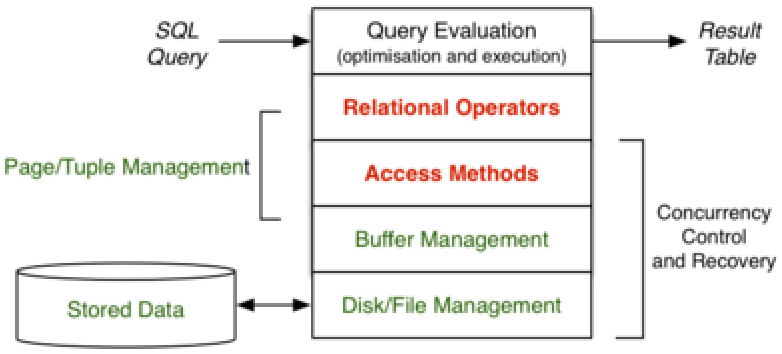
Week 06 Lectures

Implementing Relational Operators

Implementation of relational operations in DBMS:



... Implementing Relational Operators

So far, have considered file structures ...

- *heap file* ... tuples added to any page which has space
- *sorted file* ... tuples arranged in file in key order
- *hash file* ... tuples placed in pages using hash function

with relational algebra operations ...

- scanning (e.g. `select * from R`)
- sorting (e.g. `select * from R order by x`)
- projection (e.g. `select x,y from R`)
- selection (e.g. `select * from R where` *`Cond`*)

and now ...

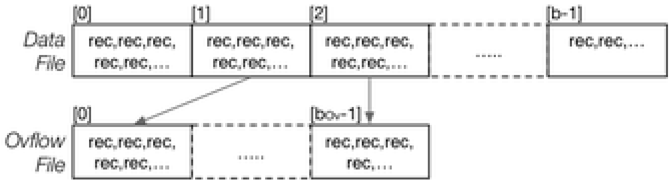
- *indexes* ... search trees based on pages/keys
- *signatures* ... bit-strings which "summarize" tuples

... Implementing Relational Operators

File/query Parameters ...

- *r* tuples of size *R*, *b* pages of size *B*, *c* tuples per page
- *Rel.k* attribute in `where` clause, b_q answer pages for query *q*
- b_{Ov} overflow pages, average overflow chain length *Ov*

File structures ...



Reminder on Cost Analyses

When showing the cost of operations ...

- for queries, simply count number of pages read
- for updates, use n_r and n_w to distinguish reads/writes

When comparing two methods for same query

- ignore the cost of writing the result (same for both)

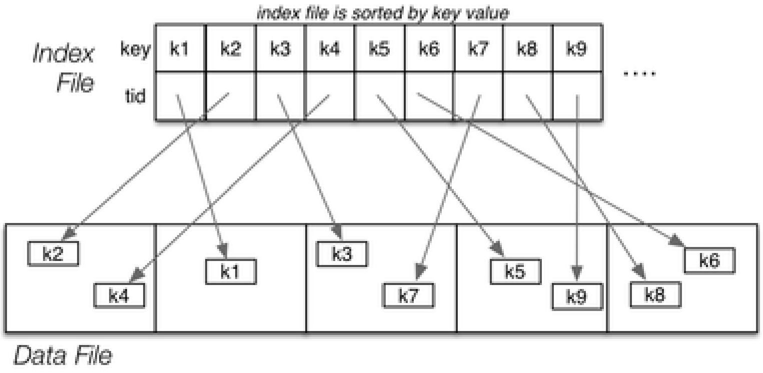
In counting reads and writes, assume minimal buffering

- each `request_page()` causes a read
- each `release_page()` causes a write (if page is dirty)

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

Indexes themselves may be structured in several ways:

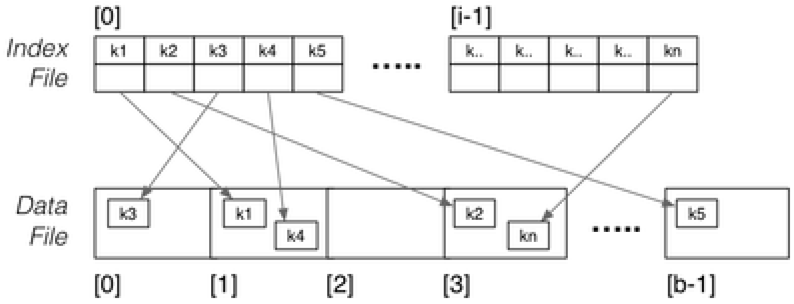
dense	every tuple is referenced by an entry in the index file
sparse	only some tuples are referenced by index file entries
single-level	tuples are accessed directly from the index file
multi-level	may need to access several index pages to reach tuple

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

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

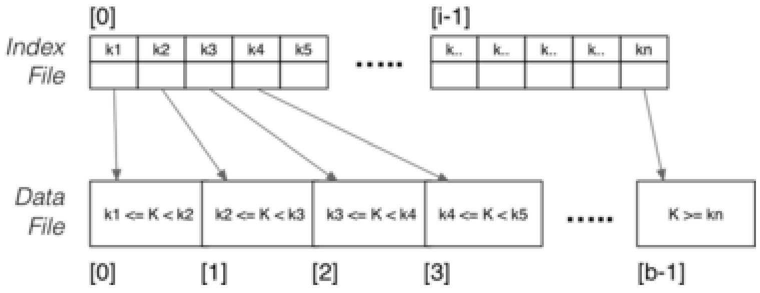
Dense index: $i = ceil(r/c_i)$ Sparse index: $i = ceil(b/c_i)$

Dense Primary Index



Data file unsorted; one index entry for each tuple

Sparse Primary Index



Data file sorted; one index entry for each page

Exercise 1: Index Storage Overheads

Consider a relation with the following storage parameters:

- *B = 8192,   R = 128,   r = 100000*
- header in data pages: 256 bytes
- key is integer, data file is sorted on key
- index entries (keyVal,tupleID): 8 bytes
- header in index pages: 32 bytes

How many pages are needed to hold a dense index?

