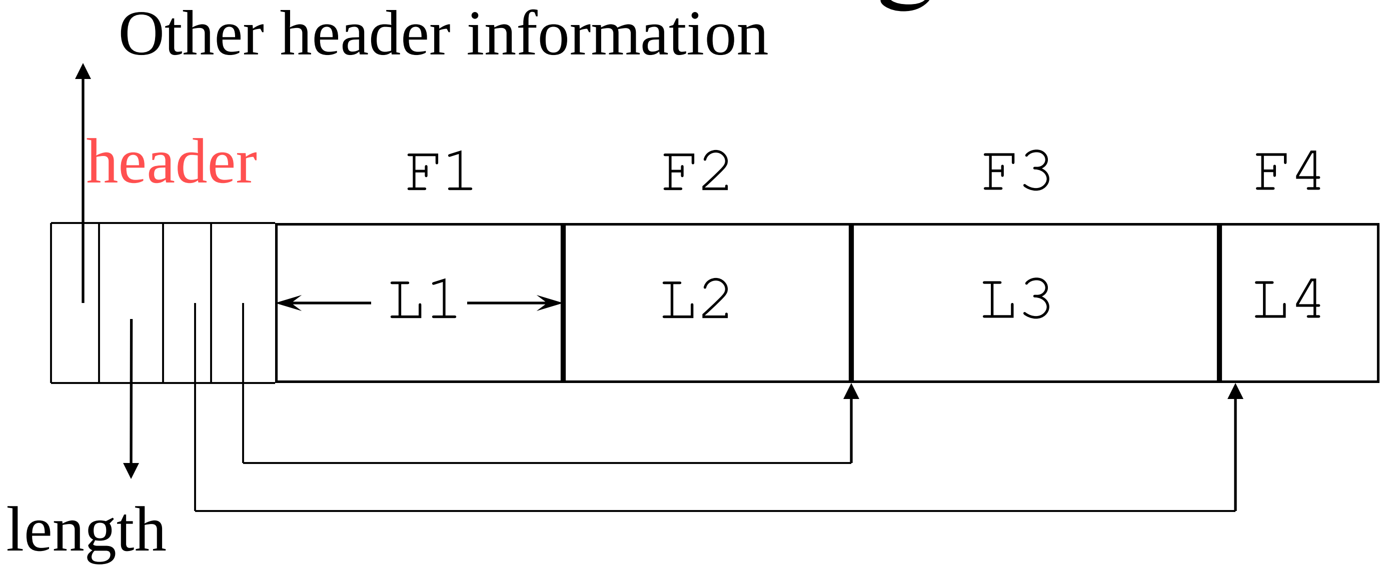
Record Header



Need the header because:

- The schema may change
    for a while new+old may coexist
- Records from different relations may coexist

Variable Length Records



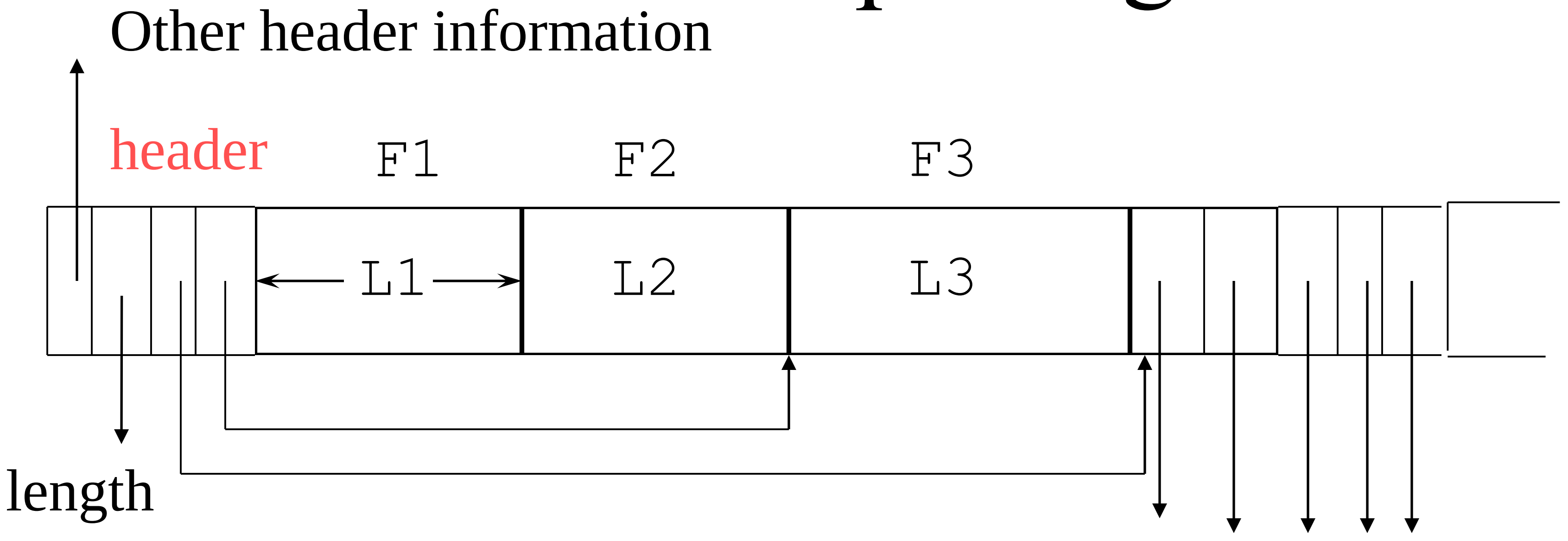
Place the fixed fields first: F1, F2

Then the variable length fields: F3, F4

Null values take 2 bytes only

Sometimes they take 0 bytes (when at the end)

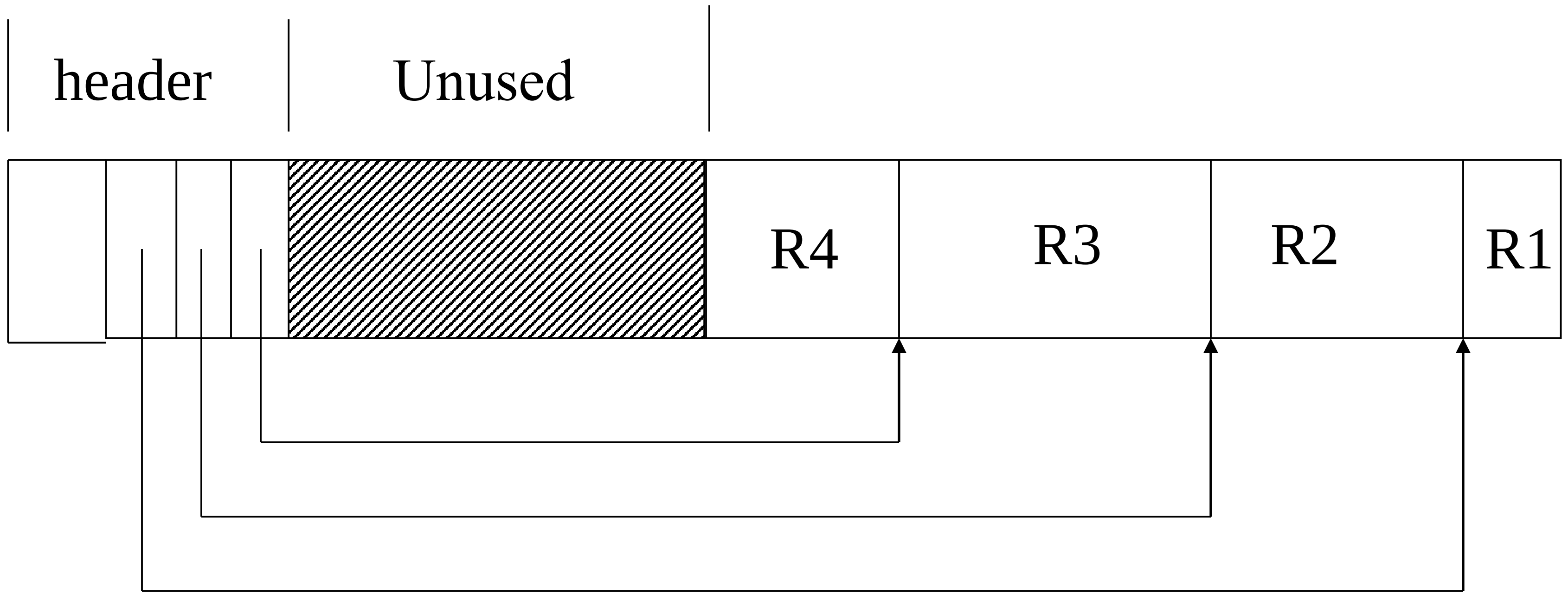
Records With Repeating Fields



Needed e.g. in Object Relational systems,
or fancy representations of many-many relationships

Storing Records in Blocks

- Blocks have fixed size (typically 4k)



Storage and Indexing

- How do we store efficiently large amounts of data?
- The appropriate storage depends on what kind of accesses we expect to have to the data.
- We consider:
  - primary storage of the data
  - additional indexes (very very important).

Cost Model for Our Analysis

- As a good approximation, we ignore CPU costs:
  - **B:** The number of data pages
  - **R:** Number of records per page
  - **D:** (Average) time to read or write disk page
  - Measuring number of page I/O's ignores gains of pre-fetching blocks of pages; thus, even I/O cost is only approximated.
  - Average-case analysis; based on several simplistic assumptions.

File Organizations and Assumptions

- Heap Files:
  - Equality selection on key; exactly one match.
  - Insert always at end of file.
- Sorted Files:
  - Files compacted after deletions.
  - Selections on sort field(s).
- Hashed Files:
  - No overflow buckets, 80% page occupancy.
- Single record insert and delete.

Indexes

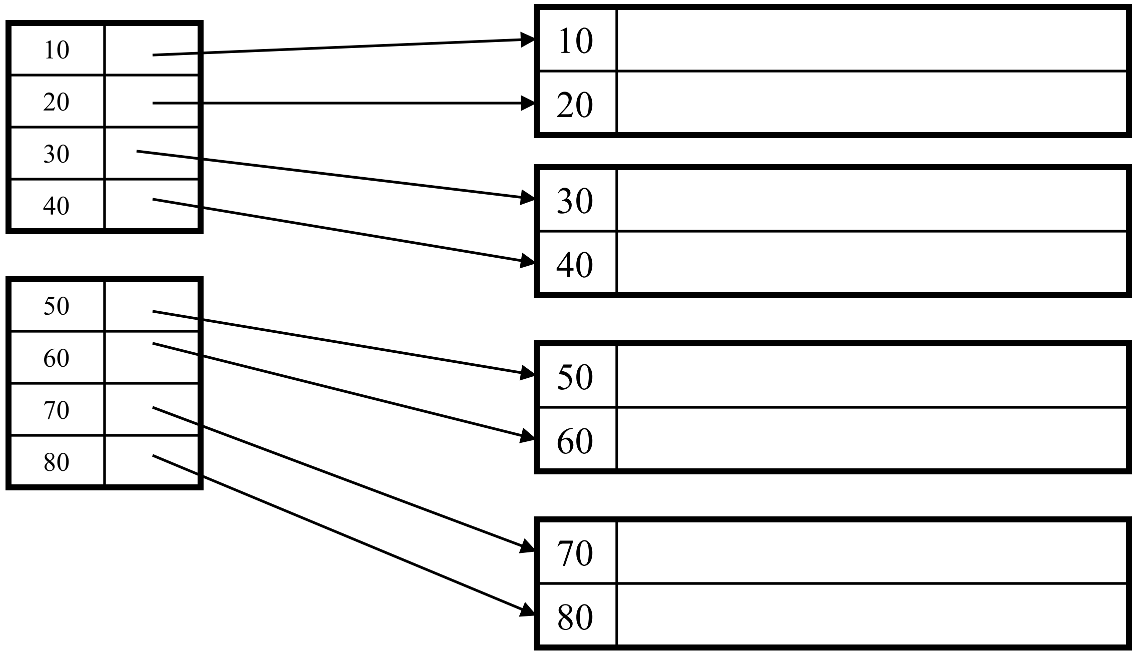
- An *index* on a file speeds up selections on the *search key fields* for the index.
  - Any subset of the fields of a relation can be the search key for an index on the relation.
  - *Search key* is not the same as *key* (minimal set of fields that uniquely identify a record in a relation).
- An index contains a collection of *data entries*, and supports efficient retrieval of all data entries with a given key value **k**.

Index Classification

- Primary/secondary
- Clustered/unclustered
- Dense/sparse
- B+ tree / Hash table / …

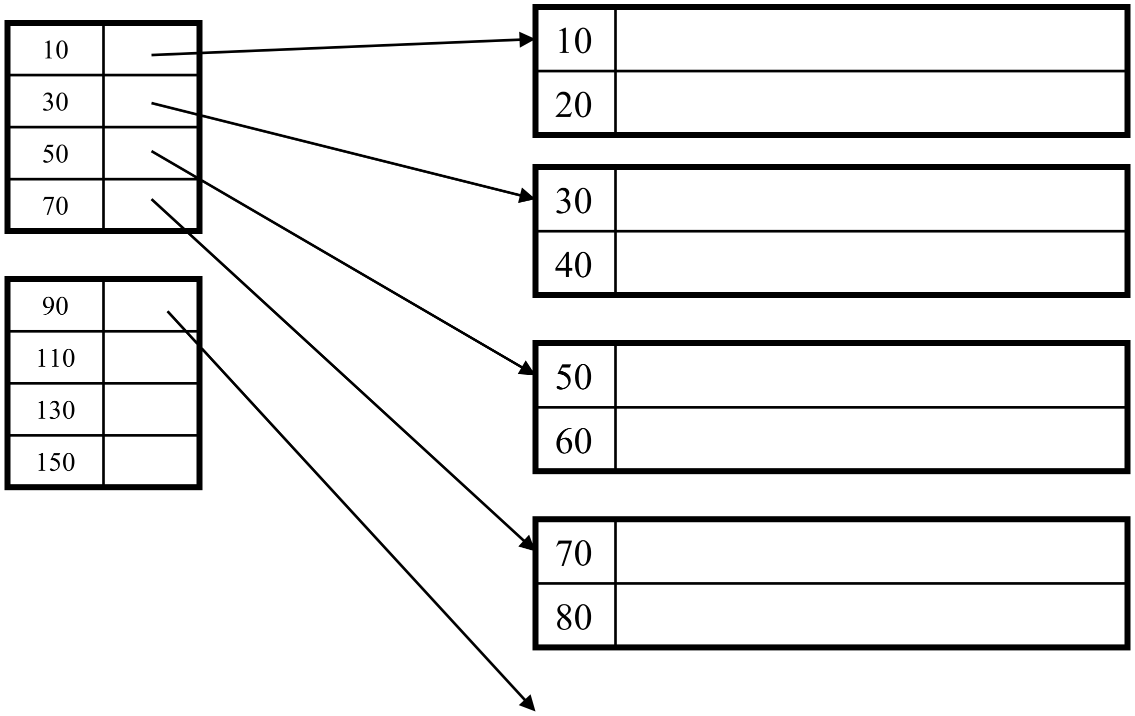
Primary Index

- File is sorted on the index attribute
- *Dense* index: sequence of (key,pointer) pairs



