Data Management Systems

- Access Methods
  - Pages and Blocks
  - Indexing
  - Access Methods in context

Hashing
B+ trees
Other indexing techniques

Gustavo Alonso
Institute of Computing Platforms
Department of Computer Science
ETH Zürich

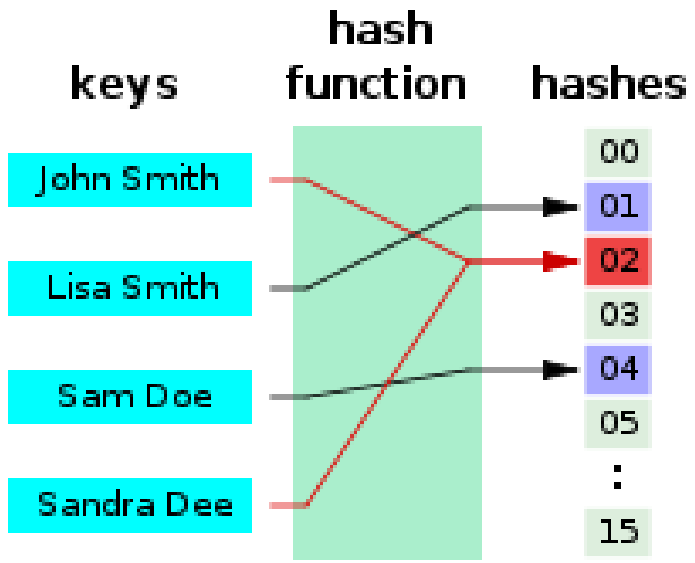
Systems@ETH Zürich

Hashing

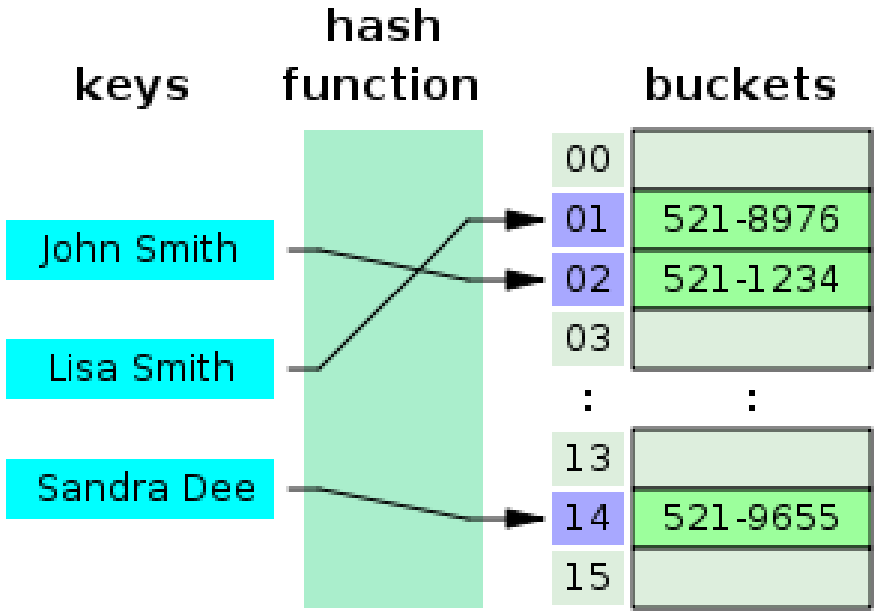
Hashing in databases

- Hashing is a very common operation in many systems and also in databases
  - Many internal data structures are implemented using a hash table (buffer cache, lock table)
  - Some operators use hashing to speed things up (hash joins)
  - Also used as an index and a partition strategy
- Hashing is the typical trade-off storage vs compute:
  - A B+ tree sacrifices space to speed up the search
  - Hashing uses compute (the hash function) to find out the slot where something is located

From hashing to hash tables



https://en.wikipedia.org/wiki/Hash_function



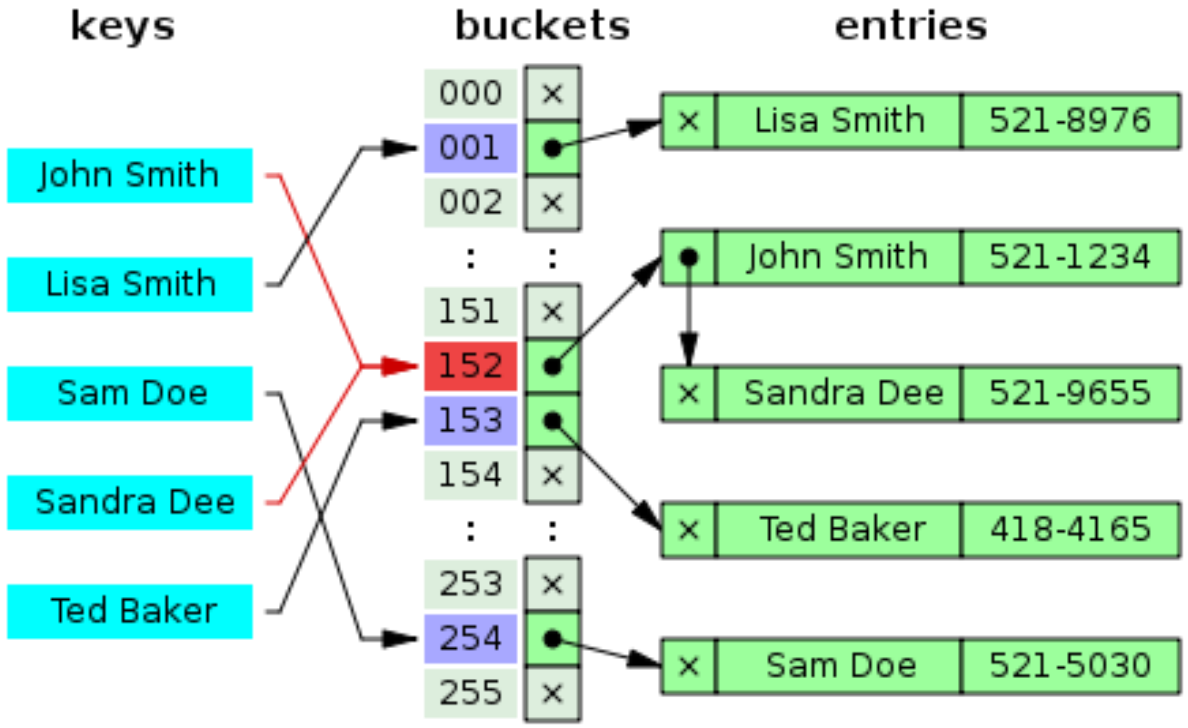
https://en.wikipedia.org/wiki/Hash_table

Limitations of hashing

- There are many hashing functions with strong properties. In a database, however, the hash function has to be computationally cheap since it is used very often
- Perfect hash functions exist but you need a hash table as big as the cardinality of the attribute (4 byte keys = 4 GB table)
- The hash table has to be big enough without wasting space:
  - If too small, too many collisions
  - If too big, lots of wasted space and occupying many blocks
  - Growing the hash table not a cheap operation
- Hash indexes only support point queries (works for the primary key, does not work for almost anything else)

