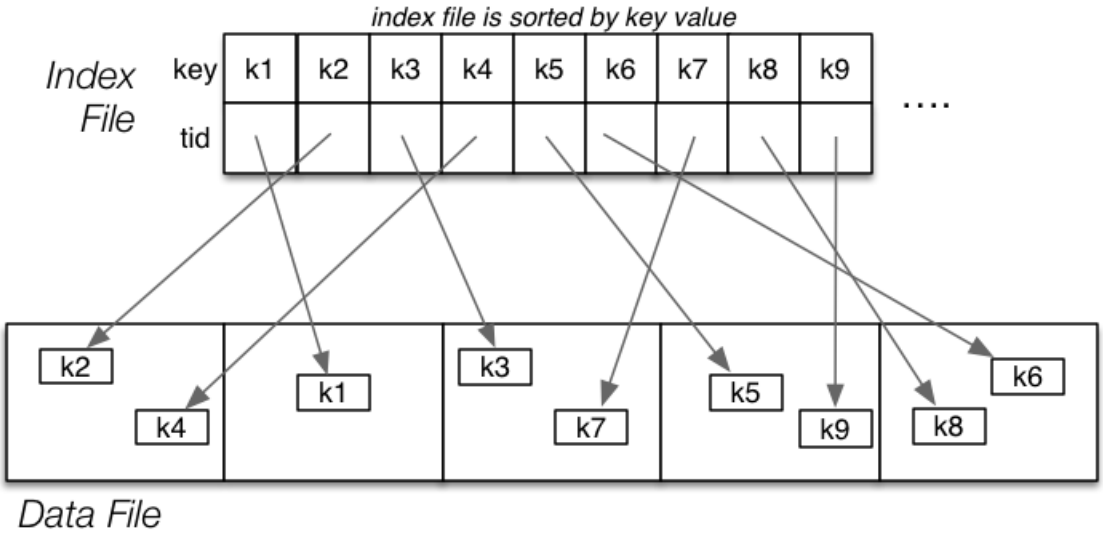
Indexing

❖ Indexing

An index is a file of (keyVal,tupleID) pairs, e.g.



❖ Indexes

A 1-d index is based on the value of a single attribute *A*.

Some possible properties of *A*:

- may be used to sort data file (or may be sorted on some other field)
- values may be unique (or there may be multiple instances)

Taxonomy of index types, based on properties of index attribute:

primary	index on unique field, may be sorted on *A*
clustering	index on non-unique field, file sorted on *A*
secondary	file *not* sorted on *A*

A given table may have indexes on several attributes.

❖ Indexes (cont)

Indexes themselves may be structured in several ways:

dense	every tuple is referenced by an entry in the index file
sparse	only some tuples are referenced by index file entries
single-level	tuples are accessed directly from the index file
multi-level	may need to access several index pages to reach tuple

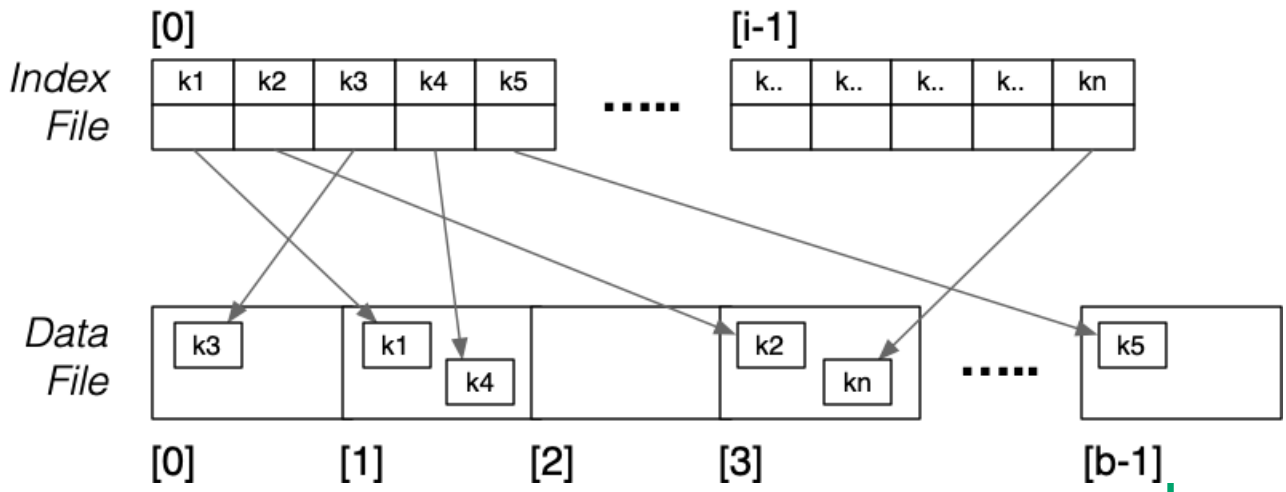
Index file has total i pages (where typically $i \ll b$)

Index file has page capacity c_i (where typically $c_i \gg c$)

Dense index: $i = ceil(r/c_i)$ Sparse index: $i = ceil(b/c_i)$

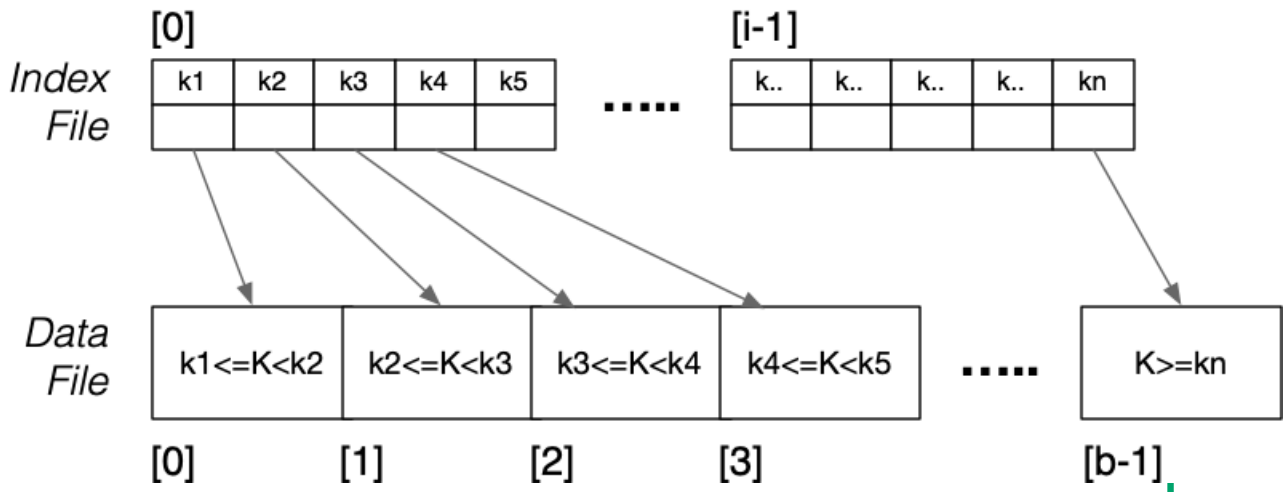
❖ Dense Primary Index

Data file unsorted; one index entry for each tuple



❖ Sparse Primary Index

Data file sorted; one index entry for each page



❖ Selection with Primary Index

For *one* queries:

```
ix = binary search index for entry with key K
if nothing found { return NotFound }
b = getPage(pageOf(ix.tid))
t = getTuple(b,offsetOf(ix.tid))
    -- may require reading overflow pages
return t
```

Worst case: read $log_2 i$ index pages + read $1+Ov$ data pages.

Thus, $Cost_{one,prim} = log_2 i + 1 + Ov$

Assume: index pages are same size as data pages ⇒ same reading cost

❖ Selection with Primary Index (cont)

For *range* queries on primary key:

- use index search to find lower bound
- read index sequentially until reach upper bound
- accumulate set of buckets to be examined
- examine each bucket in turn to check for matches

For *pmr* queries involving primary key:

- search as if performing *one* query.

For queries not involving primary key, index gives no help.

