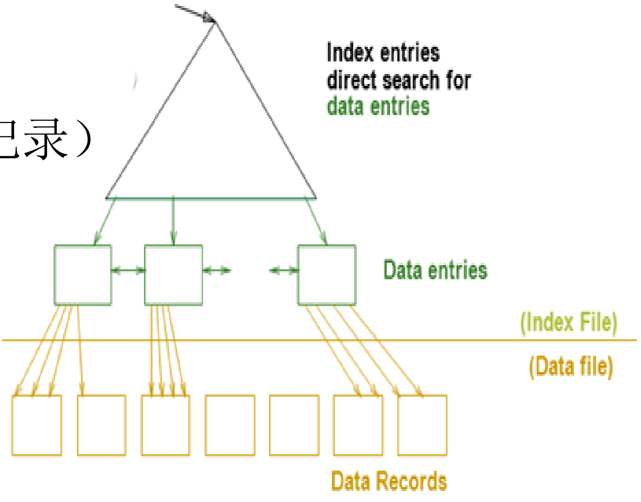
Implementation of Relational Operations

SUN YAT-SEN UNIVERSITY

Review

- Cost Model（代价模型）
  - 只关心磁盘块的IO次数
- 文件由页组成，而每个页包含一组记录
  - Record id = <page id, slot #>
- 索引文件由两部份组成
  1. 数据项部分
     - Data Entry(数据项) $\Longleftrightarrow$ data record（数据记录）
  2. 引导部份
     - 树索引技术 Cost = $\log_F N$ (2~4 I/Os)
     - Hash索引 1~2 I/Os



- 索引的 clustered？
- 外排序算法 COST= $2N(1+\lceil \log_{B-1}\lceil N/B \rceil \rceil)$

Introduction

- Next topic: QUERY PROCESSING(查询求值)
- Some database operations are EXPENSIVE
- Huge performance gained by being "smart"
  - We'll see 10,000x over naïve approach
- Main weapons are:
  - clever implementation techniques for operators
