Αλγόριθμοι για σύζευξη

Γιάννης Κωτίδης

Checkpoint

- Έχουμε δει τον αλγόριθμο SMJ
- Θα δούμε επιπλέον τους
  - Nested-Loop-Joins (NLJ)
  - Nested-Loop-Joins με ευρετήρια (INLJ)
  - Zig-Zag Joins
  - Hash Joins

Natural Join Example

R

B	C
a	10
a	20
b	10
d	30

$\bowtie$

C	D
10	cat
40	dog
15	bat
20	rat

S

Result:

a	10	~~10~~	cat
a	20	~~20~~	rat
b	10	~~10~~	cat

Nested Loop Join (NLJ)

R

B	C
a	10
a	20
b	10
d	30

$\bowtie$

C	D
10	cat
40	dog
15	bat
20	rat

S

Μπορώ να εκτελέσω τη σύζευξη με τους παρακάτω φωλιασμένους βρόγχους

```
for each r ∈ R do
    for each s ∈ S do
        if r.C = s.C then output r,s pair
```

Πόσο αποδοτικό είναι;

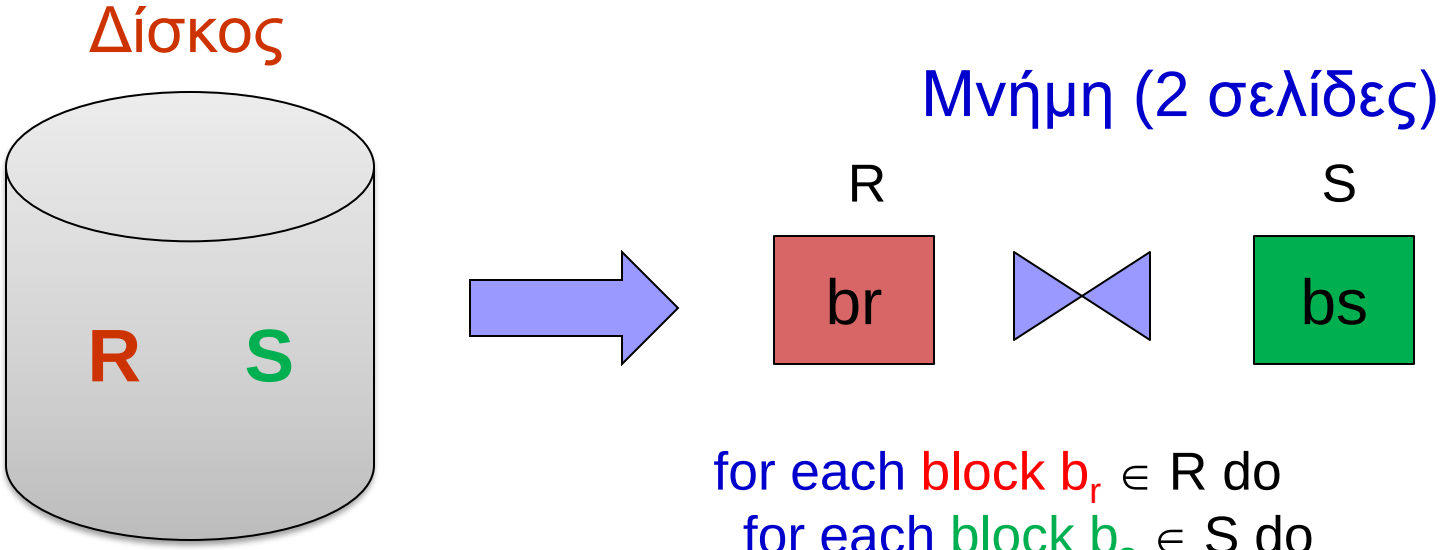
- Ουσιαστικά διαβάζω την εσωτερική σχέση S μία φορά για κάθε εγγραφή της εξωτερικής σχέσης R
  - Κόστος = $\underbrace{B(R)}_{\text{I/O για την R}} + \underbrace{T(R) \times B(S)}_{\text{I/O για την S}}$
- Το T(R) μπορεί να είναι απαγορευτικά μεγάλο

Παράδειγμα

- B(R)=1000 σελίδες, B(S)=500 σελίδες, T(R)=100000 εγγραφές
- Κόστος απλοϊκής λύσης
  - B(R)+T(R)*B(S)=1000+50,000,000 I/Os
- Για μέγεθος σελίδας 8KB, κάνω join μία σχέση R ~8MB με μία σχέση S ~4MB και διαβάζω από το δίσκο 381GB
  - Τι δεν αξιοποιώ στο αλγόριθμο;

Παρατήρηση: Χρησιμοποιώ μόνο 2 από τις Μ σελίδες μνήμης

(Block-based) NLJ

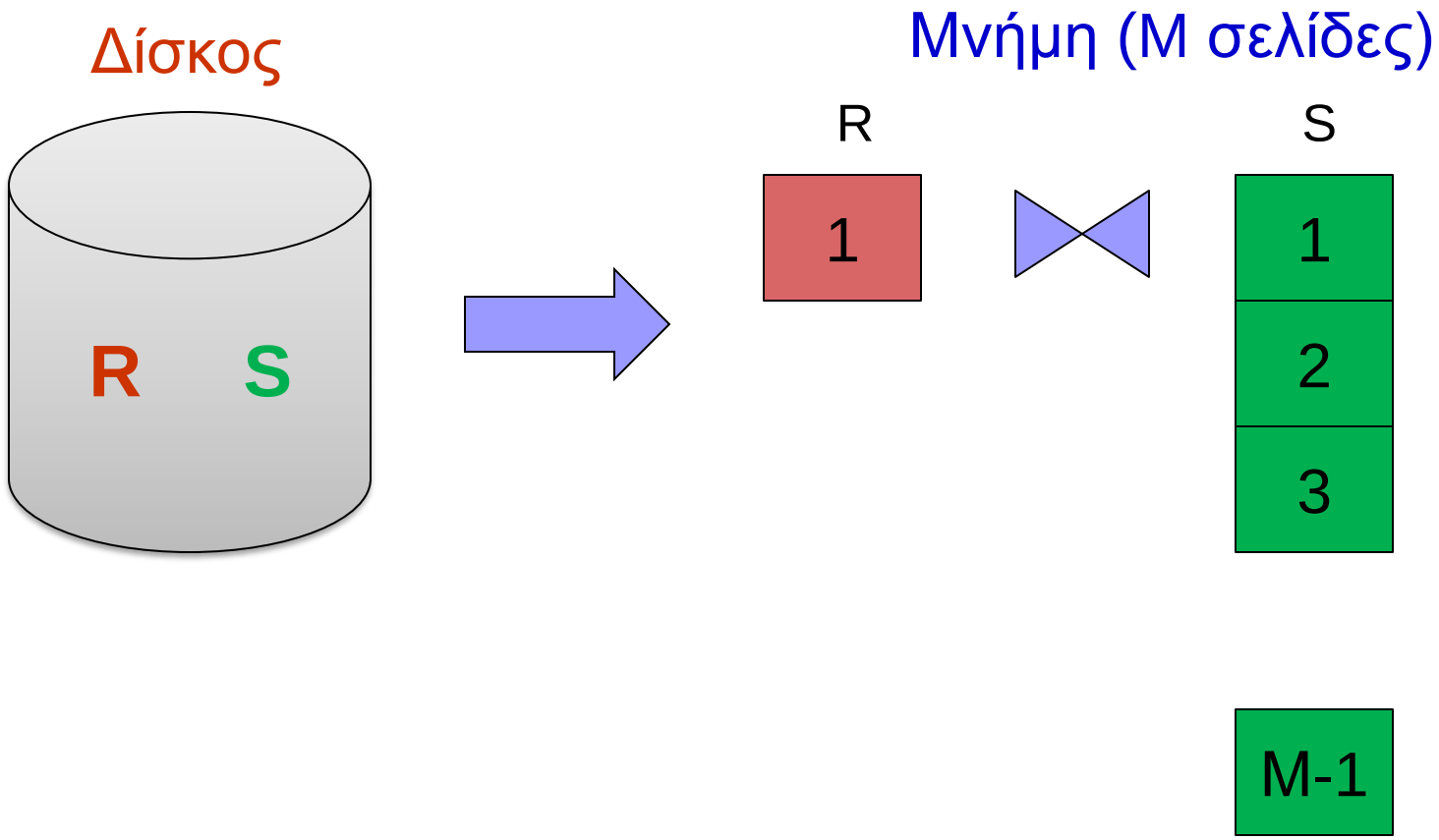


for each block $b_r \in$ R do
  for each block $b_s \in$ S do
    for each record r $\in b_r$ do
      for each record s $\in b_s$ do
        if r.C = s.C then output r,s pair