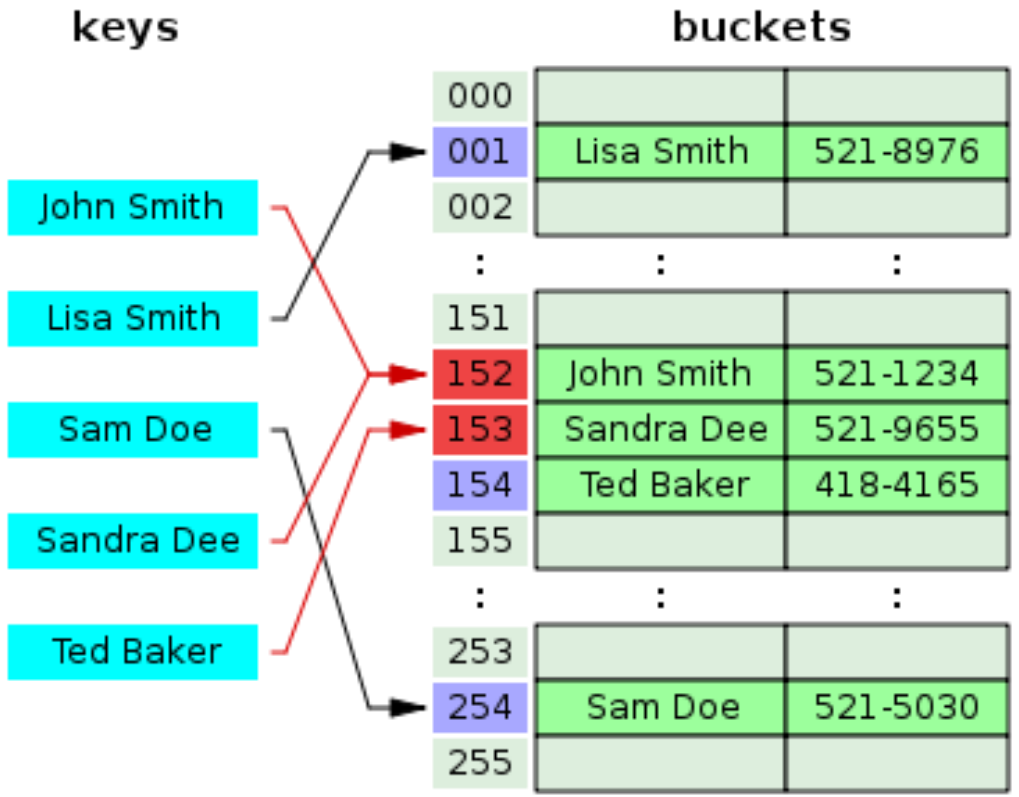
Hash table collisions - chaining

- Chaining
- When a collision occurs, add another entry in a linked list
- If lists are short, reasonably efficient
- Can already reserve space for the linked lists (more blocks but linked list traversal within the same block)



Hash table collisions – open addressing

- Open addressing
- A general strategy whereby, when a collision occurs, we look for an empty slot in the hash table using some rule
- Linear probing = just go to the next slot(s)
- Cuckoo hashing = use several hash functions, if collisions with first, use the second, if …

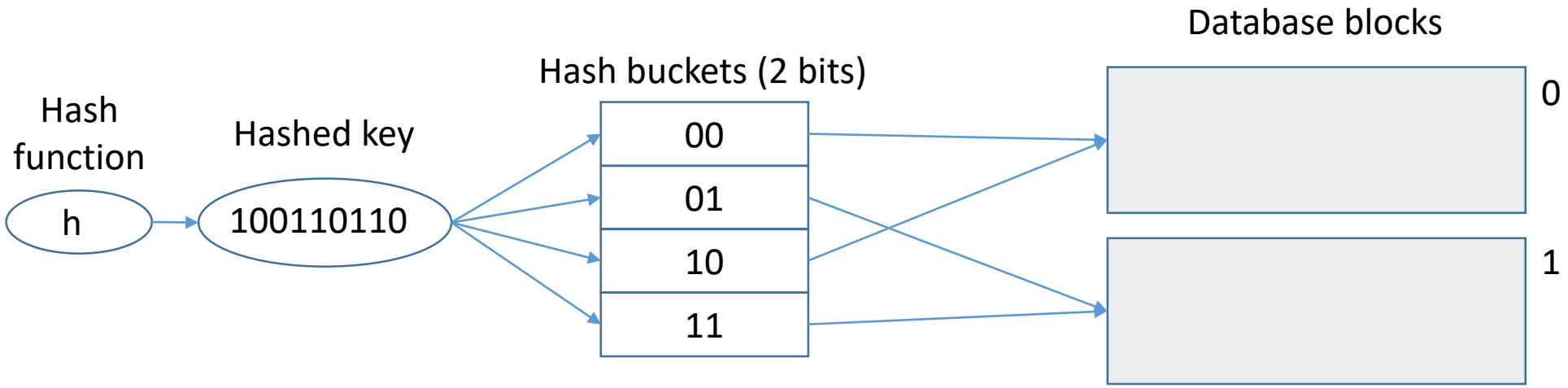


Growing pains

- When the hash table is full, growing it is not easy
- Basic approach:
  - Create a new, larger hash table with more buckets (typically 2x)
  - Rehash all existing items
- This is too expensive
  - Need to rehash every item
  - Lots of random accesses
  - Can simply not be done on disk (where hash indexes are mostly used)

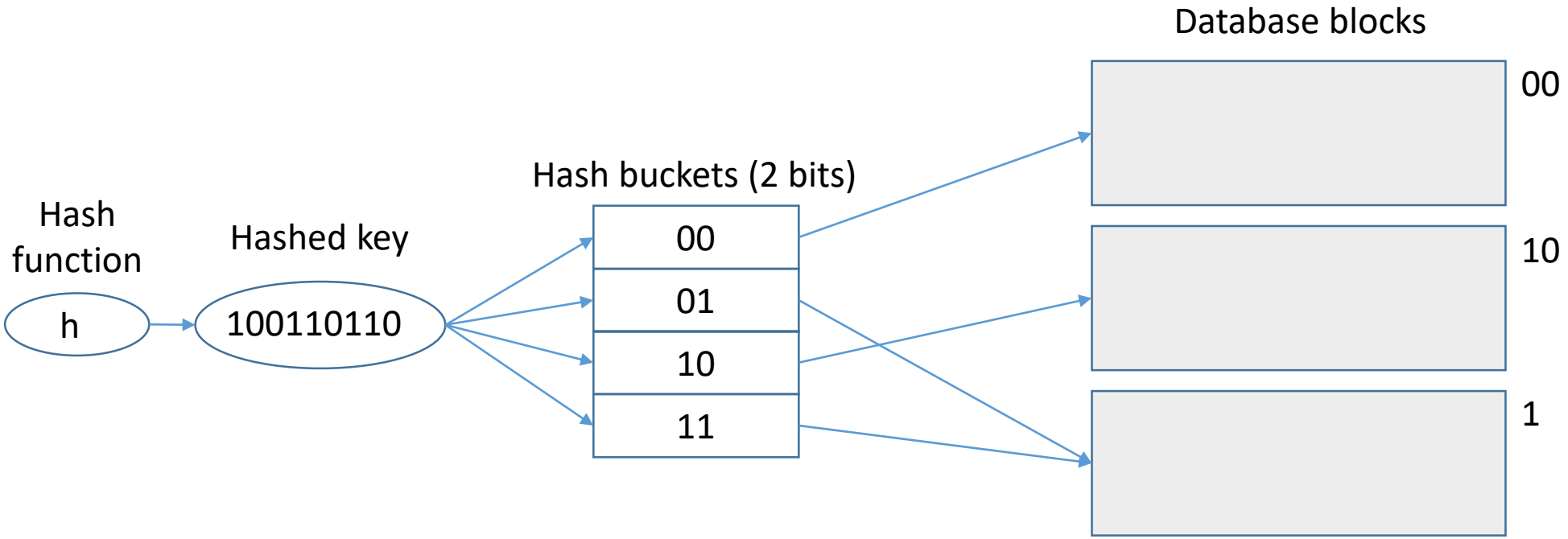
Extensible hashing

- Based on sharing buckets and un-sharing when needed
- Hash table buckets are mapped into blocks
- Initially, several buckets share the same block



