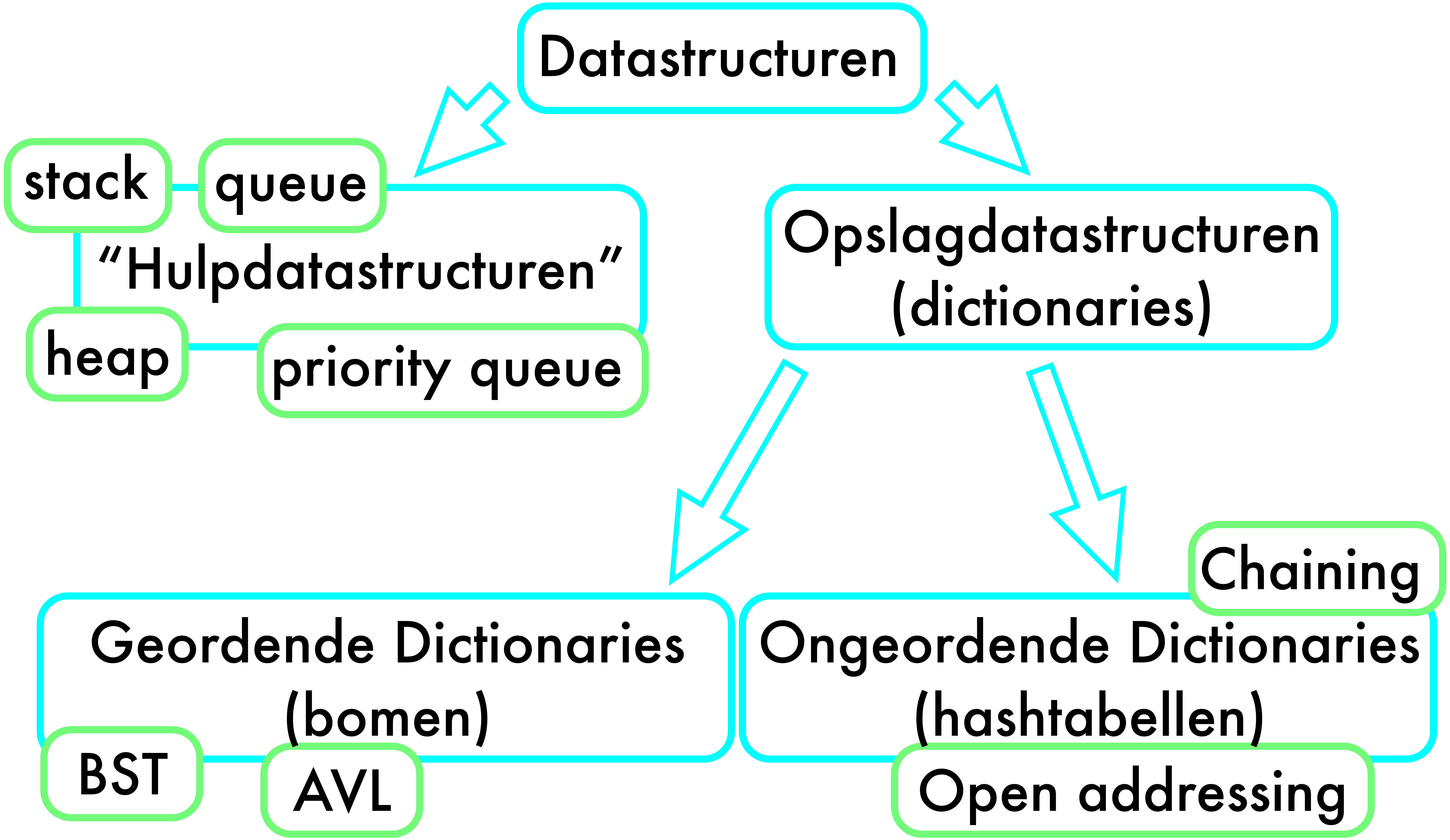
# Hoofdstuk 16

## Externe Data-opslag

# Cursus AD t.e.m. Hoofdstuk 7



# Geordende Dictionaries op Disk

Basis voor de implementatie van SQL

insert

delete

find

set-current-to-first,
set-current-to-next

sneller indien geordende data

Equality search

```
SELECT FROM dict WHERE attribute=value

SELECT FROM dict WHERE attribute>value

SELECT FROM dict WHERE attribute<value
```

Range search

steunt op orde + sneller indien geordende data

# Het Doel #1: Tabelschema's

verzameling tabellen

```
(define zonnestelsel (db:new my-disk "Zonnestelsel"))
```

schema

attributen/kolommen

```
(define manenschema  '((string 9)
                       (string 9)
                       (natural 2)
                       (natural 2)
                       (string 10)))

(define manen (db:create-table zonnestelsel "Manen" manenschema))
```

```
(define :maan-naam:        0)
(define :planeet:          1)
(define :maan-middellijn:  2)
(define :ontdekjaar:       3)
(define :ontdekker:        4)
```

tabel

key

foreign key

| maan-naam | planeet | maan-middellijn | ontdekjaar | ontdekker |
|---|---|---|---|---|
| | | | | |
| | | | | |
| | | | | |
| | | | | |

veld

tuple/rij/record/waarden

| planeet-naam | afstand-tot-zon | aard-massa | middellijn | omlooptijd | rotatie-tijd |
|---|---|---|---|---|---|
| | | | | | |
| | | | | | |
| | | | | | |
| | | | | | |

```
(define planetenschema '((string 9)
                         (decimal)
                         (decimal)
                         (natural 3)
                         (decimal)
                         (decimal)))
(define planeten (db:create-table zonnestelsel "Planeten" planetenschema))
```

```
(define :planeet-naam:     0)
(define :afstand-tot-zon:  1)
(define :aard-massa:       2)
(define :middellijn:       3)
(define :omlooptijd:       4)
(define :rotatietijd:      5)
```

# Het Doel #2: Tabelpopulatie

```
(db:insert-into-table! zonnestelsel planeten
                       (list "Mercurius" 0.3871   0.053   4840   0.241  +58.79))
(db:insert-into-table! zonnestelsel planeten
                       (list "Venus"     0.7233   0.815  12200   0.615 -243.68))
(db:insert-into-table! zonnestelsel planeten
                       (list "Aarde"     1.0000   1.000  12756   1.000   +1.00))
(db:insert-into-table! zonnestelsel planeten
                       (list "Mars"      1.5237   0.109   6790   1.881   +1.03))
(db:insert-into-table! zonnestelsel planeten
                       (list "Jupiter"   5.2028 317.900 142800  11.862   +0.41))
(db:insert-into-table! zonnestelsel planeten
                       (list "Saturnus"  9.5388  95.100 119300  29.458   +0.43))
(db:insert-into-table! zonnestelsel planeten
                       (list "Uranus"   19.1819  14.500  47100  84.013   -0.45))
(db:insert-into-table! zonnestelsel planeten
                       (list "Neptunus" 30.0578  17.500  44800 164.793   +0.63))
(db:insert-into-table! zonnestelsel planeten
                       (list "Pluto"    39.2975   1.000   5000 248.430   +0.26))
```

Values (aka Tupels (aka Records))

# Het doel #3: Tupellocalisering

```
(db:delete-where! zonnestelsel manen    :planeet:      "Aarde")
(db:delete-where! zonnestelsel planeten :planeet-naam: "Neptunus")
(db:delete-where! zonnestelsel planeten :planeet-naam: "Uranus")
(db:delete-where! zonnestelsel planeten :planeet-naam: "Pluto")
(db:delete-where! zonnestelsel planeten :omlooptijd:   1.881)
(db:delete-where! zonnestelsel planeten :omlooptijd:   11.862)
(db:delete-where! zonnestelsel planeten :omlooptijd:   29.458)
...
```

Equality search

```
(db:select-from/eq zonnestelsel  manen :ontdekker: "Cassini"))
```