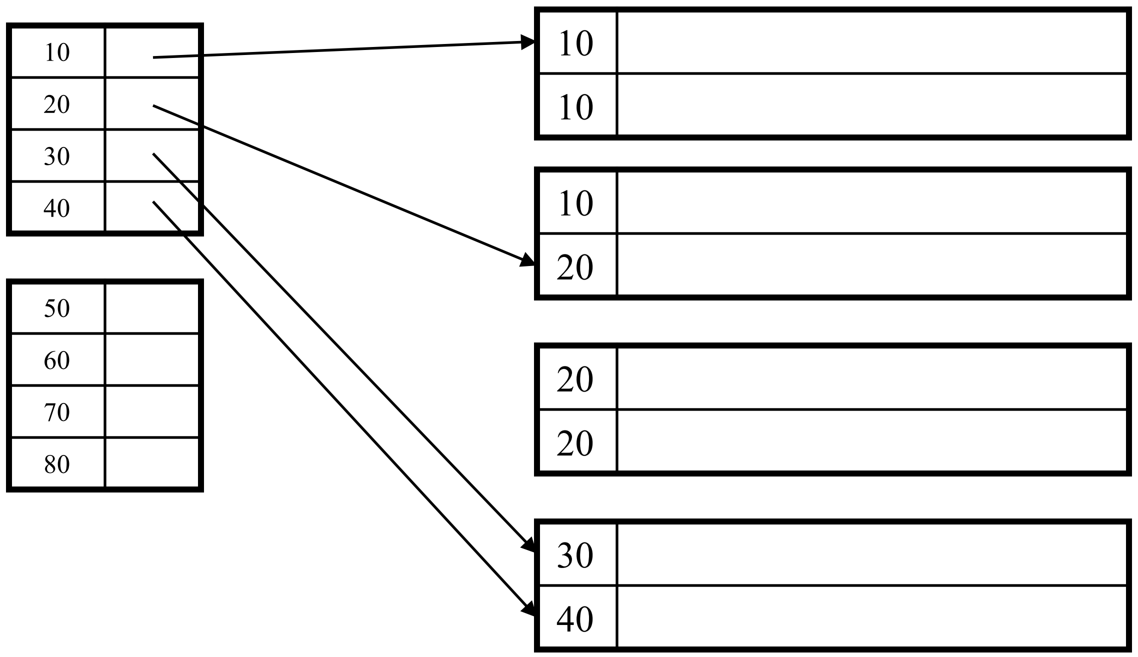
Primary Index

- ***Sparse*** index



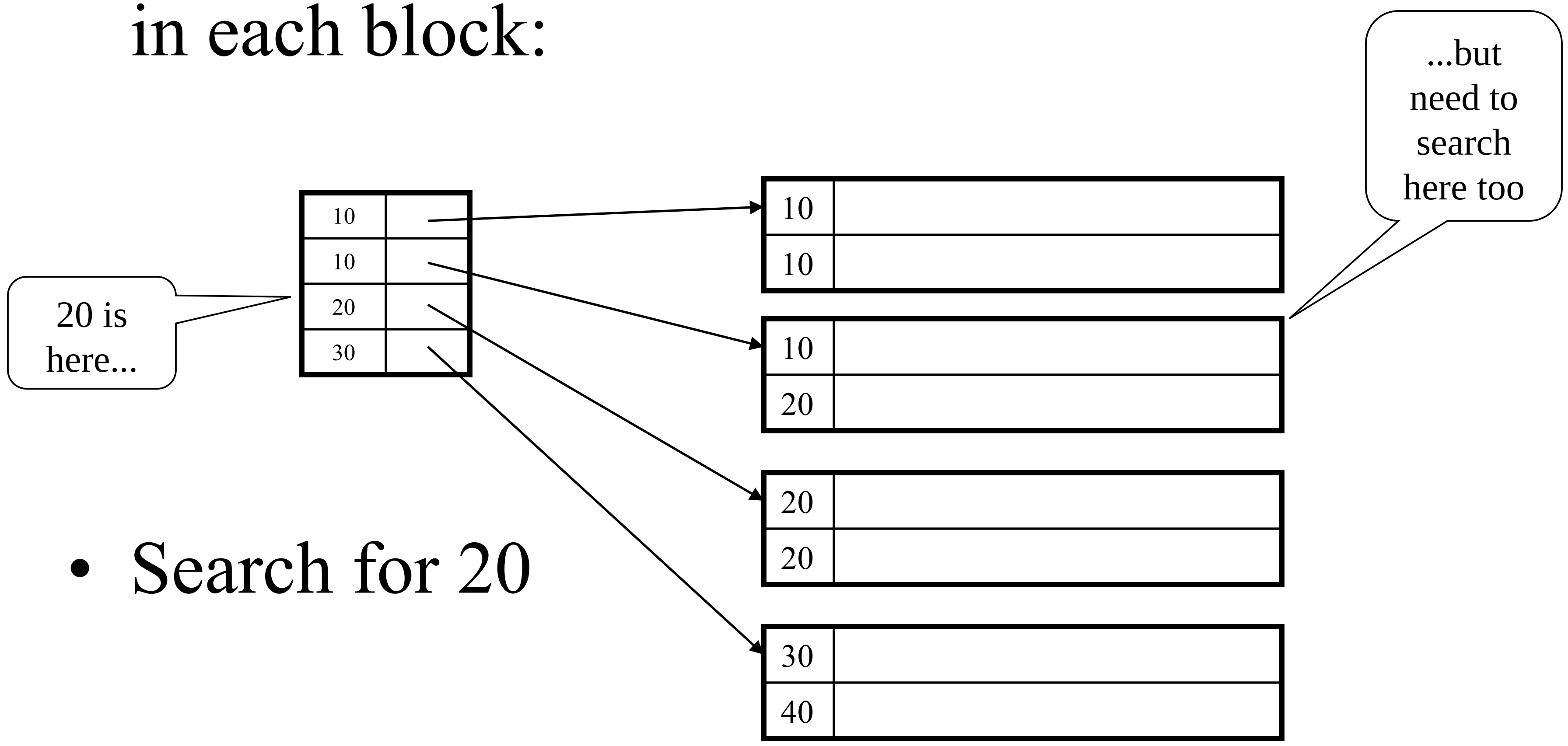
Primary Index with Duplicate Keys

- Dense index:



Primary Index with Duplicate Keys

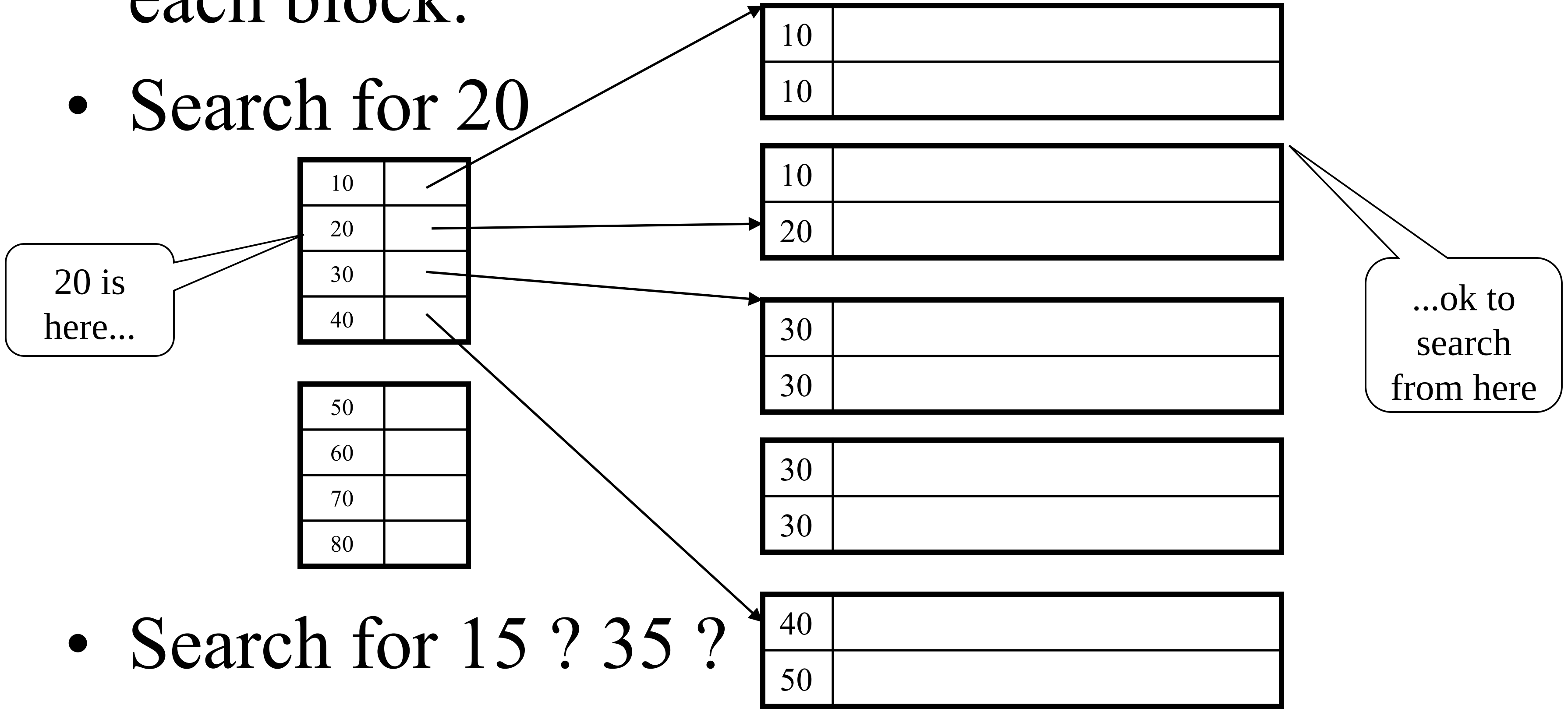
- Sparse index: pointer to lowest search key in each block:



- Search for 20

Primary Index with Duplicate Keys

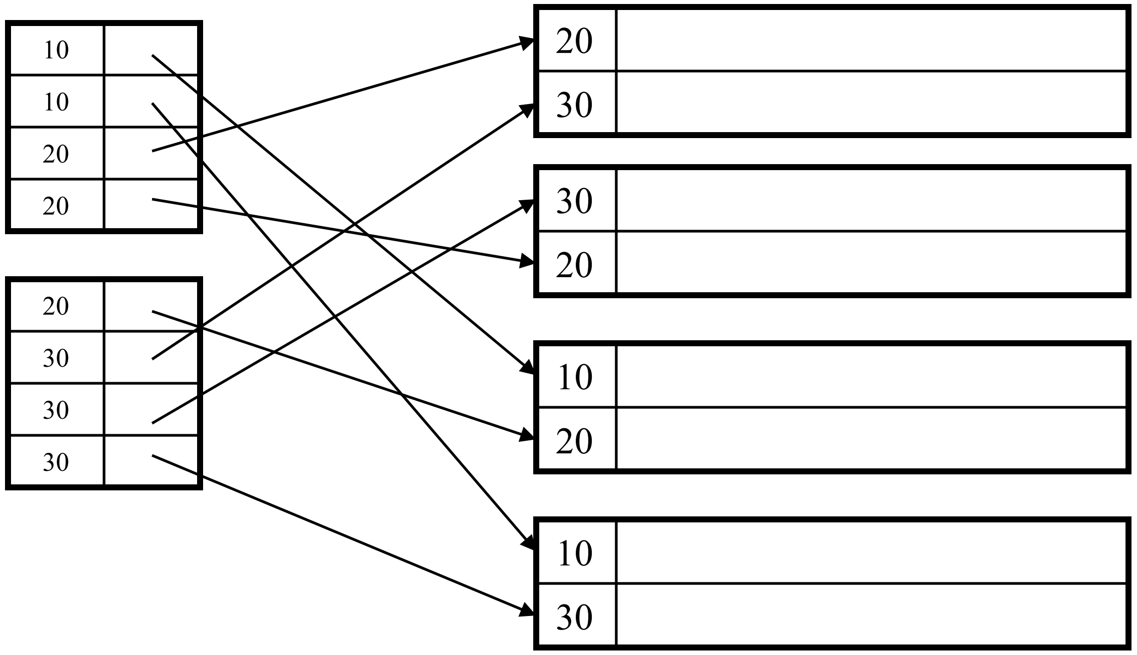
- Better: pointer to <u>***lowest new search key***</u> in each block:
- Search for 20



- Search for 15 ? 35 ?

Secondary Indexes

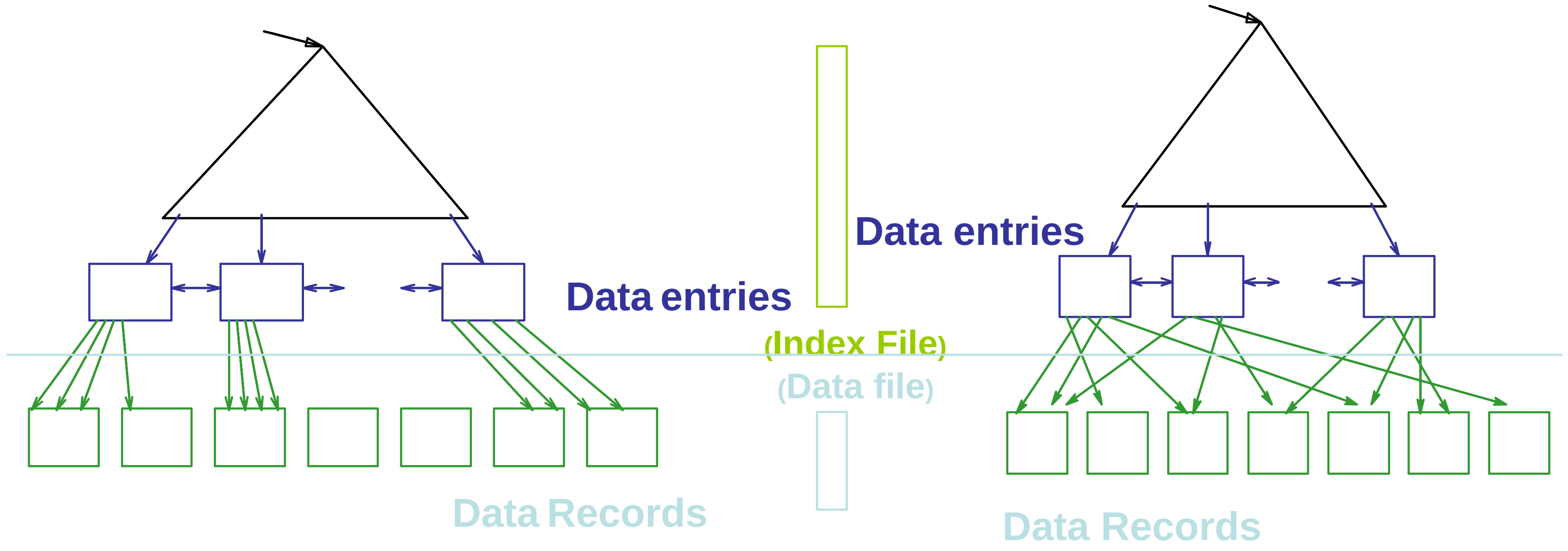
- To index other attributes than primary key
- Always dense (why ?)