How many pages are needed to hold a sparse index?

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

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

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
- either reorganise index file or mark entries

Reogranisation 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 ...

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

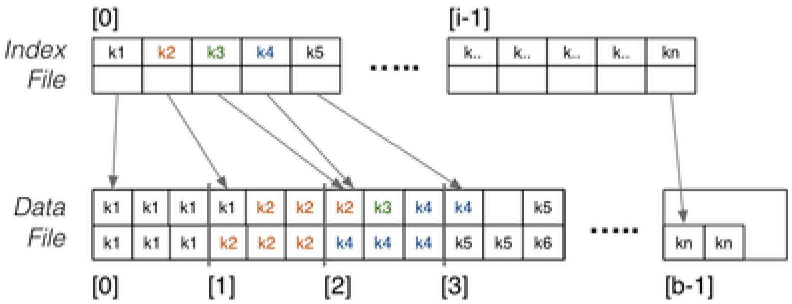
If we delete index entry by index file reorganisation ...

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

Clustering Index

17/96

Data file sorted; one index entry for each key value



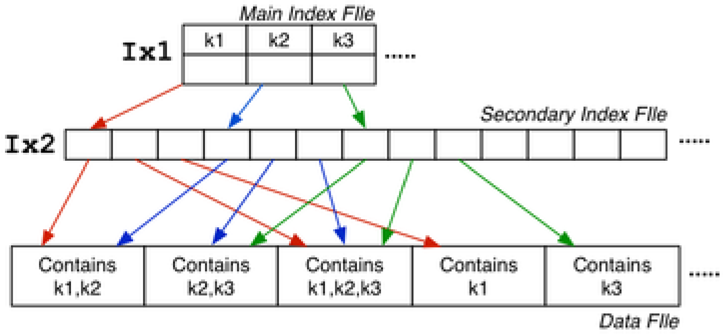
Cost penalty: maintaining both index and data file as sorted

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

Secondary Index

18/96

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

19/96

Above 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, $b_{Ix1} \ll b_{Ix2}$*

Could improve further by

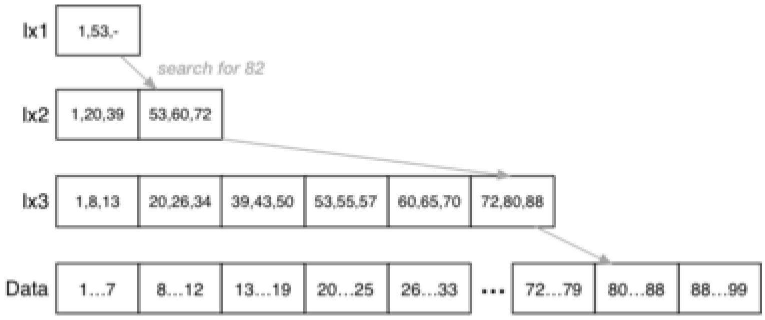
- *making `Ix1` sparse, since `Ix2` is guaranteed to be ordered*
- *in this case, $b_{Ix1} = ceil(b_{Ix2} / c_i)$*
- *if `Ix1` becomes too large, add `Ix3` and make `Ix2` sparse*
- *if data file ordered on key, could make `Ix3` sparse*

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

... Multi-level Indexes

20/96

Example data file with three-levels of index:



Assume: not primary key, $c = 100$, $c_i = 3$

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 are MSTs 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

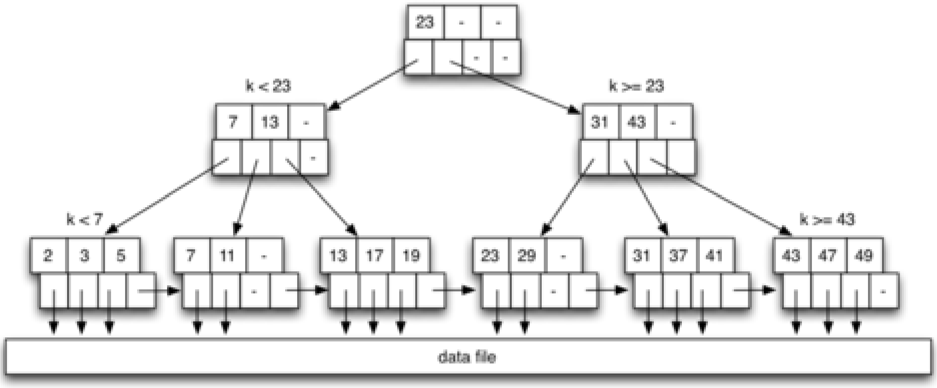
- *are moderately complicated to describe*
- *can be implemented very efficiently*