❖ Selection with Primary Index (cont)

Method for range queries (when data file is not sorted)

```
// e.g. select * from R where a between lo and hi
pages = {}   results = {}
ixPage = findIndexPage(R.ixf,lo)
while (ixTup = getNextIndexTuple(R.ixf)) {
   if (ixTup.key > hi) break;
   pages = pages ∪ pageOf(ixTup.tid)
}
foreach pid in pages {
   // scan data page plus ovflow chain
   while (buf = getPage(R.datf,pid)) {
      foreach tuple T in buf {
         if (lo<=T.a && T.a<=hi)
            results = results ∪ T
} } }
```

❖ Insertion with Primary Index

Overview:

```
tid = insert tuple into page P at position p
find location for new entry in index file
insert new index entry (k,tid) into index file
```

Problem: order of index entries must be maintained

- need to avoid overflow pages in index (but see later)
- so, reorganise index file by moving entries up

Reorganisation requires, on average, read/write half of index file:

$Cost_{insert,prim} = (log_2 i)_r + i/2.(1_r+1_w) + (1+Ov)_r + (1+\delta)_w$

❖ Deletion with Primary Index

Overview:

```
find tuple using index
mark tuple as deleted
delete index entry for tuple
```

If we delete index entries by marking ...

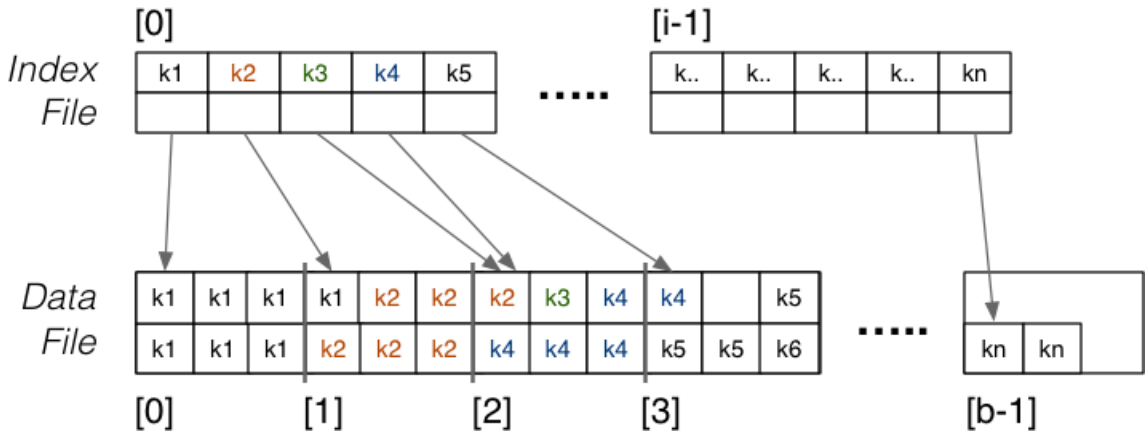
- $Cost_{delete,prim} = (log_2\ i)_r + (1 + Ov)_r + 1_w + 1_w$

If we delete index entry by index file reorganisation ...

- $Cost_{delete,prim} = (log_2\ i)_r + (1 + Ov)_r + i/2.(1_r+1_w) + 1_w$

❖ Clustering Index

Data file sorted; one index entry for each key value

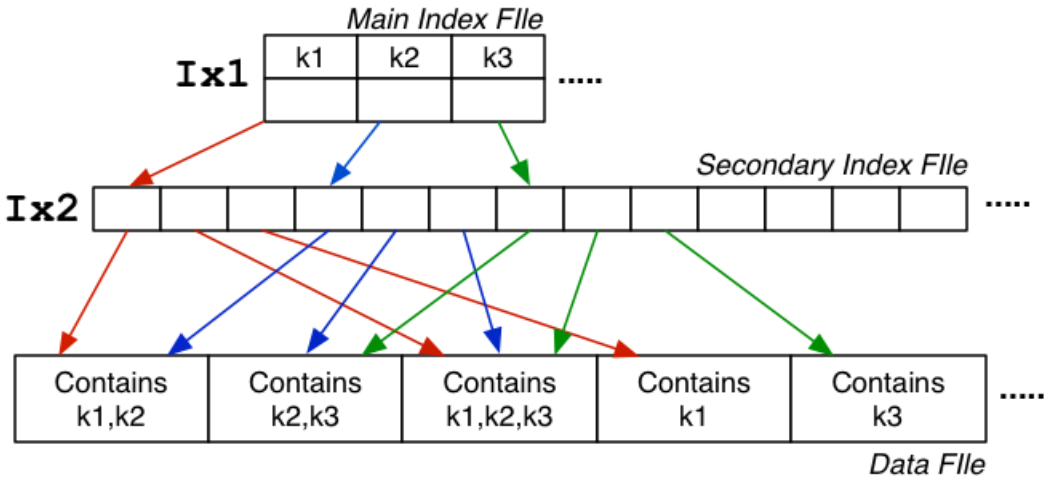


Cost penalty: maintaining both index and data file as sorted

(Note: can't mark index entry for value X until all X tuples are deleted)

❖ Secondary Index

Data file not sorted; want one index entry for each key value



$Cost_{pmr} = (log_2 i_{ix1} + a_{ix2} + b_q.(1 + Ov))$

❖ Multi-level Indexes

Secondary Index used two index files to speed up search

- by keeping the initial index search relatively quick
- **`Ix1`** small (depends on number of unique key values)
- **`Ix2`** larger (depends on amount of repetition of keys)
- typically, $b_{Ix1} \ll b_{Ix2} \ll b$

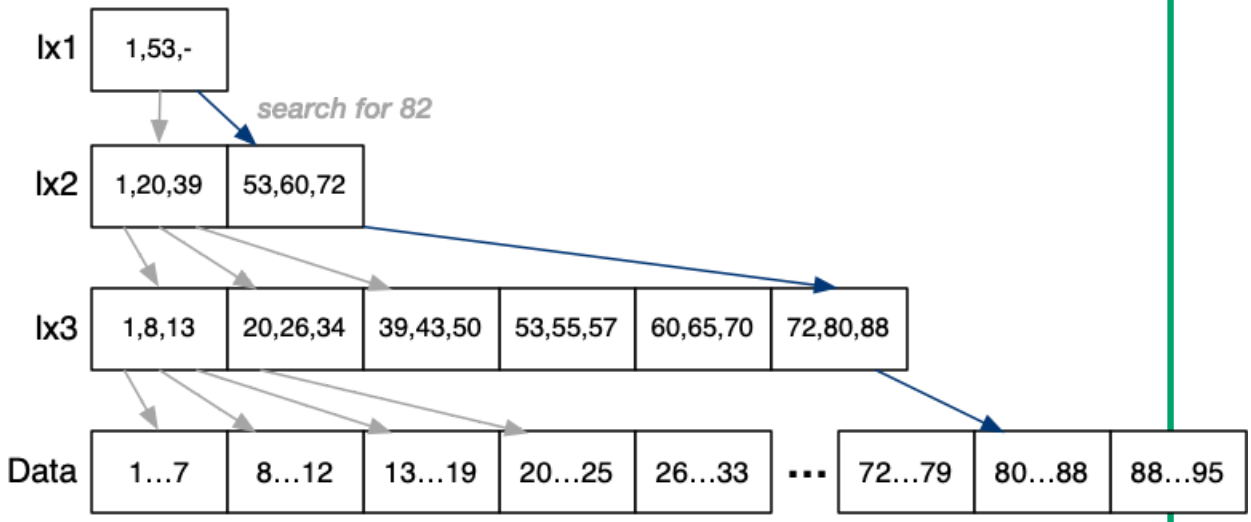
Could improve further by

- making **`Ix1`** sparse, since **`Ix2`** is guaranteed to be ordered
- in this case, $b_{Ix1} = ceil(b_{Ix2} / c_i)$
- if **`Ix1`** becomes too large, add **`Ix3`** and make **`Ix2`** sparse
- if data file ordered on key, could make **`Ix3`** sparse

Ultimately, reduce top-level of index hierarchy to one page.

