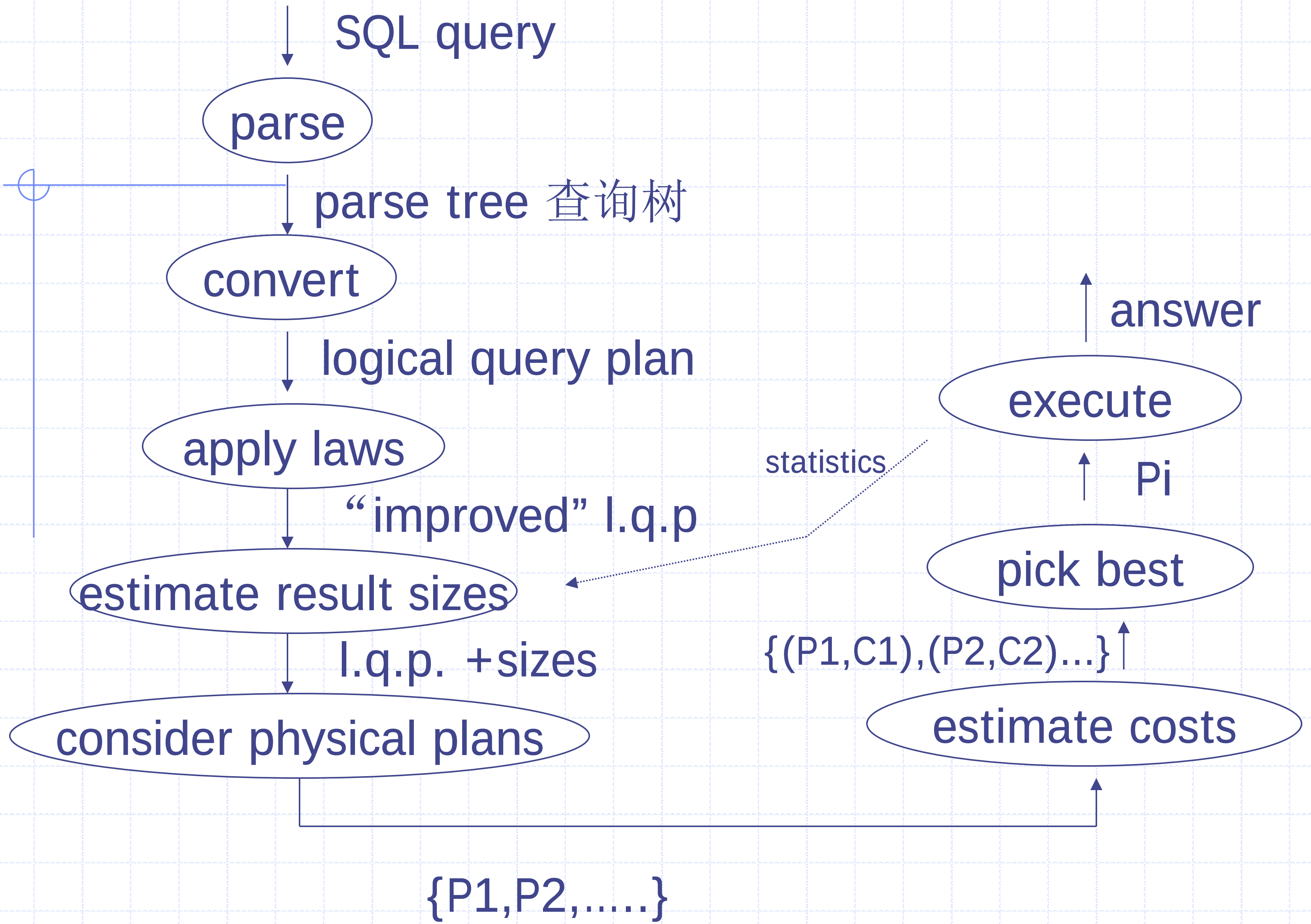
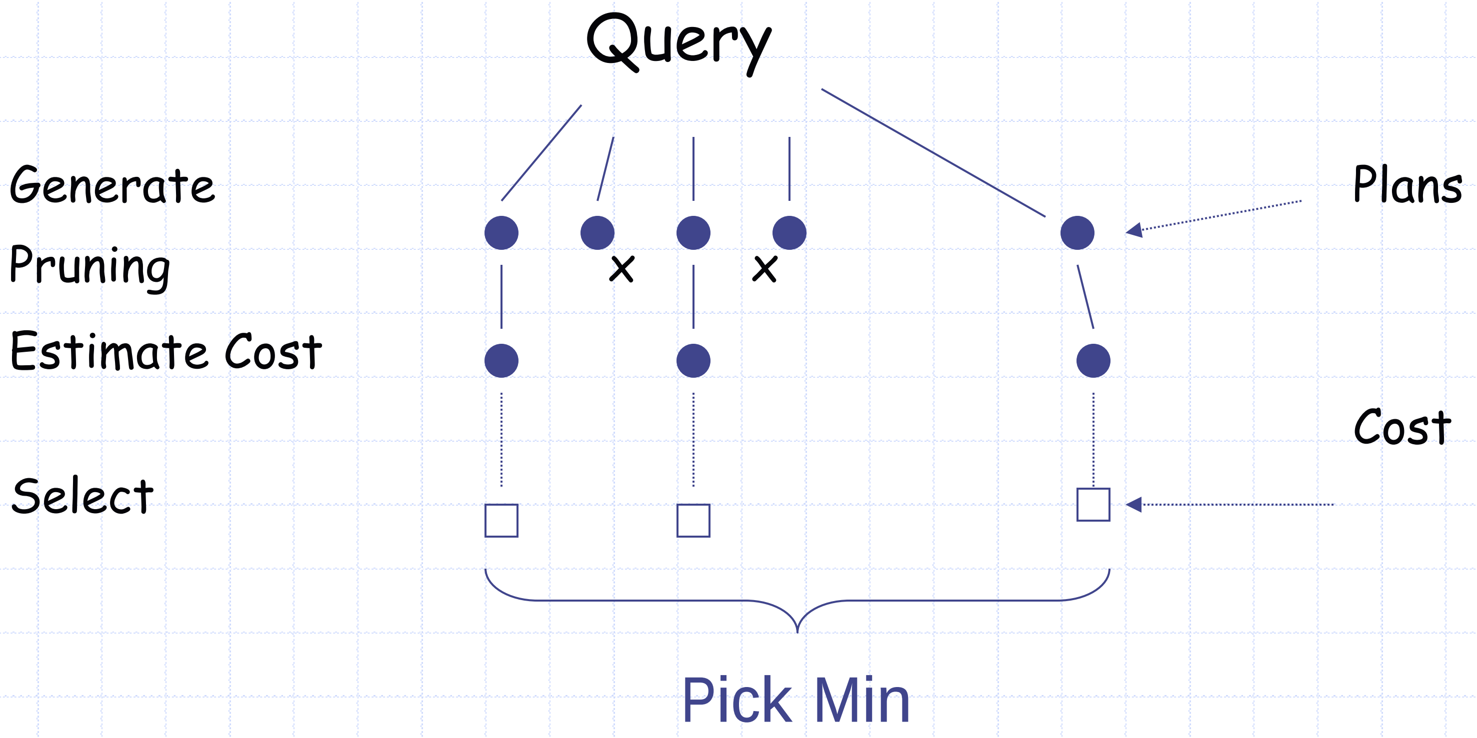
Database System Principles