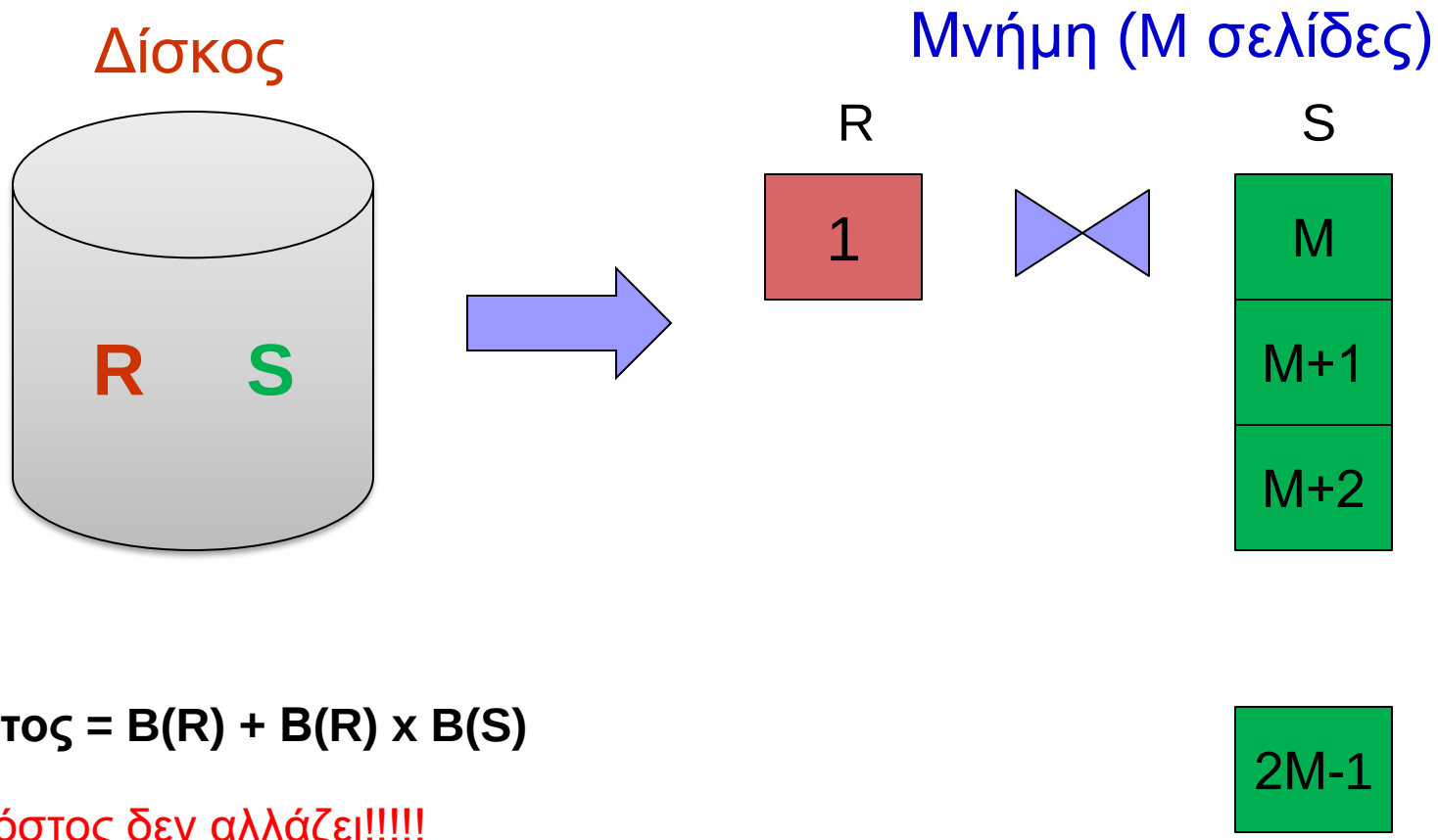
Ενώ προηγουμένως διάβαζα ολόκληρη την S για κάθε πλειάδα της R, με τη νέα έκδοση διαβάζω την S μια φορά για κάθε block (σελίδα) της R.

Επομένως: Κόστος = B(R) + B(R) x B(S)
(αντί για B(R) + T(R) x B(S))

Ας διαβάζω πολλές σελίδες της S κάθε φορά



Ας διαβάζω πολλές σελίδες της S κάθε φορά



Κόστος = B(R) + B(R) x B(S)

Το κόστος δεν αλλάζει!!!!!
Πάλι διαβάζω ολόκληρη την S
για κάθε σελίδα της R

Καλύτερη ιδέα: ας διαβάζω μεγάλα κομμάτια της εξωτερικής σχέσης R κάθε φορά

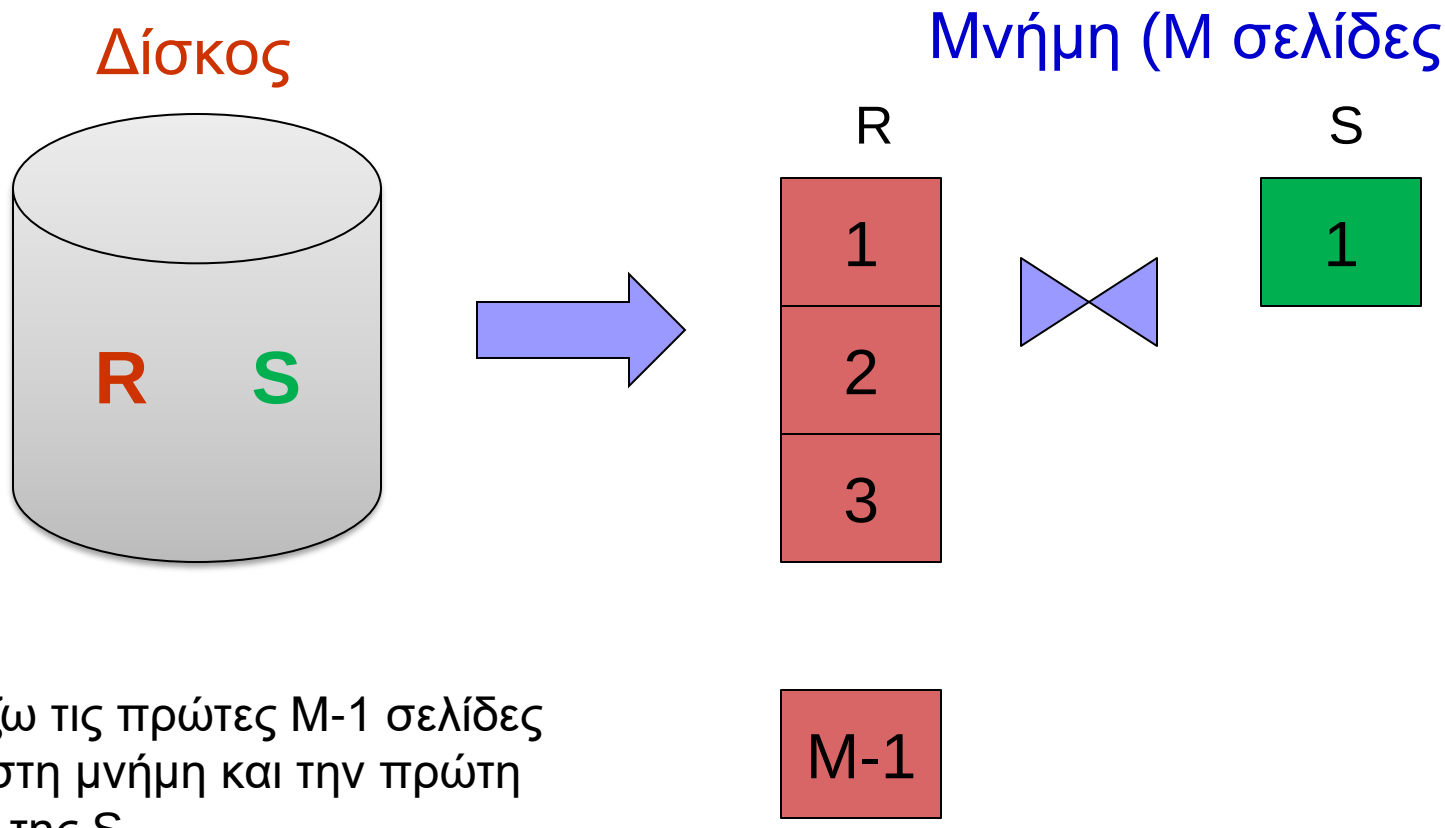
Τι θέλω να πετύχω:

Κάθε φορά που διαβάζω ένα block (σελίδα) της εσωτερικής σχέσης S θέλω να την κάνω join με όσο το δυνατό περισσότερες εγγραφές της R

Πως θα το πετύχω?

Χρησιμοποίησε όλη τη μνήμη (εκτός μιας σελίδας) ώστε να κρατάς στη μνήμη πολλές εγγραφές της εξωτερικής σχέσης R. Τις εγγραφές αυτές θα τις κάνεις join (με nested loops) με τις εγγραφές της S που διαβάζεις από το δίσκο

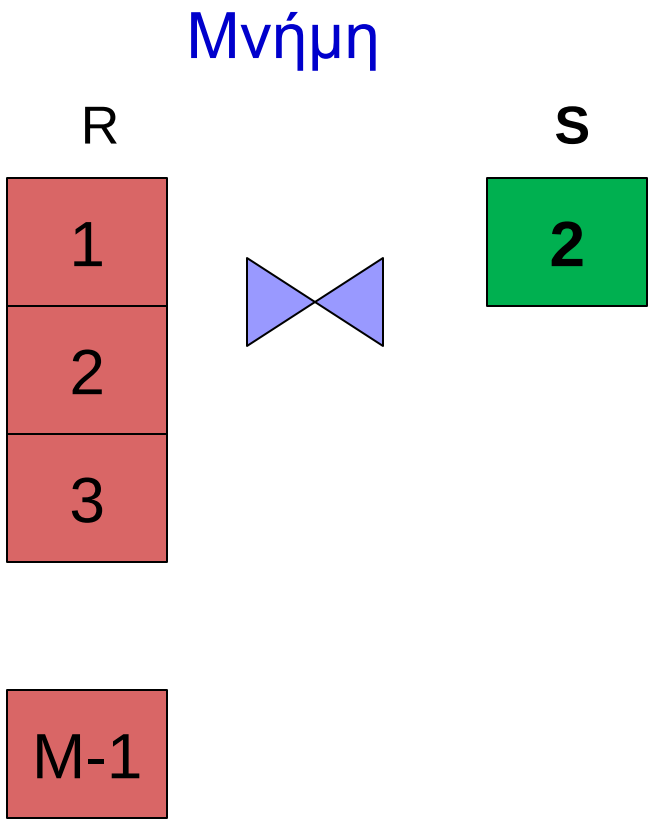
Καλύτερη ιδέα: ας διαβάζω μεγάλα κομμάτια της εξωτερικής σχέσης R κάθε φορά



Διαβάζω τις πρώτες M-1 σελίδες της R στη μνήμη και την πρώτη σελίδα της S

Συνεχίζω με την υπόλοιπη S

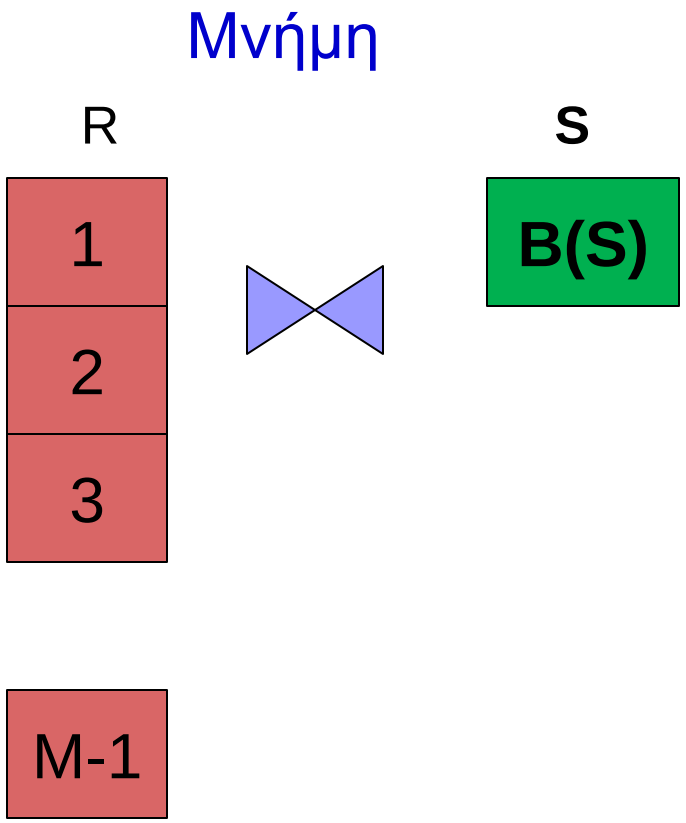
- Κράτα τις εγγραφές της R στη μνήμη, διάβασε την επόμενη σελίδα ($2^{η}$) της S
  - Κάνε join στη μνήμη
- Συνέχισε με αυτό τον τρόπο μέχρι να επεξεργαστείς όλη την S



Αφού επεξεργαστώ και την τελευταία σελίδα της σχέσης S

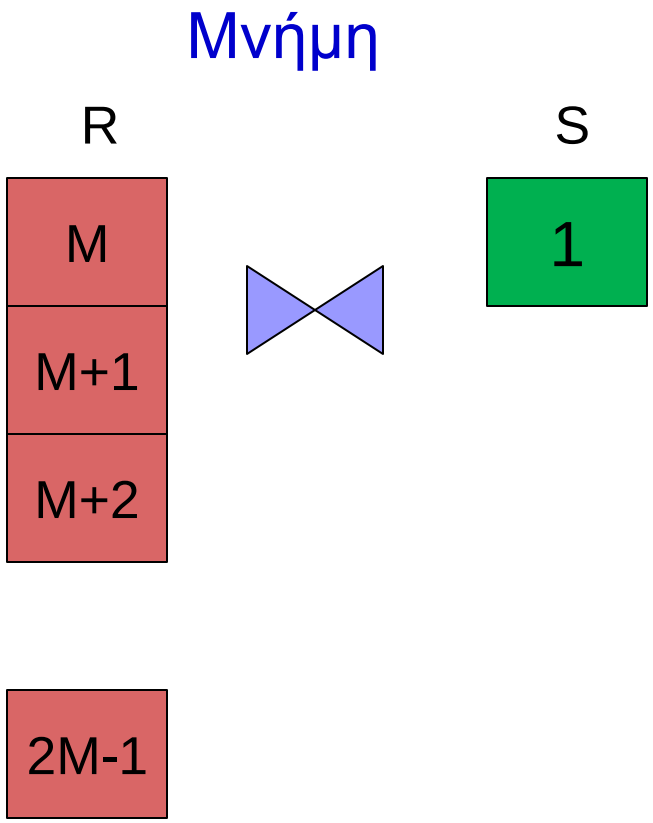
- Σε αυτό το σημείο έχεις κάνει join τις πρώτες M-1 σελίδες της R με όλη την S

- Οι πρώτες M-1 σελίδες της σχέσης R δε μου χρειάζονται πλέον
  - Ότι join results ήταν να δημιουργήσουν τα records που περιέχονται σε αυτές, έχουν υπολογιστεί!



