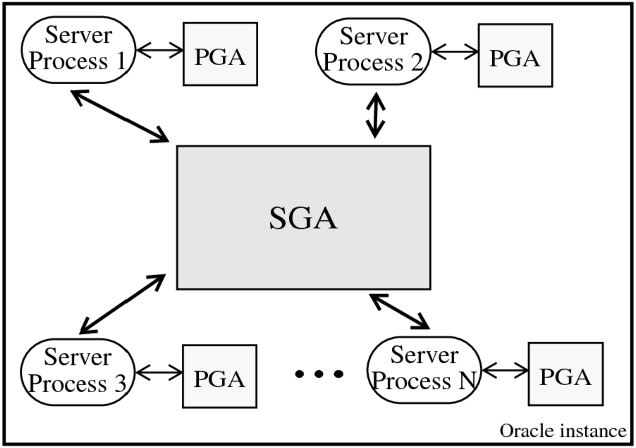
CS5226 Lecture 4
Memory Tuning

Memory Management

- **Shared data and control structures**:
  - buffer cache
  - catalog metadata
  - log buffer
  - lock structures
  - SQL execution plans
  - etc
- **Local data and control structures**:
  - operator work area = memory allocated for evaluating an SQL operator
  - Memory-intensive operators:
    - sort
    - hash join

Oracle Memory Model



(Dageville & Zait, VLDB 2002)

System Global Area (SGA) = shared memory region

Process Global Area (PGA) = private memory region

Oracle Memory Management

- **Pre 9i** - Manual memory mangement
  - **Parameters for tuning SGA**:
    - DB_CACHE_SIZE, SHARED_POOL_SIZE, LARGE_POOL_SIZE, JAVA_POOL_SIZE, etc.
  - **Parameters for tuning PGA**:
    - SORT_AREA_SIZE, HASH_AREA_SIZE, BITMAP_MERGE_AREA_SIZE, etc.
- **9i** - Automatic PGA memory management
  - PGA_AGGREGATE_TARGET parameter
- **10g** - Automatic SGA memory management
  - SGA_TARGET parameter
- **11g** - Automatic memory management
  - MEMORY_TARGET parameter

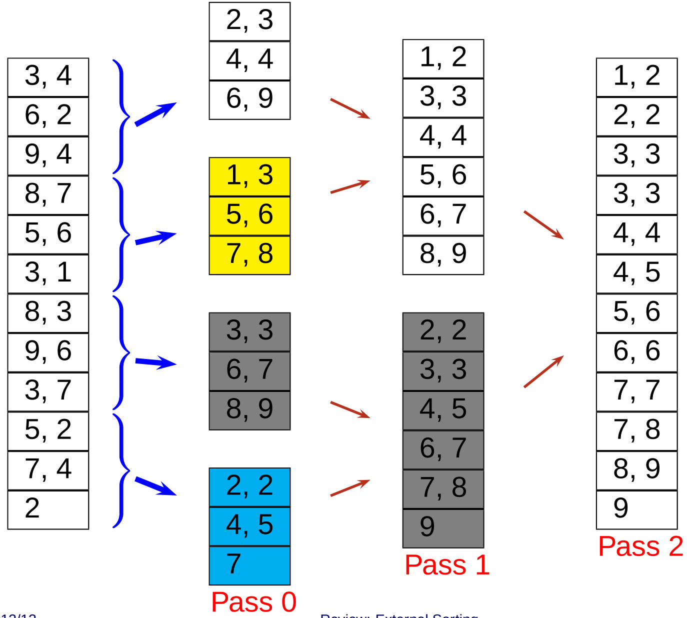
Memory-Intensive Operators

- Examples of memory-intensive operators:
  - sort
  - hash join
- How to manage memory for such operators?