Query Optimization


SQL query
parse
parse tree 查询树
convert
logical query plan
apply laws
“improved” l.q.p
estimate result sizes
statistics
l.q.p. +sizes
consider physical plans
{P1,P2,.....}
estimate costs
{(P1,C1),(P2,C2)...}
pick best
Pi
execute
answer

Query Optimization

--> Generating and comparing plans



To generate plans consider:

- Transforming relational algebra expression

  (e.g. 连接顺序的变换)
- Use of existing indexes
- Building indexes or sorting on the fly

- Implementation details:

  e.g. - Join algorithm

  - Memory management
  - Parallel processing

下面介绍几个重要的物理查询代价的参数

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

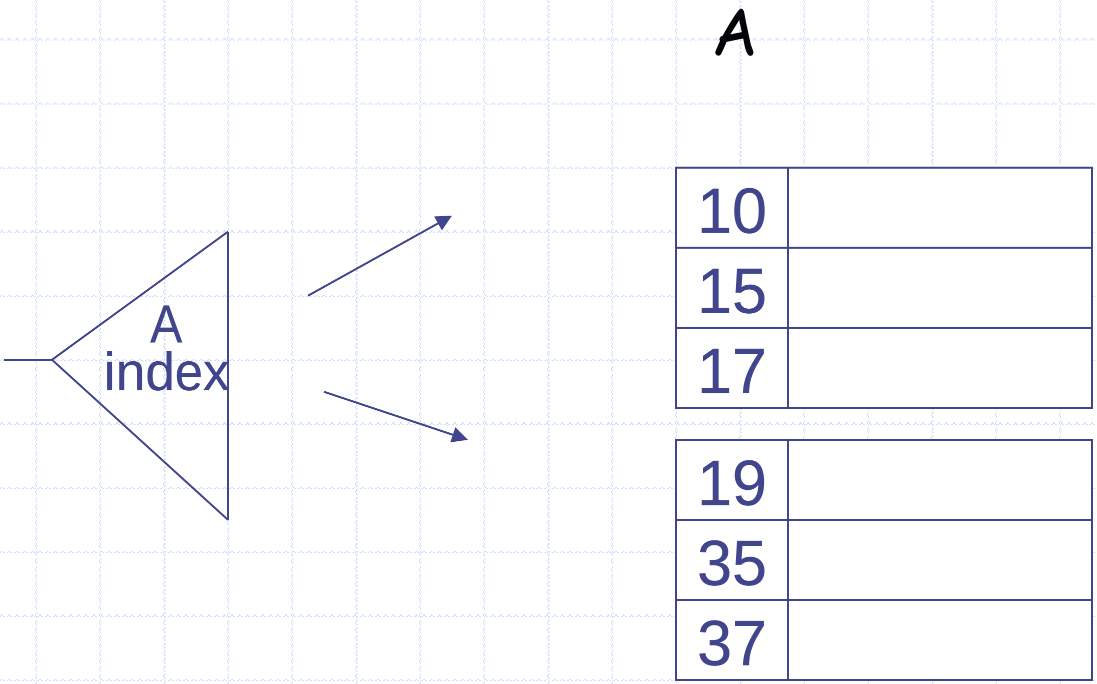
HT(i) = # levels in index i

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

簇集的几个概念 →

Clustering index

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



Notions of clustering P301

- Clustered file organization (tuples of R are placed with tuples of S which they share a common value.)

R1 R2 S1 S2	R3 R4 S3 S4	…..

- Clustered relation (tuples of the relations are exclusively stored in blocks)

R1 R2 R3 R4	R5 R5 R7 R8	…..

- Clustering index (tuples with a fixed value will be stored consecutively)

a1 a1	a1 a1 a1 a1	a1 a1

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

→ Metric: # of IOs

(ignoring writing of result)

Options

- Transformations: R1 $\bowtie$ R2, R2 $\bowtie$ R1
- Joint algorithms:
  - Iteration (nested loops嵌套循环法)
  - Merge join(排序归并法)
  - Join with index (索引连接法)
  - Hash join (散列连接法)

- <u>Iteration join</u> (conceptually)

```
for each r ∈ R1 do
    for each s ∈ R2 do
        if r.C = s.C then output r,s pair
```

取**R1**的一个元组，与**R2**的所有元组比较，凡满足连接条件的元组就进行连接并且作为结果输出；

然后再取**R1**的下一个元组，与**R2**的所有元组比较；

直至**R1**的所有元组与**R2**的所有元组比较完为止。

注：将物理块少的关系作为外关系，可以减少**IO**次数。

- Merge join (conceptually)

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

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

While $(i \leq T(R1)) \wedge (j \leq T(R2))$ 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 $i \leftarrow i+1$

如果两个关系按照连接属性排序，则可以按序比较。

归并连接法只需扫描一次两个关系。

Example

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

Join with index (Conceptually)

For each $r \in R1$ do

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

for each $s \in X$ do

output r,s pair]

Assume R2.C index

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

then X = set of rel tuples with attr = value

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

影响连接操作性能的因素

Factors that affect performance

(1) Tuples of relation stored physically together?

