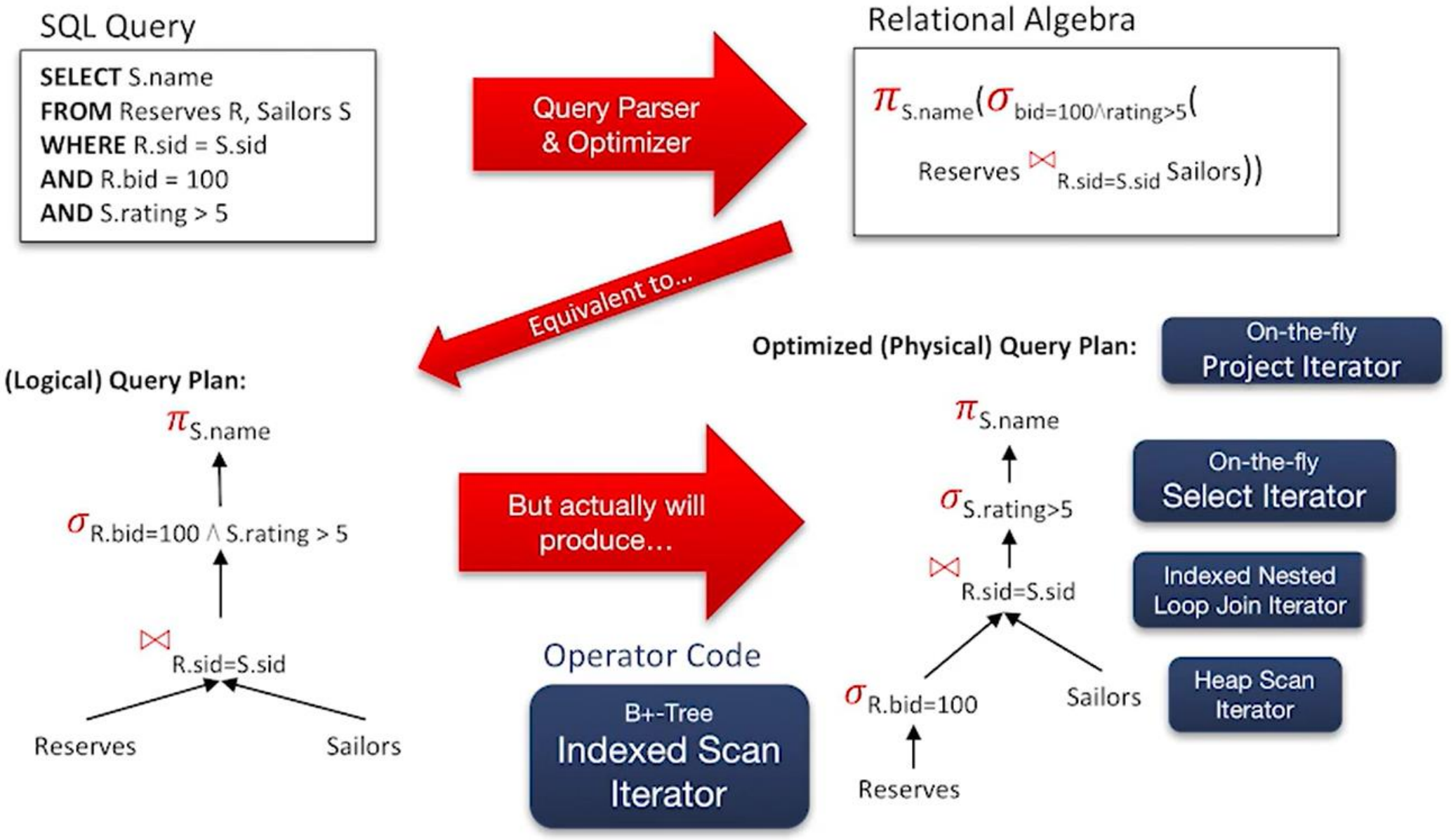
Algoritmi operatora i iteratori

Procesiranje upita



Od SQL-a do logičkog stabla

Relacioni račun

- Opisuje rezultat izračunavanja (upita)
  - Matematički “deklarativan jezik”
- Zasnovan na iskaznoj logici prvog reda
- Operandi su torke relacija
- Predstavlja osnovu SQL jezika

Relaciona algebra

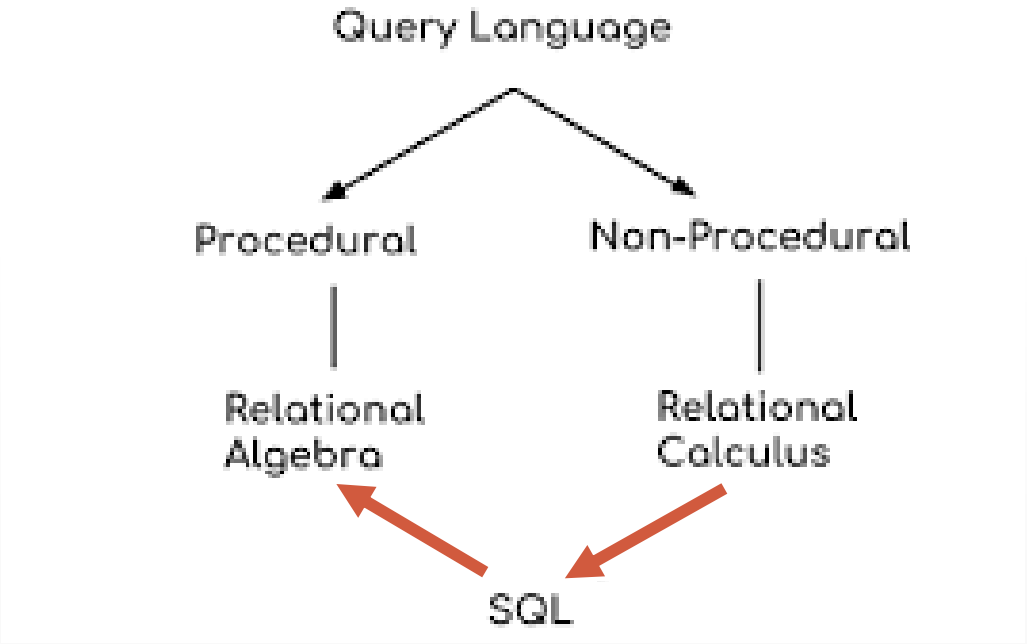
- Opisuje operacije kojima se ulazni transformišu u izlaze
  - Matematički “imperativni jezik”
- Specijalan slučaj algebre nad skupovima
  - Algebra nad instancama relacija
- Zatvorena nad skupom relacija
  - I operandi i rezultati su relacije
- Tipizirana
  - Šema rezulata zavisi od šeme operanadas

$$\{S.name | S \in Sailors\,, S.ratings > 5 \wedge (\exists R \in Reserves)\,(R.sid = S.sid \wedge R.bid = 100)$$

$$\pi_{S.name}\left(\sigma_{R.bid=100 \wedge S.raitings>5}(R \bowtie_{R.sid=S.sid})\right)$$

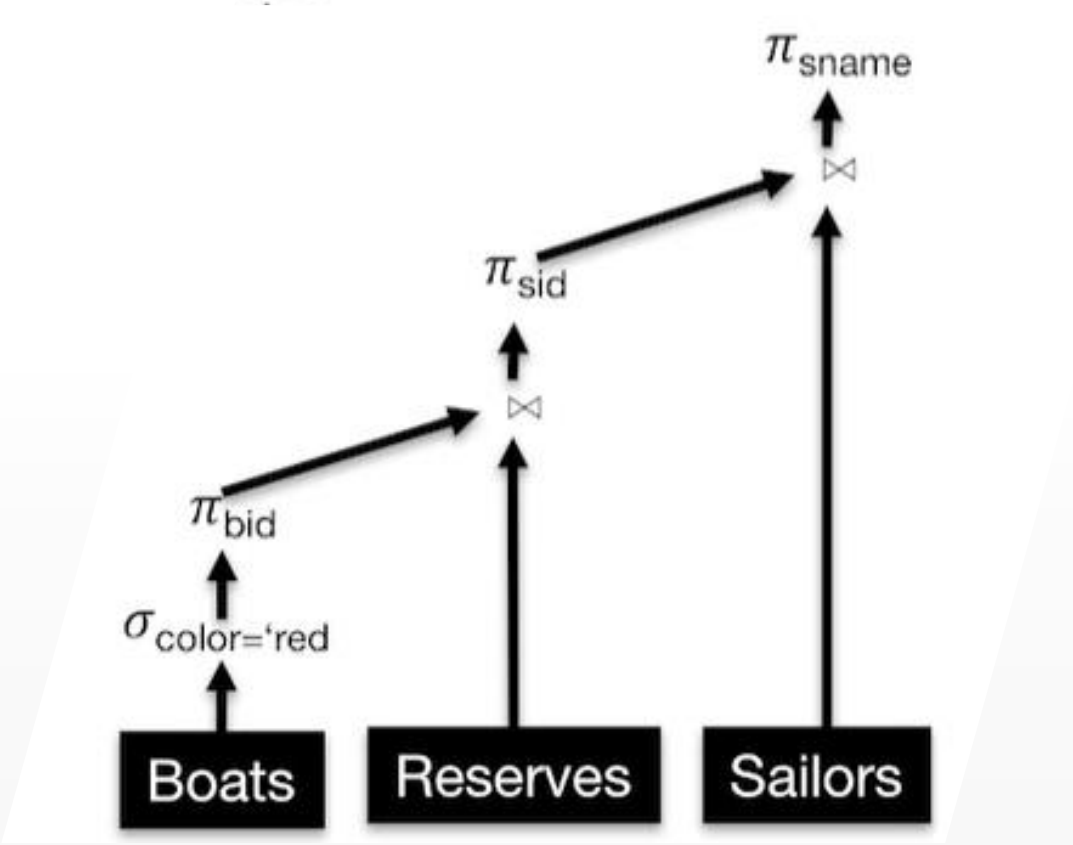
Od SQL-a do logičkog stabla

- E. Codd dokazao da je moguće svaki izraz relacionog računa iskazati operacijama relacione algebre
  - Povezao deklarativnu reprezentaciju upita sa operacionim opisom
- Zato je moguće prevesti SQL kod u izraz relacione algebre



Relacioni izraz i plan upita (query plan)

- Plan upita
  - Čvorovi – operacije relacione algebre
  - Listovi relacije
- Dataflow graph



Implementacija operatora

Tehnike koje se koriste pri implementaciji operatora su:

- **Indeksiranje** – na osnovu ideksa se vrši odvajanje onih torki koje će biti ispitivane pri realizaciji operacije
- **Iterisanje** – pretraga svih torki u tabeli, ili celog indeksa ukoliko sadrži sve atribute koji se ispituju
- **Particionisanje** – particionisanjem torki prema ključu se operacija razbija na kolekciju operacija nad manjim skupom podataka, uobičajene tehnike koje koriste particionisanje su **sortiranje** i **heširanje**

