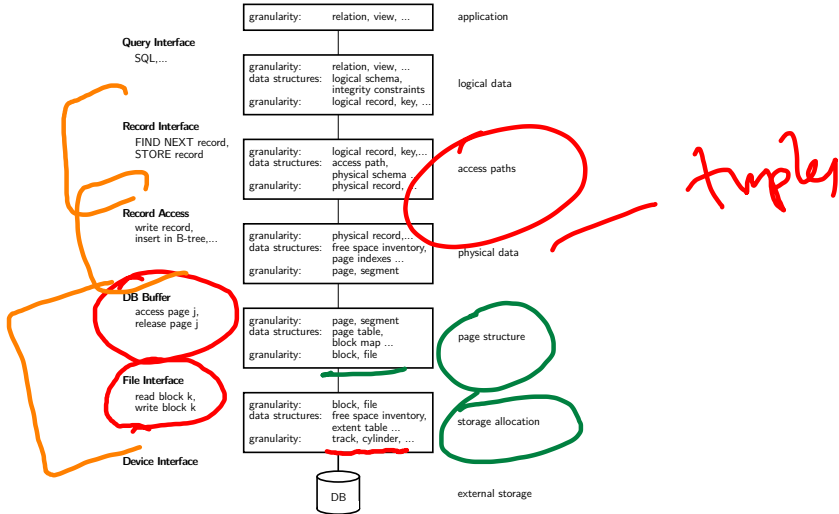




# Access Methods

# Recap

# A More Detailed Architecture



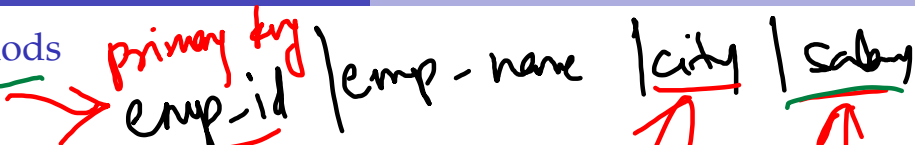
# Anatomy of a Database System [Monologue]

- Process Manager
  - ▶ Connection Manager + Admission Control
- Query Processor
  - ▶ Query Parser
  - ▶ Query Optimizer (*a.k.a.*, Query Planner)
  - ▶ Query Executor
- Transactional Storage Manager
  - ▶ Lock Manager
  - ▶ Access Methods (*a.k.a.*, Indexes)
  - ▶ Buffer Pool Manager
  - ▶ Log Manager
- Shared Utilities
  - ▶ Memory, Disk, and Networking Manager

20



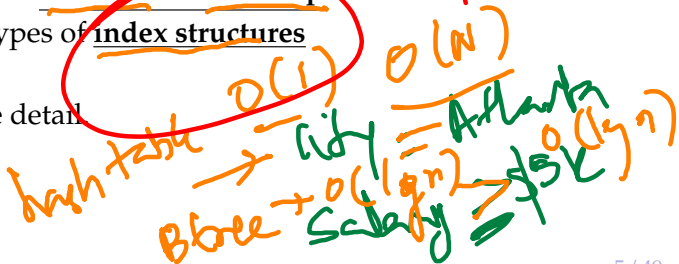
## Access Methods



Access methods are the alternative ways for retrieving specific tuples from a relation.

- Typically, there is more than one way to retrieve tuples.
- Depends on the availability of indexes and the conditions specified in the query for selecting the tuples
- Includes sequential scan method of unordered table heap
- Includes index scan of different types of index structures

We will look at these methods in more detail



# Internal Data Structures

The DBMS maintains several separate data structures

- for the data itself (storage and retrieval)
- for free space management
- for unusually large values
- for index structures to speed up access

DBMS

Seq Scan  
Page structure

(0')