  - Exploiting(利用) 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'
```

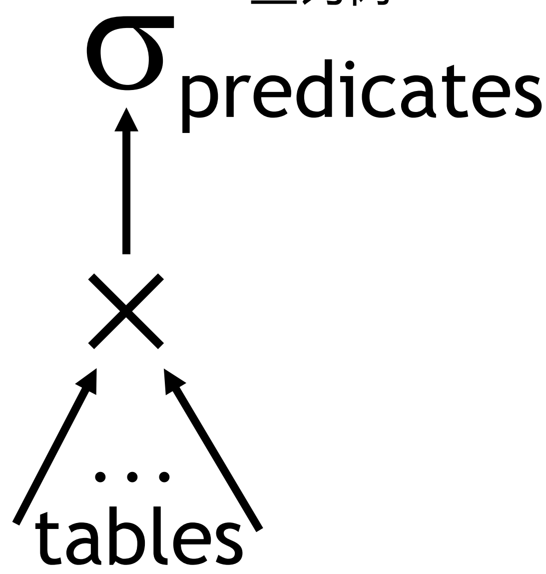
A Really Bad Query Optimizer

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

- For each Select-From-Where query block

  Create a plan（执行计划） that:
  - Forms the cross product of the FROM clause
  - Applies the WHERE clause

查询树

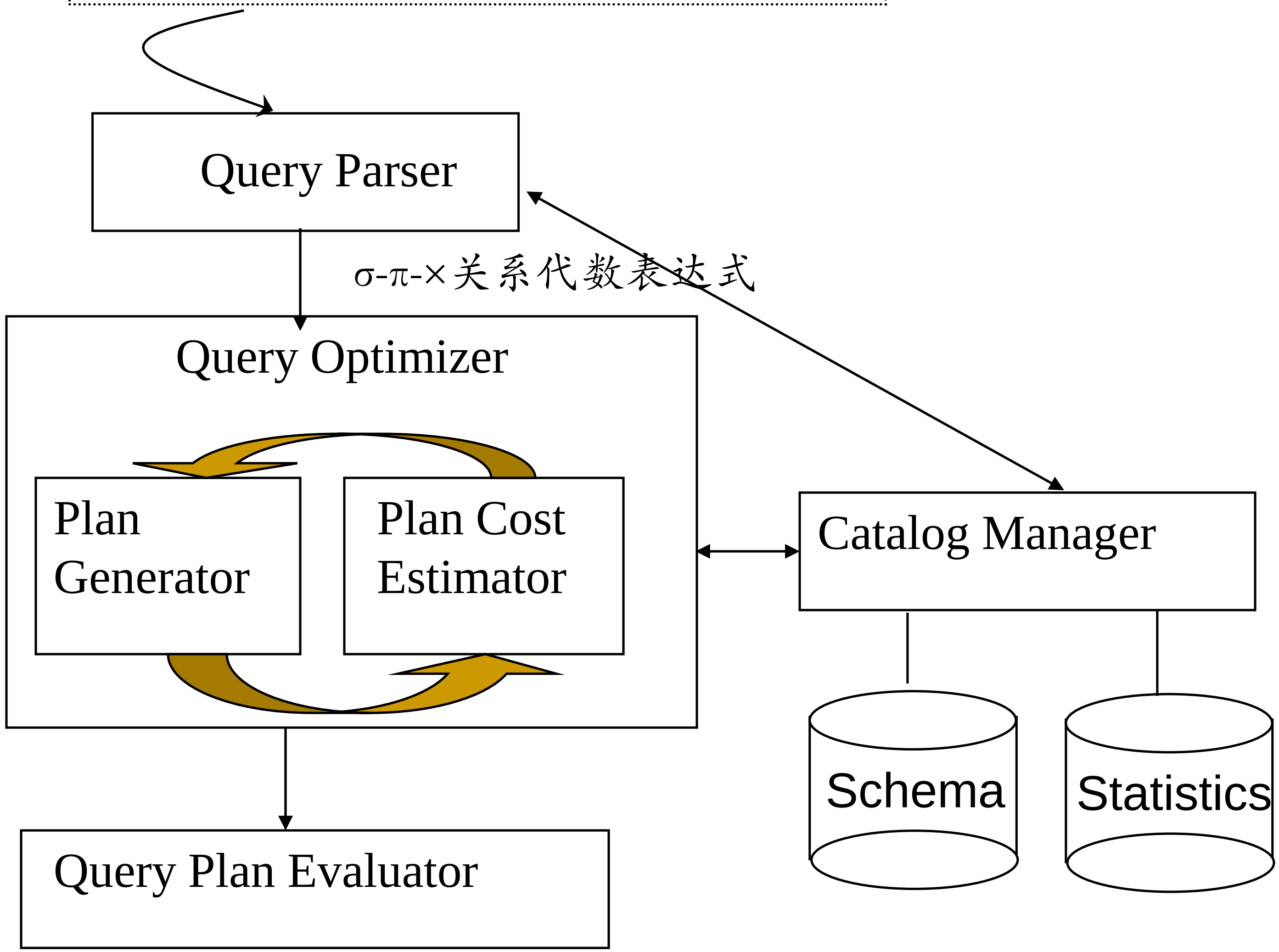


- (Then, as needed:
  - Apply the GROUP BY clause
  - Apply the HAVING clause
  - Apply any projections and output expressions
  - Apply duplicate elimination and/or ORDER BY)

Cost-based Query Sub-System

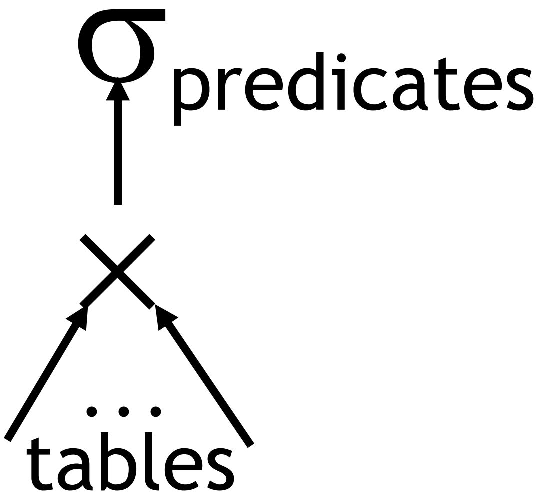
Queries

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



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?