Advantages of B-trees over general MSTs

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

... B-Trees

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	87
1	88	7656
2	7744	673728
3	681472	59288064

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

Insertion into B-Trees

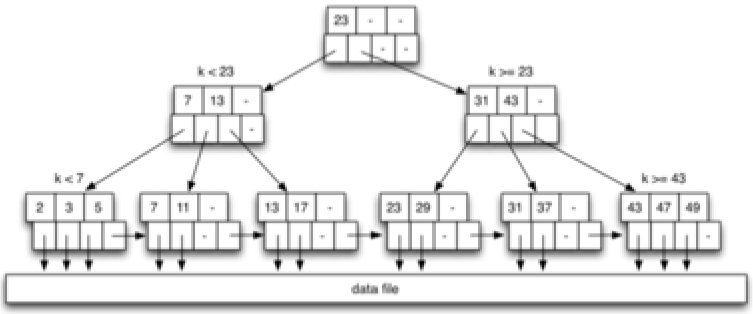
Overview of the method:

1. *find leaf node and position in node where entry would be stored*
2. *if node is not full, insert entry into appropriate spot*
3. *if node is full, split node into two half-full nodes*
   and promote middle element to parent
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.

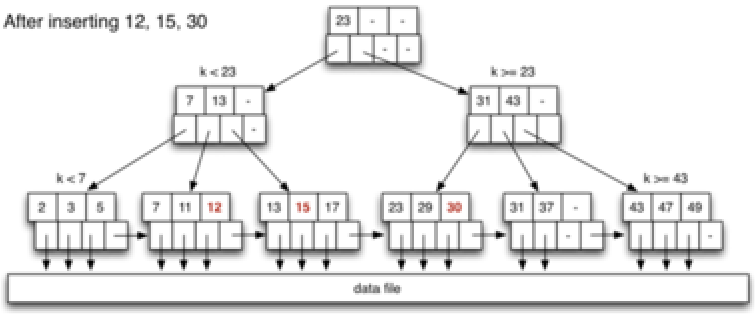
Example: B-tree Insertion

Starting from this tree:

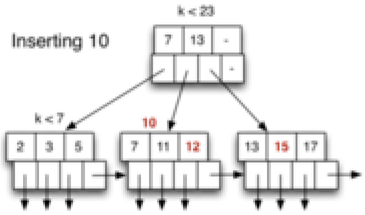


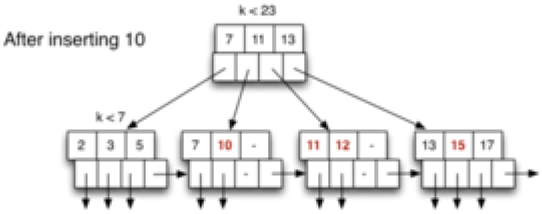
insert the following keys in the given order `12 15 30 10`

... Example: B-tree Insertion



... Example: B-tree Insertion





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

Worst case: 2D-1 node writes (propagate to root)

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

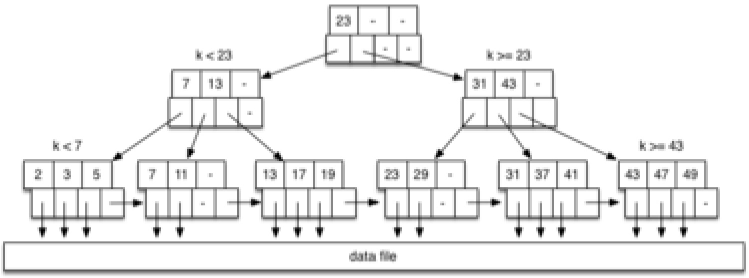
$Cost_{insert} = D_r + (2D-1)_w + 1_r + 1_w$

Selection with B-Trees

For one queries:

```
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
```

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

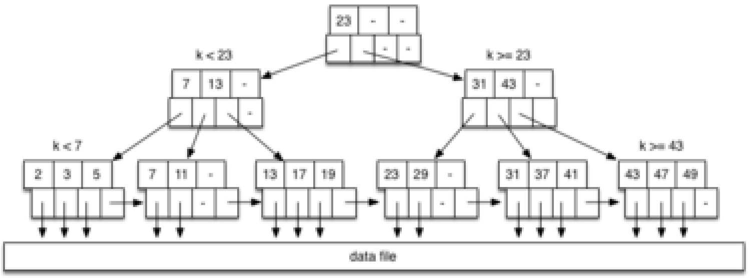


... Selection with B-Trees

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

```
search index to find leaf node for Lo
for each leaf node entry until Hi found {
        access tuple T using tid from entry
}
```

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



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*
- *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

34/96

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)
```

N-dimensional Selection

N-dimensional Queries

36/96

Have looked at one-dimensional queries, e.g.

