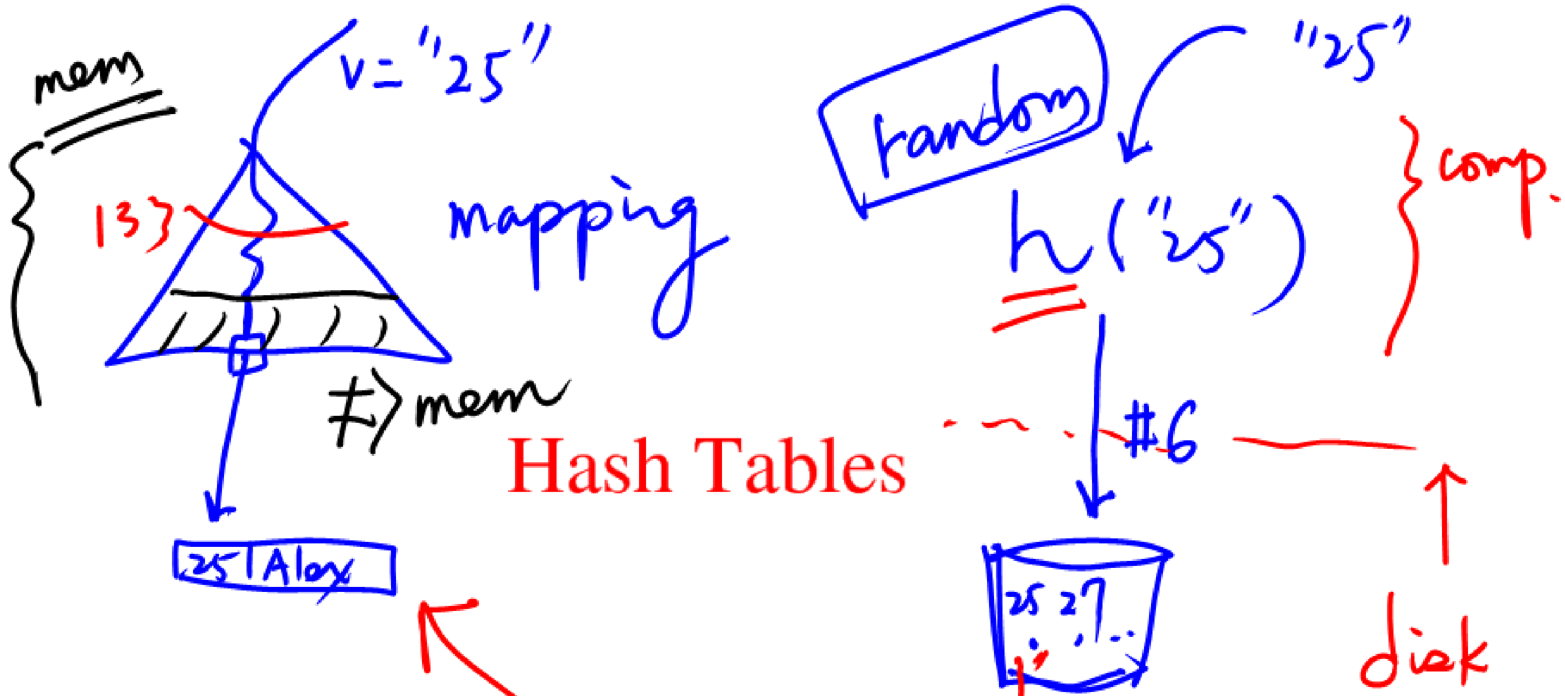
Hash Tables



+ "equal" v=25 + v=25 ✓

+ range v≥25 X v≥25?

Hash Tables

#1 h(k) #2

- Secondary storage hash tables are much like main memory ones
- Recall basics:

  disk → page/block   disk

  k/P1 k/P2 1

  - There are n *buckets*
  - A hash function f(k) maps a key k to {0, 1, …, n-1}
  - Store in bucket f(k) a pointer to record with key k
- Secondary storage: bucket = block, use overflow blocks when needed

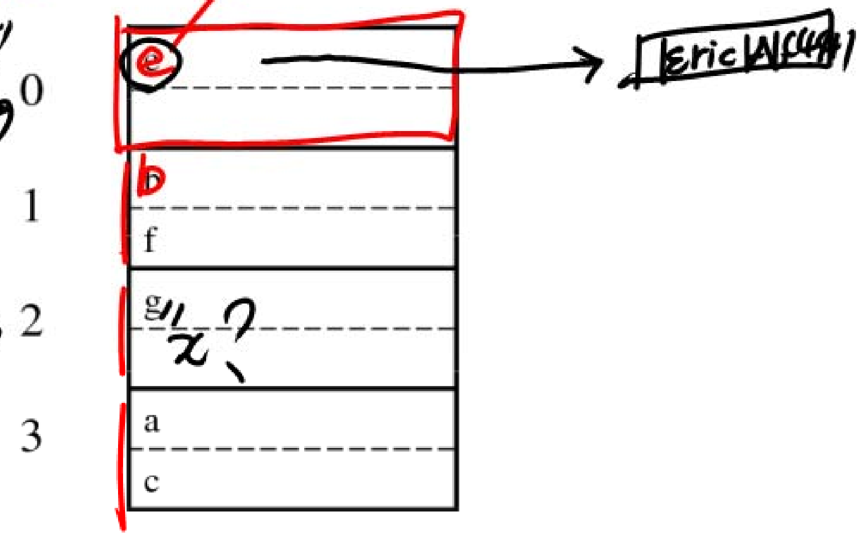
Hash Table Example

- Assume 1 bucket (block) stores 2 keys + pointers
- h(e)=0
- h(b)=h(f)=1
- h(g)=2
- h(a)=h(c)=3

select
from
where
name="eric"

4 buckets.