$$\mathbf{R \cap S = R - (R - S)}$$

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(选择性、缩减因子、选中率)* estimated via statistics – we will discuss shortly.

Our options …

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

- If no appropriate index exists:

  Must scan the whole relation

  cost = [R].

  For “reserves” = 1000 I/Os.

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

[S]=500, p_S=80.
[R]=1000, P_R=100

Our options …

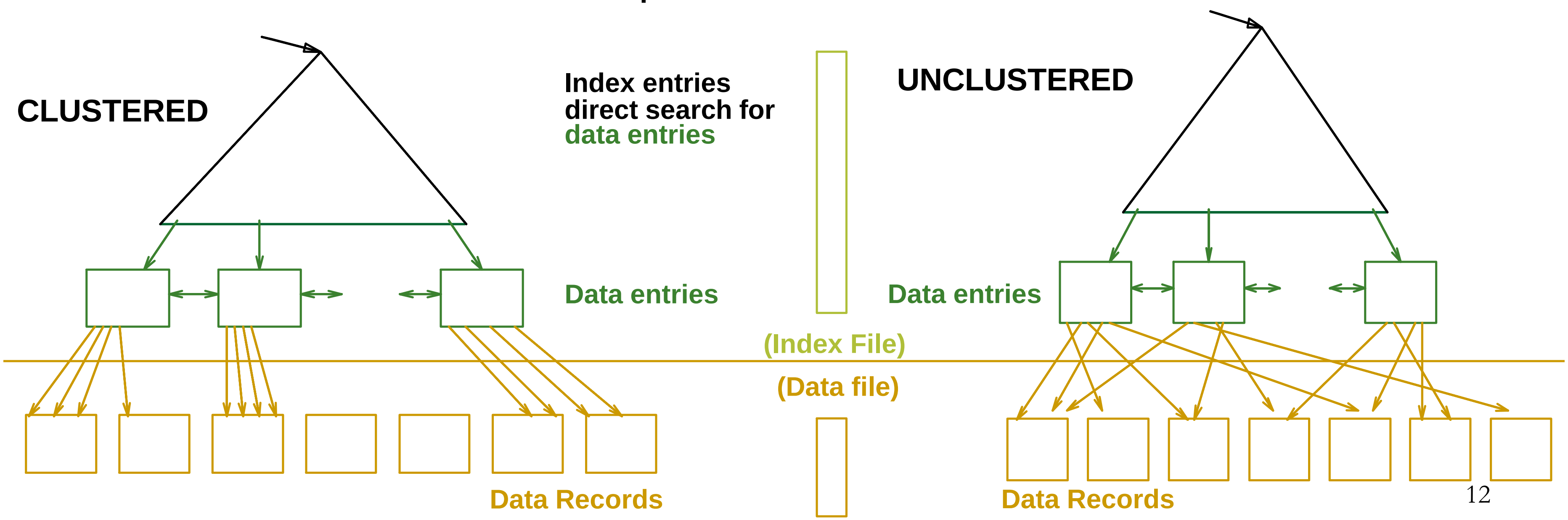
$$\sigma_{R.attr\,op\,value}(R)$$

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

[S]=500, p_S=80.
