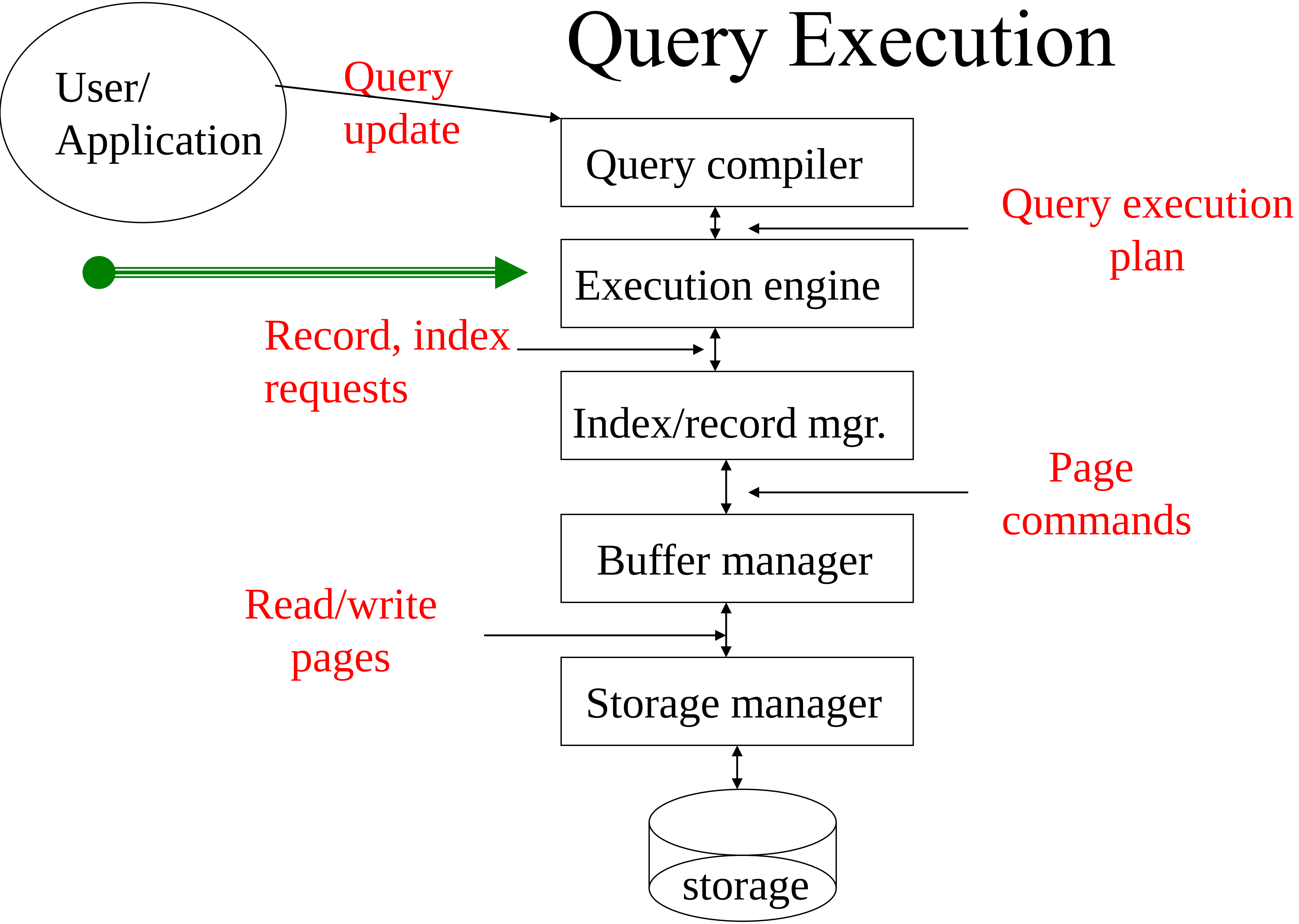
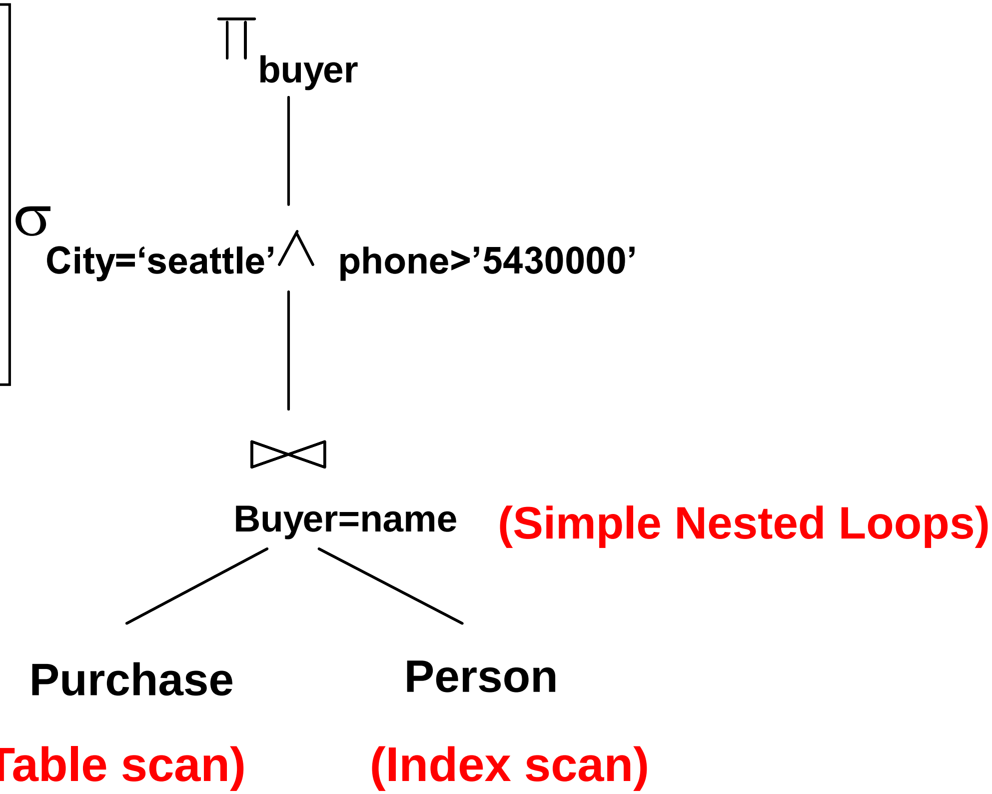
DBMS Internals
Execution and Optimization

Query Execution



Query Execution Plans

```
SELECT  buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      Q.city='seattle' 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.

The Leaves of the Plan: Scans

- Table scan: iterate through the records of the relation.
- Index scan: go to the index, from there get the records in the file (when would this be better?)
- Sorted scan: produce the relation in order. Implementation depends on relation size.

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!
- Contrast with data-driven materialize model.
- Sometimes it's the same (e.g., sorted scan).

Implementing Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Selects a subset of rows from relation.
  - *Projection* (π) Deletes unwanted columns from relation.
  - ***Join*** ($\bowtie$) Allows us to combine two relations.
  - *Set-difference* Tuples in reln. 1, but not in reln. 2.
  - *Union* Tuples in reln. 1 and in reln. 2.
  - *Aggregation* (SUM, MIN, etc.) and GROUP BY

Schema for Examples

Purchase (*buyer*:string, *seller*: string, *product*: integer),

Person (*name*:string, *city*:string, *phone*: integer)

- Purchase:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages (i.e., 100,000 tuples, 4MB for the entire relation).
- Person:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages (i.e, 40,000 tuples, 2MB for the entire relation).

Simple Selections

```
SELECT *
FROM Person R
WHERE R.phone < '543%'
```

- Of the form $$\sigma_{R.attr\ op\ value}(R)$$
- With no index, unsorted: Must essentially scan the whole relation; cost is M (#pages in R).
- With an index on selection attribute: Use index to find qualifying data entries, then retrieve corresponding data records. (Hash index useful only for equality selections.)
- Result size estimation:

  (Size of R) * reduction factor.

  More on this later.

Using an Index for Selections

- Cost depends on #qualifying tuples, and clustering.
  - Cost of finding qualifying data entries (typically small) plus cost of retrieving records.
  - In example, assuming uniform distribution of phones, about 54% of tuples qualify (250 pages, 20000 tuples). With a clustered index, cost is little more than 250 I/Os; if unclustered, up to 20000 I/Os!
- *Important refinement for unclustered indexes*:
  1. Find and sort the rid's of the qualifying data entries.
  2. Fetch rids in order. This ensures that each data page is looked at just once (though # of such pages likely to be higher than with clustering).

Two Approaches to General Selections

- First approach: Find the *most selective access path*, retrieve tuples using it, and apply any remaining terms that don't match the index:
  - *Most selective access path:* An index or file scan that we estimate will require the fewest page I/Os.
  - Consider *city="seattle AND phone<"543%" :*
    - A hash index on *city* can be used; then, *phone*<"543%" must be checked for each retrieved tuple.
    - Similarly, a b-tree index on *phone* could be used; *city="seattle"* must then be checked.

Intersection of Rids

- Second approach
  - Get sets of rids of data records using each matching index.
  - Then *intersect* these sets of rids.
  - Retrieve the records and apply any remaining terms.

Implementing Projection

```
SELECT   DISTINCT
         R.name,
         R.phone
FROM Person R
```

- Two parts:

  (1) remove unwanted attributes,

  (2) remove duplicates from the result.
- Refinements to duplicate removal:
  - If an index on a relation contains all wanted attributes, then we can do an *index-only* scan.
  - If the index contains a subset of the wanted attributes, you can remove duplicates *locally*.

Equality Joins With One Join Column

