Implementation of Relational Operations

courtesy of Joe Hellerstein, Mike Franklin, and etc for some slides.

Jianlin Feng
School of Software
SUN YAT-SEN UNIVERSITY

Introduction

- Next topic: QUERY PROCESSING
- Some database operations are EXPENSIVE
- Huge performance gained by being “smart”
  - We’ll see 1,000,000x over naïve approach
- Main weapons are:
  - clever implementation techniques for operators
  - exploiting relational algebra “equivalences”
  - using statistics and cost models to choose

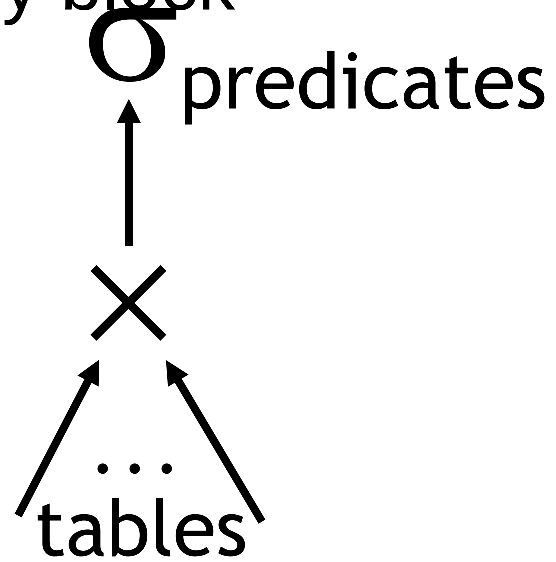
Simple SQL Refresher

```
SELECT <list-of-fields>
  FROM <list-of-tables>
 WHERE <condition>
```

```
SELECT S.name, E.cid
  FROM Students S, Enrolled E
 WHERE S.sid=E.sid AND E.grade='A'
```