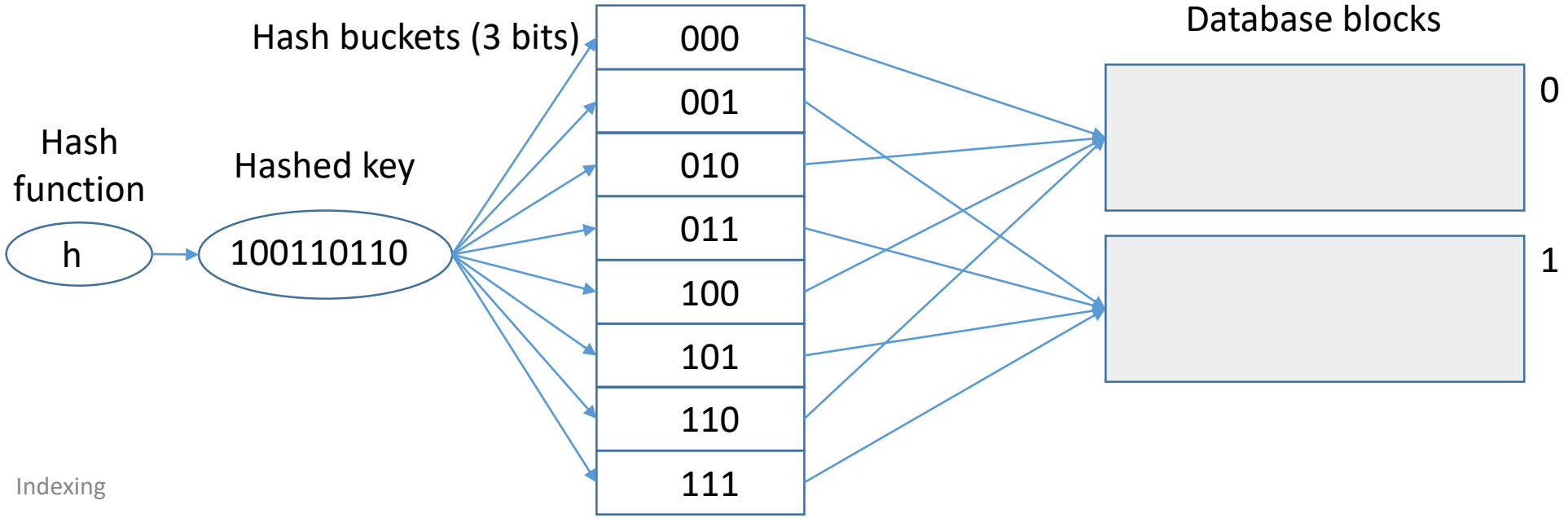
Extensible hashing overflow

- If a bucket gets full, then split the bucket
- Move entries as needed to new bucket



Extensible hashing, logical doubling

- The size of the table can be doubled without immediately adding more space, allowing for more splitting
- Simply increase the number of buckets and the degree of sharing
- Split buckets as they become full

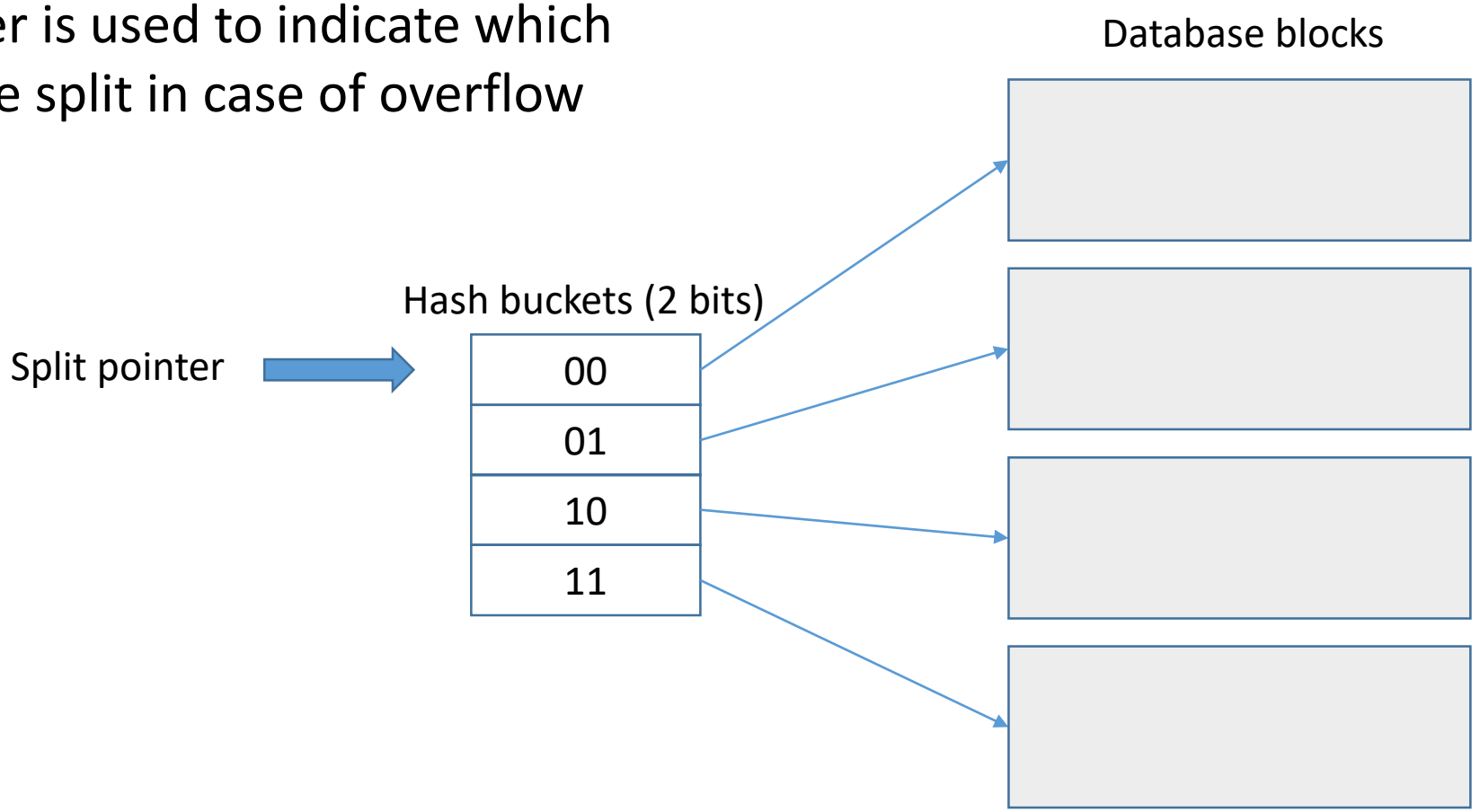


Extensible hashing

- If initial space available, allows for growth without disruption
- Two pages lookups to access an item (ideally) = bucket directory and data block
- Bucket directory can grow independently of the data blocks
- But the doubling of the bucket directory is expensive, creating many unneeded entries