# Doel: 2 Belangrijke Aspecten

Opslag van tupels in datafiles (hoofdstuk 16)

zo snel mogelijk maken

```
(define manen (db:create-table zonnestelsel "Manen" manenschema))

(db:insert-into-table! zonnestelsel manen (list "Maan" "Aarde" 3476 1877 ""))
```

Zorgen voor bewegwijzering voor snelle toegang (hoofdstuk 17)

```
(db:create-index! zonnestelsel planeten "Naam-IDX"   :planeet-naam:)
(db:create-index! zonnestelsel planeten "Omloop-IDX" :omlooptijd:)
```

```
(db:select-from/eq zonnestelsel  manen :ontdekker: "Cassini"))
```

zo snel mogelijk maken

Equality search

```
(db:delete-where! zonnestelsel manen :planeet: "Aarde")
```

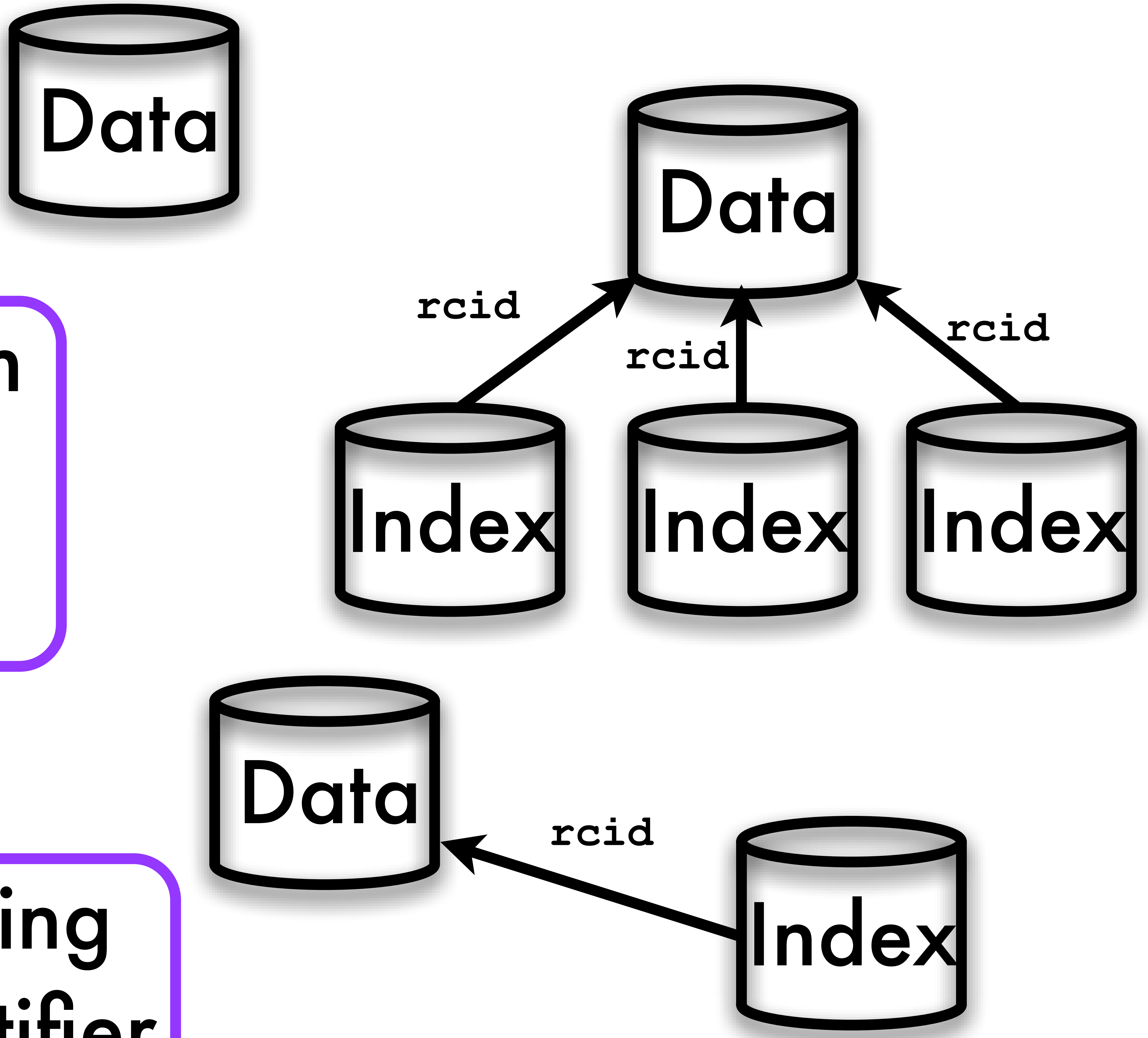
# Databank

```
ADT database

new
   ( disk string → database )
delete!
   ( database → ∅ )
create-table
   ( database string pair → table )
create-index!
   ( database table string number → ∅ )
drop-table!
   ( database table → ∅ )
insert-into-table!
   ( database table pair → ∅ )
delete-where!
   ( database table number any → ∅ )
select-from/eq
   ( database table number any → pair )
```

volledige implementatie
in H17

# Databank = { Files }

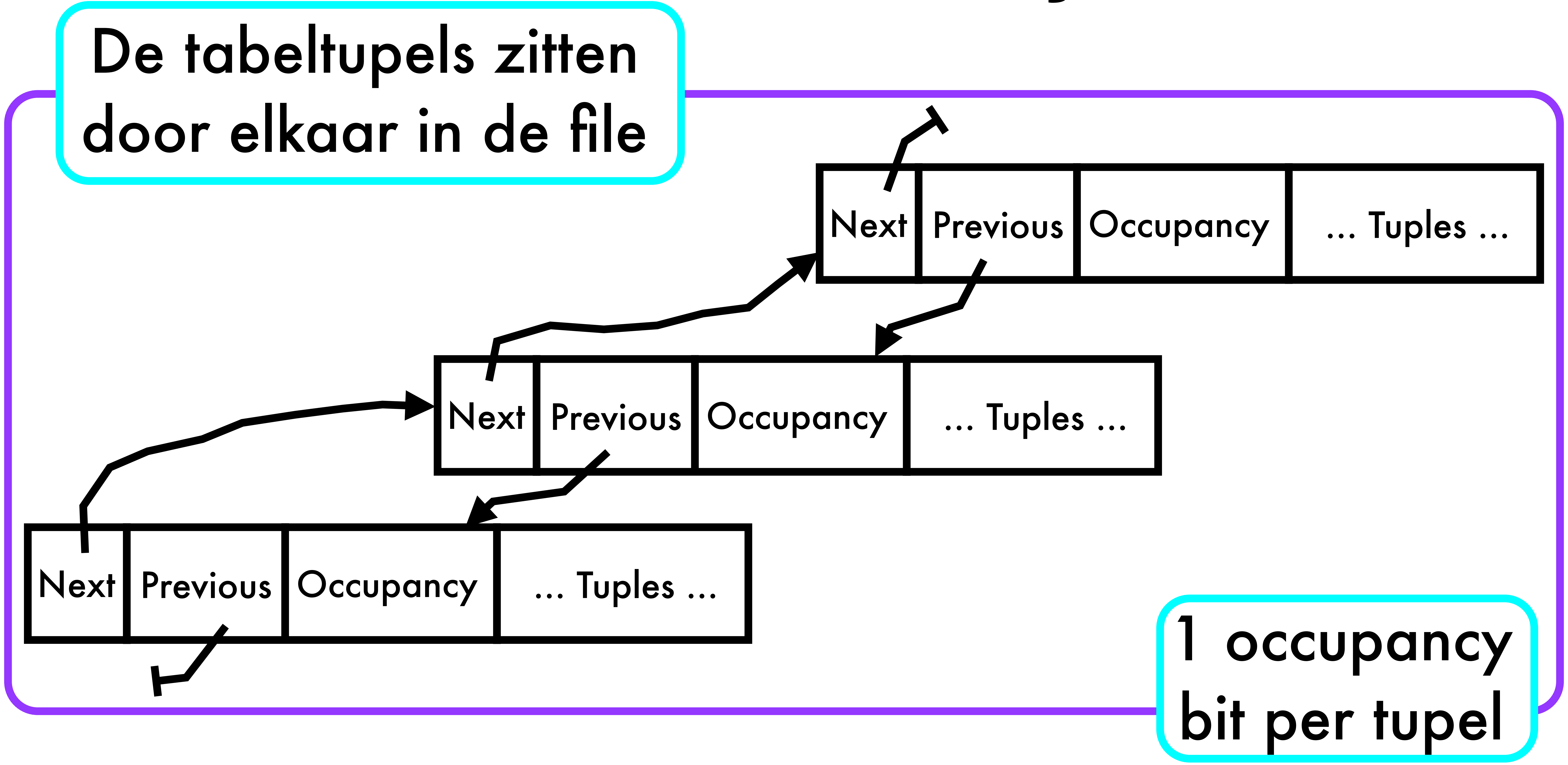
<u>Datafiles</u> bevatten de eigenlijke tupels