name attr

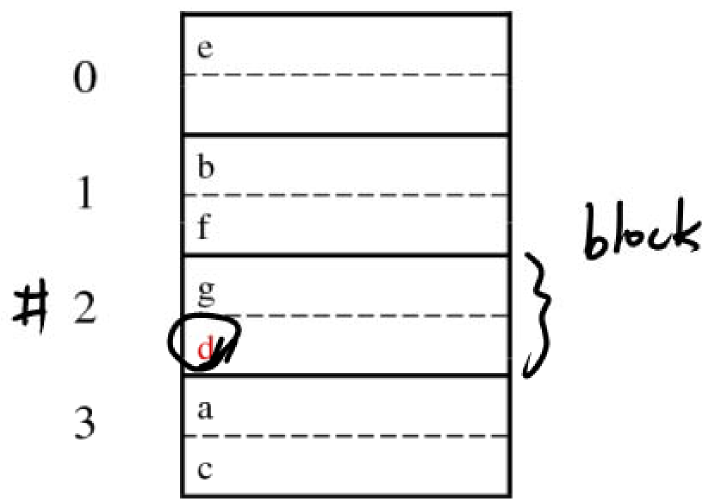


h("x")=2

nothing.

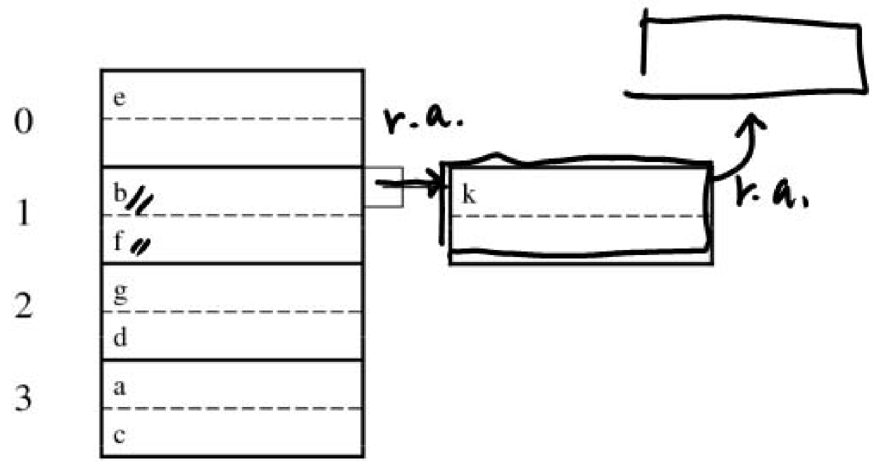
Insertion in Hash Table

- Place in right bucket, if space
- E.g. h(d)=2



Insertion in Hash Table

- Create overflow block, if no space
- E.g. h(k)=1



- More over-flow blocks may be needed

Hash Table Performance

- Excellent, if no overflow blocks
- Degrades considerably when number of keys exceeds the number of buckets (I.e. many overflow blocks).

① $\boxed{\text{Extensible}}$ Hash Table ② Linear

Extensible Hash Table

- Allows hash table to grow, to avoid performance degradation
- Assume a hash function h that returns numbers in $\{0, \ldots, 2^k - 1\}$
- Start with $n = 2^i << 2^k$, only look at first i most significant bits

k bits   k=4   0 - - - - - - 15
0000 1111
16 buc.

i bits

MSB   LSB ← Linear

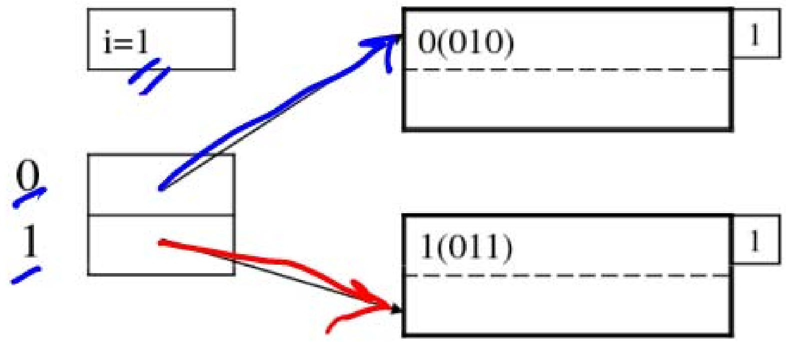
#00** — h_1 0 0 0 0
h_2 0 0 0 1
⋮
#11** 1 1 1 0
1 1 1 1

Extensible Hash Table

- E.g. i=1, n=2, k=4

$= 2^1$ max 0010 1011

$n = 2^i$

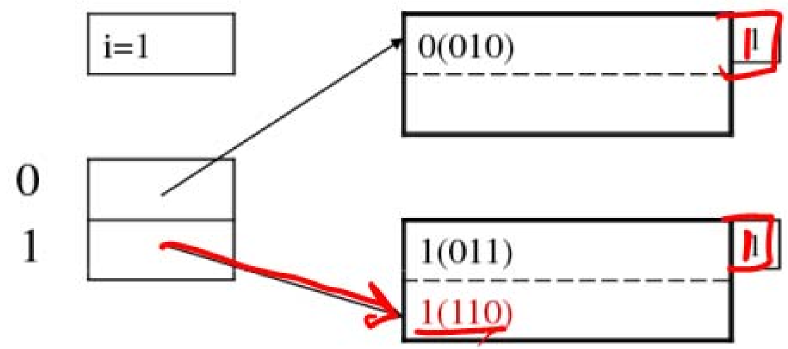


- Note: we only look at the first bit (0 or 1)