Uređivanje operanda ili rezultata

Sortiranje i heširanje

Poreba za uređivanjem

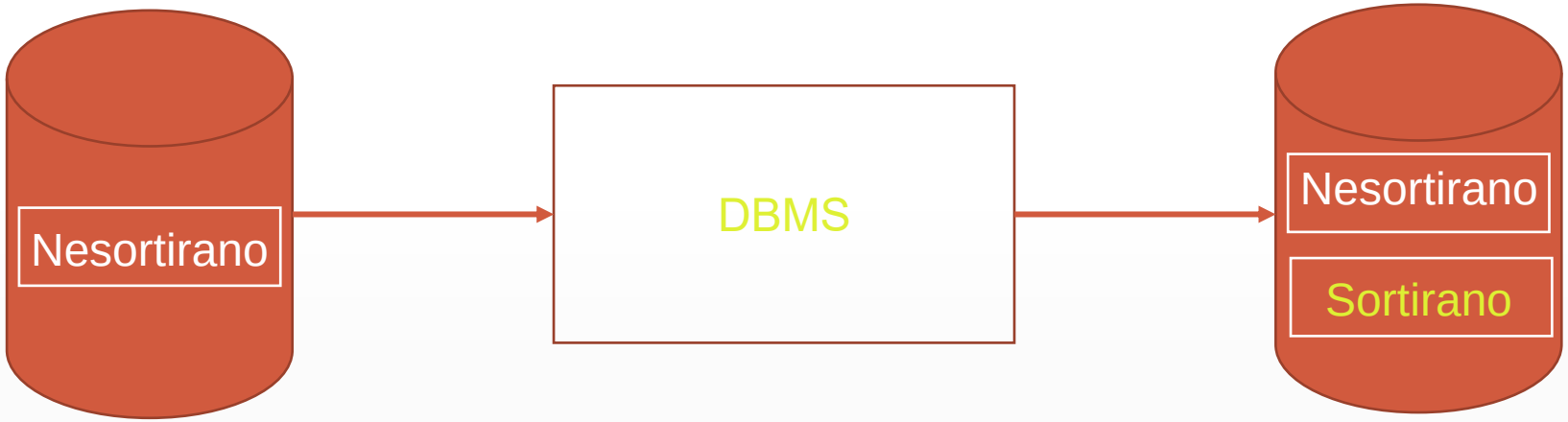
- **Indirektni zahtevi (zahtevano algoritmom neke operacije)**
  - DISTINCT
  - GROUP BY
  - Neke vrste JOIN algoritama (sort-merge)
- **Eksplicitni zahtev za sortiranjem**
  - ORDER BY
  - Kreiranje privremenih indeksa nad neuređenim slogovima (bulk-loading tree indexes)
- Kako sortirati ako je tabela veća od raspoloživog RAM prostora?

Sortiranje i heširanje

- Ulazni fajl F:
  - Koji sadrži slogove relacije R
  - Zauzima N blokova
- U RAM-u postoji prostor za fiksan broj blokova, B

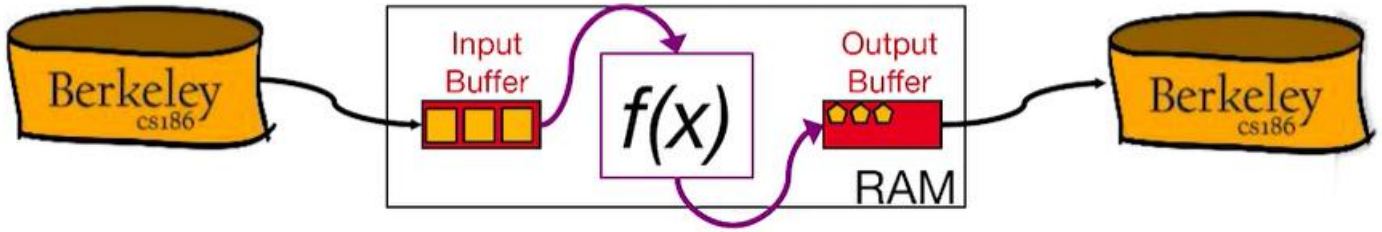
- **Sortiranje**
  - Daje izlazni fajl F_s koji sadrži sortirane torke polazne relacije
- **Heširanje**
  - Daje izlazni fajl F_h u kom su slogovi sa istom heš vrednošću spakovani jedan za drugim.

Osnovni zadatak



I/O bafer strimovi

Jednostruki bafer tok
(single-pass streaming)

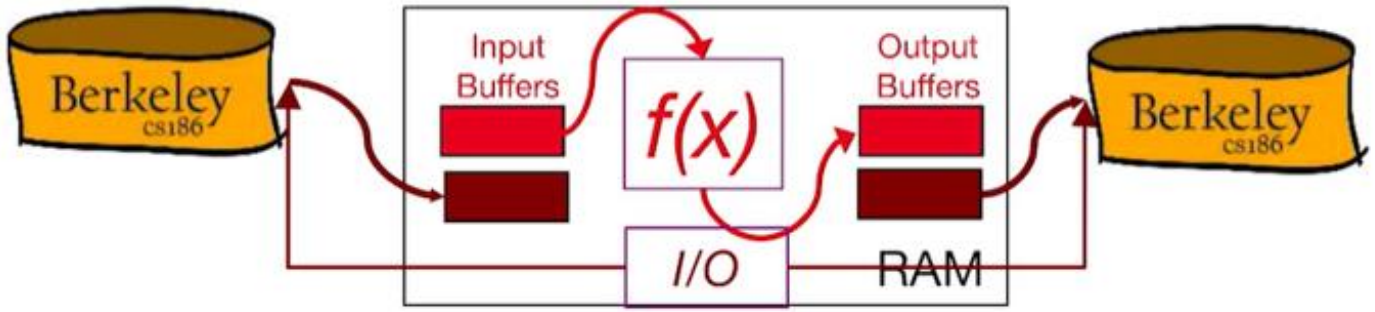


Approach

- Read a chunk from INPUT to an Input Buffer
- Write f(x) for each item to an Output Buffer
- When Input Buffer is consumed, read another chunk
- When Output Buffer fills, write it to OUTPUT

I/O bafer strimovi

Dupli bafer tokovi
(double buffering)



- **Main thread** runs f(x) on one pair I/O bufs
- 2nd **I/O thread** drains/fills unused I/O bufs in parallel
  - Why is parallelism available?
  - Theme: I/O handling usually deserves its own thread
- Main thread ready for a new buf? Swap!

Sortiranje

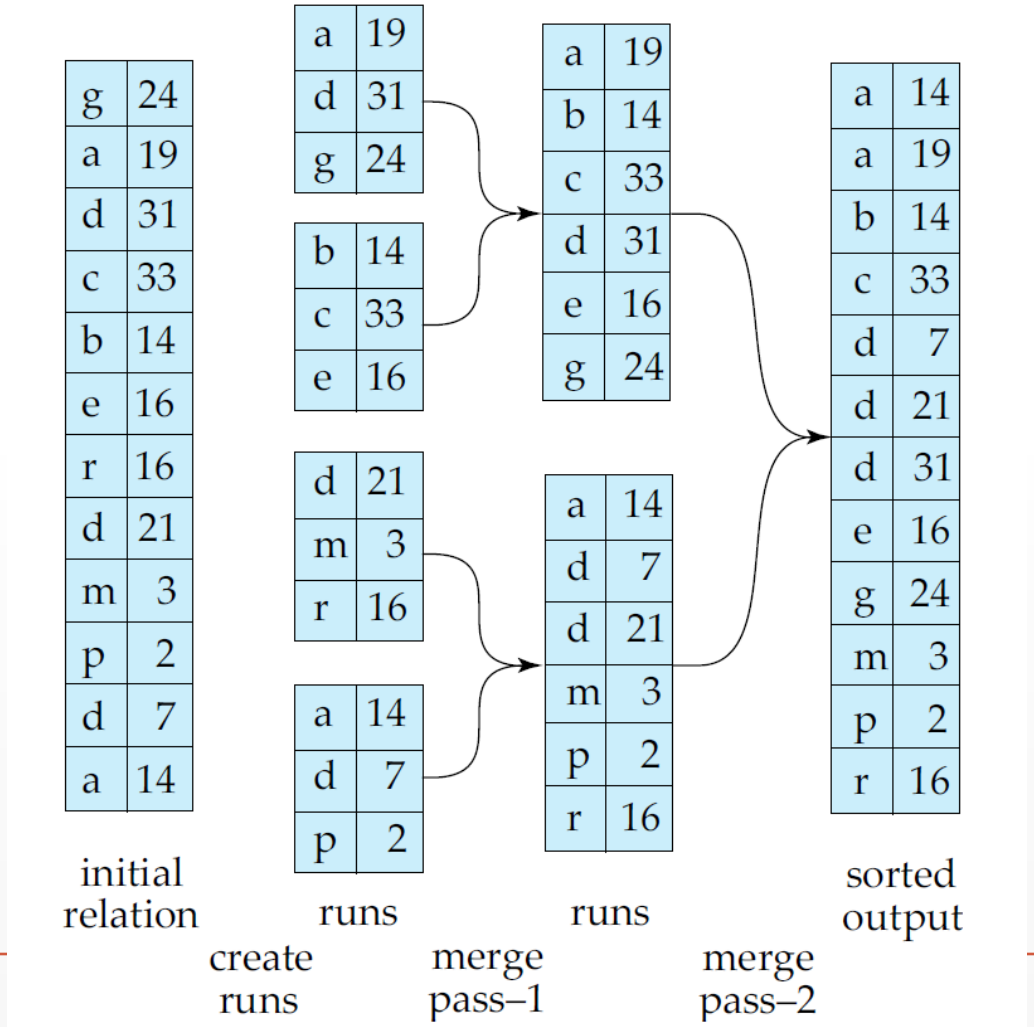
Ako je B < N:

- Single-pass streaming
- Divide and Conquer – External Sort Merge

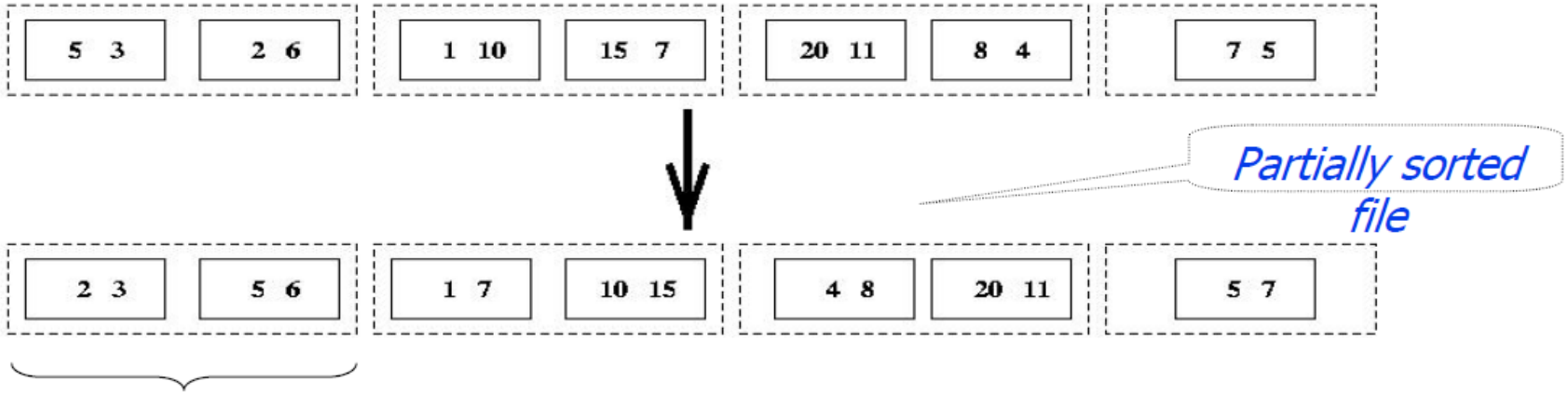
Sortiranje - External Sort-Merge

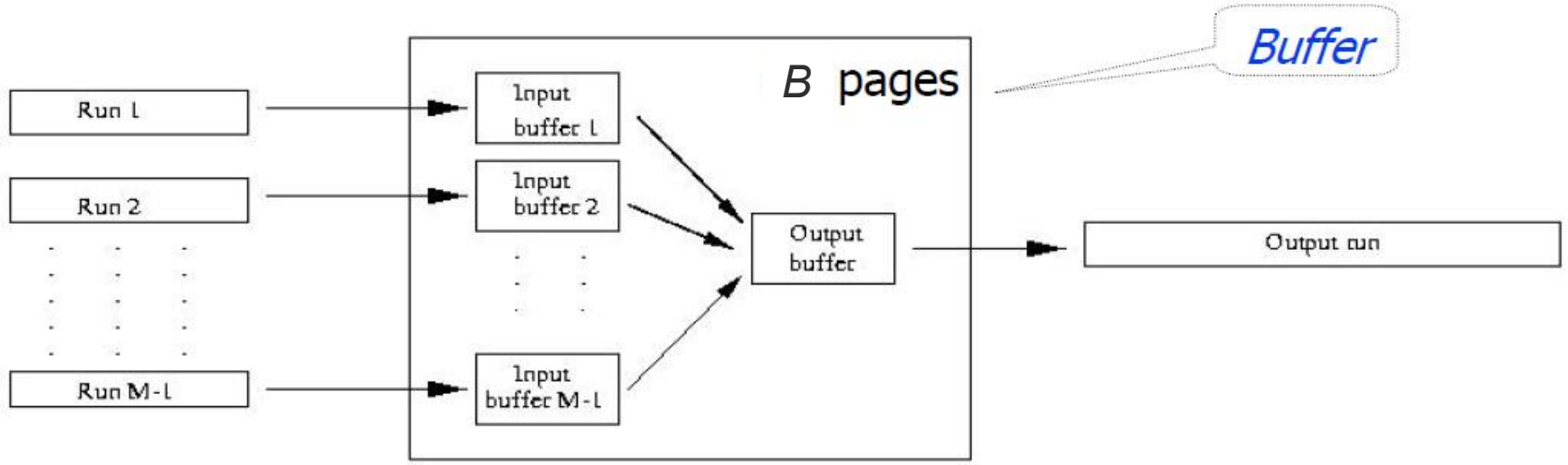
- Najčešće veličina fajla ne dozvoljava njegovo učitavanje u celosti i sortiranje na licu mesta i odmah, već se vrši u nekoliko koraka sa beleženjem sortiranih međurezultata.
- Osnovna ideja – podeliti na delove, delove sortirati, sortirane spajati u veće sortirane delove, sve dok rezultat spajanja ne bude cela tebela.
- Run (prolaz) – sortirani međurezultat
- Dve faze:
  - **Faza delimičnog sortiranja**: sortiranje B stranica odjednom; kreiranje N/B sortirana prolaza.
  - **Faza spajanja**: spajanje više prolaza u jedan koristeći B-1 bafera za ulaz i 1 izlazni bafer

External Sort-Merge

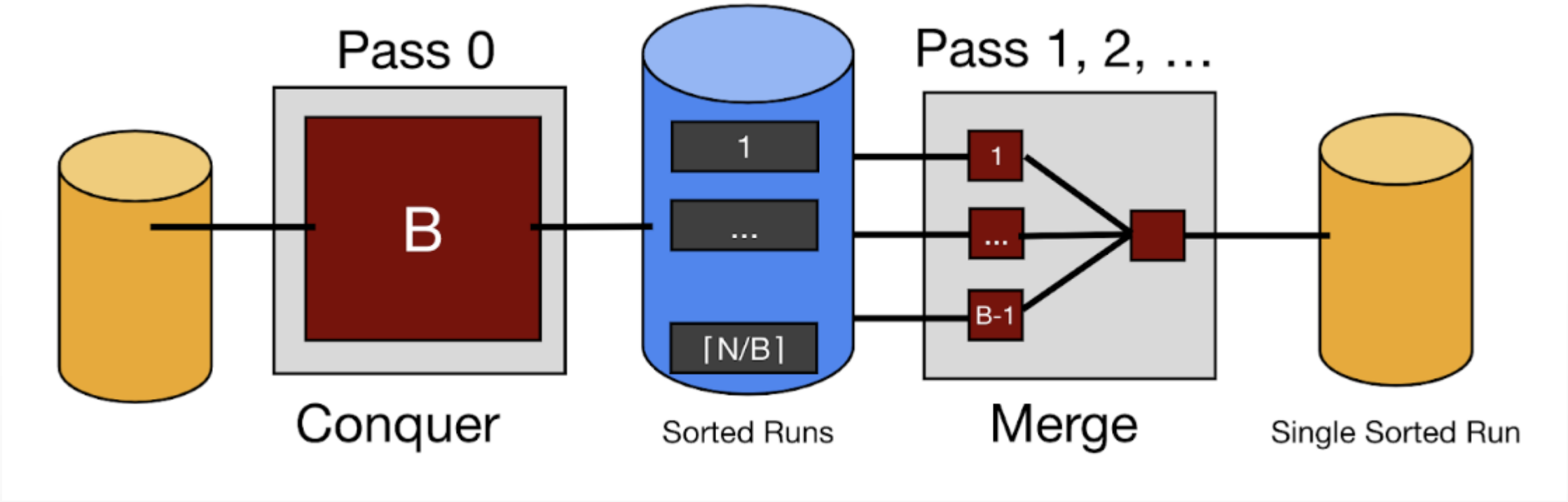


General External Sort-Merge





General External Sort-Merge



General External Sort-Merge Cost

- Number of passes: $1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$
- Total I/Os = (I/Os per pass) * (# of passes) = 2*N * ($1 + \lceil \log_{B-1} \lceil N / B \rceil \rceil$)

- E.g., with 5 buffer pages, to sort 108 page file:
  - Pass 0: $\lceil 108 / 5 \rceil$ = 22 sorted runs of 5 pages each
    - last run is only 3 pages
  - Pass 1: $\lceil 22 / 4 \rceil$ = 6 sorted runs of 20 pages each
    - last run is only 8 pages
  - Pass 2: $\lceil 6 / 4 \rceil$ 2 sorted runs, 80 pages and 28 pages
  - Pass 3: Sorted file of 108 pages

Formula check: $1 + \lceil \log_4 22 \rceil = 1+3$ → 4 passes √

General External Sort-Merge Cost

of Passes of External Sort

(Total I/O is 2N * # of passes)

N	B=3	B=5	B=9	B=17	B=129	B=257
100	7	4	3	2	1	1
1,000	10	5	4	3	2	2
10,000	13	7	5	4	2	2
100,000	17	9	6	5	3	3
1,000,000	20	10	7	5	3	3
10,000,000	23	12	8	6	4	3
100,000,000	26	14	9	7	4	4
1,000,000,000	30	15	10	8	5	4

Heširanje

Alternative: Hashing

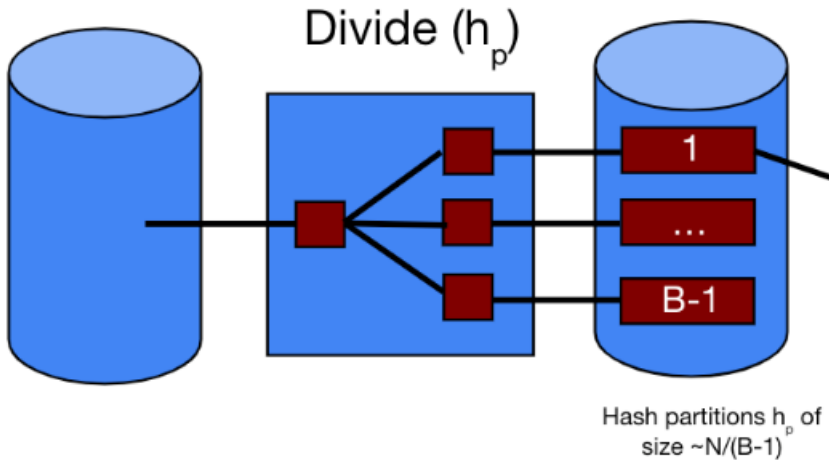
- Idea:
  - Many times we don’t require order
    - E.g., remove duplicates, form groups
- Often just need to rendezvous matches

Heširanje - strategija

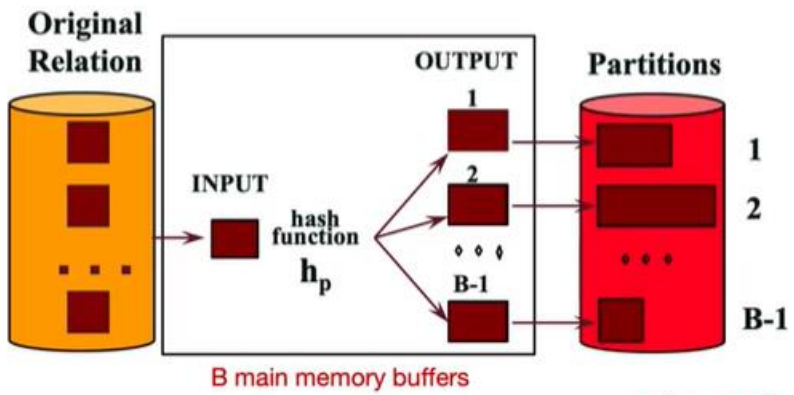
Divide

- Streaming Partition (divide):
  Use a hash function $\mathbf{h_p}$ to stream records to disk partitions
  - All matches rendezvous in the same partition.
  - Each partition a mix of values
  - Streaming algorithm to create partitions on disk:
    - “Spill” partitions to disk via output buffers

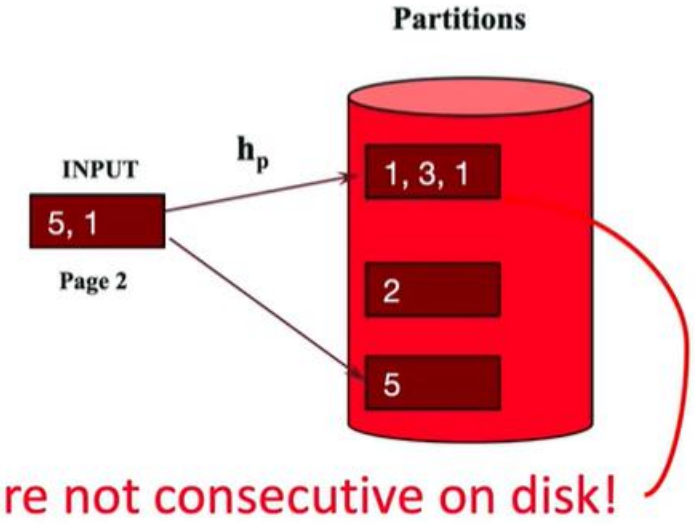
Heširanje - strategija



- Partition: (Divide)



Example



The 1's are not consecutive on disk!