<u>Indexfiles</u> dienen om tupels uit datafiles snel terug te vinden

Tupelverwijzing = <u>record identifier</u>

# Datafile = Dubbel Gelinkte Lijst




```
(define next-offset 0)
(define prev-offset (+ next-offset disk:block-pointer-size))
(define bits-offset (+ prev-offset disk:block-pointer-size))
```

# Record Identifiers

In Scheme:

| bptr | slot |
|---|---|

tupel 0, tupel 1, ...

```
(define (new bptr slot)
  (cons bptr slot))

(define bptr car)

(define slot cdr)

(define (rcid->fixed rcid)
  (+ (* (expt 256 disk:block-idx-size) (bptr rcid)) (slot rcid)))

(define (fixed->rcid num)
  (define radx (expt 256 disk:block-idx-size))
  (new (quotient num radx) (modulo num radx)))

(define size (+ disk:block-ptr-size disk:block-idx-size))

(define null (new fs:null-block 0))

(define (null? rcid)
  (and (fs:null-block? (bptr rcid))
       (= (slot rcid) 0)))
```

(H14) aantal bytes om offset in block voor te stellen

**Naar een tupel wordt <u>verwezen</u> via zijn bloknummer en zijn slotnummer in dat blok**

Op disk:

`bptr * 256`$^{\texttt{block-idx-size}}$` + slot`

size

| bptr | slot |
|---|---|
| block-ptr-size bytes | block-idx-size bytes |

**Corollarium: records niet verplaatsen!**

# Een Tabelschema

```
ADT schema

new
  ( disk pair → schema )
schema?
  ( any → boolean )
open
  ( disk number → schema )
delete!
  ( schema → ∅ )
nr-of-occupancy-bytes
  ( schema → number )
nr-of-attributes
  ( schema → byte )
record-size
  ( schema → number )
type
  ( schema number → byte )
size
  ( schema number → byte )
capacity
  ( schema → number )
disk
  ( schema → disk )
position
  ( schema → number )
```

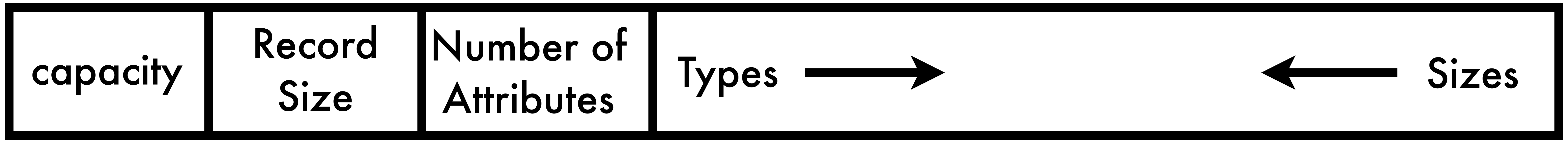
```
(define manenschema  '((string 9)   ; naam maan
                        (string 9)   ; naam planeet
                        (natural 2)  ; middellijn
                        (natural 2)  ; ontdekjaar
                        (string 10))); ontdekker
```

```
> (define d (disk:new "Harddisk"))
> (fs:format! d)
> (define scma (new d manenschema))
> (position scma)
1
> (disk scma)
#<disk>
> (capacity scma)
15
> (type scma 4)
3
> (size scma 4)
10
> (nr-of-attributes scma)
5
> (nr-of-occupancy-bytes scma)
2
> (record-size scma)
32
```

# Schema ≈ 1 blok

geen 'VARCHAR' bvb.

**Assumptie: Alle tupels zijn even groot**




```
(define schema-nr-size 2)
```

```
(define cap-offset         0)
(define rsz-offset         (+ cap-offset schema-nr-size))
(define schema-size-offset (+ rsz-offset schema-nr-size))
(define schema-info-offset (+ schema-size-offset 1))
```

```
(define manenschema  '((string 9)
                       (string 9)
                       (natural 2)
                       (natural 2)
                       (string 10)))
```

**Types**

```
(define natural-tag 0)
(define integer-tag 1)
(define decimal-tag 2)
(define string-tag  3)
```

# Encoding & Decoding

```
(define (capacity! scma nmbr)
  (define blck (block scma))
  (disk:encode-fixed-natural! blck cap-offset schema-nr-size nmbr))
(define (capacity scma)
  ...)
(define (record-size! scma nmbr)
  ...)
(define (record-size scma)
  ...)
(define (nr-of-attributes scma)
  ...)
(define (nr-of-attributes! scma nmbr)
  ...)
(define (type! scma indx type)
  ...)
(define (type scma indx)
  ...)
(define (size! scma indx nmbr)
  ...)
(define (size scma indx)
  ...)
```

# Creatie van een Schema

```
(define-record-type schema
  (make b)
  schema?
  (b block))
```

```
(define (new dsk atts)
  (define rsz (sum-of-sizes atts))
  (define cap (quotient (* (- disk:block-size fixed-header-size) 8)
                        (+ (* rsz 8) 1)))
  (if (< cap 1)
    (error "tuples must fit in a block (new schema)" rsz))
  (let* ((blck (fs:new-block dsk))
         (scma (make blck)))
    (capacity!          scma cap)
    (record-size!       scma rsz)
    (nr-of-attributes! scma (length atts))
    (types/sizes!       scma atts)
    (disk:write-block! blck)
    scma))
```

Hulpstukken op volgende slides

```
(define fixed-header-size (* 2 disk:block-pointer-size))
```

prev+next in nodes

# Grootte van één Record

```
(define (field-size fild)
  (cond ((eq? (car fild) 'natural)
         (cadr fild))
        ((eq? (car fild) 'integer)
         (cadr fild))
        ((eq? (car fild) 'decimal)
         disk:real64)
        ((eq? (car fild) 'string)
         (cadr fild))
        (else (error "unsported field type" (car fild)))))
```

```
(string 9)
(natural 2)
(string 10)
(decimal)
```

Som van de groottes van de velden

```
(define (sum-of-sizes atts)
  (let loop
    ((atts atts)
     (sizs 0))
    (if (null? atts)
        sizs
        (loop (cdr atts) (+ (field-size (car atts)) sizs)))))
```

# Opvulling van het Schema

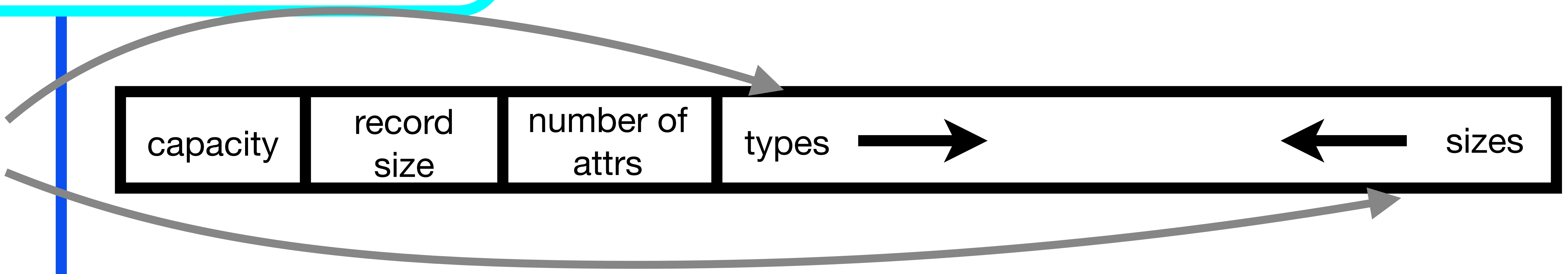
```
(define (field-size fild)
  (cond ((eq? (car fild) 'natural)
```

```
(define (field-type fild)
  (cond ((eq? (car fild) 'natural)
         natural-tag)
        ((eq? (car fild) 'integer)
         integer-tag)
        ((eq? (car fild) 'decimal)
         decimal-tag)
        ((eq? (car fild) 'string)
         string-tag)
        (else (error "unsported field type" (car fild)))))
```