(2) Relations sorted by join attribute?

(3) Indexes exist?

举例说明

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

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

Cost: for each R1 tuple:

[Read tuple + Read R2]

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

- Can we do better?

Use our memory

(1) Read 100 blocks of R1

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

(3) Repeat until done

<u>Cost:</u> for each R1 chunk:

Read chunk: 1000 IOs(10×100)

Read R2 5000 IOs

$$\frac{}{6000}$$

$$\text{Total} = \frac{10{,}000}{10 \times 100} \times 6000 = 60{,}000 \text{ IOs}$$

Each block contain 10 record

- Can we do better?

  - Reverse join order: $R2 \bowtie R1$

$$\text{Total} = \frac{5000}{1000} \times (1000 + 10{,}000) =$$

$$5 \times 11{,}000 = 55{,}000 \text{ IOs}$$

物理块少的关系作为外关系，可以减少**IO**次数

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

- Relations contiguous

Cost

For each R2 chunk:

Read chunk: 100 IOs(100块)

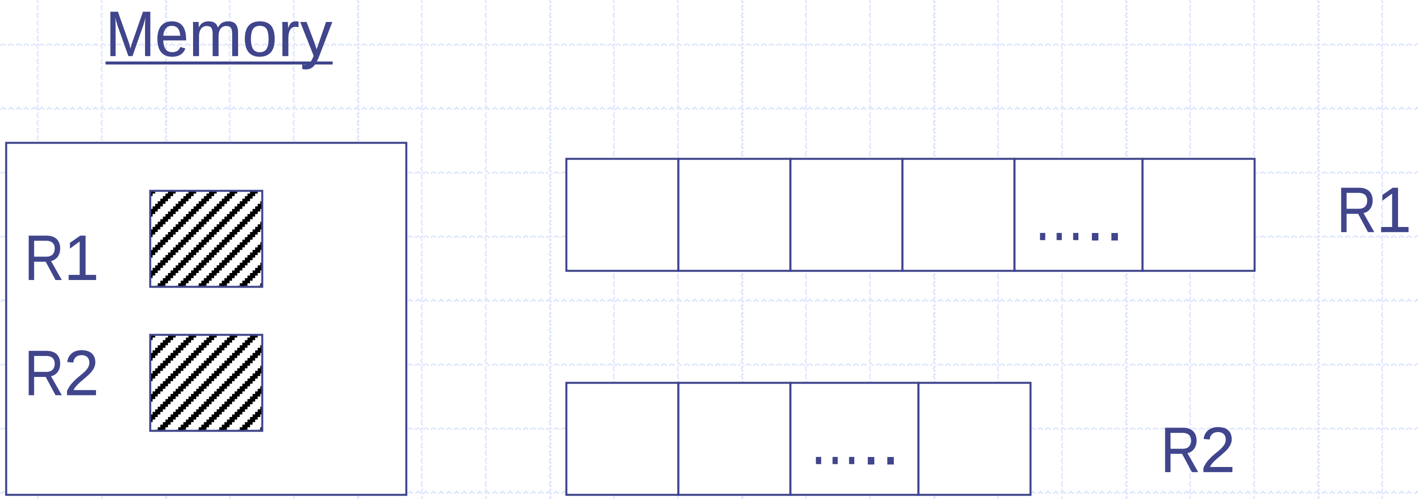
Read R1: $\underline{1000}$ IOs(1000块)

1,100

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

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

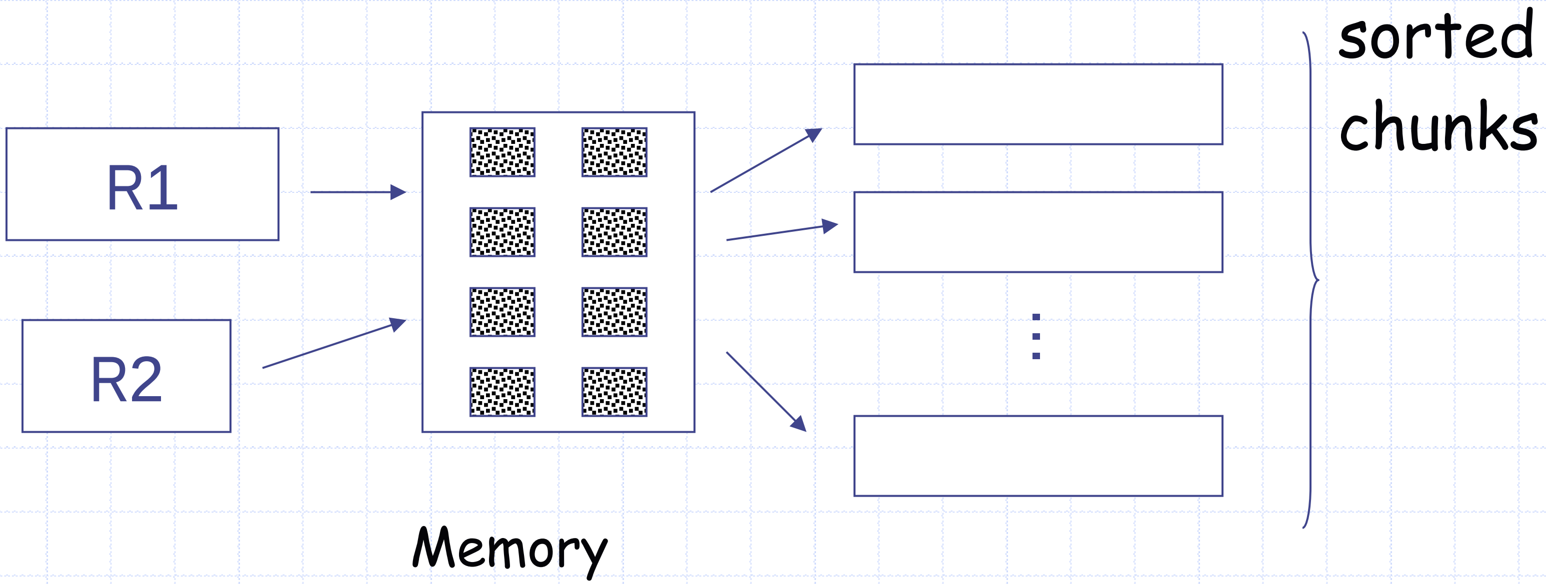
- R1, R2 not ordered, but contiguous

--> Need to sort R1, R2 first.... HOW?