```
select * from R where a = K
select * from R where a between Lo and Hi
```

and heaps, hashing, indexing as ways of efficient implementation.

Now consider techniques for efficient multi-dimensional queries.

Compared to 1-d queries, multi-dimensional queries

- *typically produce fewer results*
- *require us to consider more information*
- *require more effort to produce results*

Operations for Nd Select

37/96

N-dimensional select queries = condition on ≥1 attributes.

- *pmr = partial-match retrieval (equality tests), e.g.*

  ```
  select * from Employees
  where  job = 'Manager' and gender = 'M';
  ```

- *space = tuple-space queries (range tests), e.g.*

  ```
  select * from Employees
  where 20 ≤ age ≤ 50 and 40K ≤ salary ≤ 60K
  ```

N-d Selection via Heaps

38/96

Heap files can handle pmr or space using standard method:

```
// select * from R where C
r = openRelation("R",READ);
for (p = 0; p < nPages(r); p++) {
    buf = getPage(file(r), p);
    for (i = 0; i < nTuples(buf); i++) {
        t = getTuple(buf,i);
        if (matches(t,C))
            add t to result set
    }
}
```

$Cost_{pmr} = Cost_{space} = b$

N-d Selection via Multiple Indexes

DBMSs already support building multiple indexes on a table.

Which indexes to build depends on which queries are asked.

```
create table R (a int, b int, c int);
create index Rax on R (a);
create index Rbx on R (b);
create index Rcx on R (c);
create index Rabx on R (a,b);
create index Racx on R (a,c);
create index Rbcx on R (b,c);
create index Rallx on R (a,b,c);
```

But more indexes ⇒ space + update overheads.

N-d Queries and Indexes

Generalised view of pmr and space queries:

select * from R
where a_1 op_1 C_1 and ... and a_n op_n C_n

pmr : all op_i are equality tests.   space : some op_i are range tests.

Possible approaches to handling such queries ...

1. *use index on one a_i to reduce tuple tests*
2. *use indexes on all a_i, and intersect answer sets*

... N-d Queries and Indexes

If using just one of several indexes, which one to use?

select * from R
where a_1 op_1 C_1 and ... and a_n op_n C_n

The one with best selectivity for a_i op_i C_i (i.e. fewest matches)

Factors determining selectivity of a_i op_i C_i

- *assume uniform distribution of values in dom(a_i)*
- *equality test on primary key gives at most one match*
- *equality test on larger dom(a_i) ⇒ less matches*
- *range test over large part of dom(a_i) ⇒ many matches*

... N-d Queries and Indexes

Implementing selection using one of several indices:

```
// Query: select * from R where $a_1op_1C_1$ and ... and $a_nop_nC_n$
// choose $a_i$ with best selectivity
TupleIDs = IndexLookup(R,$a_i$,$op_i$,$C_i$)
// gives { $tid_1$, $tid_2$, ...} for tuples satisfying $a_iop_iC_i$
PageIDs = { }
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }

// PageIDs = a set of $b_{q_{ix}}$ page numbers
...
```

Cost = $Cost_{index}$ + $b_{q_{ix}}$ *(some pages do not contain answers, $b_{q_{ix}} > b_q$)*

DBMSs typically maintain statistics to assist with determining selectivity

... N-d Queries and Indexes

Implementing selection using multiple indices:

```
// Query: select * from R where $a_1op_1C_1$ and ... and $a_nop_nC_n$
// assumes an index on at least $a_i$
TupleIDs = IndexLookup(R,$a_1$,$op_1$,$C_1$)
foreach attribute $a_i$ with an index {
   tids = IndexLookup(R,$a_i$,$op_i$,$C_i$)
   TupleIDs = TupleIDs ∩ tids
}
PageIDs = { }
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }
// PageIDs = a set of $b_q$ page numbers
...
```

Cost = k.$Cost_{index}$ + b_q *(assuming indexes on k of n attrs)*

Exercise 2: One vs Multiple Indices

Consider a relation with r = 100,000, B = 4096, defined as:

```
create table Students (
    id       integer primary key,
    name     char(10), -- simplified
    gender   char(1),  -- 'm','f','?'
    birthday char(5)   -- 'MM-DD'
);
```

Assumptions:

- *data file is not ordered on any attribute*
- *has a dense B-tree index on each attribute*
- *96 bytes of header in each data/index page*

For `Students(id,name,gender,birthday)` *...*

- *calculate the size of the data file and each index*
- *describe the selectivity of each attribute*

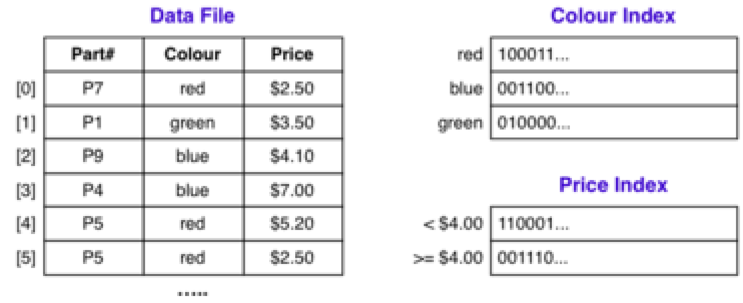
Now consider a query on this relation:

```
select * from Students
where  name='John' and birthday='04-01'
```

- *estimate the cost of answering using* `name` *index*
- *estimate the cost of answering using* `birthday` *index*
- *estimate the cost of answering using both indices*

Bitmap Indexes

Alternative index structure, focussing on sets of tuples:



Data File

	Part#	Colour	Price
[0]	P7	red	$2.50
[1]	P1	green	$3.50
[2]	P9	blue	$4.10
[3]	P4	blue	$7.00
[4]	P5	red	$5.20
[5]	P5	red	$2.50
	.....		

Colour Index

red	100011...
blue	001100...
green	010000...

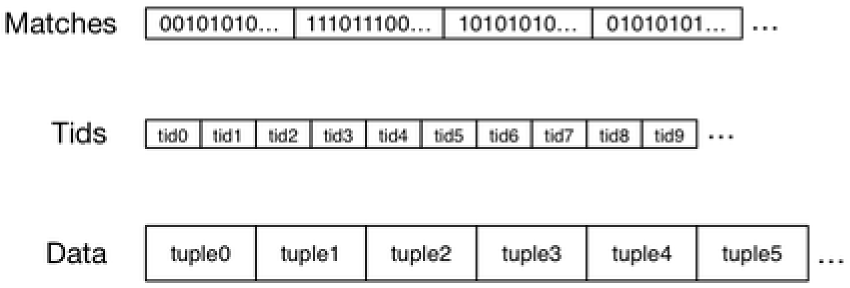
Price Index

< $4.00	110001...
>= $4.00	001110...

Index contains bit-strings of r bits, one for each value/range

... Bitmap Indexes

Also useful to have a file of `tids`*, giving file structures:*



Matches 00101010... 111011100... 10101010... 01010101... ...

Tids tid0 tid1 tid2 tid3 tid4 tid5 tid6 tid7 tid8 tid9 ...

Data tuple0 tuple1 tuple2 tuple3 tuple4 tuple5 ...

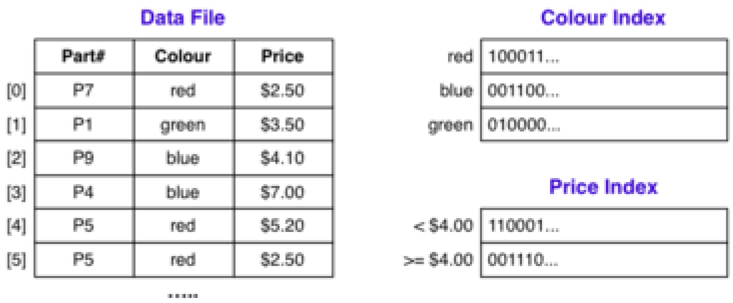
... Bitmap Indexes

Answering queries using bitmap index:

```
Matches = AllOnes(r)
foreach attribute A with index {
   // select i$^{th}$ bit-string for attribute A
   // based on value associated with A in WHERE
   Matches = Matches & Bitmaps[A][i]
}
// Matches contains 1-bit for each matching tuple
foreach i in 0..r-1 {
   if (Matches[i] == 0) continue;
   Pages = Pages U {pageOf(Tids[i])}
}
foreach pid in Pages {
   P = getPage(pid)
   extract matching tuples from P
}
```

Exercise 3: Bitmap Index

Using the following file structure:



Data File

	Part#	Colour	Price
[0]	P7	red	$2.50
[1]	P1	green	$3.50
[2]	P9	blue	$4.10
[3]	P4	blue	$7.00
[4]	P5	red	$5.20
[5]	P5	red	$2.50
		......	

Colour Index

red	100011...
blue	001100...
green	010000...

Price Index

< $4.00	110001...
>= $4.00	001110...

Show how the following queries would be answered:

```
select * from Parts
where colour='red' and price < 4.00

select * from Parts
where colour='green' or colour ='blue'
```

... Bitmap Indexes

Storage costs for bitmap indexes:

- *one bitmap for each value/range for each indexed attribute*
- *each bitmap has length ceil(r/8) bytes*
- *e.g. with 50K records and 8KB pages, bitmap fits in one page*

Query execution costs for bitmap indexes:

- *read one bitmap for each indexed attribute in query*
- *perform bitwise AND on bitmaps (in memory)*
- *read pages containing matching tuples*

Note: bitmaps could index pages rather than tuples (shorter bitmaps)

Hashing for N-d Selection

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 hash scheme.

... Hashing and *pmr*

Multi-attribute hashing parameters:

- *file size = b = 2^d pages ⇒ use d-bit hash values*
- *relation has n attributes: a_1, a_2, ...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 ...*
  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 ($d_4=0$)

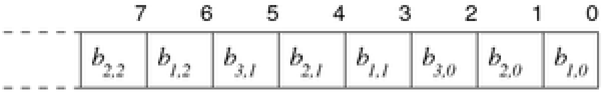
Assumes that nobody will want to ask queries like

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

Choice vector is designed taking expected queries into account.

... MA.Hashing Example

Choice vector:



This choice vector tells us:

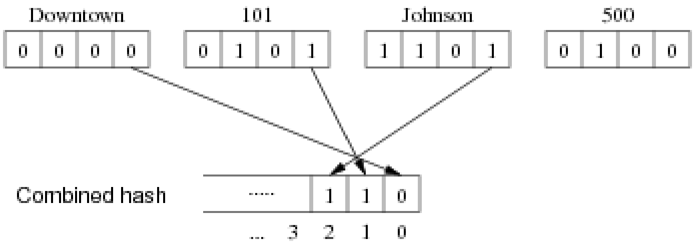
- *bit 0 in hash comes from bit 0 of $hash_1(a_1)$ ($b_{1,0}$)*
- *bit 1 in hash comes from bit 0 of $hash_2(a_2)$ ($b_{2,0}$)*
- *bit 2 in hash comes from bit 0 of $hash_3(a_3)$ ($b_{3,0}$)*
- *bit 3 in hash comes from bit 1 of $hash_1(a_1)$ ($b_{1,1}$)*
- *etc. etc. etc. (up to as many bits of hashing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

... MA.Hashing Example

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

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;
}
```

