KU LEUVEN

Inleiding tot databanken

6. Queryverwerking (Deel 2)

Prof. dr. Paolo Pilozzi

Overzicht

Queryverwerking bespreekt hoe een (R)DBMS queries behandelt en uitvoert op een zo optimaal mogelijke manier.
Hoofdstuk 18 – Oefenzitting 5

HC8 (Deel 1):
* Inleiding & Herhaling
* Queryverwerking -en optimalisatie
* Heuristische optimalisatie van querybomen

HC9 (Deel 2):
* **Extern sorteren**
* Implementaties van operatoren
* Queryuitvoering

Extern Sorteren

Sorteren nodig voor:

1. ORDER BY
2. Duplicaten verwijderen
3. Implementatie JOIN (en andere operatoren)

Vaak niet voldoende geheugen voor algoritmes zoals quicksort (= intern sorteren)!

=> Extern sorteren (= sorteren op schijf) via merge-sort

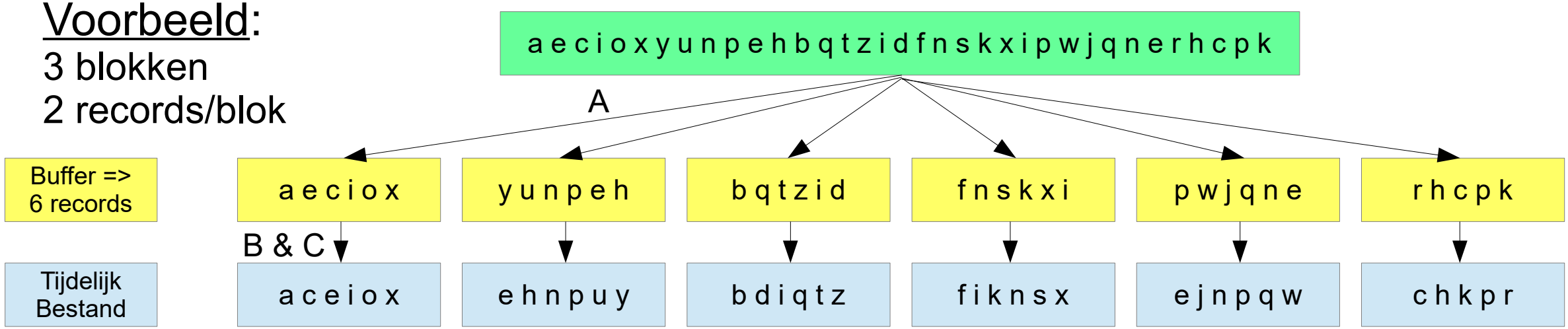
* Splits op in stukken, sorteer elk stuk en voeg samen

Extern Sorteren – Mergesort

Fase 1 (= sorteerfase): Herhaal tot hele bestand verwerkt werd:

A. Lees deel van bestand in buffer (buffer met k blokken)
B. Sorteer records in buffer (e.g., met quicksort)
C. Schrijf resultaat naar schijf (tijdelijk bestand)

Voorbeeld:
3 blokken
2 records/blok



Extern Sorteren – Mergesort

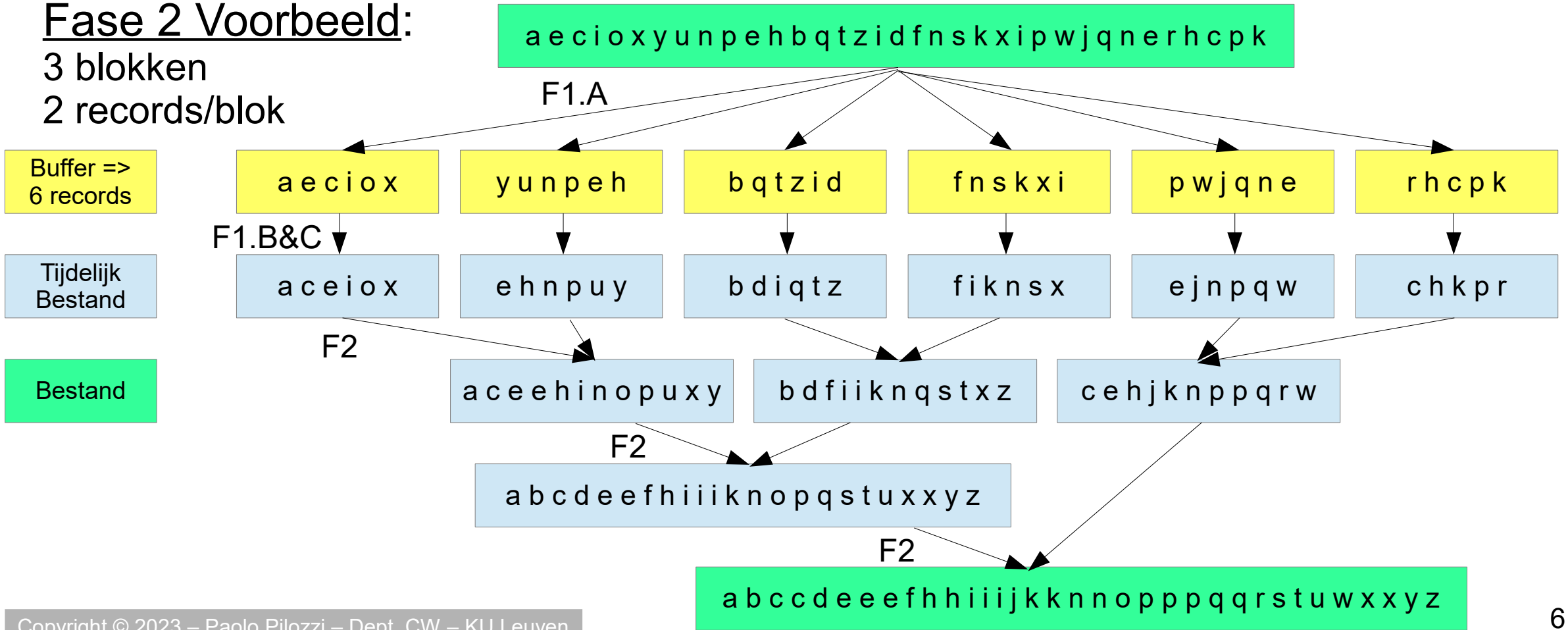
Fase 2 (= mengfase): Herhaal tot één gesorteerd bestand overblijft:

Meng k-1 gesorteerde bestanden tot één gesorteerd bestand
=> 1 blok per bestand van de k-1 bestanden in buffer
=> laatste (k-de) bufferblok (= resultaat-buffer) voor tussenresultaten

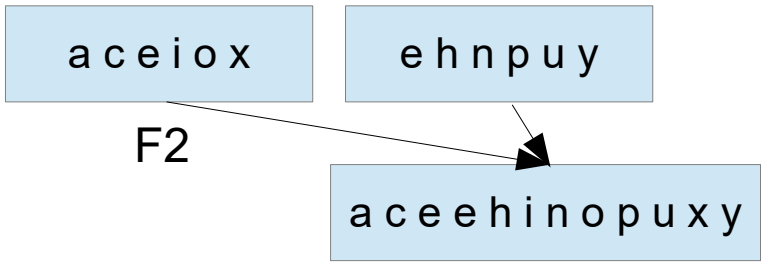
A. Vergelijk aangewezen records in alle (actieve) k-1 deelbuffers
B. Schrijf kleinste (i.e., relevante-deelbuffer) naar pointer in resultaat-buffer
C. Verschuif pointer in relevante -en resultaat-deelbuffer naar rechts
D. Als pointer relevante-deelbuffer op het einde: lees volgend blok
    D.1. Lees uit overeenkomstig bestand het volgende blok en reset pointer
    D.2. Als geen volgend blok, deactiveer deelbuffer (in A)
E. Als pointer resultaat-buffer op het einde schrijf naar sorteerbestand
    E.1 Schrijf naar sorteerbestand en reset pointer

Extern Sorteren – Mergesort

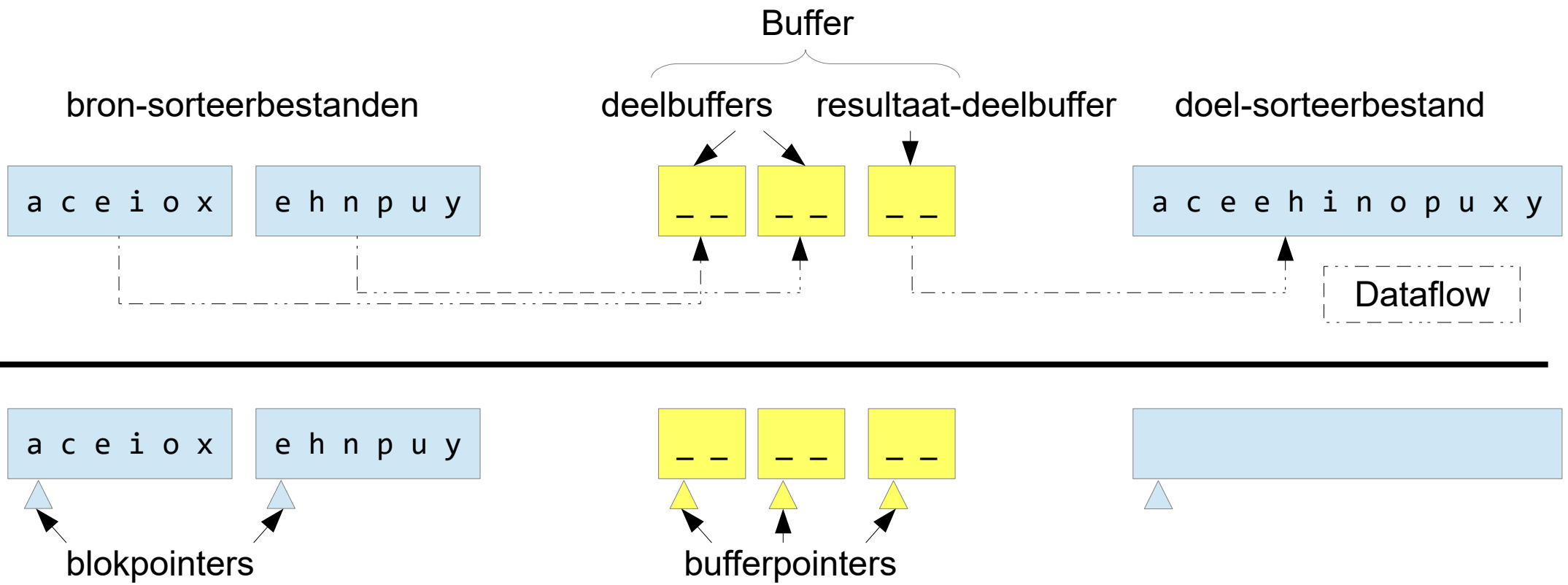
Fase 2 Voorbeeld:
3 blokken
2 records/blok