Insertion in Extensible Hash Table

- Insert 1110

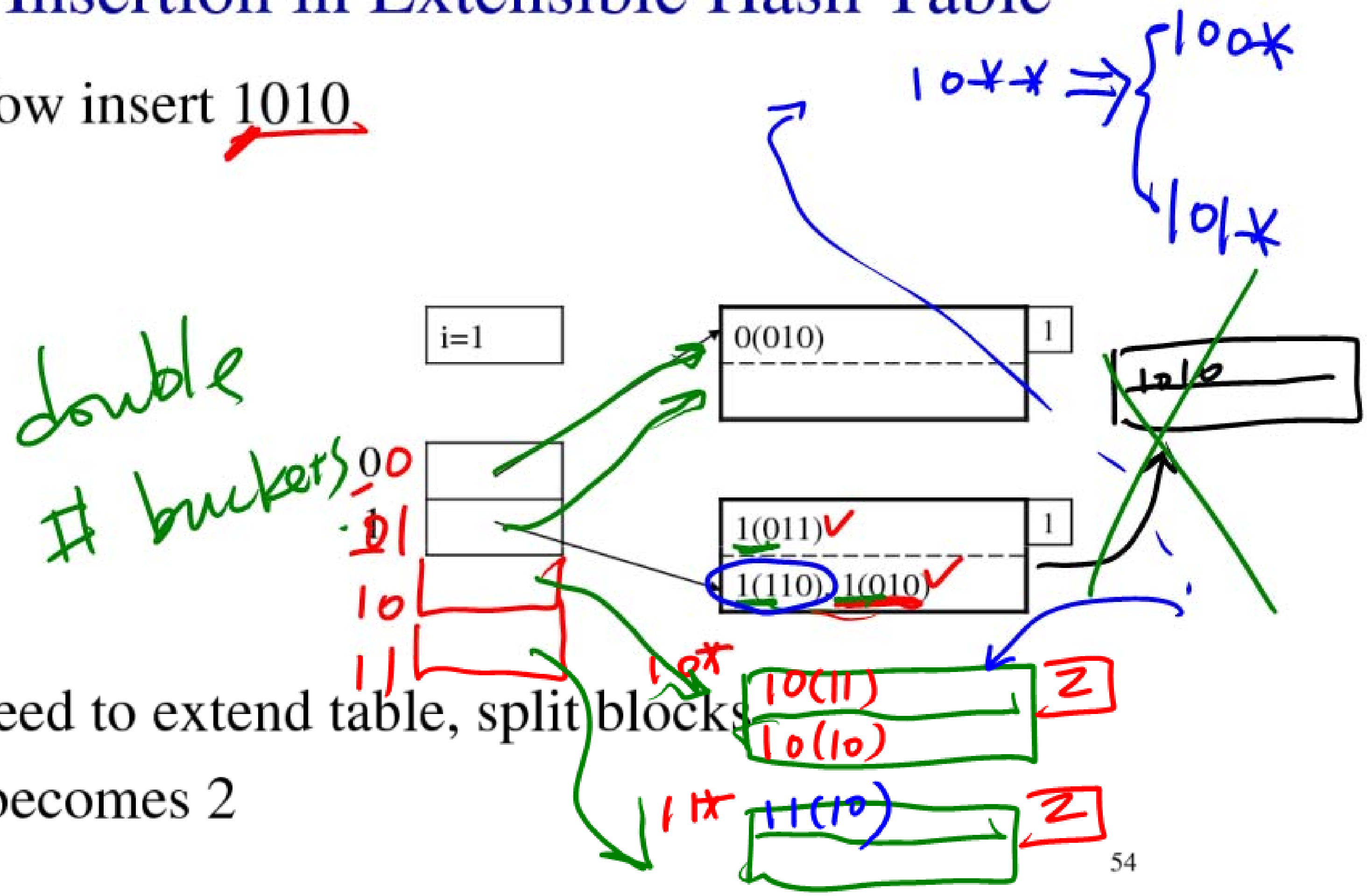
\# of M.S.b. I care about.



1(110)

Insertion in Extensible Hash Table

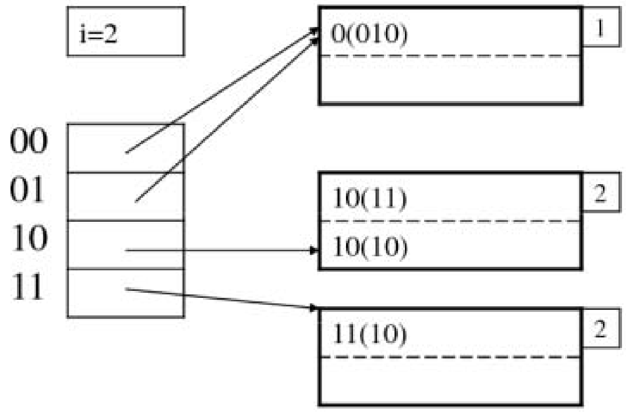
- Now insert 1010



- Need to extend table, split blocks
- i becomes 2

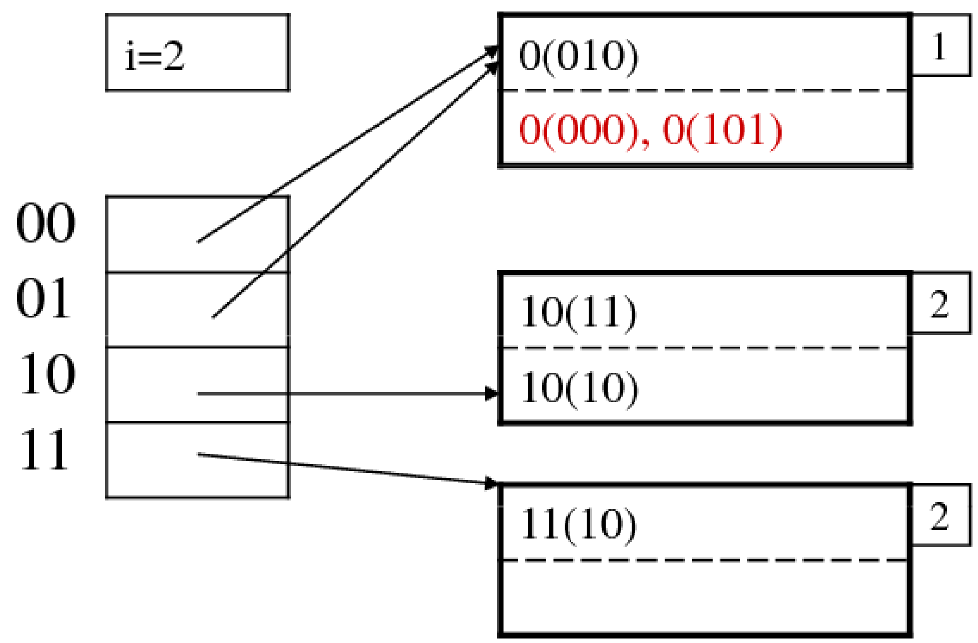
Insertion in Extensible Hash Table

- Now insert 1010 (cont.)