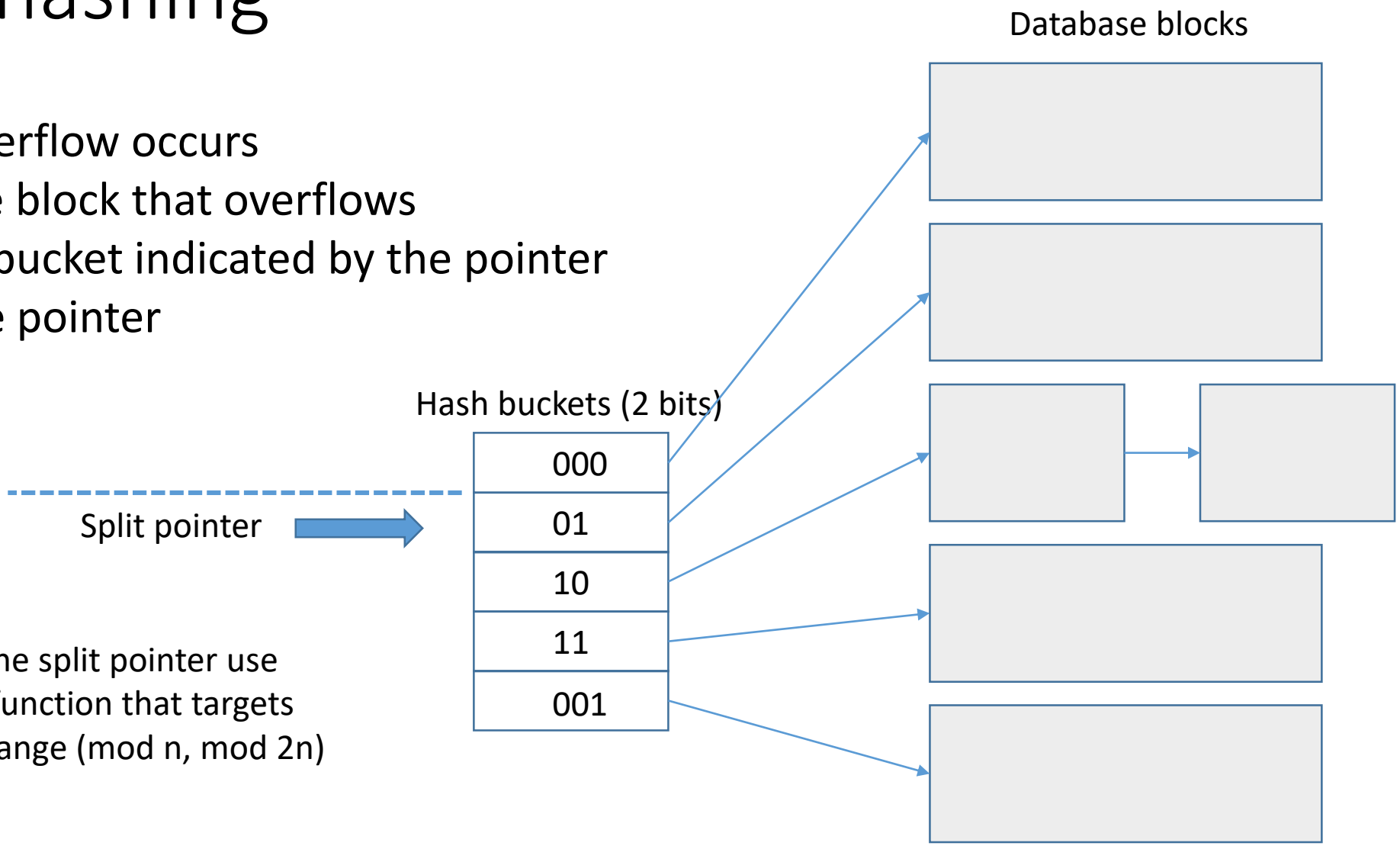
Linear hashing

A split pointer is used to indicate which bucket will be split in case of overflow



Linear hashing

When overflow occurs
Chain the block that overflows
Split the bucket indicated by the pointer
Move the pointer



Entries above the split pointer use
a second hash function that targets
the expanded range (mod n, mod 2n)

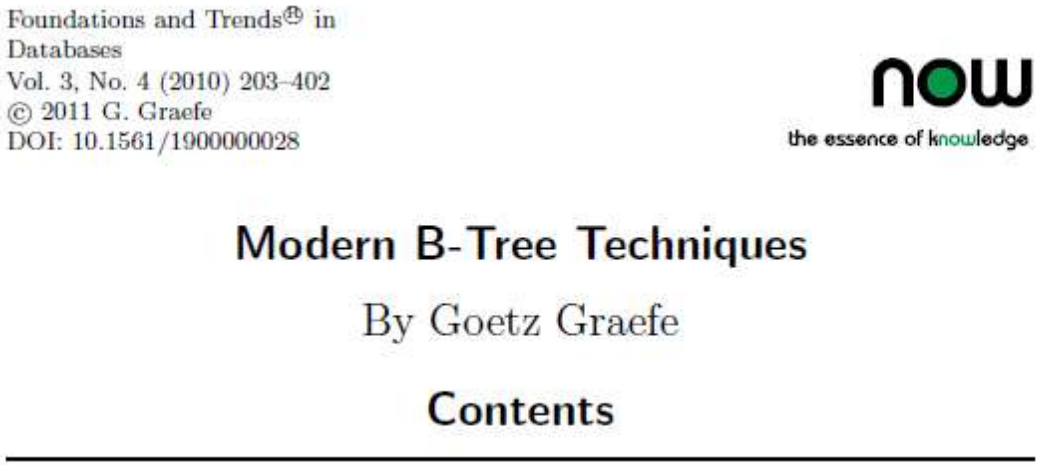
Linear hashing

- The idea is to gradually increase the size of the table and redistribute the data
  - Always split on the pointer even if overflow is somewhere else
  - The pointer will eventually reach buckets with a chain, when that bucket is split, the data will be reorganized
- Splits can be triggered by overflows, a load factor, a maximum chain length, etc.
- The advantage is that the directory (list of buckets), grows page by page (instead of doubling)
- Once all buckets have been split, start anew

Many variations

- These ideas can be combined in different ways (especially with chaining)
- Approach can be nested:
  - Bucket directory points not to data blocks but to another hash table that points to the actual data blocks
  - Helps to deal with skew