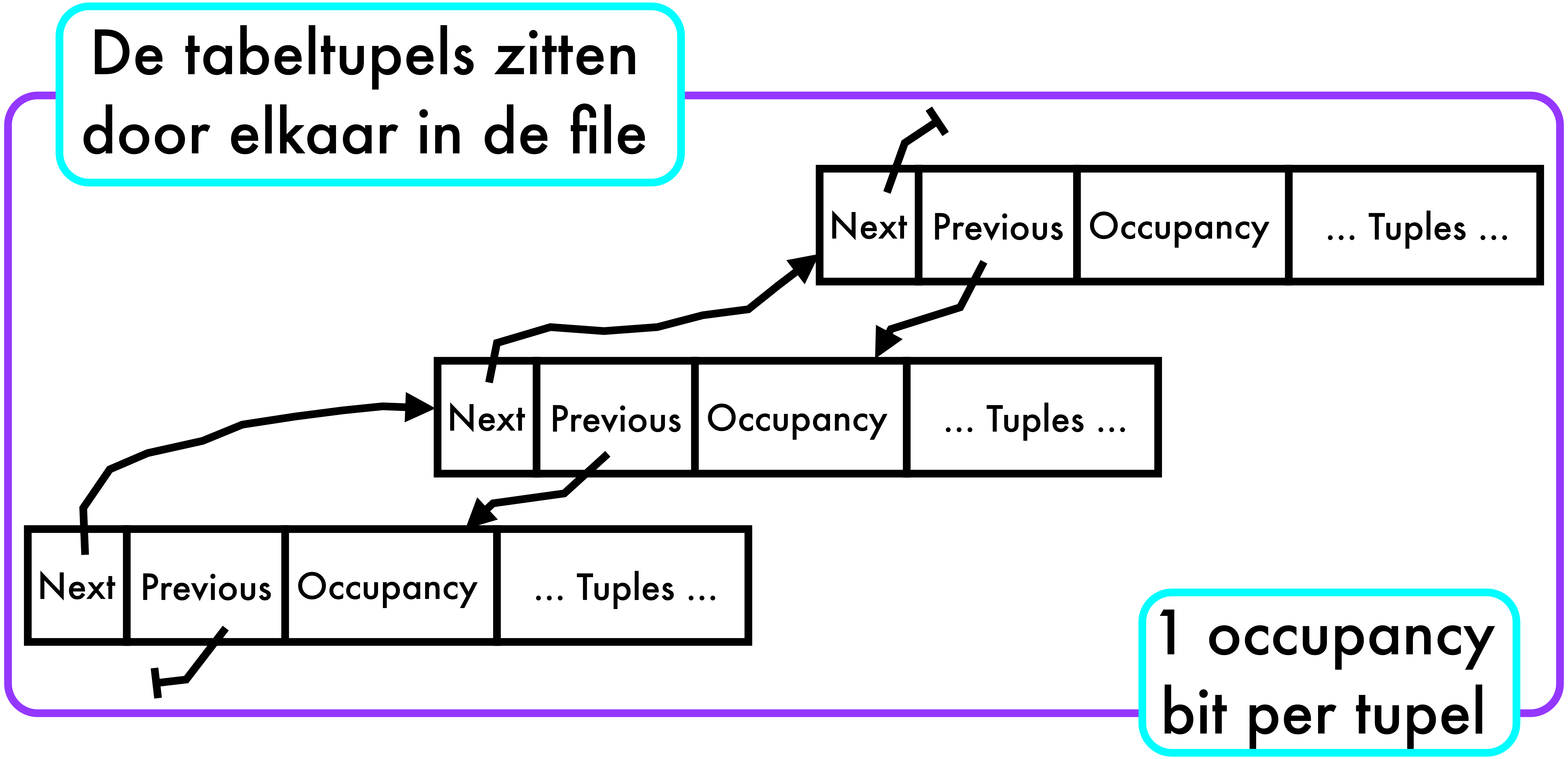
```
(string 9)
(natural 2)
(string 10)
(decimal)
```

```
(define (types/sizes! scma atts)
  (let loop
    ((indx 0)
     (atts atts))
    (type! scma indx (field-type (car atts)))
    (size! scma indx (field-size (car atts)))
    (if (not (null? (cdr atts)))
        (loop (+ indx 1) (cdr atts)))))
```

Coderen van de schema-informatie



# Datafile = Dubbel Gelinkte Lijst




```
(define next-offset 0)
(define prev-offset (+ next-offset disk:block-pointer-size))
(define bits-offset (+ prev-offset disk:block-pointer-size))
```

# Datafile Nodes

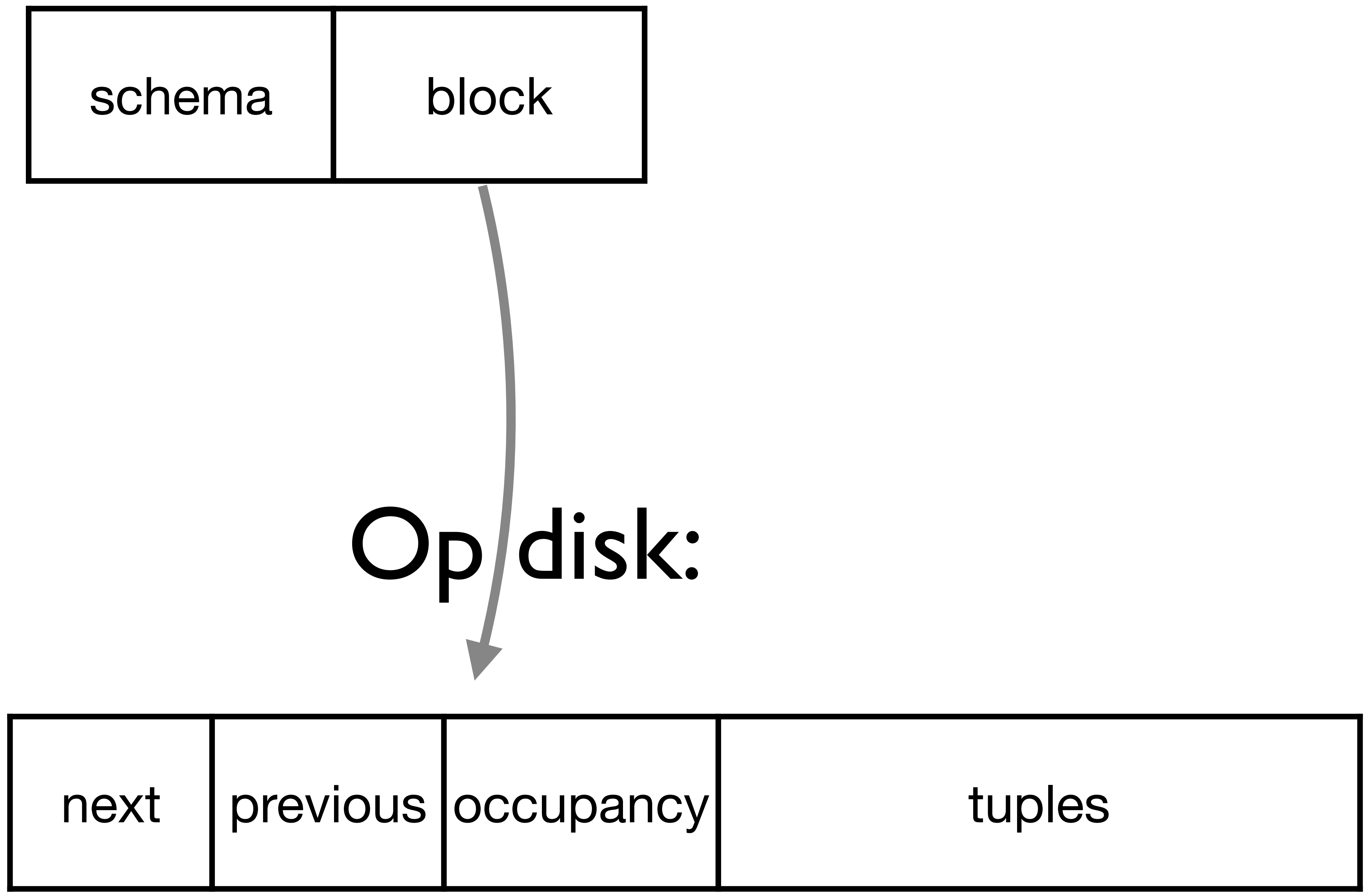
Abstractielaag boven blocks

```
ADT node

new
  ( schema node → node )
node?
  ( any → boolean )
delete!
  ( node → ∅ )
read
  ( schema number → node )
write!
  ( node → ∅ )
schema
  ( node → schema )
position
  ( node → number )
```

```
record!
  ( node number pair → ∅ )
record
  ( node number → pair )
clear-slot!
  ( node number → ∅ )
slot-occupied?
  ( node number → boolean )
all-free?
  ( node → boolean )
all-occupied?
  ( node → boolean )
next
  ( node → number )
next!
  ( node number → ∅ )
previous
  ( node → number )
previous!
  ( node number → ∅ )
```

