Access Paths

Data Structures

The DBMS needs several separate data structures

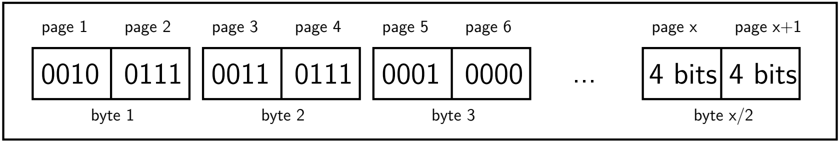
- for the free space management
- for the data itself (storage and retrieval)
- for unusually large data
- for index structures to speed up access

We will look at these in more detail.

Free Space Inventory

Problem: Where do we have space for incoming data?

Traditional solution: free space bitmap



Each nibble indicates the fill status of a given page.

Free Space Inventory (2)

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

- must approximate the status
- one possibility: data size / $\frac{\text{page size}}{2^{bits}}$
- loss of accuracy in the lower range
- logarithmic scale is often better
- $\lceil \log_2(\text{text size}) \rceil$
- or a combination (logarithmic for lower range, linear for upper range)

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

Free Space Inventory (3)

When inserting data,

- compute the required FSI entry (e.g., ≤ 7)
- scan the FSI for a matching entry
- insert the data on this page

Problem:

- linear effort
- FSI is small, for 16KB pages 1 FSI page covers 512MB
- 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
- cache avoids scanning most of the FSI entries

Allocation

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

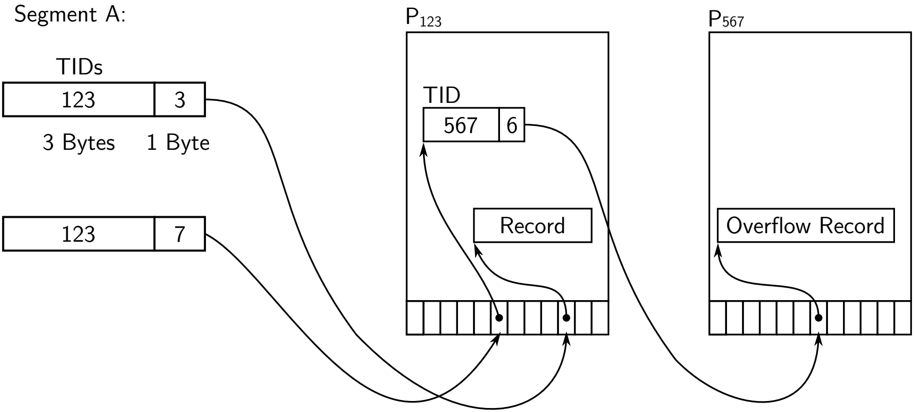
- often larger pieces are inserted soon after each other
- e.g. a set of tuples
- or one very large data item
- should be allocated close to each other

Allocation interface is usually

$$allocate(min, max)$$

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

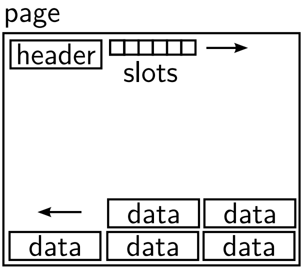
Slotted Pages



(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	for recovery
slotCount	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

Note: a slotted page can contain hundreds of entries!
Requires some care to get good performance.

Slotted Pages (4)

Slot:

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

Special cases:

- free slot: offset $= 0$, length $= 0$
- zero-length data item: offset > 0, 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)

One possible slot implementation:

T	S	O	O	O	L	L	L

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

The original TID is important for scanning.

Record Layout

The tuples have to be materialized somehow.

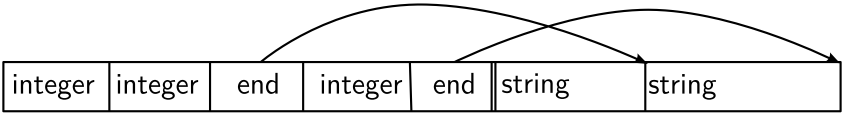
One possibility: serialize the attributes

integer	integer	length	string	integer	length	string

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 decreasing alignment
- place variable-length data at the end
- variable length has alignment 1

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 size reduces the 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

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

fixed	fixed	$\text{len}_1\text{len}_2\text{len}_3\text{len}_4$	comp_1	comp_2	comp_4

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

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

What about large tuples?