B+ trees


Foundations and Trends® in
Databases
Vol. 3, No. 4 (2010) 203–402
© 2011 G. Graefe
DOI: 10.1561/1900000028

now
the essence of knowledge

Modern B-Tree Techniques

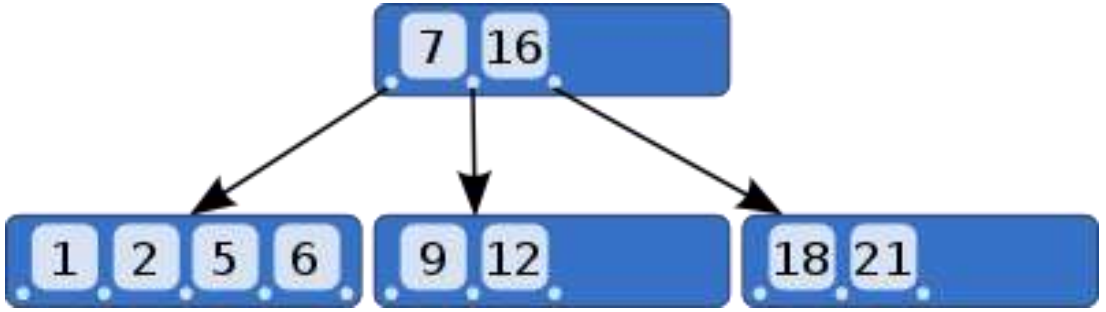
By Goetz Graefe

Contents

https://w6113.github.io/files/papers/btreesurvey-graefe.pdf

B-Tree

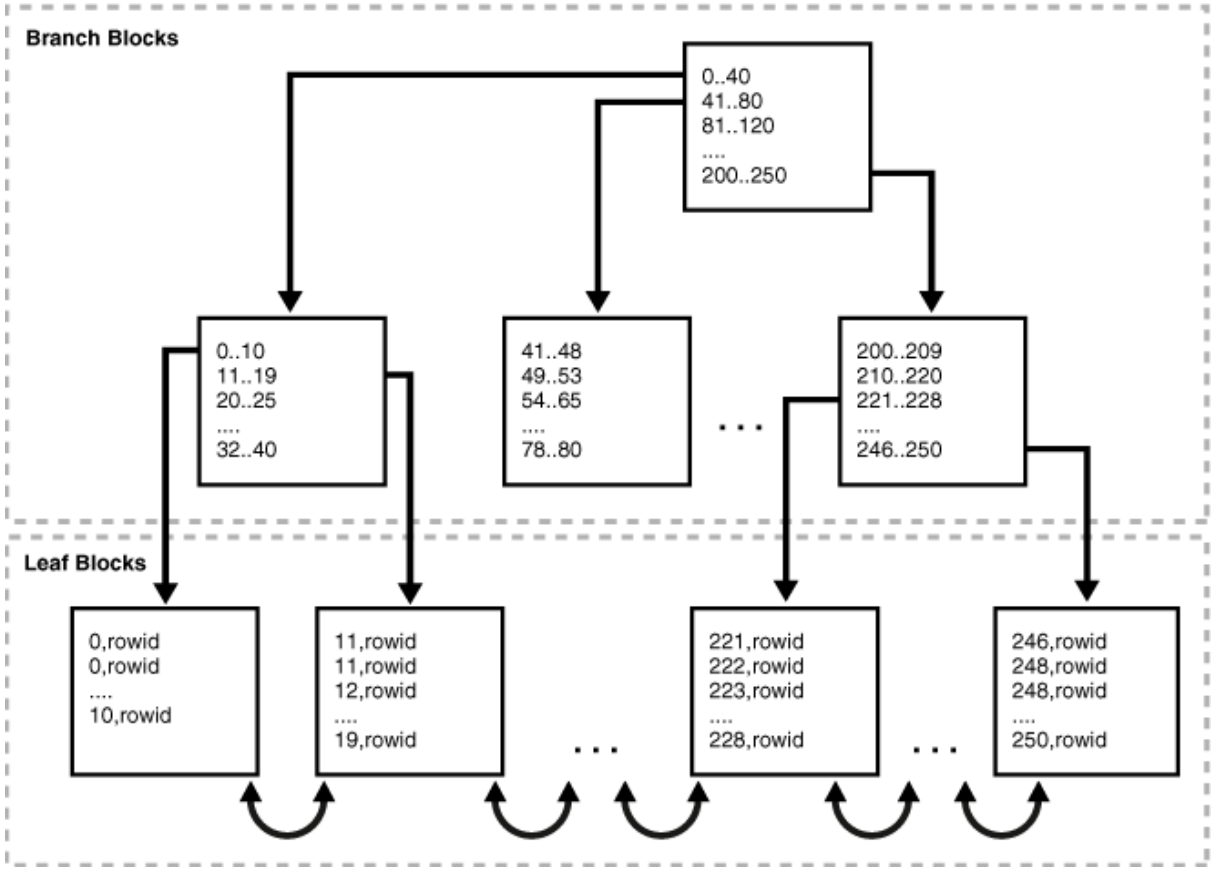
- B-tree
  - Has order k
  - Each node (except the root) has between k/2 and k child nodes
  - The root has at least two children (unless it is also a leaf)
  - A non-leaf node contains k-1 keys
- Databases do not use B-trees but B+ trees; even if they say they use B-trees!!



https://en.wikipedia.org/wiki/B-tree#/media/File:B-tree.svg

B+ tree

- B+ tree
  - Is a B-tree
  - But the data is at the leaves only
  - Leaf nodes organized as linked list
- B+ trees are balanced
- Inner nodes correspond to blocks
- Leaf nodes correspond to blocks
- Blocks organized like slotted pages for variable length data



From Oracle documentation (11g)

B+ tree details

- The keys in the inner nodes might not correspond to actual data
  - Used as separators
- Typically, the leaf nodes contain pointers to the tuples: <value,key> where “value” is the value that is being indexed and “key” is the pointer to the tuple, typically as a row id or tuple id (recall: this is at least a block id and an offset)
- Some systems allow to store the data directly on the leaves (default is as above)
- Some systems create a B+ tree index by default for all tables, indexing the key. If there is no key, the engine assigns random keys and indexes them

Clustered indexes

- An index orders the table by the attribute it is indexing
  - But the tuples in the table might not be ordered
- A clustered index forces the tuples to be stored in the same order as the index indicates => table is physically stored in a sorted manner
  - Typically done only for the primary key
  - Automatic in systems that stored the data in the leaf nodes
- Most useful for tables that are not updated frequently

What to index?

- A B+ tree can use one or more attributes as the key to the index
  - If one attribute, those are the values
  - If several, it builds a composite key
- Useful when we are looking for combinations of values on certain attributes or a table is searched by several attributes
- Compared lexicographically

  $(a1,b1) < (a2,b2) \iff (a1 < b1) \vee (a1 = b1 \wedge a2 < b2)$
- Some values can be left unspecified but typically only the ones at the beginning of the key

Composite index

```
SELECT department_id, last_name, salary
FROM   employees
WHERE  salary > 5000
ORDER BY department_id, last_name;
```