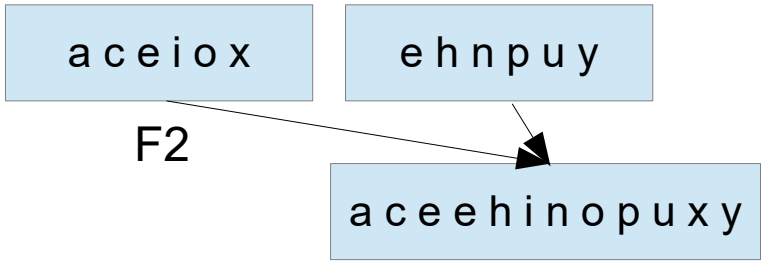
Extern Sorteren



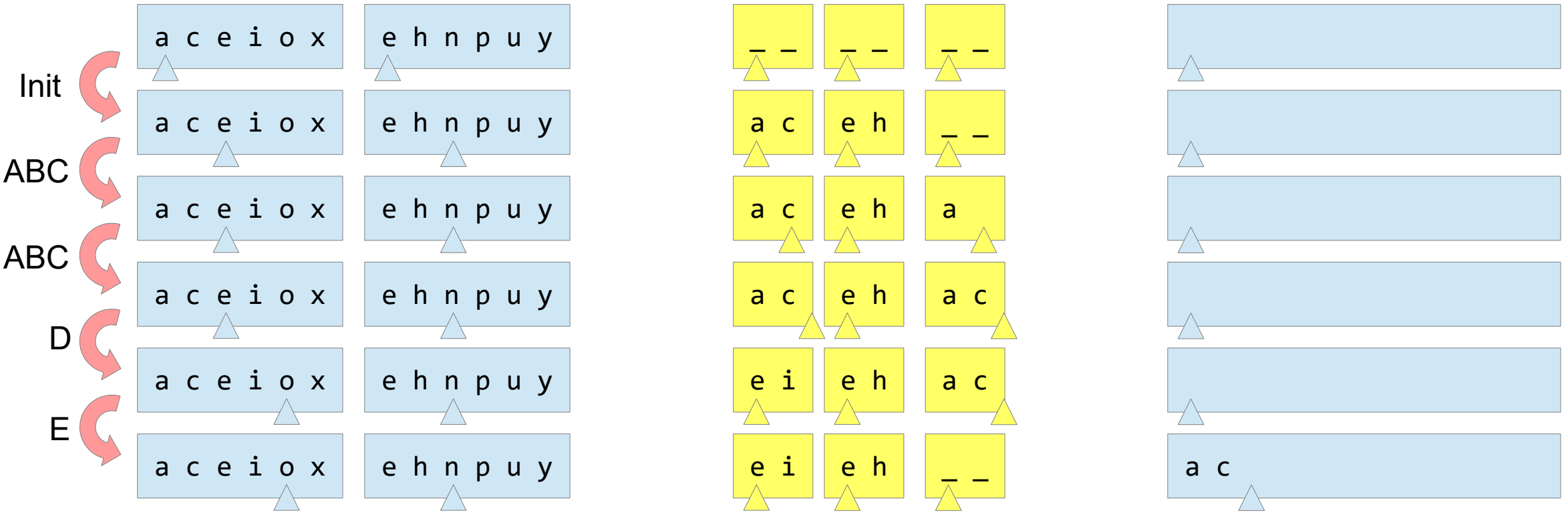
Fase 2 Voorbeeld: 3 blokken & 2 records/blok



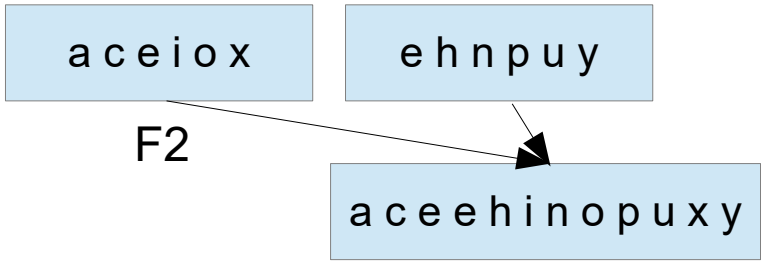
Extern Sorteren



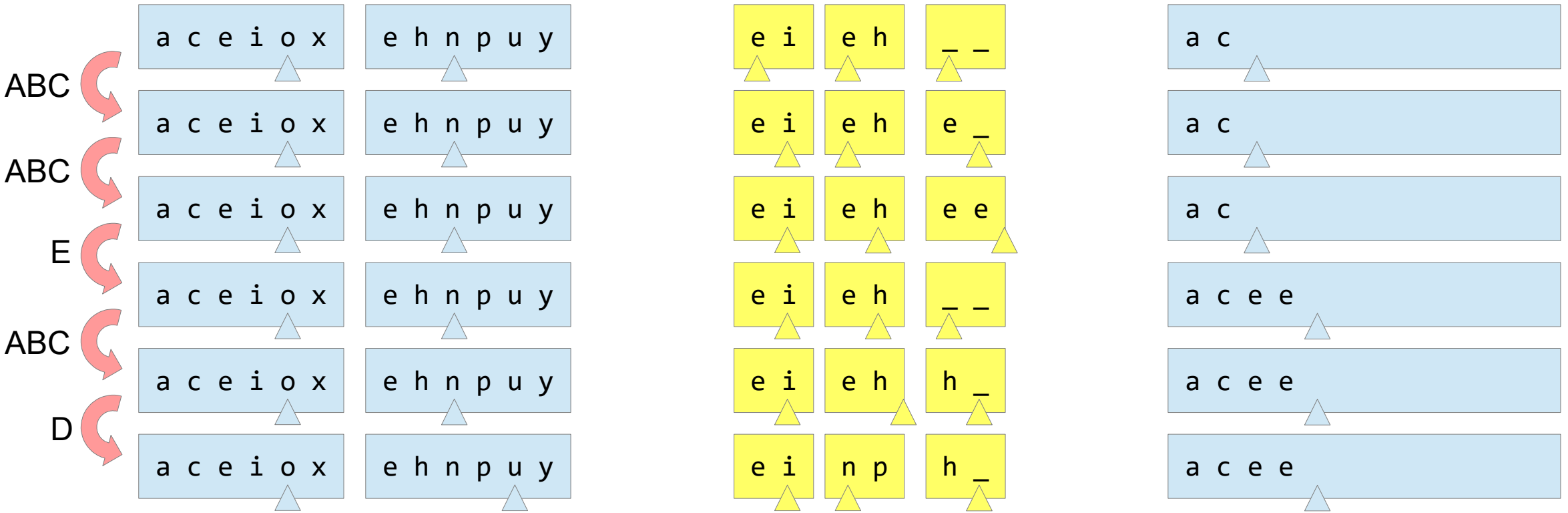
Fase 2 Voorbeeld: 3 blokken & 2 records/blok



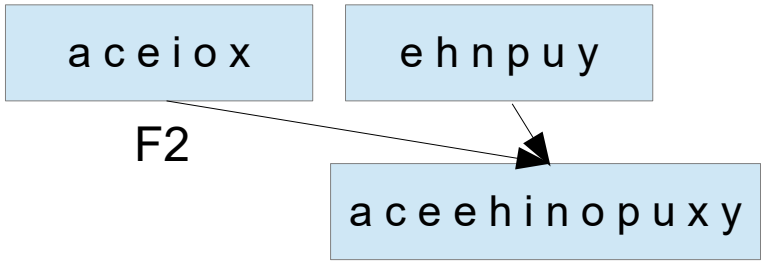
Extern Sorteren



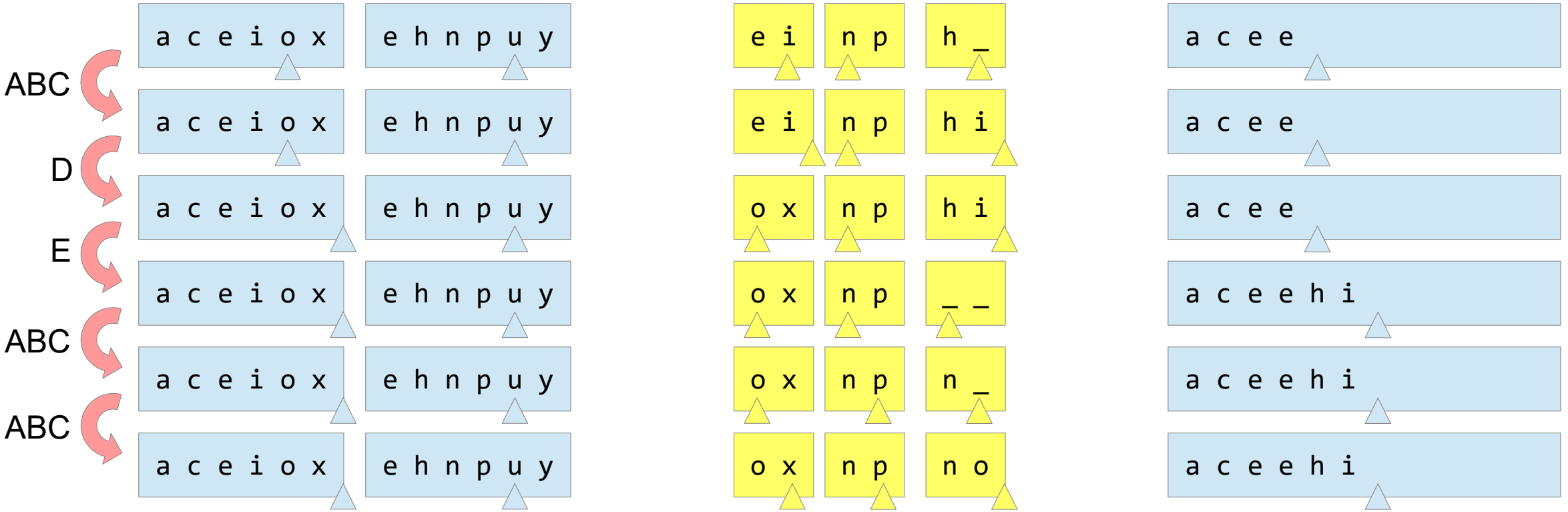
Fase 2 Voorbeeld: 3 blokken & 2 records/blok