- sometimes the user wants to store something large
- e.g., embed a document
- SQL supports this via BLOB/CLOB

Requires some mechanism so handle these large records.

Long Records (2)

Simply spanning pages is not a good idea:

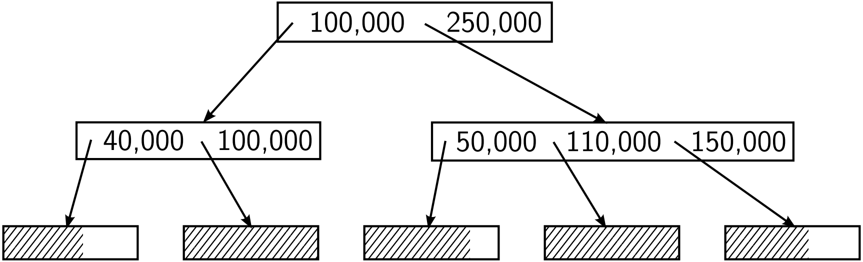
- must read an unbounded number of pages to access a tuple
- greatly complicates buffering
- a tuple might not even fit into main memory!
- updates that change the size are complicated
- intermediate results during query processing

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)

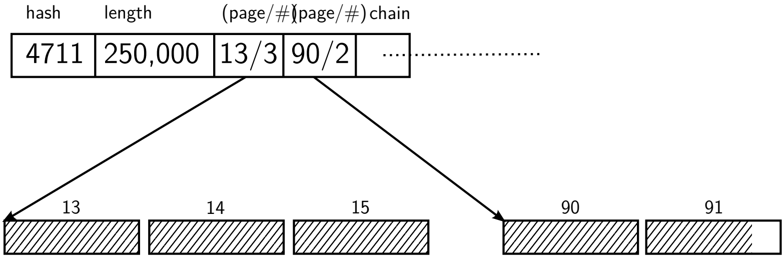
BLOBs can be stored in a B-Tree like fashion



- (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

Index Structures

Data is often indexed

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

Two important access classes:

- point queries
  find all tuples with a given value (might be a compound)
- range queries
  find all tuples within a given value range

Support for more complex predicates is rare.

B-Tree

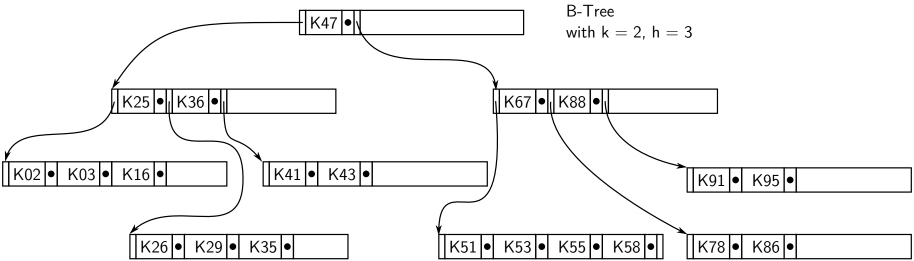
B-Trees (including variants) are the dominant data structure for external storage.

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

B-Tree (2)

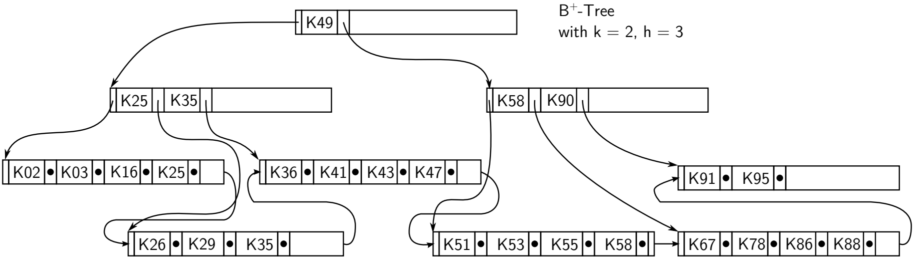
Example:



The • is the TID of the corresponding tuple.