❖ Multi-level Indexes (cont)

Example data file with three-levels of index:



Assume: not primary key, $c = 20$, $c_i = 3$

In reality, more likely $c = 100$, $c_i = 1000$

❖ Select with Multi-level Index

For *one* query on indexed key field:

```
xpid = top level index page
for level = 1 to d {
    read index entry xpid
    search index page for J'th entry
        where index[J].key <= K < index[J+1].key
    if (J == -1) { return NotFound }
    xpid = index[J].page
}
pid = xpid  // pid is data page index
search page pid and its overflow pages
```

$Cost_{one,mli} = (d + 1 + Ov)_r$

(Note that $d = ceil(log_{c_i} r)$ and c_i is large because index entries are small)

B-trees

❖ B-Trees

B-trees are multi-way search trees with the properties:

- they are updated so as to remain balanced
- each node has at least *(n-1)/2* entries in it
- each tree node occupies an entire disk page

B-tree insertion and deletion methods

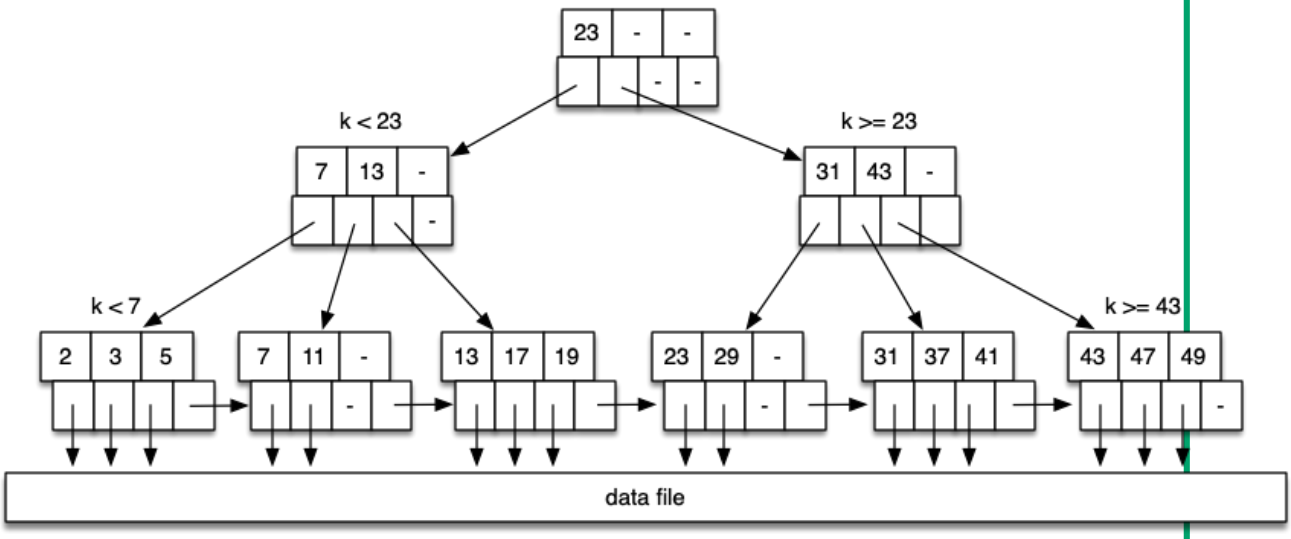
- are moderately complicated to describe
- can be implemented very efficiently

Advantages of B-trees over general multi-way search trees:

- better storage utilisation (around 2/3 full)
- better worst case performance (shallower)

❖ B-Trees (cont)

Example B-tree (depth=3, n=3) (actually B+ tree)



(Note: in DBs, nodes are pages ⇒ large branching factor, e.g. *n=500*)

❖ B-Tree Depth

Depth depends on effective branching factor (i.e. how full nodes are).

Simulation studies show typical B-tree nodes are 69% full.

Gives load $L_i = 0.69 \times c_i$ and depth of tree ~ $ceil(log_{L_i} r)$.

Example: $c_i=128$, $L_i=88$

Level	#nodes	#keys
root	1	88
1	89	7832
2	7921	697048
3	704969	62037272

Note: c_i is generally larger than 128 for a real B-tree.

❖ Selection with B-Trees

For *one* queries:

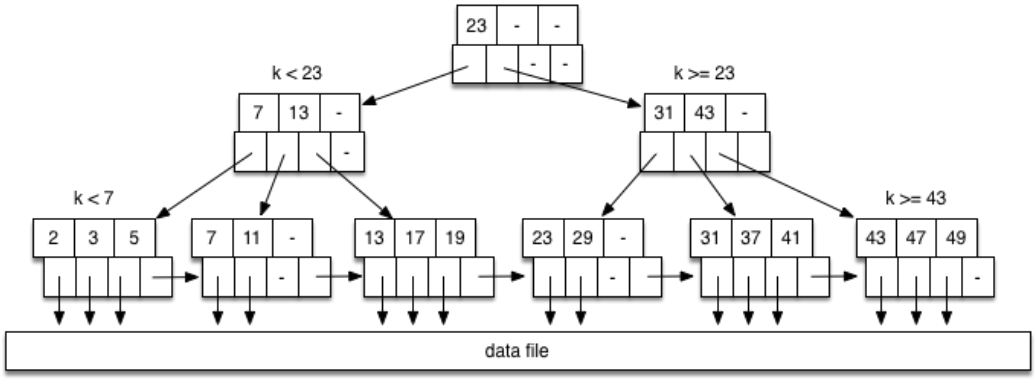
```
Node find(k,tree) {
   return search(k, root_of(tree))
}
Node search(k, node) {
   // get the page of the node
   if (is_leaf(node)) return node
   keys = array of nk key values in node
   pages = array of nk+1 ptrs to child nodes
   if (k <= keys[0])
      return search(k, pages[0])
   else if (keys[i] < k <= keys[i+1])
      return search(k, pages[i+1])
   else if (k > keys[nk-1])
      return search(k, pages[nk])
}
```

❖ Selection with B-Trees (cont)

Simplified description of search ...

```
N = B-tree root node
while (N is not a leaf node) N = scanToFindChild(N,K)
tid = scanToFindEntry(N,K)
access tuple T using tid
```

$Cost_{one} = (D + 1)_r$

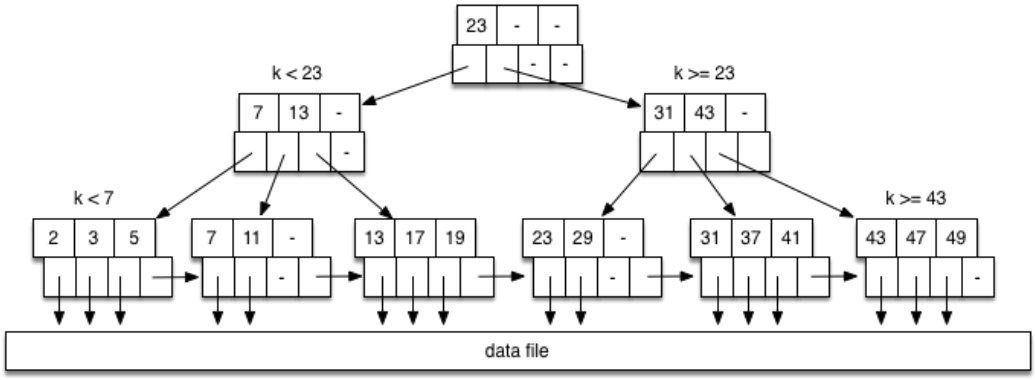


❖ Selection with B-Trees (cont)

For *range* queries (assume sorted on index attribute):

```
search index to find leaf node for Lo
for each leaf node entry until Hi found
    add pageOf(tid) to Pages to be scanned
scan Pages looking for matching tuples
```

$Cost_{range} = (D + b_i + b_q)_r$



❖ Insertion into B-Trees

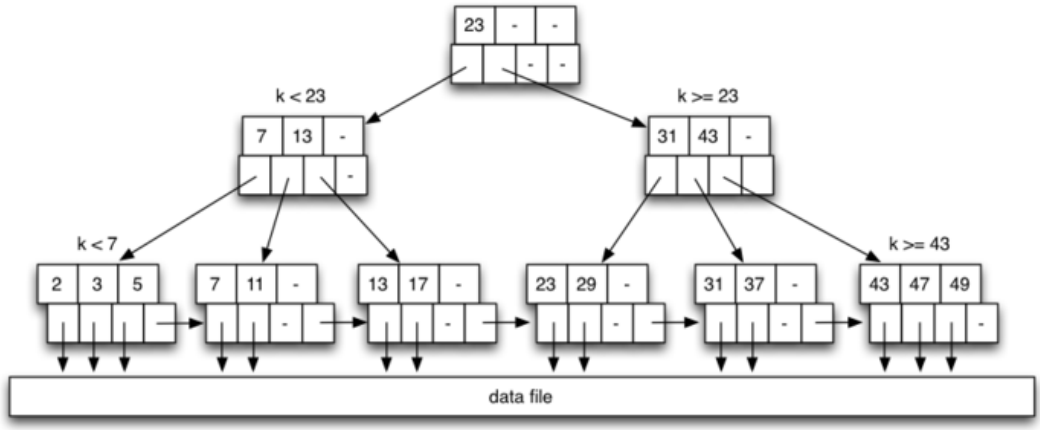
Overview of the method:

1. find leaf node and position in node where new key belongs
2. if node is not full, insert entry into appropriate position
3. if node is full ...
   - promote middle element to parent
   - split node into two half-full nodes (< middle, > middle)
   - insert new key into appropriate half-full node
4. if parent full, split and promote upwards
5. if reach root, and root is full, make new root upwards

Note: if duplicates not allowed and key exists, may stop after step 1.