Heširanje - strategija

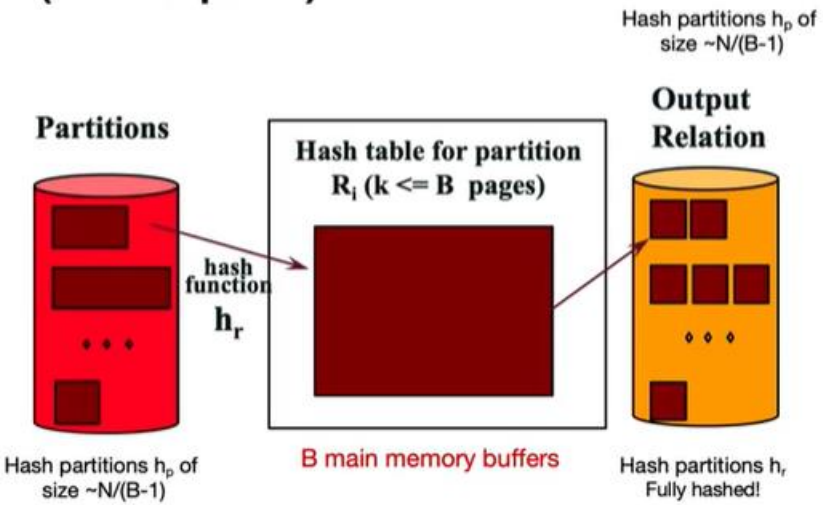
Conquer

- ReHash (conquer):
  Read partitions into RAM hash table one at a time, using *different* hash function $\mathbf{h_r}$
  - Each bucket contains a small number of distinct values
- Then read out the RAM hash table buckets and write to disk
  - Ensuring that duplicate values are contiguous

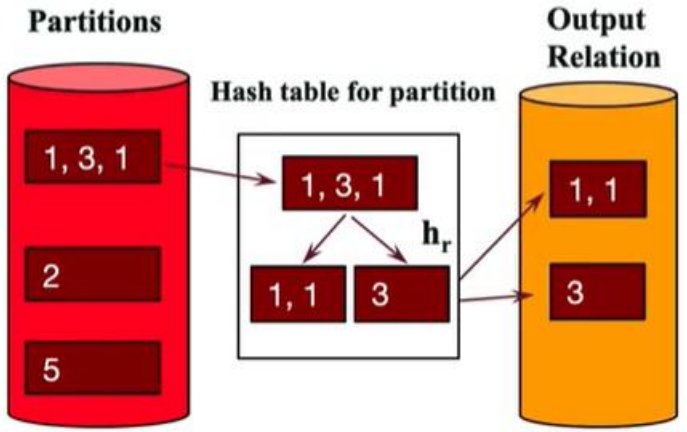
Heširanje - strategija

Two Phases: Conquer

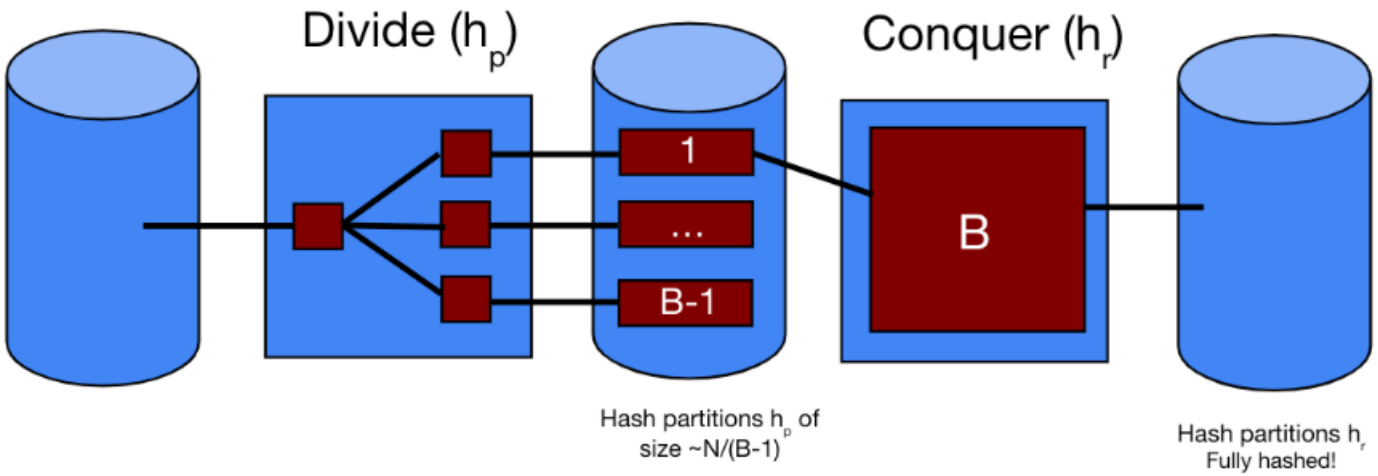
- Rehash: (Conquer)



Example



Heširanje - strategija



Heširanje - strategija

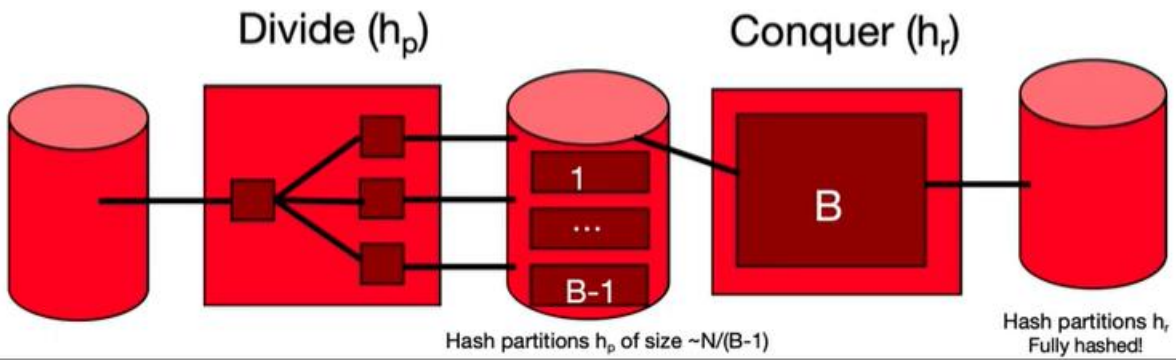
Cost of External Hashing

Total I/Os ~ 2*N*(# passes) = 4*N
(includes initial read, final write)

Heširanje - strategija

Memory Requirement

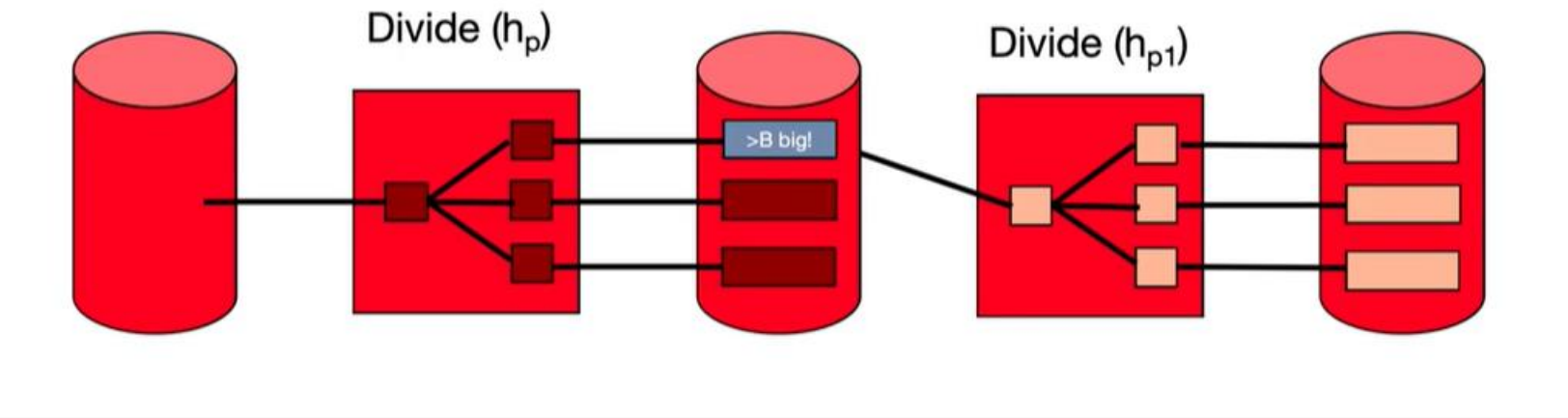
- How big of a table can we hash in exactly two passes?
  - B-1 "partitions" result from Pass 1
  - Each should be no more than B pages in size
  - Answer: B(B-1) ~ B^2
    - We can hash a table of size X in about $B = \sqrt{X}$ space (if we run only 2 passes)
  - Note: assumes hash function distributes records evenly!
- Have a bigger table? Recursive partitioning!



Cost of External Hashing

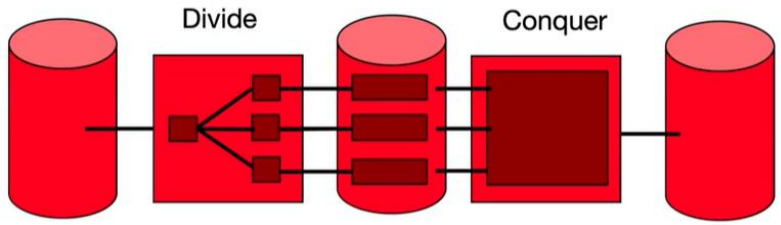
Total I/Os ~ 2*N*(# passes) = 4*N
(includes initial read, final write)

Heširanje – rekurzivno particionisanje



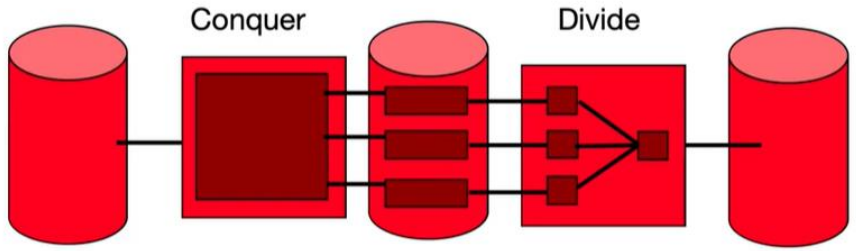
Heširanje - strategija

Cost of External Hashing



cost ~ 4*N I/Os
(including initial read, final write)
(case for 1 divide pass and 1 conquer pass only. Not the general formula!)

Cost of External Sorting



cost = 4*N I/Os
(including initial read, final write)
(case for 1 divide pass and 1 conquer pass only. Not the general formula!)