[R]=1000, P_R=100

- 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, 100*100 tuples):
  - If *clustered* index, cost is a little over 100 I/Os;
  - If *unclustered*, could be up to 100*100 I/Os!   *… unless …*



Refinement for unclustered indexes

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

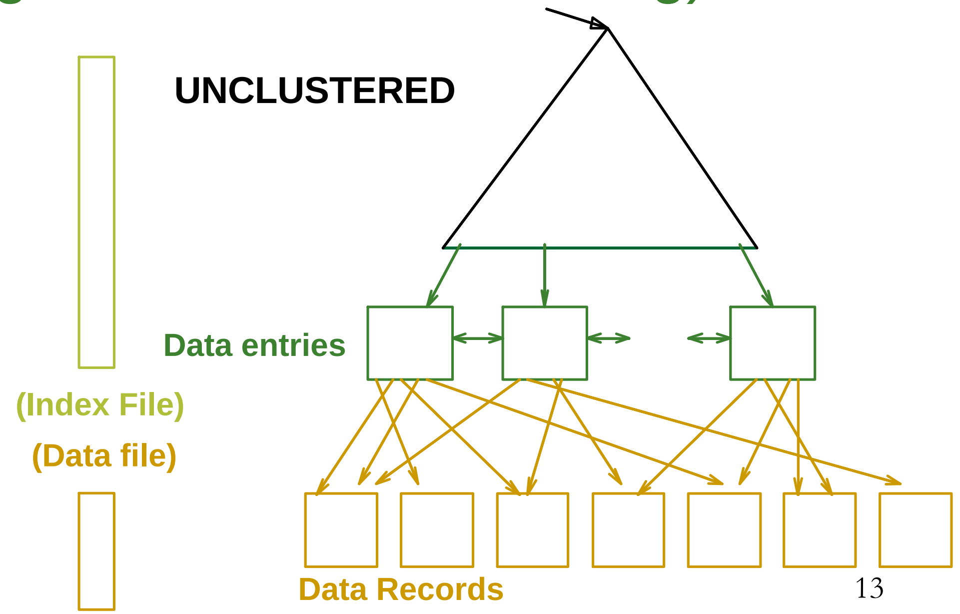
SELECT *
FROM Reserves R
WHERE R.rname < 'C%'

[S]=500, p_S=80.
[R]=1000, P_R=100

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

100*100 tuples

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 *conjunctive normal form* (CNF)-合取范式:
  - *(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 ORs
- Terminology – 索引匹配条件项:
  - A B-tree index *matches* terms(条件项) 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

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

Approach I:

1. Find the *cheapest access path(访问路径)*
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: $\sigma_{\text{day} < 8/9/94 \text{ AND bid}=5 \text{ AND sid}=3}(R)$

some options:

B+tree index on day; 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.)

<u>Approach II</u>: use 2 or more matching indexes.

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

$$\sigma_{day<8/9/94 \text{ AND } bid=5 \text{ AND } sid=3}(R)$$

EXAMPLE:

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

[S]=500, p_S=80.
[R]=1000, P_R=100

- Issue is removing duplicates.
- Use sorting!!
  1. Scan R, extract only the needed attributes
  2. Sort the resulting set
  3. Remove adjacent duplicates

Cost:

- 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

$$2N(1+\lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

Projection -- improved

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

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

- Avoid the temp files, work on the fly(实时流水线):
  - Modify Pass 0 of sort to eliminate unwanted fields.
  - Modify Pass 1+ to eliminate duplicates.

$[S]=500$, $p_S=80$.
$[R]=1000$, $P_R=100$

<u>Cost:</u>

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!)

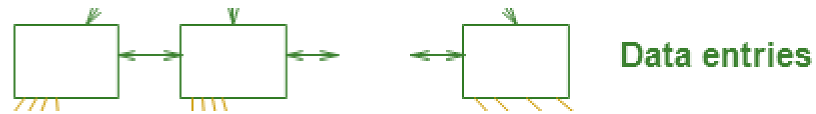
Total cost = 1000 + 250 +250 = 1500.