Ieder tupel heeft een slotnummer

In Scheme:



# Node Representatie

```
(define-record-type node
  (make s b)
  node?
  (s schema)
  (b block))
```

```
(define (next! node next)
  ...)
(define (next node)
  ...)

(define (previous! node prev)
  ...)
(define (previous node)
  ...)

(define (occupancy-bits! node bits)
  ...)
(define (occupancy-bits node)
  ...)
```

| next | previous | occupancy | tuples |
|---|---|---|---|

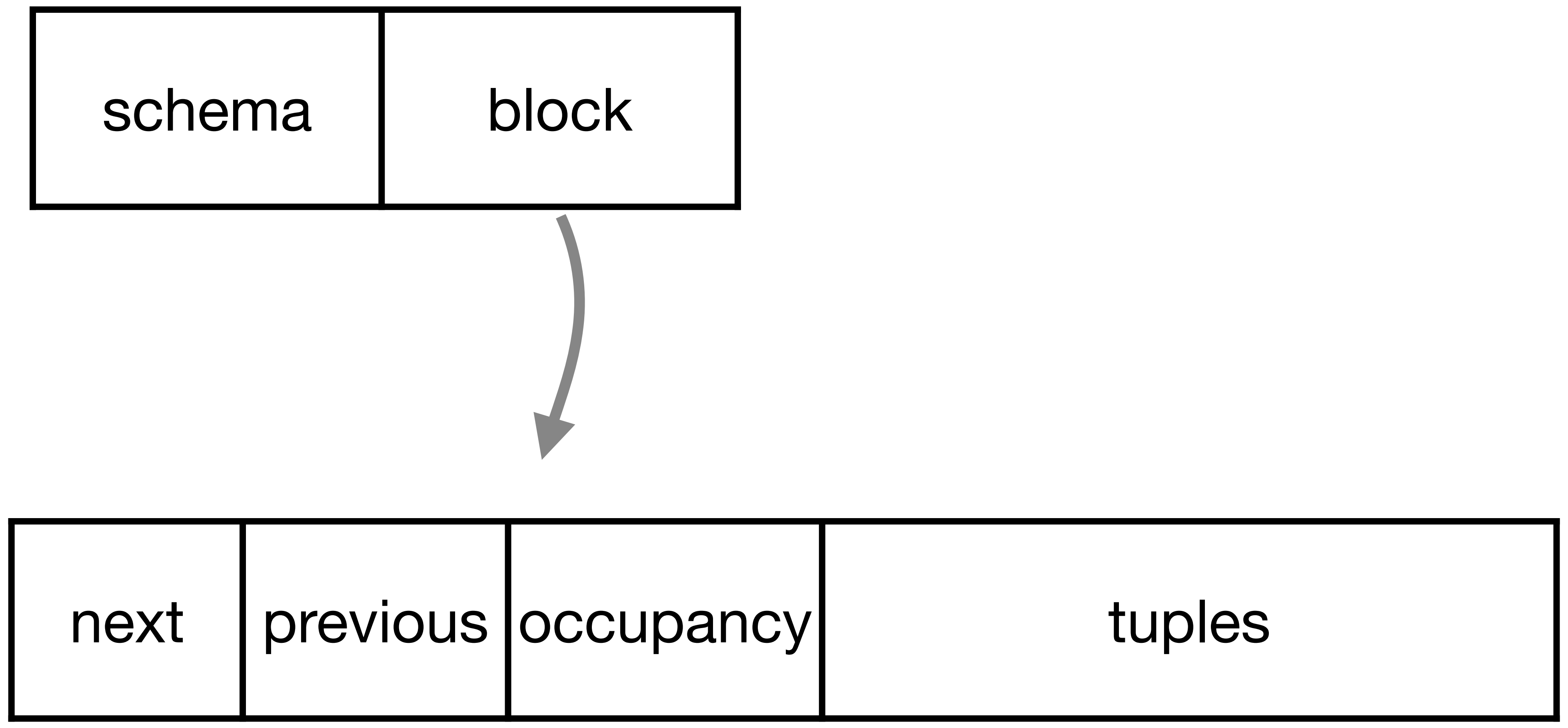
# Node ≈ Disk Block

```
(define (new scma next)
  (define blck (fs:new-block (scma:disk scma)))
  (define node (make scma blck))
  (next!     node next)
  (previous! node fs:null-block)
  (occupancy-bits! node 0)
  node)
```

"cons"

```
(define (delete! node)
  (fs:delete-block (block node)))

(define (read scma bptr)
  (define disk (scma:disk scma))
  (define blck (disk:read-block disk bptr))
  (make scma blck))

(define (write! node)
  (define blck (block node))
  (disk:write-block! blck))

(define (position node)
  (disk:position (block node)))
```