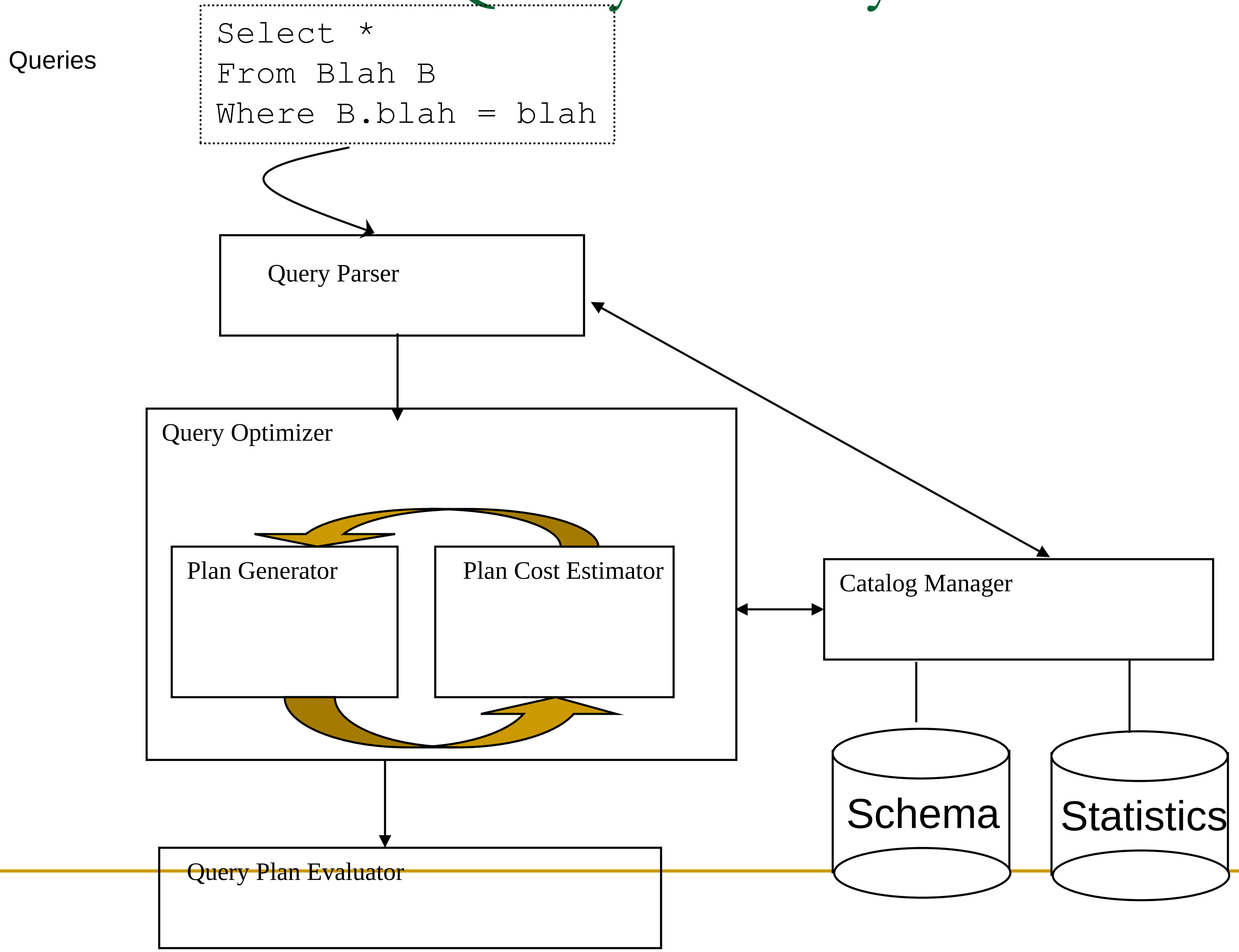
A Really Bad Query Optimizer

- For each Select-From-Where query block
  - Create a plan that:
    - Forms the cross product of the FROM clause
    - Applies the WHERE clause



- (Then, as needed:
  - Apply the GROUP BY clause
  - Apply the HAVING clause
  - Apply any projections and output expressions
  - Apply duplicate elimination and/or ORDER BY)

Cost-based Query Sub-System



The Query Optimization Game

- Goal is to pick a “good” plan
  - Good = low expected cost, under *cost model*
  - Degrees of freedom:
    - access methods
    - physical operators
    - operator orders
- Roadmap for this topic:
  - *First:* implementing individual operators
  - *Then:* optimizing multiple operators

Relational Operations

- We will consider how to implement:
  - *Selection* (σ) Select a subset of rows.
  - *Projection* (π) Remove unwanted columns.
  - *Join* ($\bowtie$) Combine two relations.
  - *Set-difference* ($-$) Tuples in reln. 1, but not in reln. 2.
  - *Union* ($\cup$) Tuples in reln. 1 and in reln. 2.

- Q: What about Intersection?

Schema for Examples

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

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - [S]=500, p_S=80.
- Reserves:
  - Each tuple is 40 bytes, 100 tuples per page, 1000 pages.
  - [R]=1000, p_R=100.

Simple Selections

```
SELECT  *
  FROM   Reserves R
WHERE   R.rname < 'C%'
```

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

- How best to perform? Depends on:
  - what indexes are available
  - expected size of result
- Size of result approximated as

  *(size of R) * selectivity*

  - *selectivity* estimated via statistics – we will discuss shortly.

Our options …

- If no appropriate index exists:

  Must scan the whole relation

  cost = [R]. For “reserves” = 1000 I/Os.

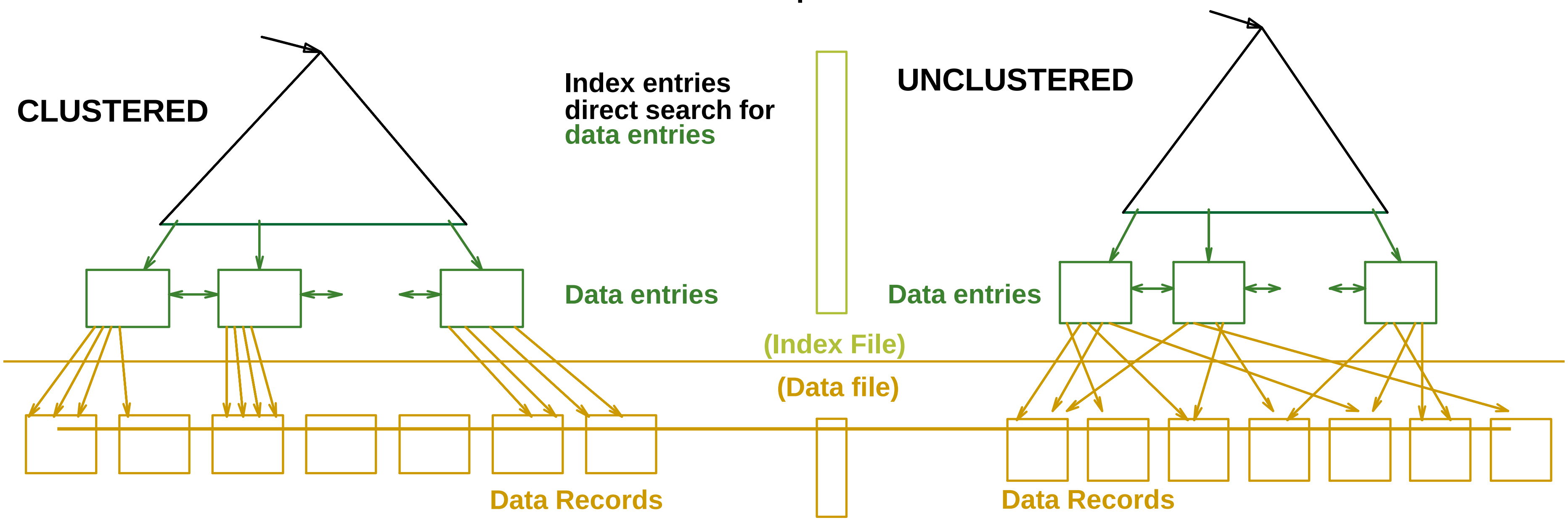
Our options ...