Clustered/Unclustered

- Primary indexes = usually clustered
- Secondary indexes = usually unclustered

Clustered vs. Unclustered Index



CLUSTERED

UNCLUSTERED

Secondary Indexes

- Applications:
  - index other attributes than primary key
  - index unsorted files (heap files)
  - index clustered data

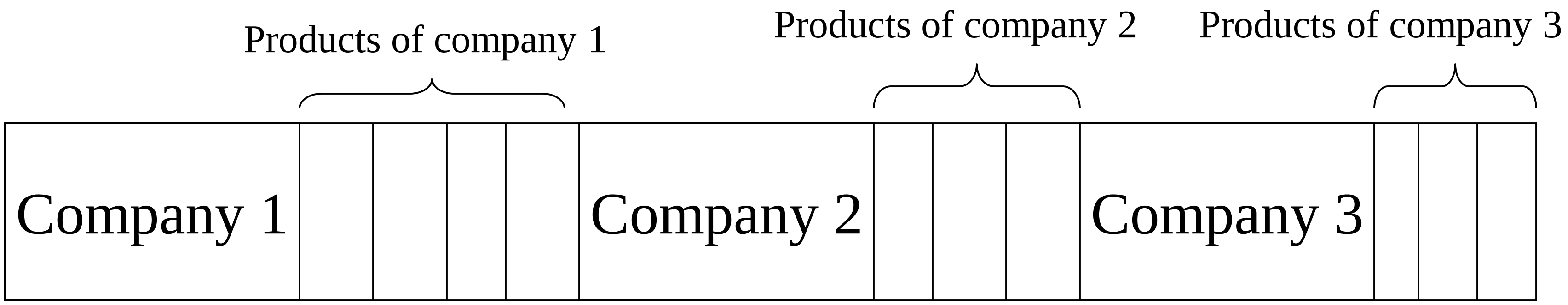
Applications of Secondary Indexes

- ***Clustered data***

Company(name, city), Product(pid, maker)

```
Select city
From Company, Product
Where name=maker
    and pid="p045"
```

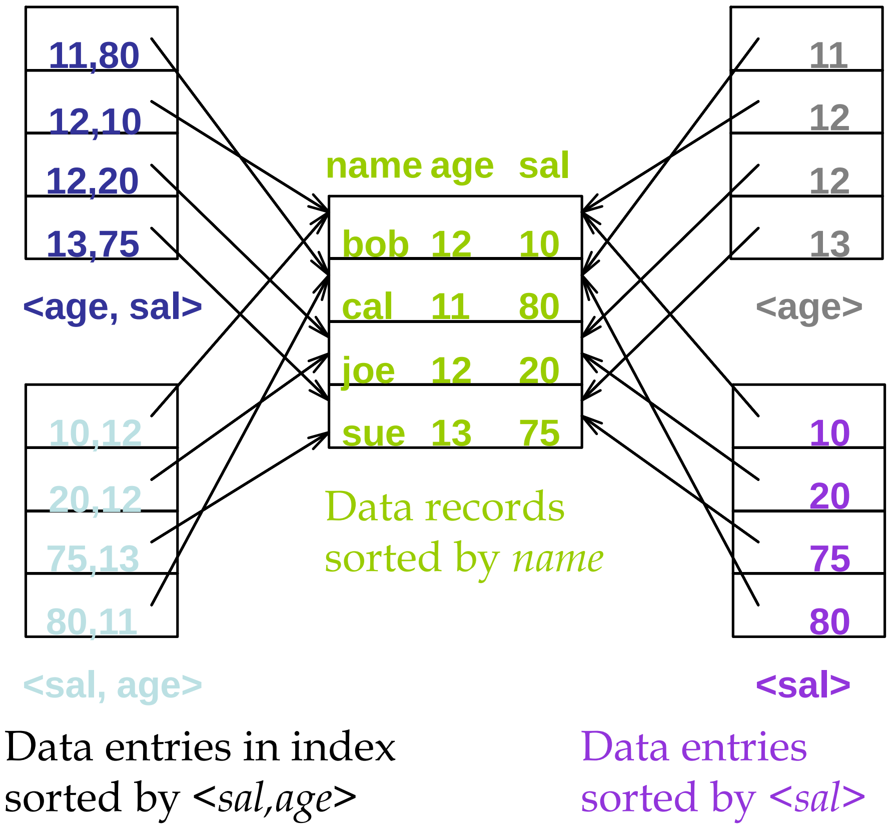
```
Select pid
From Company, Product
Where name=maker
    and city="Seattle"
```



Composite Search Keys

- *Composite Search Keys*: Search on a combination of fields.
  - Equality query: Every field value is equal to a constant value. E.g. wrt <sal,age> index:
    - age=13 and sal =75
  - Range query: Some field value is not a constant. E.g.:
    - age =13; or age=13 and sal > 10

Examples of composite key indexes using lexicographic order.

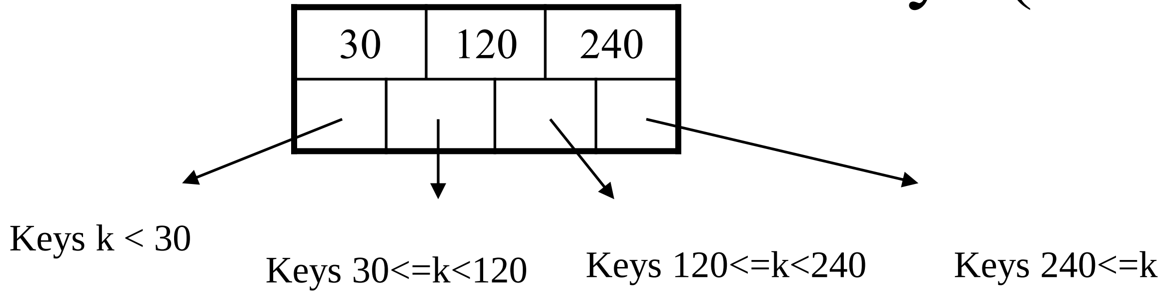


B+ Trees

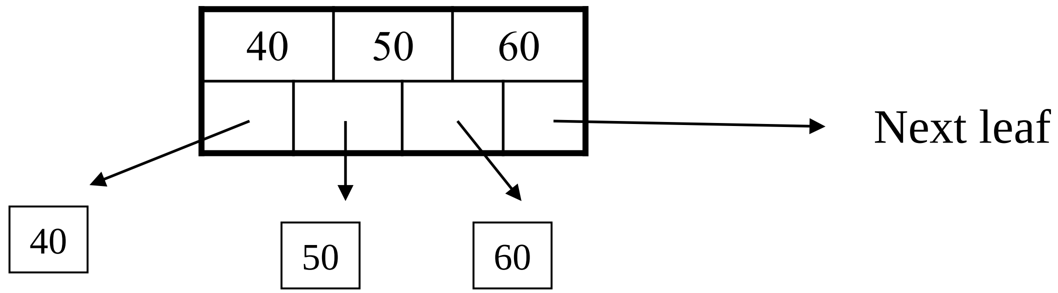
- Search trees
- Idea in B   Trees:
  - make 1 node = 1 block
- Idea in B+ Trees:
  - Make leaves into a linked list (range queries are easier)

B+ Trees Basics

- Parameter d = the *degree*
- Each node has >= d and <= 2d keys (except root)

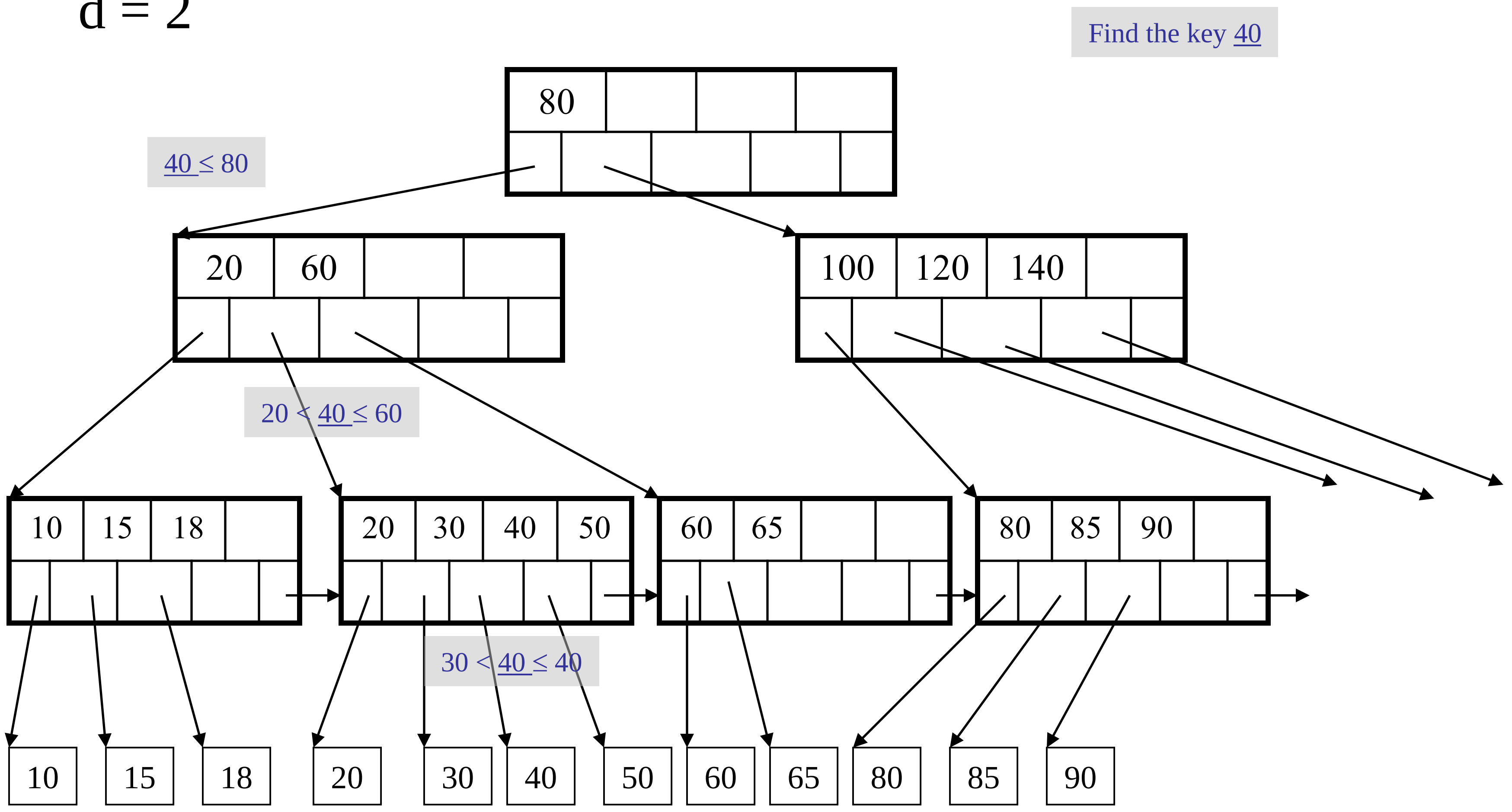


- Each leaf has >=d and <= 2d keys:



B+ Tree Example

d = 2



B+ Tree Design

- How large d ?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- $2d \times 4 + (2d+1) \times 8 <= 4096$
- $d = 170$

Searching a B+ Tree

- Exact key values:
  - Start at the root
  - Proceed down, to the leaf

```
Select name
From people
Where age = 25
```

- Range queries:
  - As above
  - Then sequential traversal

```
Select name
From people
Where 20 <= age
  and  age <= 30
```

B+ Trees in Practice

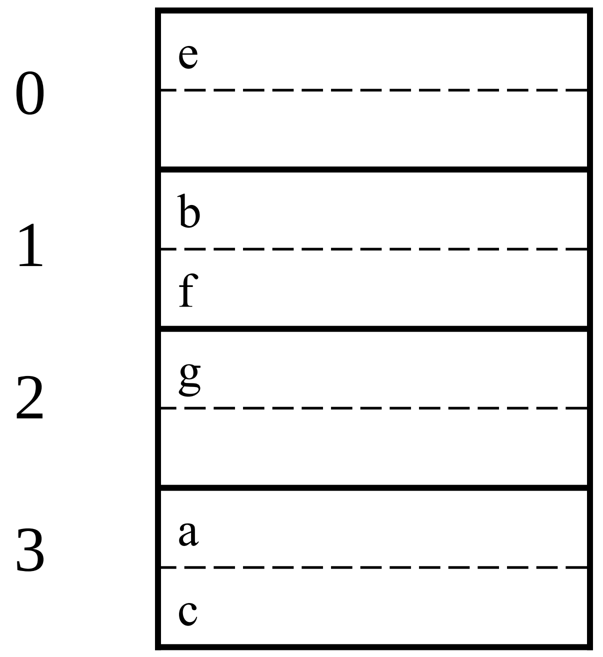
- Typical order (degree) : 100 . Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records（contained by B+ tree with 4 level)
  - Height 3: 133^3 =   2,352,637 records （contained by B+ tree with 3 level)
- Can often hold top levels in buffer pool:
  - Level 1 =   1 page =   8 Kbytes
  - Level 2 =   133 pages =   1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

Hash Tables

- Secondary storage hash tables are much like main memory ones
- Recall basics:
  - There are n ***buckets***
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

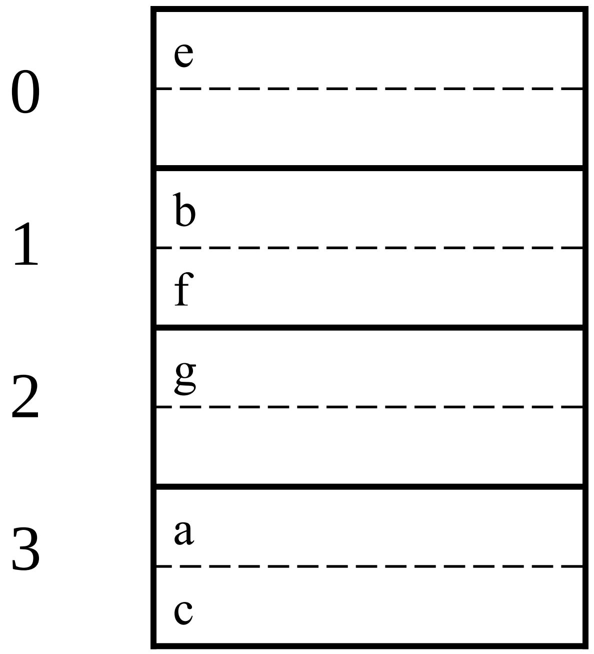
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3