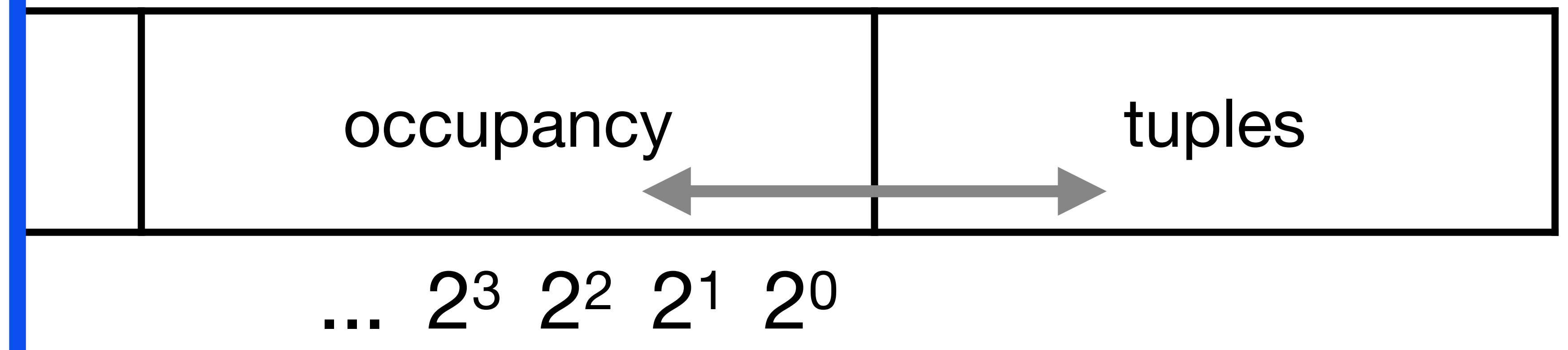
dikwijls next of previous




**Gewoon doorvertalen**

# Beheer van Tupelslots

```
(define (occupy-slot! node slot)
  (define bits (occupancy-bits node))
  (occupancy-bits! node (bitwise-ior bits (expt 2 slot))))

(define (clear-slot! node slot)
  (define bits (occupancy-bits node))
  (occupancy-bits! node (bitwise-and bits (bitwise-not (expt 2 slot)))))

(define (slot-occupied? node slot)
  (define bits (occupancy-bits node))
  (not (= (bitwise-and bits (expt 2 slot)) 0)))

(define (all-free? node)
  (define bits (occupancy-bits node))
  (= bits 0))

(define (all-occupied? node)
  (define bits (occupancy-bits node))
  (define all1 (- (expt 2 (scma:capacity (schema node))) 1))
  (= (bitwise-and bits all1) all1))
```




$$2^{cap} = 1\overbrace{00\ldots00}^{cap}$$

$$2^{cap}-1 = 11\ldots11$$

# Een tupel in een node schrijven

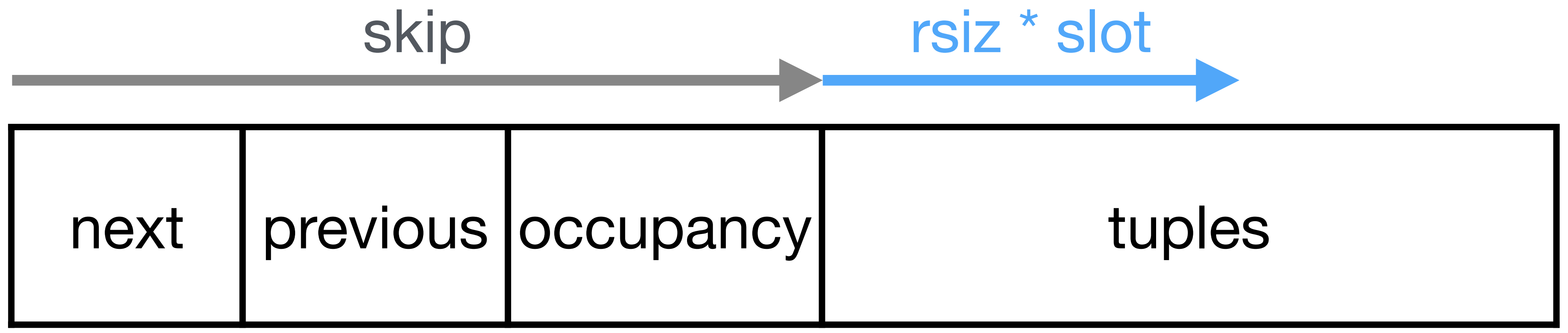
```
(define (record! node slot tupl)
  (define scma (schema node))
  (define blck (block node))
  (let loop
    ((cntr 0)
     (offs (calc-record-offset node slot))
     (vals tupl))
    (cond ((null? vals)
           (if (< cntr (scma:nr-of-attributes scma))
             (error "too few values in tuple" vals)))
          ((>= cntr (scma:nr-of-attributes scma))
           (if (not (null? vals))
             (error "too many values in tuple" vals)))
          (else ((vector-ref encoders (scma:type scma cntr))
                  blck offs (scma:size scma cntr) (car vals))
                (loop (+ cntr 1) (+ offs (scma:size scma cntr)) (cdr vals)))))
  (occupy-slot! node slot))
```

```
(define (calc-record-offset node slot)
  (define scma (schema node))
  (define skip (+ scma:fixed-header-size (scma:nr-of-occupancy-bytes scma)))
  (define rsiz (scma:record-size scma))
  (+ skip (* rsiz slot)))
```

```
(define encoders (vector disk:encode-fixed-natural!
                         disk:encode-arbitrary-integer!
                         disk:encode-real!
                         disk:encode-string!))
```

Encoder ∀ type

Waarde per waarde in het blok encoderen



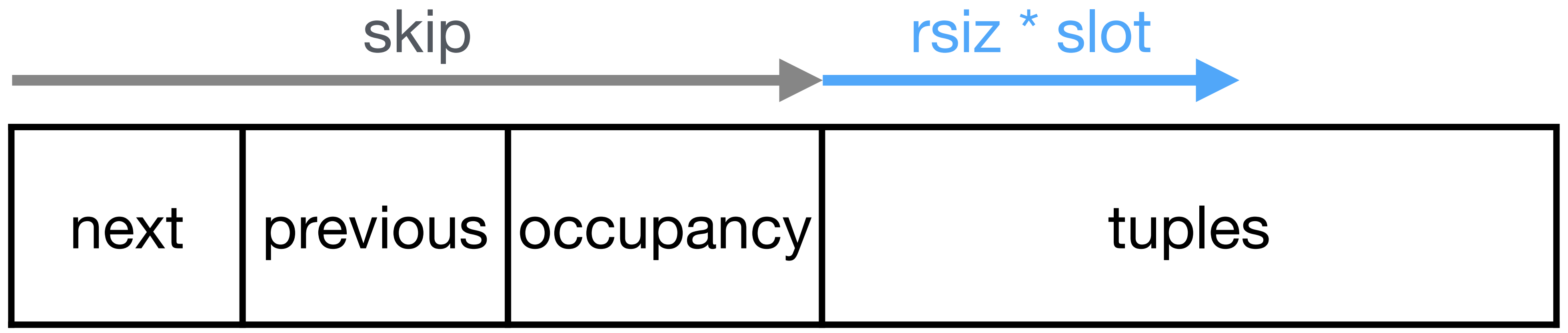
# Een tupel uit een node lezen

```
(define (record node slot)
  (define scma (schema node))
  (define blck (block node))
  (let loop
    ((cntr 0)
     (offs (calc-record-offset node slot)))
    (if (= cntr (scma:nr-of-attributes scma))
        '()
        (cons ((vector-ref decoders (scma:type scma cntr))
               blck offs (scma:size scma cntr))
              (loop (+ cntr 1) (+ offs (scma:size scma cntr)))))))
```