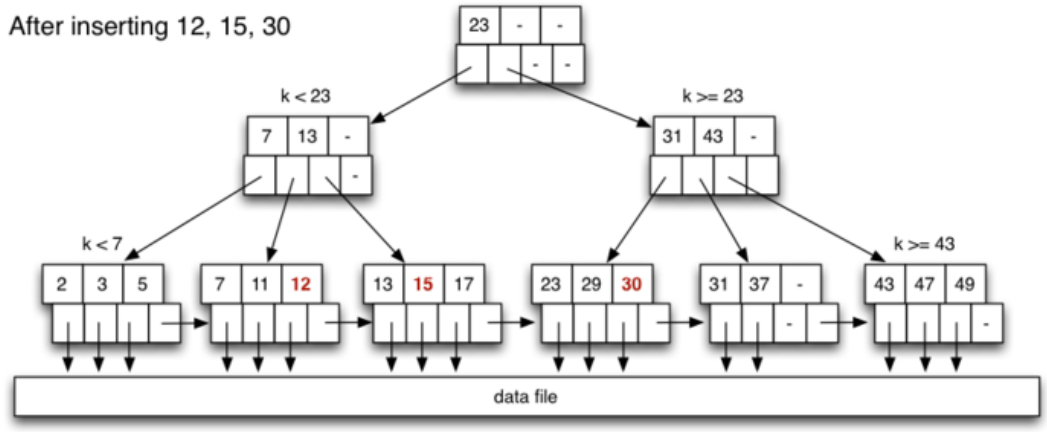
❖ Example: B-tree Insertion

Starting from this tree:

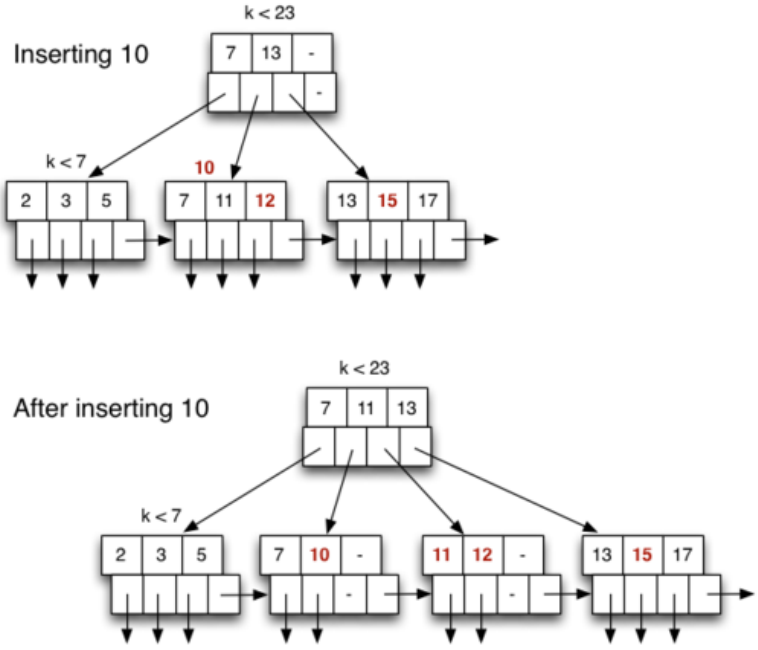


insert the following keys in the given order **12 15 30 10**

❖ Example: B-tree Insertion (cont)



❖ Example: B-tree Insertion (cont)



❖ B-Tree Insertion Cost

Insertion cost = $Cost_{treeSearch}$ + $Cost_{treeInsert}$ + $Cost_{dataInsert}$

Best case: write one page (most of time)

- traverse from root to leaf
- read/write data page, write updated leaf

$Cost_{insert} = D_r + 1_w + 1_r + 1_w$

Common case: *3* node writes (rearrange 2 leaves + parent)

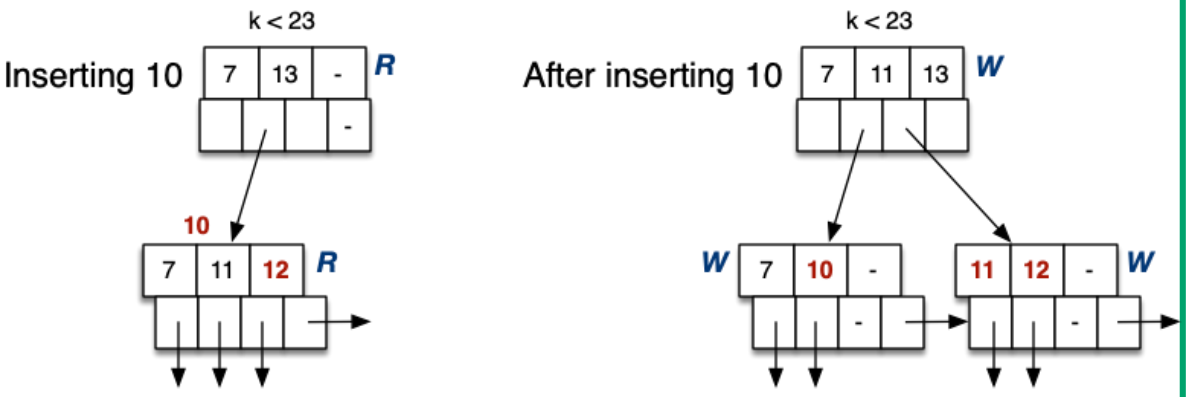
- traverse from root to leaf, holding nodes in buffer
- read/write data page
- update/write leaf, parent and sibling

$Cost_{insert} = D_r + 3_w + 1_r + 1_w$

❖ B-Tree Insertion Cost (cont)

Worst case: propagate to root $Cost_{insert} = D_r + D.3_w + 1_r + 1_w$

- traverse from root to leaf
- read/write data page
- update/write leaf, parent and sibling
- repeat previous step *D-1* times



❖ B-trees in PostgreSQL

PostgreSQL implements ≅ Lehman/Yao-style B-trees

- variant that works effectively in high-concurrency environments.

B-tree implementation: `backend/access/nbtree`

- `README` ... comprehensive description of methods
- `nbtree.c` ... interface functions (for iterators)
- `nbtsearch.c` ... traverse index to find key value
- `nbtinsert.c` ... add new entry to B-tree index

Notes:

- stores all instances of equal keys (dense index)
- avoids splitting by scanning right if key = max(key) in page
- common insert case: new key is max(key) overall; handled efficiently

❖ B-trees in PostgreSQL (cont)

Changes for PostgreSQL v12

- indexes smaller
  - for composite keys, only store first attribute
  - index entries are smaller, so c_i larger, so tree shallower
- include TID in index key
  - duplicate index entries are stored in "table order"
  - makes scanning table files to collect results more efficient

To explore indexes in more detail:

- `\di+ IndexName`
- `select * from bt_page_items(IndexName,BlockNo)`

❖ B-trees in PostgreSQL (cont)

Interface functions for B-trees

```
// build Btree index on relation
Datum btbuild(rel,index,...)
// insert index entry into Btree
Datum btinsert(rel,key,tupleid,index,...)
// start scan on Btree index
Datum btbeginscan(rel,key,scandesc,...)
// get next tuple in a scan
Datum btgettuple(scandesc,scandir,...)
// close down a scan
Datum btendscan(scandesc)
```

Multi-dimensional Search Trees

❖ Multi-dimensional Tree Indexes

Over the last 20 years, from a range of problem areas

- different multi-d tree index schemes have been proposed
- varying primarily in how they partition tuple-space

Consider three popular schemes: kd-trees, Quad-trees, R-trees.