Insertion in Extensible Hash Table

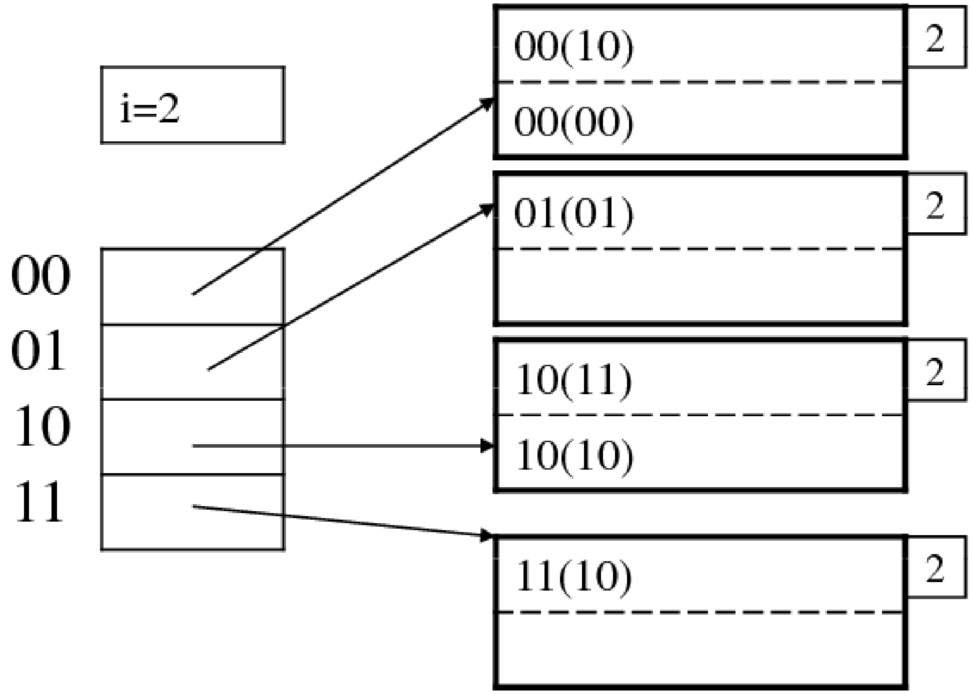
- Now insert 0000, then 0101



- Need to split block

Insertion in Extensible Hash Table

- After splitting the block



Performance Extensible Hash Table

- No overflow blocks: access always one read
- BUT:
  - Extensions can be costly and disruptive

    2X no longer fit,
  - After an extension table may no longer fit in memory

CS411
Database Systems

8: Query Processing

Why Do We Learn This?

Engine

top-3

"performance",
"performance",
"performance".