- With index on selection attribute:

  1. Use index to find qualifying data entries
  2. Retrieve corresponding data records

  Total cost = cost of step 1 + cost of step 2

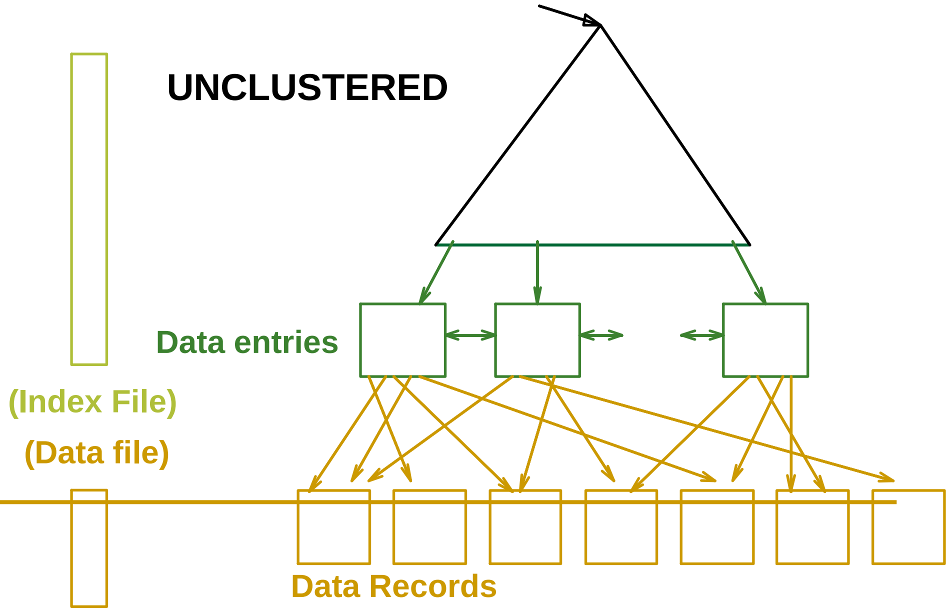
  - For "reserves", if selectivity = 10% (100 pages, 10000 tuples):
    - If *clustered* index, cost is a little over 100 I/Os;
    - If *unclustered*, could be up to 10000 I/Os! *… unless …*



Refinement for unclustered indexes

1. Find qualifying data entries.
2. Sort the rids of the data records to be retrieved.
3. Fetch rids in order.

Each data page is looked at just once (though # of such pages likely to be higher than with clustering).



General Selection Conditions

(day<8/9/94 AND rname='Paul') OR bid=5 OR sid=3

- First, convert to <u>*conjunctive normal form* (CNF)</u>:
  - *(day<8/9/94* OR *bid=5* OR *sid=3) AND*
    (rname='Paul' OR *bid=5* OR *sid=3)*
- We only discuss the case with no <u>OR</u>s
- Terminology:
  - A B-tree index <u>*matches*</u> terms that involve only attributes in a *prefix* of the search key. *e.g.*:
  - Index on <*a, b, c*> matches *a*=5 AND *b*= 3, but not *b*=3.