B$^+$-Tree

Most DBMS use the B$^+$-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
...	...

Leaf Node:

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

Similar to slotted pages for variable keys.

Operations - Lookup

Lookup a search key within the B^+-tree:

1. start with the root node
2. is the current node a leaf?
   - if yes, return the current page (locate the entry on it)
3. find the first entry $\geq$ search key (binary search)
4. if no such entry is found, go the *upper*, otherwise go to the corresponding page
5. continue with 2

Lookup can return a concrete entry or just the position on the appropriate leaf page (depends on usage pattern).

Operations - Insert

Insert a new entry into the B^+-tree:

1. *lookup* the appropriate leaf page
2. is there free space on the leaf?
   - if yes, insert entry and stop
3. split the leaf into two, insert entry on proper side
4. insert maximum of left page as separator into parent
5. if the parent overflow, split parent and continue with 4
6. create a new root if needed

Operations - Delete

Remove an entry from the B^+-tree:

1. *lookup* the appropriate leaf page
2. remove the entry from the current page
3. is the current page at least half full?
   - if yes, stop
4. is the neighboring page more than half full?
   - if yes, balance both pages, update separator, and stop
5. merge neighboring page into current page
6. remove the separator from the parent, continue with 3

Most systems simplify the delete logic and accept under-full pages.

Operations - Range Scan

Read all entries within a (start,stop(range

1. *lookup* the *start* value
2. enumerate subsequent entries on the current page
3. use the *next* pointer to find the next page
4. stop once the *stop* value is reached

Very efficient, in particular if leaf nodes are consecutive on disk.

Indexing Multiple Attribute

Compound keys are compared lexicographically:

$$(a_1, a_2) < (b_1, b_2) \Leftrightarrow (a_1 < b_1) \vee (a_1 = b_1 \wedge a_2 < b_2)$$

Otherwise compound keys are quite similar to atomic keys.

- when all attributes are bound, difference is minor
- if only a prefix is bound, the suffix is specified as range

$$a_1 = 5 \Rightarrow (5, -\infty) \leq (a_1, a_2) \leq (5, \infty)$$

Indexing Non-Unique Values

Users can create indexes on any attributes

- not necessarily unique
- in fact might contain millions of duplicates
- main problem: index maintenance
- how to locate a tuple for update/delete?

Solution: only index unique values

- append TID to non-key attributes
- TID works as a tie-breaker
- increases space consumption a bit
- but guarantees $O(\log n)$ access

Concurrent Access

How to handle concurrent access?

- simple page locking/latching is not enough
- will protect against “simple” (single page) changes
- but pages depend upon each other (pointers)

The classical technique is *lock coupling*

- a thread latches both the page and its parent page
- i.e., latch the root, latch the first level, release the root, latch the second level etc.
- prevents conflicts, as pages can only be split when the parent is latched
- no deadlocks, as the latches are ordered

Concurrent Access (2)

But what about inserts?

- when a leaf is split, the separator is propagated up
- might go up to the root
- but we have only locked one parent

Lock coupling up is not an option (deadlocks)

One way around it: “safe” inner pages

- while going down, check if the inner page has enough space for one more entry
- if not, split it
- ensures that we never go up more than one step

Concurrent Access (3)

Alternative: restart

1. first try to insert using simple lock coupling
2. if we do not have to split the inner node everything is fine
3. otherwise release all latches
4. restart the operation, but now keep all latches up to the root
5. all operations can be executed safely now

- greatly reduces concurrency
- but should happen rarely
- simpler to implement, in particular for variable-length keys

B-link Trees

- lock coupling latches two nodes at a time
- seems cheap, but effectively it locks hundreds (all children of the parent node)
- it would be nicer to lock only one page

For pure lookups that is possible when adding *next* pointers to inner nodes:

1. latch a page, find the child page, release the page
2. latch the child page
3. might have been split in between, check neighboring pages

Requires some care when deleting.

Bulkloading

How to build an B-tree for a large amount of data?

