Chapter 2 Representation of DB-Models

Chapter 2.1 Model-Comparison

Assumption: availability of an appropriate APS

e.g. Cache-SS of UNIX
I-Node-SS of UNIX
Segment-SS

Tupel-SS: management of records of a DB

1. Relation (Codd) = Set of n-tuples
   n-tuple = structure = record
   ignore for the moment internal strucutre of a record, i. e. its attributes

2. CODASYL: chained sets of records of various type structures

3. oo DBS: extension (instances) of a class
   = set of records

⇒ general basic problem: with record-Id
insert find delete update record

4. Exception: Hierarchical Model
   e.g. IMS:
   - physical DB = ordered set of records
     (PDP-record, variable length)
   - record = hierarchy of fixed length segments
   - segment = combination of fixed length fields

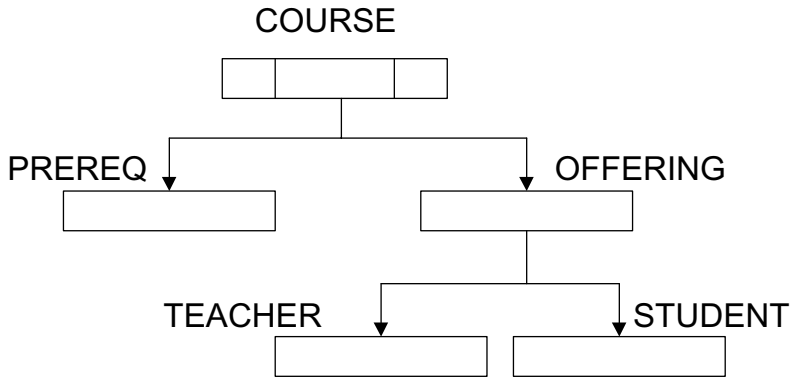
Smallest access unit via DL/I: segment
Segment in record is repeatable n-times

Special case IMS: records stored as physical units, segments within a record are ordered top-down, left-right

⇒ **Insert find delete update**
Are w.r. to IMS-Segments
= „record“ generally speaking
≠ segment of general segment-SS resp. GV

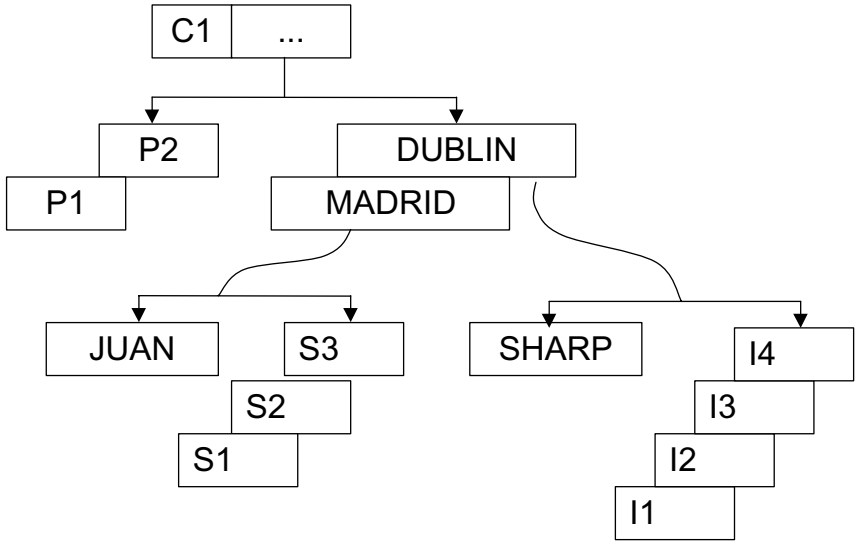
Recall w.r. to IMS: (see Date)

Record-Type:



regular espression: C(P*,(O(T,S*))*)

Corresponding object



Chapter 2.2 File Management

Management of records without internal attribute structure

Basic problems:

Identification
Storage in blocks, pages
Transport
only partially: Interpretation of internal record structure

Prerequisites resp. Postulates

1. Identifier: - unique
   - invariant during whole life
   e.g. external or internal primary key w.r. to relation
   often in addition: data base key
   TID : Tuple-Identifier

2. Random access to record, in addition sequentially

Importance of Invariance:

For stable, invariant relationships
Catalogue
in general auxiliary structures, e.g.
lock management
primary and secondary indexes
address books: {TID} → {block #}

⇒ HW-addresses of a record is not invariant, should be stored only in exactly 1 place, i.e. in address book.

