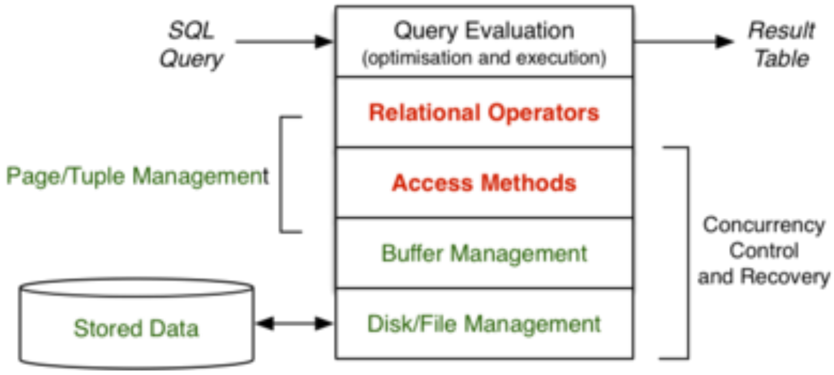
Week 06 Lectures

Implementing Relational Operators
1/76

Implementation of relational operations in DBMS:



... Implementing Relational Operators
2/76

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 ...

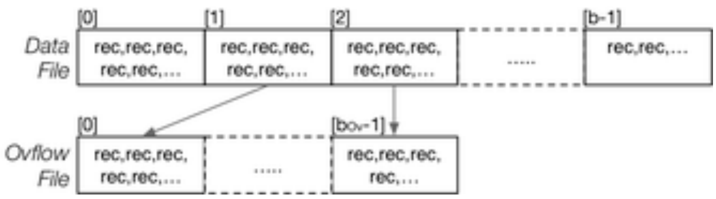
- *indexes* ... search trees based on pages/keys
- *signatures* ... bit-strings which "summarize" tuples

... Implementing Relational Operators
3/76

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

4/76

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

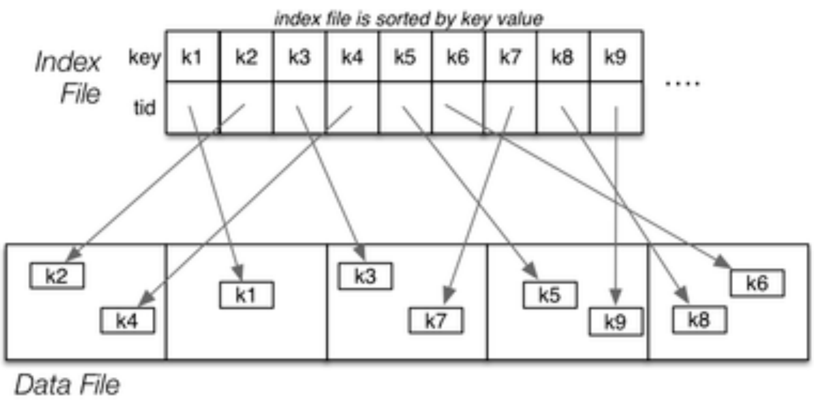
- each `request_page()` causes a read
- each `release_page()` causes a write (if page is dirty)

Indexing

Indexing

6/76

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



Indexes

7/76

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

8/76

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

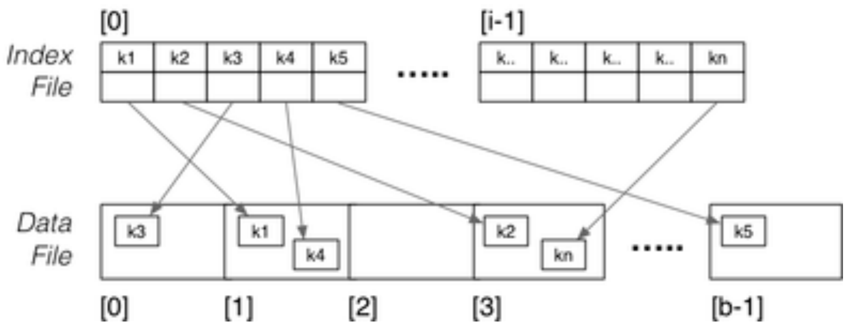
Index file has total *i* pages (where typically $i \ll b$)