2 Approaches to General Selections

Approach I:

1. Find the *cheapest access path*
2. retrieve tuples using it
3. Apply any remaining terms that don't match the index

- *Cheapest access path:* An index or file scan that we estimate will require the fewest page I/Os.

Cheapest Access Path - Example

query: *day < 8/9/94 AND bid=5 AND sid=3*

some options:

B+tree index on <u>day</u>; check bid=5 and sid=3 afterward.

hash index on <bid, sid>; check day<8/9/94 afterward.

- *How about a B+tree on <rname, day>?*
- *How about a B+tree on <day, rname>?*
- *How about a Hash index on <day, rname>?*

2 Approaches to General Selections (Contd.)

Approach II: use 2 or more matching indexes.

1. From each index, get set of rids
2. Compute intersection of rid sets
3. Retrieve records for rids in intersection
4. Apply any remaining terms

EXAMPLE: *day<8/9/94 AND bid=5 AND sid=3*

Suppose we have an index on *day*, and another index on *sid*.

- Get rids of records satisfying *day<8/9/94*.
- Also get rids of records satisfying *sid=3*.
- Find intersection, then retrieve records, then check *bid=5*.

Projection

```
SELECT   DISTINCT
           R.sid, R.bid
FROM     Reserves R
```

- Issue is removing duplicates.

pass 1: $\left\lceil \frac{250}{20} \right\rceil = 13$ 12个 20 1个 10

- Use sorting!!

pass 2: 剩下 7个 buffer 可以满足 13路 归并排序

1. Scan R, extract only the needed attributes
2. Sort the resulting set 外排序
3. Remove adjacent duplicates

消重

Cost:

Ramakrishnan/Gehrke writes to temp table at each step!

Reserves with size ratio 0.25 = 250 pages.

With 20 buffer pages can sort in 2 passes, so:

1000 +250 + 2 * 2 * 250 + 250 = 2500 I/Os

扫描 取出 2趟外排序 消重

Projection -- improved

- Avoid the temp files, work on the fly:
  - Modify Pass 0 of sort to eliminate unwanted fields.
  - Modify Passes 1+ to eliminate duplicates.

Cost:

Reserves with size ratio 0.25 = 250 pages.

With 20 buffer pages can sort in 2 passes, so:

1. Read 1000 pages
2. Write 250 (in runs of 40 pages each) = 7 runs
3. Read and merge runs (20 buffers, so 1 merge pass!)

Total cost = 1000 + 250 +250 = 1500.

Other Projection Tricks

If an index search key contains all wanted attributes:

- Do *index-only* scan
  - Apply projection techniques to data entries *(much smaller!)*

If a B+Tree index search key *prefix* has all wanted attributes:

- Do *in-order* index-only scan
  - Just retrieve the data entries in order;
  - Discarding unwanted fields;
  - Compare adjacent tuples on the fly *to check for duplicates.*

Joins

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

- Joins are <u>very</u> common.
- $R \bowtie S$ is large; so, $R \bowtie S$ followed by a selection is inefficient.
- Many approaches to reduce join cost.

- Join techniques we will cover today:
  1. Nested-loops join
  2. Index-nested loops join
  3. Sort-merge join

Simple Nested Loops Join

R ⋈ S: foreach tuple r in R do

直接 读元组

foreach tuple s in S do

if $r_i == s_j$ then add <r, s> to result

Cost = $(p_R*[R])*[S] + [R]$ = 100*1000*500 + 1000 IOs

比较 S读入 R读入

- At 10ms/IO, Total time: ???

只需要 3个缓冲区

- What if smaller relation (S) was "outer"?
- What assumptions are being made here?
- What is cost if one relation can fit entirely in memory?

Page-Oriented Nested Loops Join

R ⋈ S:

读页面

foreach page b_R in R do
  foreach page b_S in S do
    foreach tuple r in b_R do
      foreach tuple s in b_S do
        if $r_i == s_j$ then add <r, s> to result