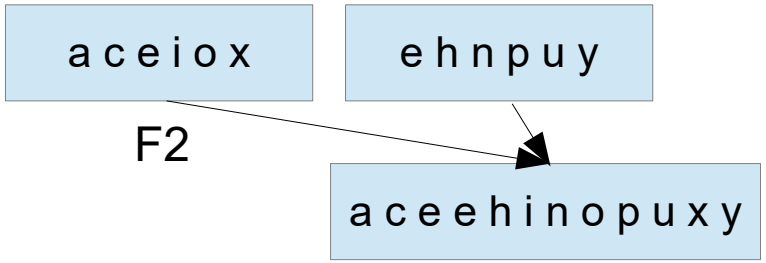
Extern Sorteren



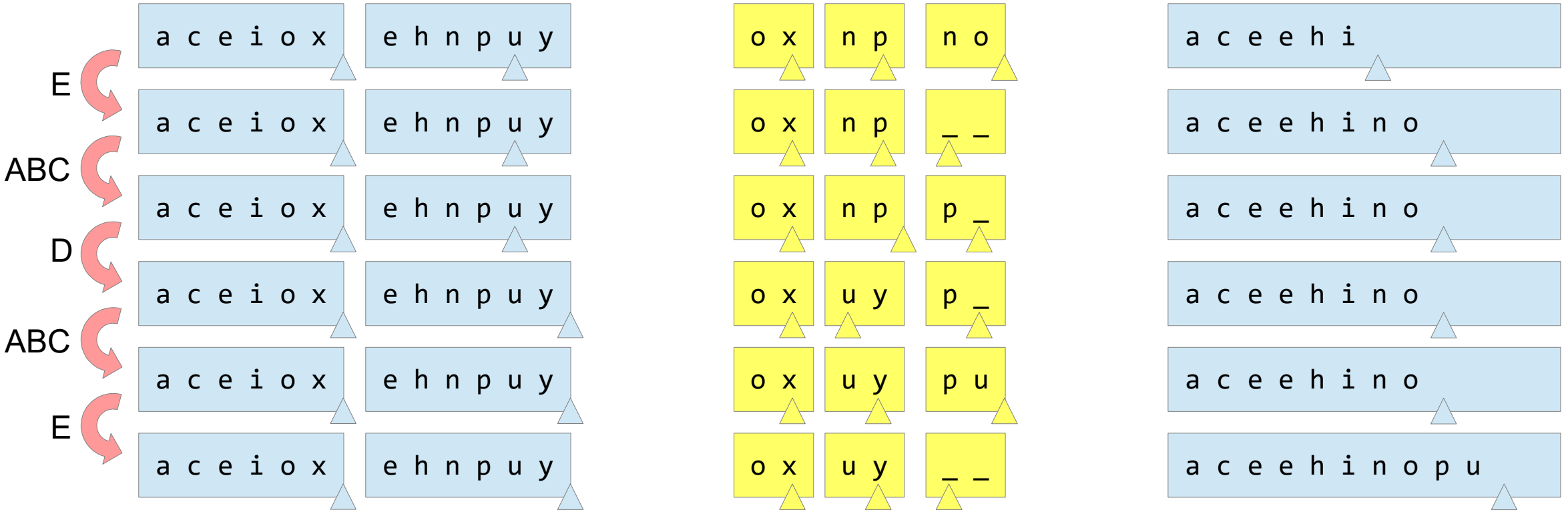
Fase 2 Voorbeeld: 3 blokken & 2 records/blok



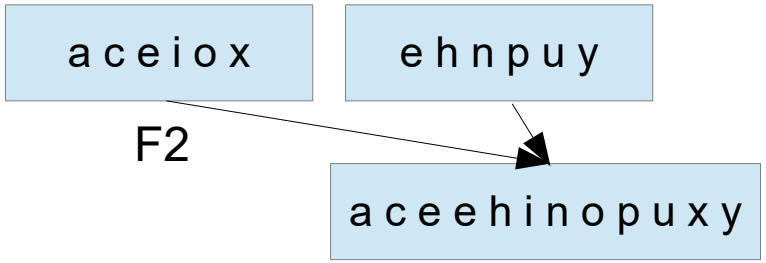
Extern Sorteren



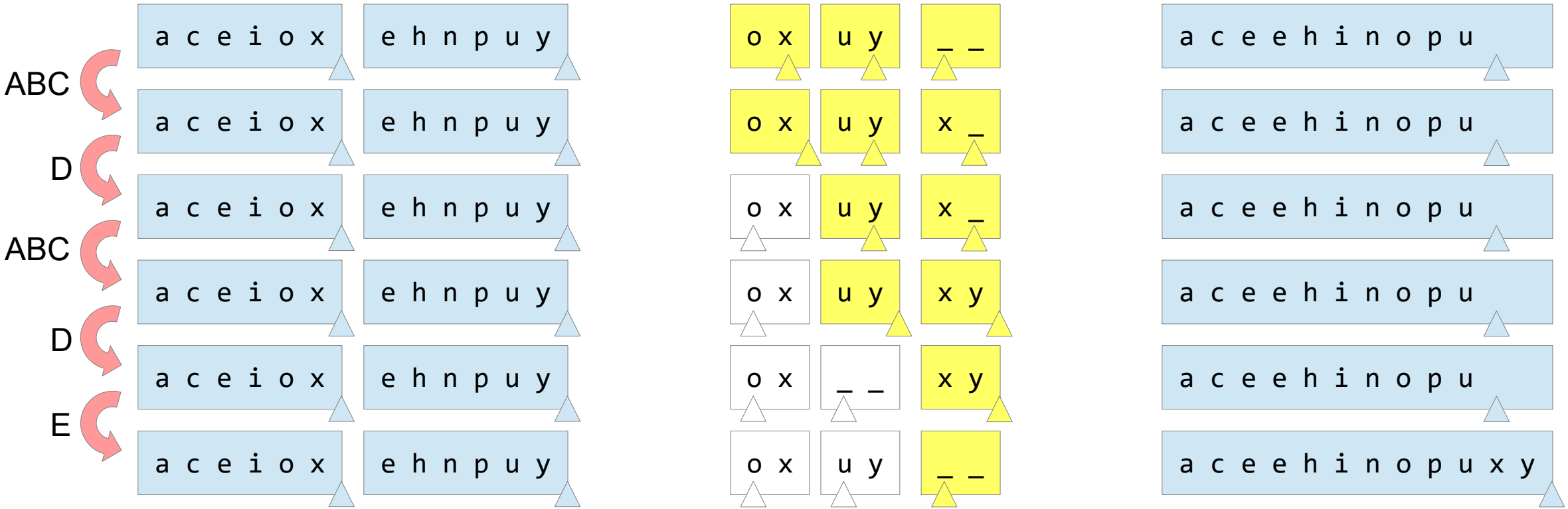
Fase 2 Voorbeeld: 3 blokken & 2 records/blok



Extern Sorteren



Fase 2 Voorbeeld: 3 blokken & 2 records/blok



Extern Sorteren – Mergesort

Complexiteit van Fase 1 (sorteerfase):

* Afhankelijk van buffer: zoveel mogelijk blokken in het geheugen
  - b blokken in origineel bestand en n_b beschikbare blokken in buffer
  - $n_r = b/n_b$ (quicksort) runs nodig
  - elke run geeft een gesorteerd deelbestand
  - er worden in totaal b blokken ingelezen en geschreven

=> 2*b blokoperaties

Voorbeeld: $n_b = 5$, $b = 1024$

=> $n_r = 1024/5 = 205$ gesorteerde deelbestanden

Extern Sorteren – Mergesort

Complexiteit van Fase 2 (mengfase):

* Afhankelijk van buffer: zoveel mogelijk blokken in het geheugen
  - n_b beschikbare blokken in buffer: *mengingsgraad* $d_m = n_b - 1$
  - aantal doorgangen/passages (= epochs): $\log_{dm}(n_r)$
  - elk epoch leest en schrijft b blokken

=> $2*b*\log_{dm}(n_r)$ blokoperaties

=> Totale complexiteit: $2*b+2*b*\log_{dm}(n_r) <=> 2*b*(1+\log_{dm}(b/n_b))$

Voorbeeld: $d_m = 5-1 = 4$

=> Gesorteerde deelbestanden per epoch: 205 → 52 → 13 → 4 → 1

Overzicht

Queryverwerking bespreekt hoe een (R)DBMS queries behandelt en uitvoert op een zo optimaal mogelijke manier.
Hoofdstuk 18 – Oefenzitting 5

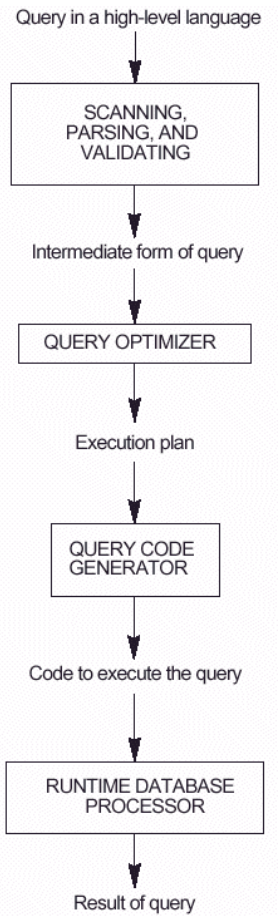
HC8 (Deel 1):
* Inleiding & Herhaling
* Queryverwerking -en optimalisatie
* Heuristische optimalisatie van querybomen

HC9 (Deel 2):
* Extern sorteren
* **Implementaties van operatoren**
* Queryuitvoering

Queryverwerking – Overzicht

Stappen in verwerking van query:

1. Lezen en ontleden van query:
   - Scanner => Lezen v. Query
   - Parser => Syntax + Voorstelling
2. Query optimalisatie
   - Genereer opties/strategieën
   - Kies meest efficiënte
3. Codegeneratie
   - Maak gekozen strategie uitvoerbaar
4. Voer uit



Optie 1: Interpreted
=> Directe uitvoering

Optie 2: Compiled
=> Code opgeslagen voor herbruik

Queryverwerking – Overzicht

Doel: Weinig en kleine records & Efficiënt blokkentransport

A. Zo klein mogelijke tussenrelaties
   1. Op schema-niveau Transformatieregels relationele algebra
      => Herwerken query-bomen: transversaal schuiven (boven-onder)
   2. Op gegevens-niveau: info rond records, selectiviteit, enz.
      => Herwerken query-bomen: lateraal schuiven (links-rechts)

B. Zo min mogelijk aantal lees, schrijf, -en vergelijkingsoperaties
   3. Op algoritme-niveau: Efficiënte implementaties
      => Gebruik maken van indexen
      => Efficiënte selectie, join, ... operaties

Merk op: A help bij B

Impl. v. Operatoren – Selectie

Verschillende strategieën op basis van:

1. Soort selectiecriterium

Voorbeeld selectiecriteria:

* OP1: $\sigma_{\text{Ssn = '123456789'}}(\text{EMPLOYEE})$
* OP2: $\sigma_{\text{Dnumber > 5}}(\text{DEPARTMENT})$
* OP3: $\sigma_{\text{Dno = 5}}(\text{EMPLOYEE})$
* OP4: $\sigma_{\text{Dno = 5 AND Salary > 3000 AND Sex = 'F'}}(\text{EMPLOYEE})$
* OP5: $\sigma_{\text{Essn = '123456789' AND Pno = 10}}(\text{WORKS\_ON})$

Impl. v. Operatoren – Selectie

Verschillende strategieën op basis van:

1. Soort selectiecriterium
2. Bestaan van indexen
   * Primaire index: op uniek veld dat ordening bepaald
   * Clusterindex: op niet-uniek veld dat ordening bepaald
   * Secundaire index: op veld dat ordening niet bepaald

Merk op: een relatie R bestaat uit b_R blokken (v. records) op schijf

Impl. v. Operatoren – Selectie

Strategie 1 (S1): Lineair zoeken

= Doorloop heel het bestand en test elk record

* Kan altijd (condities van elke vorm)

Kost: b_R blokken transporteren

Kost – Zoeken op sleutel: Gemiddeld $b_R/2$ blokken transporteren

Impl. v. Operatoren – Selectie

Strategie 2 (S2): Binair zoeken

= Doorloop het bestand in opdelende wijze (bisectie) waarbij records getest worden om zo te bepalen wat over te houden
* Kan als relatie geordend is op selectieattribuut A en
* selectiecriterium van de vorm "A = w"