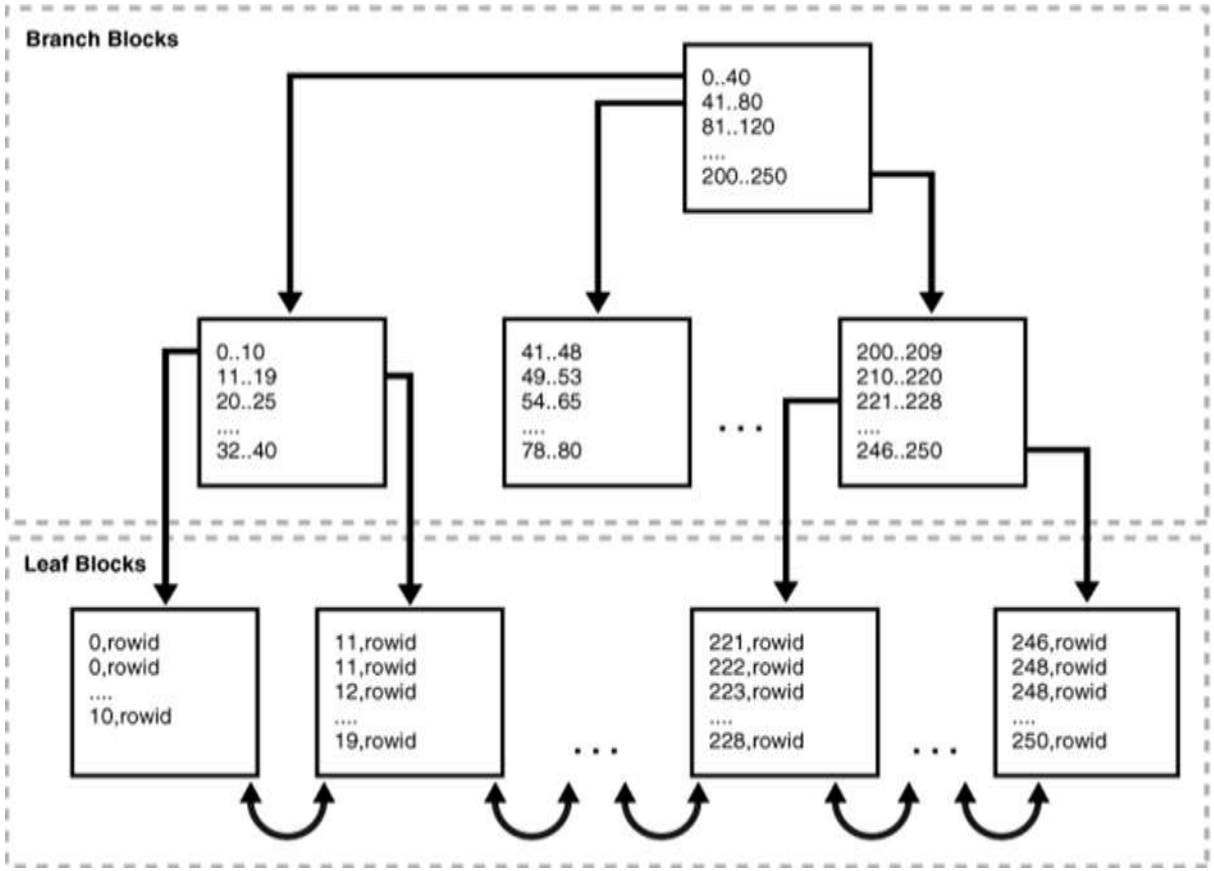
- Assume a composite index on department_id, last_name, salary
- Data is sorted by the three attributes (in that order)
- Full index scan:
  - Read the data from the leaves in sorted order
  - Filter on salary
- Avoids having to scan the entire table and result is already sorted

Non-unique values

- A B+ tree index can be built on any attribute, including those that are not unique (e.g., the department in the table of students of ETH)
- This is a problem with the basic design as we cannot find all the duplicates
  - Option 1 = repeat the key at the leaf nodes for every duplicated entry
  - Option 2 = store the key once but point to a linked list of all the matching entries
- If the data is stored in the leaves, append the tuple ID to know what tuple the entry refers too (otherwise, they are all the same)

B+ tree direct lookup

- Traverse the tree from the root
- Within each node, use binary search to look for the correct entry
- At a leaf node:
  - Return the corresponding pointer
  - Return the position