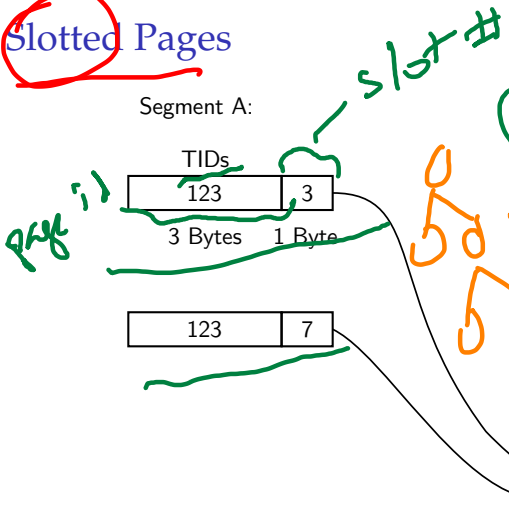
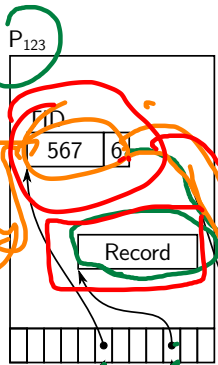
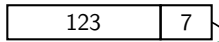
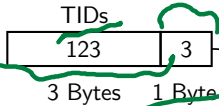
# Today's Agenda

- Sequential Access: Table Heap
- Random Access: B-Tree Index
- Random Access: Hash Index

# Sequential Access: Table Heap

# Slotted Pages

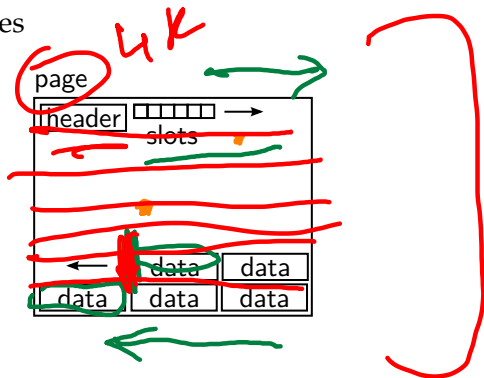
Segment A:



(TID size varies, but will most likely be at least 8 bytes on modern systems)

## Slotted Pages (2)

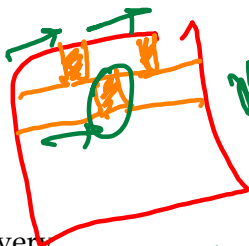
Tuples are stored in slotted pages



- data grows from one side, slots from the other
- the page is full when both meet
- updates/deletes complicate issues, though
- might require garbage collection/compactification

## Slotted Pages (3)

Header:



LSN

16K B

tuple : 100 B

LSN

slotCount

for recovery

number of used slots

firstFreeSlot

to speed up locating free slots

dataStart

lower end of the data

freeSpace

space that would be available after compactification


$$\frac{16 \times 10^3}{100} = 160$$

~ 160 slots

Note: a slotted page can contain hundreds of slots!  
 Requires careful design to get good performance.

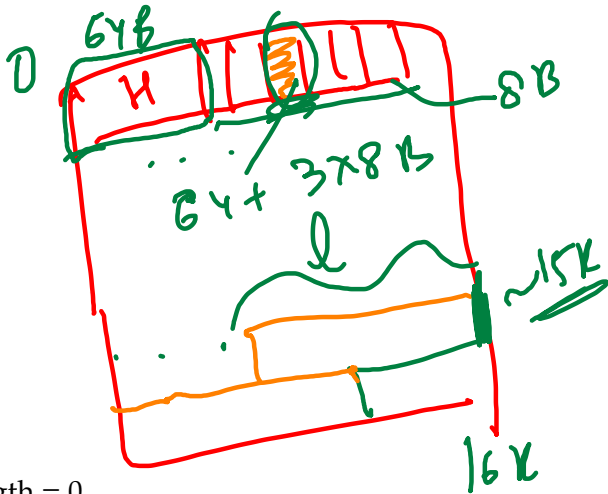
## Slotted Pages (4)

Slot:


 offset    start of the data item  
 length    length of the data item

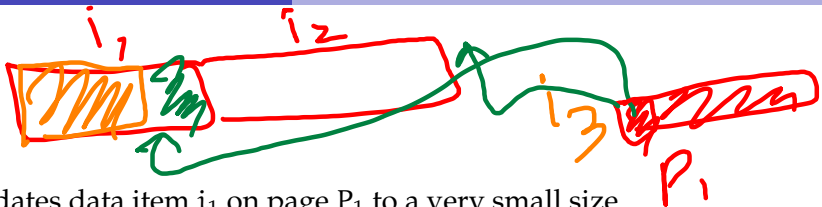
Special cases:

- free slot:  $\text{offset} = 0, \text{length} = 0$
- zero-length data item:  $\text{offset} > 0, \text{length} = 0$