1000 +250 + 2 * 2 * 250 + 250 = 2500 I/Os $\left(1+\left\lceil \log_{B-1}\left\lceil N/2B\right\rceil\right\rceil\right)$

Other Projection Tricks

[S]=500, p_S=80.
[R]=1000, P_R=100

If an index search key contains all wanted attrs:



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

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

- Do *in-order* index-only (有序唯索引) scan
  - Compare adjacent tuples on the fly *(no sorting required!)*

N

Joins

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

- Joins are very common.
- R $\times$ S is large; so, R $\times$ 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
    4. Hash-join

Simple Nested Loops Join
(简单的嵌套循环连接算法)

[S]=500, p_S=80.
[R]=1000, P_R=100

R $\times$ S: foreach tuple r in R do
    foreach tuple s in S do
        if $r_i == s_j$ then add <r, s> to result

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

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

=1000 + 50,000,000 IOs

140小时

- What if smaller relation (S) was "outer"?

  [S] + (P_S*[S])*[R]
  500 + 80*500*1000

- What assumptions are being made here? **Not all in memory**
- What is cost if one relation can fit entirely in memory?

  [R] + [S]

Page-Oriented Nested Loops Join (页嵌套循环连接算法)

R $\bowtie$ S: 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

[S]=500, p_S=80.
[R]=1000, P_R=100

Cost = [R] + [R] *[S] = 1000 + 1000*500=501,000

1小时

- If smaller relation (S) is outer, cost = 500 + 500*1000 =500,500

1%

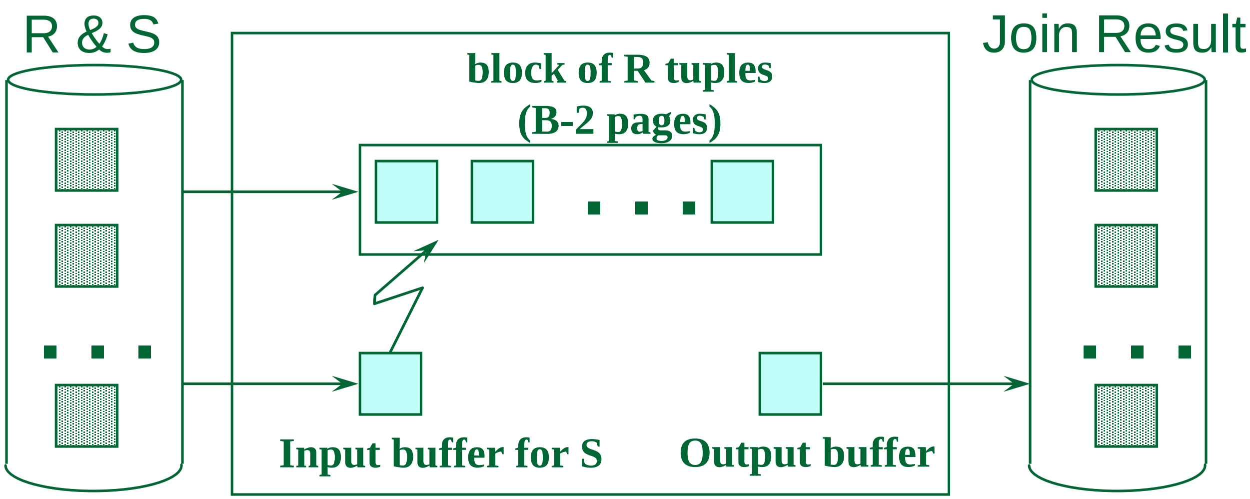
- Much better than naïve per-tuple approach!