- repeated inserts are inefficient
- a lot of random I/O
- pages are touched multiple times

Instead: *sort* the data before inserting

- now inserts become more efficient
- good locality

But we can do even better.

Bulkloading (2)

To construct an initial B^+-Tree:

1. sort the data
2. spool data into leaf pages
   - fill the pages completely
   - remember largest value (separator) in each page in a temp file
3. spool the separators into inner pages
   - fill the pages completely
   - remember largest value (separator) in each page in a temp file
4. repeat 3 until only one inner page remains (root)

Produces a compact, clustered B^+-tree.

Bulkloading (3)

Existing B^+-trees are a bit more problematic, but can still be updated in bulk

1. sort the data
2. merge the data into the existing tree
3. form pages, remember separators, etc.
4. start a new chunk once a page would contain only entries from the original tree
5. merge in the separators as above

Minimizes I/O, but destroys clustering. Usually a good compromise.

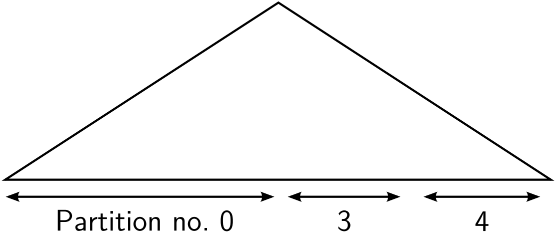
Partitioned B-Tree

Bulk operations are fine if they are rare, but they are disruptive

- usually the B-tree has to be take offline
- the new cannot be queries easily
- existing queries must be halted

Basic idea: *partition* the B-tree

- add an artificial column in front
- creates separate partitions with the B-tree



Partitioned B-Tree (2)

Benefits:

- partitions are largely independent of each other
- one can append to the "rightmost" partition without disrupting the rest
- the index stays always online
- partitions can be merged lazily
- merge only when beneficial

Drawbacks:

- no "global" order any more
- lookups have to access all partitions
- deletion is non-trivial ("anti-matter")

Variable Length Records

So far B-trees are defined for fixed-length keys

- all nodes have between k and $2k$ entries
- simplifies life considerably
- e.g., we "know" if an inner node is full

But in reality, entries can be variable length

- strings
- variable-length encoding, NULL
- compounds of these

Usually keys are *opaque*. The B-tree does not understand the structure. (One could special-case single strings).

Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

a	bbbbbbbb	c	dddddddd	e	ffffffff	g	hhhhhhhh
i	jjjjjjjj	k	llllllll	m	nnnnnnnn	o	

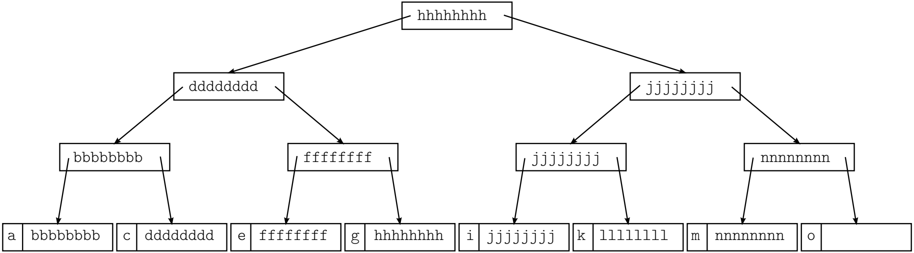
All entries have either length 1 or length 8.

Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

a	bbbbbbbb	c	dddddddd	e	ffffffff	g	hhhhhhhh
i	jjjjjjjj	k	llllllll	m	nnnnnnnn	o	

Variant A:



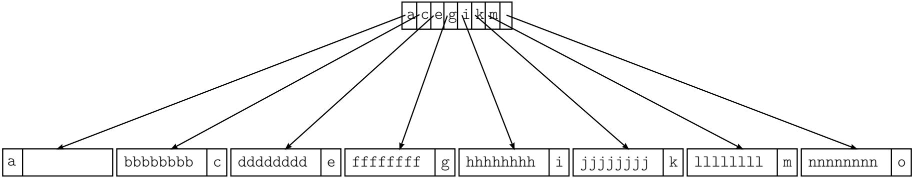
Height 4

Variable Length Records (2)

Variable length keys are problematic. Consider the following example:

a	bbbbbbbb	c	dddddddd	e	ffffffff	g	hhhhhhhh
i	jjjjjjjj	k	llllllll	m	nnnnnnnn	o	

Variant B:



Height 2 (ignores space consumption for pointers)

Variable Length Records (3)

Separator choice is crucial!