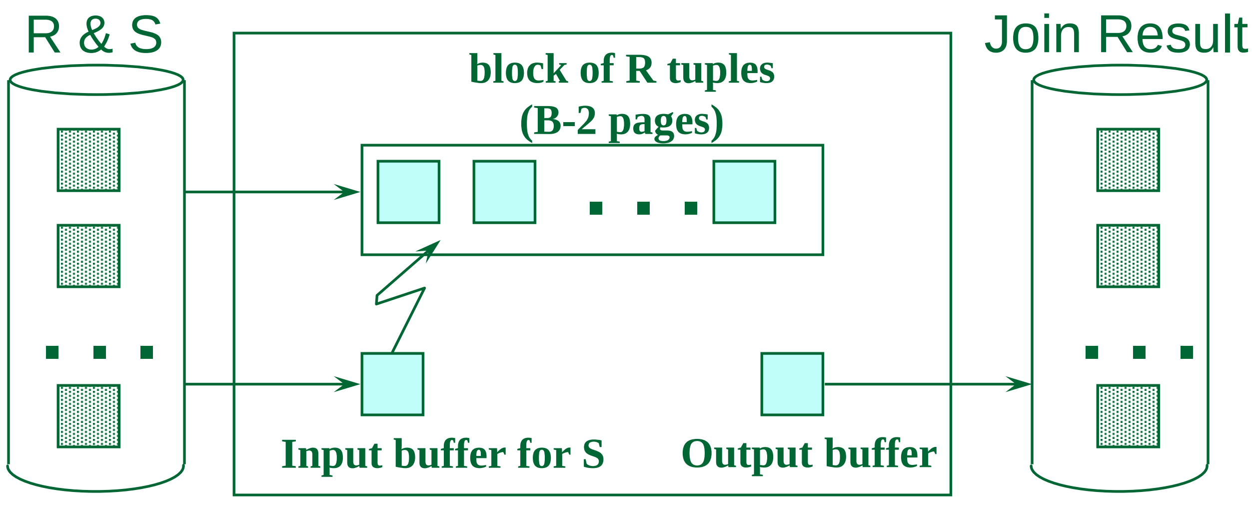
Cost = [R]*[S] + [R] = 1000*500 + 1000

outer × inner + outer

- If smaller relation (S) is outer, cost = 500*1000 + 500
- Much better than naïve per-tuple approach!

Block Nested Loops Join

- Page-oriented NL doesn't exploit extra buffers :(
- Idea to use memory efficiently:



Cost: Scan outer + (#outer blocks * scan inner)

#outer blocks = $\lceil \#of\ pagesof\ outer / blocksize \rceil$

Examples of Block Nested Loops Join

块嵌套　　2个buffer进行输入输出

- Say we have B = 100+2 memory buffers
- Join cost = [outer] + (#outer blocks * [inner])

  #outer blocks = [outer] / 100

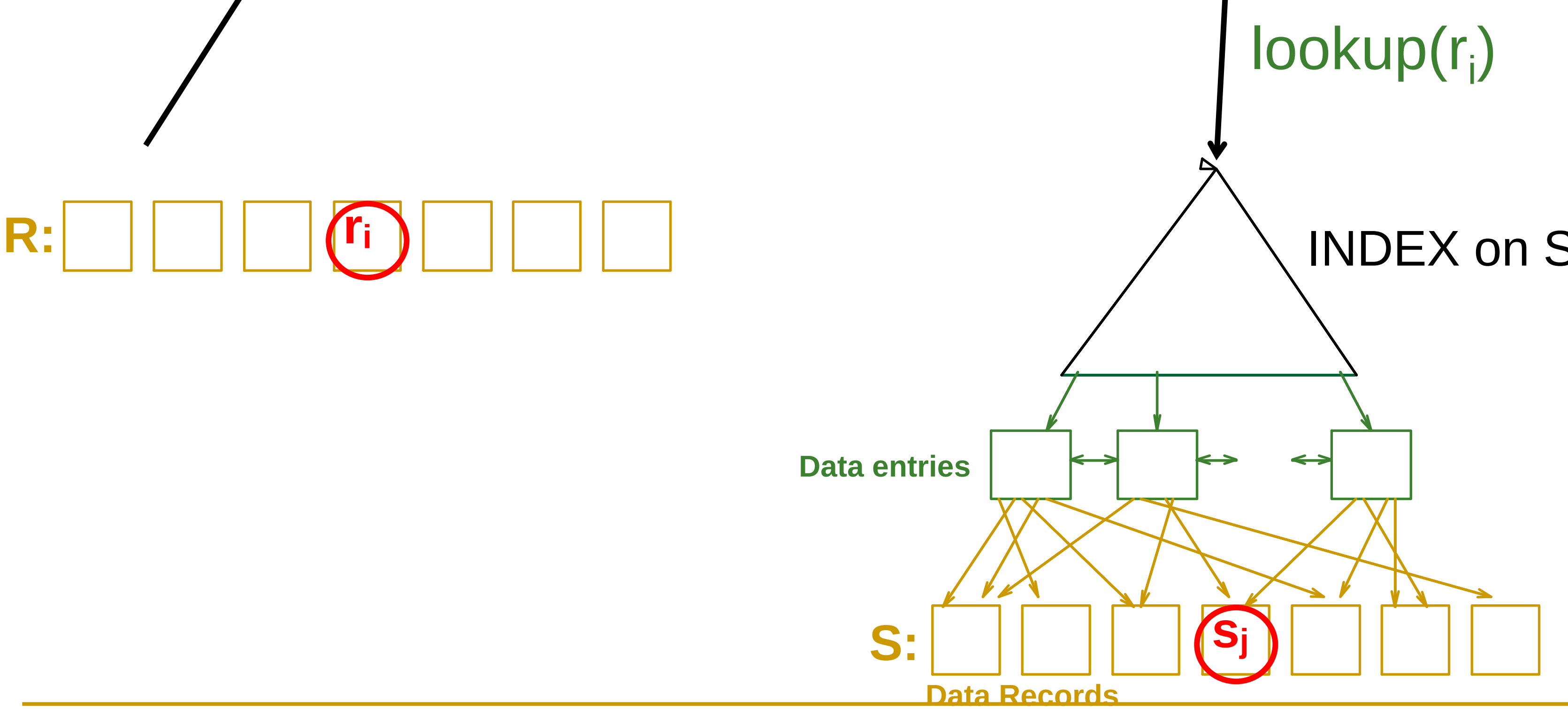
- With R as outer ([R] = 1000):
  - Scanning R costs 1000 IO's *(done in 10 blocks)*
  - Per block of R, we scan S; costs 10*500 I/Os
  - Total = 1000 + 10*500.
- With S as outer ([S] = 500):
  - Scanning S costs 500 IO's *(done in 5 blocks)*
  - Per block of S, we can R; costs 5*1000 IO's
  - Total = 500 + 5*1000.