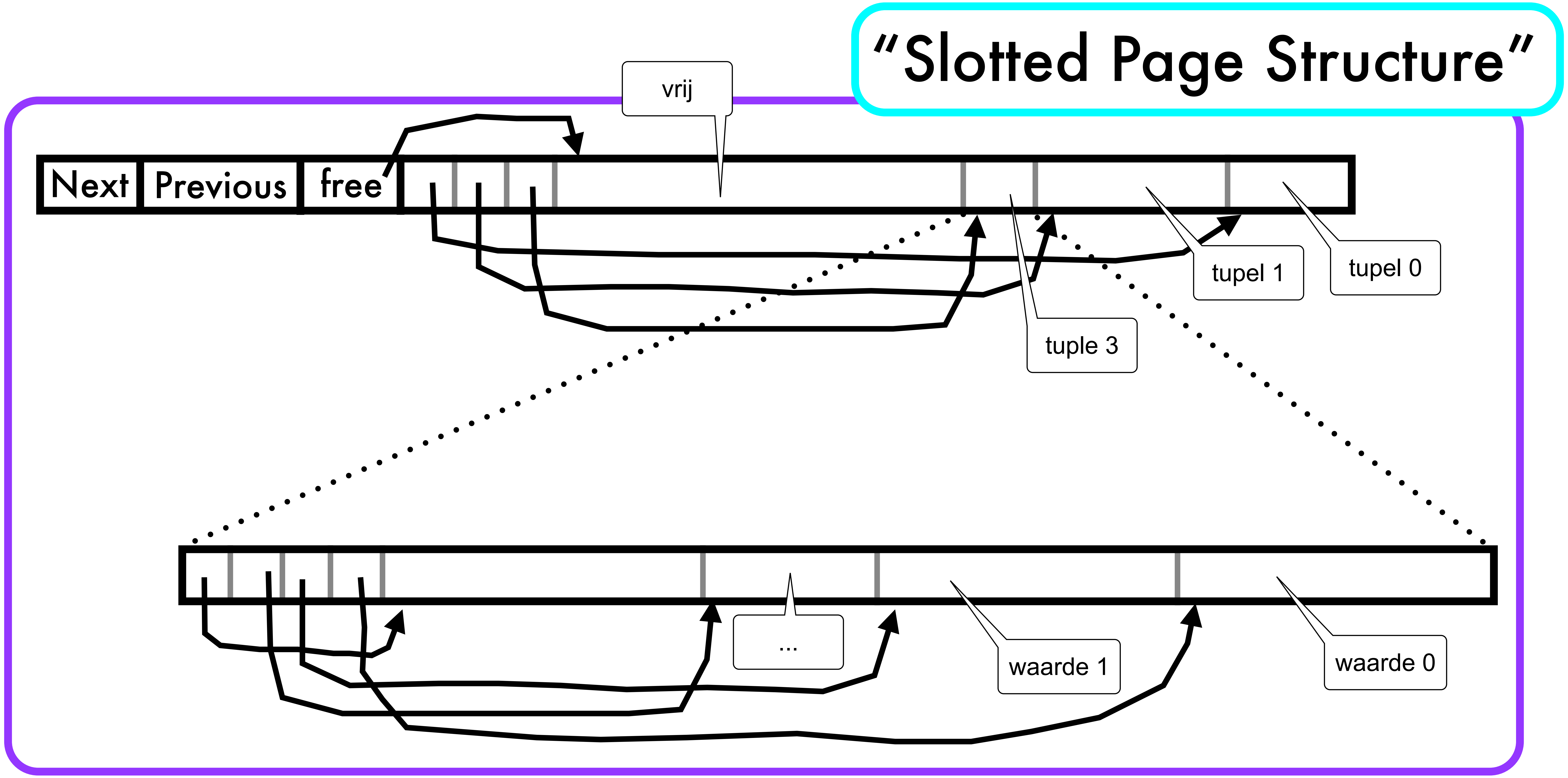
**Waarde per waarde uit het blok decoderen**

```
(define decoders  (vector disk:decode-fixed-natural
                          disk:decode-arbitrary-integer
                          disk:decode-real
                          disk:decode-string))
```

**Decoder ∀ type**



# Niet-triviale variant: variabele tupels



# Tabel ADT

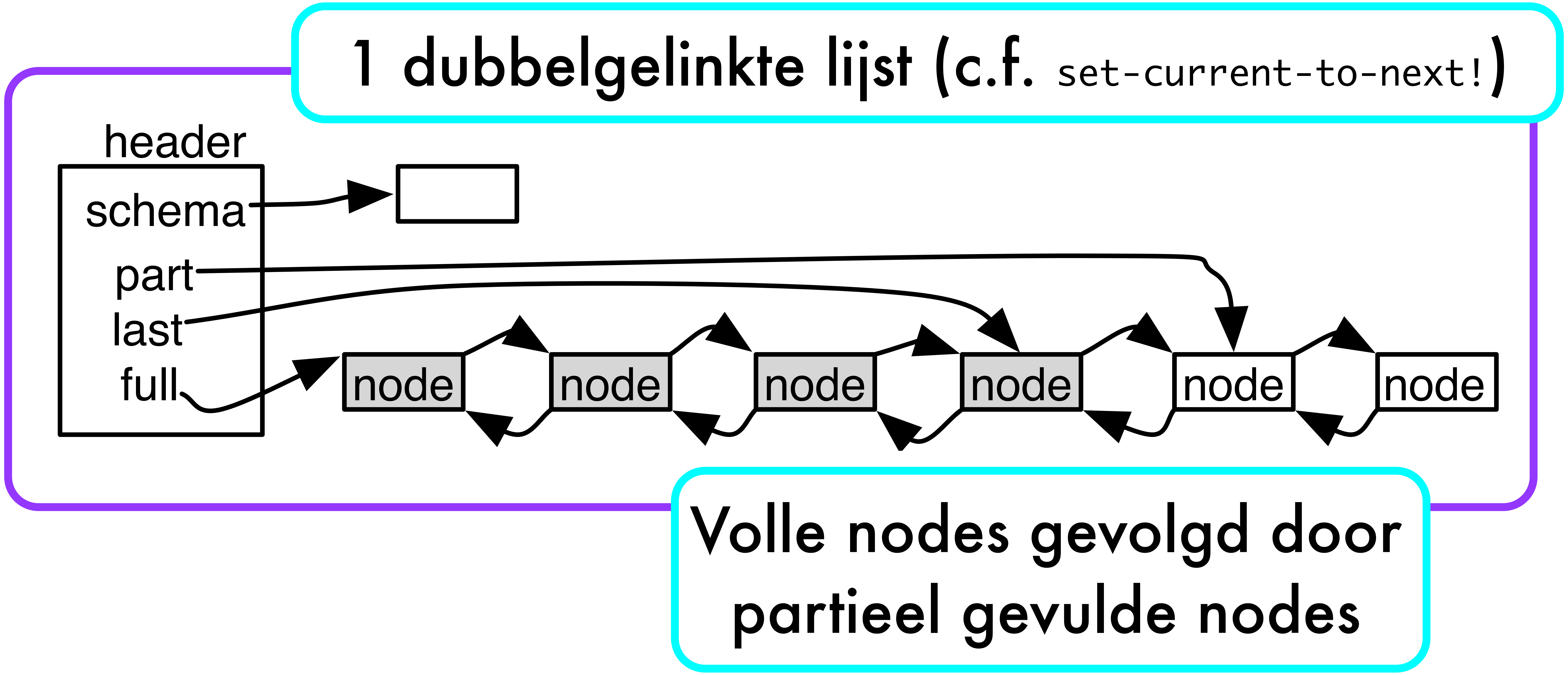
```
ADT table

new
  ( disk string pair → table )
open
  ( disk string → table )
close!
  ( table → ∅ )
table?
  ( any → boolean )
drop!
  ( table → ∅ )
schema
  ( table → schema )
disk
  ( table → disk )
set-current-to-first!
  ( table → status )
set-current-to-next!
  ( table → status )
current
  ( table → rcid U {no-current})
current!
  ( table rcid → ∅ )
```

```
status = { done,
           no-current,
           next-higher,
           duplicate,
           not-found }
```

```
insert!
  ( table pair → rcid )
delete!
  ( table rcid → ∅ )
peek
  ( table → pair )
```

**Merk op: find wordt niet ondersteund**

# Architectuur van de Datafile



# Duale Representatie van Header

disk

```
(define full-offset   0)
(define last-offset   (+ full-offset disk:block-pointer-size))
(define part-offset   (+ last-offset disk:block-pointer-size))
(define schema-offset (+ part-offset disk:block-pointer-size))

(define (full tble)
  ...)
(define (full!  tble bptr)
  ...)
(define (last tble)
  ...)
(define (last!  tble bptr)
  ...)
(define (part tble)
  ...)
(define (part! tble bptr)
  ...)
(define (schema-ptr tble)
  ...)
(define (schema-ptr! tble bptr)
  ...)
```

Scheme

```
(define-record-type table
  (make n h s b l)
  table?
  (n name)
  (h header header!)
  (s schema schema!)
  (b buffer buffer!)
  (l slot   slot!))
```

# Creatie & Destructie

```
(define (new disk name atts)
  (define scma (scma:new disk atts))
  (define hder (fs:new-block disk))
  (define tble (make name hder scma () -1))
  (full! tble fs:null-block)
  (last! tble fs:null-block)
  (part! tble fs:null-block)
  (schema-bptr! tble (scma:position scma))
  (disk:write-block! hder)
  (fs:mk disk name (disk:position hder))
  tble)
```

```
(define (open disk name)
  (define hptr (fs:whereis disk name))
  (define hder (disk:read-block disk hptr))
  (define tble (make name hder () () -1))
  (define sptr (schema-bptr tble))
  (define scma (scma:open disk sptr))
  (schema! tble scma)
  tble)
```

```
(define (close! tble)
  (disk:write-block! (header tble)))
```