Voorbeeld: [OP1] $\sigma_{Ssn = '123456789'}$(EMPLOYEE), met geordend op Ssn

Kost: $\log_2(b_R)$ blokken transporteren

Impl. v. Operatoren – Selectie

Strategie 3 (S3): Ophalen van één record (= point query)
via primaire index of hash-functie (sleutel n. blok/recordpointer)
met als mogelijke selectiecriteria:

* record met bepaalde waarde (gelijkheden) met als vorm "A = w"

Voorbeeld: [OP1] $\sigma_{Ssn = '123456789'}$(EMPLOYEE), met prim./hash-idx op Ssn

Kost – *primaire index*: # blokken om w te vinden
Kost – *hash-index*: typisch 1 blok (soms 2, en zelden meerdere)

Impl. v. Operatoren – Selectie

Strategie 4 (S4): Ophalen van meerdere records via primaire index op selectieattribuut A
met als mogelijke selectiecriteria:

* range query met als vorm "A ≥ w", "A ≤ w", "A > w" of "w < A"
* records vanaf/na een bepaald waarde (vorm "A = w")

Voorbeeld: [OP2] $\sigma_{Dnumber > 5}$(DEPARTMENT), met prim. index op Dnumber

Kost: # blokken om w te vinden + gemiddeld $b_R/2$

Impl. v. Operatoren – Selectie

Strategie 5 (S5): Ophalen van meerdere records via cluster-index op selectieattribuut A
met als mogelijke selectiecriteria:

* record met bepaalde waarde (gelijkheden) met als vorm "A = w"

Voorbeeld: [OP3] $\sigma_{Dno = 5}$(EMPLOYEE), met cluster-index op Dno

Kost: # blokken om w te vinden + s/r_b ($1+s/r_b$ is veiliger)

* s: schatting van aantal tupels dat aan de conditie voldoet
* r_b: aantal records per blok in de relatie

Impl. v. Operatoren – Selectie

Strategie 6 (S6): Ophalen van records via secundaire index op selectieattribuut A (adhv. B^+-boom)
met als mogelijke selectiecriteria:

* record met bepaalde waarde (gelijkheden) met als vorm "A = w"
* range query met als vorm "A ≥ w", "A ≤ w", "A > w" of "w < A"
* records vanaf/na een bepaald waarde (vorm "A = w")

Voorbeeld: [OP3] $\sigma_{Dno = 5}$(EMPLOYEE), met secundaire index op Dno

Impl. v. Operatoren – Selectie

Strategie 6 (S6): Ophalen van records via secundaire index op selectieattribuut A (adhv. B^+-boom)
met als mogelijke selectiecriteria:

* record met bepaalde waarde (gelijkheden) met als vorm "A = w"
* range query met als vorm "A ≥ w", "A ≤ w", "A > w" of "w < A"
* records vanaf/na een bepaald waarde (vorm "A = w")

Kost: Afhankelijk van

* vergelijkingsoperatie en uniek-zijn van selectieattribuut
* hoeveel tupels geschat voldoen aan de conditie
* kost index-gebruik

Impl. v. Operatoren – Selectie

Strategie 7 (S7): Ophalen van records via conjunctieve selectiecriteria met als vorm $c_1 \wedge c_2 \wedge \ldots \wedge c_n$:

* Als er een c_i bestaat waarvoor S2-S6 van toepassing is, dan
  => Selecteer volgens c_i en test vervolgens $c_1 \wedge c_2 \wedge \ldots \wedge c_n$

Voorbeeld: [OP4] $\sigma_{\text{Dno = 5 AND Salary > 3000 AND Sex = 'F'}}$(EMPLOYEE), m. idx op Dno

Kost: Afhankelijk van gekozen methode

Impl. v. Operatoren – Selectie

Strategie 8 (S8): Ophalen van records via conjunctieve selectiecriteria ($c_1 \wedge c_2 \wedge ... \wedge c_n$) op samengestelde index, zodat alle c_i van de vorm "A = w" zijn en zodat een samengestelde index bestaat voor de selectieattributen

Voorbeeld: [OP5] $\sigma_{\text{Essn = '123456789' AND Pno = 10}}$(WORKS_ON), met samengestelde index op (Essn,Pno)

Kost: Afhankelijk van type index + gekozen methode (bij index op deel van de selectieattributen: S7 met S2-S6 op samengestelde index)

Impl. v. Operatoren – Selectie

Strategie 9 (S9): Ophalen van records via conjunctieve selectiecriteria ($c_1 \wedge c_2 \wedge ... \wedge c_n$) met c_i van de vorm "A=w" door intersectie van recordpointers, gegeven secundaire indexen voor de selectieattributen

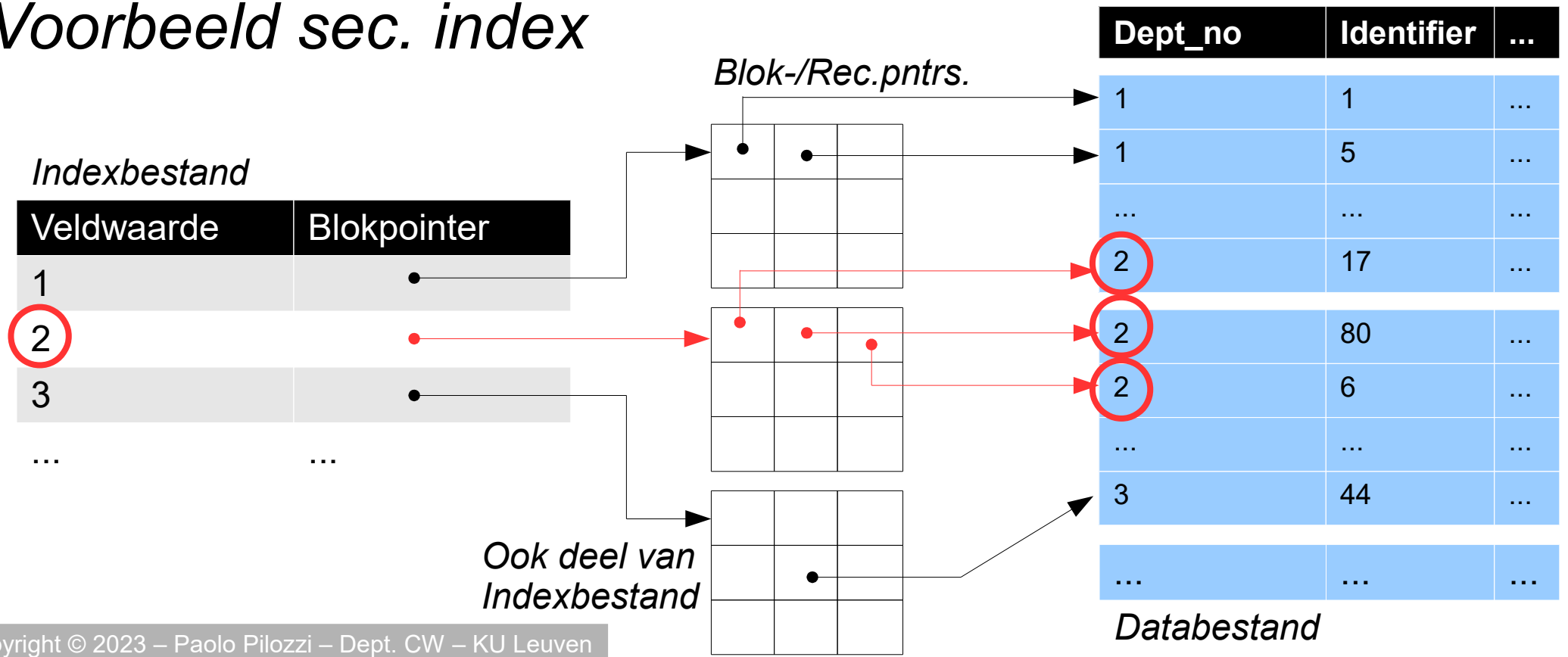
Principe: Voor elke c_i met secundaire index, haal verzameling pointers uit index, bereken hun doorsnede, haal records op en verifiëer resterende c_j

Kost: Afhankelijk van type index + gekozen methodes

Impl. v. Operatoren – Selectie

Strategie 9 (S9): intersectie van recordpointers

Voorbeeld sec. index

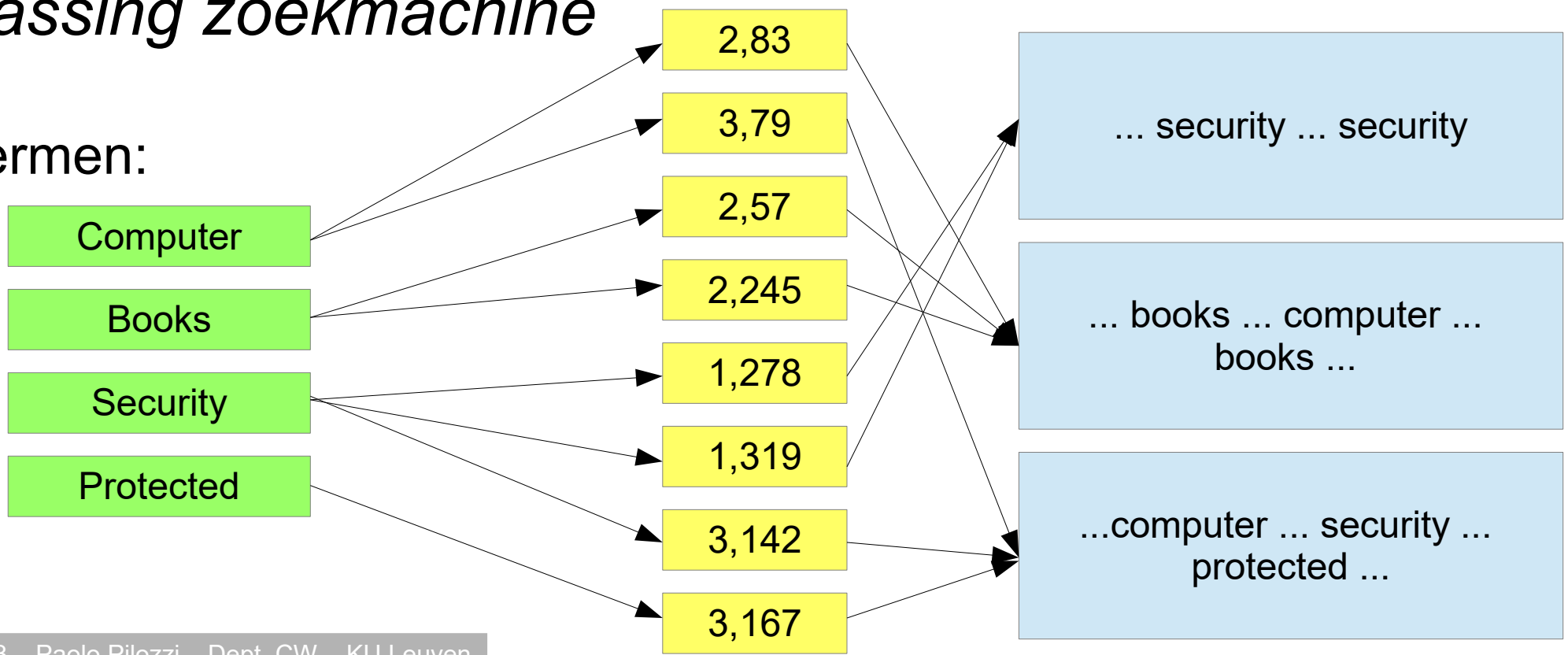


Impl. v. Operatoren – Selectie

Strategie 9 (S9): intersectie van recordpointers

Toepassing zoekmachine

Zoektermen:



Impl. v. Operatoren – Selectie

Implementatiemethode kiezen

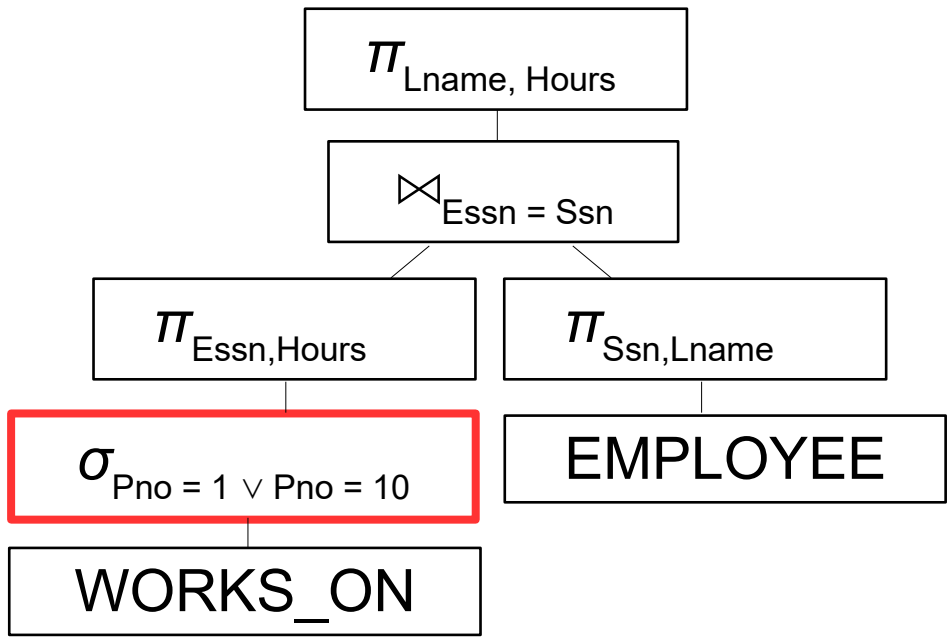
* Enkelvoudige criteria
  - Kies uit S2-S6 volgens schatting kost
  - Indien niet mogelijk: S1
* Conjunctieve criteria
  - Kies uit S7-S9 volgens schatting kost
  - Selectiviteit van condities (catalogus) in rekening brengen
* Disjunctieve criteria
  - Efficiënte methode voor elke deelvoorwaarde? JA => gebruik die
  - Zodra voor één deelvoorwaarde S1 => S1 de beste oplossing

Impl. v. Operatoren – Selectie

* Disjunctieve criteria
  - Efficiënte methode voor deelvoorwaarde? JA => gebruik die
  - Zodra voor één deelvoorwaarde S1 => S1 de beste oplossing

Tegenvoorbeeld:

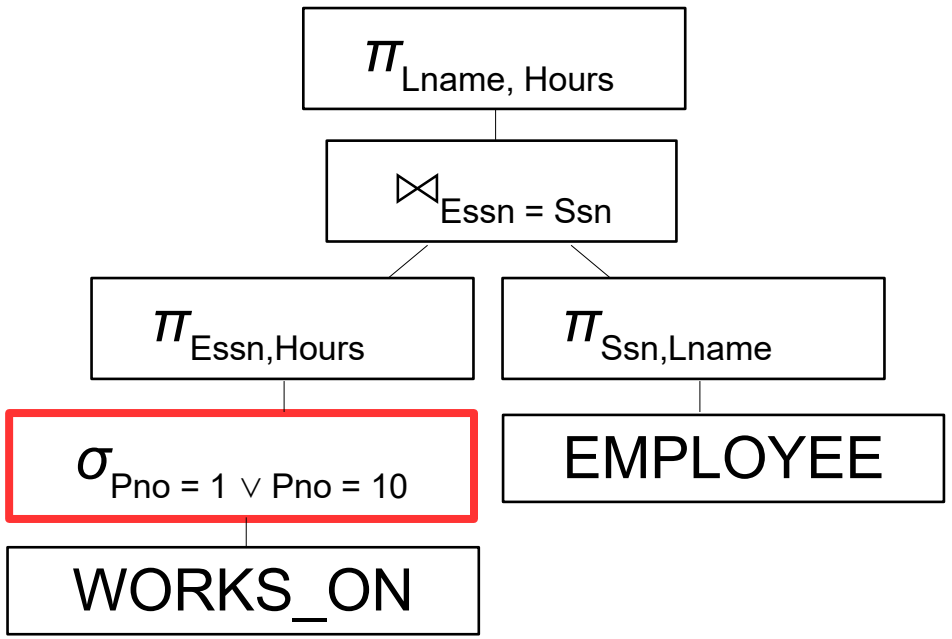
```
SELECT    Lname, Hours
FROM      EMPLOYEE, WORKS_ON

WHERE     Essn = Ssn AND
          (   Pno = 1 OR
              Pno = 10 );
```

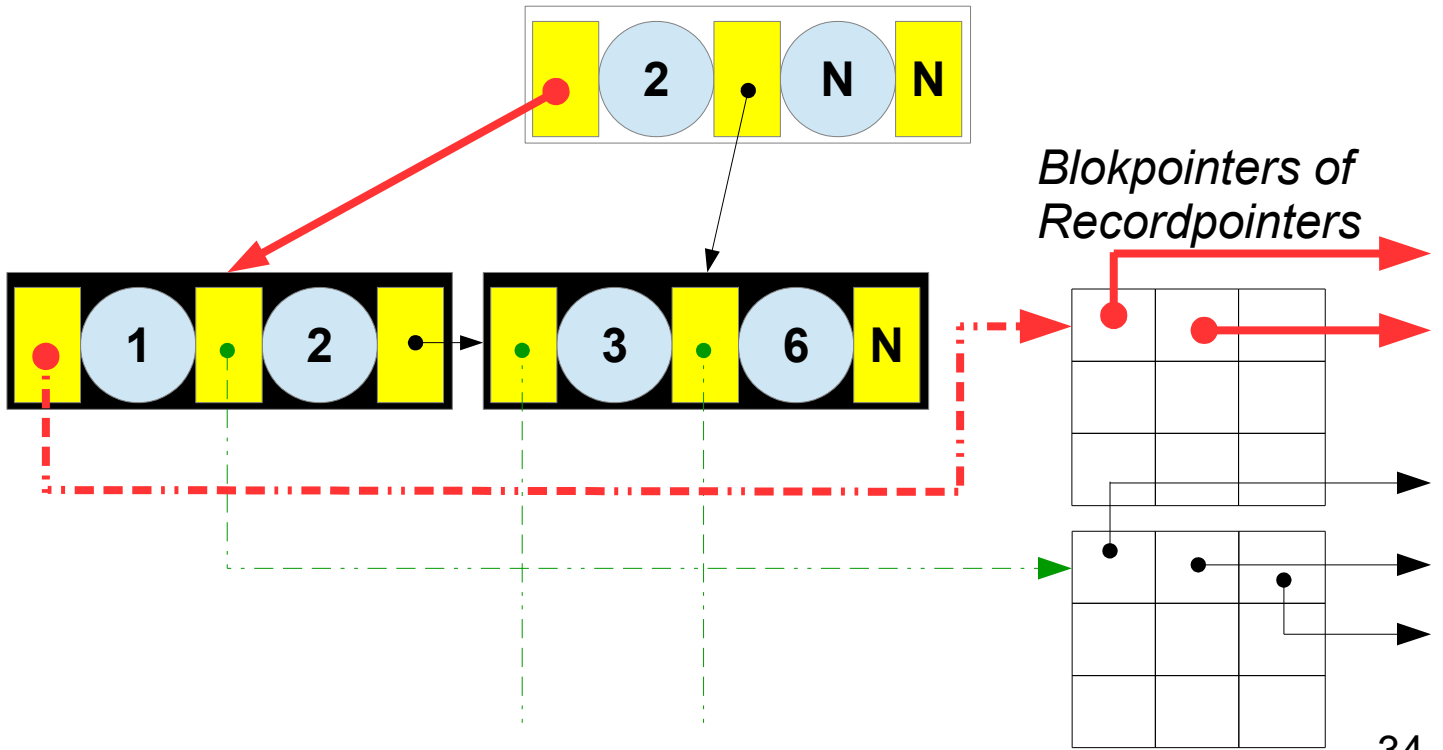


Impl. v. Operatoren – Selectie

* Disjunctieve criteria – methode
  - Efficiënte methode voor deelvoorwaarde? JA => gebruik die

Tegenvoorbeeld:

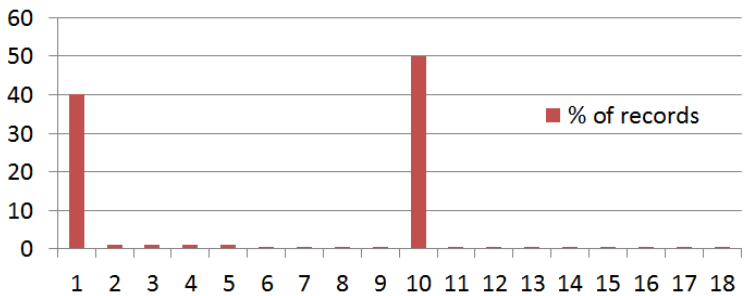


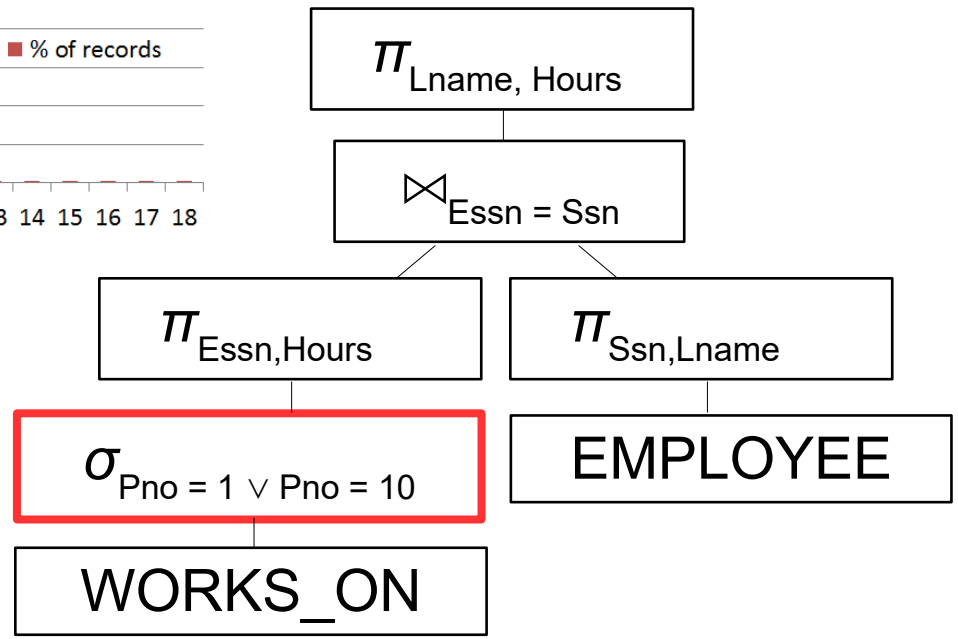


Impl. v. Operatoren – Selectie

* Disjunctieve criteria – methode
  - Efficiënte methode voor deelvoorwaarde? JA => gebruik die