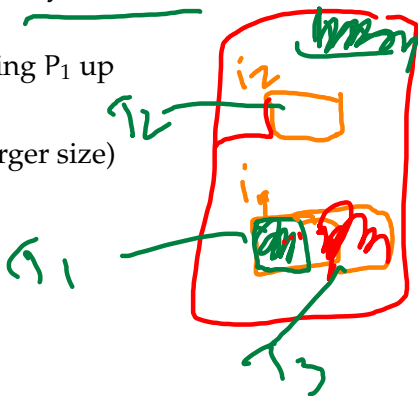


## Slotted Pages (5)



Problem:

1. transaction  $T_1$  updates data item  $i_1$  on page  $P_1$  to a very small size (or deletes  $i_1$ )
2. transaction  $T_2$  inserts a new item  $i_2$  on page  $P_1$ , filling  $P_1$  up
3. transaction  $T_2$  commits
4. transaction  $T_1$  aborts (or  $T_3$  updates  $i_1$  again to a larger size)



TID concept  $\Rightarrow$  create an indirection  
but where to put it? Would have to move  $i_1$  and  $i_2$ .

## Slotted Pages (6)

Logic is much simpler if we can store the TID inside the slot

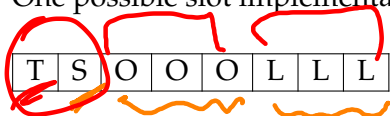
- borrow a bit from the TID (or have some other way to detect invalid TIDs)
- if the slot contains a valid TID, the entry is redirected
- otherwise, it is a regular slot

Depending on page size size, this wastes a bit space.  
But greatly simplifies the slotted page implementation.

## Slotted Pages (7)

lab 5

One possible slot implementation:



1. if  $T \neq 11111111_b$ , the slot points to another record
2. otherwise the record is on the current page
  - 2.1 if  $S = 0$ , the item is at offset  $O$ , with length  $L$
  - 2.2 otherwise, the item was moved from another page
    - ▶ it is also placed at offset  $O$ , with length  $L$
    - ▶ but the first 8 bytes contain the original TID

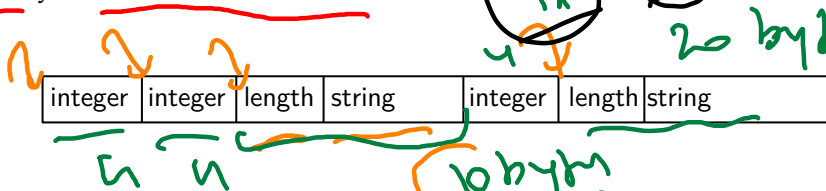
regular slot

The original TID is important for scanning.

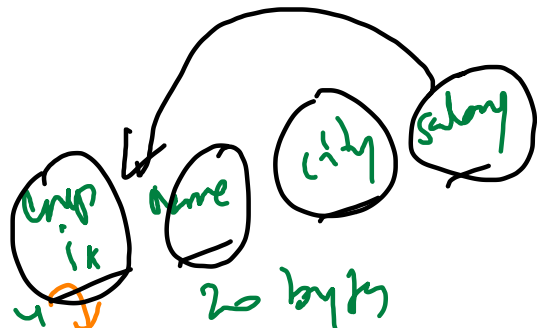
## Record Layout

The tuples have to be materialized somehow.

One possibility: serialize the attributes

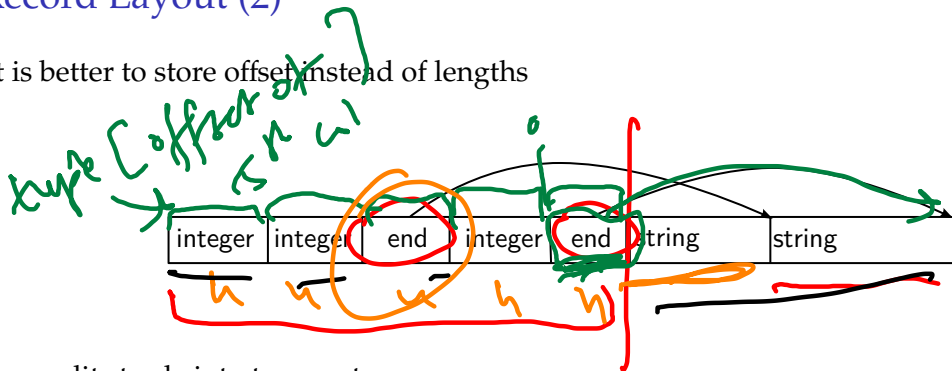


Problem: accessing an attribute is  $O(n)$  in worst case.