**Geïnvalideerde current**

```
(define (drop! tble)
  (define scma (schema tble))
  (define hder (header tble))
  (define disk (scma:disk scma))
  (define (delete-nodes bptr)
    (if (not (fs:null-block? bptr))
        (let*
            ((node (node:read scma bptr))
             (next (node:next node)))
          (node:delete! node)
          (delete-nodes next))))
  (if (fs:null-block? (full tble))
      (delete-nodes (part tble))
      (delete-nodes (full tble)))
  (scma:delete! scma)
  (fs:delete-block hder)
  (fs:rm disk (name tble)))
```

# Beheren v/d Dubbelgelinkte Lijst

```
(define (extract-node! tble first first! node)
  (define next-bptr (node:next node))
  (define prev-bptr (node:pr
  (node:next!     node fs:nu
  (node:previous! node fs:nu
  (if (not (fs:null-block? n
      (let ((next (node:read
        (node:previous! next
        (node:write! next)))
  (if (not (fs:null-block? p
      (let ((prev (node:read
        (node:next! prev nex
        (node:write! prev)))
  (if (= (first t      node:
      (first! tbl          r
  (if (= (las     ble)
      (last         rev-
```

```
(define (insert-node! tble first first! node)
  (define frst-bptr (first tble))
  (if (not (fs:null-block? frst-bptr))
      (let ((next (node:read (schema tble) frst-bptr)))
        (node:previous! next (node:position node))
        (node:next! node frst-bptr)
        (node:write! next)))
  (first! tble (node:position node)))

(define (insert-part! tble node)
  (define last-bptr (last tble))
  (insert-node! tble part part! node)
  (if (not (fs:null-block? last-bptr))
      (let ((prev (node:read (schema tble) last-bptr)))
        (node:next! prev (node:position node))
        (node:previous! node last-bptr)
          de:write! prev))))

   fine       t-full! tble node)
     fine      t-bptr (last tble))
     sert-node! tble full full! node)
   (if (fs:null-block? last-bptr)
      (last! tble (node:position node))))
```

Niet zo interessant

C.f. hoofdstuk 3

# Toevoeging van een tupel

```
(define (insert! tble tupl)
  (define scma (schema tble))
  (define room (part tble))
  (define node (if (fs:null-block? room)
                   (let ((new (node:new scma fs:null-block)))
                     (insert-part! tble new)
                     new)
                   (node:read scma room)))
  (define free (find-free-slot node -1))
  (node:record! node free tupl)
  (when (node:all-occupied? node)
    (extract-node! tble part part! node)
    (insert-full!  tble node))
  (node:write!  node)
  (buffer!      tble node)
  (slot!        tble free)
  (rcid:new (node:position node) free))
```

Ofwel verse node aanmaken, ofwel uit de "part" lijst

Node eventueel naar de "full" lijst verplaatsen

```
(define (find-free-slot node strt)
  (define scma (node:schema node))
  (let loop
    ((cntr (+ strt 1)))
    (cond ((= (scma:capacity scma) cntr)
           -1)
          ((node:slot-occupied? node cntr)
           (loop (+ cntr 1)))
          (else
           cntr))))
```

# Verwijdering van een tupel

```
(define (delete! tble rcid)
  (define scma (schema tble))
  (define node (node:read scma (rcid:bptr rcid)))
  (define was-full? (node:all-occupied? node))
  (node:clear-slot! node (rcid:slot rcid))
  (cond ((node:all-free? node)
         (if was-full?
           (extract-node! tble full full! node)
           (extract-node! tble part part! node))
         (node:delete! node))
        (else
         (when was-full?
           (extract-node! tble full full! node)
           (insert-part! tble node))
         (node:write! node)))
  (buffer! tble ())
  (slot!   tble -1))
```

De node naar de "parts" lijst verplaatsen indien ze vol <u>was</u>

Lege node terug vrijgeven aan het filesysteem

# Beheer van de current (1)

```
(define (set-current-to-first! tble)
  (define scma (schema tble))
  (if (and (fs:null-block? (full tble))
           (fs:null-block? (part tble)))
    no-current
    (let* ((fptr (if (fs:null-block? (full tble))
                     (part tble)
                     (full tble)))
           (bffr (node:read scma fptr))
           (curr (find-occupied-slot bffr -1)))
      (buffer! tble bffr)
      (slot!   tble curr)
      done)))
```

Zoek het eerste bezette slot in het de eerste node

```
(define (find-occupied-slot node strt)
  (define scma (node:schema node))
  (let loop
    ((cntr (+ strt 1)))
    (cond ((= (scma:capacity scma) cntr)
           -1)
          ((node:slot-occupied? node cntr)
           cntr)
          (else
           (loop (+ cntr 1))))))
```

Raison d'être van de occupancy bits

# Beheer van de current (2)

```
(define (set-current-to-next! tble)
 (define scma (schema tble))
 (define bffr (buffer tble))
 (define curr (slot   tble))
 (if (null? bffr)
     no-current
     (let ((indx (find-occupied-slot bffr curr)))
       (cond ((not (= indx -1))
              (buffer! tble bffr)
              (slot!   tble indx)
              done)
             ((not (fs:null-block? (node:next bffr)))
              (let* ((next (node:read scma (node:next bffr)))
                     (indx  (find-occupied-slot next -1)))
                (buffer! tble next)
                (slot!   tble indx)
                done))
             (else
              (buffer! tble ())
              (slot!   tble -1)
              no-current)))))
```

Zoek het volgende bezette slot in de huidige node

...of het eerste bezette slot in de volgende node

# Beheer van de current (3)

```
(define (current tble)
  (define bffr (buffer tble))
  (define curr (slot   tble))
  (if (null? curr)
      no-current
      (rcid:new (node:position bffr) curr)))

(define (current! tble rcid)
  (define bffr (buffer tble))
  (define curr (rcid:slot rcid))
  (if (or (null? bffr)
          (not (= (rcid:bptr rcid) (node:position bffr))))
    (set! bffr (node:read (schema tble) (rcid:bptr rcid))))
  (buffer! tble bffr)
  (slot!   tble curr))
```

Later nodig bij indexering van tabellen

# Evaluatie

Quantitatieve Data

Aantal records per blok $R = \frac{\texttt{block-size}}{\texttt{record-size}}$

Aantal benodigde blokken $B = \frac{n}{R}$

We gebruiken een zeer simpel kostenmodel

Benchmarks

INSERT INTO table VALUES (v1,v2,..)

SELECT FROM table WHERE att=value — Equality search

SELECT FROM table WHERE att>value — Range search

DELETE FROM table WHERE att=value — ≈ Equality search

# Alternatieven Organisatie

| | Insert | Eq-search | Range-search |
|---|---|---|---|
| Double-linked List | O(1) | O(B) | Θ(B) |
| Sorted List | O(B) | O(B) | O(B) |
| Hashing | O(a) | Θ(1,3B) | Θ(1,3B) |

Dit zijn het aantal bloktransfers; niet het aantal computationele stappen!

Waar zijn de logaritmen?

# Daarom: Index Files

Opslag van tupels in datafiles (hoofdstuk 16)

```
(define manen (db:create-table zonnestelsel "Manen" manenschema))

(db:insert-into-table! zonnestelsel manen (list "Maan" "Aarde" 3476 1877 ""))
```

Zorgen voor bewegwijzering voor snelle toegang (hoofdstuk 17)

```
(db:create-index! zonnestelsel planeten "Naam-IDX"   :planeet-naam:)
(db:create-index! zonnestelsel planeten "Omloop-IDX" :omlooptijd:)
```

```
(db:select-from/eq zonnestelsel  manen :ontdekker: "Cassini"))
```

```
(db:delete-where! zonnestelsel manen :planeet: "Aarde")
```

# Hoofdstuk 16