JOIN algoritmi

Baze podataka 2

2022/23

Algoritmi implementacije operacija - Spajanje

Nested Loop Join
→ Simple
→ Block
→ Index

Sort-Merge Join

Hash Join

SIMPLE NESTED LOOP JOIN

```
foreach tuple r ∈ R:        ← Outer
  foreach tuple s ∈ S:      ← Inner
    emit, if r and s match
```

Cost: $M + (m \cdot N)$

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

SIMPLE NESTED LOOP JOIN

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

Cost Analysis:

→ $M + (m \cdot N) = 1000 + (100000 \cdot 500) =$ **50,000,100 IOs**

→ At 0.1 ms/IO, Total time ≈ 1.3 hours

What if smaller table (S) is used as the outer table?

→ $N + (n \cdot M) = 500 + (40000 \cdot 1000) =$ **40,000,500 Ios**

→ At 0.1 ms/IO, Total time ≈ 1.1 hours

BLOCK NESTED LOOP JOIN

```
foreach block B_R ∈ R:
  foreach block B_S ∈ S:
    foreach tuple r ∈ B_R:
      foreach tuple s ∈ B_S:
        emit, if r and s match
```

This algorithm performs fewer disk accesses.

→ For every block in **R**, it scans **S** once

Cost: $M + (M \cdot N)$

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

M pages
m tuples

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

N pages
n tuples

MELLON
SE GROUP

Which one should be the outer table?

→ The smaller table in terms of # of pages

BLOCK NESTED LOOP JOIN

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

Cost Analysis:

→ $M + (M \cdot N) = 1000 + (1000 \cdot 500) =$ **501,000 IOs**

→ At 0.1 ms/IO, Total time ≈ 50 seconds

BLOCK NESTED LOOP JOIN

What if we have ***B*** buffers available?

→ Use ***B*-2** buffers for scanning the outer table.

→ Use one buffer for the inner table, one buffer for storing output.

```
foreach B - 2 blocks b_R ∈ R:
  foreach block b_S ∈ S:
    foreach tuple r ∈ b_R:
      foreach tuple s ∈ b_S:
        emit, if r and s match
```

This algorithm uses ***B*-2** buffers for scanning **R**.

Cost: $M + (\lceil M / (B-2) \rceil \cdot N)$

What if the outer relation completely fits in memory ($B > M+2$)?

→ **Cost:** $M + N = 1000 + 500 =$ **1500 IOs**

→ At 0.1ms/IO, Total time ≈ 0.15 seconds

INDEX NESTED LOOP JOIN

Use an index to find inner table matches.
→ We could use an existing index for the join.
→ Or even build one on the fly.

```
foreach tuple r ∈ R:
  foreach tuple s ∈ Index(r_i = s_j):
    emit, if r and s match
```

Assume the cost of each index probe is some constant C per tuple.

Cost: $M + (m \cdot C)$

NESTED LOOP JOIN

Pick the smaller table as the outer table.

Buffer as much of the outer table in memory as possible.

Loop over the inner table or use an index.

SORT-MERGE JOIN

Phase #1: Sort

→ Sort both tables on the join key(s).
→ Can use the external merge sort algorithm that we talked about last class.

Phase #2: Merge

→ Step through the two sorted tables in parallel, and emit matching tuples.
→ May need to backtrack depending on the join type.

SORT-MERGE JOIN

```
SELECT R.id, S.cdate
  FROM R, S
 WHERE R.id = S.id
   AND S.value > 100
```

R(id,name)

id	name
600	MethodMan
200	GZA
100	Andy
300	ODB
500	RZA
700	Ghostface
400	Raekwon

Sort!

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
500	7777	10/9/2018
400	6666	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018

Sort!

R(id,name)

id	name
100	Andy
200	GZA
300	ODB
400	Raekwon
500	RZA
600	MethodMan
700	Ghostface

S(id,value,cdate)

id	value	cdate
100	2222	10/9/2018
100	9999	10/9/2018
200	8888	10/9/2018
400	6666	10/9/2018
500	7777	10/9/2018

Output Buffer

R.id	R.name	S.id	S.value	S.cdate
100	Andy	100	2222	10/9/2018
100	Andy	100	9999	10/9/2018

SORT-MERGE JOIN

Sort Cost (R): $2M \cdot (\log M / \log B)$

Sort Cost (S): $2N \cdot (\log N / \log B)$

Merge Cost: $(M + N)$

Total Cost: Sort + Merge

SORT-MERGE JOIN

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

With 100 buffer pages, both **R** and **S** can be sorted in two passes:

→ Sort Cost (**R**) = $2000 \cdot (\log 1000 / \log 100)$ = **3000 IOs**

→ Sort Cost (**S**) = $1000 \cdot (\log 500 / \log 100)$ = **1350 IOs**

→ Merge Cost = $(1000 + 500)$ = **1500 IOs**

→ Total Cost = $3000 + 1350 + 1500$ = **5850 IOs**

→ At 0.1 ms/IO, Total time ≈ 0.59 seconds

HASH JOIN

Phase #1: Build

→ Scan the outer relation and populate a hash table using the hash function h_1 on the join attributes.

Phase #2: Probe

→ Scan the inner relation and use h_1 on each tuple to jump to a location in the hash table and find a matching tuple.

HASH JOIN

Cost of hash join?

→ Assume that we have enough buffers.

→ Cost: $3(M + N)$

Partitioning Phase:

→ Read+Write both tables

→ $2(M+N)$ IOs

Probing Phase:

→ Read both tables

→ M+N IOs

Example database:

→ $M = 1000$, $m = 100{,}000$

→ $N = 500$, $n = 40{,}000$

Cost Analysis:

→ $3 \cdot (M + N) = 3 \cdot (1000 + 500) =$ **4,500 IOs**

→ At 0.1 ms/IO, Total time ≈ 0.45 seconds

JOIN ALGORITHMS: SUMMARY

Algorithm	IO Cost	Example
Simple Nested Loop Join	$M + (m \cdot N)$	1.3 hours
Block Nested Loop Join	$M + (M \cdot N)$	50 seconds
Index Nested Loop Join	$M + (m \cdot C)$	~20 seconds
Sort-Merge Join	$M + N +$ (sort cost)	0.59 seconds
Hash Join	$3(M + N)$	0.45 seconds

Hashing is almost always better than sorting for operator execution.

Caveats:

→ Sorting is better on non-uniform data.
→ Sorting is better when result needs to be sorted.

Projekcija, selekcija i čitanje

Algoritmi implementacije operacija - Selekcija

$$\sigma_{R.attr\ \mathbf{op}\ value}(R)$$

- Ukoliko nema indeksa nad R.attr skenira se cela R
- Ukoliko postoji indeks u zavisnosti od uslova i toga da li je indeks klasterovan moguće je:
  - Suziti pretragu na manji broj strana
  - Dobiti direktne reference na torke koje se nalaze u uzastopnim stranama
  - Dobiti direktne reference u neklasterisanom indeksu (ako je broj torki koje treba dobaviti veci od 5%, tada je obična sekvencijala pretraga bolji izbor od pretrage po ključu)

Algoritmi implementacije operacija - Projekcija

- Projekcija bez eliminacije duplikata – jednostavno sekvencijalno učitavanje fajla sa podacima ili indeksa ako sadrži sve potrebne atribute
- Projekcija sa eliminacijom – primenjuje se neka od tehnika particionisanja

Pretraga tabela (*file scan*)

- Tri osnovne tehnike pretrage zajedničke skoro svim operatorima
  - Pretraga putem indeksa (*index scan*)
  - Iterativna/sekvencijalna pretraga fajla (*table scan*)
  - Patricionisanje torki po ključu pretrage.
    Standardne tehnike - sortiranje i heširanje.
  - *Sort-scan* – kada je potrebno dobiti sortirane podatke sa diska; više načina
    - ako postoji indeks po atributu sortitranja
    - ako je tabela dovoljno mala – sortira se po učitavanju memoriju
    - ako je tabela velika - merge-sort algoritam
- Način na koji se dobavljaju torke iz fajla - *access path/method*.
- Access path značajno utiče na cenu izvršavanja svih operatora.

Pristup podacima (access paths)

Da bi bio upotrebljen za pristup torkama indeks mora da odgovara uslovu selekcije:

- **Heš indeks** odgovara uslovu (u KNF) ako postoji term oblika **atribut = vrednost** za svaki atribut u ključu pretrage samog indeksa.
- **Uređeni indeks** odgovara uslovu ako postoji term oblika **atribut operacija vrednost** za svaki atribut nekog prefiksa ključa nad kojim je definisan indeks (prefiksi ključa (a,b,c) su (a) i (a,b))

Izvršavanje planova upita

Modeli procesiranja

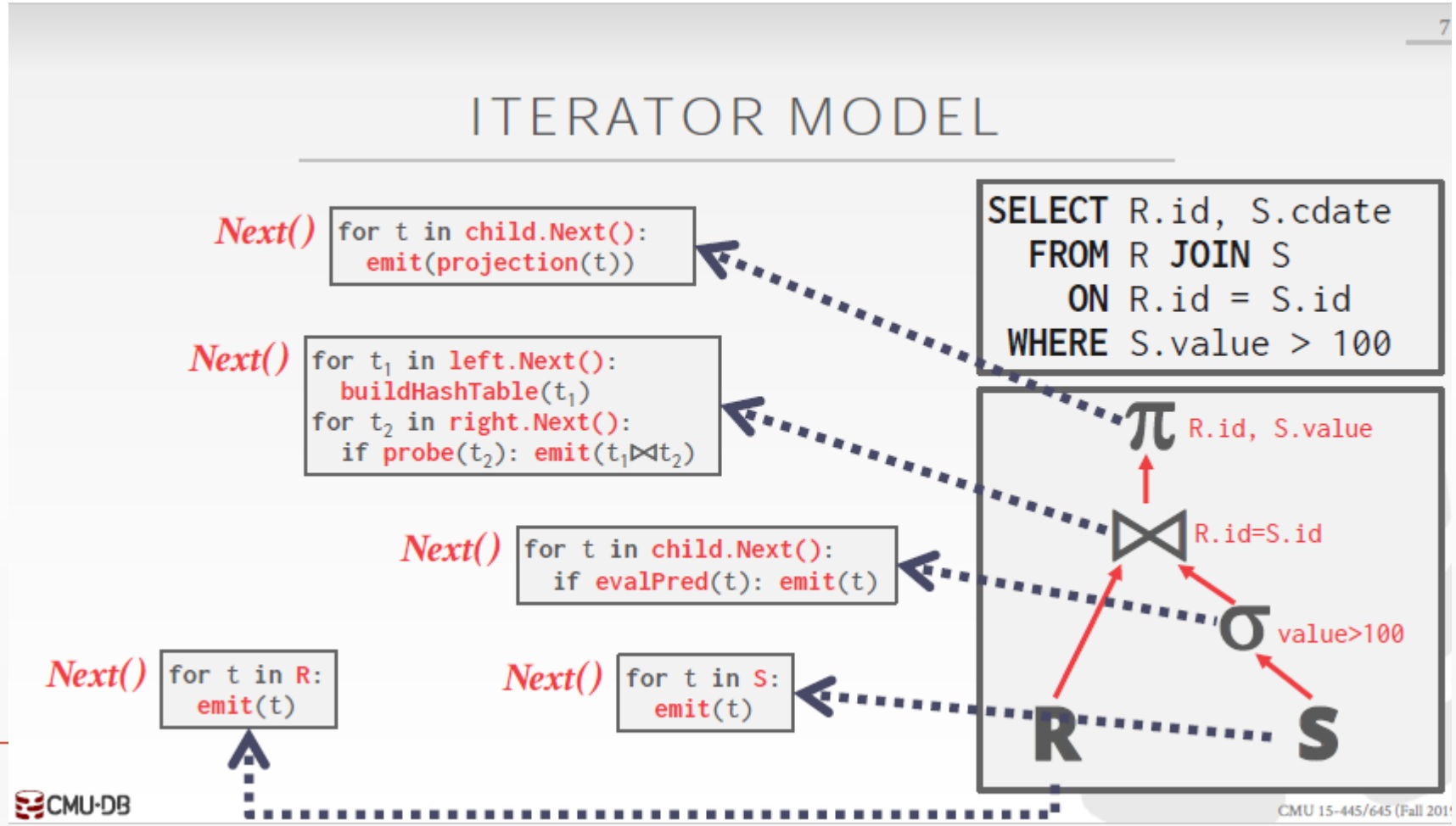
- Kako se izvršava fizički plan upita? Kada se vrši sinhronicazija operacija i prosleđivanje međurezultata?
- Iterisanje
- Materijalizacija
- Batch model