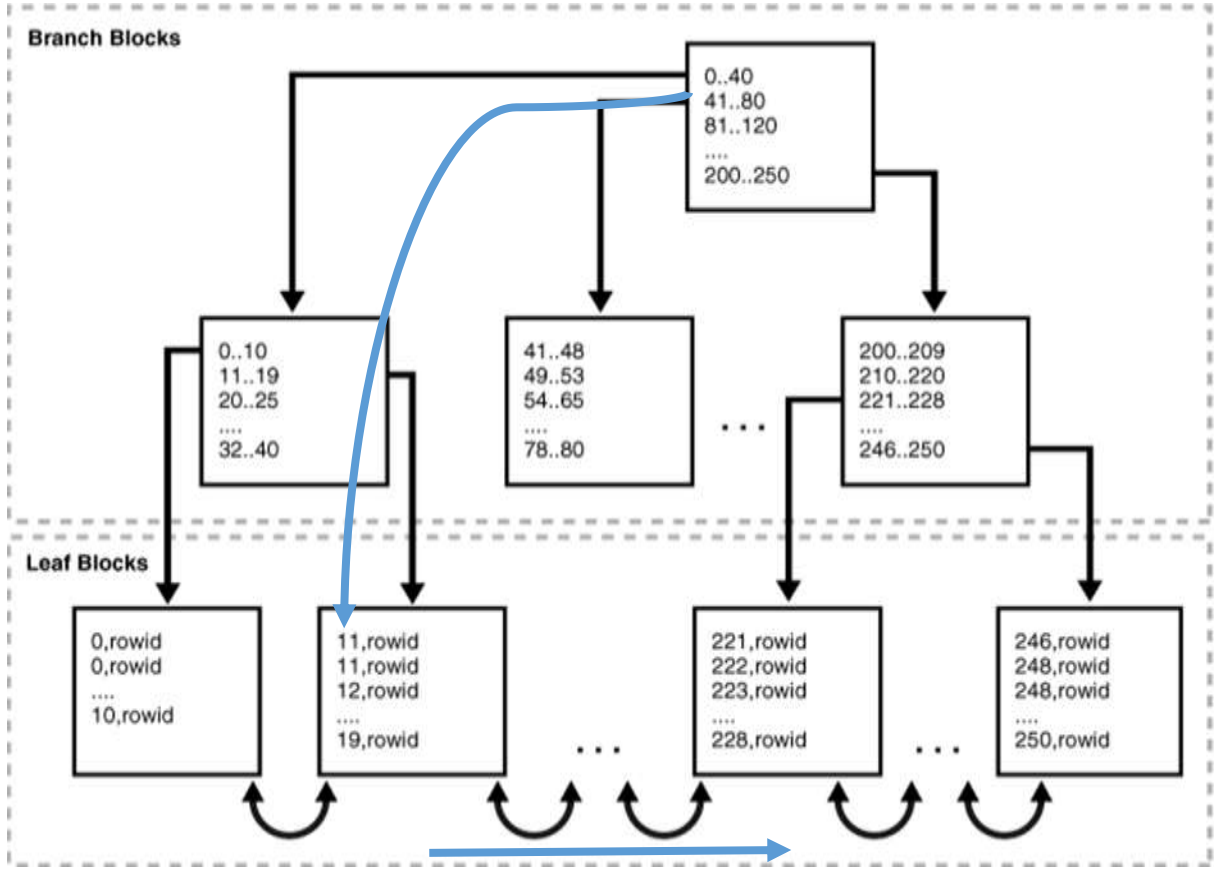


Scans: Range lookup

SELECT *
FROM T
WHERE T.x > 11 AND t.x < 222

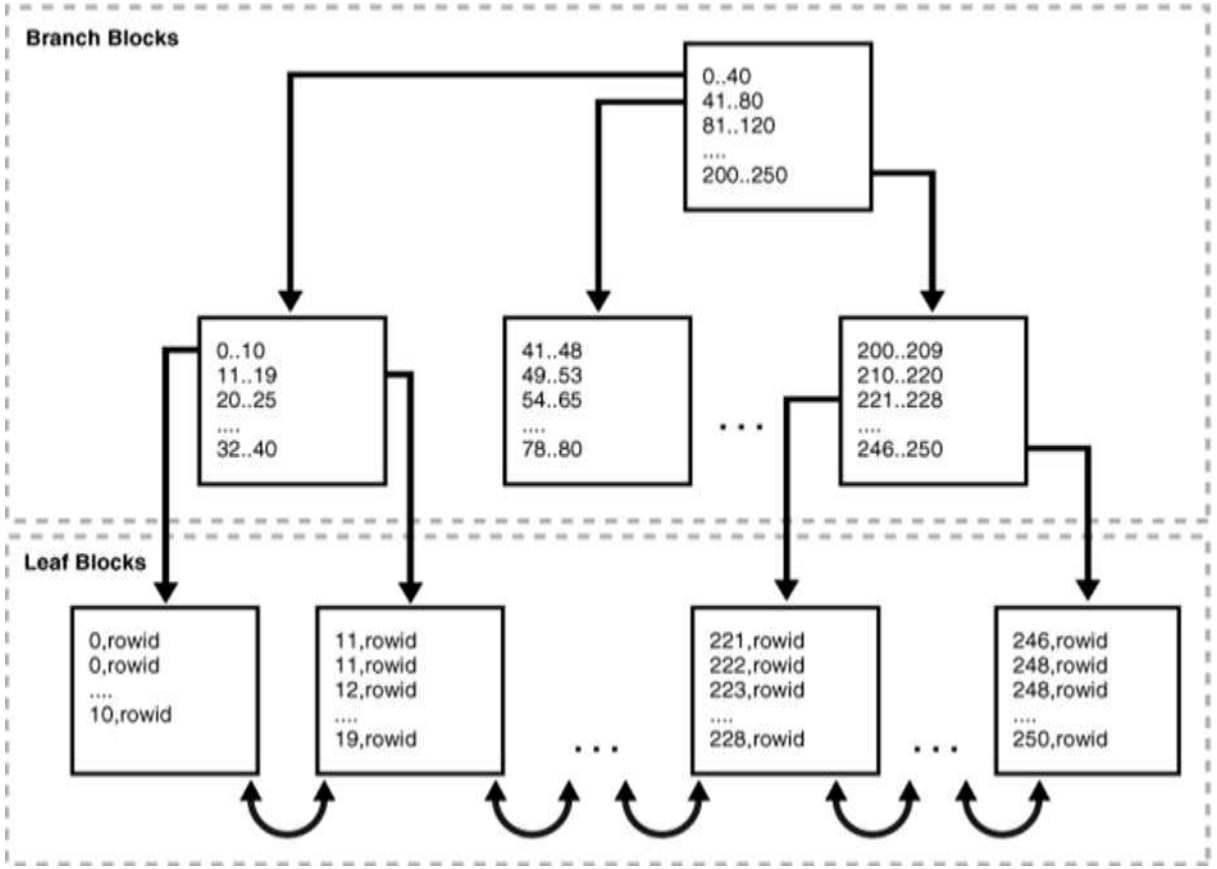
Find first tuple that matches

Traverse leaves until last match found



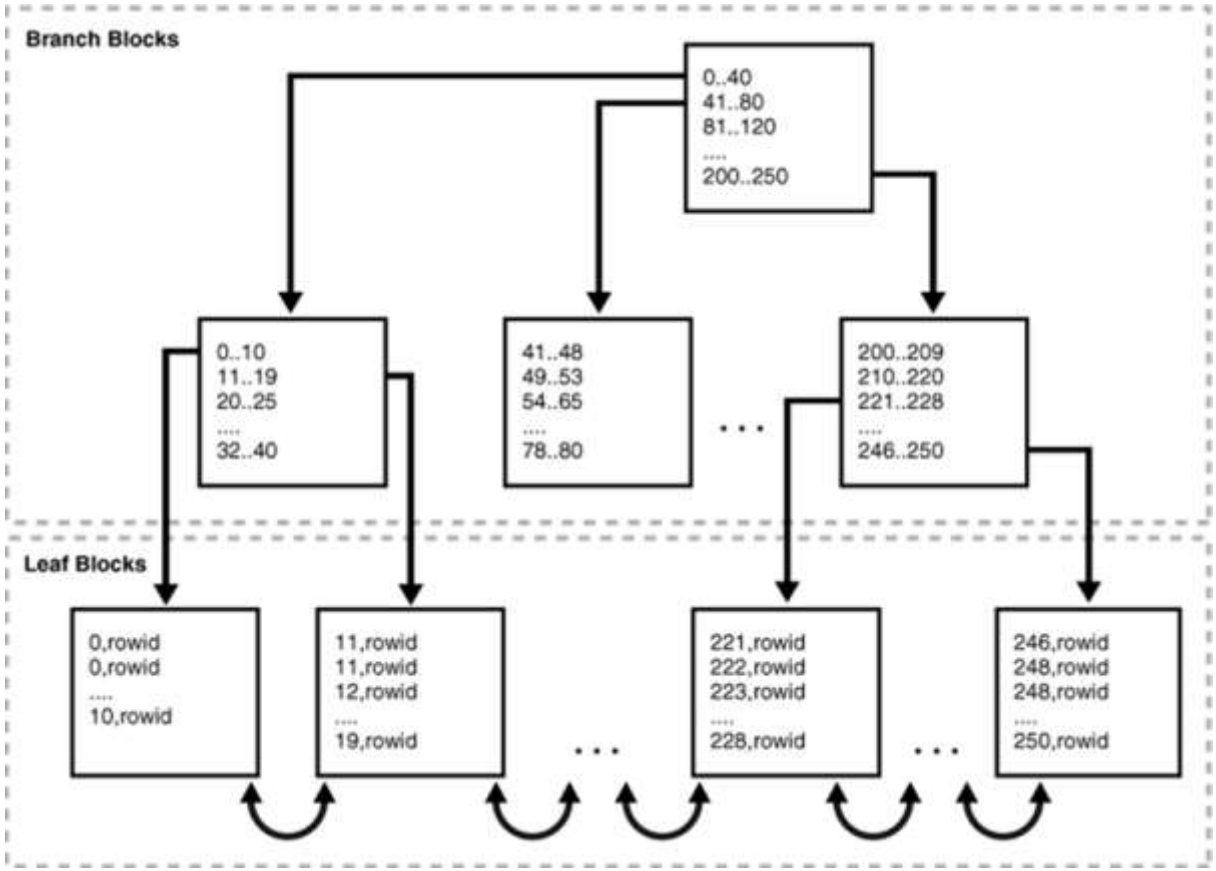
Inserting into the index

- Lookup the corresponding leaf
  - If there is space, insert
- If there is no space in the leaf
  - Split leaf into two new leaves
  - Insert new item on corresponding leaf
  - Insert new separator on parent node
- If parent node is full
  - Split node in two
  - Insert separator in parent node
  - All the way to the root if needed



Deleting from the index

- Look up the corresponding leaf
- Remove the entry from the leaf
- If the leaf is less than half full
  - Check a neighboring leaf
  - if more than half full, balance both leaves
  - If half full, merge both leaves
  - Update separator



Concurrent access I

- Indexes are heavily used while they are being maintained, these creates conflicting, concurrent accesses
- Lock coupling
  - Lock a page and its parent:
    - Look the root, lock the first level
    - Release the lock on the root, lock the second level
    - …
  - Prevents problems when a page must be split
  - Not enough if we have to go further up

Concurrent access II

- Make sure the bad case never happens:
  - Use lock coupling
  - On every node, check if there is space for one more entry
  - If not, then split the node (we can, because we checked the parent)
- This approach ensures that a split of a page never causes the changes to propagate all the way back to the root
- An alternative approach
  - Try lock coupling
  - If we have to split beyond the parent, abort operation and release everything
  - Start again but now locking the entire path from the root