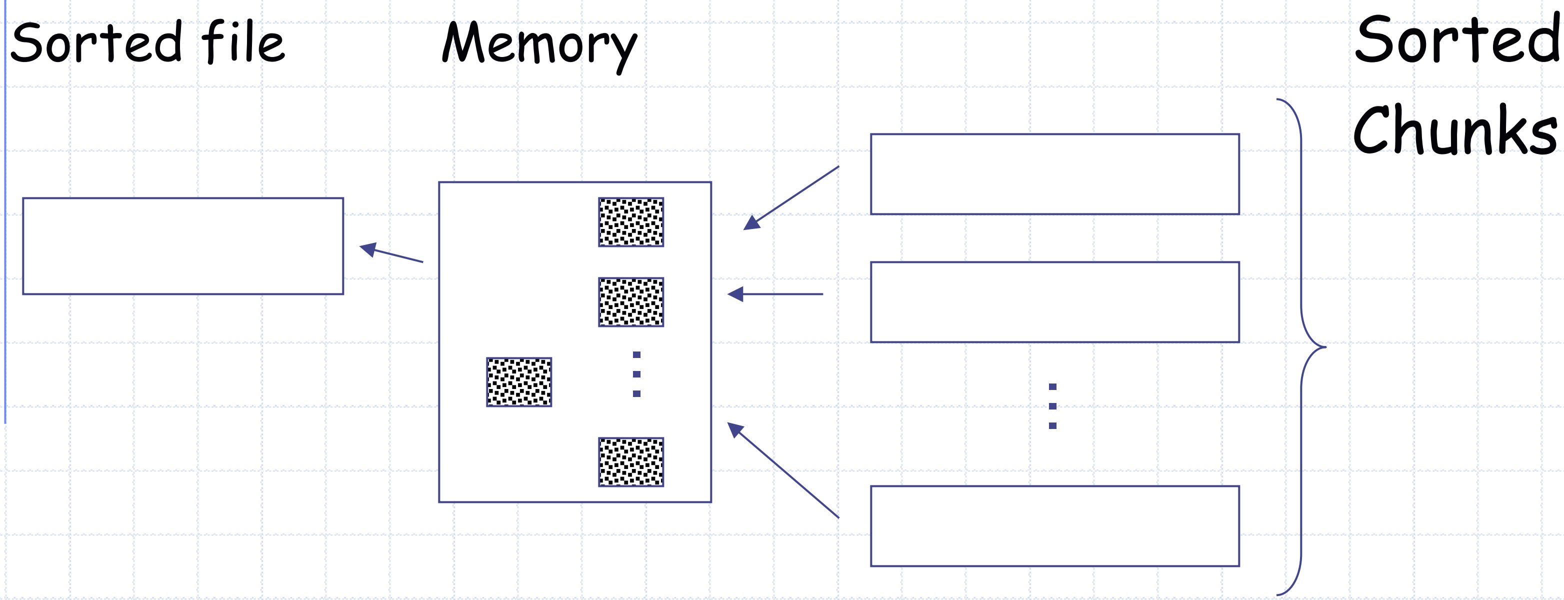
One way to sort: Merge Sort

(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: $4 \times 1{,}000 = 4{,}000$

Sort cost R2: $4 \times 500 = 2{,}000$

Example 1(d) Merge Join (continued)

R1,R2 contiguous, but unordered

Total cost = sort cost + join cost
= 6,000 + 1,500 = 7,500 IOs

But: Iteration cost = 5,500
so merge joint does not pay off!
并不是所有情况下，归并法都要好于嵌套法！

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

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

Join selectivity 连接选择因子
(a) js<=1/T(R)
(b) js =1/V(R,C)

Total cost with index join

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

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

What if index does not fit in memory?

Example: say R1.C index is 201 blocks

- 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)

So far

not contiguous

Iterate R2 $\bowtie$ R1	55,000 (best)
Merge Join	______
Sort+ Merge Join	______
R1.C Index	______
R2.C Index	______

contiguous

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

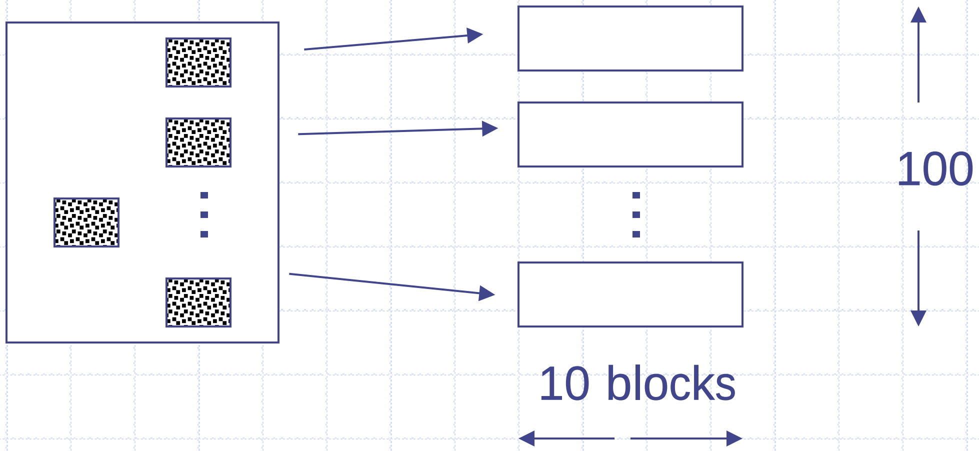
Example 1(f) Hash Join

- R1, R2 contiguous (un-ordered)