Review: External Merge Sort

- Sorting file of N pages with B buffer pages
- **Pass 0**: Creation of sorted runs
  - Read in and sort B pages at a time
  - Number of sorted runs created = $\lceil N/B \rceil$
  - Size of each sorted run = B pages (except possibly for last run)
- **Pass i,** $i \geq 1$: Merging of sorted runs
  - Use $B - 1$ buffer pages for input & one buffer page for output
  - Performs (B-1)-way merge
- Analysis:
  - N_0 = number of initial sorted runs = $\lceil N/B \rceil$
  - total number of passes = $\lceil \log_{B-1}(N_0) \rceil + 1$

External Merge Sort with B=3: Example



Review: Hash Join, $R \bowtie S$

- Idea:
  - Partition R and S into k partitions using some hash function h
    - ★ $R = R_1 \cup R_2 \cup \cdots \cup R_k$
    - ★ $S = S_1 \cup S_2 \cup \cdots \cup S_k$
  - Joins corresponding pair of partitions
    - ★ $R \bowtie S = (R_1 \bowtie S_1) \cup (R_2 \bowtie S_2) \cup \cdots \cup (R_k \bowtie S_k)$
- Algorithms:
  - Grace hash join
  - Hybrid hash join

Review: Grace Hash Join $R \bowtie S$

- Consists of three phases:
  1. Partition R into $R_1, \cdots, R_k$
  2. Partition S into $S_1, \cdots, S_k$
  3. Probing phase: probes each R_i with S_i
     - ★ Read R_i to build a hash table
     - ★ Read S_i to probe hash table
- R is called the build relation & S is called the probe relation

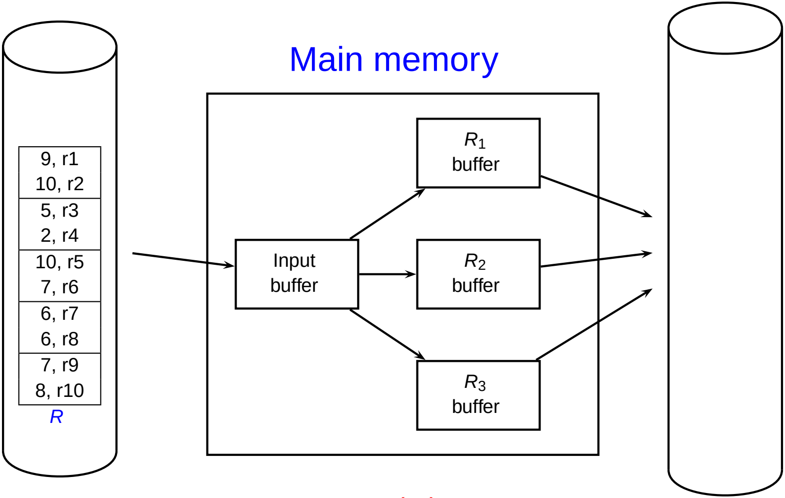
Partitioning (building) phases

initialize a hash table T with k buckets
for each tuple $r \in R$ do
    insert r into bucket $h(r)$ of T
write each bucket R_i of T to disk
initialize a hash table T with k buckets
for each tuple $s \in S$ do
    insert s into bucket $h(s)$ of T
write each bucket S_i of T to disk

Probing (matching) phase

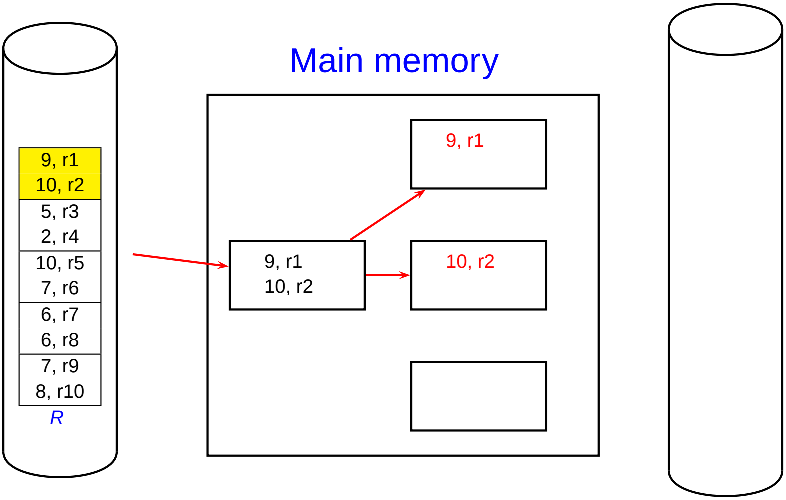
for i = 1 to k do
    initialize a hash table T
    for each tuple r in partition R_i do
        insert r into bucket $h'(r)$ of T
    for each tuple s in partition S_i do
        for each tuple r in bucket $h'(s)$ of T do
            if r and s matches then output (r, s)