```
SELECT *
FROM Person R, Purchase S
WHERE R.name=S.buyer
```

JOIN

- $R \bowtie S$ is a common operation. The cross product is too large. Hence, performing $R \times S$ and then a selection is too inefficient.
- Assume: M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.
  - In our examples, R is Person and S is Purchase.
- *Cost metric*: # of I/Os. We will ignore output costs.

Discussion

- How would you implement join?

Estimating IOs:

- Count # of disk blocks that must be read (or written) to execute query plan

To estimate costs, we may have additional parameters:

B(R) = # of blocks containing R tuples

f(R) = max # of tuples of R per block

M = # memory blocks available

To estimate costs, we may have additional parameters:

T(R) : # tuples in R

S(R) : # of bytes in each R tuple

B(R) = # of blocks containing R tuples

f(R) = max # of tuples of R per block

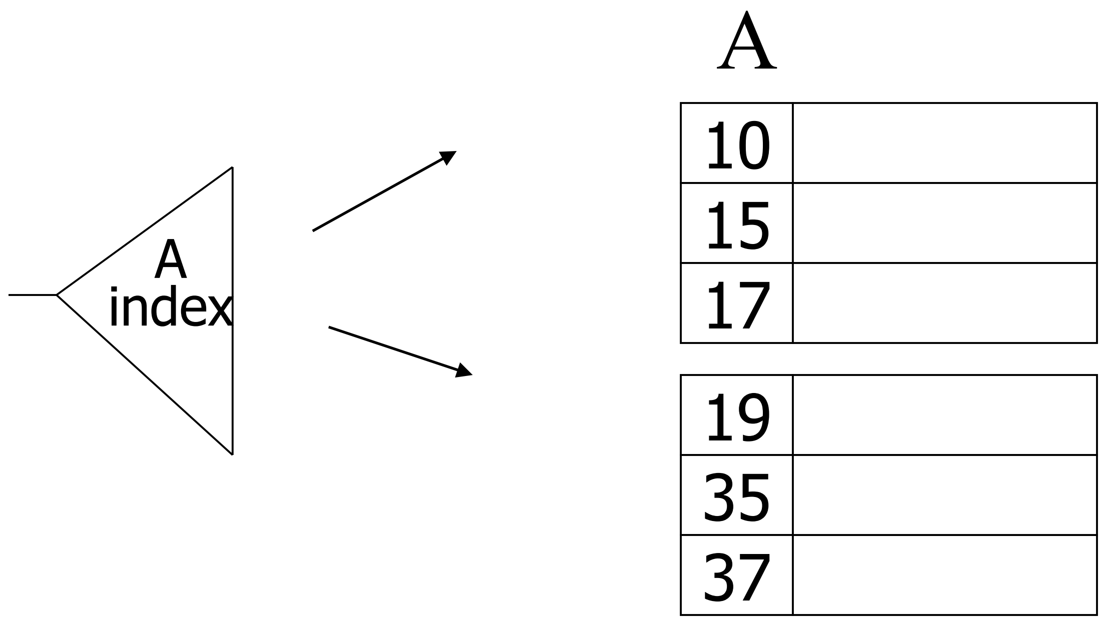
M = # memory blocks available

HT(i) = # levels in index i

LB(i) = # of leaf blocks in index i

Clustering index

Index that allows tuples to be read in an order that corresponds to physical order



Notions of clustering

- Clustered file organization

R1 R2 S1 S2	R3 R4 S3 S4	…..

- Clustered relation

R1 R2 R3 R4	R5 R6 R7 R8	…..

- Clustering index

<u>Example</u> R1 $\bowtie$ R2 over common attribute C

T(R1) = 10,000

T(R2) = 5,000

S(R1) = S(R2) = 1/10 block (per tuple)

Memory available = 101 blocks

Example $R1 \bowtie R2$ over common attribute C

$T(R1) = 10{,}000$

$T(R2) = 5{,}000$

$S(R1) = S(R2) = 1/10$ block

Memory available = 101 blocks

$\rightarrow$ Metric: # of IOs

(ignoring writing of result)

Options

- Transformations: R1 $\bowtie$ R2, R2 $\bowtie$ R1
- Joint algorithms:
  - Iteration (nested loops)
  - Merge join
  - Join with index
  - Hash join

- Iteration join (conceptually)

  for each r $\in$ R1 do

    for each s $\in$ R2 do

      if r.C = s.C then output r,s pair

- Merge join (conceptually)

  (1) if R1 and R2 not sorted, sort them

  (2) $i \leftarrow 1$; $j \leftarrow 1$;

  While $(i \le T(R1)) \wedge (j \le T(R2))$ do

  if R1{ i }.C = R2{ j }.C then outputTuples

  else if R1{ i }.C > R2{ j }.C then $j \leftarrow j+1$

  else if R1{ i }.C < R2{ j }.C then $i \leftarrow i+1$

Procedure Output-Tuples

While $(R1\{ i \}.C = R2\{ j \}.C) \wedge (i \leq T(R1))$ do

[$jj \leftarrow j$;

while $(R1\{ i \}.C = R2\{ jj \}.C) \wedge (jj \leq T(R2))$ do

[output pair $R1\{ i \}$, $R2\{ jj \}$;

$jj \leftarrow jj+1$]

$i \leftarrow i+1$]

Example

i	R1{i}.C	R2{j}.C	j
1	10	5	1
2	20	20	2
3	20	20	3
4	30	30	4
5	40	30	5
		50	6
		52	7

- Join with index (Conceptually)

Assume R2.C index

For each r $\in$ R1 do

[X $\leftarrow$ index (R2, C, r.C)

for each s $\in$ X do

output r,s pair]

Note: X $\leftarrow$ index(rel, attr, value)

then X = set of rel tuples with attr = value

- Hash join (conceptual)
  - Hash function h, range $0 \rightarrow k$
  - Buckets for R1: G0, G1, ... Gk
  - Buckets for R2: H0, H1, ... Hk

- Hash join (conceptual)
  - Hash function h, range 0 $\rightarrow$ k
  - Buckets for R1: G0, G1, ... Gk
  - Buckets for R2: H0, H1, ... Hk

<u>Algorithm</u>

(1) Hash R1 tuples into G buckets

(2) Hash R2 tuples into H buckets

(3) For i = 0 to k do

    match tuples in Gi, Hi buckets

Simple example   hash: even/odd

R1	R2
2	5
4	4
3	12
5	3
8	13
9	8
	11
	14

Buckets

	R1	R2
Even	2 4 8	4 12 8 14
Odd:	3 5 9	5 3 13 11

Factors that affect performance

(1) Tuples of relation stored
physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

Simple Nested Loops Join

For each tuple r in R do
  for each tuple s in S do
    if $r_i == s_j$ then add <r, s> to result

- For each tuple in the *outer* relation R, we scan the entire *inner* relation S.
  - Cost: $M + (p_R * M) * N = 1000 + 100*1000*500$ I/Os: **140 hours!**
- Page-oriented Nested Loops join: For each *page* of R, get each *page* of S, and write out matching pairs of tuples <r, s>, where r is in R-page and S is in S-page.
  - Cost: $M + M*N = 1000 + 1000*500$ (**1.4 hours**)

M pages in R, p_R tuples per page, N pages in S, p_S tuples per page.

Assume: M=1000; N=500; p_R=100; 100 blocks/sec could be read

Example 1(a) Iteration Join R1 $\bowtie$ R2

- Relations not contiguous
- Recall $\begin{cases} T(R1) = 10{,}000 \quad T(R2) = 5{,}000 \\ S(R1) = S(R2) = 1/10 \text{ block} \\ MEM = 101 \text{ blocks} \end{cases}$

Example 1(a) Iteration Join R1 $\bowtie$ R2

- Relations not contiguous