→ Use 100 buckets

→ Read R1, hash, + write buckets

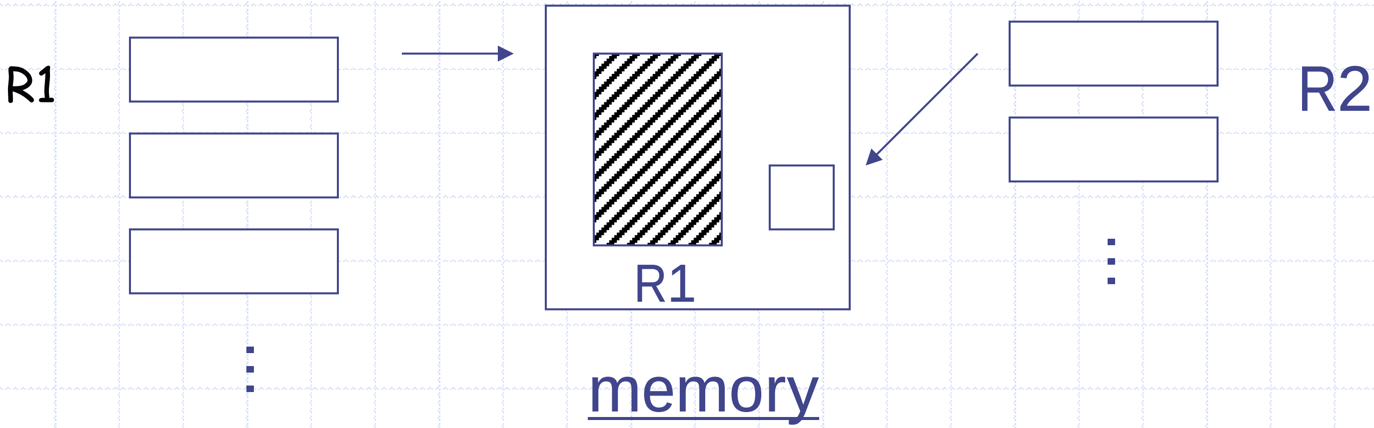
R1 →



-> 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

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

Summary

- 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 could be useful

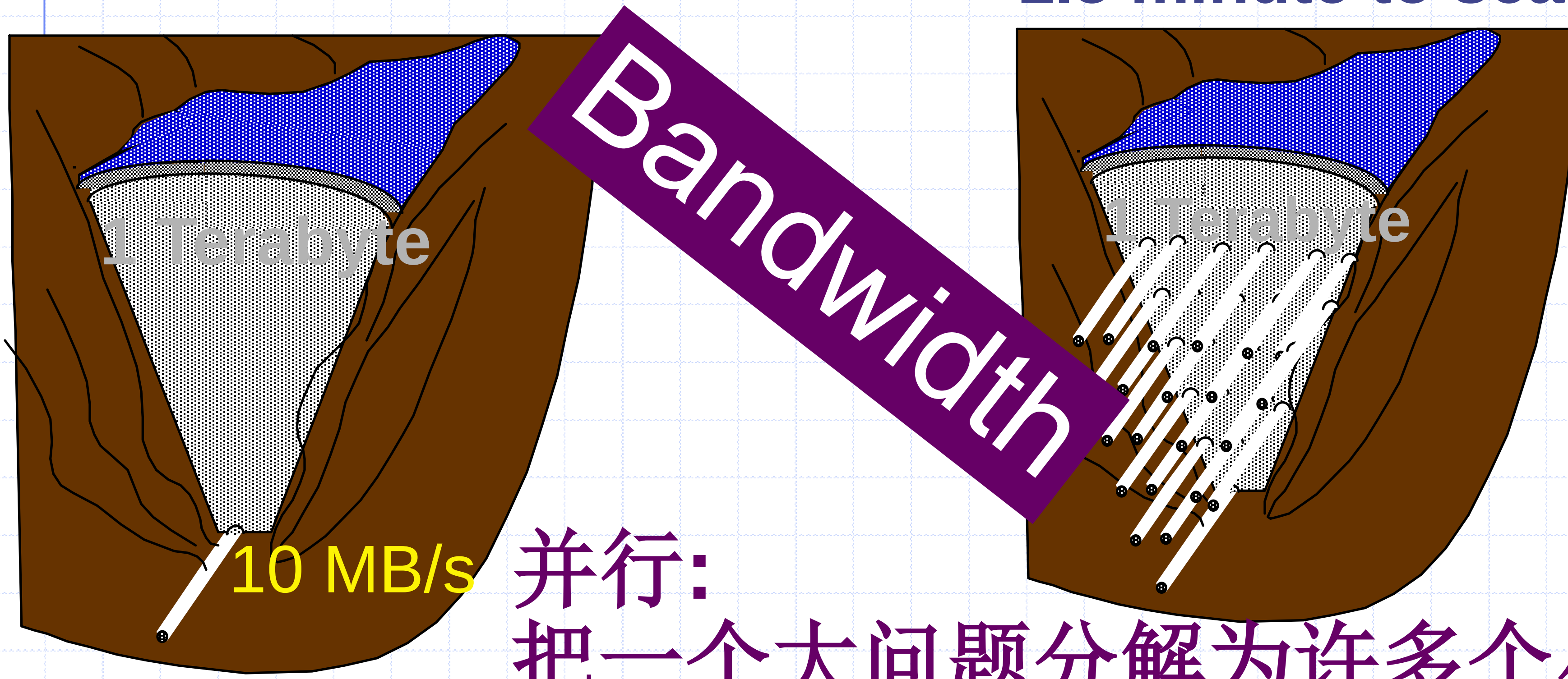
(depends on expected result size)

Join strategies for parallel processors

怎样并行存取数据?

以 **10 MB/s**的速度需要扫描
1.2天

1,000 x parallel
1.5 minute to scan.