Variant 1: only **external** primary key

X : set of keys
E : HW-address of records
Map : $\varepsilon : X \rightarrow E$

Implementation of ε via Hash-table, B-tree, etc., in general as address book

Access to record x only indirectly via ε (x), i.e. always evaluation of address book!

Variant 2: Problem: external primary key often very long, see DB-Schemata

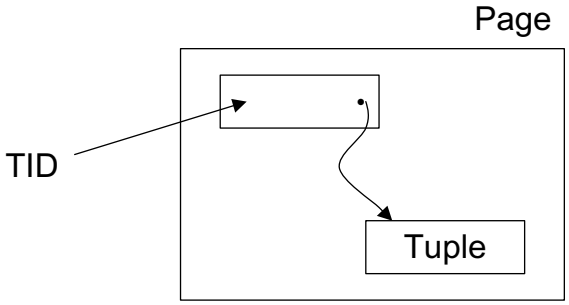
- external key X
- internal key I

Variant 2 a: TID-Method of System R

invariance by indirection
Tuple-Indentifier = TID = (S#, P#, W#)
Tuple-Address = cont (TID) = S#, P#, W#=
Tuple = cont (cont (TID))
Tuple-length: as extension of the record-address or at beginning of record

⇒ cont (TID) has fixed length, TID never has to be moved, therefore invariant.

Efficient solution:



Externalisation to other page is seldom!

Variant 2b: $I \subseteq N$

Free $i \in I$ assigned with **insert** of a record
Address-Books: for 2a) and 2 b):

$\iota : X \to I$
$\varepsilon : I \to E$
$\alpha : X \to E$
with $\alpha = \varepsilon \circ \iota$

Question: Should α be stored explicitely or computed dynamically??
Change management!!

Special considerations: for 2b) with reuse:

I is packed rather densely, if 1 free $i \in I$ is reused

ε represented as array, i. e. $\varepsilon(i) = e$
is stored in i-th array component
(similar to i-list), i. e. Trans Base Solution!

Variant 3: single usage of Identifiers

Advantages of single usage?

TID = (Segment #, page #, tuple #)
S# P# i
with: i is page-specific, running tuple #

new record on page: only if space available, page-specific high-water mark is increased

$\Rightarrow$ old space is reusable, without reusing old TIDs

Evaluation of Address-Book:

Real Adr. For TID = (S#, P#, i) is stored on (S#, P#) – page at place i

P_1#, i_1
P_1#, i_2
P_1#, i_3
} on page P_1

i_1, e_1
i_2, e_2
i_3, e_3

small sorted table

P_2#, i'_1
...
P_2#, i'_2
} on page P_2

Index- and Page-Organizations, Var. 3:

Required: $X \xrightarrow{\iota} TID \xrightarrow{\varepsilon} E$

$x\ \alpha\ (S\#, P\#, i)\ \alpha\ e$

with **cont** e = tuple

for total mapping one needs: (x, S#, P#, i, e)

Index	on page	is stored
$(x_1, S_1\#, P_1\#)$	P_1	(x_1, i_1, e_1)
$(x_2, S_2\#, P_2\#)$	P_2	(x_2, i_2, e_2)

often onlex 1 relation per segment, i. e. $S_1\# = S_2\#$, can be suppressed

$(x_1, P_1\#)$

$(x_2, P_2\#)$ ...

Access via external key:

ext-find (x):

1. Index-search with $x \rightarrow (S\#, P\#)$
2. fetch page (S#, P#)
3. find on P# entry (x, i, e)#
4. fetch record with real Adr. E

Note: i is not exploited

Access via internal key:

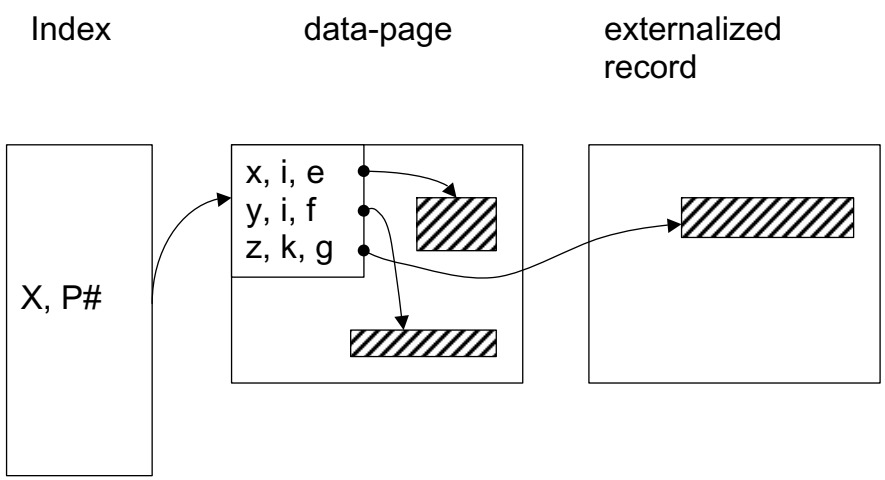
int-find (S#, P#, i):

1. %
2. fetch page (S#, P#)
3. find on P# via i entry (x, i, e)
4. as above

Advantage: Indexes become very small: (x, P#)

Disadvantage: Index does not deliver full TIDs, therefore socalled TID-algorithms for $\cap$, $\cup$, star-join are impossible

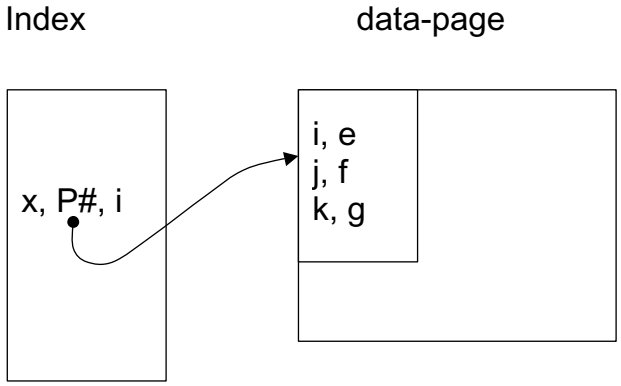
Graphical Representation:



$X \rightarrow E$	simple
$I \rightarrow E$	simple
$X \rightarrow I$	expensive

X α (S#, P#, i) can only be evaluated via page access

Alternative for TID-Algorithms:



$$X \xrightarrow{\iota} I \xrightarrow{\varepsilon} E$$

i, j, k, ... not dense, requires search table on page

X α (P#, i) can be evaluated via Index alone

$X \rightarrow I$	simple, only Index
$X \rightarrow E$	simple, Index and page
$I \rightarrow E$	simple, directly on page

Index-Organization requires in both cases:

Insert find delete maybe **next**

No **update** because of Invariance

Methods: B-Tree, Hash-Tab., ect.

Preview: TID-Algorithms

secondary indexes $A \rightarrow 2^I$
via attributes $B \rightarrow 2^I$
A, B

this allows queries:

select A, B, C from R
where $A = a \wedge B = b \wedge C \leq c$

let TID_a := set of internal Identifiers for tuple with A = a
TID_b := set of internal Identifiers for tuple with B = b

Fetch record with $t \in TID_a \cap TID_b$ and check:
$tuple(t).C \leq c$
or even:
fetch record with $t \in TID_a \cap TID_b \cap TID \leq C$

Chapter 2.3 Usage of Indexes

Set of keys X with order <
e.g. X = strings over alphabet
< is lexicogr. Order
or X = |N with <
or X = date = (year, month, day, time)
< componentwise and „lexecographically“

Set of associated data A
$a \in A$

Index $\sum = \{(x, a) \mid x \in X \wedge a \in A\}$
primary index: x is indentifier for a
secondary index: several pairs (x, a), (x, b) are allowed.

Basic Operations
insert (y, b)
find (z)
delete (z)
update (y, c)
point (z): position {for intervals}
next
reset
EOF

B-Tree: Implementation of this datatype with

1. next, delete possible
2. all operations with complexity

   log (|Σ|)

   and linear storage requirement
3. applicability for peripheral stores :
   Trees remain very flat
4. Good storage exploidation: > 50%