Block Nested Loops Join (块嵌套循环连接算法)

foreach block b_R in R do
    foreach page b_S in S do
        …

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

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

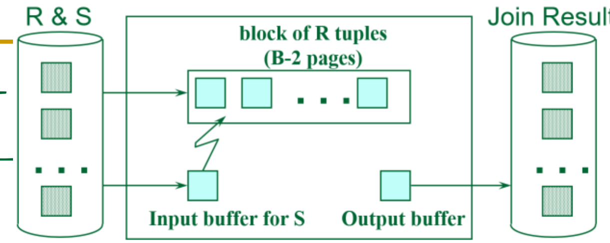


Cost: Scan outer + (#outer blocks * scan inner)

#outer blocks = $\left\lceil \# of\ pages\, of\ outer / blocksize \right\rceil$

Cost = [R] + [R]/N *[S]

Examples of Block Nested I



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

  #outer blocks = [outer] / 100

[S]=500, p_S=80.
[R]=1000, P_R=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=6000.

Cost = [R] + [R]/N *[S]

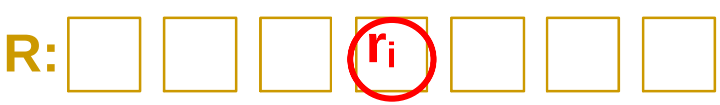
- With S as outer ([S] = 500):
  - Scanning S costs 500 IO's *(done in 5 blocks)*
  - Per block of S, we can R; costs 5*1000 IO's
  - Total = 500 + 5*1000=5500.

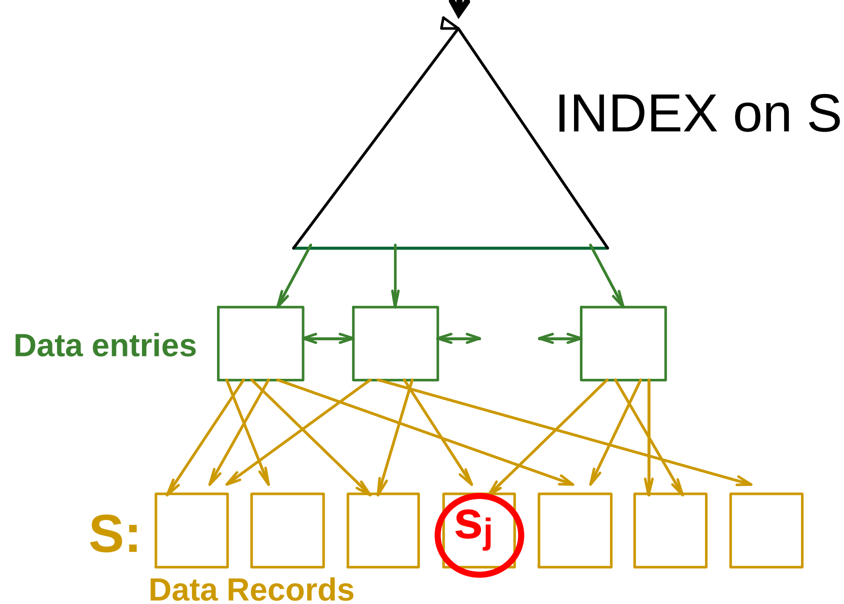
55秒 1/10000

Index Nested Loops Join
(索引嵌套循环连接算法)

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

lookup(r_i)





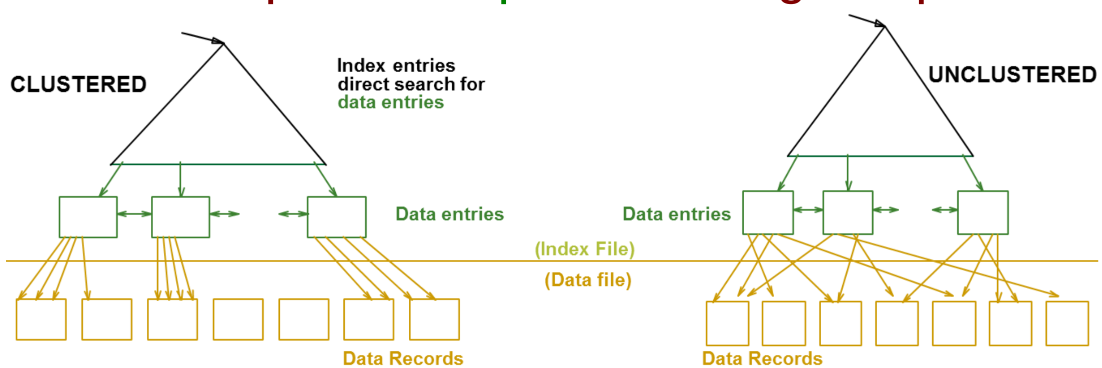
Index Nested Loops Join

[S]=500, p_S=80.