Example data for multi-d trees is based on the following relation:

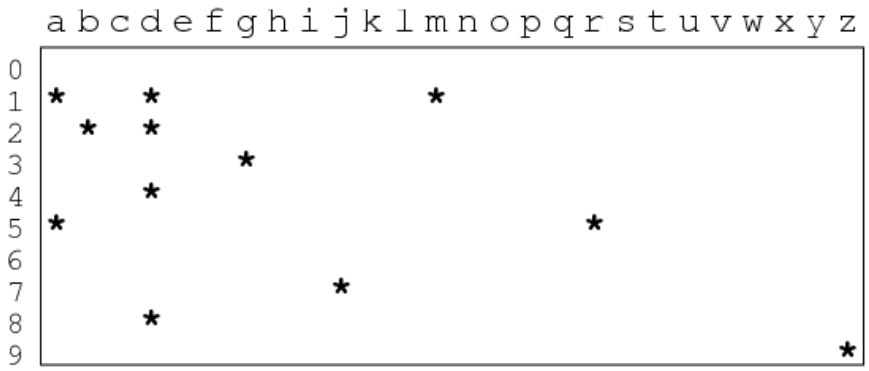
```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

❖ Multi-dimensional Tree Indexes (cont)

Example tuples:

```
R('a',1)  R('a',5)  R('b',2)  R('d',1)
R('d',2)  R('d',4)  R('d',8)  R('g',3)
R('j',7)  R('m',1)  R('r',5)  R('z',9)
```

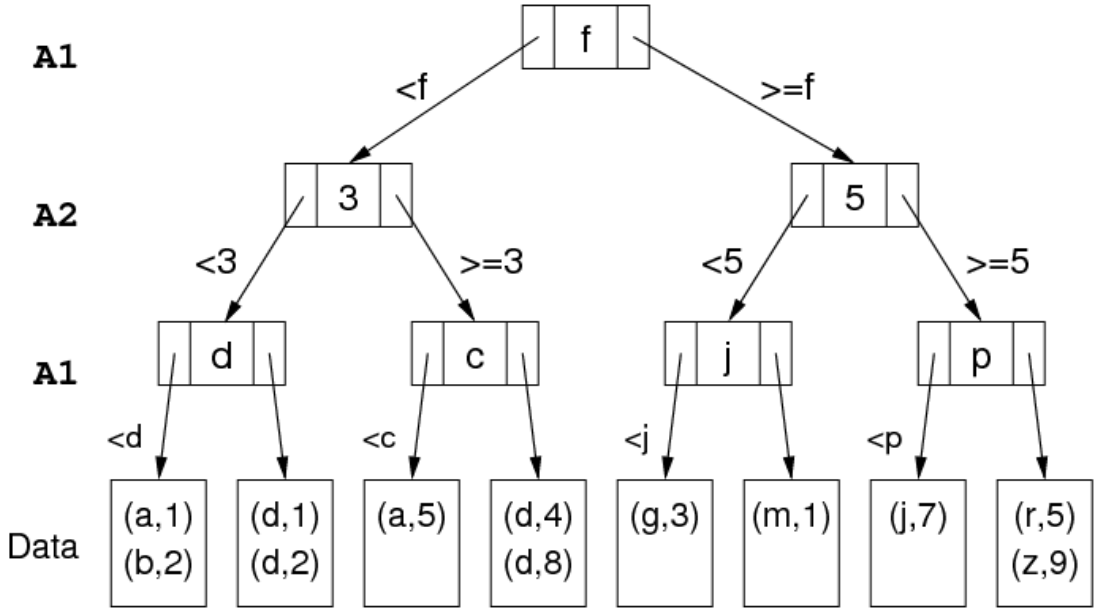
The tuple-space for the above tuples:



❖ kd-Trees

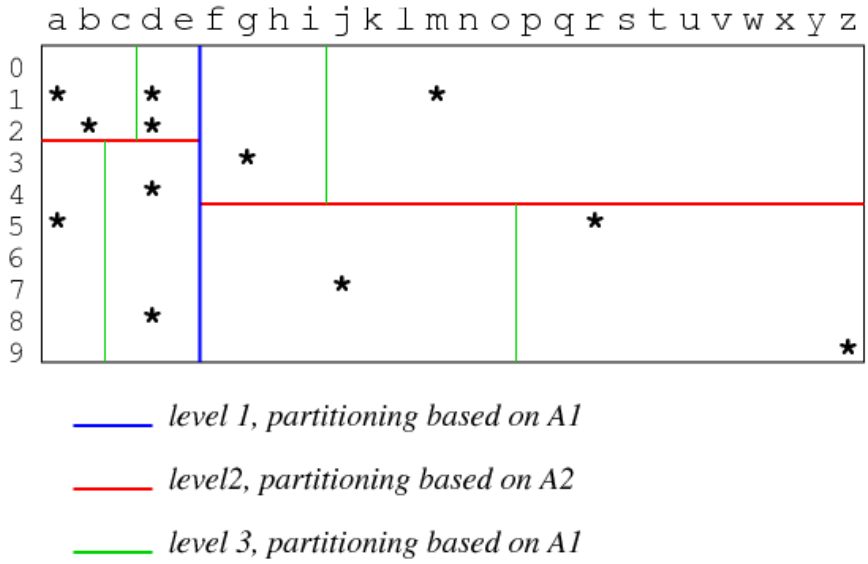
kd-trees are multi-way search trees where

- each level of the tree partitions on a different attribute
- each node contains *n-1* key values, pointers to *n* subtrees



❖ kd-Trees (cont)

How this tree partitions the tuple space:



❖ Searching in kd-Trees

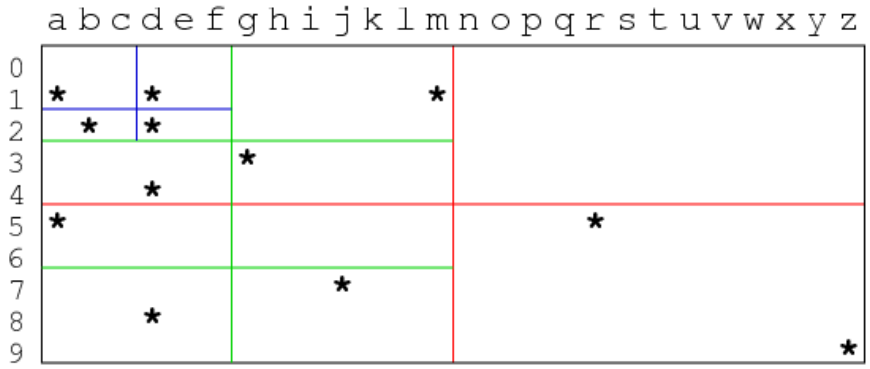
```
// Started by Search(Q, R, 0, kdTreeRoot)
Search(Query Q, Relation R, Level L, Node N)
{
   if (isDataPage(N)) {
      Buf = getPage(fileOf(R),idOf(N))
      check Buf for matching tuples
   } else {
      a = attrLev[L]
      if (!hasValue(Q,a))
         nextNodes = all children of N
      else {
         val = getAttr(Q,a)
         nextNodes = find(N,Q,a,val)
      }
      for each C in nextNodes
         Search(Q, R, L+1, C)
}  }
```

❖ Quad Trees

Quad trees use regular, disjoint partitioning of tuple space.

- for *2d*, partition space into quadrants (NW, NE, SW, SE)
- each quadrant can be further subdivided into four, etc.

Example:



❖ Quad Trees (cont)

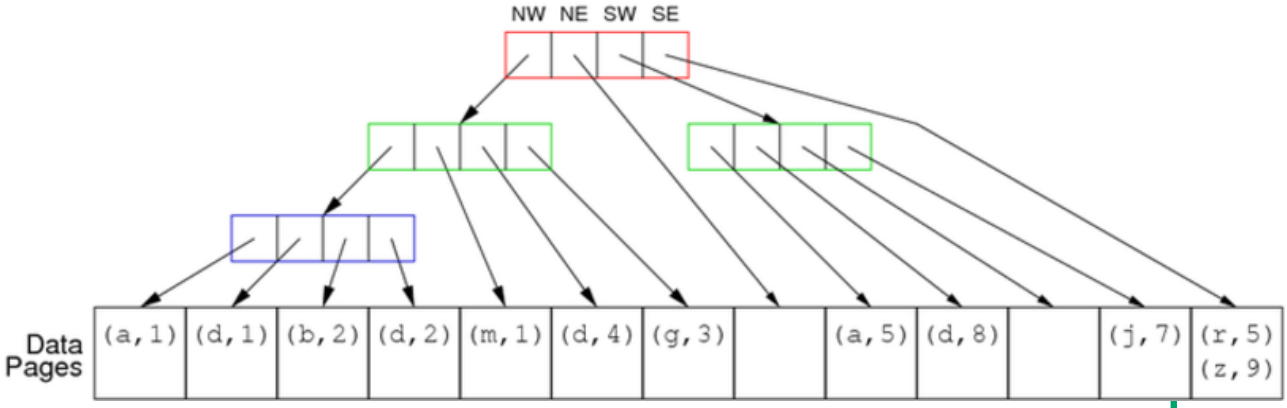
Basis for the partitioning:

- a quadrant that has no sub-partitions is a leaf quadrant
- each leaf quadrant maps to a single data page
- subdivide until points in each quadrant fit into one data page
- ideal: same number of points in each leaf quadrant (balanced)
- point density varies over space
  ⇒ different regions require different levels of partitioning
- this means that the tree is not necessarily balanced

Note: effective for $d \leq 5$, ok for $6 \leq d \leq 10$, ineffective for $d > 10$

❖ Quad Trees (cont)

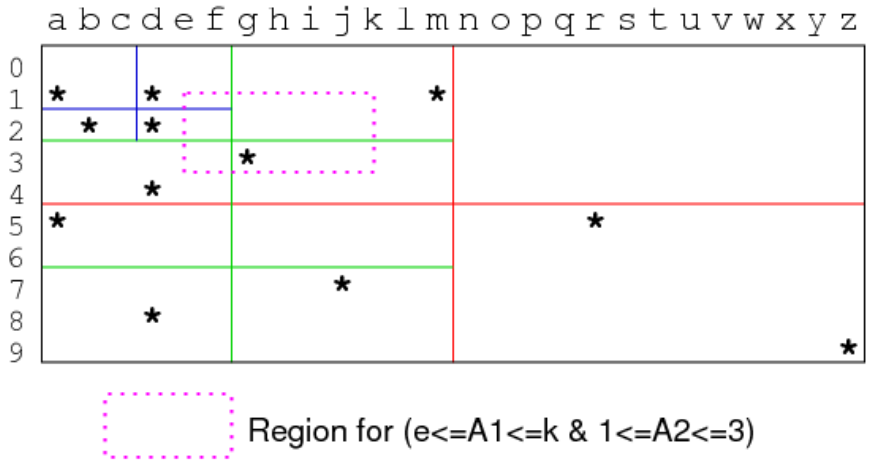
The previous partitioning gives this tree structure, e.g.



In this and following examples, we give coords of top-left,bottom-right of a region

❖ Searching in Quad-trees

Space query example:



Need to traverse: red(NW), green(NW,NE,SW,SE), blue(NE,SE).

❖ Searching in Quad-trees (cont)

Method for searching in Quad-tree:

- find all regions in current node that query overlaps with
- for each such region, check its node
  - if node is a leaf, check corresponding page for matches
  - else recursively repeat search from current node

Note that query region may be a single point.

❖ R-Trees

R-trees use a flexible, overlapping partitioning of tuple space.

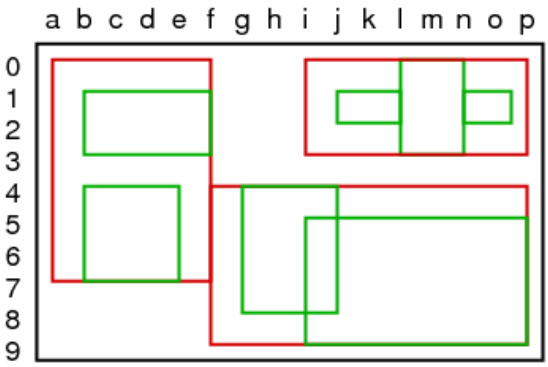
- each node in the tree represents a *k*d hypercube
- its children represent (possibly overlapping) subregions
- the child regions do not need to cover the entire parent region

Overlap and partial cover means:

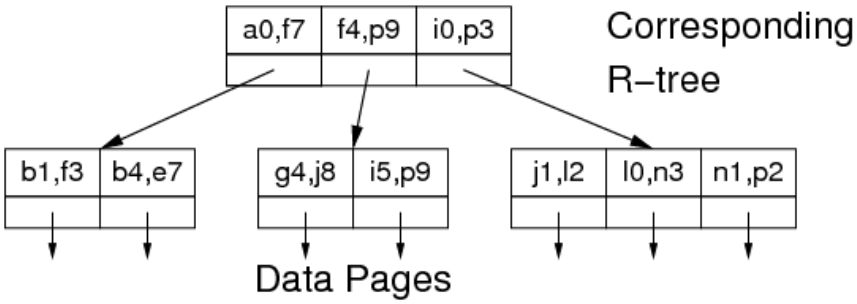
- can optimize space partitioning wrt data distribution
- so that there are similar numbers of points in each region

Aim: height-balanced, partly-full index pages (cf. B-tree)

❖ R-Trees (cont)



Space Partitioning



Corresponding R–tree

❖ Insertion into R-tree

Insertion of an object *R* occurs as follows:

- start at root, look for children that completely contain *R*