- Recall $\left\{ \begin{array}{l} T(R1) = 10{,}000 \quad T(R2) = 5{,}000 \\ S(R1) = S(R2) = 1/10 \text{ block} \\ MEM = 101 \text{ blocks} \end{array} \right.$

Cost: for each R1 tuple:

[Read tuple + Read R2]

Total =10,000 [1+5000]=50,010,000 IOs

- Can we do better?

<u>Use our memory</u>

(1) Read 100 blocks of R1

(2) Read all of R2 (using 1 block) + join

(3) Repeat until done

<u>Example 1(b)</u> Iteration Join R1 $\bowtie$ R2

- Relations contiguous
- T(R1) = 10,000 T(R2) = 5,000
- S(R1) = S(R2) =1/10 block

<u>Cost</u>

For each R1 chunk:

Read chunk: 100 IOs

Read R2: <u>500 </u>IOs

600

Total= 10chunks x 600 = 6000 IOs

Example 1(b) Iteration Join R2 $\bowtie$ R1

- Relations contiguous

Example 1(b) Iteration Join R2 $\bowtie$ R1

- Relations contiguous

Cost

For each R2 chunk:

Read chunk: 100 IOs

Read R1: $\underline{1000}$ IOs

1,100

Total= 5 chunks x 1,100 = 5,500 IOs

Index Nested Loops Join

foreach tuple r in R do
    foreach tuple s in S where $r_i == s_j$ do
        add <r, s> to result

- If there is an index on the join column of one relation (say S), can make it the inner.
  - Cost: $M + ((M*p_R) *$ cost of finding matching S tuples)
- For each R tuple, cost of probing S index is about 1.2 for hash index, 2-4 for B+ tree. Cost of then finding S tuples depends on clustering.
  - Clustered index: 1 I/O (typical), unclustered: up to 1 I/O per matching S tuple.

Examples of Index Nested Loops

- Hash-index on *name* of Person (as inner):
  - Scan Purchase: 1000 page I/Os, 100*1000 tuples.
  - For each Person tuple: 1.2 I/Os to get data entry in index, plus 1 I/O to get (the exactly one) matching Person tuple. Total: 1000+2.2*100000=221,000 I/Os. (**36 minutes**)
- Hash-index on *buyer* of Purchase (as inner):
  - Scan Person: 500 page I/Os, 80*500 tuples.
  - For each Person tuple: 1.2 I/Os to find index page with data entries, plus cost of retrieving matching Purchase tuples. Assuming uniform distribution, 2.5 purchases per buyer (100,000 / 40,000). Cost of retrieving them is 1 or 2.5 I/Os depending on clustering.
  - Total: 500+40000*(1.2+1)(**14 minutes, clustered**)

    max 500+40000*(1.2+2.5)(**24 minutes, unclustered**)

Example 1(e) Index Join

- Assume R1.C index exists; 2 levels
- Assume R2 contiguous, unordered

- Assume R1.C index fits in memory

Cost: Reads: 500 IOs

for each R2 tuple:

- probe index - free
- if match, read R1 tuple: 1 IO

Recall

- T(R1) = 10,000   T(R2) = 5,000
- S(R1) = S(R2) =1/10 block

Selection cardinality

SC(R,A) = average # records that satisfy equality condition on R.A

$$SC(R,A) = \begin{cases} \dfrac{T(R)}{V(R,A)} \\ \\ \dfrac{T(R)}{DOM(R,A)} \end{cases}$$

What is expected # of matching tuples?

(a) say R1.C is key, R2.C is foreign key

then expect = 1

(b) say V(R1,C) = 5000, T(R1) = 10,000

with uniform assumption

expect = 10,000/5,000 = 2

V(R, A) : # distinct values in R for attribute A

What is expected # of matching tuples?

(c) Say DOM(R1, C)=1,000,000

$T(R1) = 10{,}000$

with alternate assumption

$$\text{Expect} = \frac{10{,}000}{1{,}000{,}000} = \frac{1}{100}$$

Total cost with index join

(a) Total cost = 500+5000(1)1 = 5,500

(b) Total cost = 500+5000(2)1 = 10,500

(c) Total cost = 500+5000(1/100)1=550

What if index does not fit in memory?

Example: say R1.C index is 201 blocks

Recall: Assume R1.C index exists; 2 levels

- Keep root + 99 leaf nodes in memory
- Expected cost of each probe is

$$E = (0)\frac{99}{200} + (1)\frac{101}{200} \approx 0.5$$

Total cost (including probes)

= 500+5000 [Probe + get records]

= 500+5000 [0.5+2] uniform assumption

= 500+12,500 = 13,000 (case b)

Total cost (including probes)

= 500+5000 [Probe + get records]

= 500+5000 [0.5+2] uniform assumption

= 500+12,500 = 13,000 (case b)

For case (c):

= 500+5000[0.5 $\times$ 1 + (1/100) $\times$ 1]

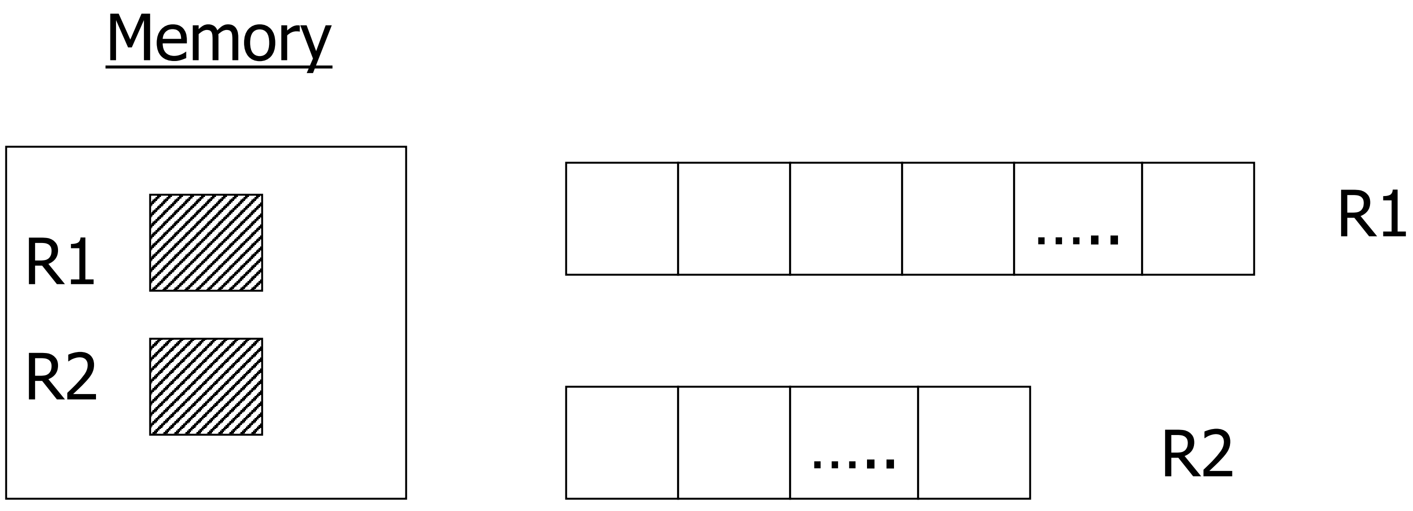
= 500+2500+50 = 3050 IOs

Sort-Merge Join $(R \bowtie_{i=j} S)$

- Sort R and S on the join column, then scan them to do a ``merge” on the join column.
  - Advance scan of R until current R-tuple >= current S tuple, then advance scan of S until current S-tuple >= current R tuple; do this until current R tuple = current S tuple.
  - At this point, all R tuples with same value and all S tuples with same value *match*; output <r, s> for all pairs of such tuples.
  - Then resume scanning R and S.

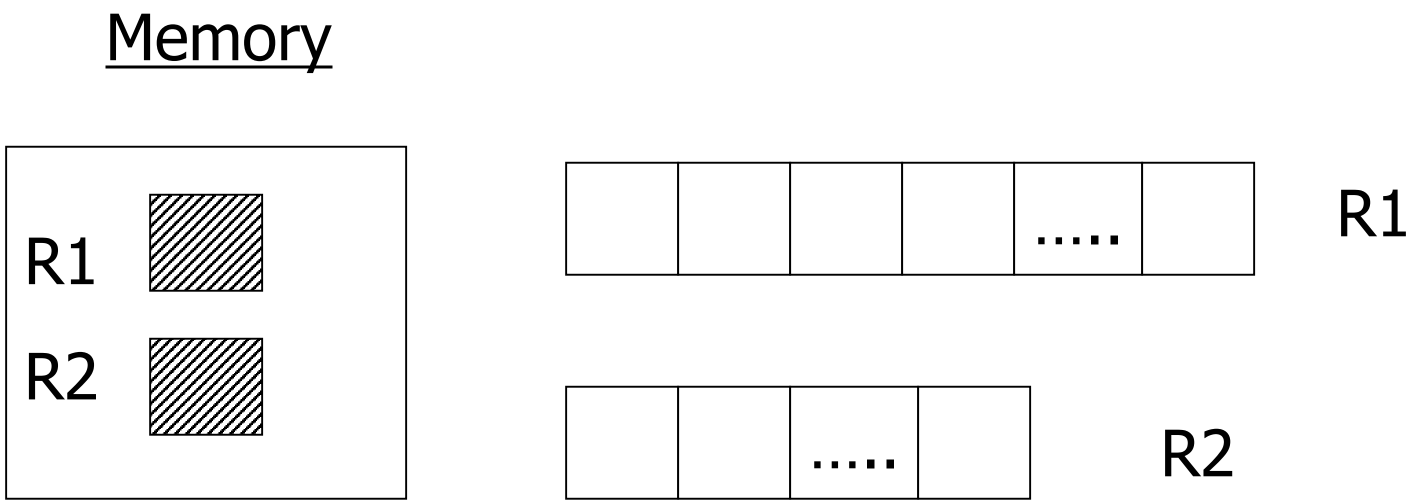
Example 1(c)   Merge Join

- Both R1, R2 ordered by C; relations contiguous