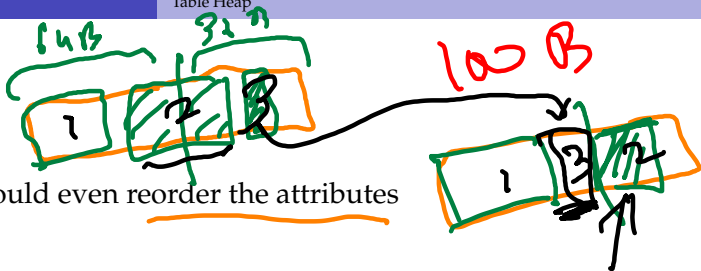
## Record Layout (2)

It is better to store offset instead of lengths



- splits tuple into two parts
- fixed size header and variable size tail
- header contains pointers into the tail
- allows for accessing any attribute in  $O(1)$

## Record Layout (3)



For performance reasons one should even reorder the attributes

- split strings into length and data
- re-order attributes by changing alignment
- place variable-length data at the end
- variable length: alignment = 1

Data alignment

64B

Gives better performance without wasting any space on padding.

## NULL Values

What about NULL values?

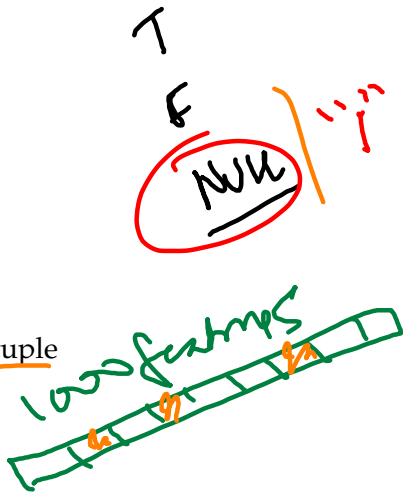
- represent an unknown/unspecified value
- is a special value outside the regular domain

Multiple ways to store it

- either pick an invalid value (not always possible)
- or use a separate NULL bit

NULL bits allow for omitting NULL values from the tuple

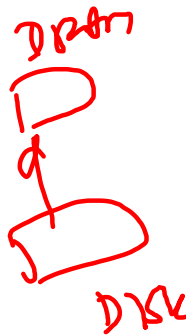
- complicates the access logic
- but saves space
- useful if NULL values are common.



# Compression

Some DBMS apply compression techniques to the tuples

- most of the time, compression is **not** added to save space
- disk is cheap after all
- compression is used to **improve performance**
- reducing the database size reduces disk bandwidth consumption



Some people really care about space consumption, of course.  
But outside embedded DBMSs it is usually an afterthought.



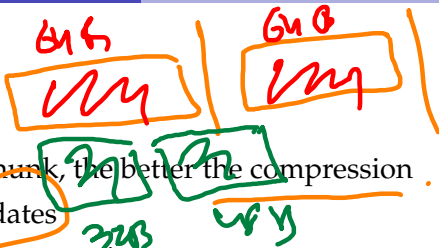
## Compression (2)

What to compress?

- the larger data compressed chunk, the better the compression.
- but: DBMS has to handle updates
- usually rules out page-wise compression
- individual tuples can be compressed more easily

How to compress?

- general purpose compression like LZ77 too expensive
- compression is about performance, after all
- most system use special-purpose compression
- byte-wise to keep performance reasonable



Workload type  
compression ratio

RLR, Dict

## Compression (3)

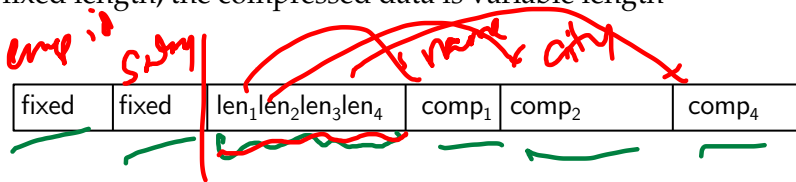
A useful technique for integer: variable length encoding

length (2 bits)	data (0-4 bytes)
-----------------	------------------

	Variant A	Variant B
00	1 byte value	NULL, 0 bytes value
01	2 bytes value	1 byte value
10	3 bytes value	2 bytes value
11	4 bytes value	4 bytes value

## Compression (4)

The length is fixed length, the compressed data is variable length



Problem: locating compressed attributes

- depends on preceding compression
- would require decompressing all previous entries
- not too bad, but can be sped up
- use a lookup tuples per length byte

## Compression (5)

Another popular technique: dictionary compression

Dictionary:

1	Berlin
2	München
3	Passauerstraße
...	...

Tuples:

city	street	number
1	3	5
2	3	7
...	...	...