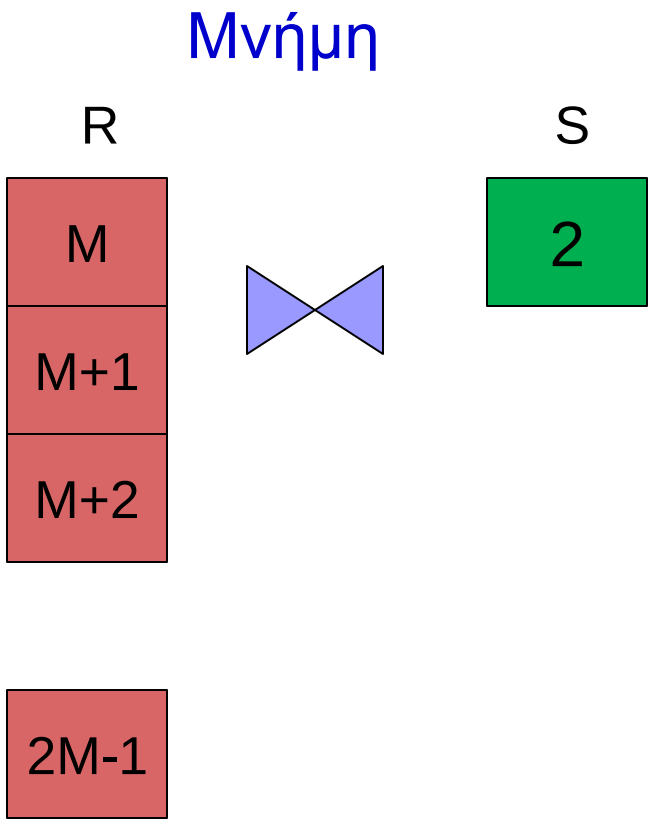
Συνεχίζω με τις επόμενες M-1 σελίδες της R

- Τις κάνω join με μία-μία τις σελίδες της S



Συνεχίζω με τις επόμενες M-1 σελίδες της R

- Τις κάνω join με μία-μία τις σελίδες της S



Τι κέρδισα;

- Σε ποσά κομμάτια μεγέθους M-1 σελίδων έχω σπάσει την εξωτερική σχέση R?
  - $\lceil B(R)/(M-1) \rceil$
- Για κάθε κομμάτι διαβάζω την S ολόκληρη
- Ποια είναι τα συνολικά I/Os για την S;
  - $\lceil B(R)/(M-1) \rceil * B(S)$
- Ποιο το κόστος για την R;
  - B(R) (μία φορά μόνο διαβάζω την κάθε σελίδα της)
- Κόστος Αλγορίθμου (συνολικά I/Os για R και S)
  - $B(R)+\lceil B(R)/(M-1) \rceil *B(S)$

Παράδειγμα

- B(R)=1000 σελίδες, B(S)=500 σελίδες, T(R)=100000 εγγραφές, M=251 σελίδες
- Κόστος απλοϊκής λύσης
  - B(R)+T(R)*B(S)=1000+50000000 I/Os
- Κόστος βελτιωμένου αλγορίθμου NLJ
  - $B(R)+\lceil B(R)/(M-1) \rceil *B(S)$ = 1000+4*500 = 3000 σελίδες

Παρατήρηση

- B(R)=1000 σελίδες, B(S)=500 σελίδες, T(R)=100000 εγγραφές, M=251 σελίδες
- Αν κάνω την S εξωτερική σχέση
- Κόστος βελτιωμένου αλγορίθμου NLJ(S,R)
  - $B(S)+\lceil B(S)/(M-1) \rceil * B(R)$=500+2*1000 = 2500 σελίδες
- Άρα: ως εξωτερική διαλέγω τη μικρότερη σχέση
  - Προσοχή ο κανόνας δεν ισχύει πάντα, πχ αν υπάρχουν επιλογές

Παρατήρηση

- Κόστος(SMJ)=3(1000+500)=4,500
  - Έλεγχος: (1000+500) < 251^2
- Κόστος(NLJ)=2,500 < Κόστος(SMJ)
- ΌΜΩΣ
  - Κόστος NLJ = O(B(R)*B(S))=O(n^2) [Quadratic]
  - Κόστος SMJ = O(B(R)+B(S))=O(n) [Linear]
- Άρα για μικρές σχέσεις ο NLJ είναι ΟΚ αλλά για μεγαλύτερες ο SMJ θα είναι καλύτερος (next slide)

Αν οι σχέσεις είναι x100 μεγαλύτερες (M=321)

	Κόστος SMJ	Κόστος NLJ
B(R) = 1,000 B(S) = 500	3*(100+500) = 4,500	500+2*1000 = 2,500
B(R) = 100,000 B(S) = 50,000	5*(100000+50000)=750,000	50000+157*100000=15,750,000

Άλλη μία διαφορά

- Επίσης ο SMJ είναι blocking (πρέπει να τελειώσει η πρώτη φάση για να αρχίσω να παράγω αποτελέσματα κατά τη δεύτερη φάση)
- Στον NLJ όπως κάνω τις συγκρίσεις μπορώ να παράγω αποτελέσματα και να τα δίνω στον επόμενο τελεστή (pipelined execution)

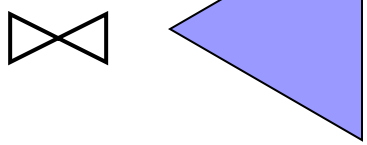
Indexed Nested Loop Joins (INLJ)

- Η **εσωτερική** σχέση έχει ευρετήριο στο γνώρισμα που κάνω join (το κοινό γνώρισμα αν πρόκειται για φυσική σύζευξη)
- Διαβάζω μία-μία τις εγγραφές της εξωτερικής σχέσης
  - Κοιτάω την τιμή του γνωρίσματος της σύζευξης στην εγγραφή που διάβασα και ρωτάω το ευρετήριο για εγγραφές της εσωτερικής σχέσης με την ίδια τιμή
    - Όσες βρω, εξ ’ορισμού κάνουν join

Joins Using Existing Indexes

R

B	C
a	10
a	20
b	10
d	30



C	D
10	cat
40	dog
15	bat
20	rat

S

- Indexed NLJ (conceptually)

```
for each r ∈ R {
   probe(I,r.C)
      for each s returned by index-probe
       output r,s pair
}
```

Πίσω στο παράδειγμα μας

- R(B,C)⋈S(C,D)
  - Non-clustering index I(S.C)
  - B(R)=1000 σελίδες, B(S)=500 σελίδες, T(R)=100000 εγγραφές
  - Το ευρετήριο βρίσκεται στη μνήμη
- Ας υποθέσουμε ότι το ευρετήριο I μας γυρίζει σε κάθε probe (αναζήτηση) 2 εγγραφές κατά μ.ό.
- Ποιο το κόστος του να φέρω αυτές τις 2 εγγραφές στη μνήμη;