Exercise 4: Multi-attribute Hashing

Compute the hash value for the tuple

`('John Smith','BSc(CompSci)',1990,99.5)`

where $d=6$, $d_1=3$, $d_2=2$, $d_3=1$, and

- *cv = <(1,0), (1,1), (2,0), (3,0), (1,2), (2,1), (3,1), (1,3), ...>*
- *$hash_1$*`('John Smith')` = `...0101010110110100`
- *$hash_2$*`('BSc(CompSci)')` = `...1011111101101111`
- *$hash_3$*`(1990)` = `...0001001011000000`

Queries with MA.Hashing

In a partial match query:

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

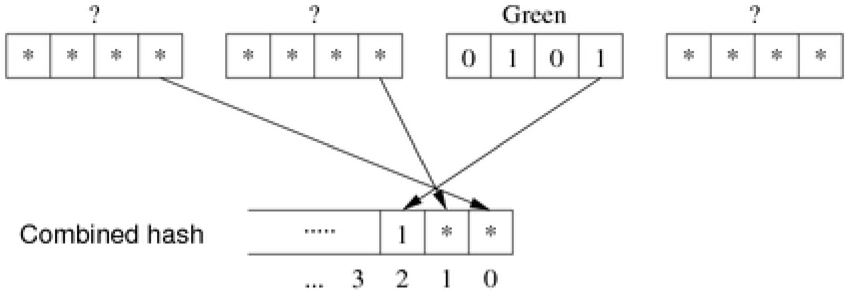
E.g.

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

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

... Queries with MA.Hashing

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



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

Exercise 5: Partial hash values in MAH

Given the following:

- *d=6, b=2^6, CV = <(0,0),(0,1),(1,0),(2,0),(1,1),(0,2), ...>*
- *hash (a) = ...00101101001101*
- *hash (b) = ...00101101001101*
- *hash (c) = ...00101101001101*

What are the query hashes for each of the following:

- *(a,b,c), (?,b,c), (a,?,?), (?,?,?)*

MA.Hashing Query Algorithm

```
// Builds the partial hash value (e.g. 10*0*1)
// Treats query like tuple with some attr values missing
nstars = 0;
for each attribute i in query Q {
    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
        nstars += d[i]
    }
}
...
```

... MA.Hashing Query Algorithm

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

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

Exercise 6: Representing Stars

Our hash values are bit-strings (e.g. `100101110101`*)*

MA.Hashing introduces a third value (`*` = unknown)

How could we represent "bit"-strings like `1011*1*0**010`?

Exercise 7: MA.Hashing Query Cost

Consider `R(x,y,z)` using multi-attribute hashing where

$d = 9 \quad d_x = 5 \quad d_y = 3 \quad d_z = 1$

How many buckets are accessed in answering each query?

```
1. select * from R where x = 4 and y = 2 and z = 1
2. select * from R where x = 5 and y = 3
3. select * from R where y = 99
4. select * from R where z = 23
5. select * from R where x > 5
```

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

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)

Exercise 8: MA.Hashing Design

Consider relation `Person(name,gender,age)` *...*

p_Q	Query Type *Q*

0.5	`select name from Person`
	`where gender=X and age=Y`

0.25	`select age from Person`
	`where name=X`

0.25	`select name from Person`
	`where gender=X`

Assume that all other query types have p_Q=0.

Design a choice vector to minimise average selection cost.

Tree Indexes for N-d Selection

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:

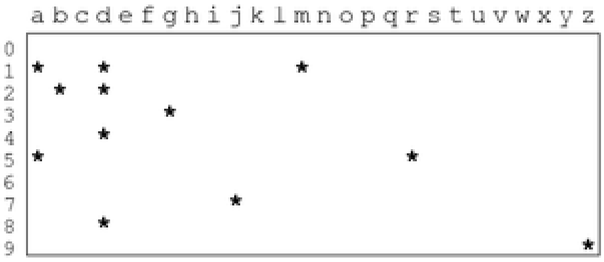
```
create table Rel (
    X char(1) check (X between 'a' and 'z'),
    Y integer check (Y between 0 and 9)
);
```

... Multi-dimensional Tree Indexes

Example tuples:

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

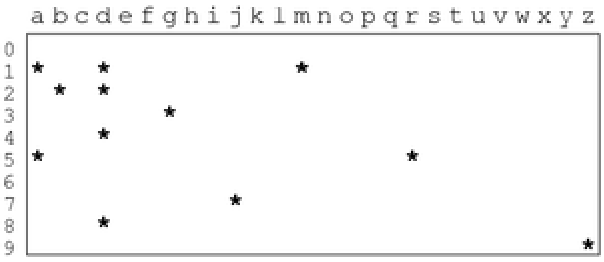
The tuple-space for the above tuples:



Exercise 9: Query Types and Tuple Space

Which part of the tuple-space does each query represent?