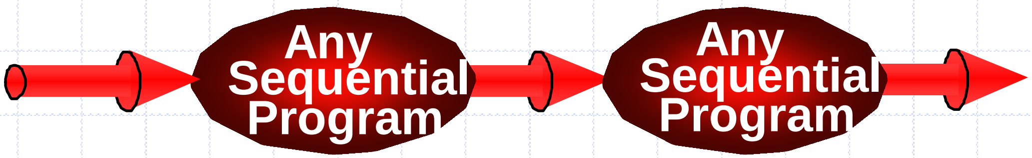


并行:
把一个大问题分解为许多个小问题来并行解决.

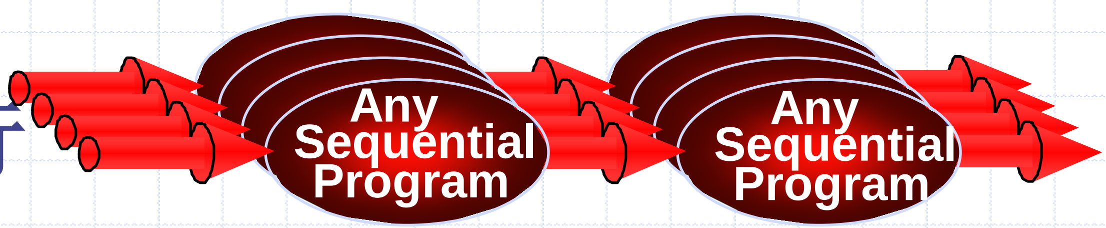
并行 DBMS: 简介

- 并行对于DBMS 处理很自然
  - *流水线并行Pipeline parallelism:* 在一个多步的处理中，每个机器做其中一步.
  - *分块并行Partition parallelism:*许多机器对数据的不同部分作相同的事情.
  - 都出现在DBMS的技术中!

流水线



分块并行

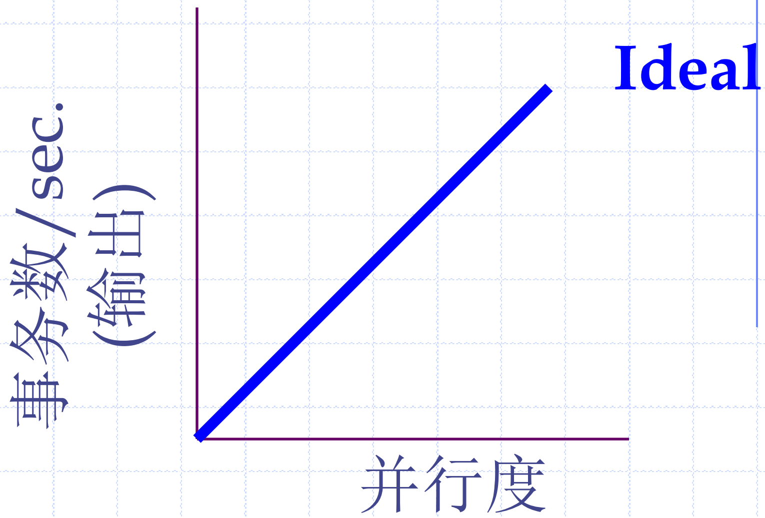


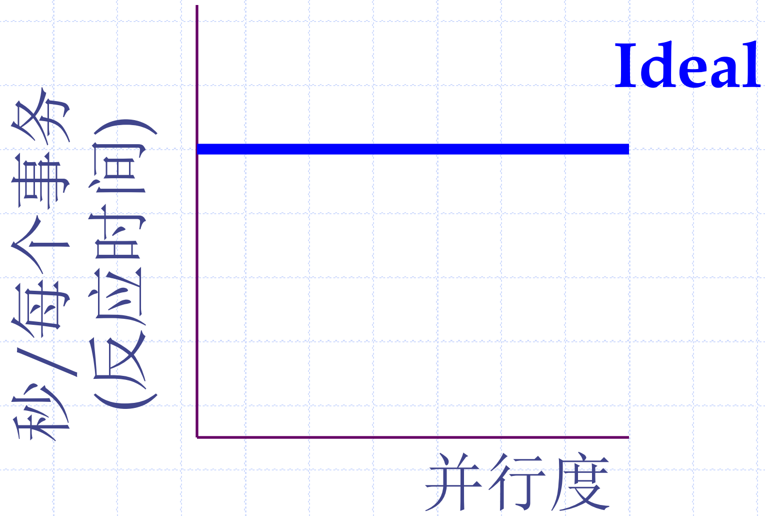
DBMS: || 并行的成功历史

- DBMSs 是并行应用的最成功的领域.
  - 每个大的 DBMS厂商都有并行服务器
  - 工作站厂商依靠并行数据库服务器的销售.
- 成功的原因:
  - 块处理Bulk-processing (= 分块并行).
  - 流水线特性Natural pipelining.
  - 硬件花费不高!
  - 用户/应用程序可以不关心并行

并行的术语

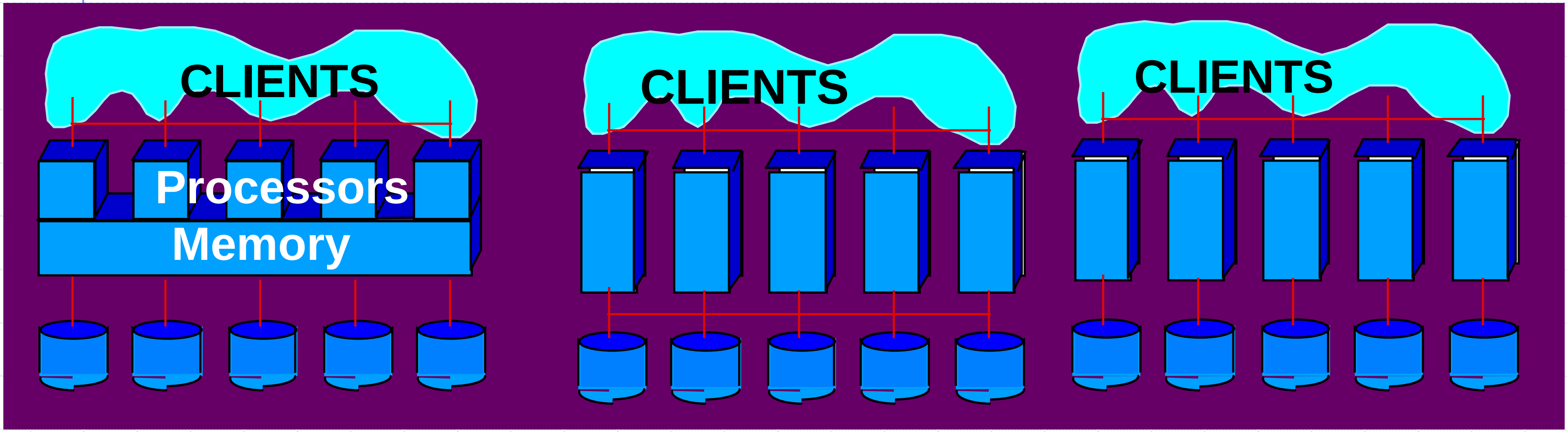
- 加快速度Speed-Up
  - 对于给定的数据，多的处理器可以花费小的时间.
- 规模扩大Scale-Up
  - 如果资源随着数据的规模而增长，处理时间是常量.