Tegenvoorbeeld:





Schatting selectiegrootte:

1000 records in EMPLOYEE, 5000 in WORKS_ON

18 unieke Pno-waarden in WORKS_ON

=> Zonder histogram: (5000/18)*2 = 556

=> Met histogram: 0.4*5000+0.5*5000 = 4500

Impl. v. Operatoren – Selectie

* Via B^+-boom index op Pno
  * Voor elke waarde in de disjunctie (dus 1 en 10):
    - Opzoeken waarde in index => 3 blokken lezen (3 niveaus)
    - Opzoeken record/blokpointers => ≥1 blok lezen
    - Opzoeken records => 4500 blokken lezen
      (2000 records voor 1 en 2500 voor 10)

* Alternatief: Lineair zoeken (S1) in WORKS_ON
  * Veronderstel 10 WORKS_ON records per blok
  => 5000/10 = 500 blokken lezen

Impl. v. Operatoren – Join

Join is een dure operatie (uitvoeringstijd):

We behandelen enkel equi-join (of natural join) tussen 2 relaties met 1 join-attribuut: $R \bowtie_{A=B} S$

Voorbeelden Join operatoren:

* OP6: EMPLOYEE $\bowtie_{Dno=Dnumber}$ DEPARTMENT
* OP7: DEPARTMENT $\bowtie_{Mgr\_ssn=Ssn}$ EMPLOYEE

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Algoritme (zonder schrijven van resultaat):

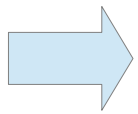
```
Voor elk tupel r in R:
    Voor elk tupel s in S:
        Als r[A] = s[B]: ...
```

Voorbeeld:

A	C
a	1
a	3
e	2
f	4
f	6
f	7

R $\bowtie_{A=B}$ S

B	D
a	x
b	x
e	x
e	y
f	y



A	C	B	D
a	1	a	x
a	3	a	x
a	2	a	x
e	2	e	x
e	2	e	y
f	4	f	Y
f	6	f	y
f	7	f	y

Impl. v. Operatoren – Join

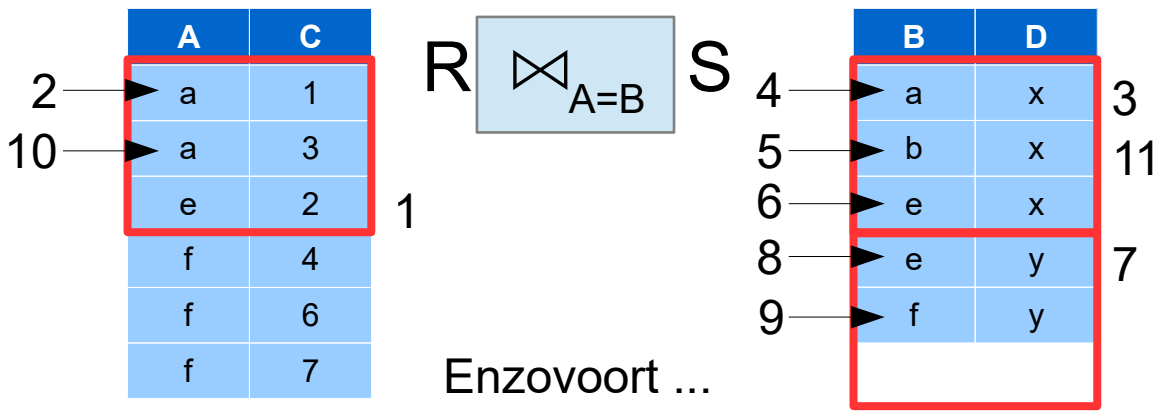
Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Naïeve methode

Voor elk tupel r in R:
  Voor elk tupel s in S:
    Als r[A] = s[B]: ...

Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Voor elk blok b_S in S:
      Voor elk record s in b_S: Als r[A] = s[B]: ...

Voorbeeld: (3 rec/blok)
3 blokken in werkgeheugen:
* 2 in (1R&1S) + 1 out
* out niet in voorbeeld



Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren: Naïeve methode

* Notatie: b_R en b_S blokken, en n_R en n_S records, in R en S, resp.

Kost:

* Min. 3 blokken nodig in geheugen: 2 IN (R & S) + 1 UIT ($R \bowtie_{A=B} S$)

=> Blokkentransport: $b_R + n_R * b_S$ blokken

Voorbeeld: R – 10000 rec. in 400 blokken; S – 5000 rec. in 100 blokken

=> $R \bowtie_{A=B} S$: 400+10000*100 = 1000400 blokken transporteren

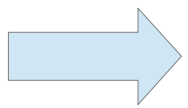
=> $S \bowtie_{A=B} R$: 100+5000*400 = 2000100 blokken transporteren

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

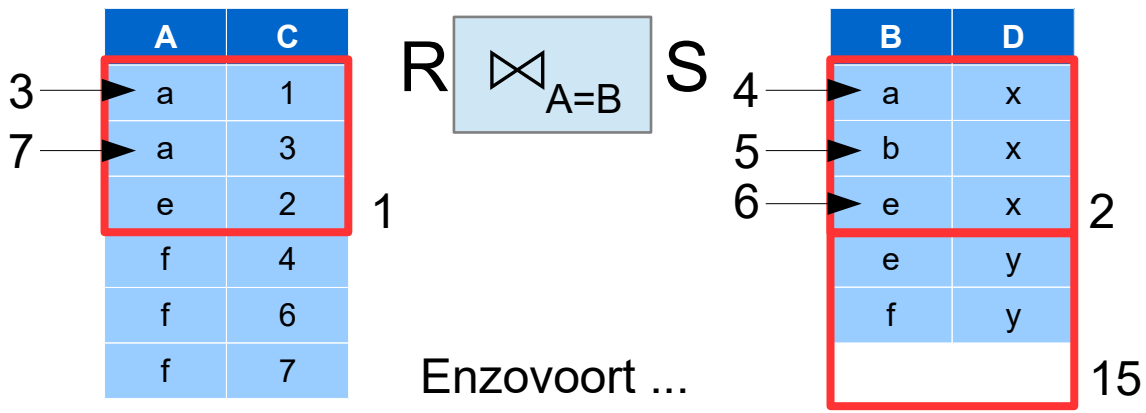
Blokken lezen – Join per blok

Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Voor elk blok b_S in S:
      Voor elk record s in b_S:



Voor elk blok b_R in R:
  Voor elk blok b_S in S:
    Voor elk record r in b_R:
      Voor elk record s in b_S:

Voorbeeld: (3 rec/blok)
3 blokken in werkgeheugen:
* 2 in (1R&1S) + 1 out
* out niet in voorbeeld



Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren: Join per blok

* Notatie: b_R en b_S blokken, en n_R en n_S records, in R en S, resp.

Kost:

* Met 3 blokken in geheugen: 2 IN (R & S) + 1 UIT ($R \bowtie_{A=B} S$)

=> Blokkentransport: $b_R + b_R * b_S$ blokken

Voorbeeld: R – 10000 rec. in 400 blokken; S – 5000 rec. in 100 blokken

=> $R \bowtie_{A=B} S$: 400+400*100 = 40400 (ipv. 1000400) blokken transporteren

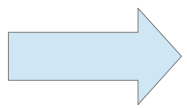
=> $S \bowtie_{A=B} R$: 100+100*400 = 40100 (ipv. 2000100) blokken transporteren

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

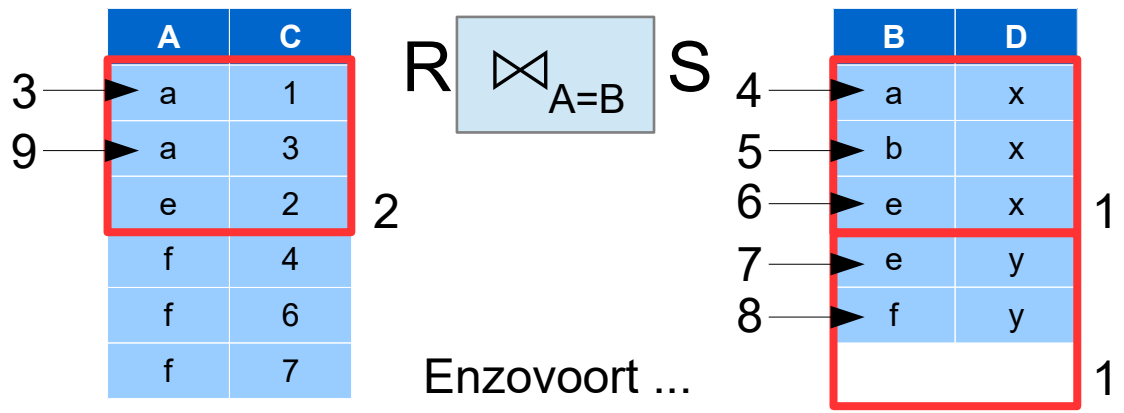
Blokken lezen – S past in geheugen

Voor elk blok b_R in R:
  Voor elk blok b_S in S:
    Voor elk record r in b_R:
      Voor elk record s in b_S:



Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Voor elk record s in S:

Voorbeeld: (3 rec/blok)
4 blokken in werkgeheugen:
* 3 in (1R&2S) + 1 out
* out niet in voorbeeld



Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren: S past in geheugen

* Notatie: b_R en b_S blokken, en n_R en n_S records, in R en S, resp.