[R]=1000, P_R=100

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

Cost = [R] + ([R]*P_R) * <u>cost to find matching S tuples</u>

- 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 <u>per page</u> of matching S tuples.
     - Unclustered: up to 1 I/O <u>per</u> matching S <u>tuple</u>.



Sort-Merge Join
(排序归并连接算法)

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

[S]=500, p_S=80.
[R]=1000, P_R=100

- Cost: Sort R + Sort S + ([R]+[S])
  - But in worst case, last term could be [R]*[S] *(very unlikely!)*
  - Q: what is worst case? 每个元组的连接属性值都一样

Suppose B = 35 buffer pages:

$$2N(1+\lceil \log_{B-1} \lceil N/B \rceil \rceil)$$

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

Suppose B = 300 buffer pages:

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

Join cost = [outer] + (#outer blocks * [inner])
500+500/(35-2) * 1000
500+500/(300-2) * 1000

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

Other Considerations …

[S]=500, p_S=80.
[R]=1000, P_R=100

1. An important refinement:

   Do the join during the final merging pass of sort !

   - If have enough memory, can do:
     1. Read R and write out sorted runs
     2. Read S and write out sorted runs
     3. Merge R-runs and S-runs, while finding R⋈S matches

   Cost = 3*[R] + 3*[S]

   Q: how much memory is "enough" ?

   $B>2^*\sqrt{Max([R],[S])}$
   超过R和S的有序段的数目

2. **Sort-merge join an especially good choice if:**