- if no child completely contains *R*, choose one of the children and expand its boundaries so that it does contain *R*
- if several children contain *R*, choose one and proceed to child
- repeat above containment search in children of current node
- once we reach data page, insert *R* if there is room
- if no room in data page, replace by two data pages
- partition existing objects between two data pages
- update node pointing to data pages
  (may cause B-tree-like propagation of node changes up into tree)

Note that *R* may be a point or a polygon.

❖ Query with R-trees

Designed to handle *space* queries and "where-am-I" queries.

"Where-am-I" query: find all regions containing a given point *P*:

- start at root, select all children whose subregions contain *P*
- if there are zero such regions, search finishes with *P* not found
- otherwise, recursively search within node for each subregion
- once we reach a leaf, we know that region contains *P*

Space (region) queries are handled in a similar way

- we traverse down any path that intersects the query region

❖ Costs of Search in Multi-d Trees

Cost depends on type of query and tree structure

Best case: *pmr* query where all attributes have known values

- in kd-trees and quad-trees, follow single tree path
- cost is equal to depth *D* of tree
- in R-trees, may follow several paths (overlapping partitions)

Typical case: some attributes are unknown or defined by range

- need to visit multiple sub-trees
- how many depends on: range, choice-points in tree nodes

❖ Multi-d Trees in PostgreSQL

Up to version 8.2, PostgreSQL had R-tree implementation

Superseded by GiST = Generalized Search Trees

GiST indexes parameterise: data type, searching, splitting

- via seven user-defined functions (e.g. **`picksplit()`**)

GiST trees have the following structural constraints:

- every node is at least fraction *f* full (e.g. 0.5)
- the root node has at least two children (unless also a leaf)
- all leaves appear at the same level

Details: Chapter 68 in PG Docs or src/backend/access/gist

Multi-dimensional Hashing

❖ Hashing and *pmr*

For a *pmr* query like

`select * from R where` $a_1 = C_1$ `and ...` `and` $a_n = C_n$

- if one a_i is the hash key, query is very efficient
- if no a_i is the hash key, need to use linear scan

Can be alleviated using *multi-attribute hashing* (*mah*)

- form a composite hash value involving all attributes
- at query time, some components of composite hash are known

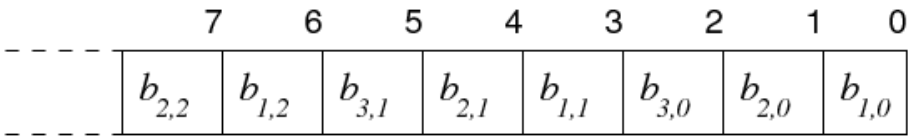
  (allows us to limit the number of data pages which need to be checked)

MA.hashing works in conjunction with any dynamic hashing scheme.

❖ Hashing and *pmr* (cont)

Multi-attribute hashing parameters:

- file size = $b = 2^d$ pages ⇒ use d-bit hash values
- relation has n attributes: $a_1, a_2, \ldots a_n$
- attribute a_i has hash function h_i
- attribute a_i contributes d_i bits (to the combined hash value)
- total bits $d = \sum_{i=1..n} d_i$
- a choice vector (cv) specifies for all $k \in 0..d\text{-}1$
  bit j from $h_i(a_i)$ contributes bit k in combined hash value



❖ MA.Hashing Example

Consider relation
`Deposit(branch,acctNo,name,amount)`

Assume a small data file with 8 main data pages (plus overflows).

Hash parameters: $d=3$ $d_1=1$ $d_2=1$ $d_3=1$ $d_4=0$

Note that we ignore the **`amount`** attribute ($d_4=0$)

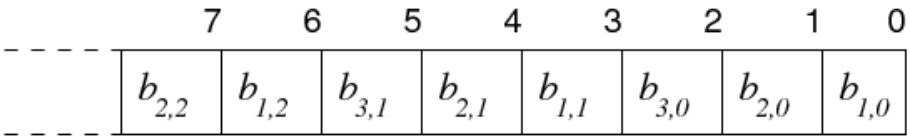
Assumes that nobody will want to ask queries like