① Work at G/
M$/Oracle

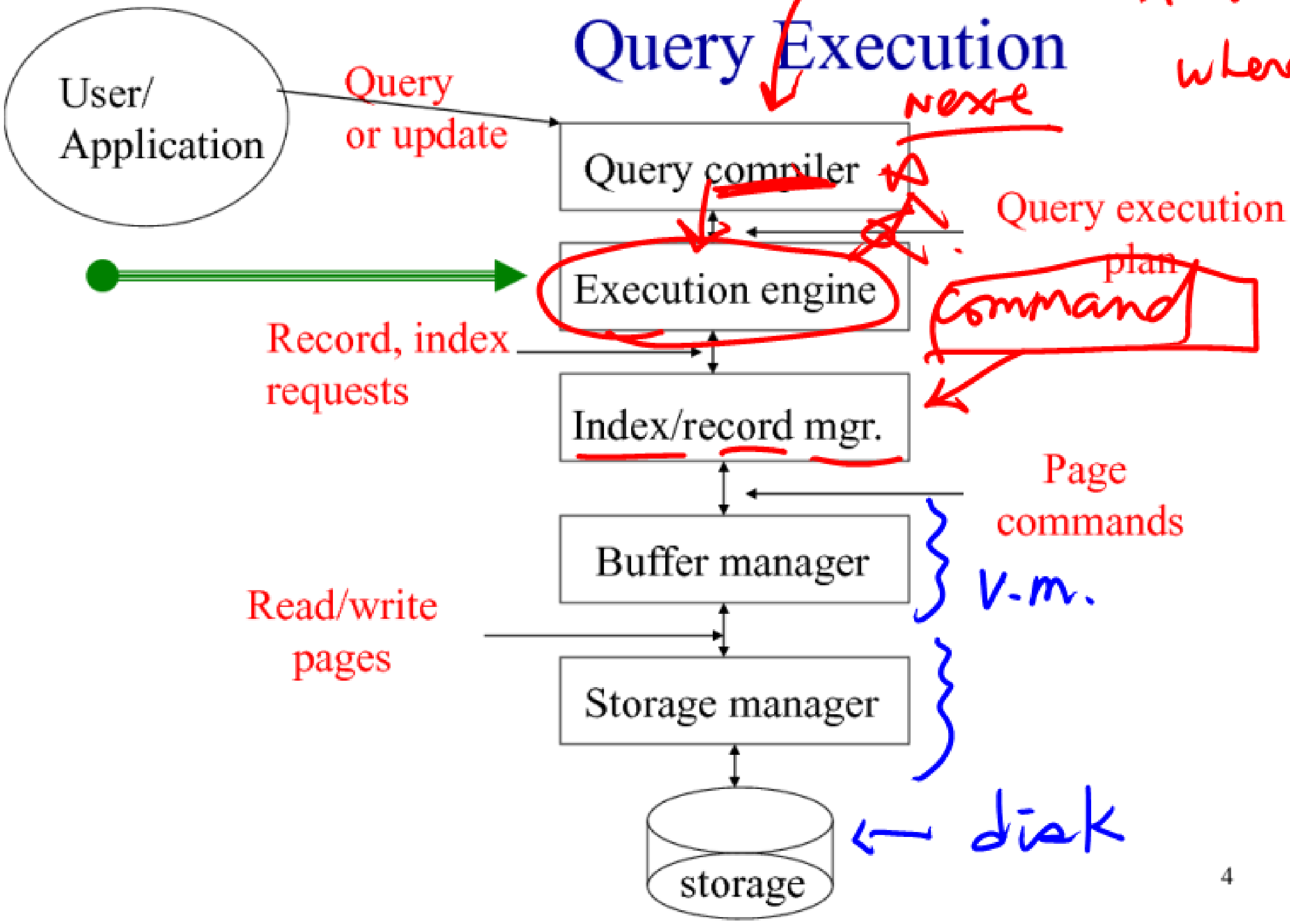
② Insight.
of SQL tune

Outline

- Logical/physical operators
- Cost parameters and sorting
- One-pass algorithms
- Nested-loop joins
- Two-pass algorithms

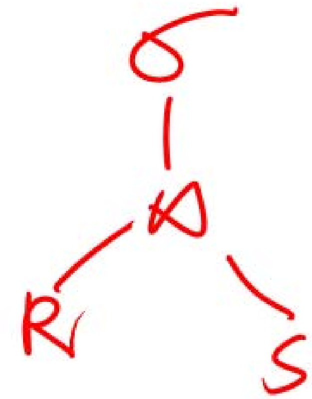
Query Execution

select
from stu
where name
= "Alex"



Logical v.s. Physical Operators

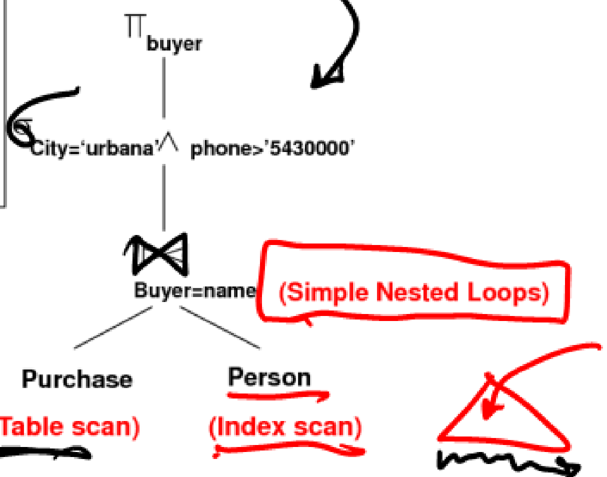
- Logical operators $\bowtie, \sigma, \Pi$. R
  - *what* they do
  - e.g., union, selection, project, join, grouping
- Physical operators
  - *how* they do it $\bowtie$ sort-merge
  - e.g., nested loop join, sort-merge join, hash join, index join



Query Execution Plans

compiler

```
SELECT   S.sname
FROM     Purchase P, Person Q
WHERE    P.buyer=Q.name AND
         Q.city='urbana' AND
         Q.phone > '5430000'
```

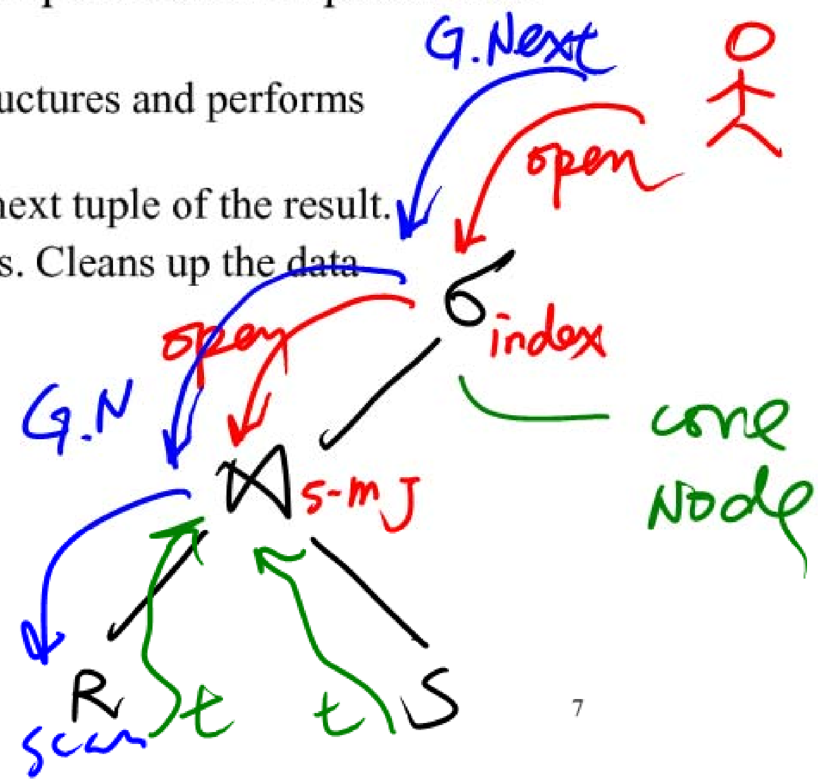


Query Plan:
- logical tree
- implementation choice at every node
- scheduling of operations.

Some operators are from relational algebra, and others (e.g., scan, group) are not.