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

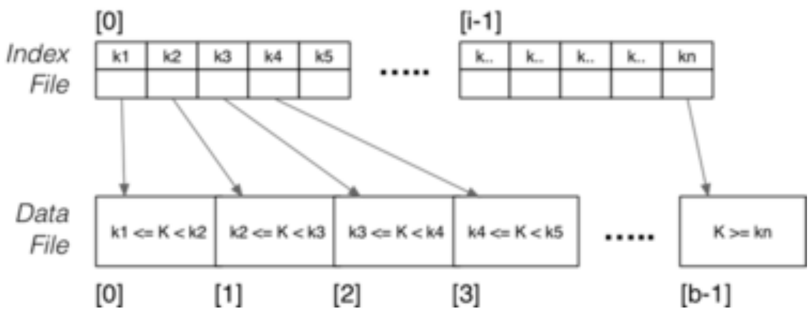
9/76



Data file unsorted; one index entry for each tuple

Sparse Primary Index

10/76



Data file sorted; one index entry for each page

Exercise 1: Index Storage Overheads

11/76

Consider a relation with the following storage parameters:

- $B = 8192, \quad R = 128, \quad r = 100000$
- 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

12/76

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

13/76

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

14/76

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 += 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 += T
}  }  }
```

Insertion with Primary Index

15/76

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

16/76

Overview:

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

If we delete index entries by marking ...

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

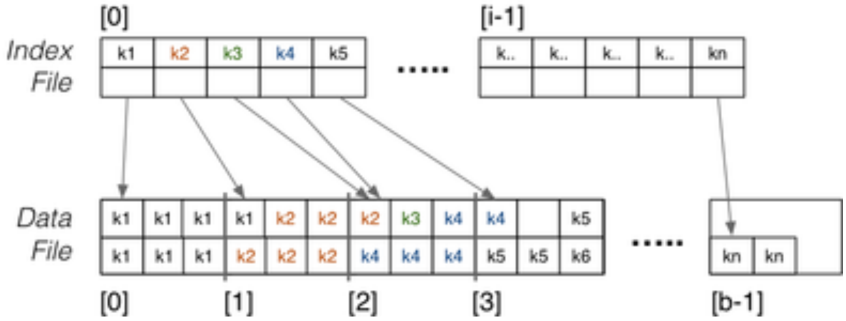
If we delete index entry by index file reorganisation ...

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

Clustering Index

17/76

Data file sorted; one index entry for each key value



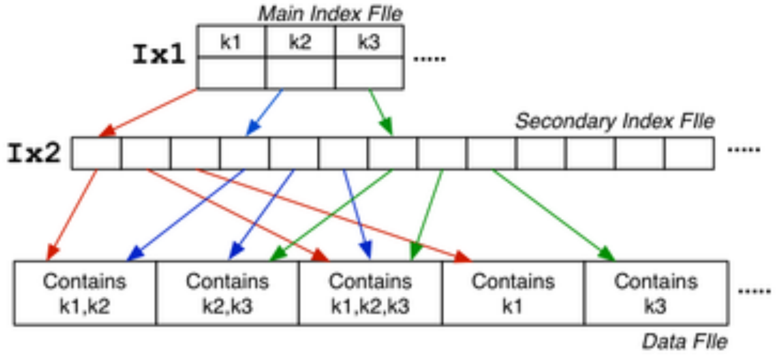
Cost penalty: maintaining both index and data file as sorted

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

Secondary Index

18/76

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

19/76

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

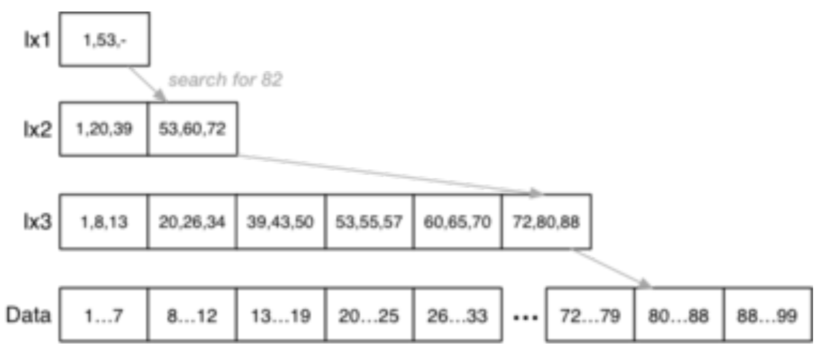
- making `Ix1` sparse, since `Ix2` is guaranteed to be ordered