Iteratori za implementaciju fizičkih operatora

- Svaki operator implementira interfejs Iterator.
- Metodi *Iterator*-a:
  - Open()
    - Inicijalizuje stanje iteratora, tj. sve potrebne strukture podataka
    - Setuje parametre, npr. uslov selekcije
  - Get_next()
    - Izvršava se nad ulaznim podacima
    - Izvodi potrebno procesiranja ulaznih podataka i proizvodi izlaz
  - Close()
    - Oslobađa zauzete resurse
- Mogu biti: **Blokirajući** – vraćaju kolekcije torki; **Neblokirajući** – vraćaju jedan po jedan zapis.

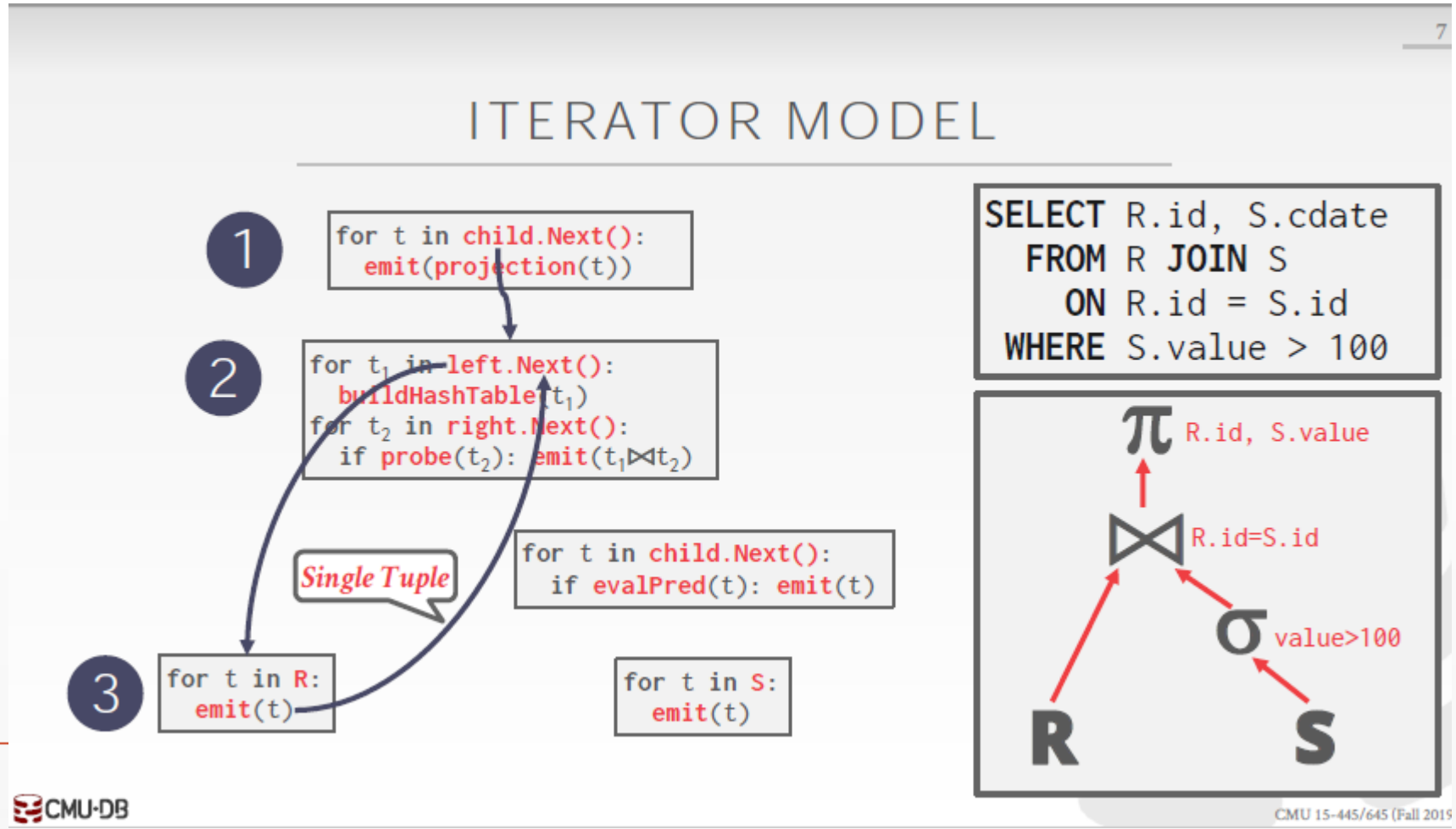
Iteratori (procesiranje iterisanjem)

Volcano ili Pipeline Model
Za neblokirajuće operatore



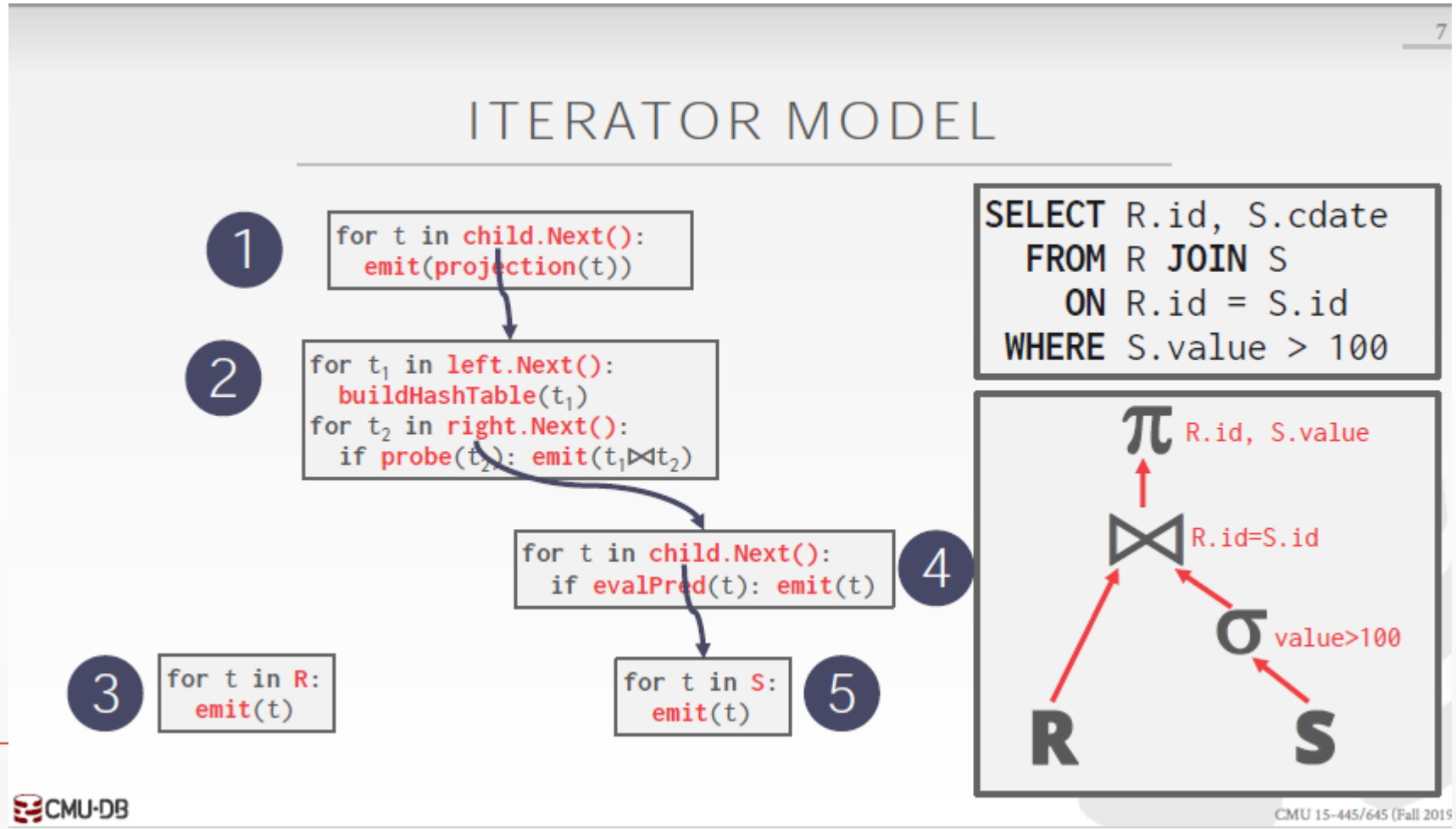
Neblokirajući iteratori (procesiranje iterisanjem)