Index Nested Loops Join 索引嵌套

$R \bowtie S$: foreach tuple r in R do

foreach tuple s in S where $r_i == s_j$ do

add <r, s> to result



Index Nested Loops Join

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

Cost = [R] + ([R]*p_R) * cost to find matching S tuples

- If index uses Alt. 1, cost = cost to traverse tree from root to leaf.
- For Alt. 2 or 3:
  1. Cost to lookup RID(s); typically 2-4 IO's for B+Tree.
  2. Cost to retrieve records from RID(s); depends on clustering.
     - Clustered index: 1 I/O per page of matching S tuples.
     - Unclustered: up to 1 I/O per matching S tuple.

Reminder: Schema for Examples

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

- Sailors:
  - Each tuple is 50 bytes long, 80 tuples per page, 500 pages.
  - [S]=500, p_S=80.
- Reserves:
  - Each tuple is 40 bytes, 100 tuples per page, 1000 pages.
  - [R]=1000, p_R=100.

Sort-Merge Join

外排序归并

1. Sort R on join attr(s)
2. Sort S on join attr(s)
3. Scan sorted-R and sorted-S in tandem, to find matches

Example:

```
SELECT  *
FROM    Reserves R1, Sailors S1
WHERE   R1.sid=S1.sid
```

sid	sname	rating	age
22	dustin	7	45.0
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

sid	bid	day	rname
28	103	12/4/96	guppy
28	103	11/3/96	yuppy
31	101	10/10/96	dustin
31	102	10/12/96	lubber
31	101	10/11/96	lubber
58	103	11/12/96	dustin

Cost of Sort-Merge Join

- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be [R]*[S] *(very unlikely!)*
  - Q: what is worst case?

Suppose B = 35 buffer pages:

- Both R and S can be sorted in 2 passes
- Total join cost = 4*1000 + 4*500 + (1000 + 500) = 7500

$\left\lceil \frac{1000}{35} \right\rceil < 35$ $\left\lceil \frac{500}{35} \right\rceil < 35$

都只需要 2 趟外排序

Suppose B = 300 buffer pages:

- Again, both R and S sorted in 2 passes
- Total join cost = 7500

Block-Nested-Loop cost = 2500 … 15,000

Refinement of Sort-Merge Join

- We can combine the merging phases in the *sorting* of R and S with the merging required for the join.
  - If B > $\sqrt{L}$, where *L* is the size of the larger relation,
  - using the sorting refinement (13.3.1) that produces runs of length 2B in Pass 0, #runs of each relation is < B/2.
  - In "Merge" phase: Allocate 1 page per run of each relation, and `merge' while checking the join condition
  - Cost: read+write each relation in Pass 0 + read each relation in (only) merging pass (+ writing of result tuples).
  - In example, cost goes down from 7500 to 4500 I/Os.