- affects fanout and space consumption
- all standard guarantees are off if normal algorithms are used for variable-length keys

Non-trivial issue

- greedy algorithms exist for the bulkloading case
- idea: recursively pick the smallest value as separator such that the resulting group sizes vary within a constant factor
- can also be extended to the dynamic case (rebuild as need), but non-trivial
- difficult, but gives good amortized bounds

Variable Length Records (4)

Minimal support: modify the split logic

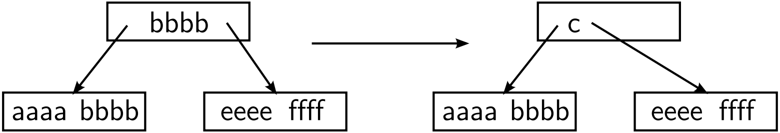
1. when a page overflows, build the sorted list of all values
2. instead of picking the median value as separator, pick the smallest value within 20% around the median
3. for ties, prefer values closer to the median

- pragmatic solution, but not optimal
- can still degenerate
- avoids the worst mistakes

Prefix B^+-tree

A B^+-tree can contain separators that do not occur in the data

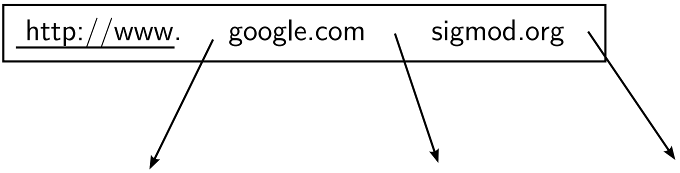
We can use this to save space:



- choose the smallest possible separator.
- no change to the lookup logic required

Prefix B$^+$-tree (2)

We can do even better by factoring out a common prefix:



- only one prefix per page
- the change to the lookup logic is minor
- the lookup key itself is adjusted
- sometimes only inner nodes, to keep scans cheap

Prefix B^+-tree (3)

The lexicographic sort order makes prefix compression attractive:

- neighboring entries tend to differ only at the end
- a common prefix occurs very frequently
- not only for strings, also for compound keys etc.
- in particular important if partitioned B-trees
- with big-endian ordering any value might get compressed

Hash-Based Indexes

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

- compute a hash-value h, compute a slot (e.g., $s = h \mod |T|$, 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

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
- range queries are very problematic
- order preserving hashing exists, but is questionable
- quality of the hash function is critical

As a consequence, mainly useful for primary key indexes

- unique keys
- key collisions would be very dangerous
- how to delete a tuple with an indexes attribute of there are 1 million other tuples with the same value?
- can be fixed by separate indexing within duplicate values (complicated)

Extendible Hashing

A central problem of hashing schemes on disk are the table size

- hard to known beforehand
- too small ⇒ too many collisions
- chaining is expensive on disk
- too large ⇒ waste of space
- would have to grow over time

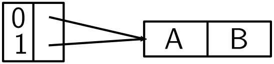
Traditional solution for main memory: *rehashing*

- re-map items to hash table sizes
- involves touching every item
- poor locality, a lot of random I/O
- prohibitive for disk

Idea: Allow for growing the hash table without rehashing by *sharing* table entries.