- in this case, $b_{Ix1} = ceil(b_{Ix2} / c_i)$
- if `Ix1` becomes too large, add `Ix3` and make `Ix2` sparse
- if data file ordered on key, could make `Ix3` sparse

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

... Multi-level Indexes

20/76

Example data file with three-levels of index:



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

Select with Multi-level Index

21/76

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

22/76

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

23/76

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

24/76

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

25/76

Overview of the method:

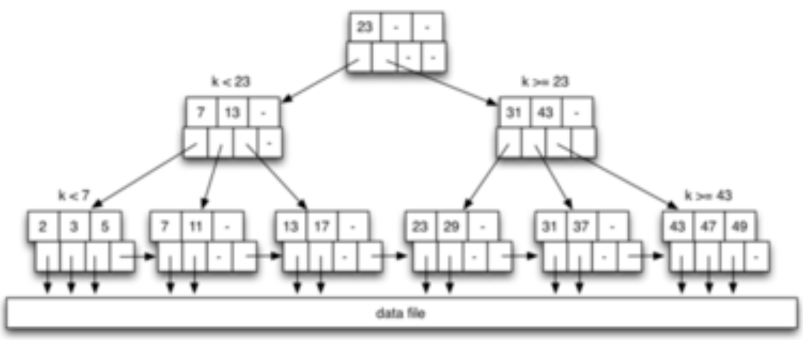
1. find leaf node and position in node where entry would be stored
2. if node is not full, insert entry into appropriate spot
3. if node is full, split node into two half-full nodes
   and promote middle element to parent
4. if parent full, split and promote upwards
5. if reach root, and root is full,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

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

Example: B-tree Insertion

26/76

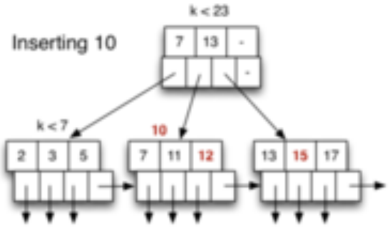
Starting from this tree:

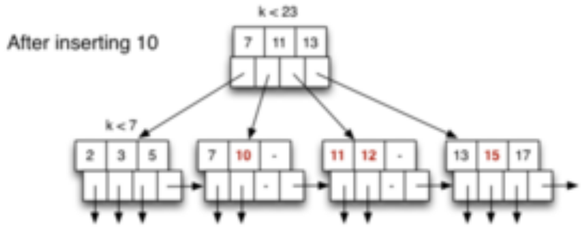


insert the following keys in the given order `12 15 30 10`

... 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

30/76

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

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

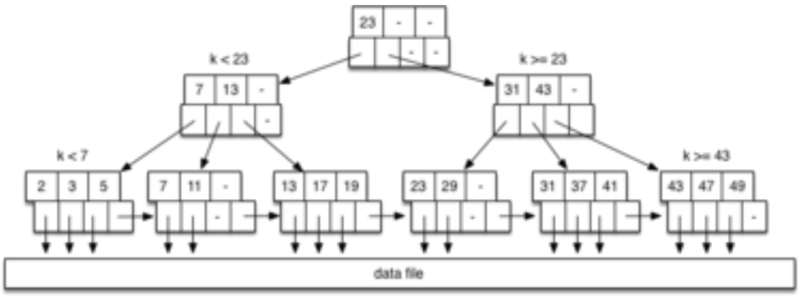
Selection with B-Trees

31/76

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
```