Grace Hash Join: Partitioning Relation R

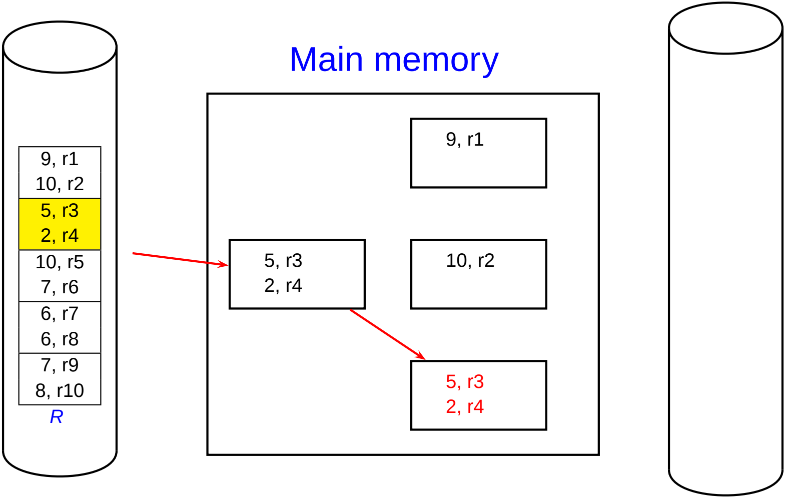


Hash function: $h(v) = v \mod 3$

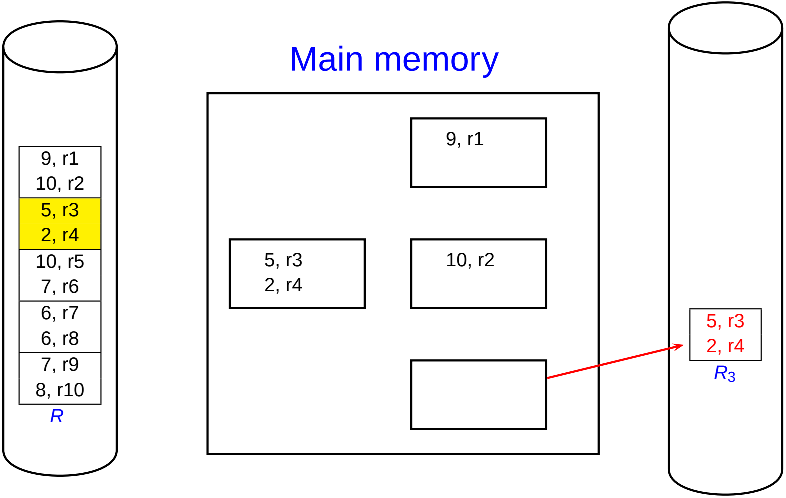
Grace Hash Join: Partitioning Relation R



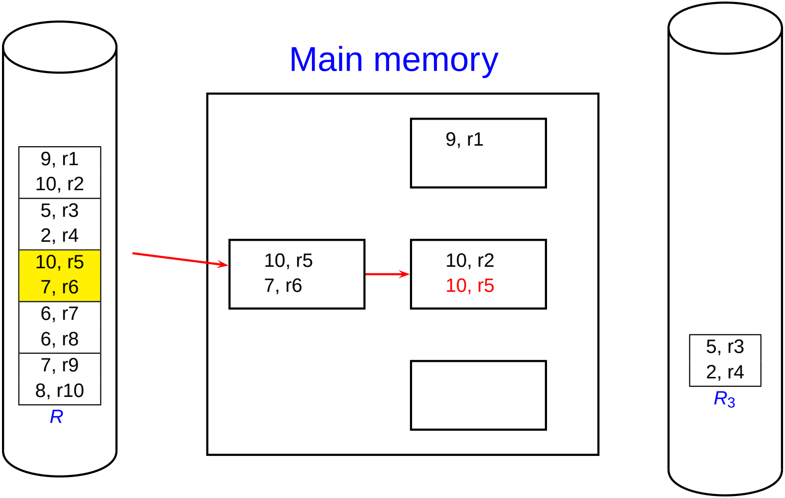
Grace Hash Join: Partitioning Relation R