INLJ με non-clustering index

- Κόστος ανάγνωσης μέσω ευρετηρίου = αριθμός εγγραφών που αυτό μου γυρίζει =2 στο παράδειγμα αυτό
  - Non-clustering index→ υποθέτω ότι κάθε εγγραφή στην οποία μας οδηγεί το ευρετήριο βρίσκεται σε διαφορετική σελίδα στο δίσκο
    - Worst case scenario
    - Αγνοώ πιθανά cache-hits στον buffer manager (δηλαδή το ενδεχόμενο η εγγραφή που ψάχνω να βρίσκεται σε μια σελίδα που έχω διαβάσει νωρίτερα)

INLJ με non-clustering index

- Κόστος INLJ
  - Διαβάζω μία-μία τις εγγραφές της R. Συνολικά θα διαβάσω όλες τις σελίδες της R από το δίσκο
    - Κόστος = B(R)
  - Κάνω T(R) ερωτήσεις στο ευρετήριο
    - Κόστος = 2 I/O η ανάκτηση των εγγραφών της S (στο παράδειγμα αυτό) για κάθε ερώτηση

INLJ με non-clustering index (συνέχεια παραδείγματος)

- Κόστος INLJ= 1000+100000*2 = 201000
- Αν το ευρετήριο ήταν clustering το κόστος γίνεται
  - 1000+100000*1 = 101000 (γιατί?)

INLJ με dense clustering index

- Προσέξτε την παρακάτω αλλαγή στο παράδειγμα
  - Έστω ότι από τις 100 φορές που ρωτάω το ευρετήριο μόνο 1/100 φορές παίρνω πίσω 2 εγγραφές. Τις υπόλοιπες 99/100 φορές δεν βρίσκω σχετική εγγραφή
- Κόστος INLJ = B(R) + T(R) * (1/100) * 1 = 1000 + 100000/100*1 = 2000 I/Os
  - Αν το ευρετήριο ήταν sparse το κόστος θα ήταν 1000 + 100000*1 = 101000 (γιατί;)

Παράγοντες που επηρεάζουν την απόδοση του INLJ

- Έστω ότι κάθε index probe γυρνάει κατά μέσο όρο $X \geq 1$ εγγραφές της S
- Ευρετήριο clustering ή όχι;
  - Clustering:
    - κόστος = B(R)+T(R) * $\lceil$X/(αριθμός εγγραφών ανά σελίδα της S)$\rceil$
  - Non-clustering:
    - κόστος = B(R)+T(R)*X

Dense vs Sparse (clustering)

- Αν X<1
- Dense: μπορώ να ξέρω αν η τιμή που ψάχνω υπάρχει ή όχι στη S χωρίς να πάω στην S
  - Κόστος = B(R)+T(R) * X (πολύ καλή απόδοση αν X<<1, δηλαδή το join result είναι μικρό)
- Sparse: πρέπει να πάω στη σελίδα της S που δείχνει το ευρετήριο να δω αν υπάρχει ή όχι εγγραφή με τη τιμή που ψάχνω
  - Κόστος = B(R) + T(R)

NLJ ή Indexed NLJ;

- Αν το μέγεθος του αποτελέσματος της σύζευξης είναι μικρό το INLJ είναι συνήθως πολύ καλό
  - Αλλιώς είναι σημαντικό το ευρετήριο να είναι clustering
- Μία άλλη διαφορά
  - INLJ → random I/O
  - NLJ → sequential I/O (αν έχουμε αποθηκεύσει κατάλληλα τις σχέσεις στο δίσκο)

Sort-Merge Join with Indexes on both relations

- Can avoid sorting
- Zig-zag join

Στη συνέχεια

- Σύζευξη με χρήση κατακερματισμού

R JOIN S

Ας υποθέσουμε και οι 2 χωράνε στη μνήμη

Table R

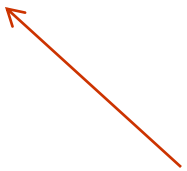
1
8
2
3
7
7

Table S

1
0
4
2
7
9

Ας χρησιμοποιήσουμε nested loops

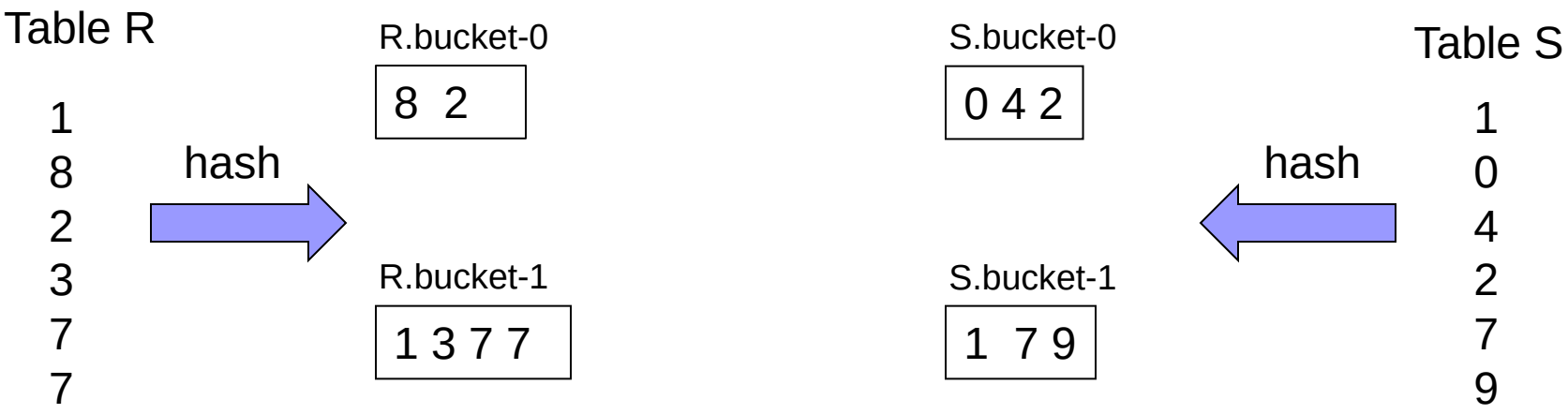
Assume CPU-Cost = number of comparisons = ?



Τιμή κοινού γνωρίσματος

Divide and Conquer

hash function
h(x) = x mod 2



CPU-Cost = ????

Hash Join – Ιδέα

- Κατακερματίζουμε τις δύο σχέσεις χρησιμοποιώντας την ίδια συνάρτηση h() στο κοινό γνώρισμα της σύζευξης (έστω x)
  - Έστω η εγγραφή r της σχέσης R *κάνει join* (equijoin) με την εγγραφή s της σχέσης S:
    - "κάνει join" $\Rightarrow$ r.x = s.x $\Rightarrow$ h(r.x) = h(s.x)
    - αν h(r.x) ≠ h(s.x) $\Rightarrow$ r.x ≠ s.x $\Rightarrow$ δεν κάνουν join οι r, s εγγραφές
  - Επομένως θα ελέγξω αν κάνουν join μόνο εγγραφές των σχέσεων που αντιστοιχίζονται στο ίδιο bucket μέσω της h()

Αλγόριθμος Hash Join

- Hash join (conceptual)
  - Hash function h, range 1 $\rightarrow$ k
  - Buckets for R1: G1, G2, ... Gk
  - Buckets for R2: H1, H2, ... Hk