Bulk inserts

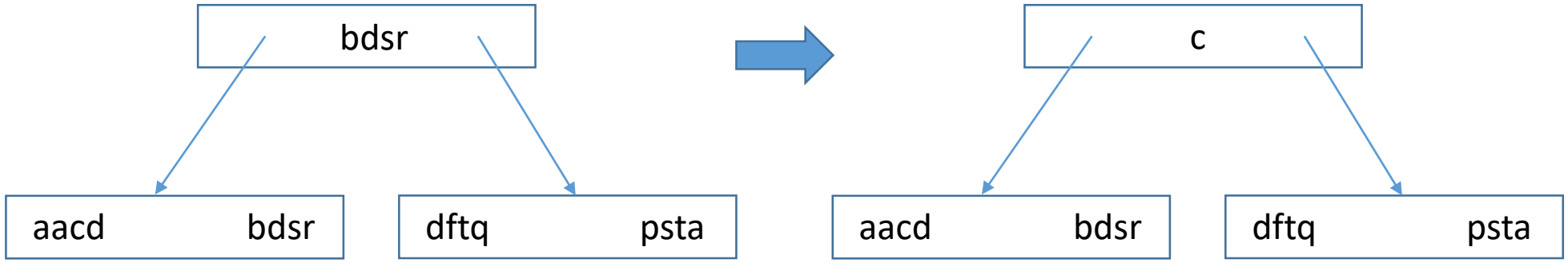
- To create a B+ tree index, proceed as follows:
  - Tree will be created bottom up (from the leaves up)
  - Sort the data
  - Use the sorted data to fill blocks one after each other
  - Remember the largest value in each block
  - Create the inner nodes by using the largest value in each block as separator
  - Iterate upwards to next level until there is only one block (the root)
- If blocks are filled completely, the results is a clustered and compact tree
- If data is to be updated, better leave space in each block

Optimizations

- Reverse index
- If the attribute being indexed is a sequence and new items are constantly being produced, inserting into a B+ tree is a problem:
  - Values next to each other go to the same block
  - Concurrent updates fight to insert on the same block
  - Depending on how updates are handled and if the data is in the index, many copies of the block are produced
- The reverse index is an useful tricK:
  - Take the key (e.g., 1234) and reverse it before inserting (1234 becomes 4321)
  - That way sequential values (1234 and 1235) go to different pages (inserted 4321 and 5321)

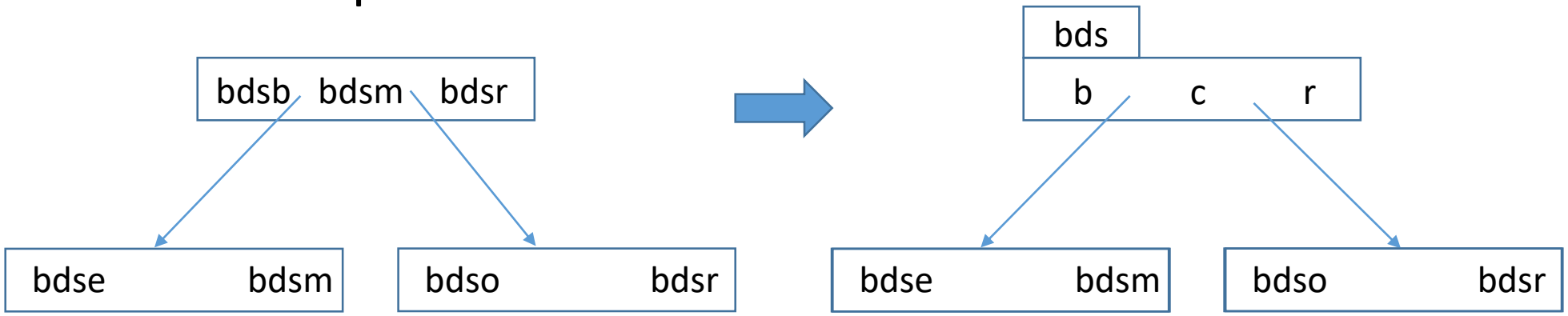
Optimizations

- A B+ tree contains many keys in the inner nodes. Depending on the type of the attribute, this can become expensive
- Several optimizations possible:
  - Replace separators that correspond to actual keys with shorter separators that have the same effect



Optimizations

- Factor common prefix



- This makes a lot of sense since entries that are next to each other are likely to be very similar (same as Frame of Reference compression)
- Mostly on inner nodes to keep scans cheap

Optimizations

- Ignore the rules of the B+ tree
  - Do not merge nodes when they do not have enough data (it takes times)
  - Delaying a merge could minimize changes (an insert may arrive later)
  - Instead, periodically rebuild the tree
- Variable length keys (a bit of a problem)
  - Number of entries is no longer between k/2 and k
  - Use smaller values as separators, place long values in leaves

Optimizations

- Similar to blocks:
  - Make sure there is always space so that insertions can be done quickly
  - Do not fill blocks to the max
- Depth vs breadth:
  - If the nodes is very large, it contains a lot of data and the tree is shallower => good for slow storage devices
  - If the node is small, there are many nodes, tree must be deeper => good in fast storage
  - Slow disks => larger nodes (potentially over several blocks but sequential)
  - Fast memory => smaller nodes (several nodes in a block)

Relation to segments and blocks

- Segments are used to refer to database entities like tables and indexes
- An index is stored in its own segment and can use blocks of different size than the data blocks
- An index can have its own memory buffer to avoid that working on the index affects working on the data
- Indexes are treated as first class citizens, like tables
- Often, queries can be answered by looking at the index rather than at the data

Other indexing techniques

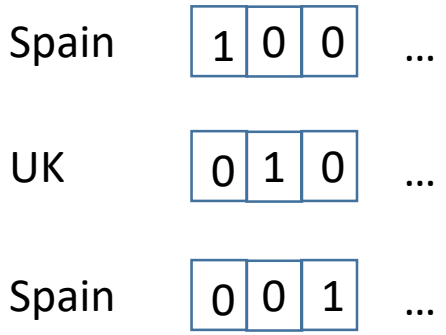
Query selectivity

- Query selectivity is important when considering using an index
  - Selectivity: how many tuples are returned (highly selective => few tuples in the results; low selectivity => many tuples in the results)
  - Index discussed so far work best for queries with high selectivity
- Alternative to use an index is a table scan (read the entire table)
  - Could be faster if table is small
  - It is faster if query has low enough selectivity:
    - Many tuples are selected
    - Each tuple has to traverse index and follow indirection (if data not in index)
  - Selection predicates can be complex, with many constraints

Bitmaps

- For every possible value that an attribute can take, represent that value with an array where position i is set to 1 if tuple i contains that value:

Name	Country
Juan	Spain
Mary	UK
Ueli	Switzerland



Bitmaps

- Simple predicate selections very efficient using bitmap
- Can be used to perform operations directly on the bitmaps
  - T.a = 3 ^ T.b =5 0 > just intersect the corresponding bitmap for every column
- Usually sparse 0> compress very well (run length encoding)
- Often used for large tables in analytics as it is faster than other indexes for low selectivity queries
  - Especially useful for data types where comparison is expensive

Materialized aggregates

- For groups of data that does not change very often, one can compute and store some basic statistics:
  - Sum, Avg, count, min, max …
- The aggregates can be used as a small index to check whether the data needed is within that group
- The aggregates can be used to compute aggregates over the whole table without having to read all the data, only the aggregates

Specialized indexes

- Text:
  - Tries
  - Patricia Trees
  - Inverted indexes (mapping of words to documents)
- Spatial:
  - R trees
  - Grid file
- …