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



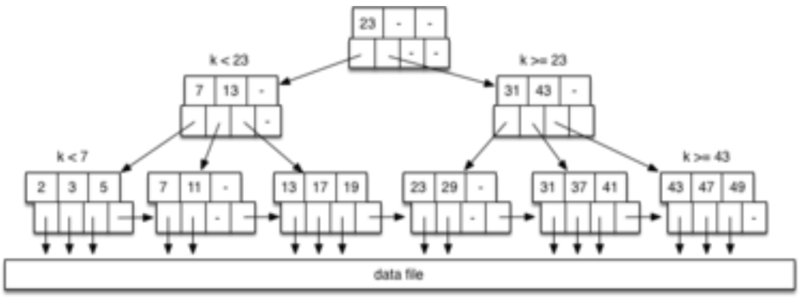
... Selection with B-Trees

32/76

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
}
```

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



33/76

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

34/76

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

36/76

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

37/76

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

38/76

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

39/76

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

40/76

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

41/76

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

```
select * from R
where  a1 op1 C1 and ... and an opn Cn
```

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

42/76

Implementing selection using *one of several* indices:

```
// Query: select * from R where a1op1C1 and ... and anopnCn
// choose ai with best selectivity
TupleIDs = IndexLookup(R,ai,opi,Ci)
// gives { tid1, tid2, ...} for tuples satisfying aiopiCi
PageIDs = { }
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }

// PageIDs = a set of bqix 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

43/76

Implementing selection using *multiple* indices:

```
// Query: select * from R where a1op1C1 and ... and anopnCn
// assumes an index on at least ai
TupleIDs = IndexLookup(R,a1,op1,C1)
foreach attribute ai with an index {
   tids = IndexLookup(R,ai,opi,Ci)
   TupleIDs = TupleIDs ∩ tids
}
PageIDs = { }
```

```
foreach tid in TupleIDs
   { PageIDs = PageIDs ∪ {pageOf(tid)} }
// PageIDs = a set of bq page numbers
...
```

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

Exercise 2: One vs Multiple Indices

44/76

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:

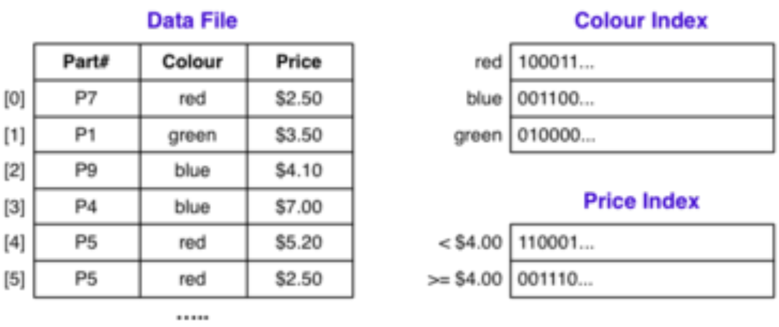
```
select * from Students
where  name='John' and birthday='04-01'
```

- 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

46/76

Alternative index structure, focussing on sets of tuples:



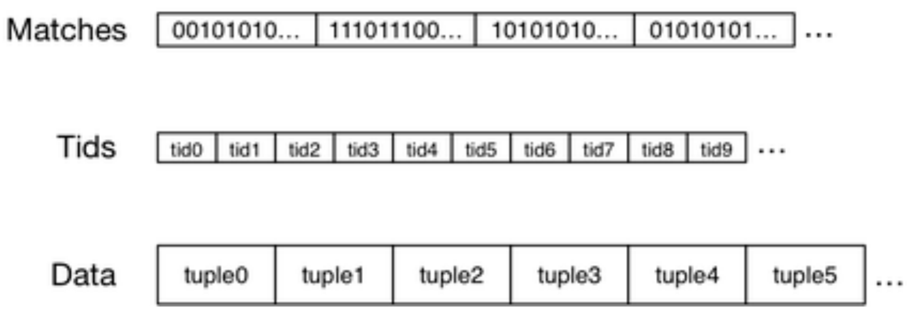
Index contains bit-strings of r bits, one for each value/range

---

... Bitmap Indexes

47/76

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



... Bitmap Indexes

48/76

Answering queries using bitmap index:

```
Matches = AllOnes(r)
foreach attribute A with index {
   // select ith 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 ∪ {pageOf(Tids[i])}
}
foreach pid in Pages {
   P = getPage(pid)
   extract matching tuples from P
}
```