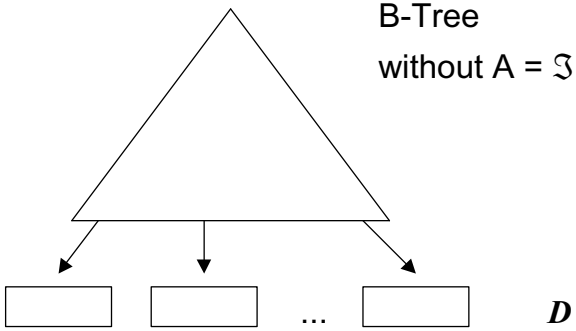
B*-Tree: for $\iota : X \to I$

resp. $\varepsilon : X \to E$

resp. $\iota_1 : X \to \{\text{page \#}\}$

Index part
< 1%

B-Tree
without A = ℑ



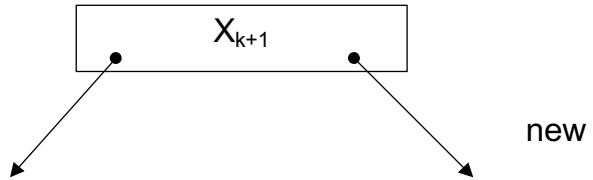
sequ.
File

D

1. All associated Info on leaves (x, e), (x, i), (x, P#) etc.
2. For y ∈ X use ℑ only to steer search, i.e. to find leaf in ***D*** on which (y, e) must be stored.

⇒ upper levels of ℑ are cached in AS; cache all of ℑ on disk for CD-ROM databases, < 6 MB:

Split of a Leaf



new

$(x_1, e_1), (x_2, e_2), ..., (x_k, e_k), (x_{k+1}, e_{k+1}) ... (x_{2k+1}, e_{2k+1})$

tuple $\tau(x_2)$ = record outside of the tree or on same leaf-page

Simple Prefix-B-Tree:

Separator: instead of x_{k+1}
Use s with
$X_k < s \leq x_{k+1}$

Goal: shortest possible s

$\Rightarrow$ larger branching degree in $\Im$, $\Im$ smaller and flatter.

Prefix Property: for lexicogr. <

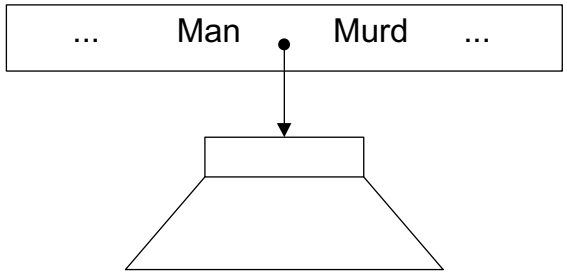
Let x, y be strings with x < y.

Then there exists unique prefix $\bar{y}$ of y so that:

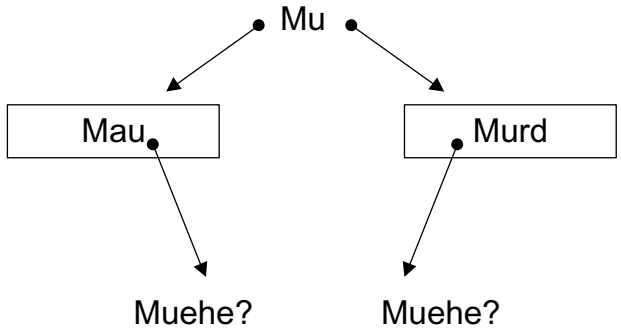
a) $\bar{y}$ is Separator between x and y
   $x < \bar{y} \leq y$

b) no other Separator is shorter than $\bar{y}$

Precaution: Separators work only when splitting leaves,
i.e. for transition from $\boldsymbol{D}$ to $\Im$.



Pulling up the Separator Mu does not work:



$\Rightarrow$ treat $\Im$ like standard B-Tree, but without associated info.

Def.: Simple Prefix-B-tree is a B*-Tree for which the B*-Index-part is replaced by a B-Tree of not always shortest separators (of variable length).

Note: B*-Trees $\subset$ Simple Prefix-B-Trees

Search Alg.: almost exactly as for B-Tree

Split-Intervals:

δ_l: „Intervals“ around the middle of a leaf within δ_l search split point x_{i-1}, x_i so that Separator s_i $x_{i-1} < s_i \leq x_i$ becomes as short as possible.

δ_b: similar interval around middle of node of $\mathfrak{I}$

$\Rightarrow$ short separator for father node, i.e. higher degree of branching near root of $\mathfrak{I}$.