Algorithm

(1) Hash R1 tuples into G1--Gk

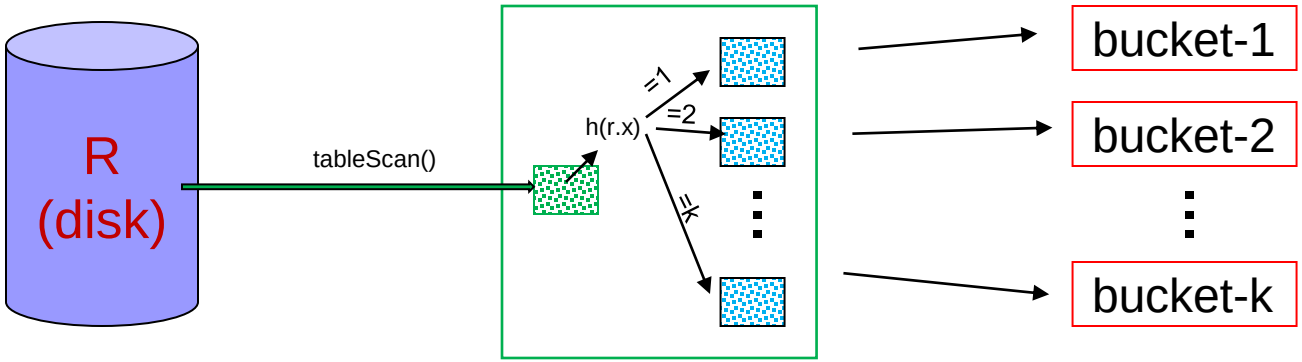
(2) Hash R2 tuples into H1--Hk

(3) For i = 1 to k do

Match tuples in Gi, Hi buckets

Οργάνωση μνήμης (buffers)

- 1 input buffer στο οποίο θα διαβάζω μια-μια τις σελίδες της σχέσης.
- 1 output buffer για κάθε bucket στο οποίο θα συλλέγω τις εγγραφές της σχέσης που τοποθετούνται στο bucket μέσω του κατακερματισμού. Όταν γεμίζει θα «αδειάζει» στο δίσκο στο αντίστοιχο αρχείο.