结构问题: 共享什么?

共享主存 (SMP) | 共享磁盘 | 什么也不共享 (network)



容易编程
建立代价高
很难扩展

编程困难
搭建代价低
容易扩展

Sequent, SGI, Sun | **VMScluster, Sysplex** | **Tandem, Teradata, SP2**

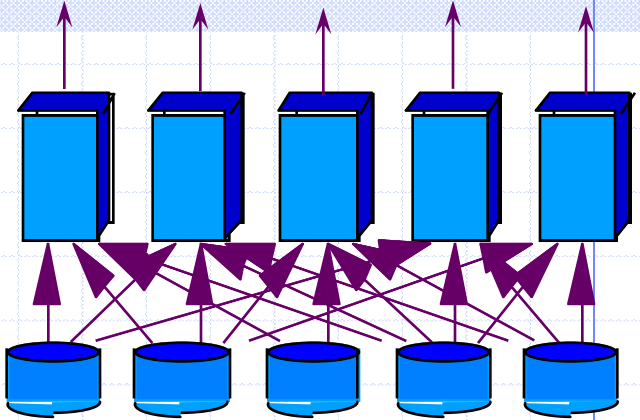
不同类型的并行

- 操作内部的并行
  - 所有的机器执行一个给定任务 (扫描,排序, 连接)
- 操作之间的并行
  - 每个操作可以并发地在不同地方执行 (exploits pipelining)
- 查询之间的并行
  - 不同的查询在不同的地点执行

并行扫描

- 并行扫描，归并
- 索引可以建立在每个数据分割上.

并行排序



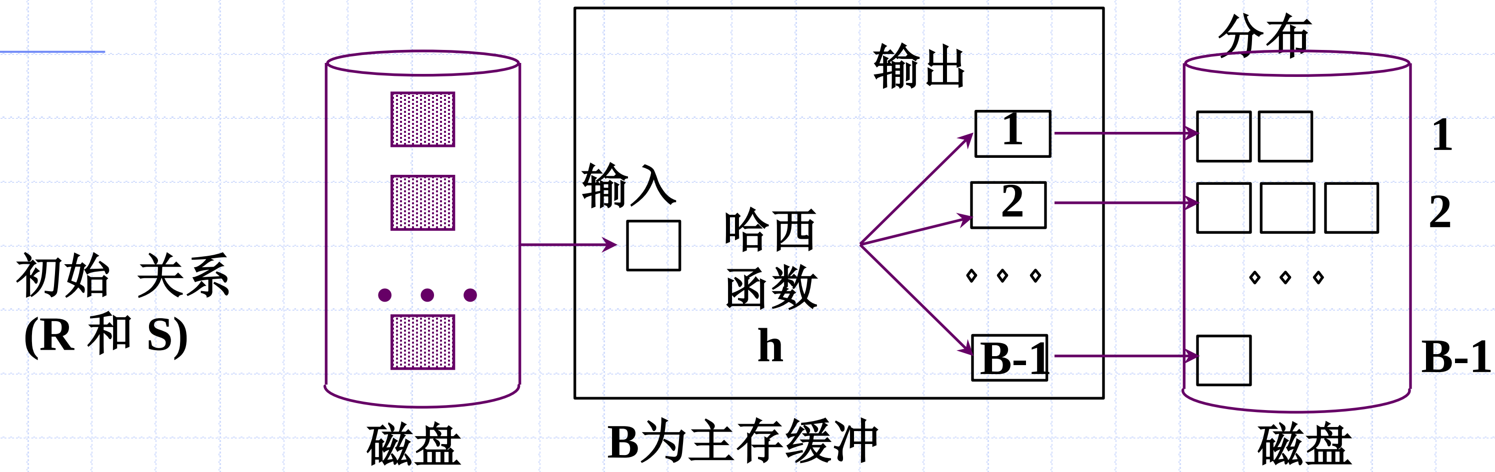
- 2000年左右:
  - 8.5 Gb/minute, shared-nothing; Datamation benchmark in 2.41 secs (UCB! http://now.cs.berkeley.edu/NowSort/)
- 方法:
  - 并行扫描, 进行基于范围(区域)的分布.
  - 当元组进入的时候, 在每个元组上进行 “local”局部排序
  - 结果数据是排序的, 并且是范围分割的.
  - 问题: *数据分布问题skew!（如何进行区域划分，使得每个处理机分布的记录数目近似相等。）*
  - 解决方法:开始时，取样 “sample” 数据获得分割点.

并行连接

- 嵌套循环:
  - 每个内部元组必须和每个可能连接的外部元组比较.
  - 在连接列上的范围分割比较容易, 其它的操作很难!
- 排序-归并 (or plain Merge-Join):
  - 根据连接属性的取值范围进行分布.
  - 分布表的合并是局部进行的.