Example 1(c)   Merge Join

- Both R1, R2 ordered by C; relations contiguous



Total cost: Read R1 cost + read R2 cost

= 1000 + 500 = 1,500 IOs

Example 1(d)   Merge Join

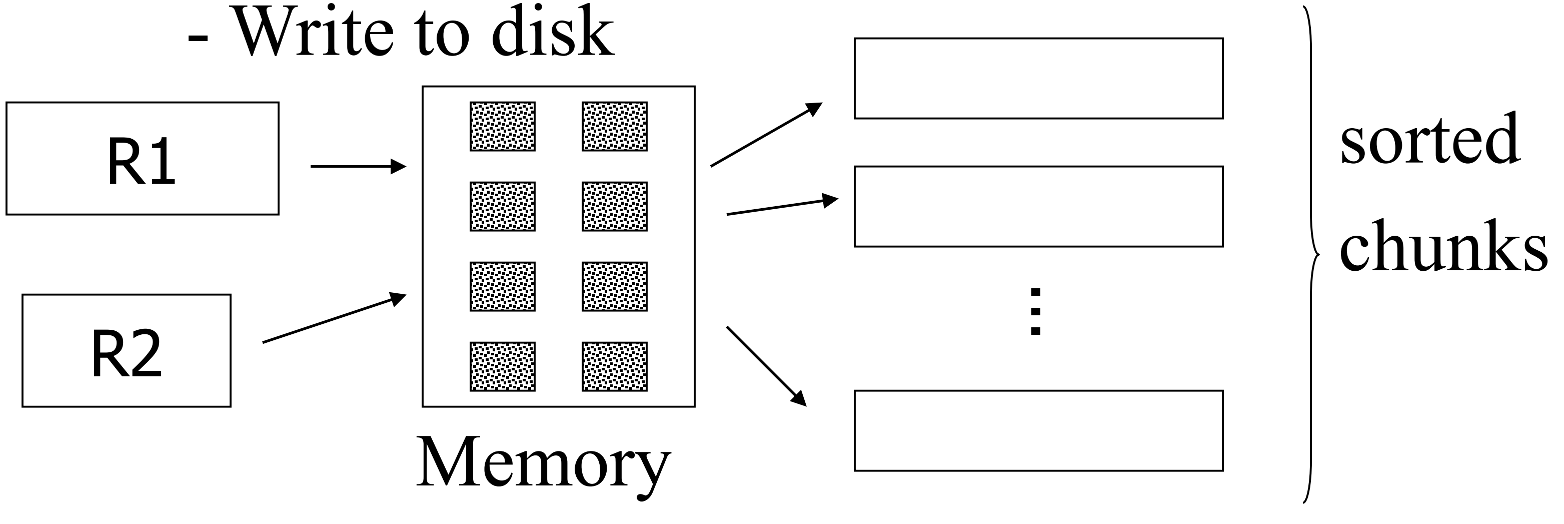
- R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first…. HOW?

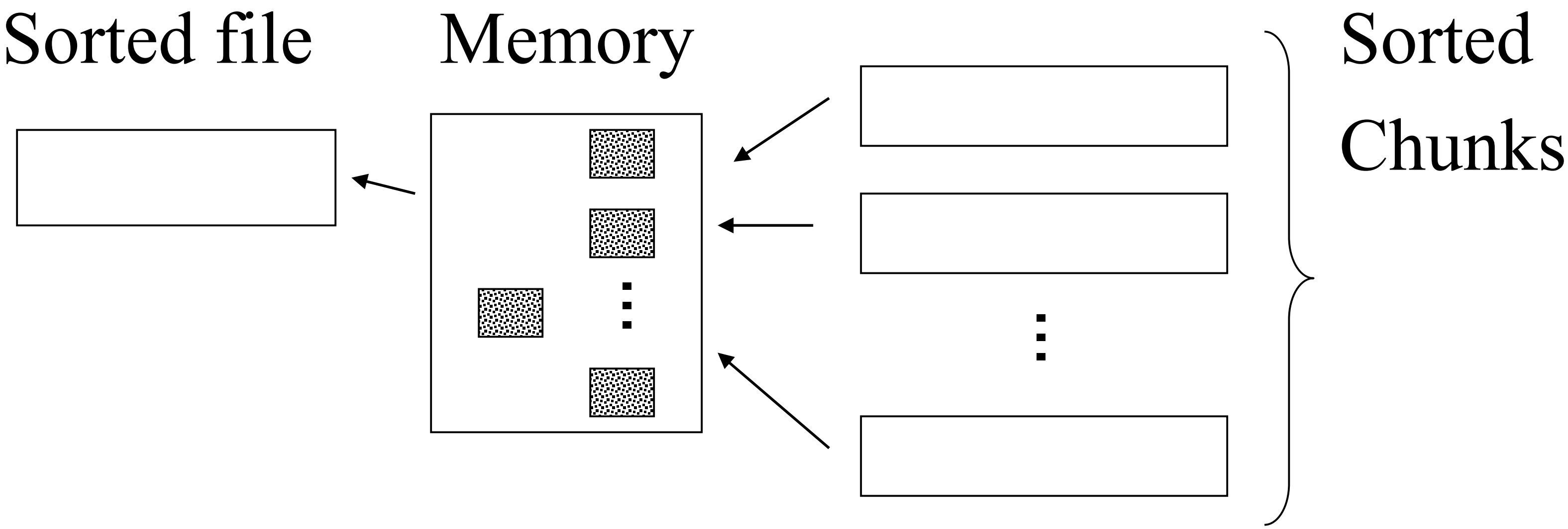
One way to sort: Merge Sort

(i) For each 100 blk chunk of R:

- Read chunk
- Sort in memory
- Write to disk



(ii) Read all chunks + merge + write out



Cost: Sort

Each tuple is read,written,
read, written

so...

Sort cost R1: 4 x 1,000 = 4,000

Sort cost R2: 4 x 500 = 2,000

Example 1(d) Merge Join (continued)

R1,R2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

Example 1(d) Merge Join (continued)

R1,R2 contiguous, but unordered

Total cost = sort cost + join cost

= 6,000 + 1,500 = 7,500 IOs

<u>But:</u> Iteration cost = 5,500

so merge joint does not pay off!

But say R1 = 10,000 blocks

R2 = 5,000 blocks

Contiguous， not ordered

Iterate: $\frac{5000}{100}$ x (100+10,000) = 50 x 10,100

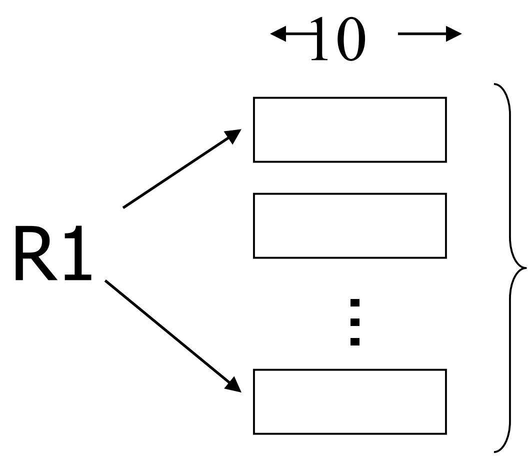
= 505,000 IOs

Merge join: 5(10,000+5,000) = 75,000 IOs

Merge Join (with sort) WINS!

How much memory do we need for merge sort?

E.g: Say I have 10 memory blocks



100 chunks $\Rightarrow$ to merge, need 100 blocks!

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k)   size of chunk = k

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k)   size of chunk = k

chunks $\leq$ buffers available for merge

In general:

Say k blocks in memory

x blocks for relation sort

chunks = (x/k)   size of chunk = k

\# chunks $\leq$ buffers available for merge

so... $(x/k) \leq k$

or $k^2 \geq x$ or $k \geq \sqrt{x}$

In our example

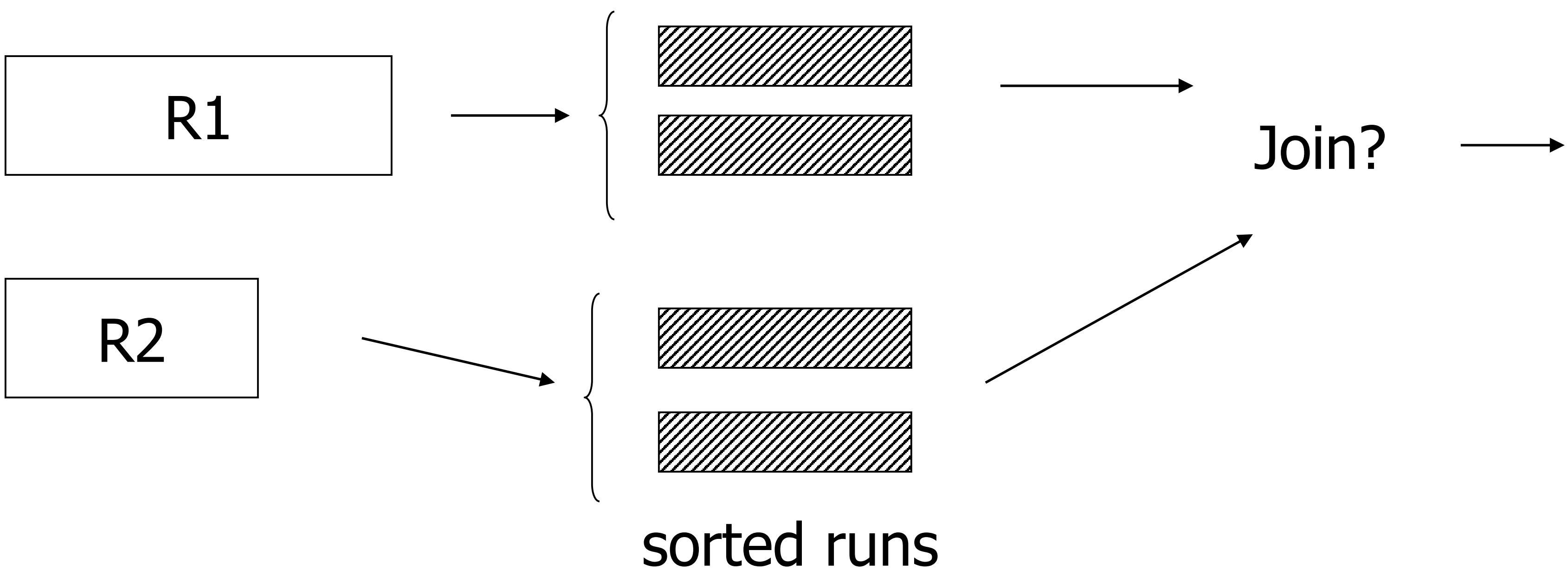
R1 is 1000 blocks, $k \geq 31.62$

R2 is 500 blocks, $k \geq 22.36$

Need at least 32 buffers

Can we improve on merge join?

Hint: do we really need the fully sorted files?



Cost of improved merge join:

C = Read R1 + write R1 into runs

+ read R2 + write R2 into runs

+ join

$= 2 \times 1000 + 2 \times 500 + 1500 = 4500$

--> Memory requirement?

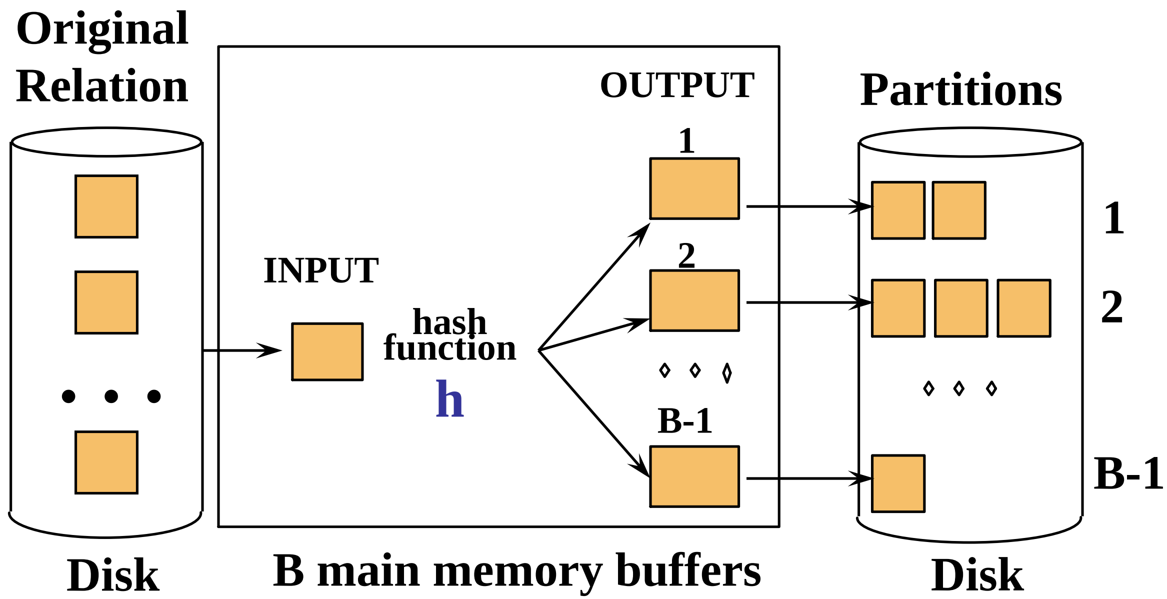
So far

contiguous

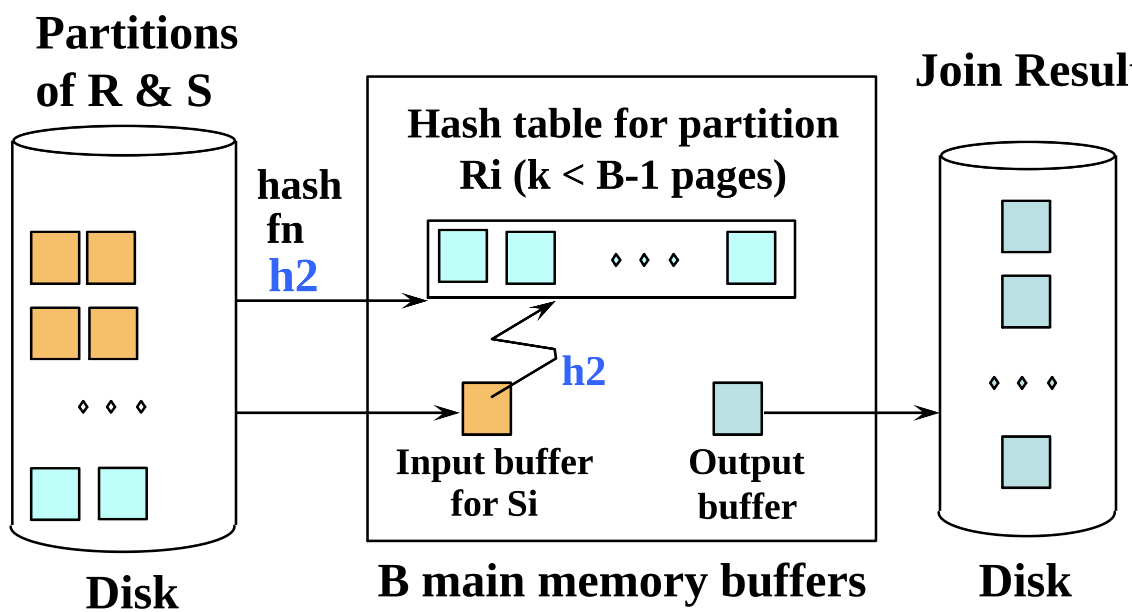
Iterate R2 $\bowtie$ R1	5500
Merge join	1500
Sort+Merge Join	7500 $\rightarrow$ 4500
R1.C Index	5500 $\rightarrow$ 3050 $\rightarrow$ 550
R2.C Index	________

Hash-Join

- Partition both relations using hash fn **h**: R tuples in partition i will only match S tuples in partition i.



❖ Read in a partition of R, hash it using h2 (<> **h**!). Scan matching partition of S, search for matches.



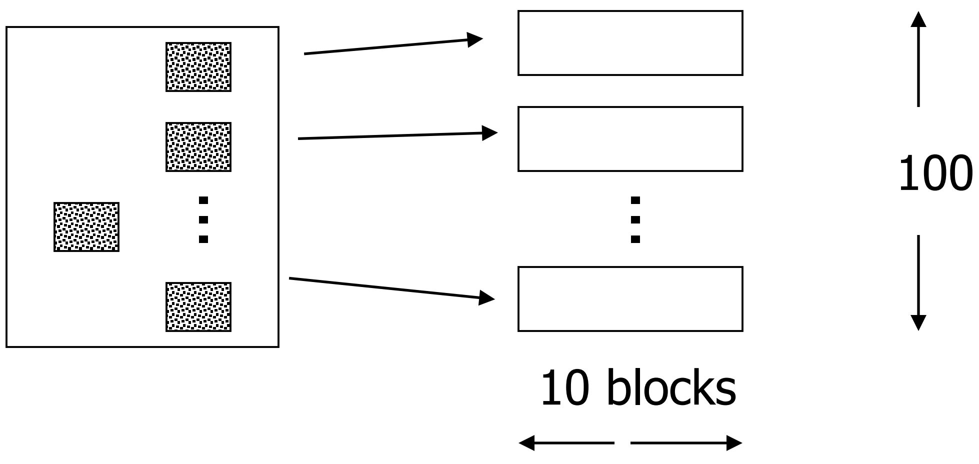
Example 1(f)   Hash Join

- R1, R2 contiguous (un-ordered)

→ Use 100 buckets

→ Read R1, hash, + write buckets

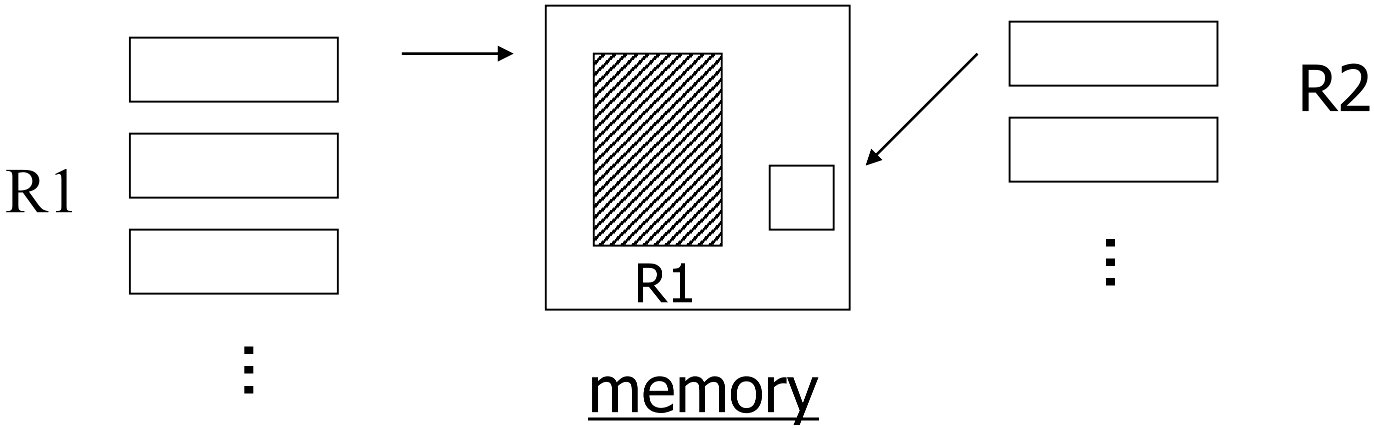
R1 →



-> Same for R2

-> Read one R1 bucket; build memory hash table

-> Read corresponding R2 bucket + hash probe



Then repeat for all buckets

Cost:

"Bucketize:" Read R1 + write

Read R2 + write

Join: Read R1, R2

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

Cost:

"Bucketize:" Read R1 + write

Read R2 + write

Join: Read R1, R2

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

Minimum memory requirements:

Size of R1 bucket = (x/k)

k = number of memory buffers

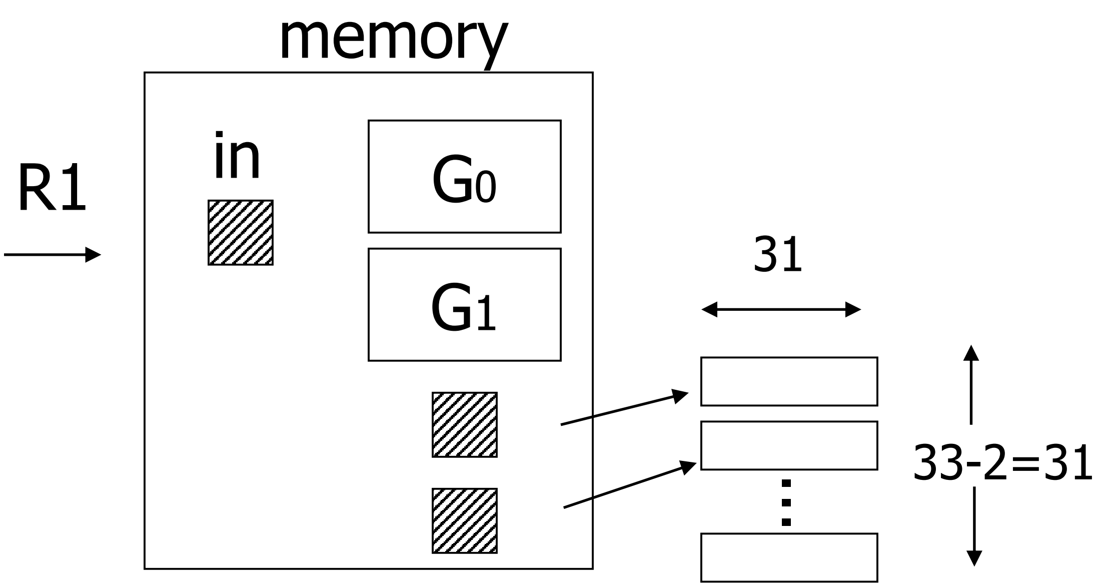
x = number of R1 blocks

So... $(x/k) < k$

$k > \sqrt{x}$ need: $k+1$ total memory buffers

<u>Trick:</u> keep some buckets in memory

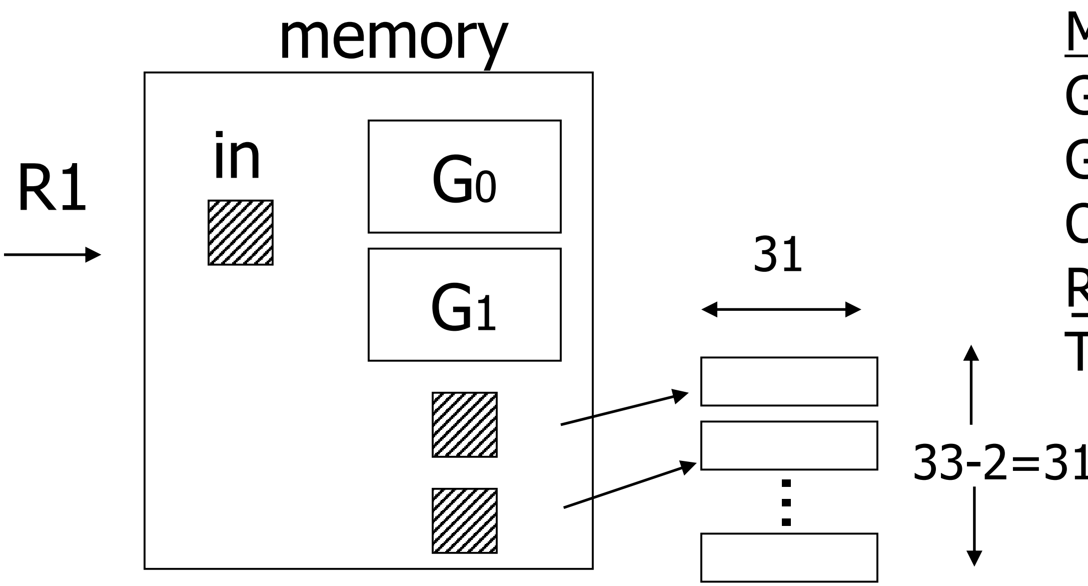
E.g., k'=33 R1 buckets = 31 blocks
keep 2 in memory



called hybrid hash-join

Trick: keep some buckets in memory

E.g., k'=33 R1 buckets = 31 blocks
keep 2 in memory



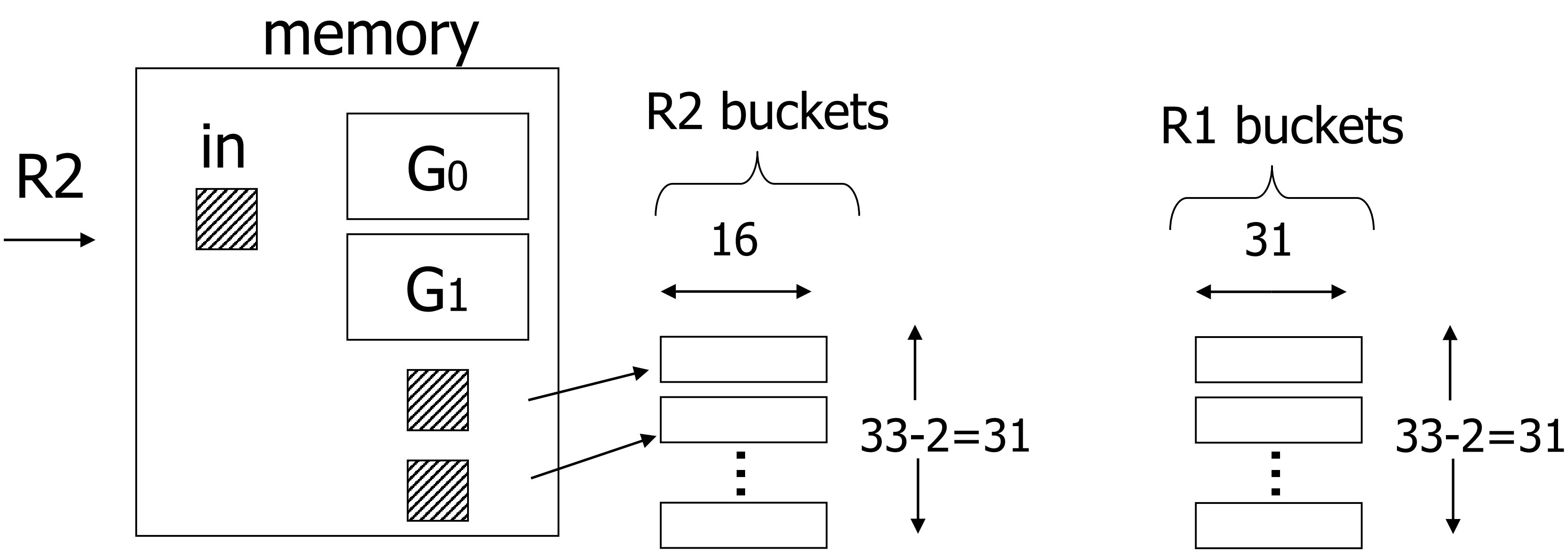
Memory use:	
G0	31 buffers
G1	31 buffers
Output	33-2 buffers
R1 input	1
Total	94 buffers

6 buffers to spare!!

called hybrid hash-join

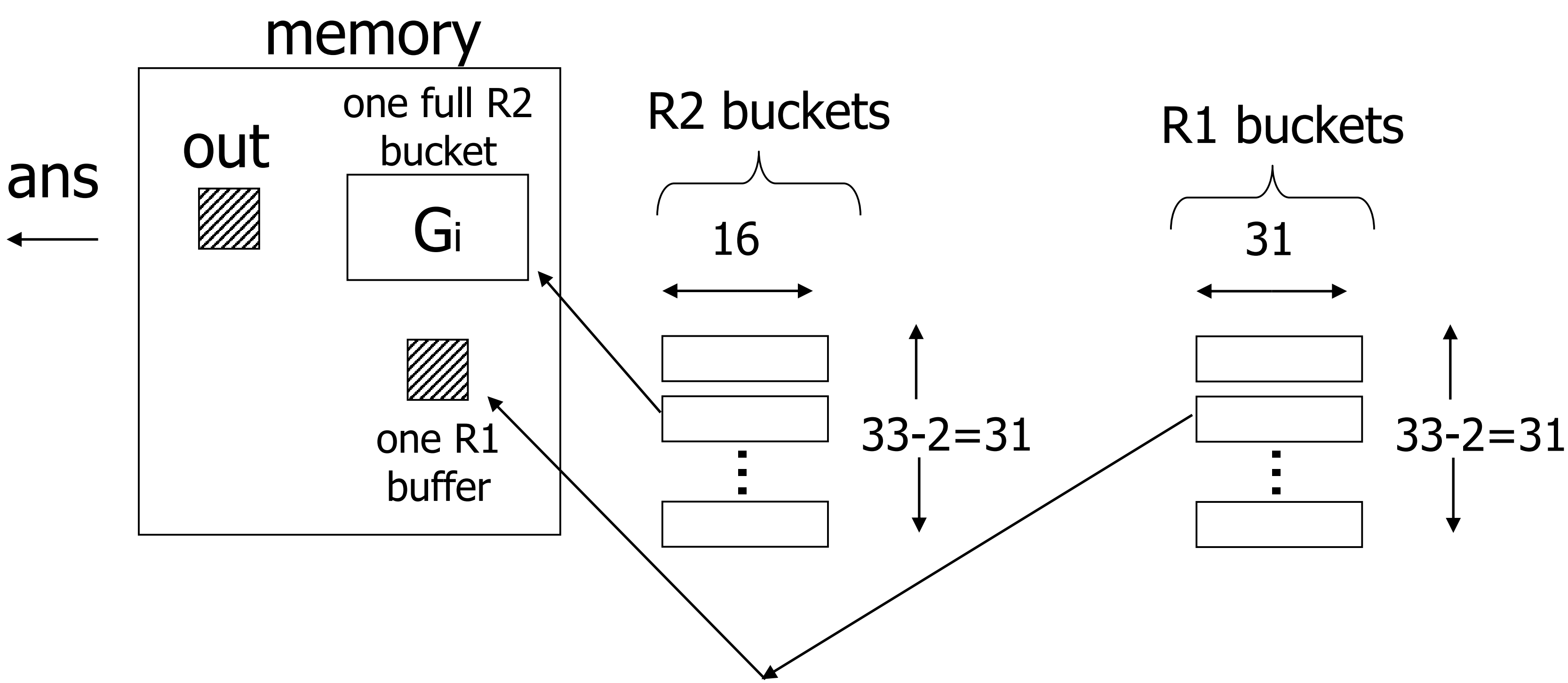
Next: Bucketize R2

- R2 buckets =500/33= 16 blocks
- Two of the R2 buckets joined immediately with G0,G1



Finally: Join remaining buckets

- for each bucket pair:
  - read one of the buckets into memory
  - join with second bucket

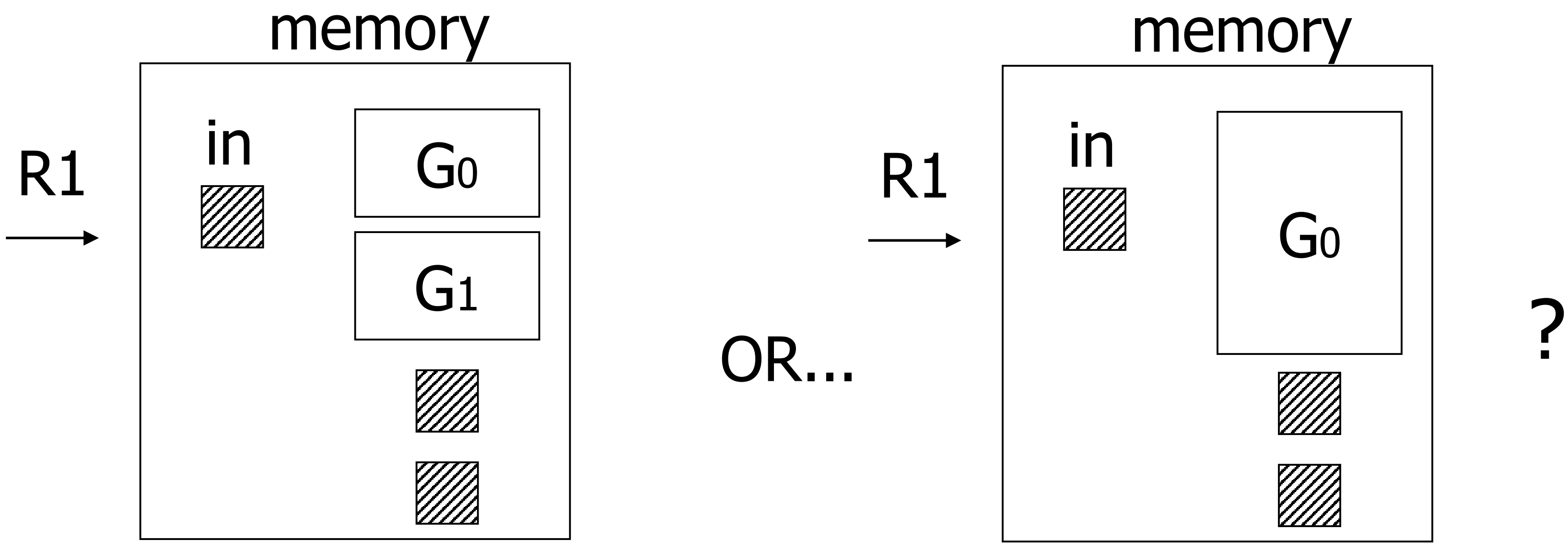


Cost

- Bucketize R1 = 1000+31×31=1961
- To bucketize R2, only write 31 buckets: so, cost = 500+31×16=996
- To compare join (2 buckets already done)
  read 31×31+31×16=1457

Total cost = 1961+996+1457 = 4414

• <u>How many buckets in memory?</u>