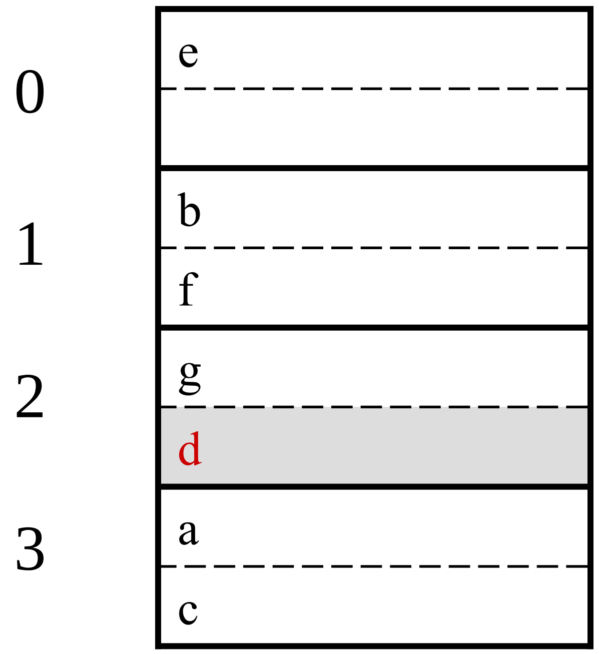
Searching in a Hash Table

- Search for a:
- Compute h(a)=3
- Read bucket 3
- 1 disk access



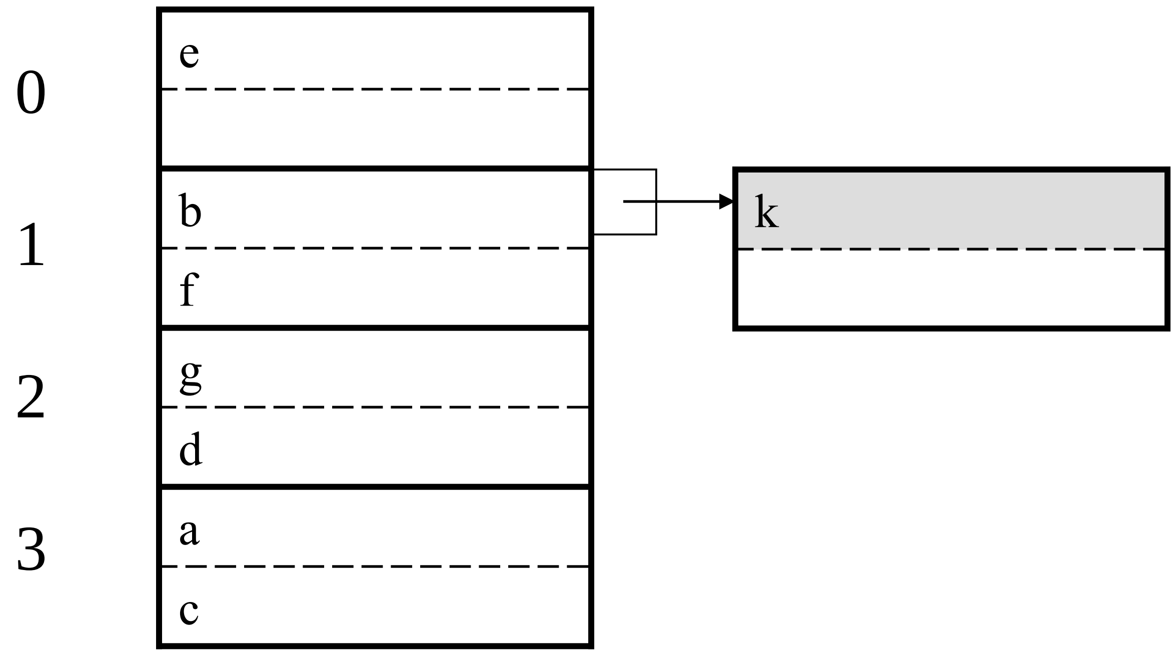
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More over-flow blocks may be needed

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).
- Typically, we assume that a hash-lookup takes 1.2 I/Os.

Where are we?

- File organizations: sorted, hashed, heaps.
- Indexes: hash index, B+-tree
- Indexes can be clustered or not.
- Data can be stored *in* the index or not.

- Hence, when we access a relation, we can either scan or go through an index:
  - Called an *access path.*

Next:

- Indexing vs Hashing
- Index definition in SQL
- Multiple key access

Indexing vs Hashing

- Hashing good for probes given key

  e.g.,
```
SELECT …
FROM R
WHERE R.A = 5
```

Indexing vs Hashing

- INDEXING (Including B Trees) good for

  Range Searches:

  e.g.,

```
SELECT
    FROM R
    WHERE R.A > 5
```

Index definition in SQL

- Create index name on rel (attr)
- Create unique index name on rel (attr)

$\hookrightarrow$ defines candidate key

- Drop INDEX name

$\boxed{\text{Note}}$ CANNOT SPECIFY TYPE OF INDEX

(e.g. B-tree, Hashing, …)

OR PARAMETERS

(e.g. Load Factor, Size of Hash,...)

... at least in SQL...

Note ATTRIBUTE <u>LIST</u> $\Rightarrow$ MULTIKEY INDEX

(next)

e.g., <u>CREATE</u> <u>INDEX</u> foo <u>ON</u> R(A,B,C)

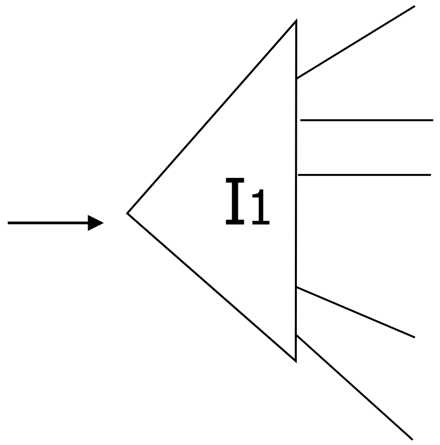
Multi-key Index

Motivation: Find records where

DEPT = "Toy" AND SAL > 50k

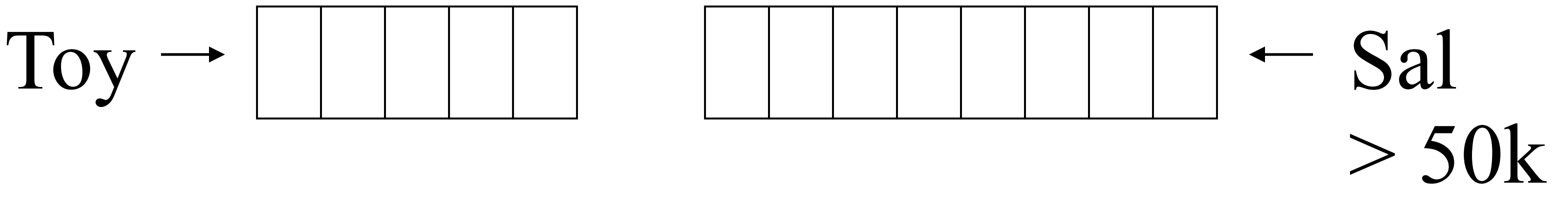
Strategy I:

- Use one index, say Dept.
- Get all Dept = “Toy” records
  and check their salary

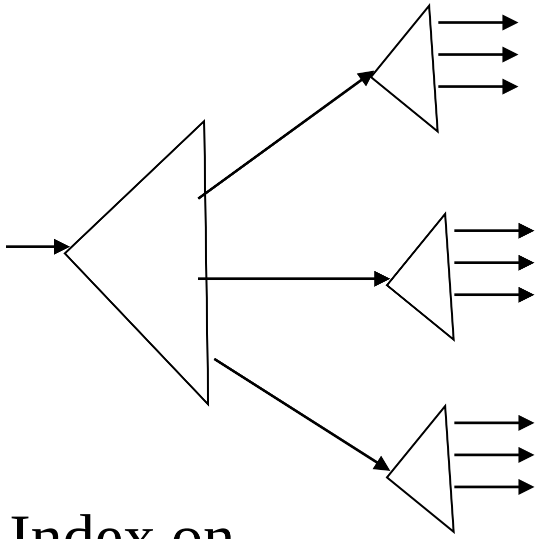


Strategy II:

- Use 2 Indexes; Manipulate Pointers

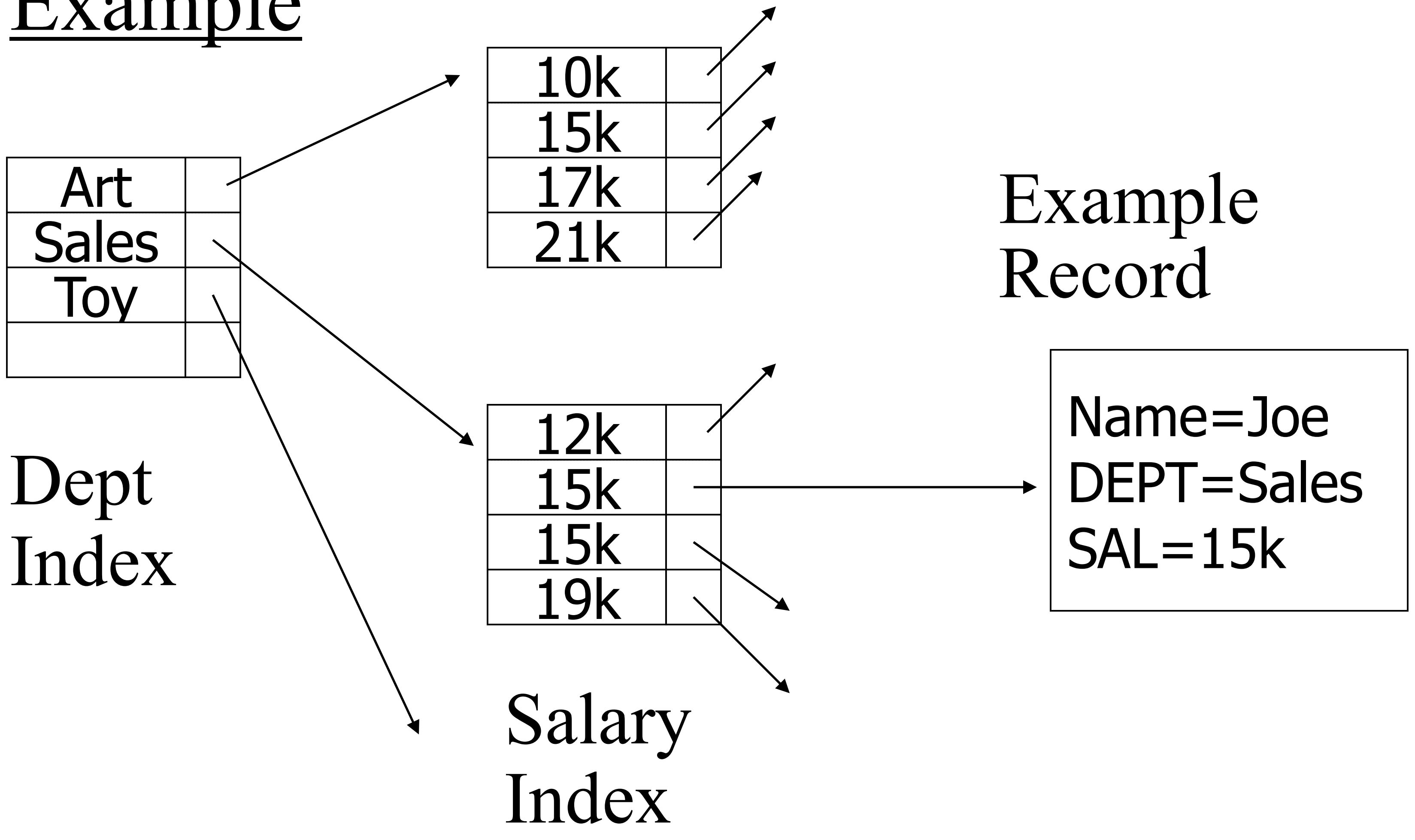


Strategy III: Multiple Key Indexes



- Each level as an index for one of the attributes.
- Works well for partial matches if the match includes the first attributes.

Example


Art
Sales
Toy
Dept Index
10k
15k
17k
21k
12k
15k
15k
19k
Salary Index
Example Record
Name=Joe
DEPT=Sales
SAL=15k

For which queries is this index good?

- ☐ Find RECs Dept = "Sales" $\wedge$SAL=20k
- ☐ Find RECs Dept = "Sales" $\wedge$SAL $\geq$ 20k
- ☐ Find RECs Dept = "Sales"
- ☐ Find RECs SAL = 20k

Current Issues in Indexing

- Multi-dimensional indexing:
  - how do we index regions in space?
  - Document collections?
  - Multi-dimensional sales data
  - How do we support nearest neighbor queries?

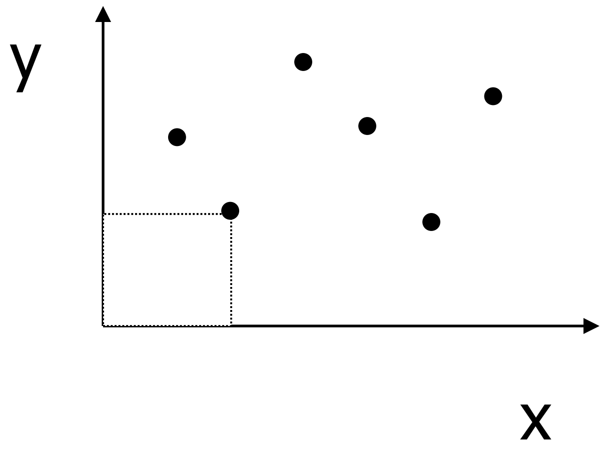
- Indexing is still a hot and unsolved problem!

Multidimensional Indexes

- Applications: geographical databases, data cubes.
- Types of queries:
  - partial match (give only a subset of the dimensions)
  - range queries
  - nearest neighbor
  - Where am I? (DB or not DB?)
- Conventional indexes don't work well here.

Interesting application:

- Geographic Data



DATA:

<X_1, Y_1, Attributes>

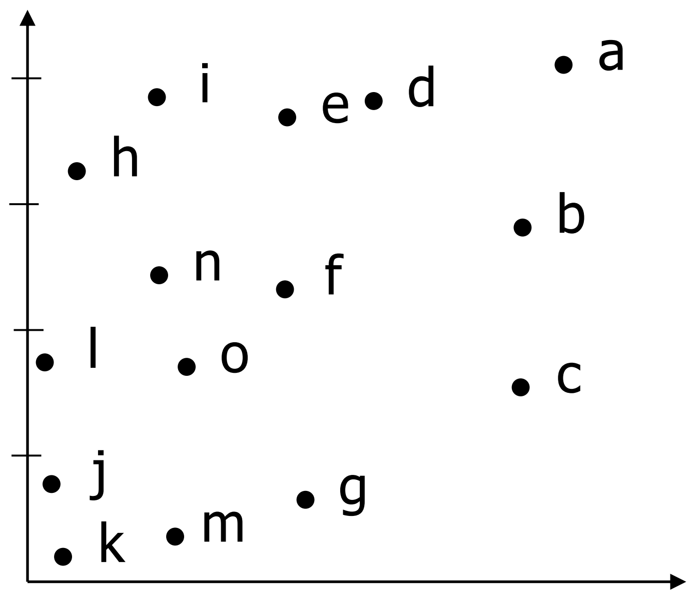
<X_2, Y_2, Attributes>

⋮

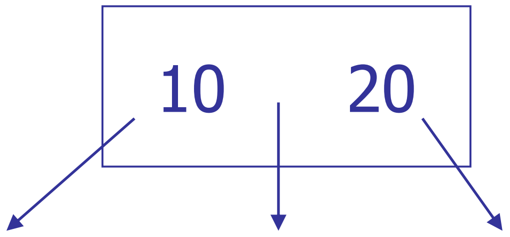
Queries:

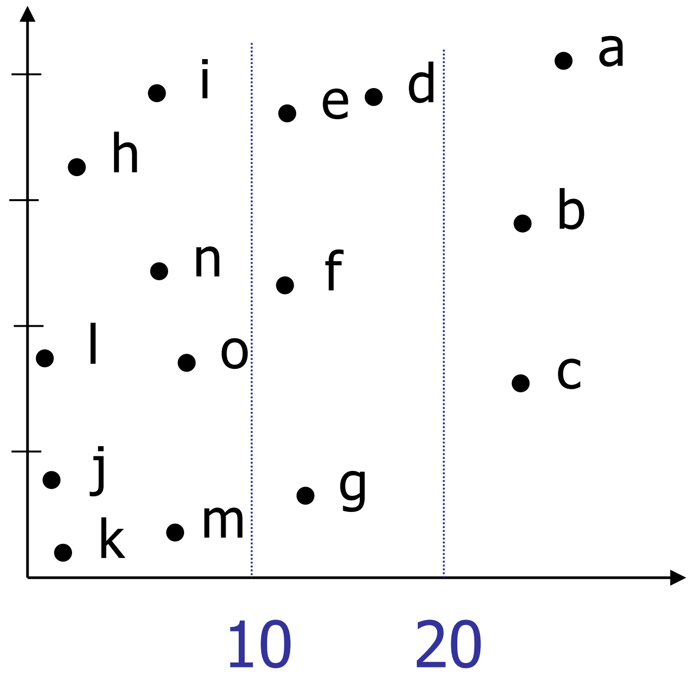
- What city is at <Xi,Yi>?
- What is within 5 miles from <Xi,Yi>?
- Which is closest point to <Xi,Yi>?