Effect of δ_l and δ_b:

δ_l : short separators in $\mathfrak{I}$

δ_b : shorter separators near root

$\Rightarrow$ high branching degrees in $\mathfrak{I}$

$\Rightarrow$ especially near roos of $\mathfrak{I}$

$\Rightarrow$ $\mathfrak{I}$ smaller (caching better)

$\Rightarrow$ Tree flatter, i. e. fewer disk accesses

Experience: small intervalls bring already large performance advantages

Algorithms:

find: like for B*-Tree

insert: additionally determine shortest separator within split interval

delete: (x,e) is always deleted from leaf, concatenation normal

Overflow between leaves: determine new separator.

Chapter 2.4 Genuine Prefix-B-Trees

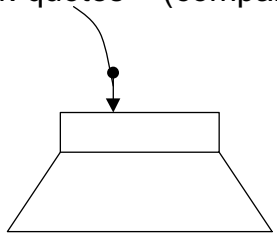
For multiattribute Indexes, e.g. for time series, Data-Warehouse applications in which the attribute sequence implies „lexikographic“ order,

e.g.

(year, month, day, hour, min) = timepoint

measurement-sequence = (measurepoint, timepoint, value)

stock-quotes = (company, timepoint, DM, turnover)



T(P): subtree With root P

Tree structure, path to P, determines:

greatest lower bound: λ (P)

smallest upper bound: μ (P)

$\forall x \in T(P) : \lambda(P) \leq x < \mu(P)$

$\forall s \in T(P) : \lambda(P) \leq s < \mu(P)$

x key, s Separator

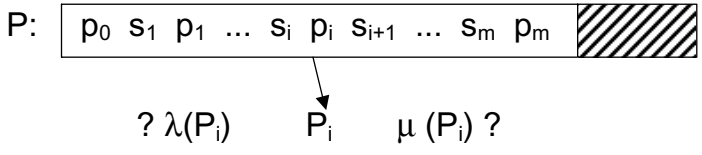
l_0 : smallest letter of Alphabet

∞ : $\infty > l_i \ \forall$ letters l_i

for root R of the whole tree:

λ (R) = l_0; μ (R) = ∞

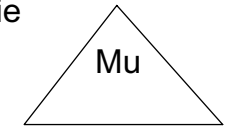
Iterative Determination of λ (P) and μ (P):



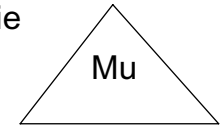
$$\lambda(P_i) = \begin{cases} s_i \text{ for } i = 1, 2, ..., m \\ \lambda(P) \text{ for } i = 0 \end{cases}$$

$$\mu(P_i) = \begin{cases} s_{i+1} \text{ for } i = 0, 1, ..., m-1 \\ \mu(P) \text{ for } i = m \end{cases}$$

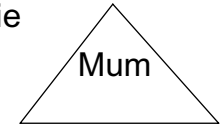
$\Rightarrow$ all Separators or keys in T (P) have common prefix κ (P) which can be determined from λ (P) and μ (P). (mostly as longest common prefix of λ (P), μ (P) except for case 3 in example)

case 1: λ (P) = Mumie 

μ (P) = Murnau

case 2: Mumie 

Mund

case 3: Mumie 

Mun

Compute λ (P), μ (P) and κ (P) with tree search, κ (P) is split off in T (P), i. e. need not be stored.

s = κ (P) $\hat{s}$

Store only $\hat{s}$, partial Separator.

On leaves store κ (P) once and $\hat{s}$, this makes sequential processing of $\boldsymbol{D}$ possible without $\Im$ and error repairs.

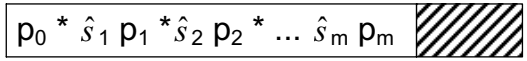
$\Rightarrow$ resulting trees are **genuine** Prefix-B-Trees

For mulitattribute-keys:

e.g. Timepoint = (year, month, day, time)

δ_l: leads to shortest partial separators, e.g. change of month or day.

Page Organization



Special character * for binary search

Performance Analysis:

decisive: length of separators or partial separators resp.

Parameter:

α: cardinality of the key-alphabet

n: cardinality of the file

Conclusion:

a) simple Prefix-B-tree much better than B*-tree.

b) genuine Prefix-B-trees are not very effective for one-attribute string-keys.

c) genuine Prefix-B-trees excellent for multiattribute keys

Additional Problem: Tradeoff between storage accesses and increased CPU-load