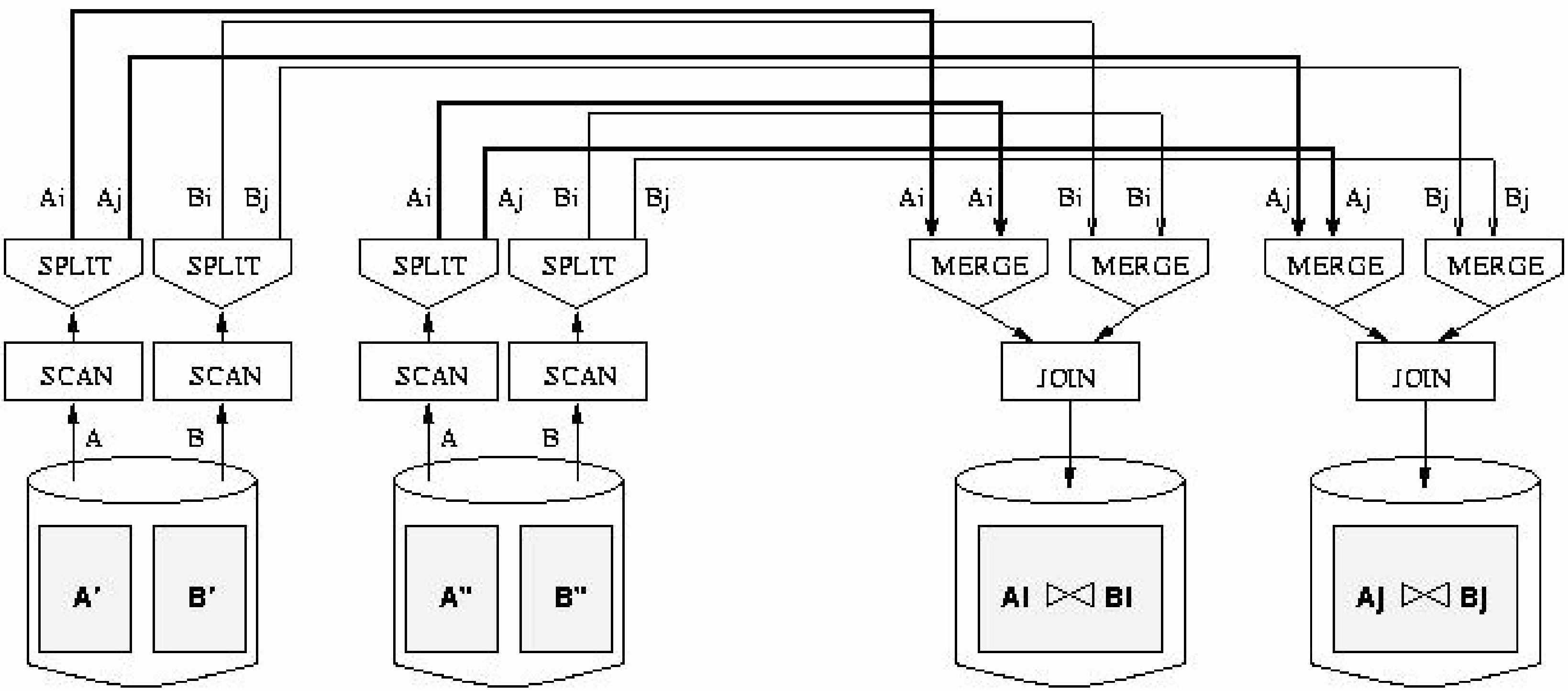
并行哈希连接

第一阶段



- 在第一阶段, 分割分布在不同站点的数据:
  - 一个好的哈希函数自动分布工作负载!
- 在每个站点作第二阶段.
- 以等值连接为例.

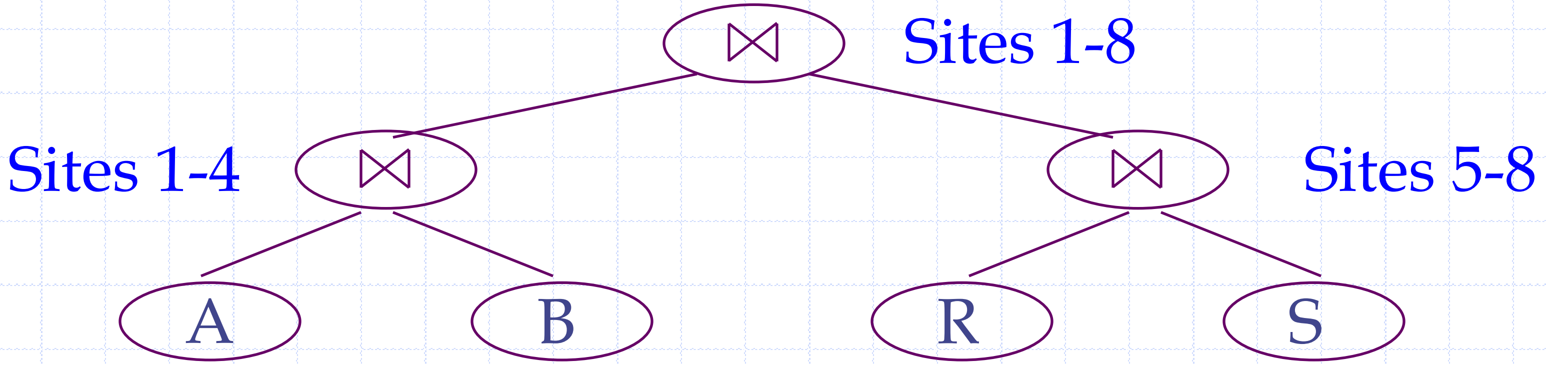
并行连接的数据流网络



- 能处理好分割和合并这两个问题，可以使建立并行的顺序连接容易实现.

复杂的并行查询计划

- 复杂查询: 操作之间可以并行
  - 操作之间的并行:



结论

- 建立一个快速的并行查询执行器相对容易
- 但是，写一个强壮的查询优化器很难.
  - 存在很多问题.
  - 仍在研究中!

Chapter 7 summary

- Relational algebra level
- Detailed query plan level
  - Estimate costs
  - Generate plans
    - Join algorithms
  - Compare costs