Example

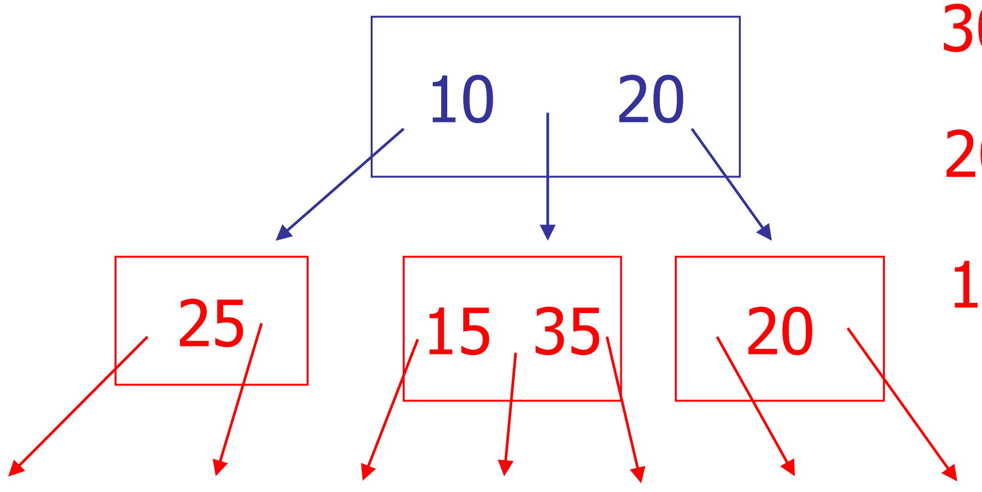


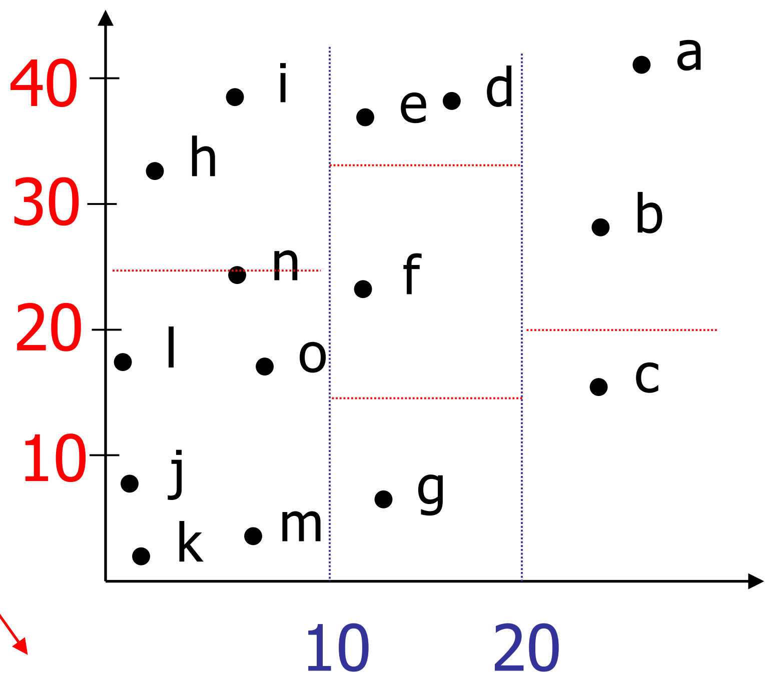
Example



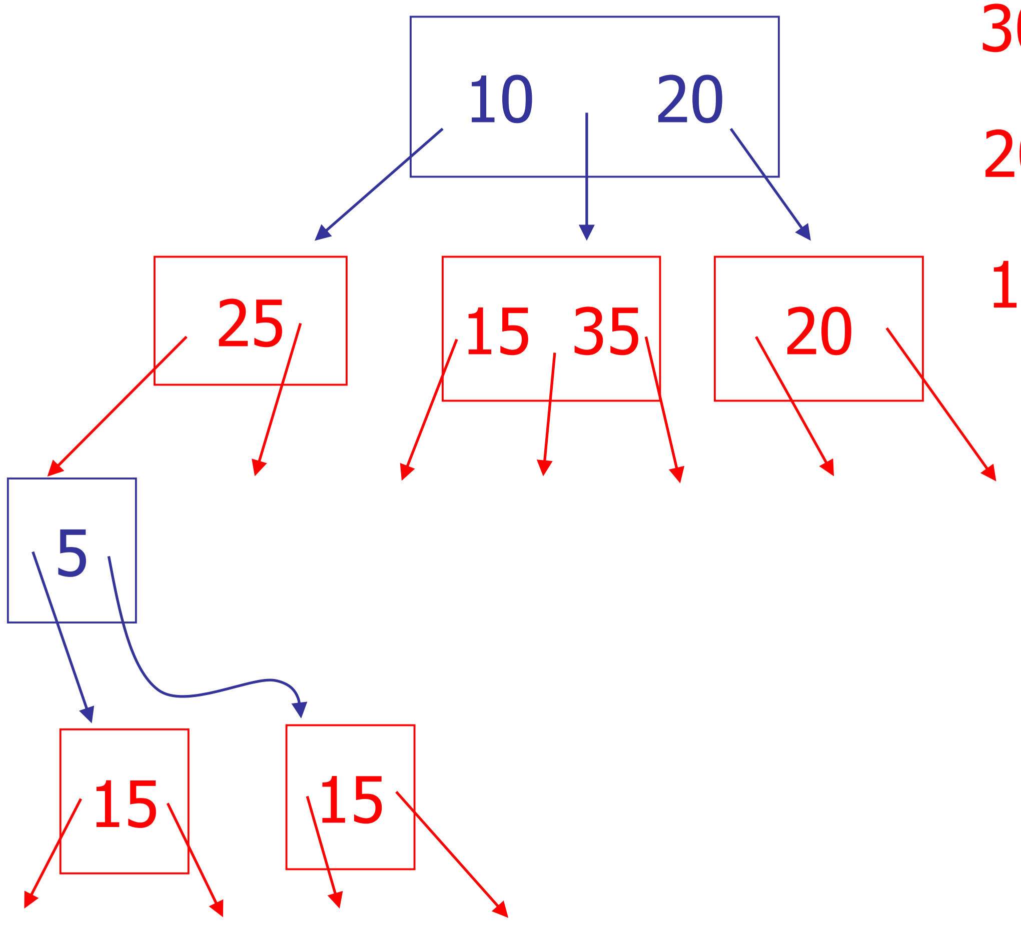


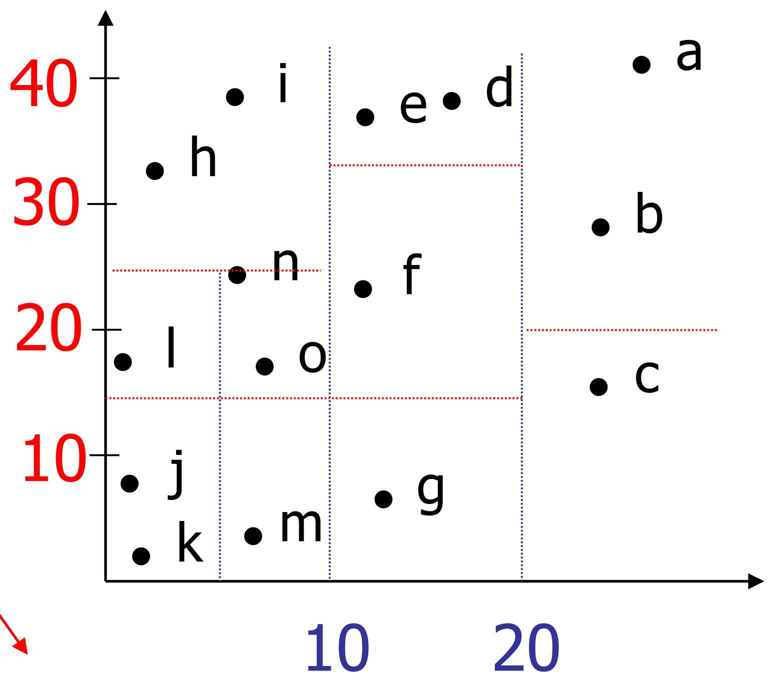
Example



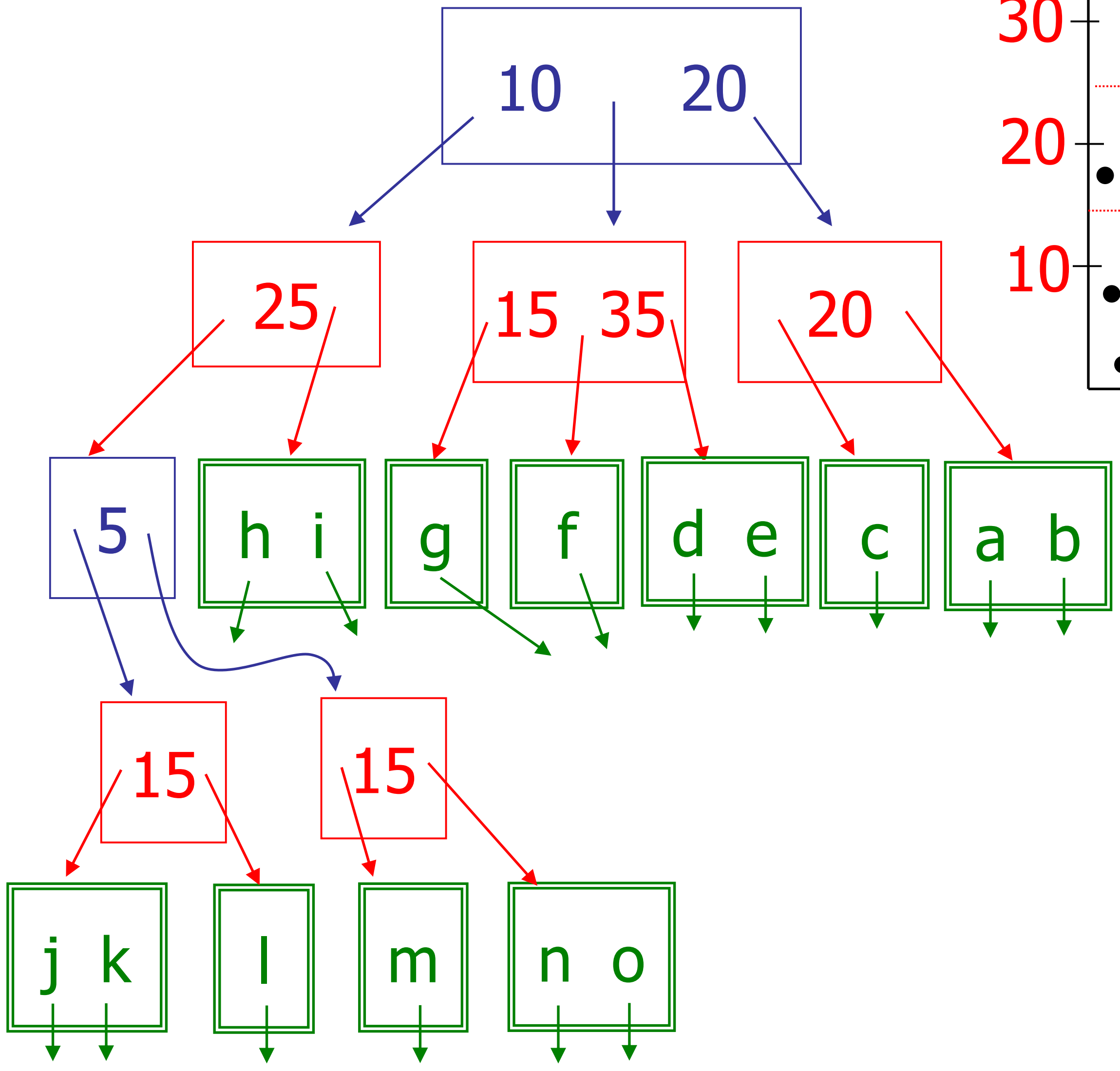


Example

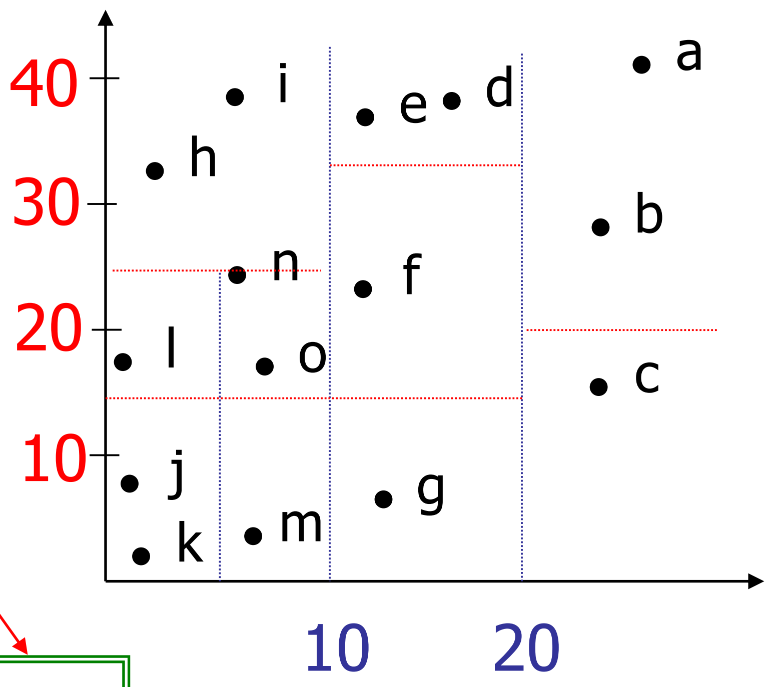




Example

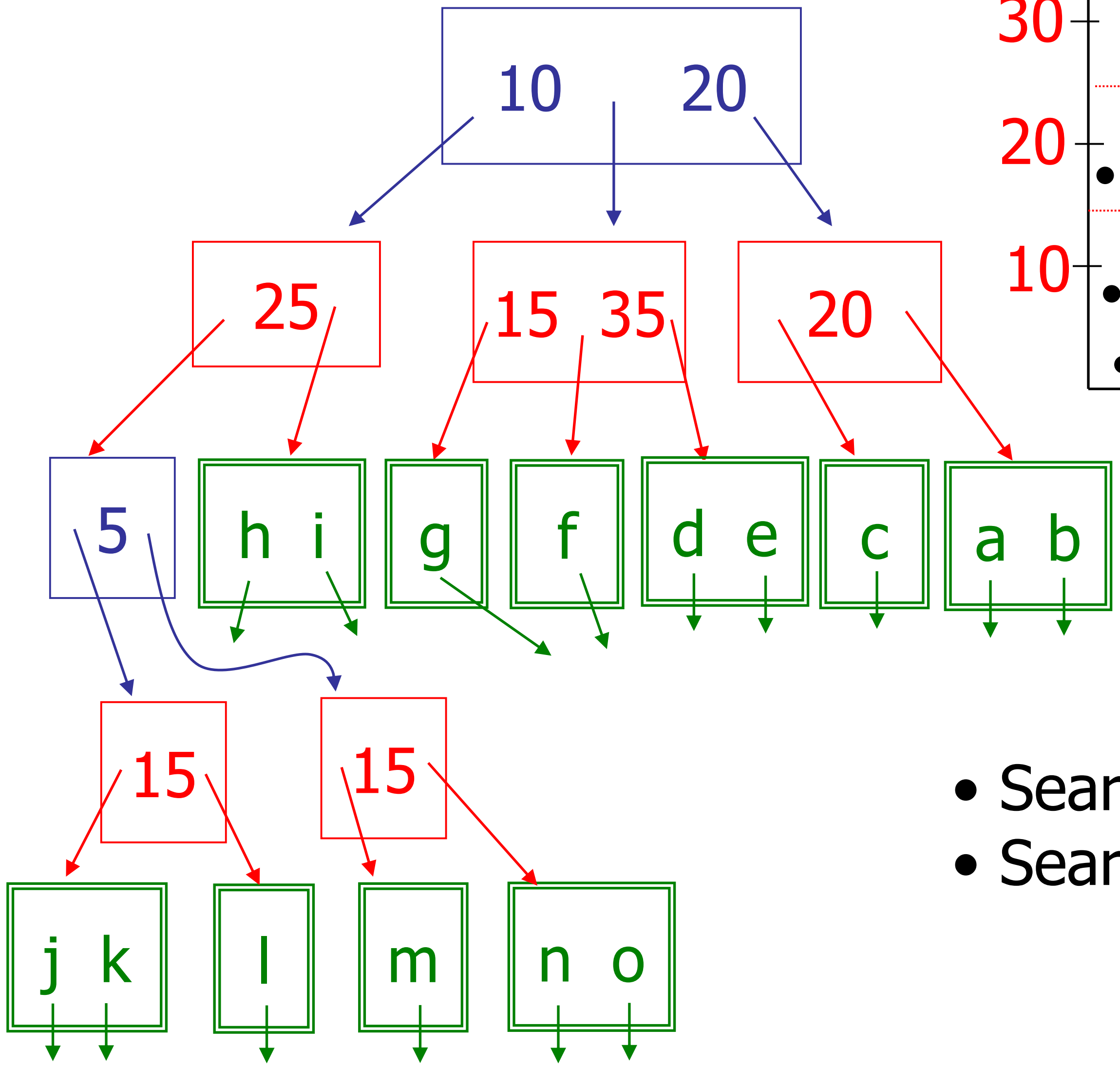


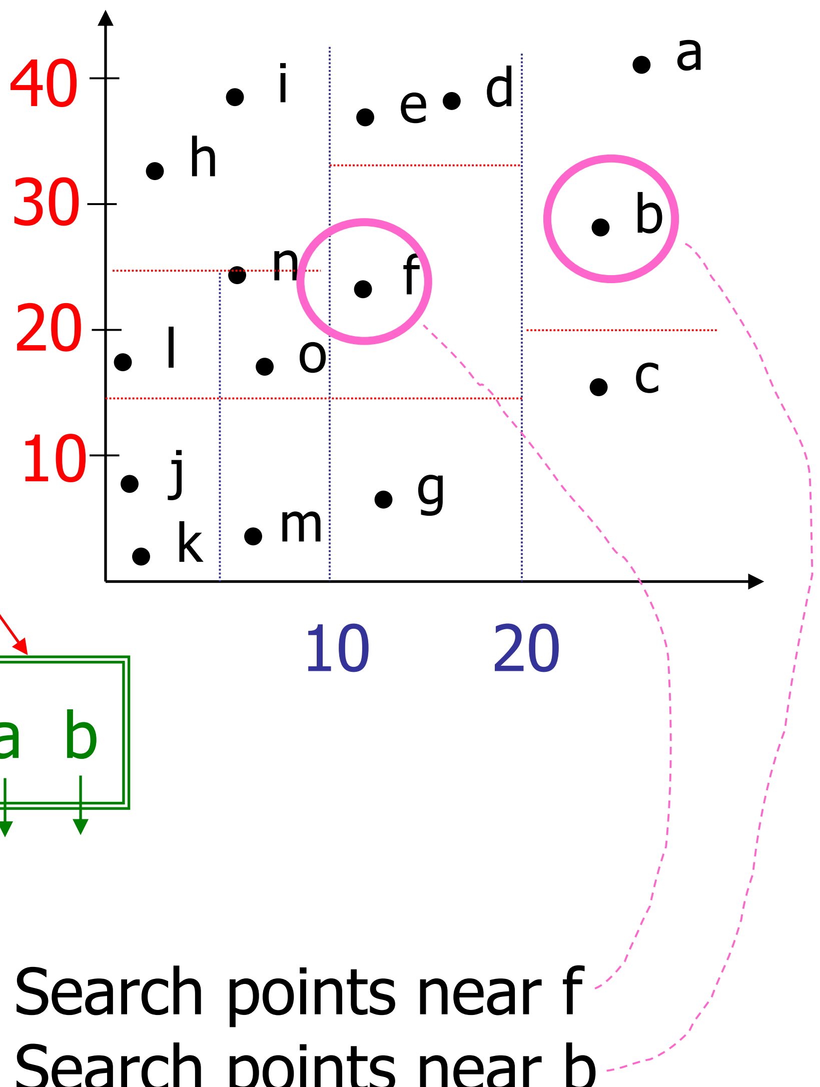
10 20
25
15 35
20
5
h i
g
f
d e
c
a b
15
15
j k
l
m
n o



40
30
20
10
10
20
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o