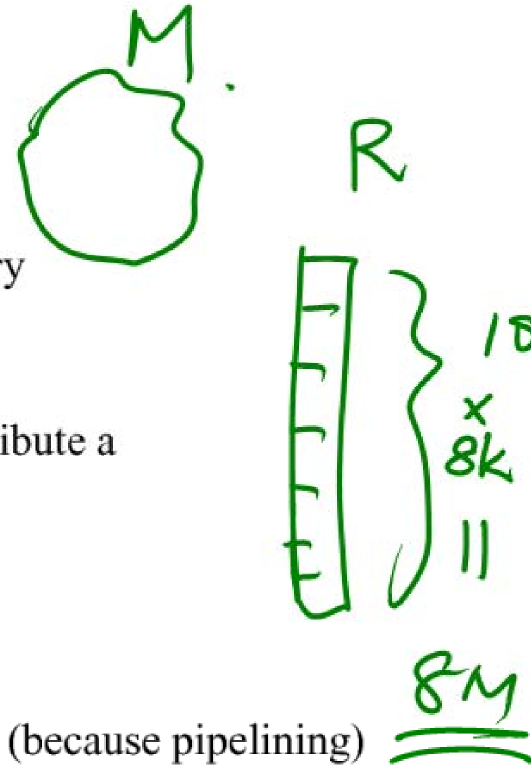
How do We Combine Operations?

- The iterator model. Each operation is implemented by 3 functions:
  - Open: sets up the data structures and performs initializations
  - GetNext: returns the the next tuple of the result.
  - Close: ends the operations. Cleans up the data structures.
- Enables pipelining!



Cost Parameters

- Cost parameters
  - M = number of blocks that fit in main memory
  - B(R) = number of blocks holding R
  - T(R) = number of tuples in R
  - V(R,a) = number of distinct values of the attribute a
- Estimating the cost:
  - Important in optimization (next lecture)
  - Compute I/O cost only
  - We compute the cost to *read* the tables
  - We don't compute the cost to *write* the result (because pipelining)

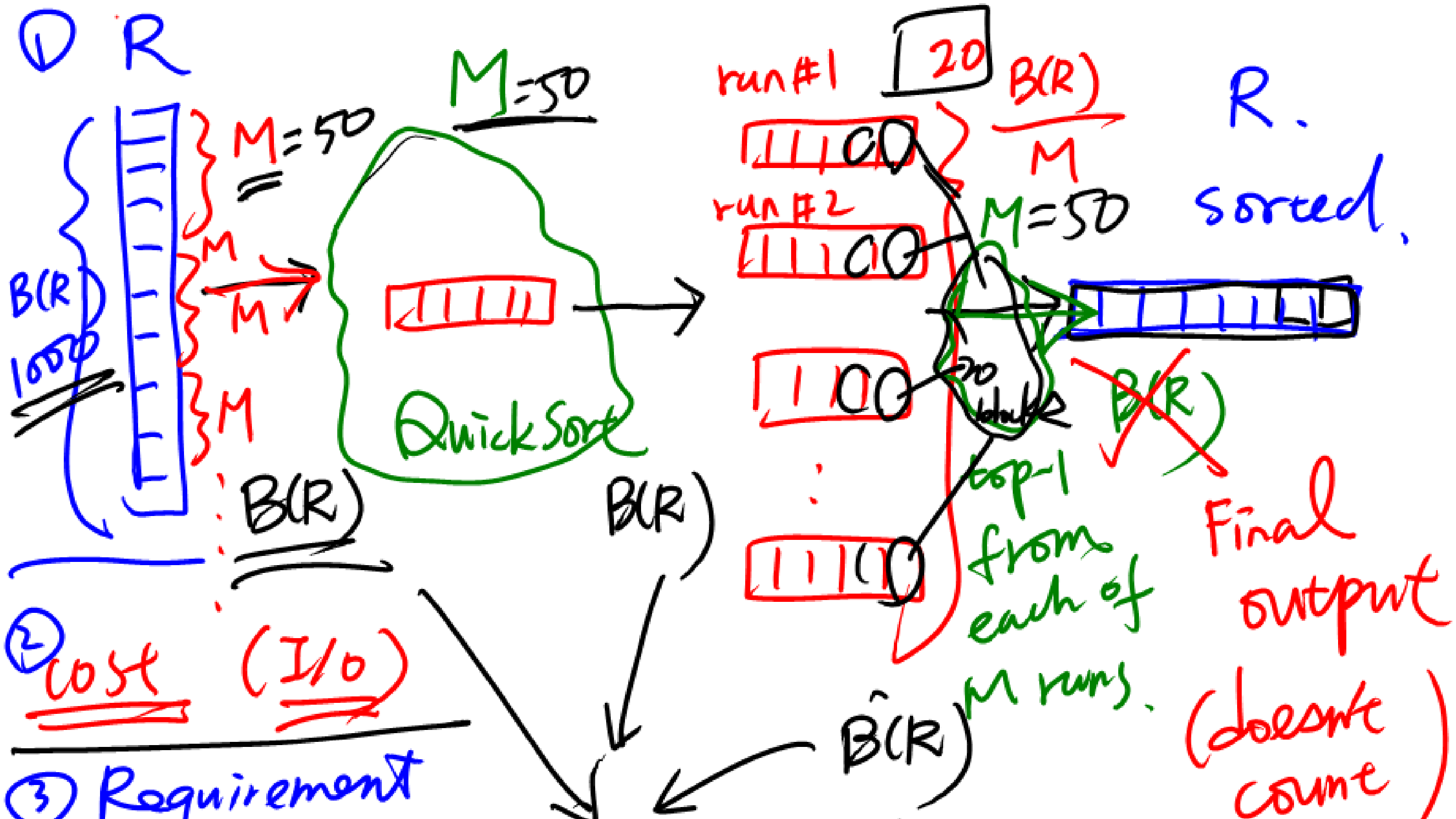


all alg $\boxed{B(R) < M}$ → 1G

$\boxed{\text{Sorting}}$

- Two pass multi-way merge sort
- Step 1:
  - Read M blocks at a time, sort, write
  - Result: have runs of length M on disk
- Step 2:
  - Merge M-1 at a time, write to disk
  - Result: have runs of length $M(M-1) \approx M^2$
- Cost: 3B(R), Assumption: $B(R) \leq M^2$ → $1G^2$

① R



② cost (I/O)

③ Requirement

$20 \le 50$

$\frac{B(R)}{M} \le M \Rightarrow B(R) \le M^2$

$3\,B(R) = 3000$ blocks

Scanning Tables

- The table is *clustered* (I.e. blocks consists only of records from this table): B(R)
  - Table-scan: if we know where the blocks are
  - Index scan: if we have index to find the blocks
- The table is unclustered (e.g. its records are placed on blocks with other tables) T(R)
  - May need one read for each record

$R \bowtie S$

tuples.