   –one or both inputs are already sorted on join attribute(s)
   –output is required to be sorted on join attributes(s)

$$2N(1+\lceil \log_{B-1}\lceil N/B \rceil \rceil)$$

Hash-join(哈希连接算法)

Cost = 3 * [R] + 3 *[S]

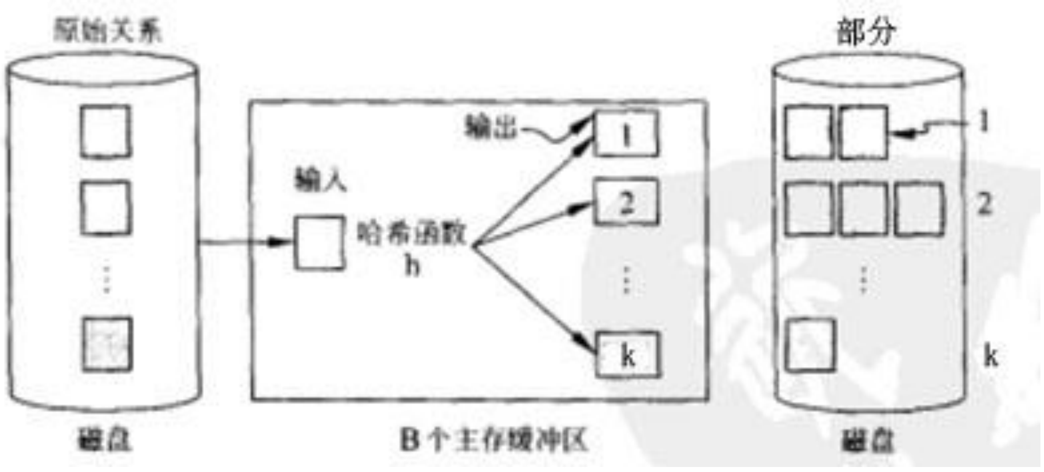
- 基本思想：先（第1步）对两个关系的连接属性应用同一个Hash 函数，将每个关系划分到 k 个部分。显然可以保证部分 i 的 R_i 元组只能与同一部分的 S_i元组连接。
  然后（第2步），我们可以读入较小关系R的一个部分R_i （即可以将 R_i 全部读入内存中），然后扫描关系S的对应部分S_i，来找到匹配的元组。

- 算法分为两个阶段
  - partitioning phase划分阶段
  - probing phase连接阶段 (也称匹配、探查阶段)

Hash-join(哈希连接算法)-划分阶段

Cost = 3 * [R] + 3 *[S]

- partitioning phase划分阶段
  - 使用同一个Hash函数 h 对关系 R 和 S 进行划分。完成后，所有元组划分为k个部分，每个部分中的元组有相同的哈希值（对元组的连接属性计算哈希函数h）。



  - 在划分阶段需要扫描关系 R 和 S，还要写出一次，因此

Cost = 2 * [R] + 2 *[S]

Hash-join(哈希连接算法)-连接阶段

Cost = 3 * [R] + 3 *[S]

- probing phase连接阶段
  - 比较属于同一部分的 R_i 元组和 S_i 元组，找到匹配元组。
  - 实际处理时，为了减少CPU开销，使用与 h 不同的Hash 函数h_2对较小的部分（如R_i）进行划分，在内存中创建R_i部分的哈希表。

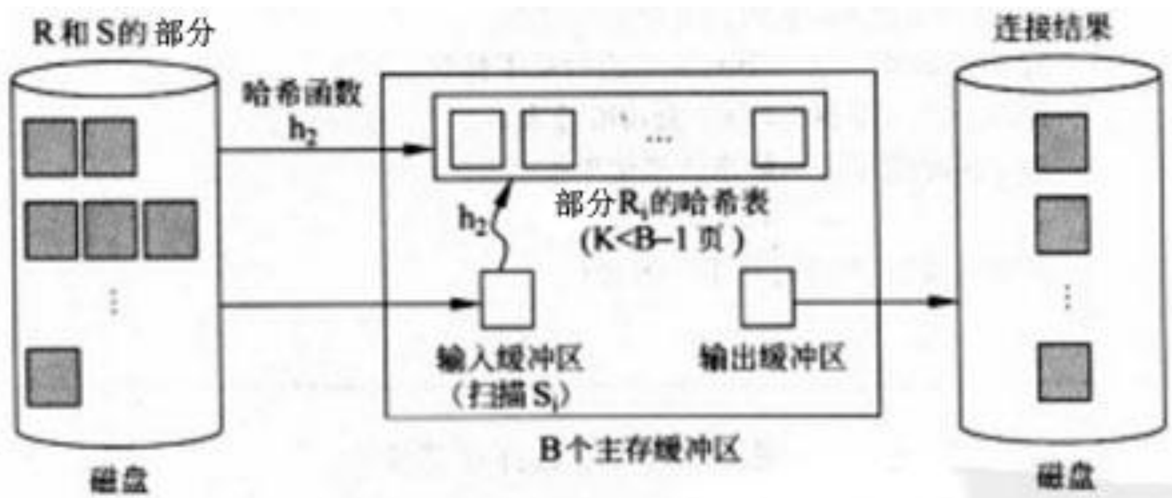


图 14.11 哈希连接算法的连接阶段

  - 在连接阶段，每个部分需要扫描一次，故 Cost = [R] + [S]

Summary

- A virtue of relational DBMSs:

> queries are composed of a few basic operators

- Many alternative implementation techniques for each operator
  - No universally superior technique for most operators.
- Must consider available alternatives
  - Called “Query optimization” -- we will study this topic soon!
- 要求:
  - 根据给定条件，计算关系运算 σ、π和⋈ 的 COST，如：

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

Cost = [R] + [R]/N * [S]

Cost: Sort R + Sort S + ([R]+[S])