Example





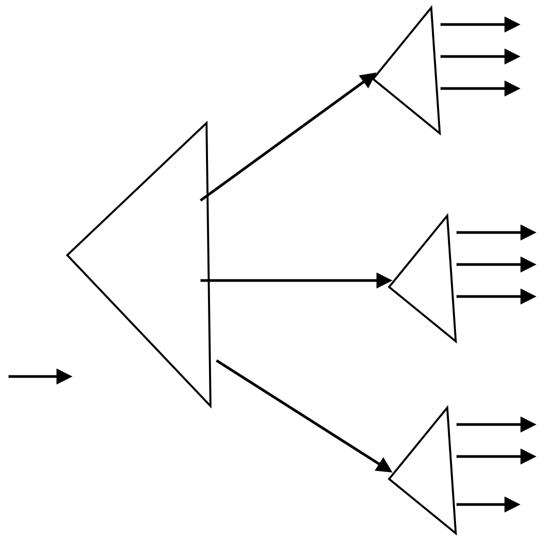
- Search points near f
- Search points near b

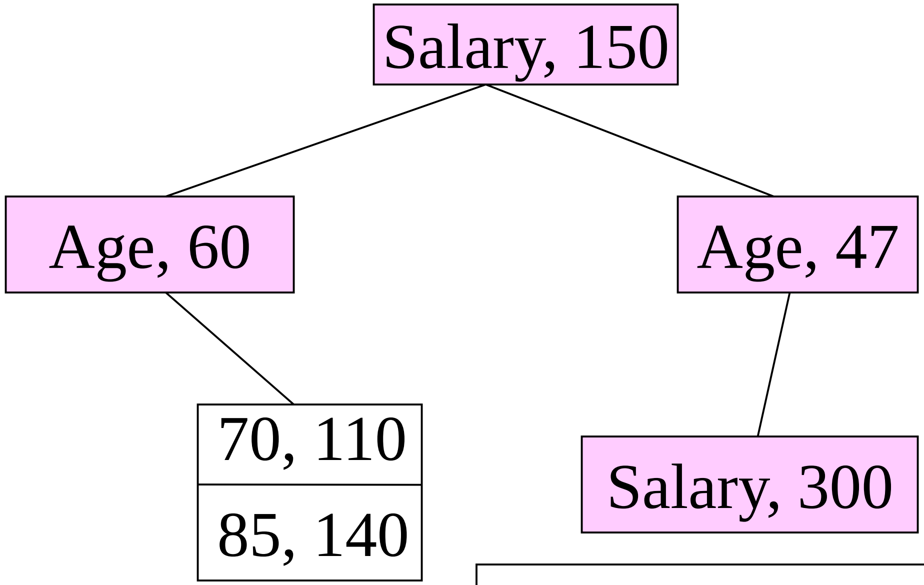
Queries

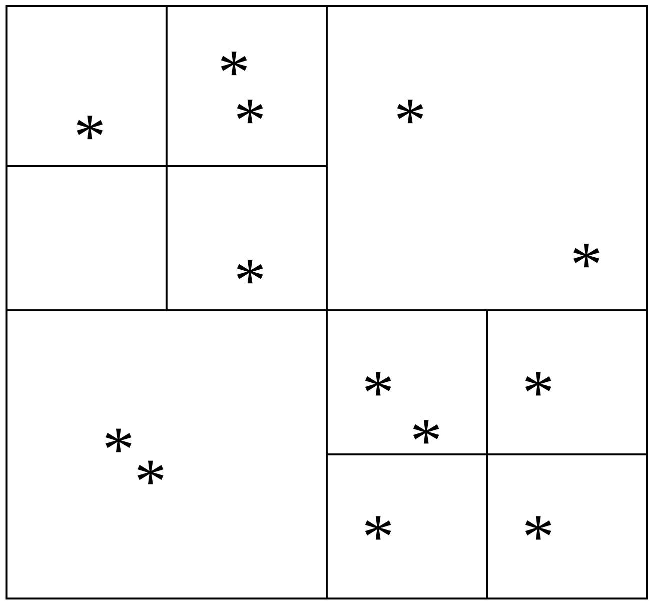
- Find points with $Y_i > 20$
- Find points with $X_i < 5$
- Find points “close” to f = <12,24>
- Find points “close” to b = <24,28>

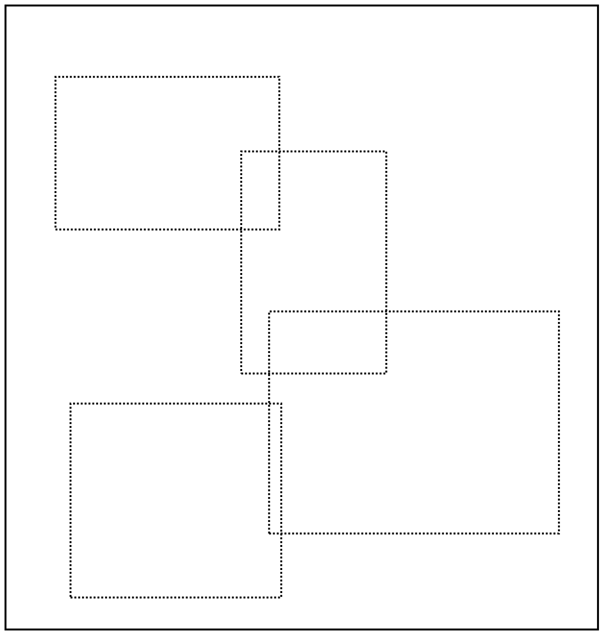
- Many types of geographic index structures have been suggested
  - kd-Trees (very similar to what we described here)
  - Quad Trees
  - R Trees
  - ...