Exercise 3: Bitmap Index

49/76

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

50/76

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*

52/76

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*

53/76

Multi-attribute hashing parameters:

- file size = $b = 2^d$ pages $\Rightarrow$ 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 ...
  bit j from $h_i(a_i)$ contributes bit k in combined hash value

MA.Hashing Example

54/76

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

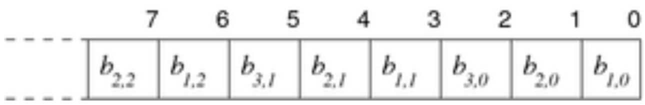
```
select * from Deposit where amount=533
```

Choice vector is designed taking expected queries into account.

... MA.Hashing Example

55/76

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)

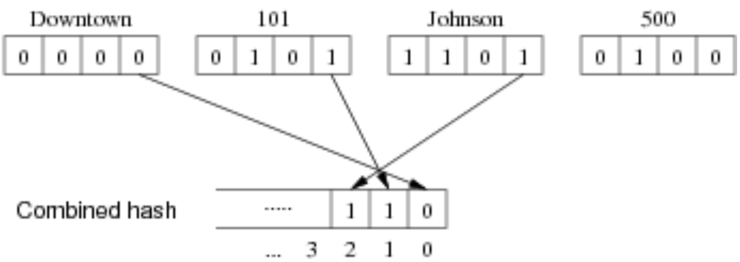
... MA.Hashing Example

56/76

Consider the tuple:

branch	acctNo	name	amount
Downtown	101	Johnston	512

Hash value (page address) is computed by:



MA.Hashing Hash Functions

57/76

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 hash1(Tuple t, int i) { ... }
```

... MA.Hashing Hash Functions

58/76

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] = hash1(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

59/76

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

60/76

In a partial match query:

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

E.g.

```
select amount
from   Deposit
```

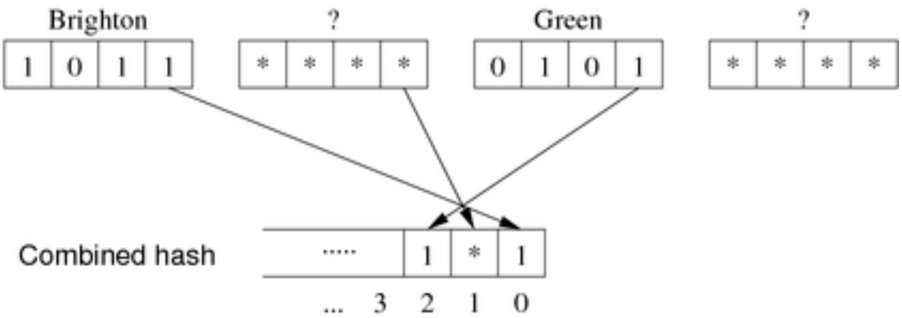
```
where  branch = 'Brighton' and name = 'Green'
```

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

... Queries with MA.Hashing

61/76

In composite hash for query, values for some bits are unknown:



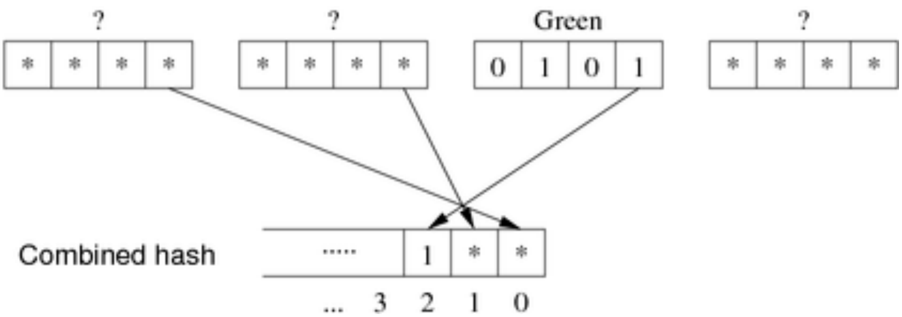
What this tells us: any matching tuples *must* be in pages `101`, `111`

... Queries with MA.Hashing

62/76

Consider the query:

```
select amount from Deposit where name = 'Green'
```



Need to check pages: `100`, `101`, `110`, `111`.

Exercise 5: Partial hash values in MAH

63/76

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

64/76

```
// 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

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

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)

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

... Query Cost for MA.Hashing

69/76

For a relation `R(a,b,c,d)` ...

```
select * from R where a=1
-- has 1 specified attribute (a)
-- has 3 unspecified attributes (b,c,d)

select * from R where b=5 and d=2
-- has 2 specified attributes (b,d)
-- has 2 unspecified attributes (a,c)

select * from R
where a=1 and b=5 and c=3 and d=2
-- has 4 specified attributes (a,b,c,d)
-- has 0 unspecified attributes
```

... Query Cost for MA.Hashing

70/76

Consider a query of type Q with m attributes unspecified.

Each unspecified A_i contributes d_i `*`'s.

Total number of `*`'s is $s = \Sigma_{i \notin Q} d_i$.

⇒ Number of pages to read is $2^s = \Pi_{i \notin Q} 2^{d_i}$.

Ignoring overflows, $Cost(Q) = 2^s$ (where s is determined by Q)

Including overflows, $Cost(Q) = 2^s(1+Ov)$

... Query Cost for MA.Hashing

71/76

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:

$Avg\ Cost_{pmr} = \Sigma_Q p_Q \Pi_{i \notin Q} 2^{d_i}$

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

---

Optimising MA.Hashing Cost

72/76

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 query type Q_j more efficient makes Q_k less efficient.

This is a combinatorial optimisation problem, and can be handled by standard optimisation techniques e.g. simulated annealing.

MA.Hashing Cost Example

73/76

Consider a table with four attributes:

(branch, account, name, amount) (abbreviated to *(br,ac,nm,amt)*)

Possible query types, and likelihood of each:

Query type	Cost	p_Q
(?, ?, ?, ?)	8	0
(*br*, ?, ?, ?)	4	0.25
(?, *ac*, ?, ?)	4	0
(*br*, *ac*, ?, ?)	2	0
(?, ?, *nm*, ?)	4	0
(*br*, ?, *nm*, ?)	2	0
(?, *ac*, *nm*, ?)	2	0.25
(*br*, *ac*, *nm*, ?)	1	0
(?, ?, ?, *amt*)	8	0
(*br*, ?, ?, *amt*)	4	0
(?, *ac*, ?, *amt*)	4	0
(*br*, *ac*, ?, *amt*)	2	0
(?, ?, *nm*, *amt*)	4	0
(*br*, ?, *nm*, *amt*)	2	0.5
(?, *ac*, *nm*, *amt*)	2	0
(*br*, *ac*, *nm*, *amt*)	1	0

Cost values are based on choice vector ($d_{br} = d_{ac} = d_{nm} = 1$)
p_Q values can be determined by observation of DB use.

... MA.Hashing Cost Example

74/76

Consider $r=10^6$, $N_r=100$, $b=10^4$, $d=14$.

Attribute *br* occurs in 0.5+0.25 used query types
⇒ allocate many bits to it e.g. $d_1=6$.

Attribute *nm* occurs in 0.5+0.25 of queries
⇒ allocate many bits to it e.g. $d_3=4$.

Attribute *amt* occurs in 0.5 of queries
⇒ allocate less bits to it e.g. $d_4=2$.

Attribute *ac* occurs in 0.25 of queries
⇒ allocate least bits to it e.g. $d_2=2$.

... MA.Hashing Cost Example

75/76

With bits distributed as: $d_1=6, d_2=2, d_3=4, d_4=2$

Query type	**Cost**	p_Q
(*br*, ?, ?, ?)	$2^8 = 256$	0.25
(?, *ac*, *nm*, ?)	$2^8 = 256$	0.25
(*br*, ?, *nm*, *amt*)	$2^2 = 4$	0.5

$Cost \ = \ 0.5 \times 2^2 + 0.25 \times 2^8 + 0.25 \times 2^8 \ = \ 130$

Exercise 8: MA.Hashing Design

76/76

Consider relation `Person(name,gender,age)` with $b=32$ and ...

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.

Produced: 10 Jul 2019