Volcano ili Pipeline Model



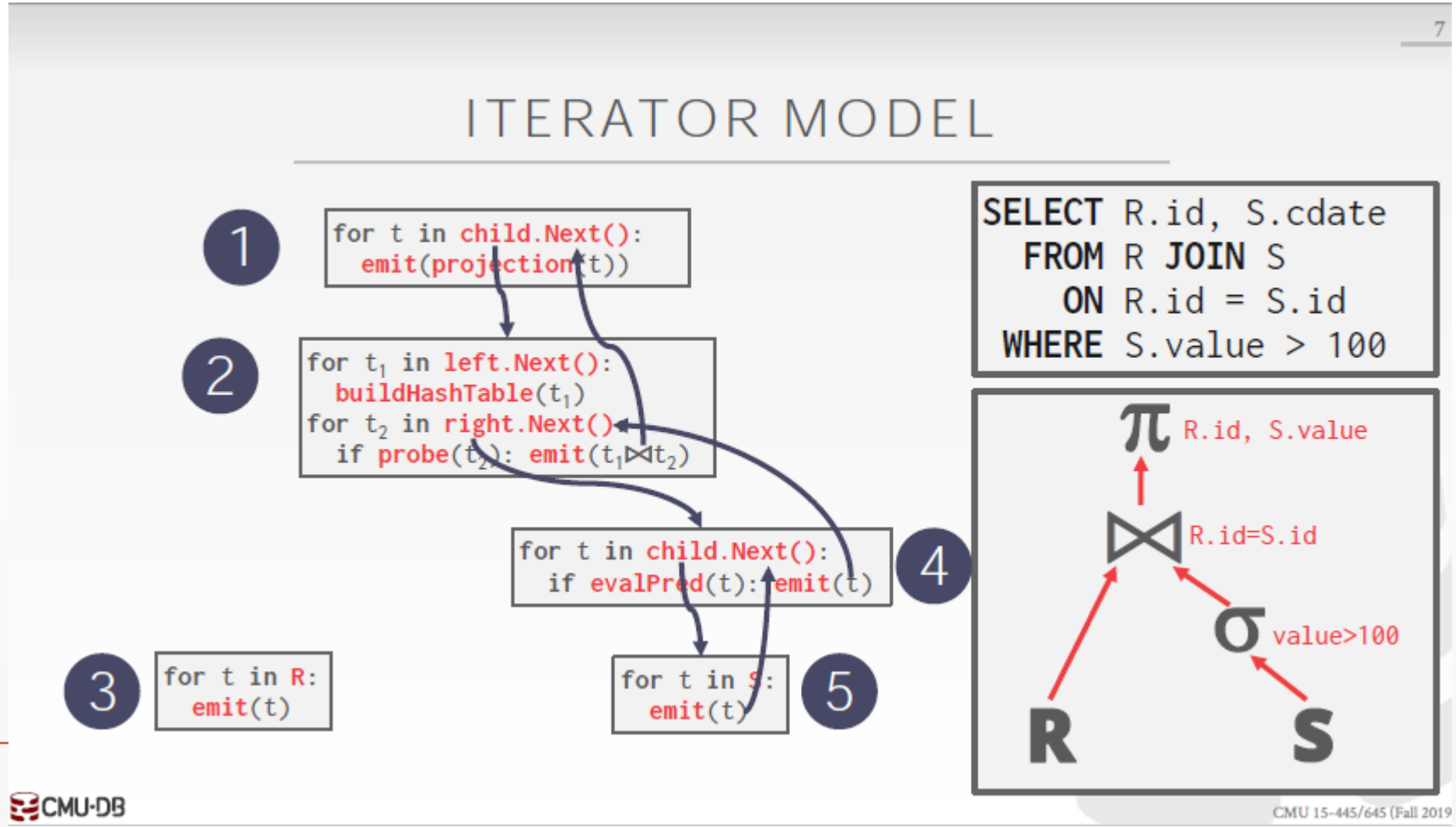
Neblokirajući iteratori (procesiranje iterisanjem)

Volcano ili Pipeline Model



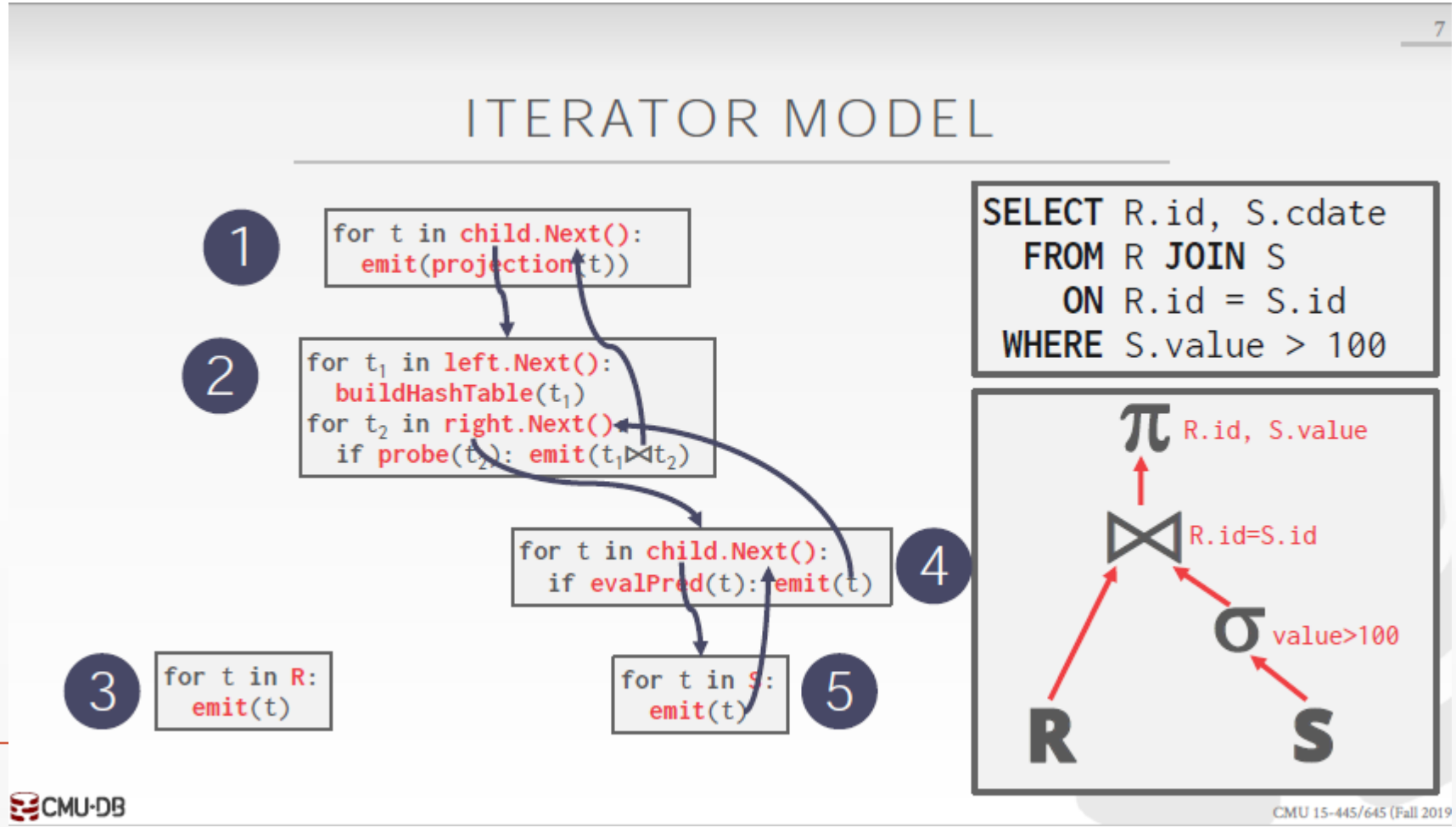
Neblokirajući iteratori (procesiranje iterisanjem)

Volcano ili Pipeline Model



Neblokirajući iteratori (procesiranje iterisanjem)