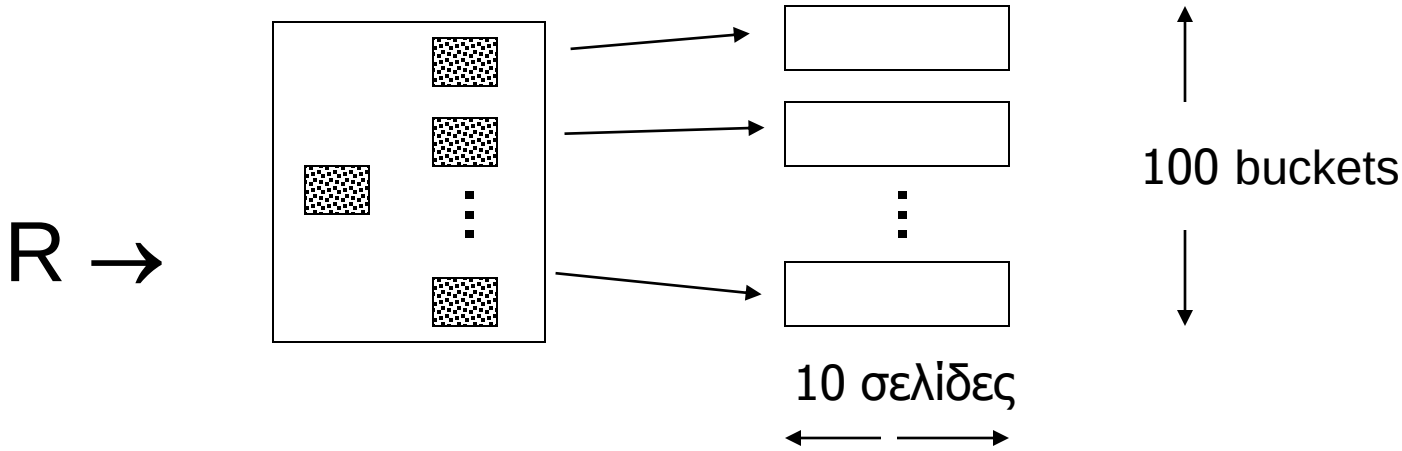


Προηγούμενο παράδειγμα με Hash Join

- B(R)=1000, B(S)=500

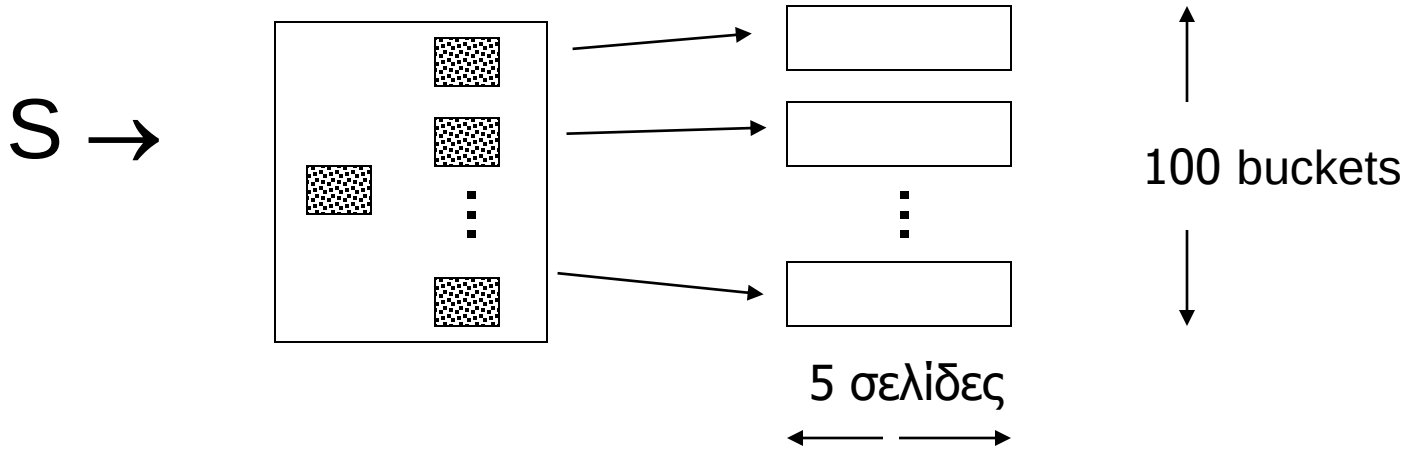
→ Use k=100 buckets

→ Read R, hash, + write buckets



Ίδια διαδικασία για την S

B(S)=500 σελίδες



Επόμενο βήμα

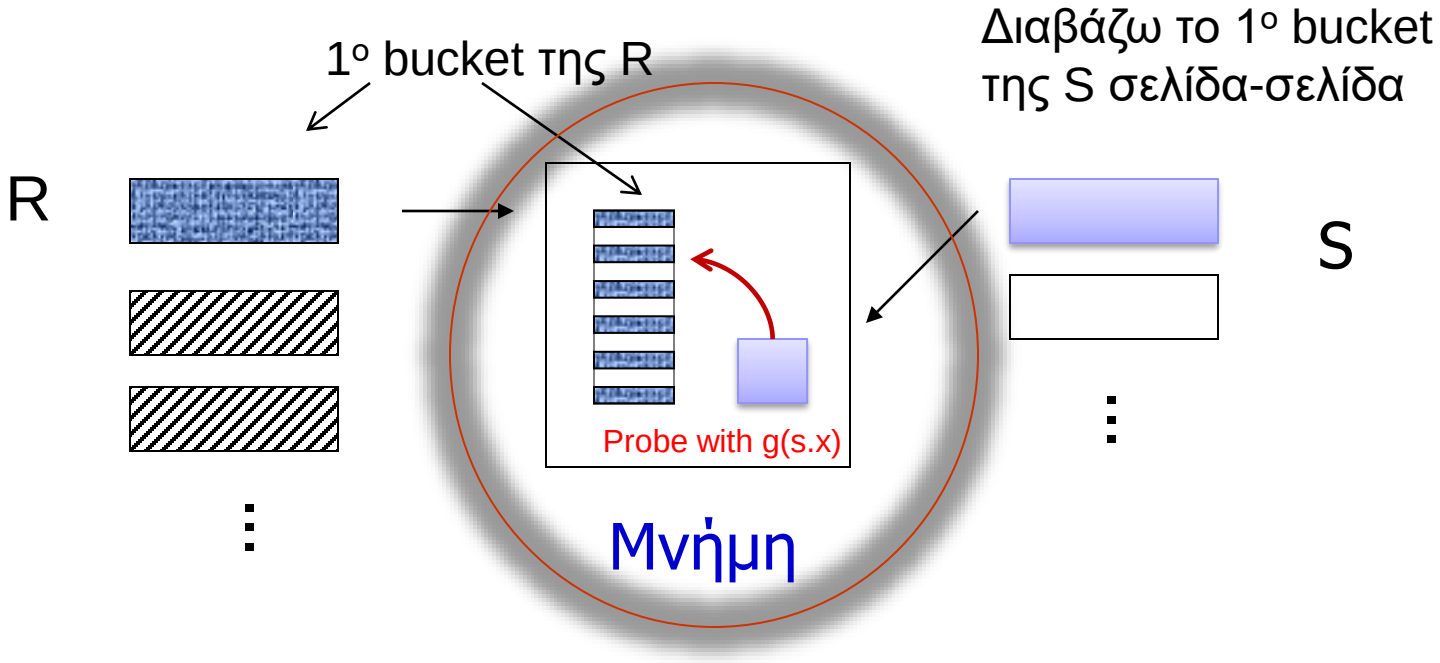
- Διάβασε ολόκληρο το πρώτο bucket της R (~10 σελίδες)
  - Διάβασε μία-μία τις σελίδες του πρώτου bucket της S
  - Έλεγξε αν οι εγγραφές της R στη μνήμη κάνουν join με τις εγγραφές στη μνήμη της S
  - Πως;
    - SMJ ή NLJ στη μνήμη
    - Hash Join ?

Hash-Join στη μνήμη

- Διάλεξε μία διαφορετική συνάρτηση κατακερματισμού g() και κάνε hash τις εγγραφές στη μνήμη από το πρώτο bucket της R
  - Οι κάδοι της g() διατηρούνται στη μνήμη
- Για κάθε εγγραφή s από τη σελίδα της S που έχεις στη μνήμη ψάξε στο bucket g(s.x) της R για εγγραφές που κάνουν join
  - Το πρώτο bucket της R στη μνήμη έχει σπάσει σε μικρότερα buckets με βάση την g()

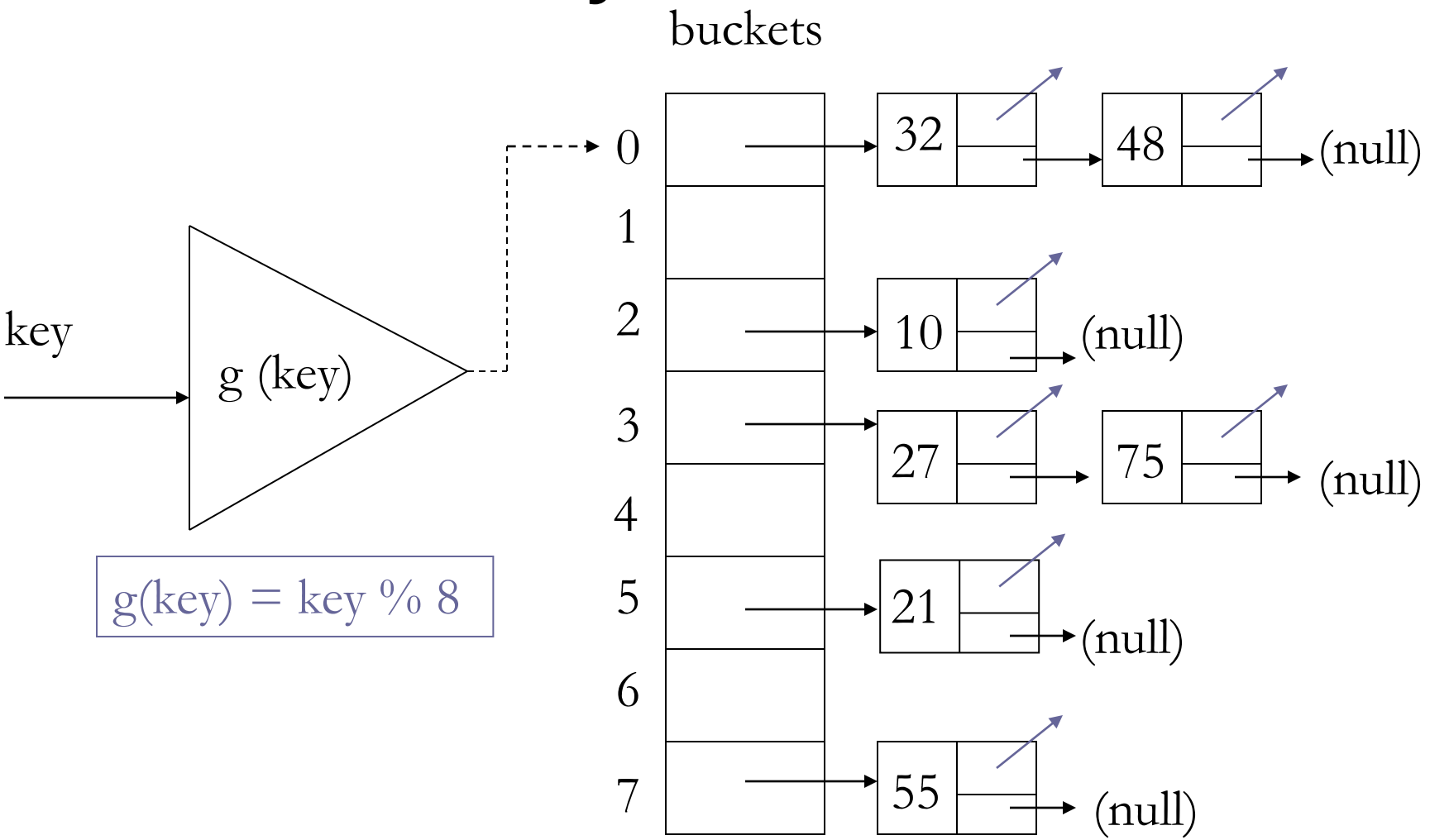
-->Read one R bucket; build memory hash table
[R is called the build relation of the hash join]

-> Read **corresponding** S bucket + hash probe
[S is called the probe relation of the hash join]



Επαναλαμβάνω την ίδια διαδικασία για τα υπόλοιπα buckets

Main Memory Hash Table



Κόστος Hash-Join:

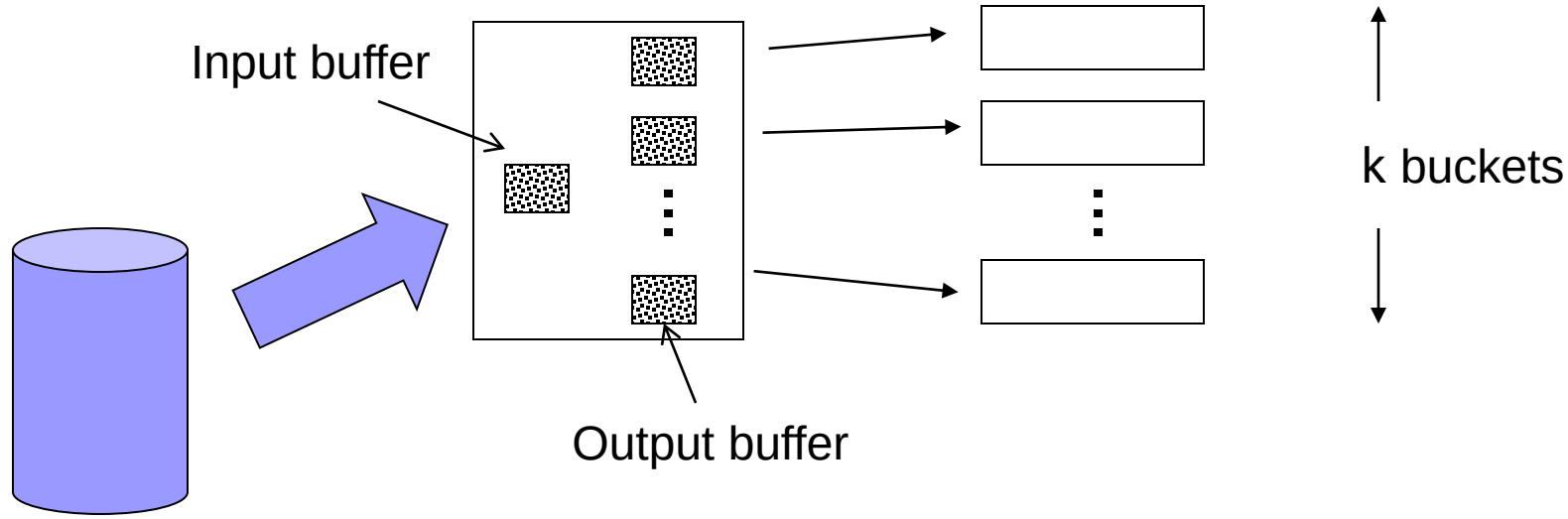
"Bucketize:"	Read R + write buckets
	Read S + write buckets
Join:	Read R, S buckets

Total cost = 3 x [1000+500] = 4500

Note: this is an approximation since buckets will vary in size and we have to round up to blocks

Απαιτήσεις σε μνήμη

- Αριθμός buckets k = M-1
  - Όταν φτιάχνω τα buckets θέλω output buffer 1 σελίδα για κάθε bucket + 1 σελίδα input buffer



Απαιτήσεις σε μνήμη

- Μέγιστος αριθμός buckets k = M-1(μιας και θέλω 1 output buffer για το καθένα στη πρώτη φάση, δες προηγούμενο σχήμα)
- Στη δεύτερη φάση τα buckets της R πρέπει να χωράνε στη μνήμη
- (Αναμενόμενο) μέγεθος bucket της R σε σελίδες: B(R)/k
  = B(R)/(M-1) ≤ M-1 ➔ B(R) ≤ $(M-1)^2$

  ➔ $M > \sqrt{B(R)}$
- Το μέγεθος της σχέσης S δεν παίζει ρόλο
  - Άρα ως «εξωτερική» σχέση διαλέγω τη μικρότερη!

Απαιτήσεις σε μνήμη Hash-Join

$M > \sqrt{\min(B(R),B(S))}$

[$M > \sqrt{(B(R)+B(S))}$ για το (efficient) Sort-Merge Join]

Στο παράδειγμα μας $M > \sqrt{500}=23$ σελίδες για Hash Join και

$M>\sqrt{1500}=39$ σελίδες για SMJ

Παρατήρηση

- Αν μία από τις δύο σχέσεις χωράει στη μνήμη (πχ η R) και περισσεύει μια σελίδα μνήμης για την S, το join μπορεί να εκτελεστεί σε ένα πέρασμα
  - Κάνω hash την R στη μνήμη
  - Διαβάσω μία μία τις σελίδες της S
    - Κάνω probe με βάση την τιμή h(s.x) για κάθε εγγραφή s
    - Αν στον κάδο που θα καταλήξω βρω εγγραφές της R που κάνουν join επιστρέφω το αποτέλεσμα
- Κόστος=B(R)+B(S)

Βελτιστοποιήσεις

- Έστω η σχέσεις Plays3D(Title, Cinema, Address) και Movies(Title, Actor, Trailer)
- Το γνώρισμα Trailer είναι ένα MP4 αρχείο μερικών MB
- Όταν κάνω Sort ή Hash Join τις εγγραφές πχ με βάση το γνώρισμα Title δε κρατάμε όλη την εγγραφή της Movies αλλά το ζεύγος <Title,Rec_pointer>
  - Rec_pointer είναι δείκτης προς την εγγραφή στο δίσκο (πχ το primary key αν υπάρχει)
  - Μειώνουμε τις απαιτήσεις σε μνήμη (πιθανών πλέον η σχέση να χωράει στη μνήμη και το sort/hash να γίνει σε ένα πέρασμα)
- Μειονέκτημα: στο τέλος για όσες εγγραφές κάνουν join πρέπει να ακολουθήσω τους pointers για να πάρω τα υπόλοιπα γνωρίσματα

Παράδειγμα

- Plays3D(Title, Cinema, Address)
  - B(Plays3D)=1000 σελίδες, T(Plays3D)=10000 εγγραφές
- Movies(Title, Actor, Trailer)
  - B(Movies)=10000 σελίδες, T(Movies) =100 εγγραφές
- 20 ζεύγη <Title,Rec_pointer> χωράνε σε κάθε σελίδα
- M=50 σελίδες
- Αλγόριθμος Hash-Join

Συμβατική εκτέλεση

- Plays3D(Title, Cinema, Address) B(Plays3D)=1000 σελίδες, T(Plays)=10000 εγγραφές
- Movies(Title, Actor, Trailer) B(Movies)=10000 σελίδες, T(Movies) = 100 εγγραφές

- Build relation η Plays3D
  - B(Plays3D)=1000 < M^2 = 50*50=2500
- Κόστος = 3*(1000+10000) = 33000 I/O

Με το τρικ των pointers

- 20 ζεύγη <Title,Rec_pointer> σε κάθε σελίδα
- Οι 100 εγγραφές της Movies δημιουργούν 100 ζεύγη και χωράνε σε 5 σελίδες !
- Hash Join σε ένα πέρασμα:
  - Φτιάχνω και κρατάω τα buckets της Movies στη μνήμη
  - Διαβάζω μία-μία τις σελίδες της Plays3D. Για κάθε εγγραφή υπολογίζω την τιμή h(Title) και ψάχνω στο αντίστοιχο bucket της Movies για εγγραφές με τον ίδιο τίτλο
  - Αν βρω εγγραφή, ακολουθώ τον rec_pointer για να διαβάσω τα 2 γνωρίσματα που λείπουν (Actor και Trailer).
  - Έστω 50 τέτοιες εγγραφές στο αποτέλεσμα της σύζευξης

Με τη βελτιστοποίηση

- Κόστος =

  0 I/Os

  Read Movies + Bucketize στη μνήμη

  + Read Plays3D

  + Follow 50 Pointers

  = 10000 + 1000 + 50 = 11050

  Σύγκρινε το με το 33000 του συμβατικού αλγόριθμου.

  …επίσης πολύ λιγότερες απαιτήσεις σε μνήμη (πόσες?)

Hash-based Vs. Sort-based Joins

- Hash-Join για equi-joins
- SMJ και για θ-joins πχ R.a>S.a
- Hash-Join μικρότερες απαιτήσεις σε μνήμη
- SMJ: το γεγονός ότι το αποτέλεσμα παράγεται ταξινομημένο μπορεί να μου χρησιμεύσει αργότερα στο πλάνο εκτέλεσης: R⋈ S⋈ T

Summary

- NLJ ok for “small” relations (relative to memory size)
- For equi-join, where relations not sorted and no indexes exist, Hash Join usually best

Summary (contd.)

- Sort-Merge Join good for non-equi-join (e.g., R1.C > R2.C)
  - If relations already sorted, can use Merge Join (skip sorting)
- If indices exist, they could be useful
  - Depends on expected result size and index clustering
- Join techniques apply to Set Union, Intersection, Difference