Hash-Join

哈希

- Partition both relations on the join attributes using hash function h.
- R tuples in partition R_i will only match S tuples in partition S_i.

For i= 1 to #partitions {

Read in partition R_i and hash it using h2 (not h).

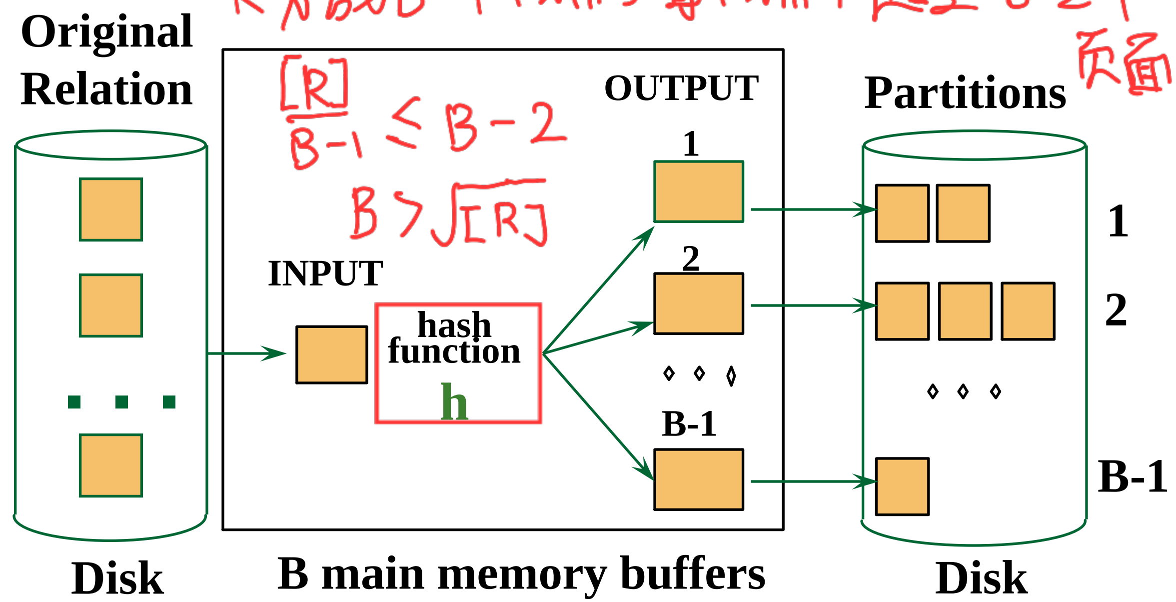
Scan partition S_i and probe hash table for matches.