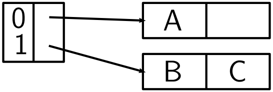
Extendible Hashing (2)

- hash table size is always a power of 2
- hash table points to buckets (pages)
- multiple table entries can point to the same bucket
- but always systematically (buddy systems)
- thus, the “depth” of the buckets varies



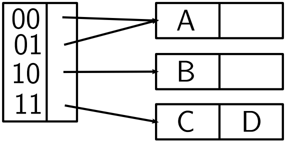
Extendible Hashing (3)

- when a bucket overflows, it is split
- if not a maximum depth, the depth is increased
- achieved by de-sharing slot entries
- one more bit becomes relevant
- items are distributed according to hash values



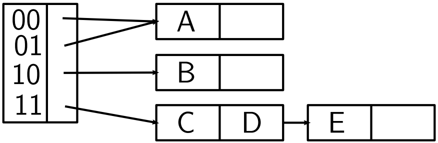
Extendible Hashing (4)

- if the depth cannot be increased, the table is doubled
- other buckets are unaffected
- entries are duplicated, resulting in new sharing
- new buckets are linked as usual



Extendible Hashing (5)

- once a maximum table size is reached start chaining
- ideally occurs rarely, traversing chains is expensive
- optionally one can balance chaining vs. table growth via load factor



Extendible Hashing (6)

Advantages:

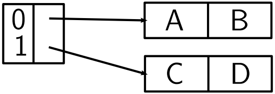
- ideally exactly two page accesses per lookup
- less than in a B-tree
- table can grow independent of existing buckets
- no need for re-hashing

Disadvantages:

- table growth is a very invasive operation
- large steps in space consumption
- what about hash collisions?
- same care is needed to avoid extreme table growth

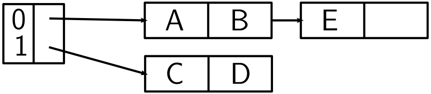
Linear Hashing

- linear hashing avoids the exponential directory growth
- it starts with a regular hash table with buckets



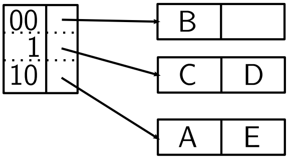
Linear Hashing (2)

- when a bucket overflows it uses chaining
- degrades performance, but ok if the chain is short



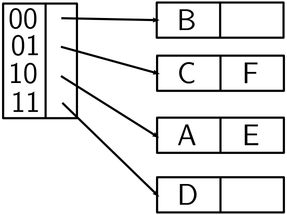
Linear Hashing (3)

- triggered by load factor or chain length a bucket is split
- chains are re-integrated into the bucket
- only one (i.e., the next) bucket is split, the rest remains untouched
- the range of buckets $[1, k[$ has been split, the range $[k, n]$ is unsplit (the range $[n, n + 2k - 2[$ contains the second halves)
- the directory grows page by page



Linear Hashing (3)

- more buckets are split on demand
- at some point all buckets have been split once
- then the cycle starts anew



Linear Hashing (4)

Advantages:

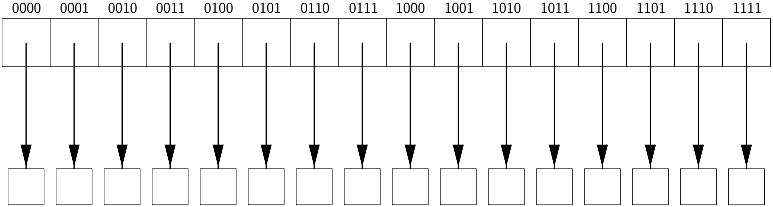
- avoids the disruptive directory growth of EH
- index grows linearly
- amortized the index structure is nice

Disadvantages:

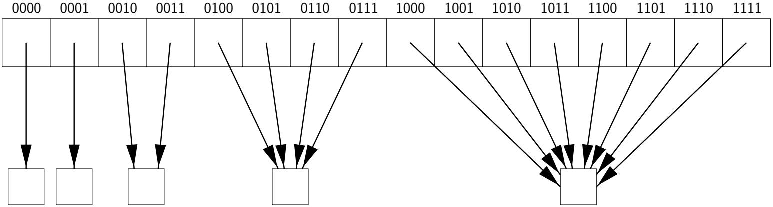
- chaining hurts performance
- it can take a while until chains are re-integrated
- page allocation for the directory problematic

Multi-Level Extendible Hashing

For uniform distributions the EH directory is nice:



But data skew causes poor space utilization

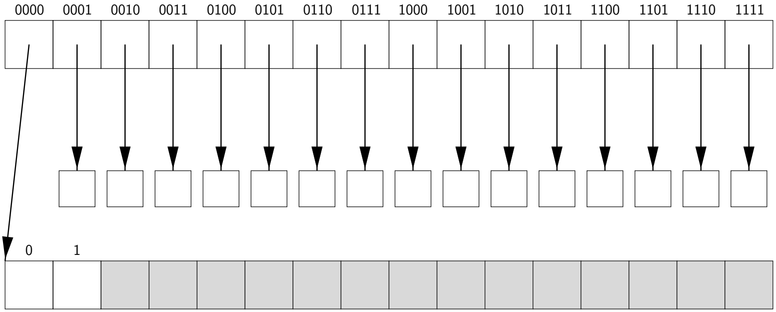


The directory size explodes.

- skew is an unfortunate reality

Multi-Level Extendible Hashing (2)

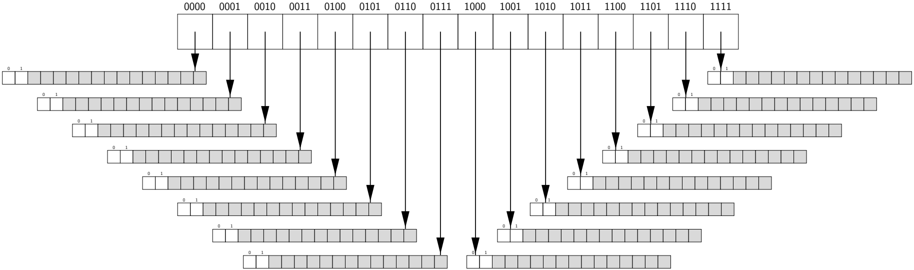
Basic idea: construct a tree of hash tables



- the next level uses the next k bits
- node size is page size
- additional page faults, but fanout is very large
- only the heavily used parts get additional levels

Multi-Level Extendible Hashing (2)

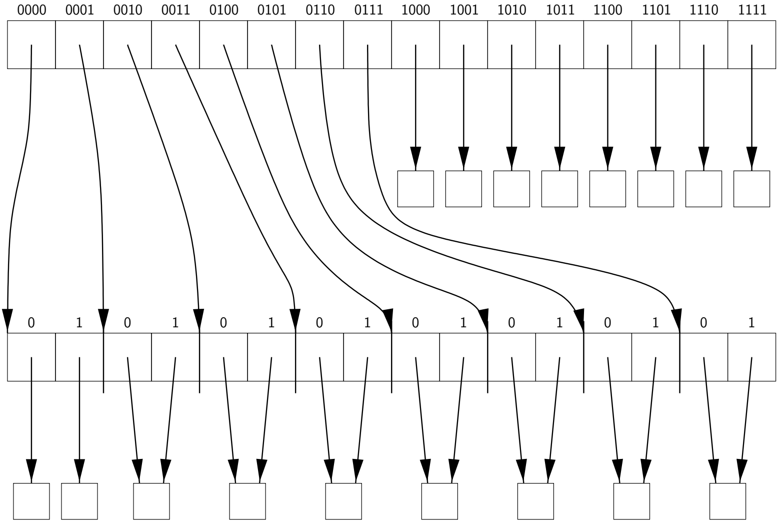
Problem: space utilization



- now uniform is the worst case
- all buckets will overflow at the same time
- second-level hash tables will be nearly empty
- leads to poor space utilization (and poor performance)

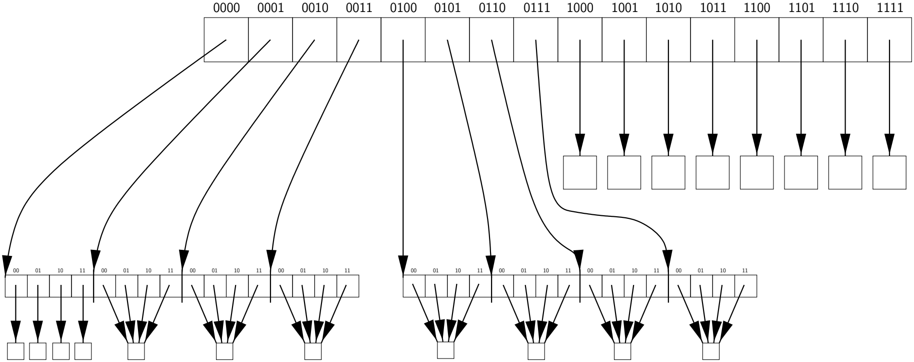
Multi-Level Extendible Hashing (3)