10M

T(R)

10m

#blocks

Cost of the Scan Operator

- Clustered relation:
  - Table scan: B(R); to sort: 3B(R)
  - Index scan: B(R); to sort: B(R) or 3B(R)
- Unclustered relation
  - T(R); to sort: T(R) + 2B(R)

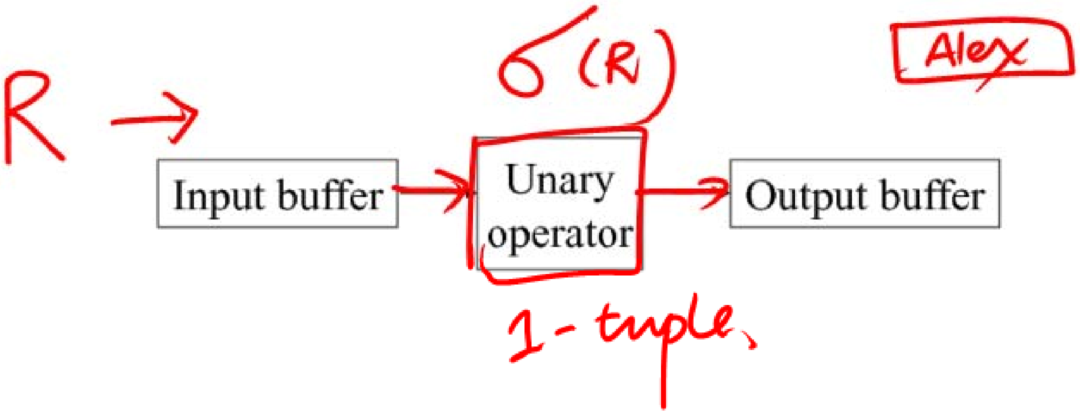
One pass algorithms

One-pass Algorithms

Selection σ(R), projection Π(R)

$\sigma_{name = \text{"axle"}}$ Student

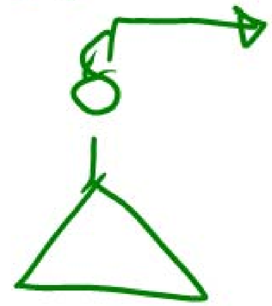
- Both are <u>*tuple-at-a-Time*</u> algorithms
- Cost: B(R)



One-pass Algorithms

Select distinct . . .
From . . .
Where

Duplicate elimination $\delta(R)$



- Need to keep a dictionary in memory:
  - balanced search tree
  - hash table
  - etc

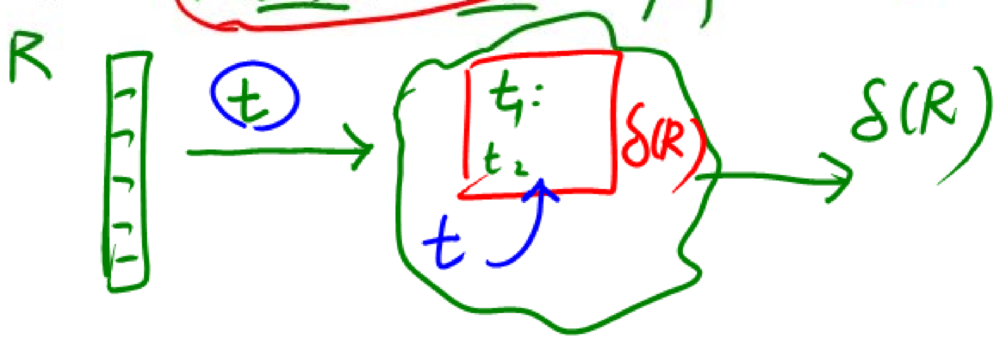
1-pass
exp

- Cost: $B(R)$

1G ~~0.5G~~

- Assumption: $B(\delta(R)) <= M$

$\Rightarrow$ 2-pass



One-pass Algorithms

Grouping: $\gamma_{\text{city, sum(price)}}$ (R)

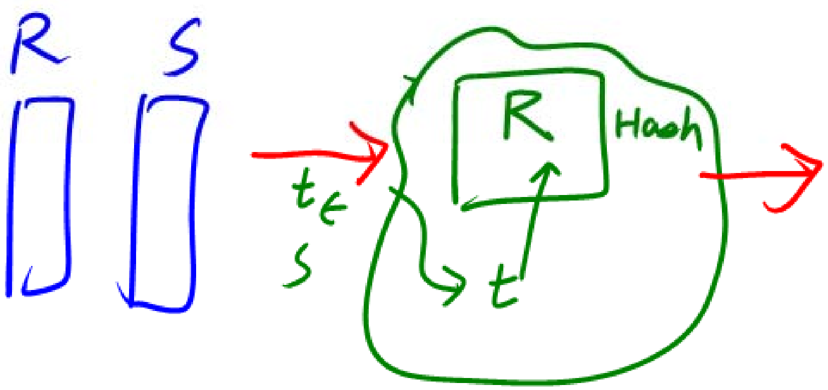
- Need to keep a dictionary in memory
- Also store the sum(price) for each city
- Cost: B(R)
- Assumption: number of cities fits in memory