Tree Based Indexing Techniques





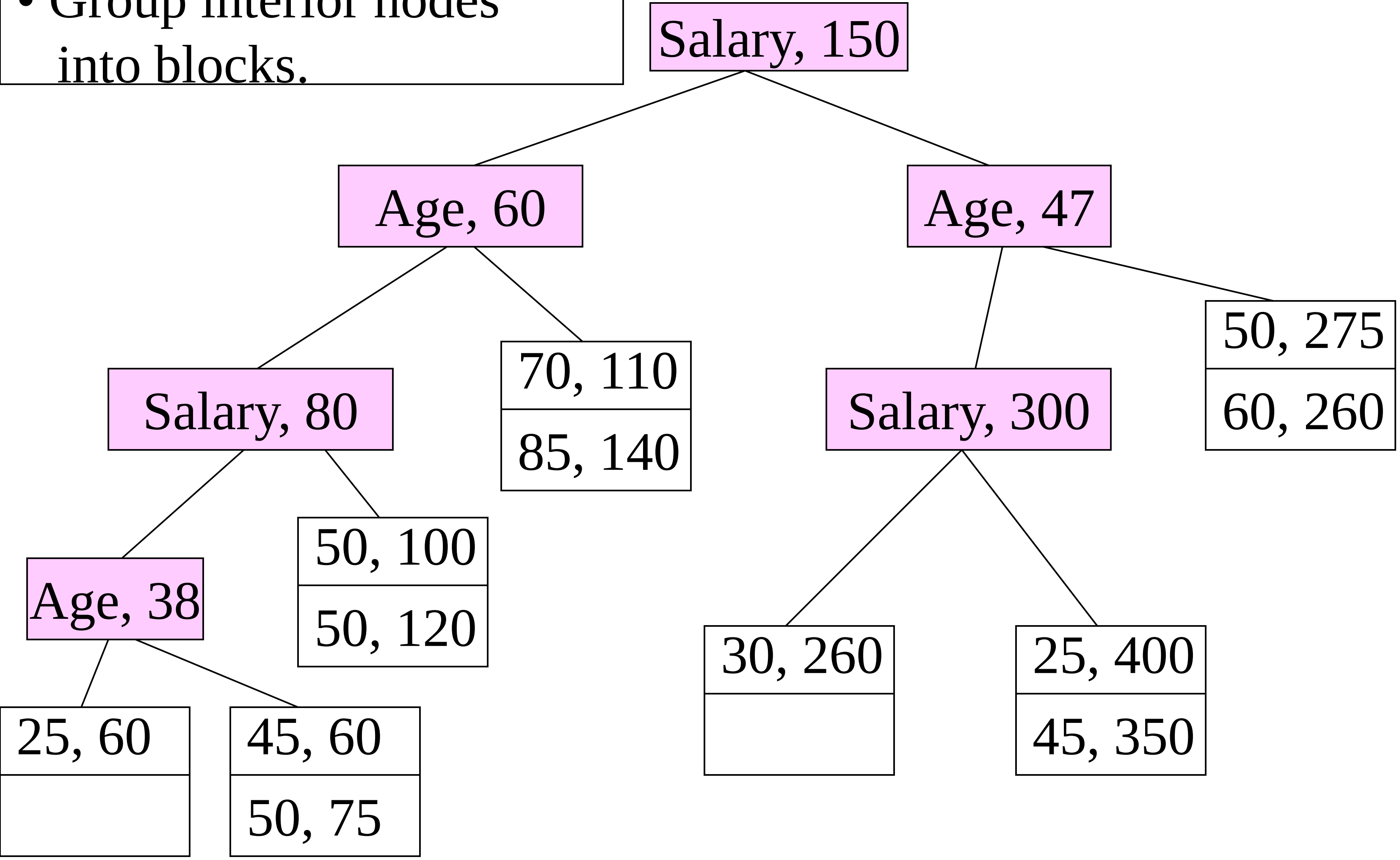




KD Trees

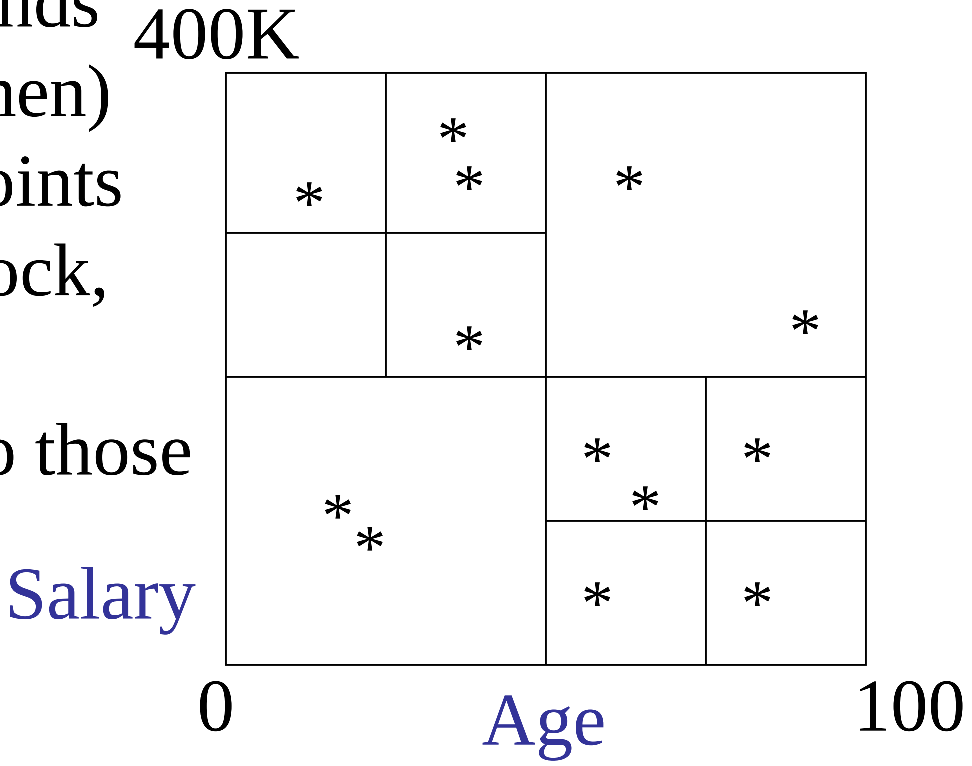
Adaptation to secondary storage:

- Allow multiway branches at the nodes, or
- Group interior nodes into blocks.



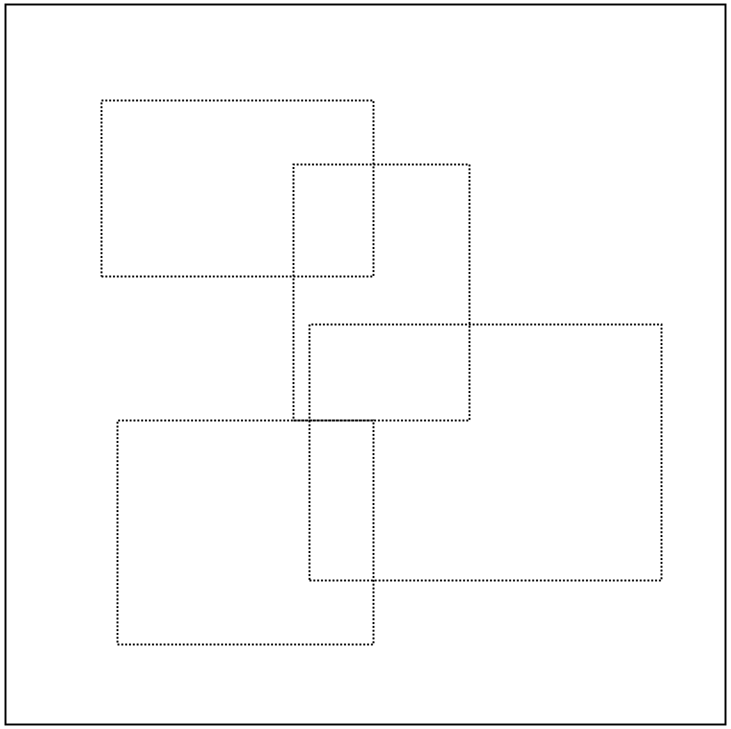
Quad Trees

- Each interior node corresponds to a square region (or k-dimen)
- When there are too many points in the region to fit into a block, split it in 4.
- Access algorithms similar to those of KD-trees.



R-Trees

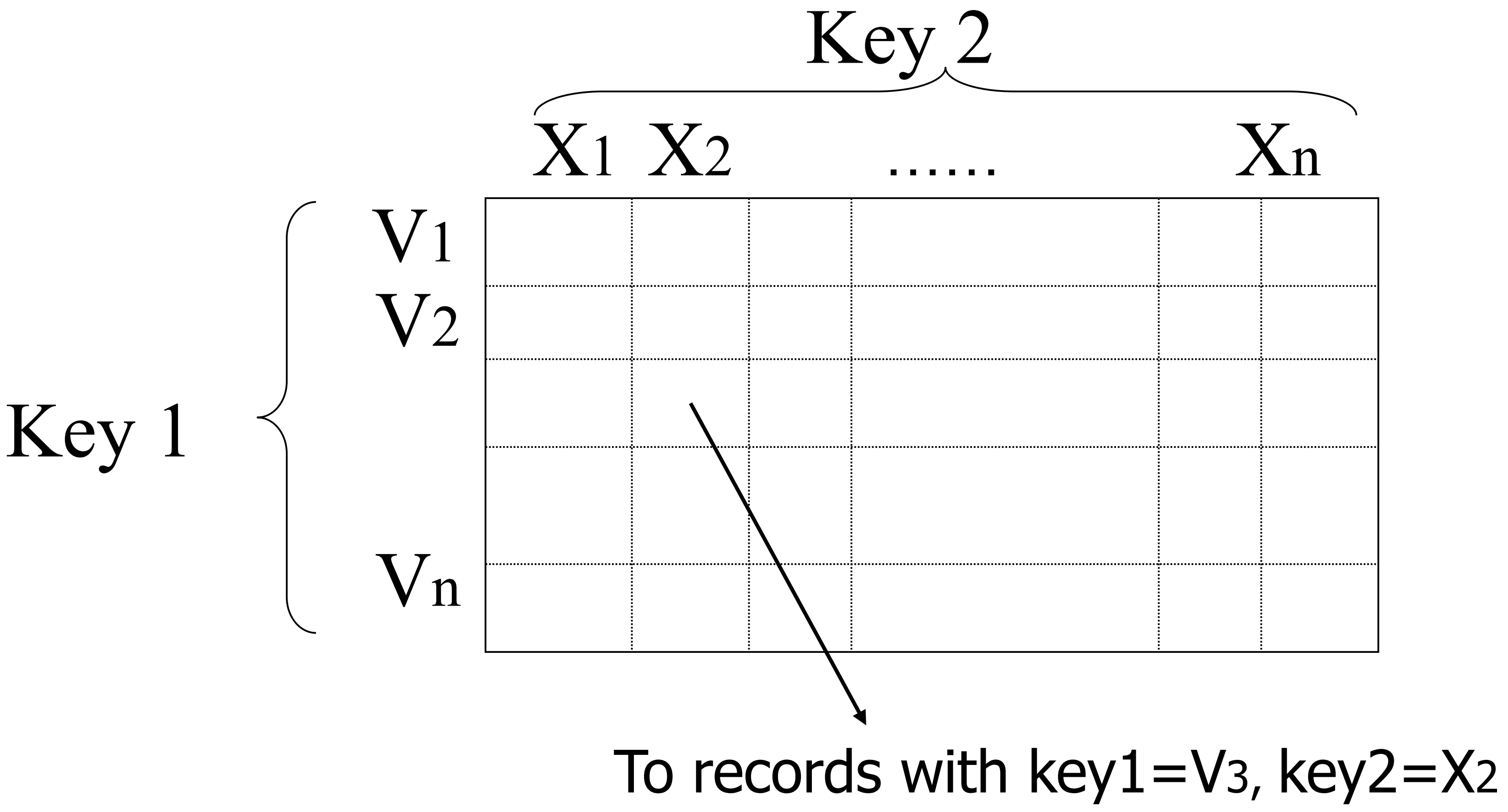
- Interior nodes contain sets of regions.
- Regions can overlap and not cover all parent's region.
- Typical query:
  - Where am I?
- Can be used to store regions as well as data points.
- Inserting a new region may involve extending one of the existing regions (minimally).
- Splitting leaves is also tricky.



Two more types of multi key indexes

- Grid
- Partitioned hash

Grid Index



CLAIM

- Can quickly find records with
  - key 1 = $V_i \wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

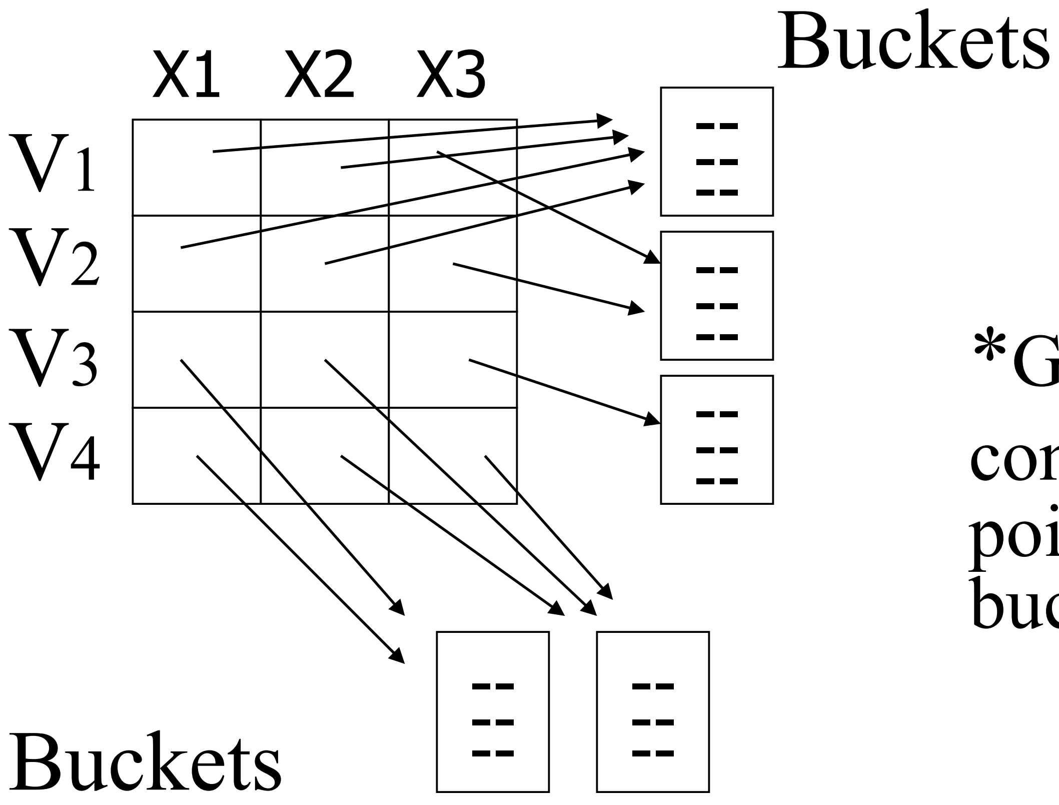
CLAIM

- Can quickly find records with
  - key 1 = $V_i \wedge$ Key 2 = X_j
  - key 1 = V_i
  - key 2 = X_j

- And also ranges….
  - E.g., key 1 $\geq V_i \wedge$ key 2 $< X_j$

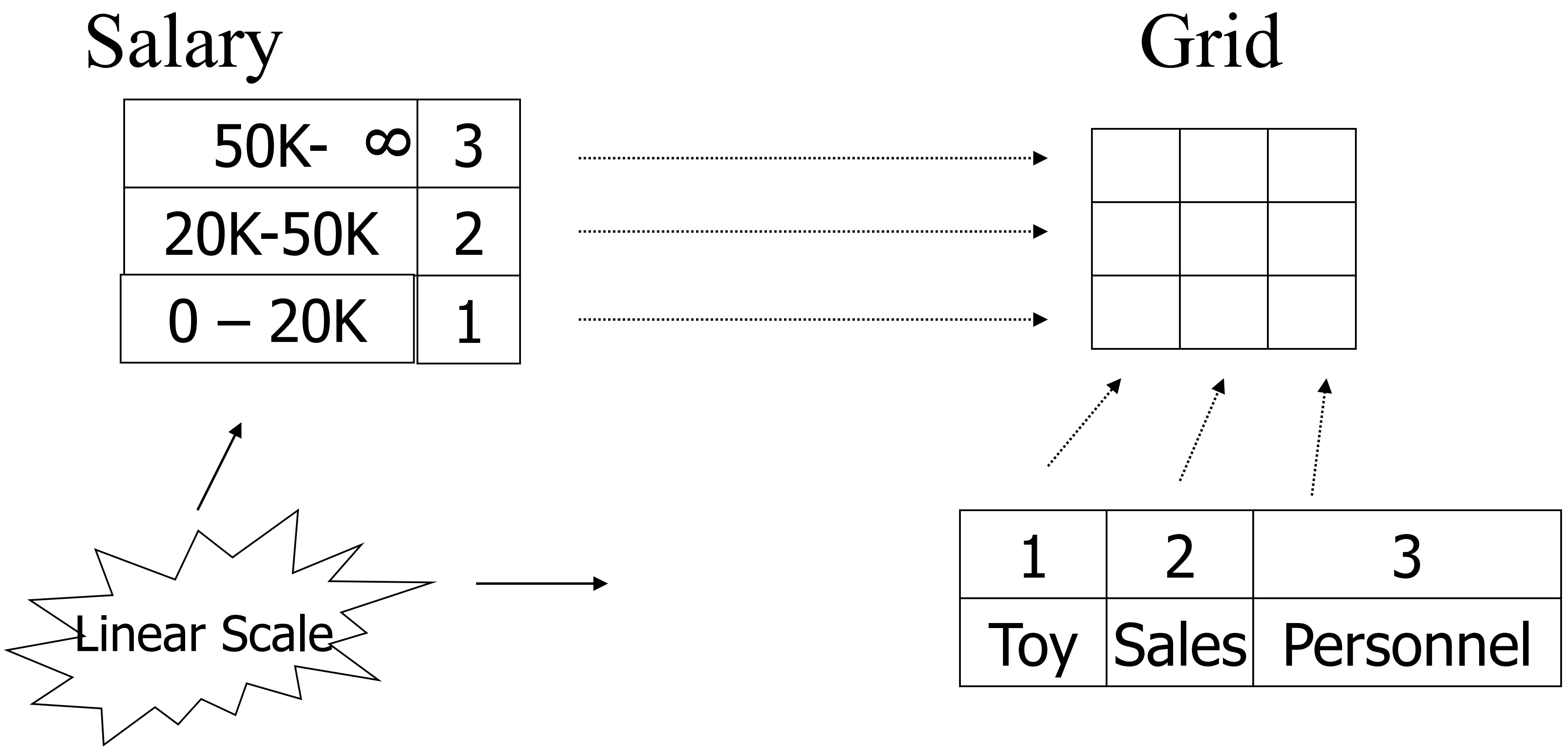
- How is Grid Index stored on disk?