☒ See textbook for answer...

Another hash join trick:

- Only write into buckets
  <val,ptr> pairs
- When we get a match in join phase,
  must fetch tuples

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R2 in memory
  5000 tuples $\rightarrow$ 5000/100 = 50 blocks
- Read R1 and match
- Read ~ 100 R2 tuples

- To illustrate cost computation, assume:
  - 100 <val,ptr> pairs/block
  - expected number of result tuples is 100
- Build hash table for R2 in memory
  5000 tuples $\rightarrow$ 5000/100 = 50 blocks
- Read R1 and match
- Read ~ 100 R2 tuples

Total cost =	Read R2:	500
	Read R1:	1000
	Get tuples:	100
		1600

Cost of Hash-Join

- In partitioning phase, read+write both relations; 2(M+N). In matching phase, read both relations; M+N I/Os.
- In our running example, this is a total of 4500 I/Os. (**45 seconds**!)
- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory both have a cost of 3(M+N) I/Os. Hash Join superior on this count if relation sizes differ greatly. Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

So far:

contiguous

Iterate	5500	
Merge join		1500
Sort+merge joint		7500
R1.C index		5500 → 550
R2.C index		______
Build R.C index		______
Build S.C index		______
Hash join		4500+
with trick,R1 first		4414
with trick,R2 first		______
Hash join, pointers	1600	

Summary

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

- Sort + merge join good for
  non-equi-join (e.g., R1.C > R2.C)
- If relations already sorted, use
  merge join
- If index exists, it <u>could</u> be useful

  (depends on expected result size)

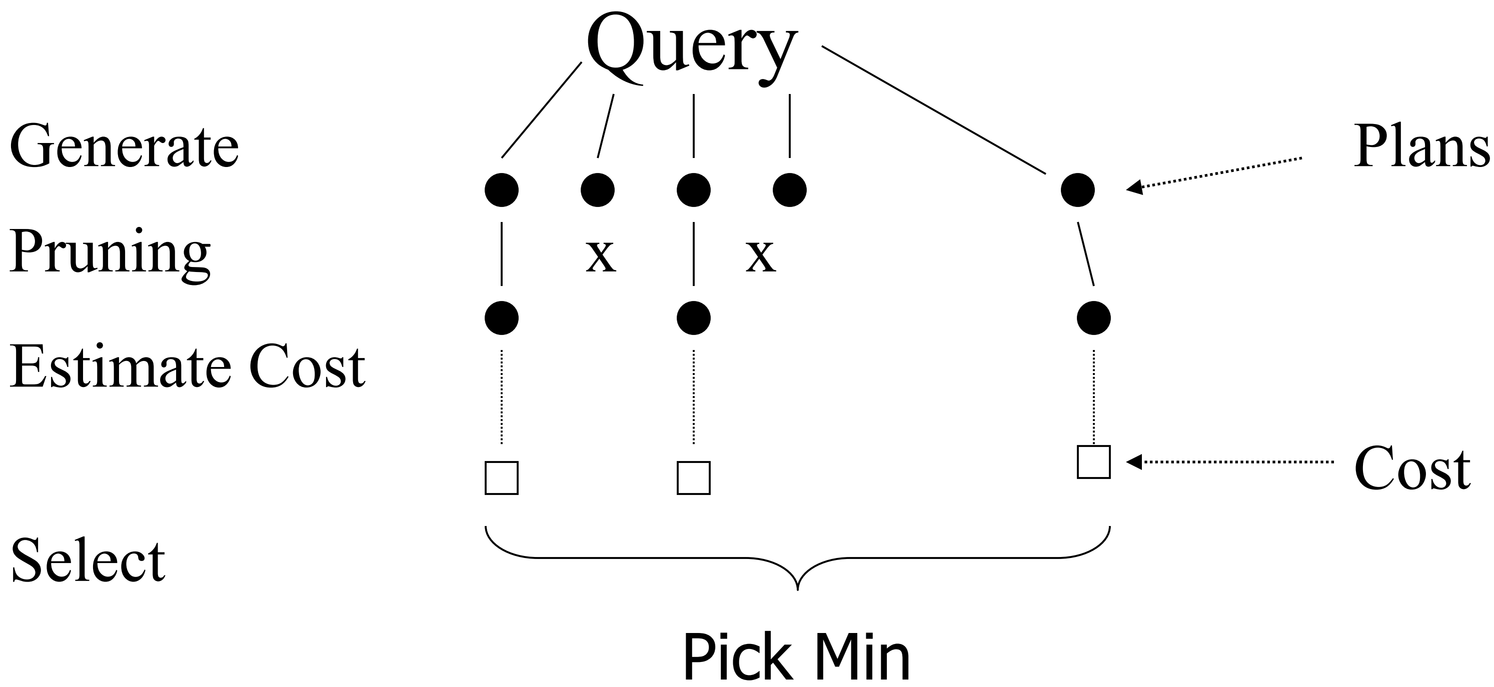
Query Optimization

Discussion

- How would you build a query optimizer?

Query Optimization

--> Generating and comparing plans



Query Optimization Process (simplified a bit)

- Parse the SQL query into a logical tree:
  - identify distinct blocks (corresponding to nested sub-queries or views).
- Query rewrite phase:
  - apply algebraic transformations to yield a cheaper plan.
  - Merge blocks and move predicates between blocks.
- Optimize each block: join ordering.
- Complete the optimization: select scheduling (pipelining strategy).

Building Blocks

- Algebraic transformations (many and wacky).
- Statistical model: estimating costs and sizes.
- Finding the best join trees:
  - Bottom-up (dynamic programming): System-R
- *Newer* architectures:
  - Starburst: rewrite and then tree find
  - Volcano: all at once, top-down.

Key Lessons in Optimization

- There are many approaches and many details to consider in query optimization
  - Classic search/optimization problem!
  - Not completely solved yet!
- Main points to take away are:
  - Algebraic rules and their use in transformations of queries.
  - Deciding on join ordering: System-R style (Selinger style) optimization.
  - Estimating cost of plans and sizes of intermediate results.

Operations (revisited)

- Scan ([index], table, predicate):
  - Either index scan or table scan.
- Selection (filter)
- Projection (always need to go to the data?)
- Joins: nested loop (indexed), sort-merge, hash, outer join.
- Grouping and aggregation (usually the last).

Algebraic Laws

- Commutative and Associative Laws
  - $R \cup S = S \cup R$, $R \cup (S \cup T) = (R \cup S) \cup T$
  - $R \cap S = S \cap R$, $R \cap (S \cap T) = (R \cap S) \cap T$
  - $R \bowtie S = S \bowtie R$, $\mathbf{R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T}$
- Distributive Laws
  - $R \bowtie (S \cup T) = (R \bowtie S) \cup (R \bowtie T)$

Algebraic Laws

- Laws involving selection:
  - $\sigma_{C \text{ AND } C'}(R) = \sigma_C(\sigma_{C'}(R)) = \sigma_C(R) \cap \sigma_{C'}(R)$
  - $\sigma_{C \text{ OR } C'}(R) = \sigma_C(R) \cup \sigma_{C'}(R)$
  - $\sigma_C(R \bowtie S) = \sigma_C(R) \bowtie S$
    - When C involves only attributes of R
  - $\sigma_C(R - S) = \sigma_C(R) - S$
  - $\sigma_C(R \cup S) = \sigma_C(R) \cup \sigma_C(S)$
  - $\sigma_C(R \cap S) = \sigma_C(R) \cap S$

Algebraic Laws

- Example: R(A, B, C, D), S(E, F, G)
  - $\sigma_{F=3}\,(R \underset{D=E}{\bowtie} S) =$?
  - $\sigma_{A=5 \text{ AND } G=9}\,(R \underset{D=E}{\bowtie} S) =$?

Algebraic Laws

- Laws involving projections
  - $\Pi_M(R \bowtie S) = \Pi_N(\Pi_P(R) \bowtie \Pi_Q(S))$
    - Where N, P, Q are appropriate subsets of attributes of M
  - $\Pi_M(\Pi_N(R)) = \Pi_{M,N}(R)$
- Example R(A,B,C,D), S(E, F, G)
  - $\Pi_{A,B,G}(R \bowtie_{D=E} S) = \Pi_{?}(\Pi_{?}(R) \bowtie_{D=E} \Pi_{?}(S))$

Query Rewrites: Sub-queries