```
select * from Deposit where amount=533
```

Choice vector is designed taking expected queries into account.

❖ MA.Hashing Example (cont)

Choice vector:



This choice vector tells us:

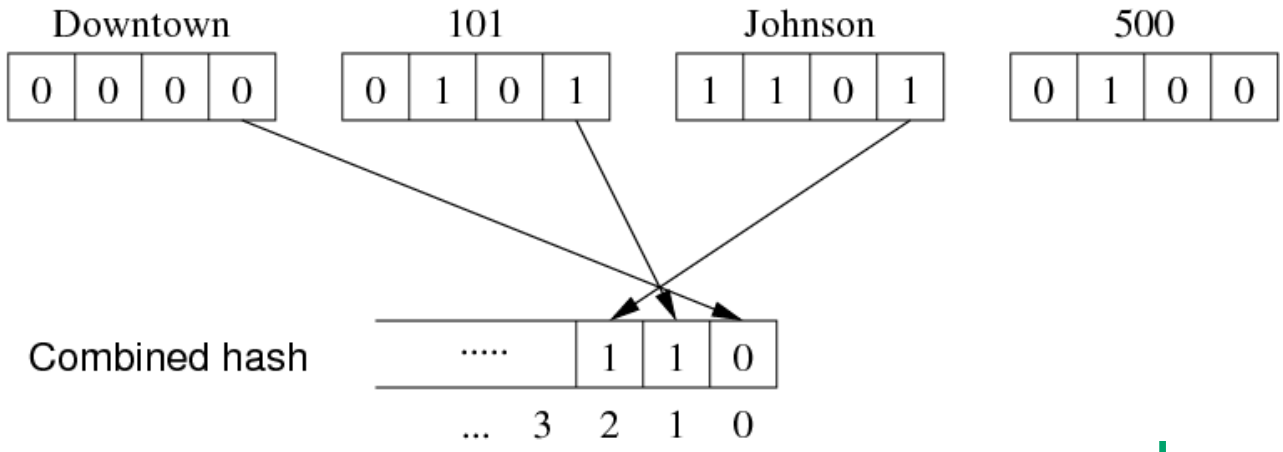
- bit 0 in hash comes from bit 0 of $hash_1(a_1)$ ($b_{1,0}$)
- bit 1 in hash comes from bit 0 of $hash_2(a_2)$ ($b_{2,0}$)
- bit 2 in hash comes from bit 0 of $hash_3(a_3)$ ($b_{3,0}$)
- bit 3 in hash comes from bit 1 of $hash_1(a_1)$ ($b_{1,1}$)
- etc. etc. etc. (up to as many bits of hashing as required, e.g. 32)

❖ MA.Hashing Example (cont)

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



❖ MA.Hashing Hash Functions

Auxiliary definitions:

```
#define MaxHashSize 32
typedef unsigned int HashVal;

// extracts i'th bit from hash value
#define bit(i,h) (((h) & (1 << (i))) >> (i))

// choice vector elems
typedef struct { int attr, int bit } CVelem;
typedef CVelem ChoiceVec[MaxHashSize];

// hash function for individual attributes
HashVal hash_any(char *val) { ... }
```

❖ MA.Hashing Hash Functions (cont)

Produce combined *d*-bit hash value for tuple *t* :

```
HashVal hash(Tuple t, ChoiceVec cv, int d)
{
    HashVal h[nAttr(t)+1];  // hash for each attr
    HashVal res = 0, oneBit;
    int     i, a, b;
    for (i = 1; i <= nAttr(t); i++)
        h[i] = hash_any(attrVal(t,i));
    for (i = 0; i < d; i++) {
        a = cv[i].attr;
        b = cv[i].bit;
        oneBit = bit(b, h[a]);
        res = res | (oneBit << i);
    }
    return res;
}
```

❖ Queries with MA.Hashing

In a partial match query:

- values of some attributes are known
- values of other attributes are unknown

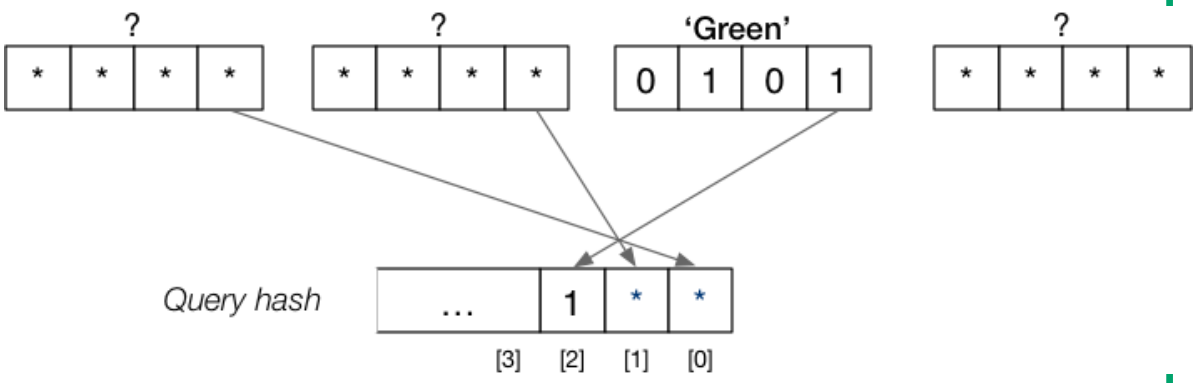
E.g.

```
select amount
from   Deposit
where  branch = 'Brighton' and name = 'Green'
```

for which we use the shorthand **(Brighton, ?, Green, ?)**

❖ Queries with MA.Hashing (cont)

Consider query: select amount from Deposit where name='Green'



Matching tuples must be in pages: **100**, **101**, **110**, **111**.

❖ MA.Hashing Query Algorithm

```
// Builds the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing

ns = 0;   // # unknown bits = # stars
for each attribute i{
    if (hasValue(Q,i)) {
        set d[i] bits in composite hash
            using choice vector and hash(Q,i)
    } else {
        set d[i] *'s in composite hash
            using choice vector
        ns += d[i]
    }
}
...
```

❖ MA.Hashing Query Algorithm (cont)

```
...
// Use the partial hash to find candidate pages

r = openRelation("R",READ);
for (i = 0; i < 2^ns; i++) {
    P = composite hash
    replace *'s in P
        using ns bits in i
    Buf = getPage(fileOf(r), P);
    for each tuple T in Buf {
        if (T satisfies pmr query)
            add T to results
    }
}
```

❖ Query Cost for MA.Hashing

Multi-attribute hashing handles a range of query types, e.g.