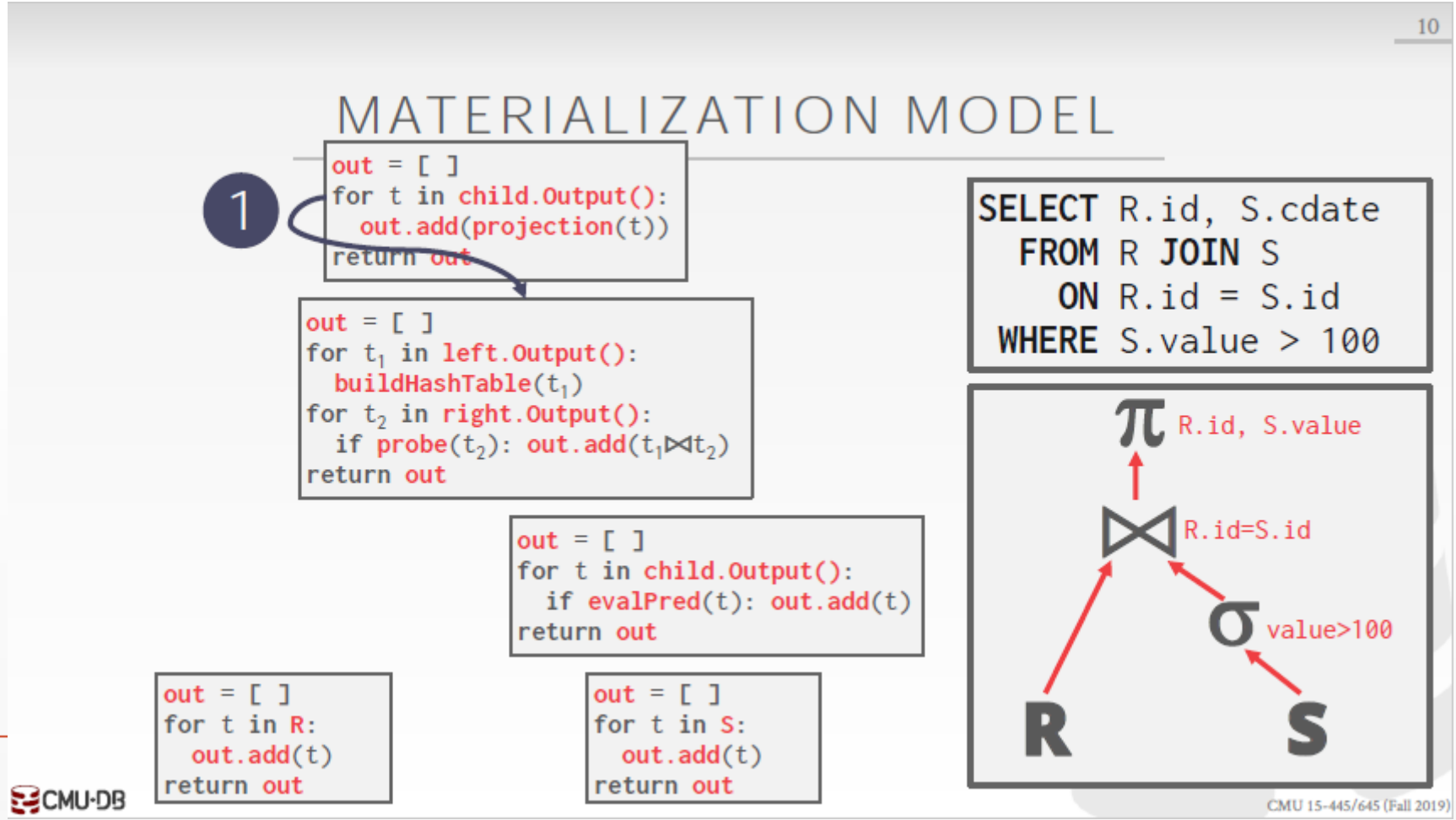
Volcano ili Pipeline Model



Materijalizacija

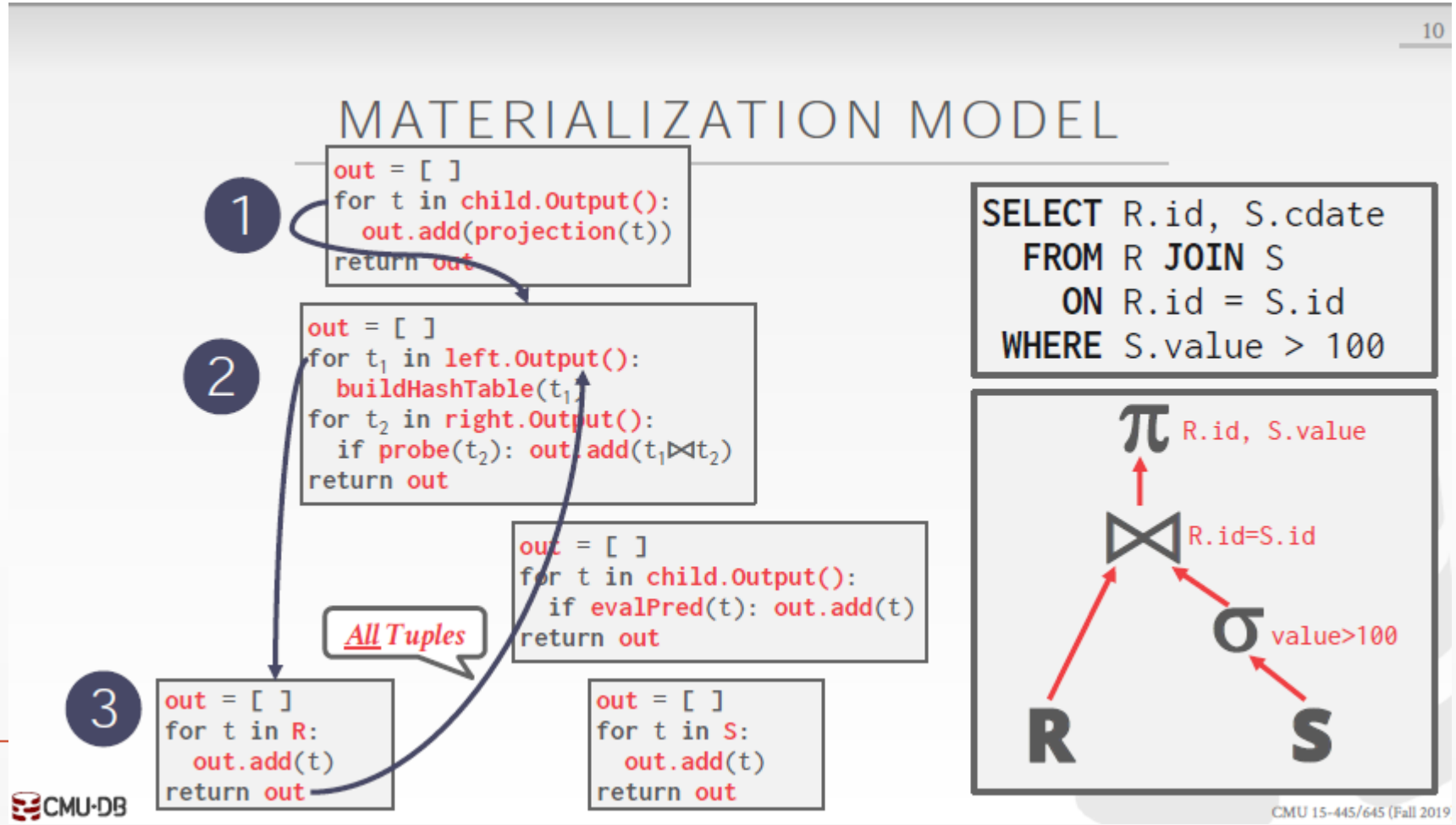
Blokirajući operatori

Ceo rezultat se prosleđuje roditelju odjednom.



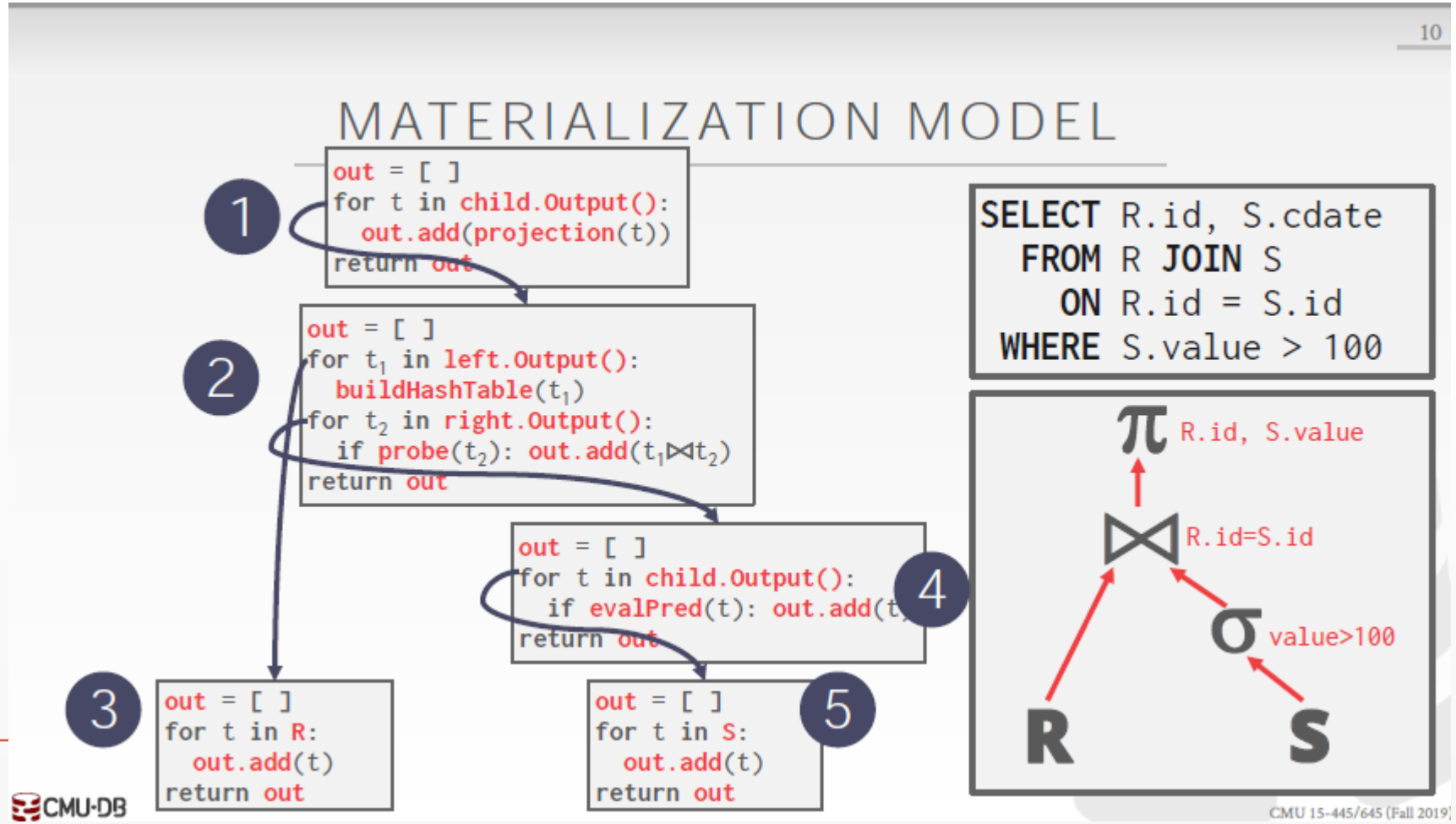
Materijalizacija

Blokirajući operatori



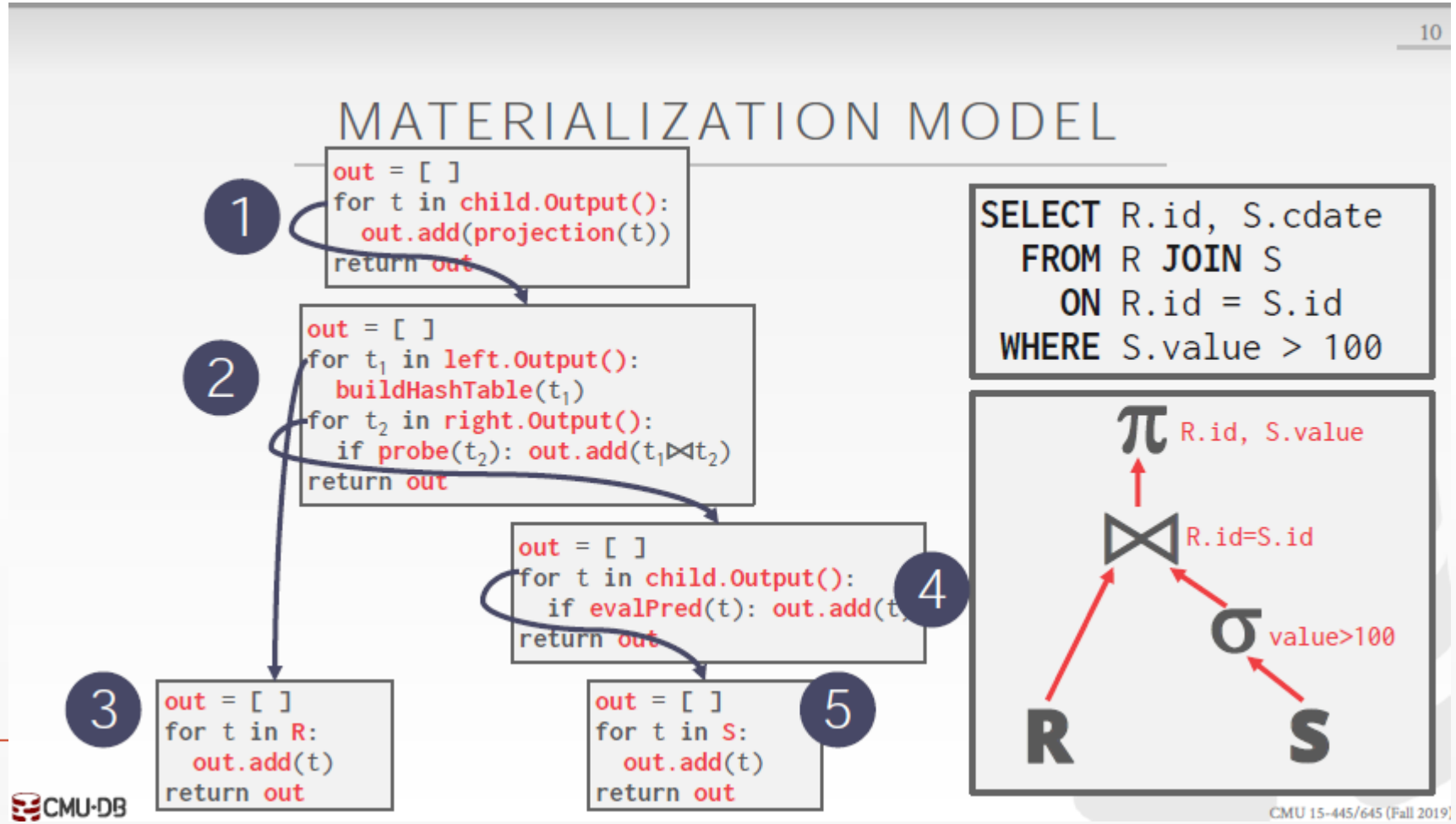
Materijalizacija

Blokirajući operatori



Materijalizacija

Blokirajući operatori



Batch procesiranje

Operator emituje deo po deo rezultata