```
SELECT Emp.Name
FROM Emp
WHERE Emp.Age < 30
        AND   Emp.Dept# IN
          (SELECT Dept.Dept#
           FROM Dept
           WHERE  Dept.Loc = "Seattle"
            AND       Emp.Emp#=Dept.Mgr)
```

The Un-Nested Query

SELECT Emp.Name

FROM Emp, Dept

WHERE Emp.Age < 30

AND Emp.Dept#=Dept.Dept#

AND Dept.Loc = “Seattle”

AND Emp.Emp#=Dept.Mgr

Converting Nested Queries

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
  AND x.price >= ALL (Select y.price
                      From  product y
                      Where x.maker = y.maker
                        AND y.color="blue")
```

Converting Nested Queries

Let's compute the complement first:

```
Select distinct x.name, x.maker
From product x
Where x.color= "blue"
  AND x.price < SOME (Select y.price
                      From  product y
                      Where x.maker = y.maker
                        AND y.color="blue")
```

Converting Nested Queries

This one becomes a SFW query:

```
Select distinct x.name, x.maker
From product x, product y
Where x.color= “blue” AND x.maker = y.maker
  AND y.color=“blue”  AND x.price < y.price
```

This returns exactly the products we DON’T want, so…

Converting Nested Queries

```
(Select x.name, x.maker
 From product x
 Where x.color = "blue")

EXCEPT

(Select x.name, x.maker
 From product x, product y
 Where x.color= "blue" AND x.maker = y.maker
   AND y.color="blue"  AND x.price < y.price)
```

Semi-Joins, Magic Sets

- You can’t always un-nest sub-queries (it’s tricky).
- But you can often use a semi-join to reduce the computation cost of the inner query.
- A magic set is a superset of the possible bindings in the result of the sub-query.
- Also called “sideways information passing”.
- ***Great idea; reinvented every few years on a regular basis.***

Semijoin

- $R \ltimes S = \Pi_{A1,\ldots,An}(R \bowtie S)$
- Where $A_1, \ldots, A_n$ are the attributes in R
- Example:
  - Employee $\ltimes$ Departments

Rewrites: Magic Sets

```
Create View DepAvgSal AS
  (Select  E.did, Avg(E.sal) as avgsal
   From   Emp E
   Group By E.did)

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
  And  E.age < 30 and D.budget > 100k
  And  E.sal > V.avgsal
```

Rewrites: SIPs

Select E.eid, E.sal
From Emp E, Dept D, DepAvgSal V
Where E.did=D.did AND D.did=V.did
  And E.age < 30 and D.budget > 100k
  And E.sal > V.avgsal

- DepAvgsal needs to be evaluated only for departments where V.did IN

  Select E.did
  From Emp E, Dept D
  Where E.did=D.did
    And E.age < 30 and D.budget > 100K

Supporting Views

1. Create View PartialResult as
   (Select E.eid, E.sal, E.did
   From Emp E, Dept D
   Where E.did=D.did
   And E.age < 30 and D.budget > 100K)
2. Create View Filter AS
   Select DISTINCT P.did FROM PartialResult P.
2. Create View LimitedAvgSal as
   (Select F.did Avg(E.Sal) as avgSal
   From Emp E, Filter F
   Where E.did=F.did
   Group By F.did)