- stores strings in a dictionary
- stores only the string id in the tuple
- factors out common strings
- can greatly reduce the data size
- can be combined with integer compression

## Long Records

BLOB

1MB  
JOY

sykhate

Data is organized in pages

- many reasons for this, including recovery, buffer management, etc.
- a tuple must fit on a single page
- limits the maximum size of a tuple

16 KB

What about large tuples?

- sometimes the user wants to store something large
- e.g., embed a document
- SQL supports this via BLOB (Binary Large Object)/CLOB (Character Large Object)

o racle

Requires some mechanism so handle these large records.

## Long Records (2)

Simply spanning pages is not a good idea:

- must read an unbounded number of pages to access a tuple
- greatly complicates buffering
- a tuple might not even fit into main memory!
- updates that change the size are complicated
- intermediate results during query processing

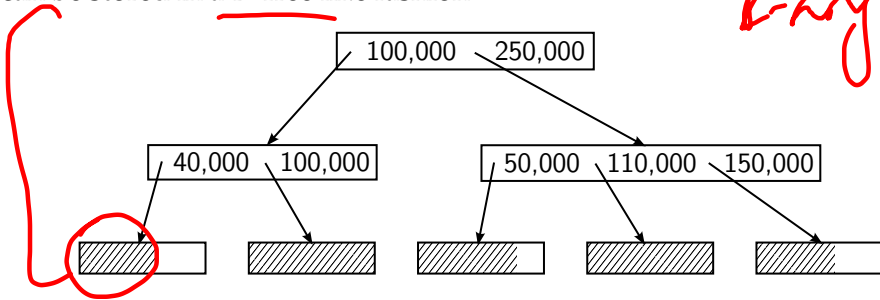
1970s  
2MB

Instead, keep the main tuple size down

- BLOBS/CLOBS are stored separate from the tuple
- tuple only contains a pointer
- increases the costs of accessing the BLOB, but simplifies tuple processing

## Long Records (3)

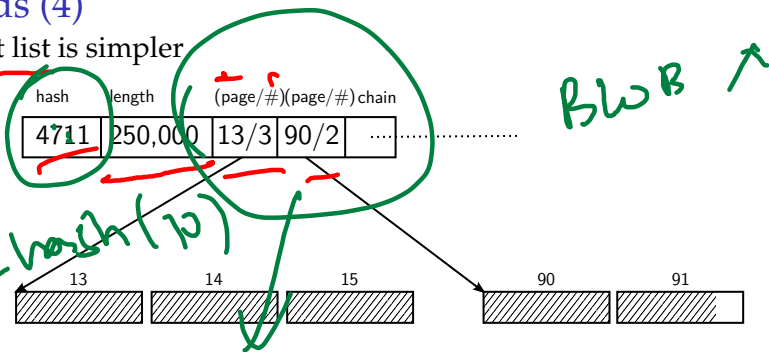
BLOBs can be stored in a B-Tree like fashion



- (relative) offset is search key
- allows for accessing and updating arbitrary parts
- very flexible and powerful
- but might be over-sophisticated
- SQL does not offer this interface anyway

## Long Records (4)

Using an extent list is simpler



- real tuple points to BLOB tuple
- BLOB tuple contains a header and an extent list
- in worst case the extent list is chained, but should rarely happen
- extent list only allows for manipulating the BLOB in one piece
- but this is usually good enough
- hash and length to speed up comparisons



## Long Records (5)

It makes sense to optimize for short BLOBs/CLOBs

- users misuse BLOBs/CLOBs
- they use CLOB to avoid specifying a maximum length
- but most CLOBs are short in reality
- on the other hand some BLOBs are really huge
- the DBMS cannot know
- so BLOBs can be arbitrary large, but short BLOBs should be more efficient

Approach:

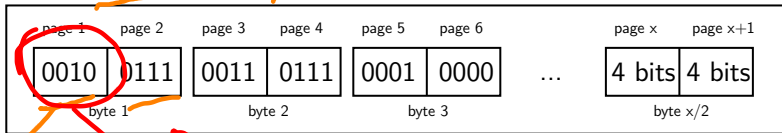
1. BLOBs smaller than TID are encoded in BLOB TID
2. BLOBs smaller than page size are stored in BLOB record
3. only larger BLOBs use the full mechanism

# Free Space Inventory

Problem: Where do we have space for incoming data?