```
select * from R where a=1
select * from R where d=2
select * from R where b=3 and c=4
select * from R where a=5 and b=6 and c=7
```

A relation with n attributes has 2^n different query types.

Different query types have different costs (different no. of *'s)

$Cost(Q) = 2^s$ where $s = \sum_{i \notin Q} d_i$ (alternatively $Cost(Q) = \prod_{i \notin Q} 2^{d_i}$)

Query distribution gives probability p_Q of asking each query type Q.

❖ Query Cost for MA.Hashing (cont)

Min query cost occurs when all attributes are used in query

$Min\ Cost_{pmr} = 1$

Max query cost occurs when no attributes are specified

$Max\ Cost_{pmr} = 2^d = b$

Average cost is given by weighted sum over all query types:

$Avg\ Cost_{pmr} = \sum_Q p_Q \prod_{i \notin Q} 2^{d_i}$

Aim to minimise the weighted average query cost over possible query types

❖ Optimising MA.Hashing Cost

For a given application, useful to minimise $Cost_{pmr}$.

Can be achieved by choosing appropriate values for d_i (cv)

Heuristics:

- distribution of query types (more bits to frequently used attributes)
- size of attribute domain (≤ #bits to represent all values in domain)
- discriminatory power (more bits to highly discriminating attributes)

Trade-off: making Q_j more efficient makes Q_k less efficient.

This is a combinatorial optimisation problem
(solve via standard optimisation techniques e.g. simulated annealing)

Signature-based Indexing

❖ Indexing with Signatures

Signature-based indexing:

- designed for *pmr* queries (conjunction of equalities)
- does not try to achieve better than *O(n)* performance
- attempts to provide an "efficient" linear scan

Each tuple is associated with a signature

- a compact (lossy) descriptor for the tuple
- formed by combining information from multiple attributes
- stored in a signature file, parallel to data file

Instead of scanning/testing tuples, do pre-filtering via signatures.

❖ Indexing with Signatures (cont)

File organisation for signature indexing (two files)

One signature slot per tuple slot; unused signature slots are zeroed.

Signatures do not determine record placement ⇒ can use with other indexing.

❖ Signatures

A signature "summarises" the data from one tuple

A tuple consists of n attribute values A_1 .. A_n

A codeword $cw(A_i)$ is

- a bit-string, m bits long, where k bits are set to 1 ($k \ll m$)
- derived from the value of a single attribute A_i

A tuple descriptor (signature) is built

- by combining $cw(A_i)$, $i=1..n$
- aim to have roughly half of the bits set to 1

Two strategies for building signatures: overlay, concatenate

❖ Generating Codewords

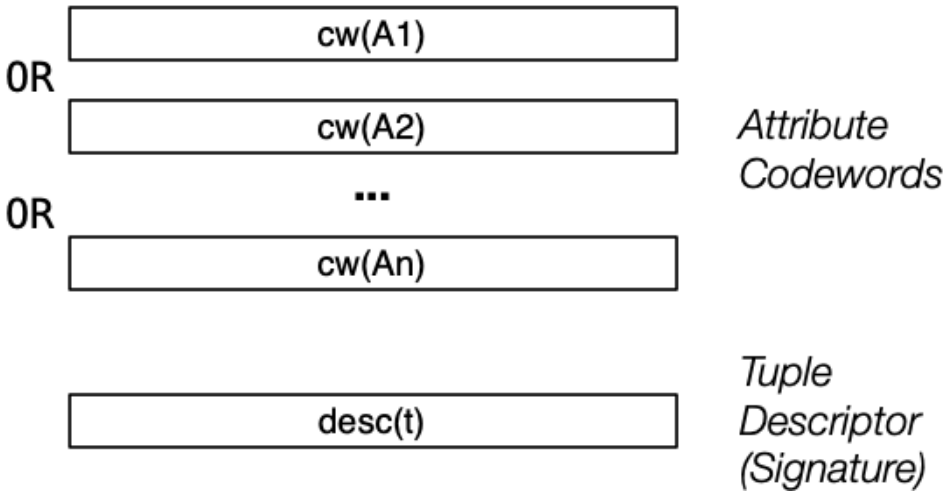
Generating a *k-in-m* codeword for attribute A_i

```
bits codeword(char *attr_value, int m, int k)
{
    int  nbits = 0;    // count of set bits
    bits cword = 0;    // assuming m <= 32 bits
    srandom(hash(attr_value));
    while (nbits < k) {
        int i = random() % m;
        if (((1 << i) & cword) == 0) {
            cword |= (1 << i);
            nbits++;
        }
    }
    return cword;  // m-bits with k 1-bits and m-k 0-bits
}
```

❖ Superimposed Codewords (SIMC)

In a superimposed codewords (simc) indexing scheme

- tuple descriptor formed by *overlaying* attribute codewords (bitwise-OR)



❖ Superimposed Codewords (SIMC) (cont)

A SIMC tuple descriptor *desc(t)* is

- a bit-string, m bits long, where $j \leq nk$ bits are set to 1
- $desc(t) = cw(A_1)$ **OR** $cw(A_2)$ **OR** ... **OR** $cw(A_n)$

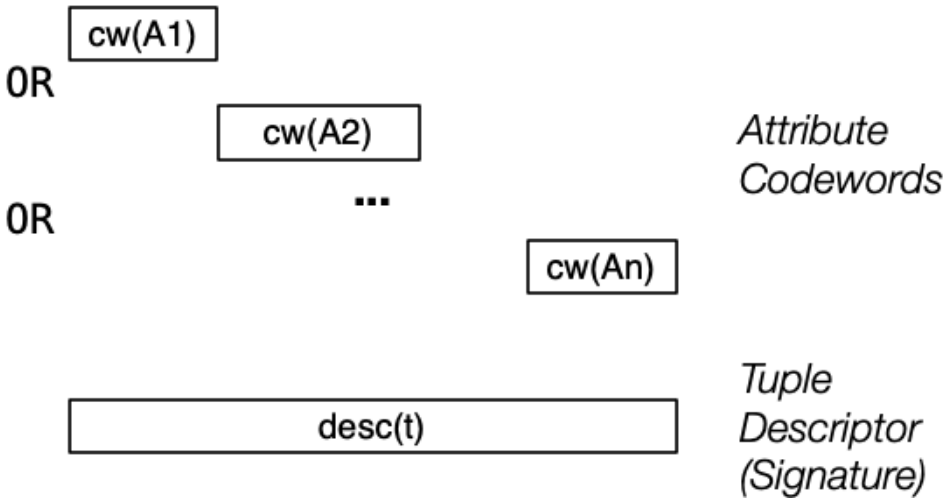
Method (assuming all n attributes are used in descriptor):

```
Bits desc = 0
for (i = 1; i <= n; i++) {
   bits cw = codeword(A[i],m,k)
   desc = desc | cw
}
```

❖ Concatenated Codewords (CATC)

In a concatenated codewords (catc) indexing schema

- tuple descriptor formed by *concatenating* attribute codewords



❖ Concatenated Codewords (CATC) (cont)

A CATC tuple descriptor *desc(t)* is

- a bit-string, *m* bits long, where *j* = *nk* bits are set to 1
- *desc(t)* = $cw(A_1)$ + $cw(A_2)$ + ... + $cw(A_n)$ (+ is concatenation)

Each codeword is *p* = *m/n* bits long, with *k* = *p/2* bits set to 1

Method (assuming all *n* attributes are used in descriptor):

```
Bits desc = 0 ;  int p = m/n
for (i = 1; i <= n; i++) {
   bits cw = codeword(A[i],p,k)
   desc = desc | (cw << p*(n-i))
}
```

❖ Queries using Signatures

To answer query *q* with a signature-based index

- first generate a query descriptor *desc(q)*
- then scan the signature file using the query descriptor
- if sig_i matches *desc(q)*, then tuple *i* may be a match

desc(q) is formed from codewords of known attributes.

Effectively, any unknown attribute A_i has $cw(A_i) = 0$

E.g. for SIMC **(a,?,c,?,e)** = *cw*(**a**) **OR** 0 **OR** *cw*(**c**) **OR** 0 **OR** *cw*(**e**)

❖ Queries using Signatures (cont)

Once we have a query descriptor, we search the signature file:

```
pagesToCheck = {}
// scan r descriptors
for each descriptor D[i] in signature file {
    if (matches(D[i],desc(q))) {
        pid = pageOf(tupleID(i))
        pagesToCheck = pagesToCheck ∪ pid
    }
}
// scan b_q + δ data pages
for each pid in pagesToCheck {
    Buf = getPage(dataFile,pid)
    check tuples in Buf for answers
}
```

❖ False Matches

Both SIMC and CATC can produce false matches

- `matches(D[i],desc(q))` is true, but `Tup[i]` is not a solution for `q`

Why does this happen?

- signatures are based on hashing, and it is possible that

  $hash(key_1) == hash(key_2)$ even though $key_1 != key_2$

- for SIMC, overlaying could also produce "unfortunate" bit-combinations

To mitigate this, need to choose "good" *m* and *k*

❖ SIMC vs CATC

Both build *m*-bit wide signatures, with ~1/2 bits set to 1

Both have codewords with ~*m/2n* bits set to 1

CATC: codewords are *m/n* = *p*-bits wide

- shorter codewords → more hash collisions

SIMC: codewords are also *m*-bits wide

- longer codewords ⇒ less hash collisions, but also has overlay collisions

CATC has option of different length codeword p_i for each A_i $(\sum p_i = m)$