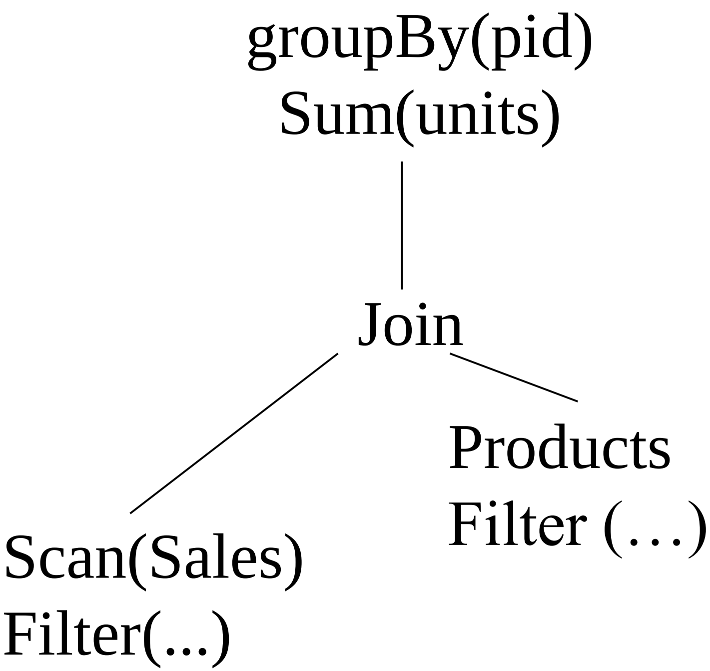
And Finally…

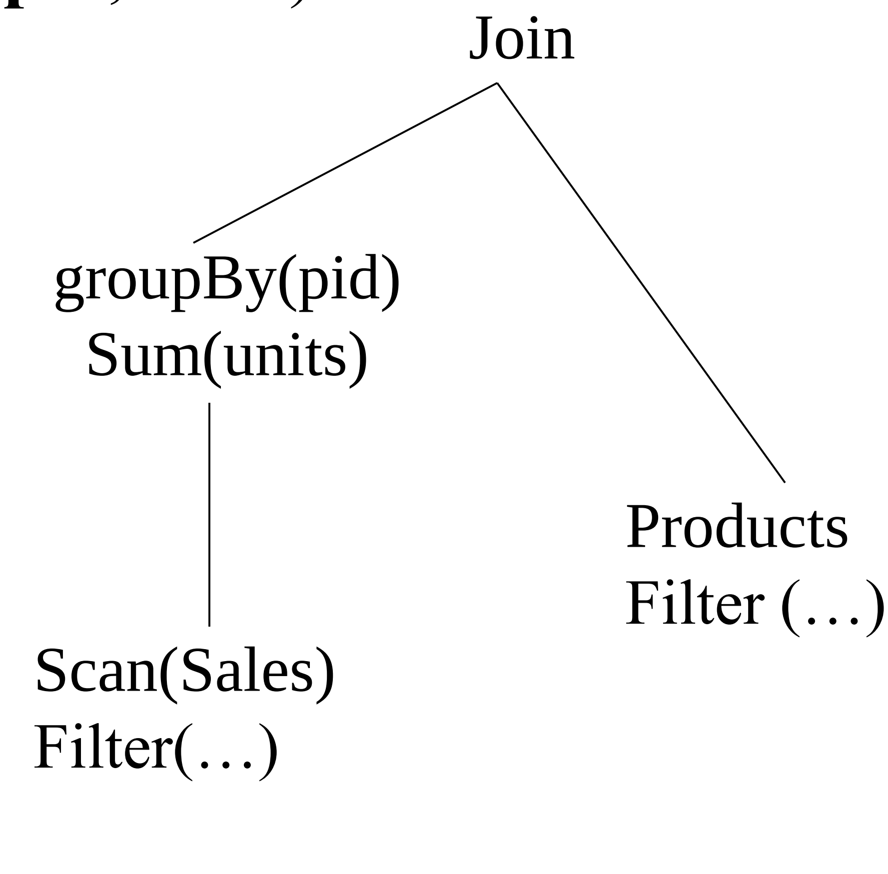
Transformed query:

```
Select  P.eid, P.sal
From  PartialResult P,  LimitedAvgSal V
Where P.did=V.did
    And  P.sal > V.avgsal
```

Rewrites: Group By and Join

- Schema:
  - Product (**pid**, unitprice,…)
  - Sales(tid, date, store, **pid**, units)
- Trees:





Rewrites:Operation Introduction

- Schema: (pid determines cid)
  - Category (**pid**, cid, details)
  - Sales(tid, date, store, **pid**, amount)
- Trees:

groupBy(cid)
Sum(amount)

Join

Scan(Sales)
Filter(store IN {CA,WA})

Category
Filter (…)

groupBy(cid)
Sum(amount)

Join

groupBy(pid)
Sum(amount)

Scan(Sales)
Filter(store IN {CA,WA})

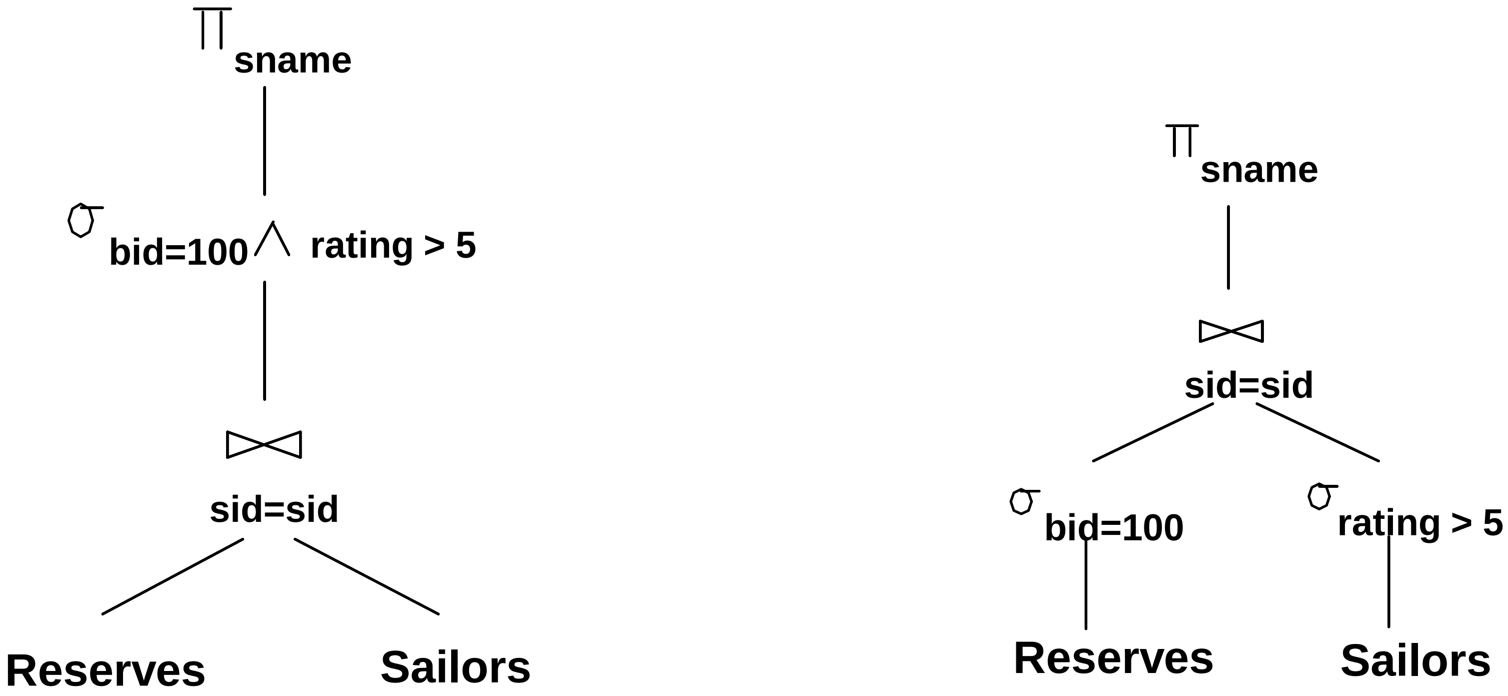
Category
Filter (…)

Schema for Some Examples

Sailors (*sid*: integer, *sname*: string, *rating*: integer, *age*: real)
Reserves (*sid*: integer, *bid*: integer, *day*: dates, *rname*: string)

- Reserves:
  - Each tuple is 40 bytes long, 100 tuples per page, 1000 pages
- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.

Query Rewriting: Predicate Pushdown



The earlier we process selections, less tuples we need to manipulate higher up in the tree.

Query Rewrites: Predicate Pushdown (through grouping)

```
Select   bid, Max(age)
From     Reserves R, Sailors S
Where    R.sid=S.sid
GroupBy  bid
Having   Max(age) > 40
```

```
Select   bid, Max(age)
From     Reserves R, Sailors S
Where    R.sid=S.sid  and
         S.age > 40
GroupBy  bid
```

- *For each boat, find the maximal age of sailors who've reserved it.*
- Advantage: the size of the join will be smaller.
- Requires transformation rules specific to the grouping/aggregation operators.
- **Will it work if we replace Max by Min?**

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age, age < 20
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid
```

Query Rewrite: Predicate Movearound

Sailing wiz dates: when did the youngest of each sailor level rent boats?

First, move predicates up the tree.

Then, move them down.

```
Select   sid, date
From     V1, V2
Where    V1.rating = V2.rating  and
         V1.age = V2.age, and age < 20
```

```
Create View V1 AS
Select   rating, Min(age)
From     Sailors S
Where    S.age < 20
Group By  rating
```

```
Create View V2 AS
Select   sid, rating, age, date
From     Sailors S, Reserves R
Where    R.sid=S.sid, and
         S.age < 20.
```

Query Rewrite Summary

- The optimizer can use any *semantically correct* rule to transform one query to another.
- Rules try to:
  - move constraints between blocks (because each will be optimized separately)
  - Unnest blocks
- Especially important in decision support applications where queries are very complex.
- In a few minutes of thought, you'll come up with your own rewrite. Some query, somewhere, will benefit from it.
- Theorems?

Cost Estimation

- For each plan considered, must estimate cost:
  - Must estimate *cost* of each operation in plan tree.
    - Depends on input cardinalities.
  - Must estimate *size of result* for each operation in tree!
    - Use information about the input relations.
    - For selections and joins, assume independence of predicates.
- We'll discuss the System R cost estimation approach.
  - Very inexact, but works ok in practice.
  - More sophisticated techniques known now.

Statistics and Catalogs

- Need information about the relations and indexes involved. ***Catalogs*** typically contain at least:
  - # tuples (NTuples) and # pages (NPages) for each relation.
  - # distinct key values (NKeys) and NPages for each index.
  - Index height, low/high key values (Low/High) for each tree index.
- Catalogs updated periodically.
  - Updating whenever data changes is too expensive; lots of approximation anyway, so slight inconsistency ok.
- More detailed information (e.g., histograms of the values in some field) are sometimes stored.

Size Estimation and Reduction Factors

```
SELECT  attribute list
FROM  relation list
WHERE  term1 AND ... AND termk
```

- Consider a query block:
- Maximum # tuples in result is the product of the cardinalities of relations in the FROM clause.
- *Reduction factor (RF)* associated with each *term* reflects the impact of the *term* in reducing result size. *Result cardinality* = Max # tuples * product of all RF's.
  - Implicit assumption that *terms* are independent!
  - Term *col=value* has RF *1/NKeys(I),* given index I on *col*
  - Term *col1=col2* has RF *1/MAX(NKeys(I1), NKeys(I2))*
  - Term *col>value* has RF *(High(I)-value)/(High(I)-Low(I))*

Histograms

- Key to obtaining good cost and size estimates.
- Come in several flavors:
  - Equi-depth
  - Equi-width
- Which is better?
- Compressed histograms: special treatment of frequent values.

Histograms

- Statistics on data maintained by the RDBMS
- Makes size estimation much more accurate (hence, cost estimations are more accurate)

Histograms

Employee(ssn, name, salary, phone)

- Maintain a histogram on salary:

Salary:	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
Tuples	200	800	5000	12000	6500	500

- T(Employee) = 25000, but now we know the distribution

Histograms

Ranks(rankName, salary)

- Estimate the size of Employee $\bowtie_{\text{Salary}}$ Ranks

Employee	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	200	800	5000	12000	6500	500

Ranks	0..20k	20k..40k	40k..60k	60k..80k	80k..100k	> 100k
	8	20	40	80	100	2

Histograms

- Assume:
  - V(Employee, Salary) = 200
  - V(Ranks, Salary) = 250
- Then T(Employee $\bowtie_{Salary}$ Ranks) =

  $= \Sigma_{i=1,6}\ T_i\ T_i' / 250$

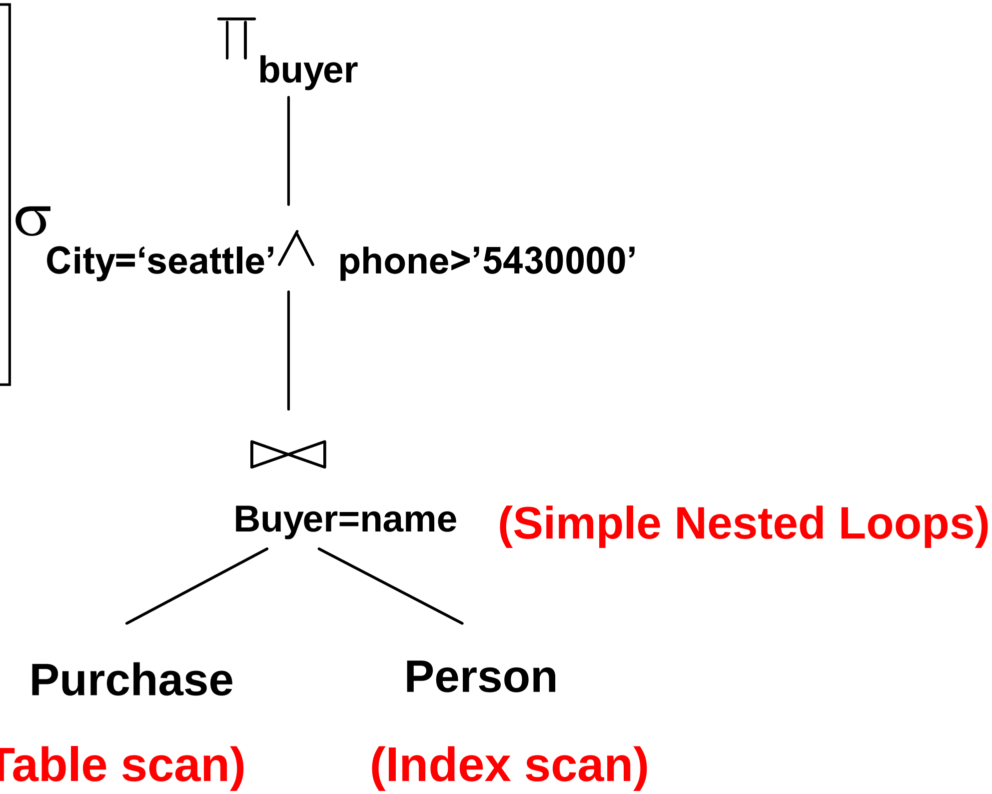
  $= (200x8 + 800x20 + 5000x40 +$

  $12000x80 + 6500x100 + 500x2)/250$

  = ….

Query Execution Plans

```
SELECT  buyer
FROM Purchase P, Person Q
WHERE P.buyer=Q.name AND
      Q.city='seattle' 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.