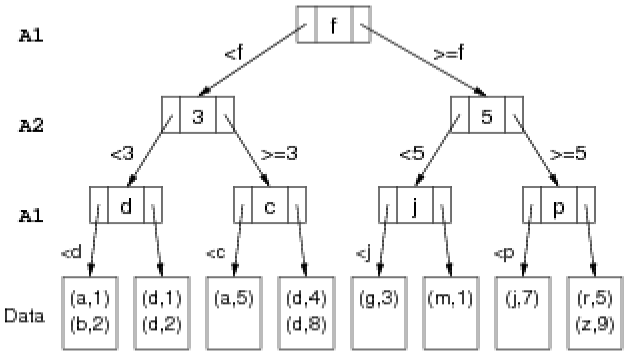
```
Q1: select * from Rel where X = 'd' and Y = 4
Q2: select * from Rel where 'j' < X ≤ 'r'
Q3: select * from Rel where X > 'm' and Y > 4
Q4: select * from Rel where 'k' ≤ X ≤ 'p' and 3 ≤ Y ≤ 6
```



kd-Trees

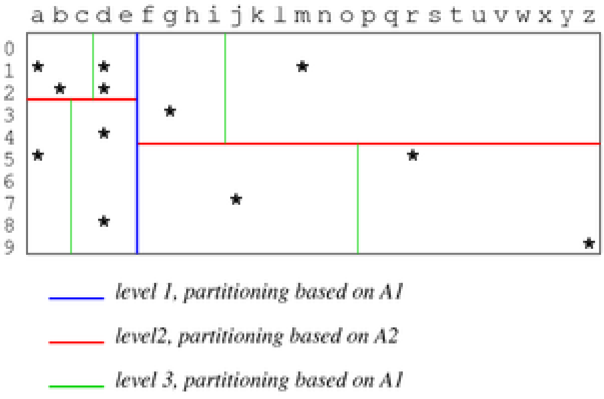
kd-trees are multi-way search trees where

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



... kd-Trees

How this tree partitions the tuple space:

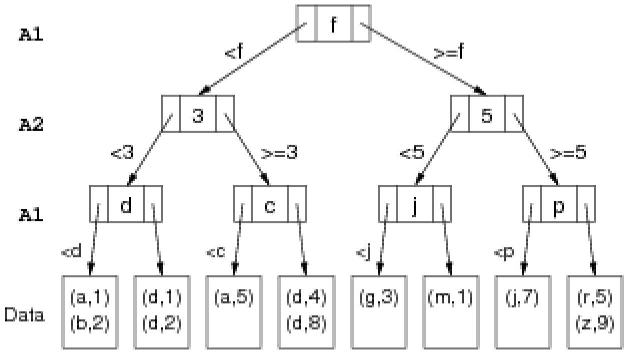

a b c d e f g h i j k l m n o p q r s t u v w x y z
0
1
2
3
4
5
6
7
8
9
level 1, partitioning based on A1
level2, partitioning based on A2
level 3, partitioning based on A1

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)
}  }
```

Exercise 10: Searching in kd-Trees

Using the following kd-tree index



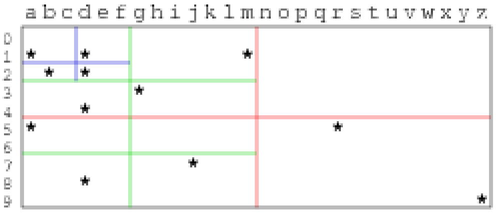
Answer the queries `(m,1), (a,?), (?,1), (?,?)`

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

Basis for the partitioning:

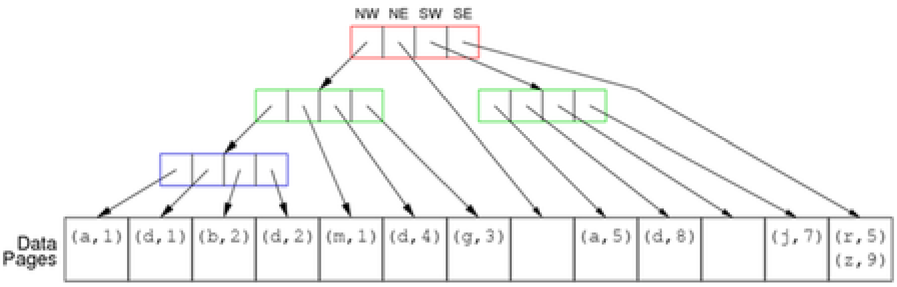
- *a quadrant that has no sub-partitions is a leaf quadrant*
- *each leaf quadrant maps to a single data page*
- *subdivide until points in each quadrant fit into one data page*
- *ideal: same number of points in each leaf quadrant (balanced)*
- *point density varies over space*

  ⇒ different regions require different levels of partitioning
- *this means that the tree is not necessarily balanced*

Note: effective for d≤5, ok for 6≤d≤10, ineffective for d>10

... Quad Trees

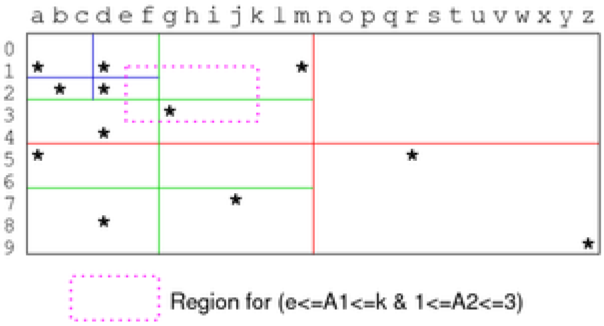
The previous partitioning gives this tree structure, e.g.



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

Searching in Quad-tree

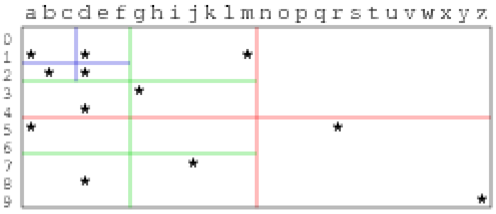
Space query example:


a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9
Region for (e<=A1<=k & 1<=A2<=3)

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

Exercise 11: Searching in Quad-trees

Using the following quad-tree index



Answer the queries `(m,1), (a,?), (?,1), (?,?)`

R-Trees

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

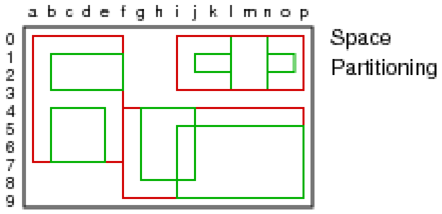
- *each node in the tree represents a kd hypercube*
- *its children represent (possibly overlapping) subregions*
- *the child regions do not need to cover the entire parent region*

Overlap and partial cover means:

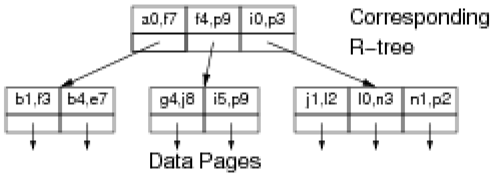
- *can optimize space partitioning wrt data distribution*
- *so that there are similar numbers of points in each region*

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

... R-Trees



Space Partitioning



Corresponding R-tree

Data Pages

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:

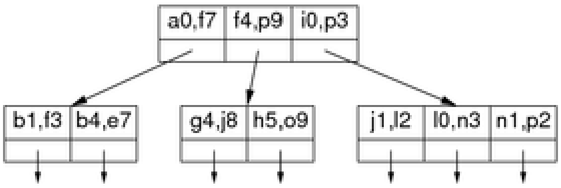
- *start at root, select all children whose subregions contain P*
- *if there are zero such regions, search finishes with P not found*
- *otherwise, recursively search within node for each subregion*
- *once we reach a leaf, we know that region contains P*

Space (region) queries are handled in a similar way

- *we traverse down any path that intersects the query region*

Exercise 12: Query with R-trees

Using the following R-tree:



Show how the following queries would be answered:

```
Q1: select * from Rel where X='a' and Y=4
Q2: select * from Rel where X='i' and Y=6
Q3: select * from Rel where 'c'≤X≤'j' and Y=5
Q4: select * from Rel where X='c'
```

Note: can view unknown value `X=?` as range min(`X`) ≤ `X` ≤ max(`X`)

Multi-d Trees in PostgreSQL

89/96

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: src/backend/access/gist

Costs of Search in Multi-d Trees

90/96

Difficult to determine cost precisely.

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

Similarity Retrieval

Similarity Selection

92/96

Relational selection is based on a boolean condition C

- evaluate C for each tuple t
- if C(t) is true, add t to result set
- if C(t) is false, t is not part of solution
- result is a set of tuples { t_1, t_2, ..., t_n } all of which satisfy C

Uses for relational selection:

- precise matching on structured data
- using individual attributes with known, exact values

... Similarity Selection

93/96

Similarity selection is used in contexts where

- cannot define a precise matching condition
- can define a measure d of "distance" between tuples
- d=0 is an exact match, d>0 is less accurate match
- result is a list of pairs [(t_1,d_1), (t_2,d_2), ..., (t_n,d_n)] (ordered by d_i)

Uses for similarity matching:

- text or multimedia (image/music) retrieval
- ranked queries in conventional databases

Similarity-based Retrieval

94/96

Similarity-based retrieval typically works as follows:

- *query is given as a query object q* *(e.g. sample image)*
- *system finds objects that are like q* *(i.e. small distance)*

The system can measure distance between any object and q ...

How to restrict solution set to only the "most similar" objects:

- *threshold d_{max} (only objects t such that $dist(t,q) \le d_{max}$)*
- *count k (k closest objects (k nearest neighbours))*

... Similarity-based Retrieval

Tuple structure for storing such data typically contains

- `id` *to uniquely identify object (e.g. PostgreSQL* `oid`*)*
- *metadata (e.g. artist, title, genre, date taken, ...)*
- `value` *of object itself (e.g. PostgreSQL BLOB or* `bytea`*)*

Properties of typical distance functions *(on objects x,y,z)*

- *$dist(x,y) \ge 0$, $dist(x,x) = 0$, $dist(x,y) = dist(y,x)$*
- *$dist(x,z) < dist(x,y) + dist(y,z)$* *(triangle inequality)*

Distance calculation often requires substantial computational effort

... Similarity-based Retrieval

Naive approach to similarity-based retrieval

```
q = ...     // query object
dmax = ... // dmax > 0   =>  using threshold
knn = ...   // knn > 0   =>  using nearest-neighbours
Dists = [] // empty list
foreach tuple t in R {
    d = dist(t.val, q)
    insert (t.oid,d) into Dists  // sorted on d
}
n = 0;  Results = []
foreach (i,d) in Dists {
    if (dmax > 0 && d > dmax) break;
    if (knn > 0 && ++n > knn) break;
    insert (i,d) into Results  // sorted on d
}
return Results;
```

Cost = read all r feature vectors + compute distance() for each