One-pass Algorithms

Binary operations: $R \cap S$, $R \cup S$, $R - S$

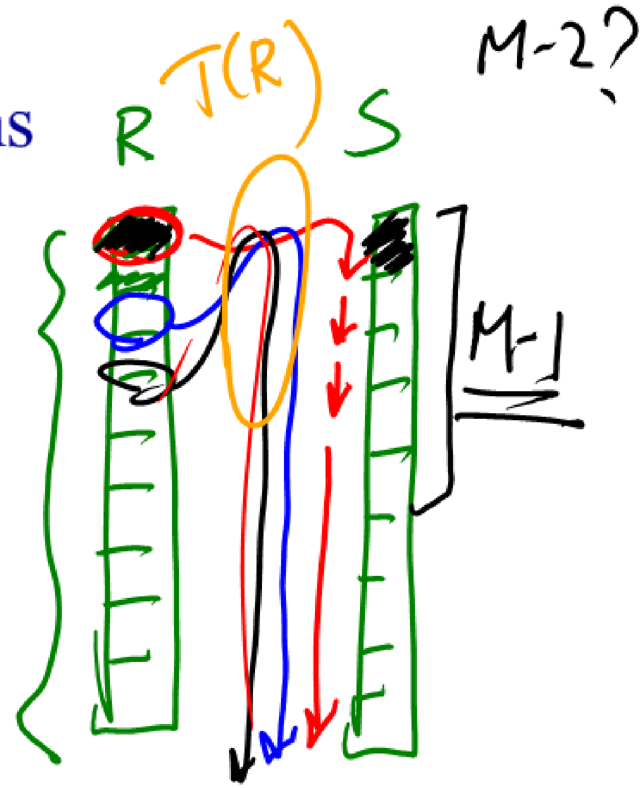
- Assumption: $\min(B(R), B(S)) \le M$
- Scan one table first, then the next, eliminate duplicates
- Cost: $B(R)+B(S)$

$B(R) \le M$. or $B(S) \le M$



Nested Loop Joins

- Tuple-based nested loop $R \bowtie S$
- R=outer relation, S=inner relation



for each tuple r in R **do**

  for each tuple s in S **do**

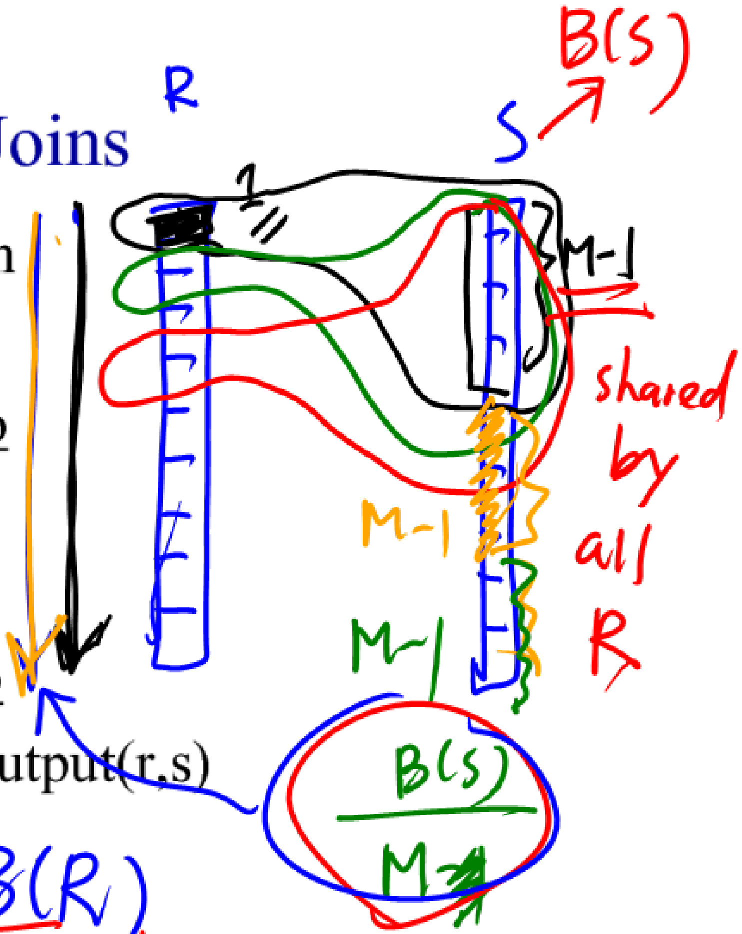
    if r and s join **then** output (r,s)

- Cost: T(R) T(S), sometimes T(R) B(S)

$B(R) + T(R) \times T(S)$ $(B(S))$

Nested Loop Joins

- Block-based Nested Loop Join



for each (M-1) blocks bs of S do

  for each block br of R do

    for each tuple s in bs do

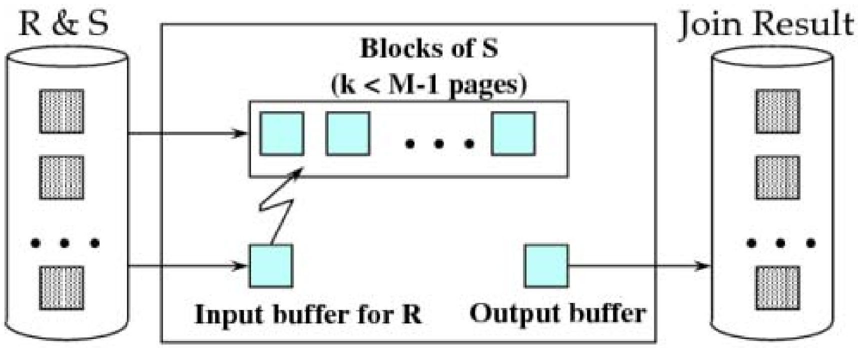
      for each tuple r in br do

        if r and s join then output(r,s)

$$B(S) + \frac{B(S)}{M} \times B(R)$$

want smaller rel to be S

Nested Loop Joins



Nested Loop Joins

- Block-based Nested Loop Join
- Cost:
  - Read S once: cost B(S)
  - Outer loop runs B(S)/(M-1) times, and each time need to read R: costs B(S)B(R)/(M-1)
  - Total cost: B(S) + B(S)B(R)/(M-1)
- Notice: it is better to iterate over the smaller relation first– i.e., S smaller