Use Indirection



*Grid only contains pointers to buckets

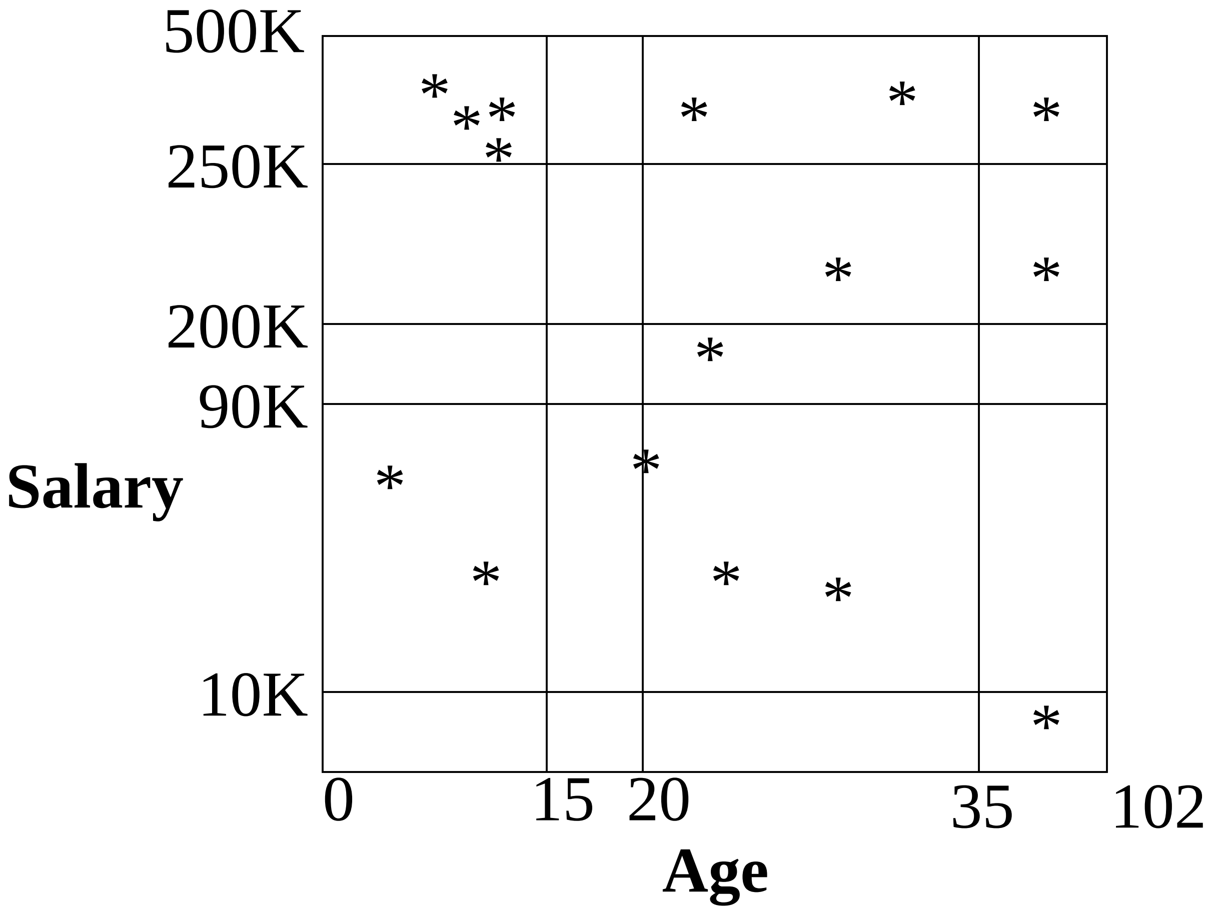
Can also index grid on value ranges



50K- ∞	3
20K-50K	2
0 – 20K	1

1	2	3
Toy	Sales	Personnel

Grid Files



- Each region in the corresponds to a bucket.
- Works well even if we only have partial matches
- Some buckets may be empty.
- Reorganization requires moving grid lines.
- Number of buckets grows exponentially with the dimensions.

Grid files

⊕ Good for multiple-key search

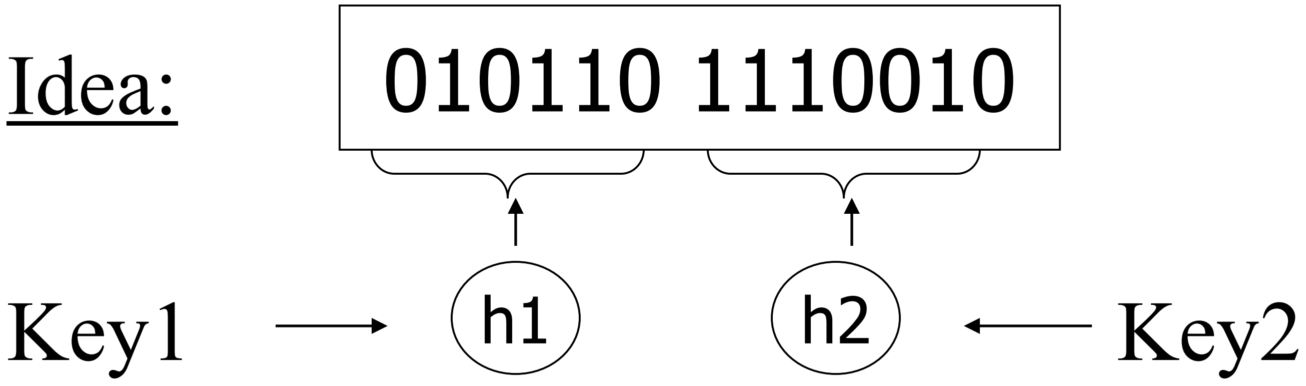
⊖ Space, management overhead

(nothing is free)

⊖ Need partitioning ranges that evenly split keys

Partitioned hash function

Idea:



Partitioned Hash Functions

- A hash function produces *k* bits identifying the bucket.
- The bits are partitioned among the different attributes.
- Example:
  - Dept produces the first bit of the bucket number.
  - Salary produces the last 2 bits.
- Supports partial matches, but is useless for range queries.

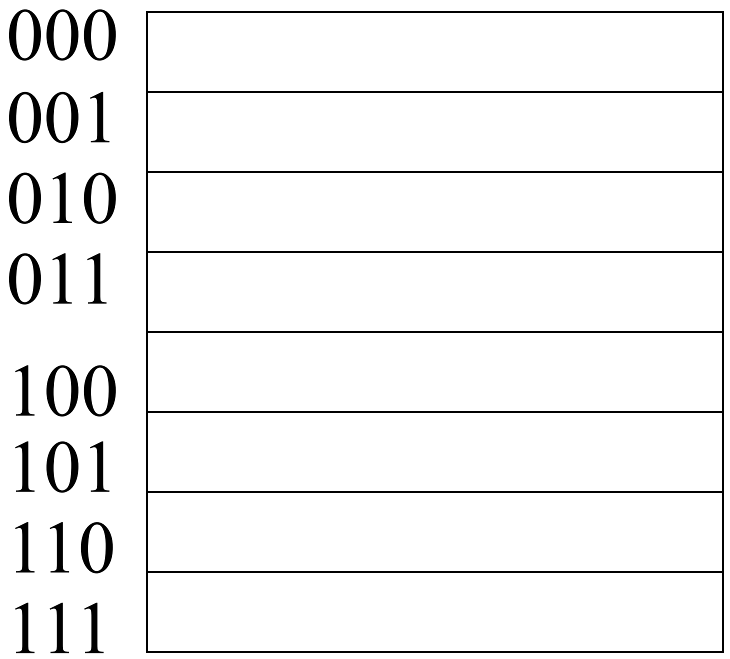
EX:

h1(toy)	=0
h1(sales)	=1
h1(art)	=1

.
.

h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00

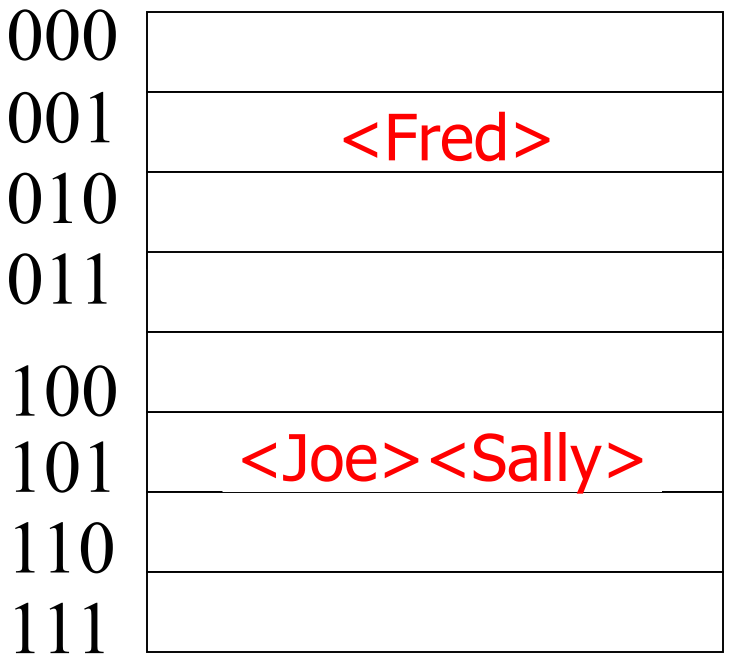
.
.



Insert ⇨ <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

EX:

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
.	
.	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
.	
.	



Insert ⇒ <Fred,toy,10k>,<Joe,sales,10k>
<Sally,art,30k>

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
:	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
:	

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales $\wedge$ Sal=40k

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
:	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
:	

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

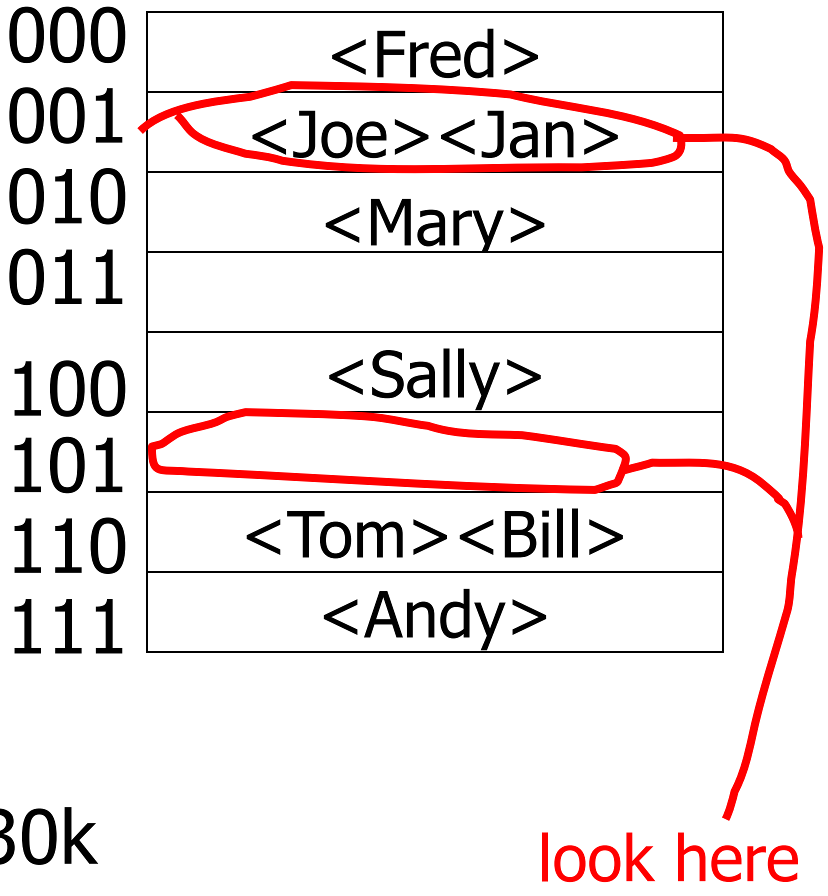
- Find Emp. with Dept. = Sales $\wedge$ Sal=40k

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
:	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
:	

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Sal=30k

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
:	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
:	



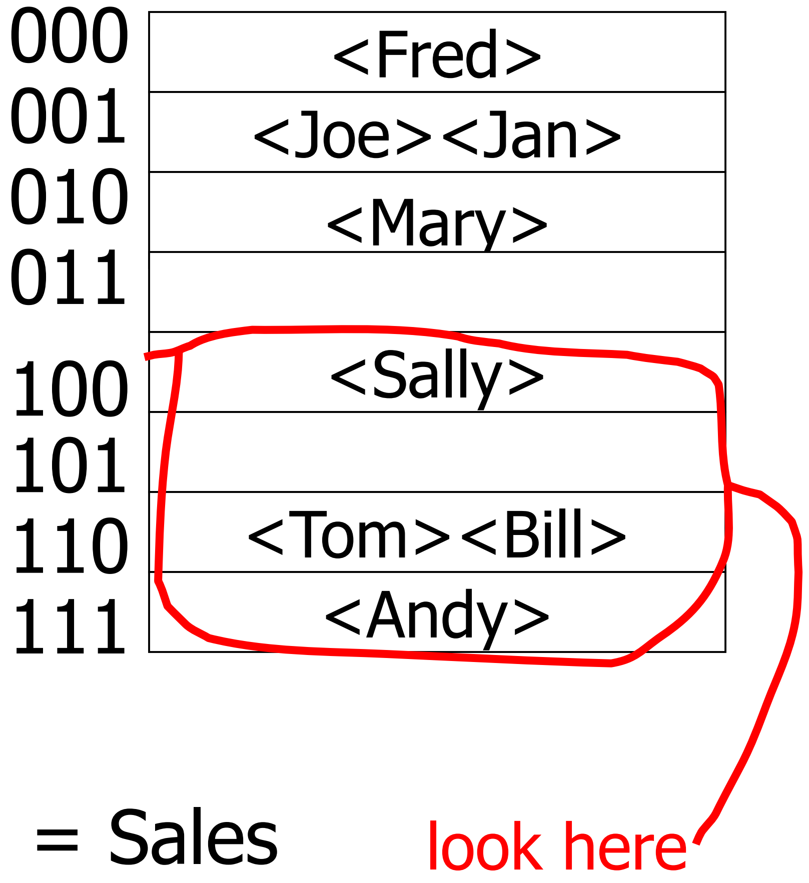
- Find Emp. with Sal=30k

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
:	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
:	

000	<Fred>
001	<Joe><Jan>
010	<Mary>
011	
100	<Sally>
101	
110	<Tom><Bill>
111	<Andy>

- Find Emp. with Dept. = Sales

h1(toy)	=0
h1(sales)	=1
h1(art)	=1
.	
.	
h2(10k)	=01
h2(20k)	=11
h2(30k)	=01
h2(40k)	=00
.	
.	



- Find Emp. with Dept. = Sales

Indexing Techniques Summary

- Hash like structures:
  - Grid files
  - Partitioned indexing functions
- Tree like structures:
  - Multiple key indexes
  - kd-trees
  - Quad trees
  - R-trees