}

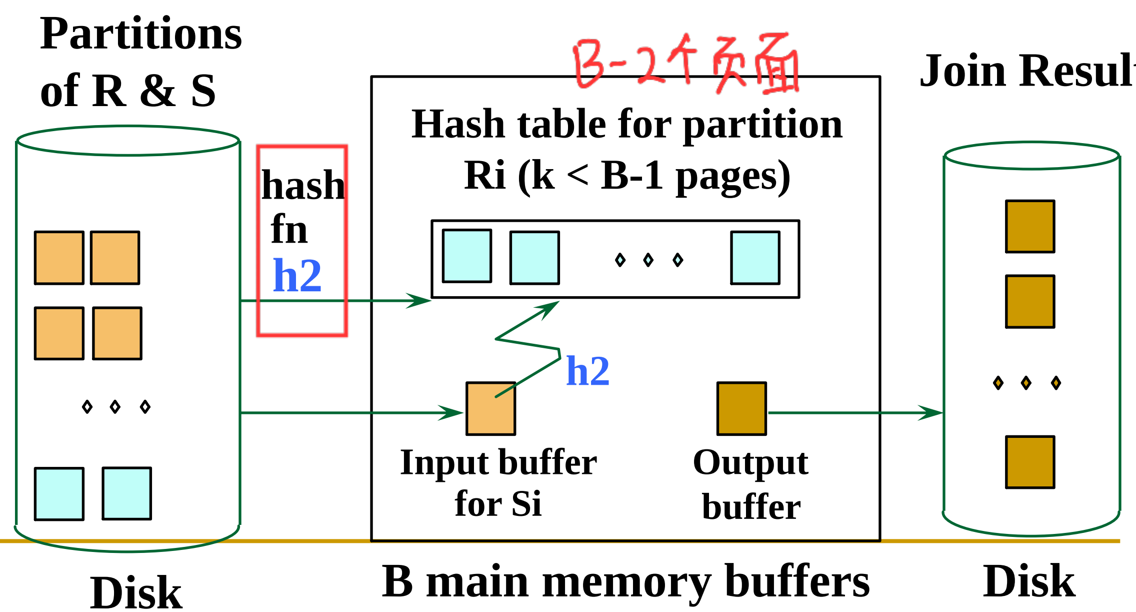
R分成B-1个桶，每个桶不超过B-2个页面

$\frac{[R]}{B-1} \leq B-2$

$B > \sqrt{[R]}$



B-2个页面



Memory Requirements of Hash-Join

- #partitions k < B, and B-1 > size of largest partition to be held in memory. Assuming uniformly sized partitions, and maximizing k, we get:

  k= B-1, and M/(B-1) < B-2, i.e., B must be > $\sqrt{M}$

  从用小的表

- Since we build an in-memory hash table to speed up the matching of tuples in the second phase, a little more memory is needed.

- If the hash function does not partition uniformly, one or more R partitions may not fit in memory.
  - Can apply hash-join technique recursively to do the join of this R-partition with corresponding S-partition.

Cost of Hash-Join

$3 \times (M+N) = 3 \times (1000 + 500)$

- In partitioning phase, read+write both relns; 2(M+N). In matching phase, read both relns; M+N I/Os.

$\sqrt{500} = 23$ $\left\lceil \frac{1000}{23} \right\rceil = 44$ $\frac{44}{22} = 2$

- In our running example, this is a total of 4500 I/Os.

$2 \times 3 \times 1000 + 2 \times 2 \times 500 + 1000 + 500 = 9500 > 4500$

- Sort-Merge Join vs. Hash Join:
  - Given a minimum amount of memory (*what is this, for each?*) both have a cost of 3(M+N) I/Os. Hash Join superior if relation sizes differ greatly (e.g., if one reln fits in memory). Also, Hash Join shown to be highly parallelizable.
  - Sort-Merge less sensitive to data skew; result is sorted.

Set Operations

- Intersection and cross-product as special cases of join.
- Union (Distinct) and Except similar; we'll do union.

- Sorting based approach to union:
  - Sort both relations (on combination of all attributes).
  - Scan sorted relations and merge them.
  - *Alternative*: Merge runs from Pass 0 for *both* relations.

- Hash based approach to union: 用主键进行hash
  - Partition R and S using hash function *h*.
  - For each S-partition, build in-memory hash table (using *h2*), scan corresponding R-partition and add tuples to table while discarding duplicates.

General Join Conditions

- Equalities over several attributes (e.g., *R.sid=S.sid* AND *R.rname=S.sname*):
  - For Index NL, build index on *<sid, sname>* (if S is inner); or use existing indexes on *sid* or *sname*.
  - For Sort-Merge and Hash Join, sort/partition on combination of the two join columns.

- Inequality conditions (e.g., *R.rname < S.sname*):
  - For Index NL, need (clustered!) B+ tree index.
    - Range probes on inner; # matches likely to be much higher than for equality joins.
  - Hash Join, Sort Merge Join not applicable!
  - Block NL quite likely to be the best join method here.

Aggregate Operations (AVG, MIN, etc.)

Example:
SELECT AVG(S.age)
FROM Sailors S

- Without grouping:
  - In general, requires scanning the relation.
  - Given a tree index whose search key includes all attributes in the SELECT or WHERE clauses, can do index-only scan.

Aggregate Operations (continued)

- With grouping:
  - Sort on group-by attributes, then scan relation and compute aggregate for each group. (Better: combine sorting and aggregate computation.)
  - Similar approach based on hashing on group-by attributes.
  - Given a tree index whose search key includes all attributes in SELECT, WHERE and GROUP BY clauses, can do index-only scan;
    - if group-by attributes form prefix of search key, can retrieve data entries/tuples in group-by order.

Summary

- *Queries are composed of a few basic operators*;
  - The implementation of these operators can be carefully tuned (and it is important to do this!).
- Many alternative implementation techniques for each operator; no universally superior technique for most.
- Must consider alternatives for each operation in a query and choose best one based on statistics, etc.
- This is part of the broader task of Query Optimization, which we will cover next!