Instead: share inner page between buddies



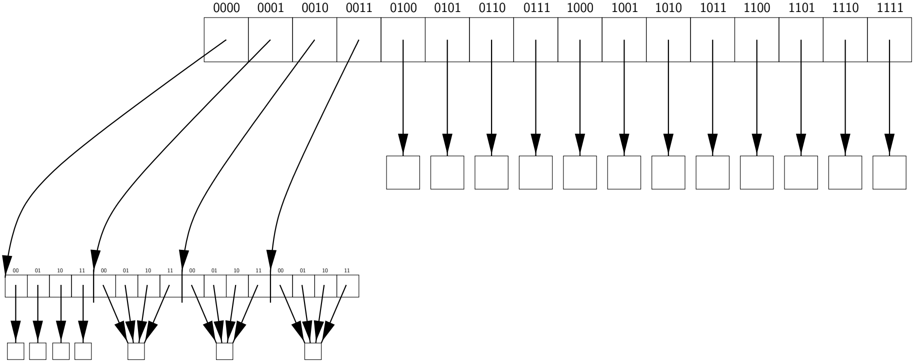
Multi-Level Extendible Hashing (4)

We can get additional bits by splitting the inner page



Multi-Level Extendible Hashing (5)

In fact here we can even move buddies up again



Multi-Level Extendible Hashing (6)

- uses a buddy system (boundaries are powers of 2)
- bit width etc. derived implicitly
- pointer structure contains enough information
- results in large fanout

It has some nice properties

- naturally adapts to data skew
- directory growth is not a problem
- but additional page accesses
- tree is very shallow, however

Bitmap Indexes

Classical indexes do not handle unselective predicates very well

- large fraction of tuples is returned
- index access is more expensive than a table scan
- but a combination of predicates might be selective
- index intersection would help, but is still expensive
- predicates might also contain disjunctions etc.

Expample: $\sigma_{(a=2 \vee b=5) \wedge (c \neq 1)}(R)$

Could be answered by B-trees, but often not very efficient.

Bitmap Indexes (2)

When the attribute domain is small, it can be indexed using **bitmap indexes**

	a=2
tid_1	1
tid_2	0
tid_3	0
tid_4	1
...	...

	a=3
tid_1	0
tid_2	1
tid_3	0
tid_4	0
...	...

...

- one bitmap for every attribute value
- index intersections become bit operations
- very efficient
- remaining ones indicate matching tuples

Bitmap Indexes (3)

- bitmap indexes are compact (one bit per tuple)
- (plus tid directory if needed)
- and are usually sparse
- can be compressed very well
- run-length encoding in particular attractive
- intersection can be performed on the compressed form

Often outperforms other indexes for unselective attributes.

Small Materialized Aggregates

- data is usually stored in physical chunks
- pages, or chunks of pages
- clustering usually insertion order
- older chunks tend to remain static
- we can cheaply pre-computed (i.e., cache) some info

Some useful aggregates:

min	max	sum	count
10	20	1500	1000

Small Materialized Aggregates (2)

Before scanning a chunk, we examine the aggregate

- some predicates can be evaluated on the pure aggregate
- only for the current chunk, of course
- in particular skipping chunks is often possible
- aggregates can be directly re-used
- greatly saves I/O

What about updates?

- small materialized aggregates are like a cache
- can be updated eagerly or lazily
- an invalidation flag is enough

Multi-Dimensional Indexing

- huge field
- R-tree, grid file, pyramid schema, index intersection, ...
- hundreds of approaches
- we do not discuss them here

But: remember the **curse of dimensionality**

- we can only index a relative low number of dimensions
- for higher dimensions, range queries/proximity queries fail
- scan becomes faster than index structures