Kost:

* Min. b_S+2 blokken nodig: 1+b_S IN (R & S) + 1 UIT ($R \bowtie_{A=B} S$)

=> Blokkentransport: b_R + b_S blokken

Voorbeeld: R – 10000 rec. in 400 blokken; S – 5000 rec. in 100 blokken

=> $R \bowtie_{A=B} S$: 400+100 = 500 (ipv. 40400 en 1000400) blokken transporteren

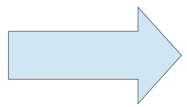
=> $S \bowtie_{A=B} R$: 100+400 = 500 (ipv. 40100 en 2000100) blokken transporteren

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

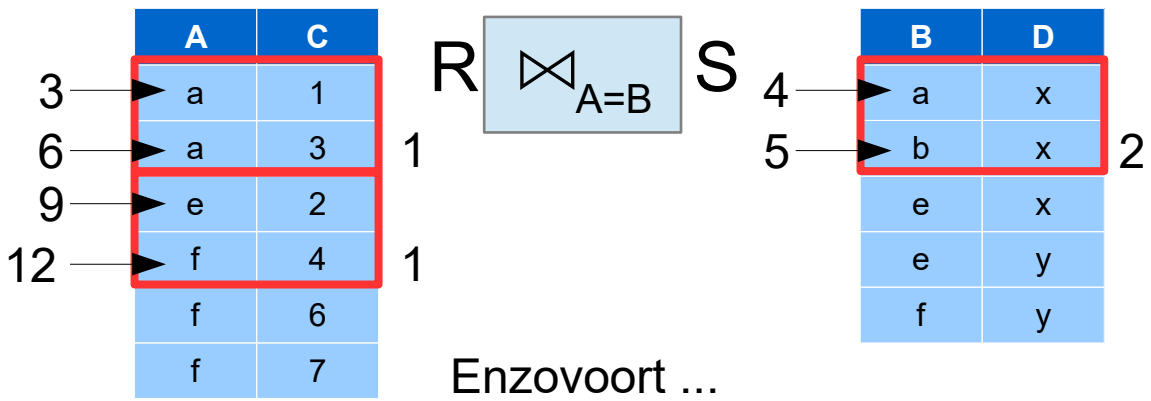
Blokken lezen – S past gedeeltelijk in geheugen

Voor elk blok b_R in R:
  Voor elk blok b_S in S:
    Voor elk record r in b_R:
      Voor elk record s in b_S:



Voor elk groep G van n blokken in R:
  Voor elk blok b_S in S:
    Voor elk record r in G:
      Voor elk record s in b_S:

Voorbeeld: (2 rec/blok)
4 blokken in werkgeheugen:
* 3 in (2G&1S) + 1 out
* out niet in voorbeeld



Enzovoort ...

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren: S past gedeeltelijk

* Notatie: b_R en b_S blokken, en n_R en n_S records, in R en S, resp.

Kost:

* Geheugen met b > 3 blokken: b_G+1 IN (R & S) + 1 UIT ($R \bowtie_{A=B} S$)

=> Blokkentransport: $b_R + (b_R/b_G)*b_S$ blokken

Voorbeeld: R – 10000 rec. in 400 blokken; S – 5000 rec. in 100 blokken

Geheugen met b = 50 blokken => $b_G = 48$

=> $R \bowtie_{A=B} S$: 400+(400/48)*100 = 1300 (ipv. 500 en 40400) blokken transporteren

=> $S \bowtie_{A=B} R$: 100+(100/48)*400 = 1300 (ipv. 500 en 40100) blokken transporteren

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

* Volgorde van relaties is belangrijk!
* Groote van geheugen is belangrijk!
* Veralgemeenbaar!
  - Elke soort Join-conditie
  - Mogelijks alle attributen

Wat met geïndexeerde attributen?

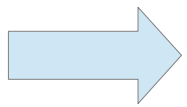
=> Equi-join (natural join) op een attribuut met index

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Index op join-attribuut v. S

Voor elk blok b_R in R:
  Voor elk blok b_S in S:
    Voor elk record r in b_R:
      Voor elk record s in b_S:



Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Find with index, s in S: r[A] = s[B]:

Kost:

Index steeds in binnenste lus!

=> Blokkentransport: $b_R + n_R * c$ blokken, met c kost om s in Index te vinden

* Goed voor queries met joins en selecties:

SELECT * FROM EMPLOYEE, DEPARTMENT WHERE Dnumber=5 AND Ssn=Mgr_Ssn

=> Slechts 1 EMPLOYEE record per departement

Impl. v. Operatoren – Join

Joins tot nu toe:

Geneste lussen zonder index: voor alle Join-condities
* Meest efficient: Kleinere relatie past in geheugen => br+bs transfers
* Binnenste lus heeft hoge CPU kost
* Meestal als:
  - Kleinere relatie heel klein is
  - Geneste lussen met index niet kan

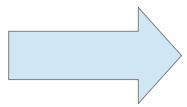
Geneste lussen met index: alleen als nodige index bestaat
* Efficientie afhankelijk van Index => br+bs*c, met c de indexkost
* Heel nuttig als selecties weinig records teruggeven (e.g., op sleutels)

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Hash Join (S past in geheugen => maak hash index)

Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Voor elk record s in S:



Voor elk blok b_R in R:
  Voor elk record r in b_R:
    Find with index, s in S: r[A] = s[B]:

Kost: Ook $b_R + b_S$, maar minder CPU kost

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Partition Hash Join (S noch R past in geheugen)

Fase 1: Partitioneer met hash-functie h_1

* Splits R in bestanden $R_1, R_2, ..., R_k$ via hash-functie h_1 op join-attribuut
* Splits S in bestanden $S_1, S_2, ..., S_k$ via hash-functie h_1 op join-attribuut
  - Conditie Voor i in 1,2,...,k: [2 + #blokken-in-S_i] past in het werkgeheugen

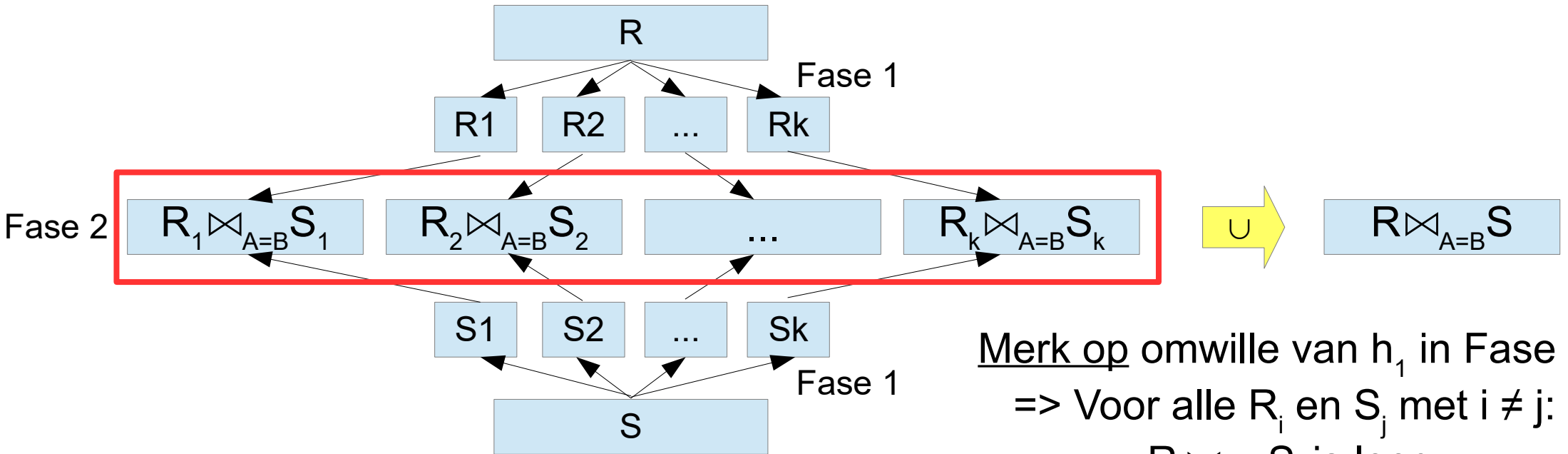
Fase 2: Zoek met hash-functie h_2

* Gebruik hash-join voor elke i in 1,2,...,k:
  - Lees S_i en bouw hash-index (met andere functie h_2)
  - Lees R_i per blok en gebruik hash-index om records in S_i te vinden

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Partition Hash Join (S noch R past in geheugen)



Merk op omwille van h_1 in Fase 1:
=> Voor alle R_i en S_j met $i \neq j$:
$R_i \bowtie_{A=B} S_j$ is leeg

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Partition Hash Join (S noch R past in geheugen)

Fase 1: Partitioneer met hash-functie h_1 (naar k tijdelijke bestanden)

Fase 2: Zoek met hash-functie h_2 (bereken k hash joins)

Kost: $3*(b_R+b_S)$

=> Fase 1: lezen en schrijven + Fase 2 lezen

- Veronderstelt ongeveer evenveel blokken onafhankelijk v. bestandsopdeling

* Past S_i in geheugen? Hash functie met voldoende range

- Gelijkmatige partitionering!

* Verder optimalisaties mogelijk, vb. Hybrid hash join

Impl. v. Operatoren – Join

Een Join, $R \bowtie_{A=B} S$, met geneste lussen uitvoeren:

Blokken lezen – Sort-Merge Join (R en S gesorteerd op join-attribuut)

Fase 1: Pre-sorteerstap indien nodig
Fase 2: Zelfde principe als bij extern sorteren: bekijk records in volgorde

Kost: $b_R + b_S$ (Als al gesorteerd)

=> Zelfs met nog sorteren (Fase 1) vaak goede oplossing
=> Gesorteerd resultaat maakt daaropvolgende operaties in de uitvoering van queries mogelijks makkelijker

Impl. v. Operatoren – Join

Overzicht Join Operatoren:

Geneste lussen zonder index: voor alle Join-condities

Geneste lussen met index: alleen als nodige index bestaat

(Hybrid) Hash Join: goed voor grote relaties

Sort-Merge Join: Veel gebruikt (relaties zijn vaak gesorteerd, of index)
=> Gesorteerde join resultaten zijn vaak nuttig in volgende stappen

Impl. v. Operatoren – Andere

Projectie: $\pi_{attribuutlijst}(R)$

Als attribuutlijst

* een sleutel van R bevat:
  - Resultaat bevat evenveel tupels als R
  - Overloop R, schrijf voor elk tupel r[attribuutlijst] na resultaat
* geen sleutel van R bevat:
  - Resultaat kan dubbels bevatten
  - Gebruik zelfde principe als eerste stap
  - Verwijder duplicaten

Impl. v. Operatoren – Andere

Projectie: $\pi_{attribuutlijst}(R)$

Duplicaten verwijderen: Best via sorteren (maar hash-index kan)

* Algoritme:

```
R' := Projecteer R volgens projectie-attributen
R'' := Sorteer R' op alle (projectie-)attributen
Prev := NULL
Voor alle records r in R'':
    Als r != Prev: schrijf r naar resultaat en Prev := r
```

Impl. v. Operatoren – Andere

Verzameling operaties:

* Cartesisch product R×S:
  – Heel duur, groot resultaat
  – Te vermijden (vervang tijdens optimalisatie als mogelijk)

* Unie, doorsnede, verschil
  – alleen voor relaties met zelfde attributen
  – implementatie:
    • sorteer beide relaties volgens zelfde attributen
    • doorloop in parallel en voeg samen
  – hashing gebruiken kan ook

Impl. v. Operatoren – Andere

Aggregaat operaties:

* Indien geen index op attribuut: doorlopen
* Indien dichte index op attribuut bestaat: resultaat uit index berekenen

Voorbeeld:

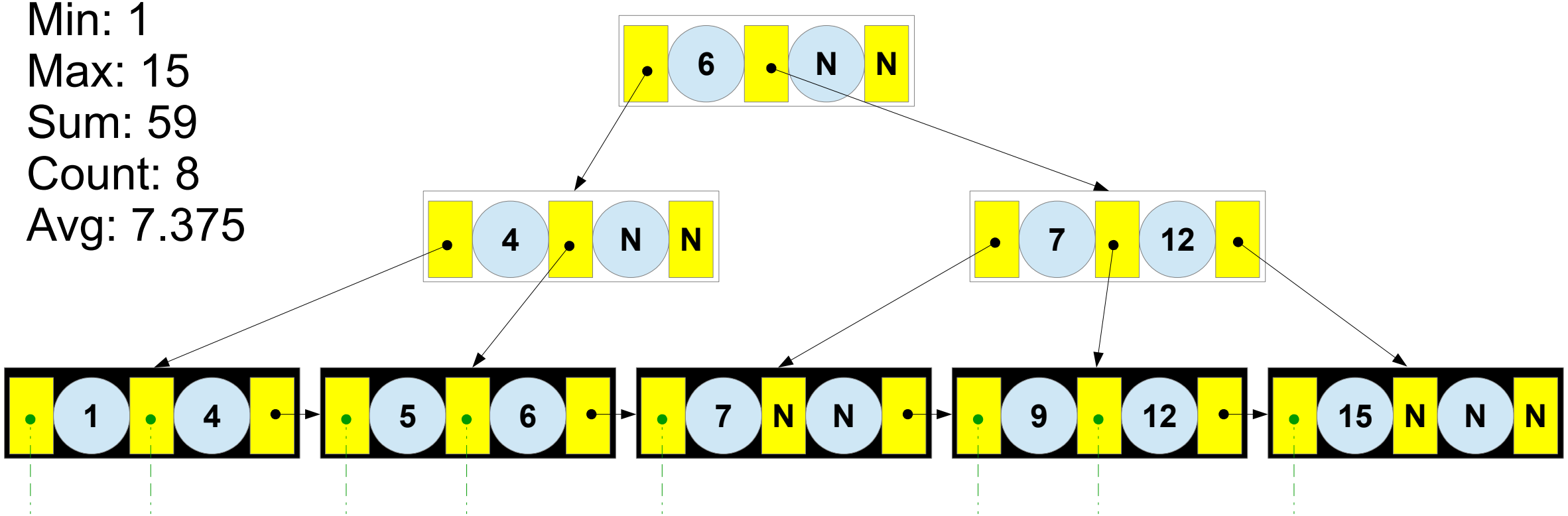
```
SELECT    MAX(Salary)
FROM      EMPLOYEE;
```

=> Minimum: Meest linkse tak van B^+-boom
=> Maximum: Meest rechtse tak van B^+-boom
=> Average, Count, Sum: Doorlop bladeren van B^+-boom

Impl. v. Operatoren – Andere

Voorbeeld: Aggregaten met B+-boom als index

Min: 1
Max: 15
Sum: 59
Count: 8
Avg: 7.375



Impl. v. Operatoren – Andere

Aggregaat operaties met GROUP BY:

Voorbeeld vorm: SELECT A, COUNT(B) FROM R GROUP BY A;

* Met hashing: maak hash-tabel op A en hou cnt(B) bij voor elke A

  Voor elk record r in R, ga na of r[A] in hash-tabel
  - Ja: incermenteer cnt(B)
  - Nee: hash-tabel uitbreiden en nieuwe cnt(B) initialiseren (op 1)

Impl. v. Operatoren – Andere

Aggregaat operaties met GROUP BY:

Voorbeeld vorm: SELECT A, COUNT(B) FROM R GROUP BY A;

* Met hashing: maak hash-tabel op A en hou cnt(B) bij voor elke A

* Met sorteren: sorteer R op A => Records binnen groep bij elkaar

    * Lees R in volgorde en bereken aggregaten (per groep)

=> Kunnen beide met alle soorten aggregaten
- Min, Max, Count, Sum: 1 variabele per groep; Avg: 2 per groep

Merk op: Een cluster-index op A deelt ook op in de nodige groepen.
Merk op: COUNT(DISTINCT R) => eerst duplicaten verwijderen

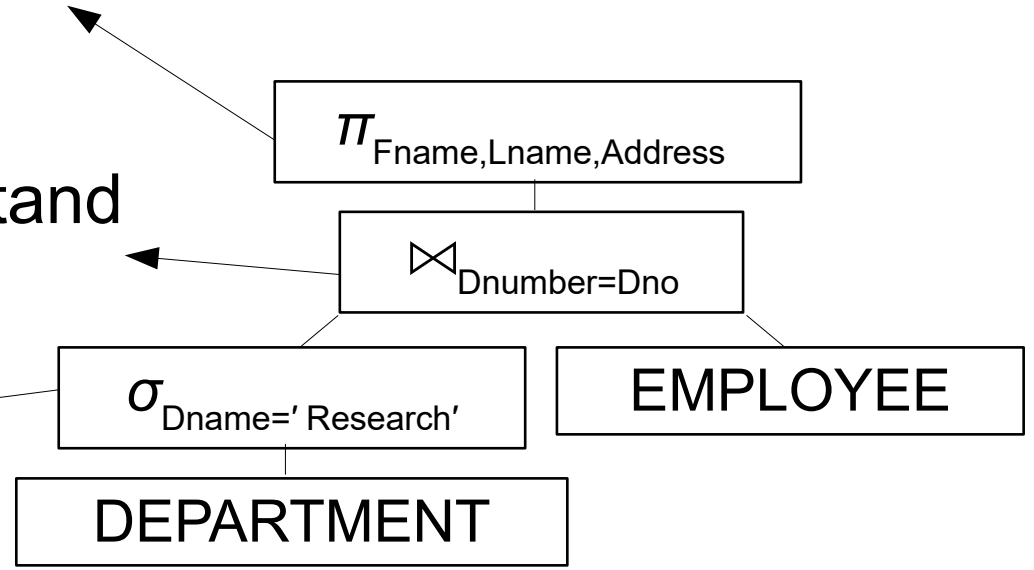
Voorbeeld Uivoeringsplan

3. Doorloop resultaat van join voor projectie

2. Gebruik geneste lussen voor join
   - Doorloop volledige EMPLOYEE bestand

Of als index op Dno: single-loop join

1. Gebruik index voor selectie op DEPARTMENT (als die bestaat)




```
SELECT   Fname, Lname, Address
FROM     EMPLOYEE, DEPARTMENT
WHERE    Dno = Dnumber AND Dname = 'Research';
```

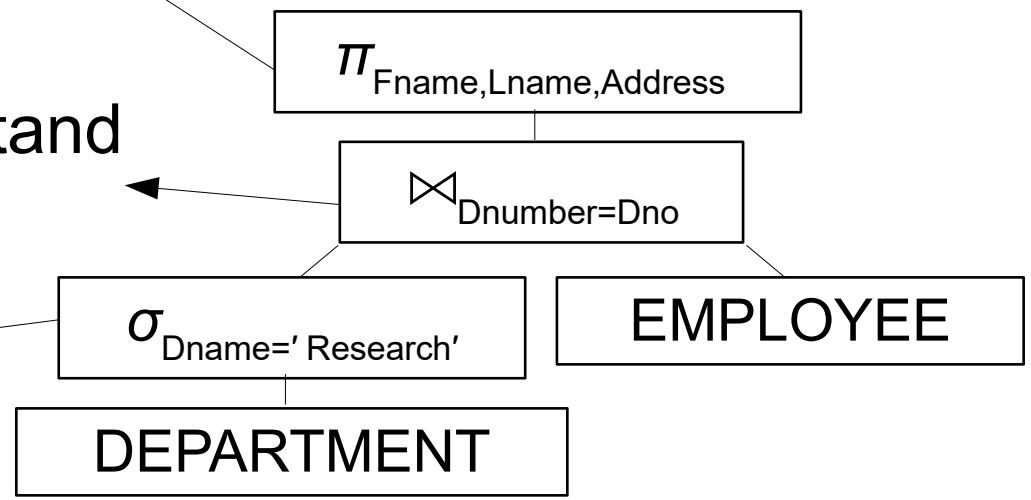
Voorbeeld Uivoeringsplan

3. Doorloop resultaat van join voor projectie

2. Gebruik geneste lussen voor join
   - Doorloop volledige EMPLOYEE bestand

Of als index op Dno: single-loop join

1. Gebruik index voor selectie op DEPARTMENT (als die bestaat)



Wat met tussenresultaten?

Overzicht

Queryverwerking bespreekt hoe een (R)DBMS queries behandelt en uitvoert op een zo optimaal mogelijke manier.
Hoofdstuk 18 – Oefenzitting 5

HC8 (Deel 1):
* Inleiding & Herhaling
* Queryverwerking -en optimalisatie
* Heuristische optimalisatie van querybomen

HC9 (Deel 2):
* Extern sorteren
* Implementaties van operatoren
* **Queryuitvoering**

Queryuitvoering

Gegeven bepaalde uitvoeringsstrategie:

Materialization:

* Evalueer operaties één per één
* Schrijf tussenrelatie (tijdelijke relatie) naar schijf
* Lees voor volgende operatie

Pipelining:

* Evalueer meerdere operatie simultaan
* Schrijf niet naar harde schijf
* Meestal sneller, maar meer geheugen nodig
* Niet altijd mogelijk (vb. Sort-Merge Join)

Queryuitvoer – Materialization

Tussenrelaties naar schijf schrijven kan altijd

* Tijdskost (lezen/schrijven bottleneck)

Optimalisatie: Dubbel bufferen (double buffering):

=> Gebruik twee buffers voor resultaat operatie

* Wanneer ene vol
  - Schrijf naar harde schijf
  - Gebruik andere vanaf beschikbaar om operatie verder te zetten

=> Snelheidswinst omdat evaluatie en schrijven samen kan

Queryuitvoer – Pipelining

Resultaten doorgeven aan volgende operatie

* Om lezen en schrijven te vermijden (bottleneck)
* Niet altijd mogelijk en meer geheugen nodig
* Bokkerende operaties: kunnen niet verder zonder volledige invoer

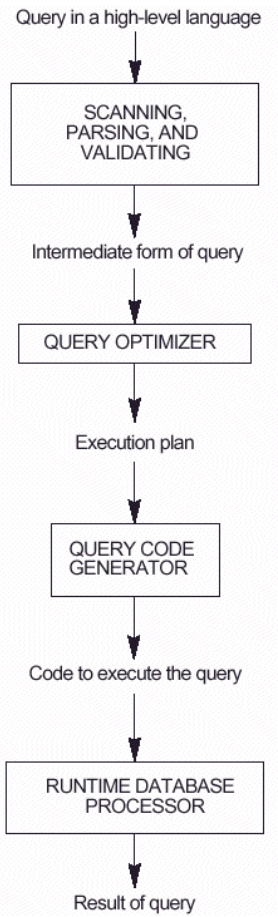
Operaties genereren records in resultaat terwijl input records toekomen:

* Selectie: Pipelining kan meestal
* Sorteren: Blokkeert
* Sort-Merge Join: De laatste merge-fase kan via pipelining
* Hash-Join: Partitioning blokkeert, tweede fase kan via pipelining
* Aggregaten: Hebben meestal volledige invoer nodig
* Duplicaten weghalen: Sorteren blokkeert
* Operatie op verzamelingen: Duplicaten weghalen blokkeert

Queryverwerking – Overzicht

Stappen in verwerking van query:

1. Lezen en ontleden van query:
   - Scanner => Lezen v. Query
   - Parser => Syntax + Voorstelling
2. Query optimalisatie
   - Genereer opties/strategieën
   - Kies meest efficiënte
3. Codegeneratie
   - Maak gekozen strategie uitvoerbaar
4. Voer uit



Optie 1: Interpreted
=> Directe uitvoering

Optie 2: Compiled
=> Code opgeslagen voor herbruik