We've Seen So Far

- Transformation rules
- The cost module:
  - Given a candidate plan: what is its expected cost and size of the result?

- Now: putting it all together.

Plans for Single-Relation Queries (Prep for Join ordering)

- **Task:** create a query execution plan for a single Select-project-group-by block.
- **Key idea**: consider each possible ***access path*** to the relevant tuples of the relation. Choose the cheapest one.
- The different operations are essentially carried out together (e.g., if an index is used for a selection, projection is done for each retrieved tuple, and the resulting tuples are *pipelined* into the aggregate computation).

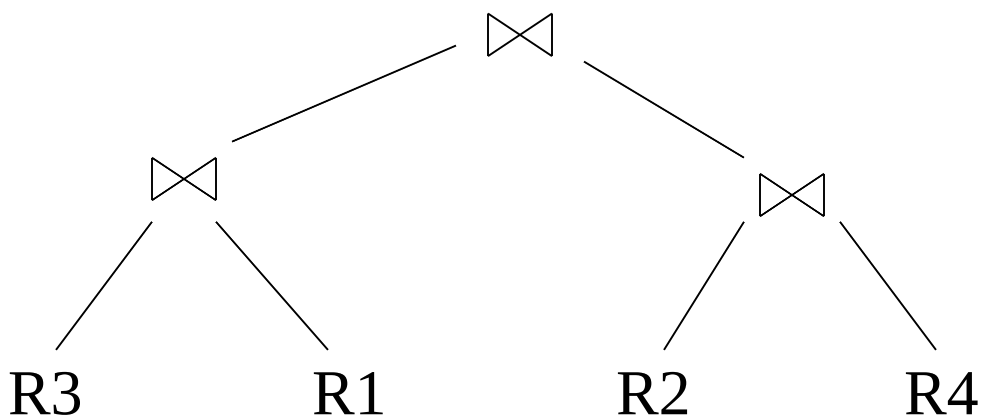
Example

```
SELECT  S.sid
FROM  Sailors S
WHERE  S.rating=8
```

- If we have an Index on *rating*:
  - (1/NKeys(I)) * NTuples(S) = (1/10) * 40000 tuples retrieved.
  - Clustered index: (1/NKeys(I)) * (NPages(I)+NPages(S)) = (1/10) * (50+500) pages are retrieved (= 55).
  - Unclustered index: (1/NKeys(I)) * (NPages(I)+NTuples(S)) = (1/10) * (50+40000) pages are retrieved.
- If we have an index on *sid*:
  - Would have to retrieve all tuples/pages. With a clustered index, the cost is 50+500.
- Doing a file scan: we retrieve all file pages (500).

Determining Join Ordering

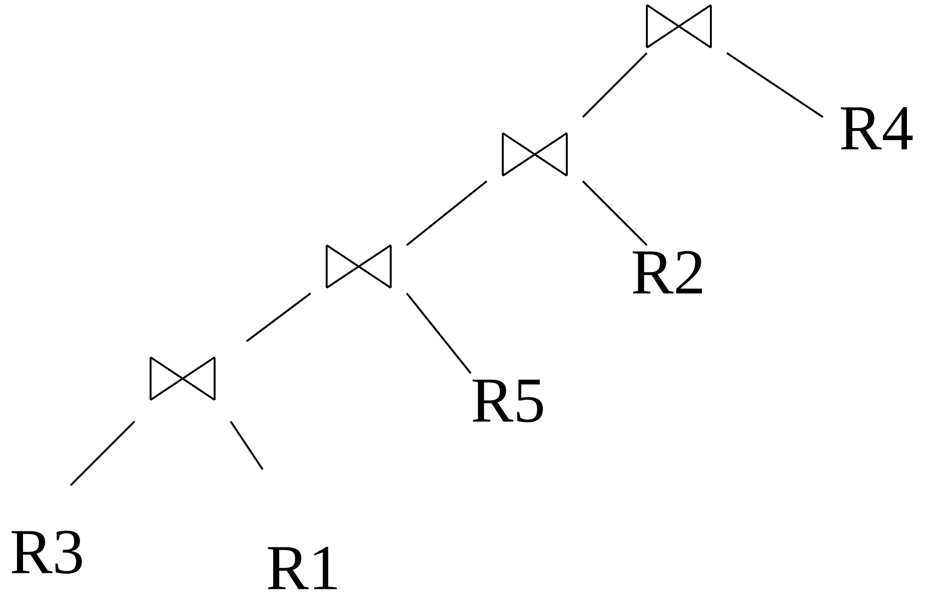
- R1 $\bowtie$ R2 $\bowtie$ …. $\bowtie$ Rn
- Join tree:



- A join tree represents a plan. An optimizer needs to inspect many (all ?) join trees

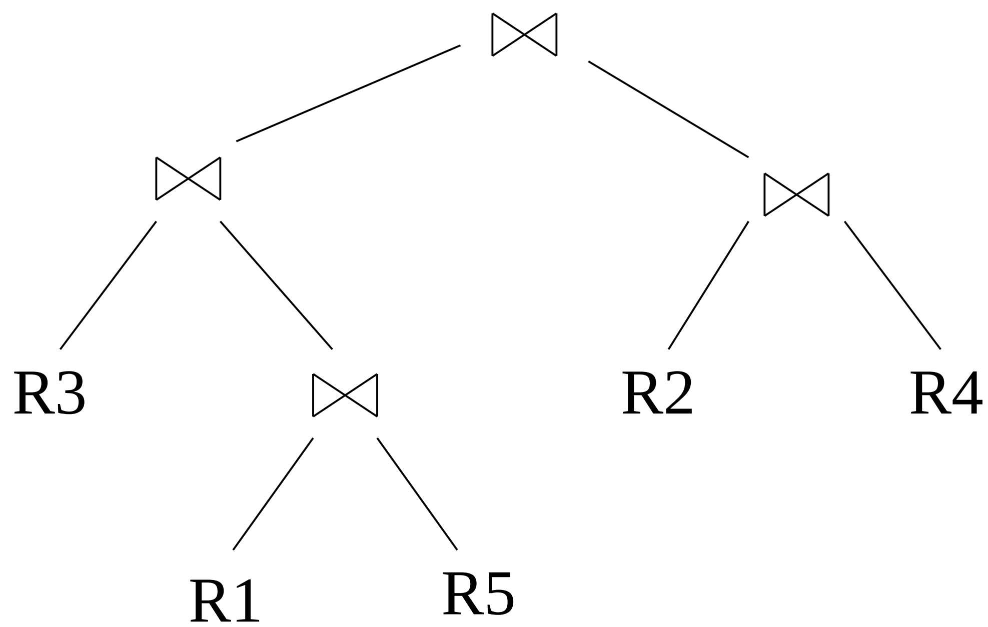
Types of Join Trees

- Left deep:



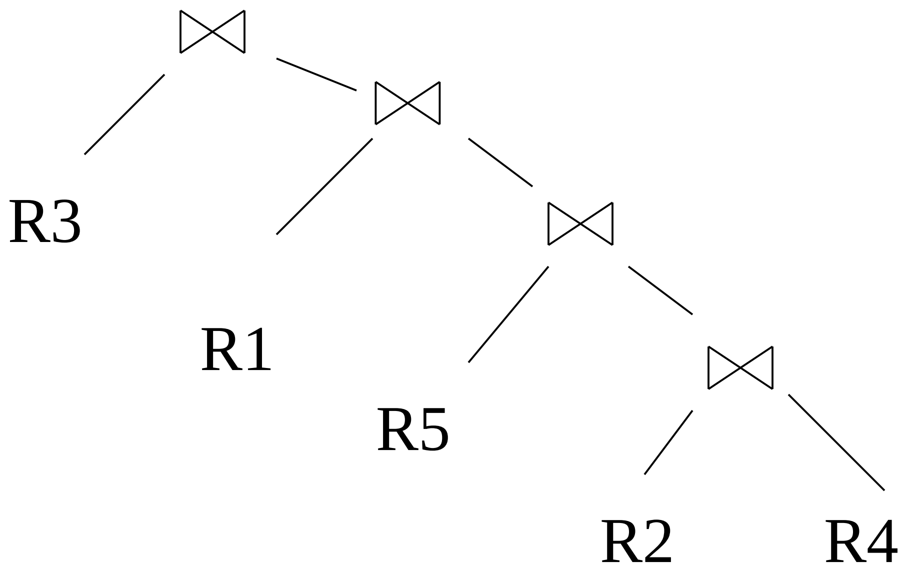
Types of Join Trees

- Bushy:



Types of Join Trees

- Right deep:



Problem

- Given: a query $R1 \bowtie R2 \bowtie \ldots \bowtie Rn$
- Assume we have a function cost() that gives us the cost of every join tree
- Find the best join tree for the query

Dynamic Programming

- Idea: for each subset of {R1, …, Rn}, compute the best plan for that subset
- In increasing order of set cardinality:
  - Step 1: for {R1}, {R2}, …, {Rn}
  - Step 2: for {R1,R2}, {R1,R3}, …, {Rn-1, Rn}
  - …
  - Step n: for {R1, …, Rn}
- A subset of {R1, …, Rn} is also called a *subquery*

Dynamic Programming

- For each subquery $Q \subseteq \{R1, \ldots, Rn\}$ compute the following:
  - Size(Q)
  - A best plan for Q: Plan(Q)
  - The cost of that plan: Cost(Q)

Dynamic Programming

- **Step 1**: For each {Ri} do:
  - Size({Ri}) = B(Ri)
  - Plan({Ri}) = Ri
  - Cost({Ri}) = (cost of scanning Ri)

Dynamic Programming

- **Step i**: For each Q ⊆ {R1, …, Rn} of cardinality i do:
  - Compute Size(Q) (later…)
  - For every pair of subqueries Q', Q''
    s.t. Q = Q' $\bowtie$ Q''
    compute cost(Plan(Q') $\bowtie$ Plan(Q''))
  - Cost(Q) = the smallest such cost
  - Plan(Q) = the corresponding plan

Dynamic Programming

- Return Plan({R1, …, Rn})

Dynamic Programming

- Summary: computes optimal plans for subqueries:
  - Step 1: {R1}, {R2}, …, {Rn}
  - Step 2: {R1, R2}, {R1, R3}, …, {Rn-1, Rn}
  - …
  - Step n: {R1, …, Rn}
- We used naïve size/cost estimations
- In practice:
  - more realistic size/cost estimations (next)
  - heuristics for Reducing the Search Space
    - Restrict to left linear trees
    - Restrict to trees "without cartesian product"
  - need more than just one plan for each subquery:
    - "interesting orders"

Query Optimization Summary

- Create initial (naïve) query execution plan.
- Apply transformation rules:
  - Try to un-nest blocks
  - Move predicates and grouping operators.
- Consider each block at a time:
  - Determine join order
  - Push selections, projections if possible.