*data used*

Traditional solution: free space bitmap



*0000  
1111 — full*

*0001 — 16 KB  
0000 — 256 B*

Each 4-bit nibble indicates the fill status of a given page.

*0010*

## Free Space Inventory

Encode the fill status in 4 bits (some system use only 1 or 2):

- must approximate the status
- one possibility:  $\text{data size} / \frac{\text{page size}}{2^{\text{bits}}}$
- loss of accuracy in the lower range
- logarithmic scale is often better:  $\lceil \log_2(\text{free size}) \rceil$
- or a combination
- 8 states: linear for upper range | 8 states: logarithmic for lower range
- 16: FULL, 15: 8 B, ..., 9: 512 B | 8: 256 B, ..., 1: 4 B

linear

16 KB

fs: 256 B

Encodes the free space (alternative: the used space) in a few bits.

# Free Space Inventory FSI

When inserting data,

- compute the required FSI entry (e.g.,  $\leq 7$ )
- scan the FSI for a matching entry
- insert the data on this page
- update the FSI entry if needed

27  
0010  
60001

# Free Space Inventory

## Problem:

- linear sequential scan
- FSI is small. With 16 KB pages, 1 FSI page covers 512 MB.
- but scan still not free
- only 16 FSI values, cache the next matching page (range)
- most pages will be static (and full anyway)
- segments will mostly grow at the end
- caching FSI state avoids scanning most of the FSI entries

*over-time*

*100 GB*

## Space Allocation

1500p  
16 KB  
↓  
Drops

Allocating pages (or parts of a page) benefits from application knowledge

- e.g., a set of tuples may be inserted in a sequence
- or one very large data item
- should be allocated close to each other

Allocation interface is usually

`allocate_space(min, max)`

Example:

`allocate_space(200 B, 20 KB)`

- max is a hint to improve spatial data locality
- some interfaces (e.g., segment growth) even implement over-allocation
- reduces fragmentation

# Index Structures

Data is often indexed

- speeds up lookup
- de-facto mandatory for primary keys
- useful for selective queries

index scan

Two important access classes:

- point queries

find all tuples that take a given value for particular column

- range queries

find all tuples that take a given range of values for a particular column

city = Atlanta  
salary  $\in [1000, 2000]$

Support for more complex predicates is rare.

# Random Access: B-Tree Index



# B-Tree

1970s

B-Trees (including variants) are the dominant index data structure for external storage.

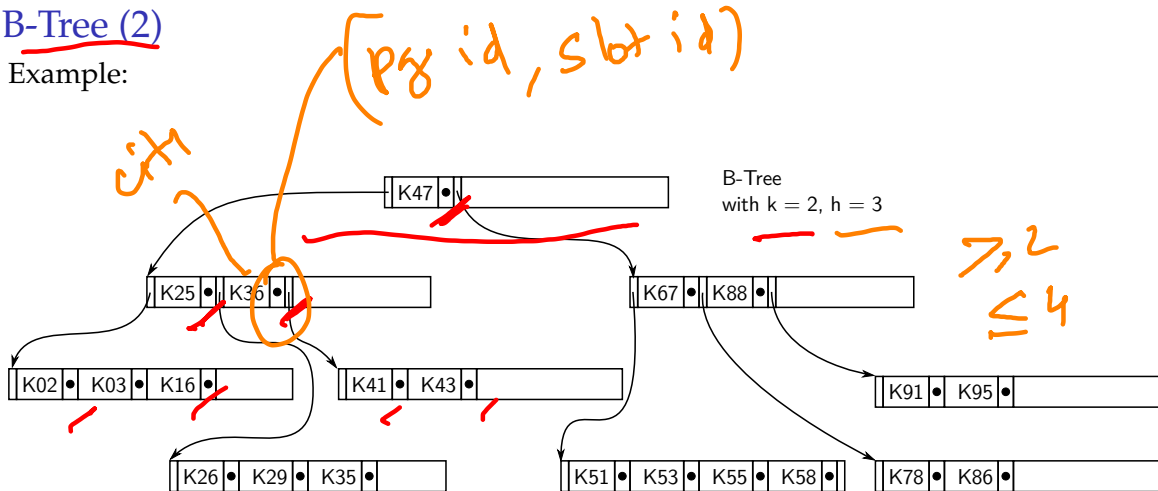
Classical definition:

- a B-Tree has a degree  $k$
- each node except the root has at least  $k$  entries
- each node has at most  $2k$  entries
- all leaf nodes are at the same depth

$k$ -ary tree  
2-ary tree

## B-Tree (2)

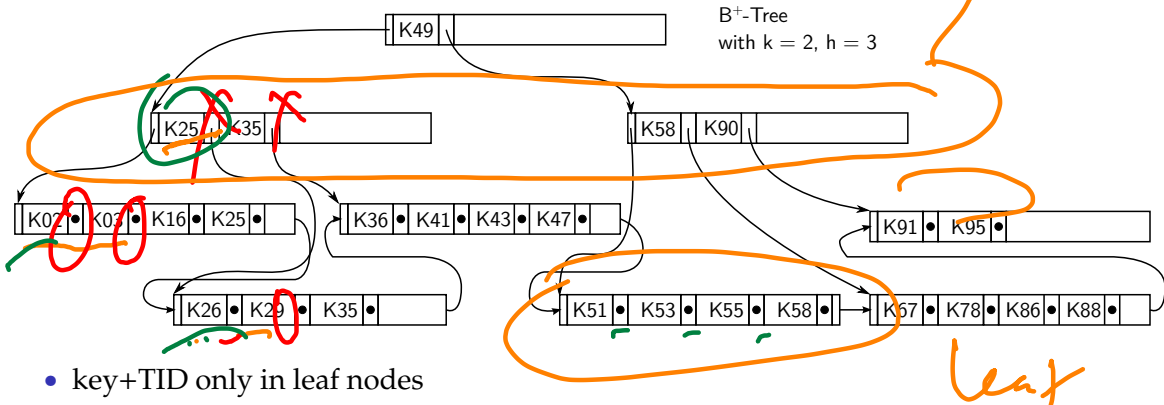
Example:



The • is the TID of the corresponding tuple.

# B<sup>+</sup>-Tree

Most DBMS use the B<sup>+</sup>-Tree variant



- key+TID only in leaf nodes
- inner nodes contain separators, might or might not occur in the data
- increases the fanout of inner nodes
- simplifies the B-Tree logic

## Page Structure

Inner Node:

LSN for recovery  
 upper page of right-most child ✓  
 count number of entries ✓  
 key/child key/child-page pairs ✓

... *other node ptr*

Leaf Node:

LSN for recovery  
 ~0 ✓ leaf node marker  
 next next leaf node ✓  
 count number of entries ✓  
 key/tid key/TID pairs ✓

... *suplex*

Similar to slotted pages for variable keys.

# Random Access: Hash Index

city = Atlanta

## Hash-Based Indexes

$O(1)$

$hash(T)$

In main memory a hash table is usually faster than a search tree

- compute a hash-value  $h$ , compute a slot (e.g.,  $s = h \bmod |T|$ ), access the table  $T[s]$
- promises  $O(1)$  access
- (if everything works out fine)

A DBMS could profit from this, too. But:

- random I/O is very expensive on disk
- collisions are problematic (e.g., when chaining)
- rehashing is prohibitive

CPU op.

But there are hashing schemes for external storage.

## Hash-Based Indexes (2)

Hash indexes are not as versatile as tree indexes:

- only support point query
- order-preserving hashing exists, but does **not** work well
- choice of the hash function is critical

As a consequence, mainly useful for primary key indexes

- unique keys
- key collisions would be very dangerous
- for other attributes, need to support duplicates (complicated)

# Conclusion

- Access methods are the alternative ways for retrieving specific tuples
- Two types of access methods: sequential scan and index scan
- Sequential scan is done over an unordered table heap (base data structure)
- Index scan is done over an ordered B-Tree or an unordered hash table (or another derived data structure)
- In the next lecture, we will learn about hash indexes