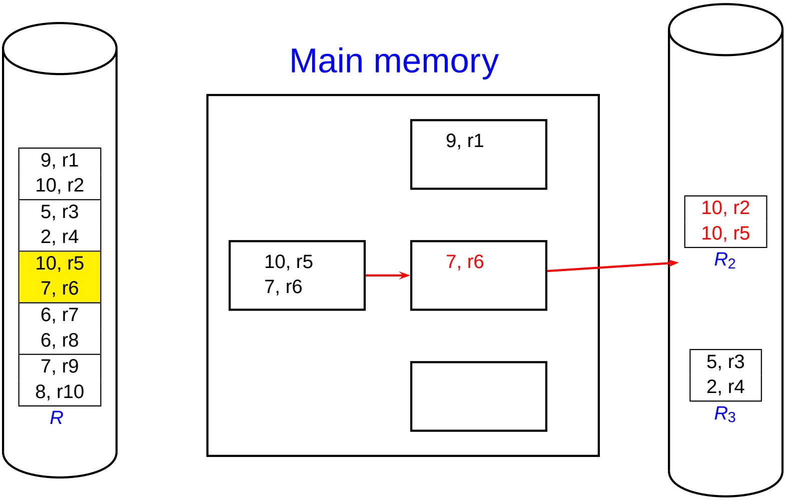
Grace Hash Join: Partitioning Relation R



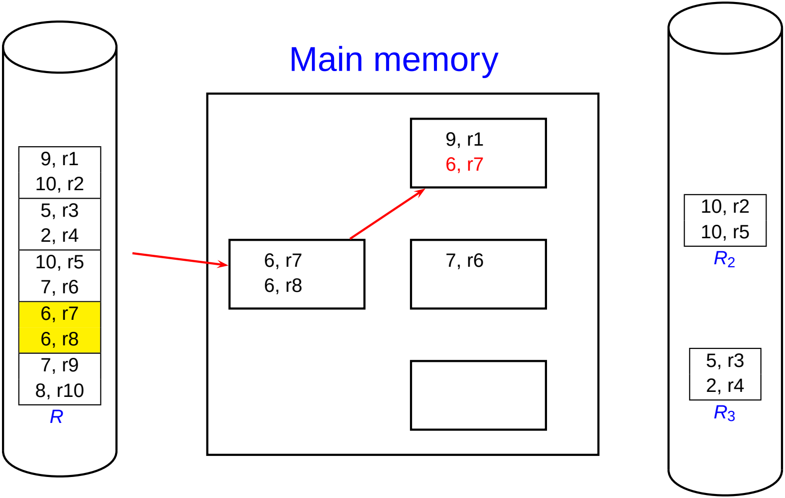
Grace Hash Join: Partitioning Relation R



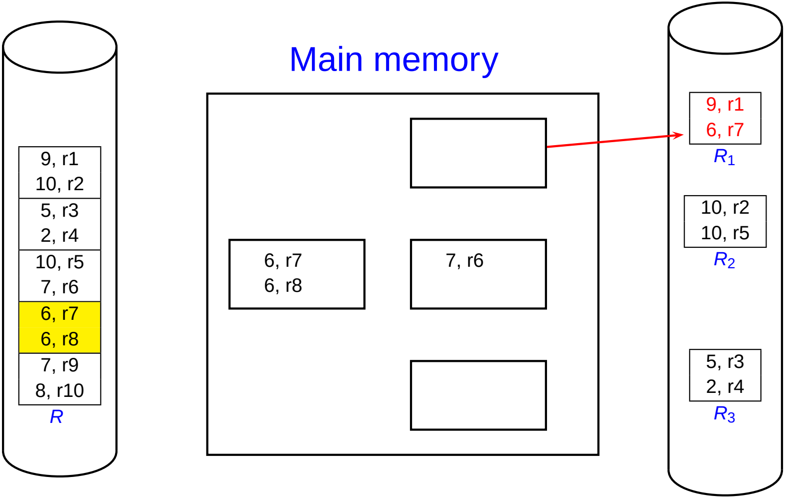
Grace Hash Join: Partitioning Relation R



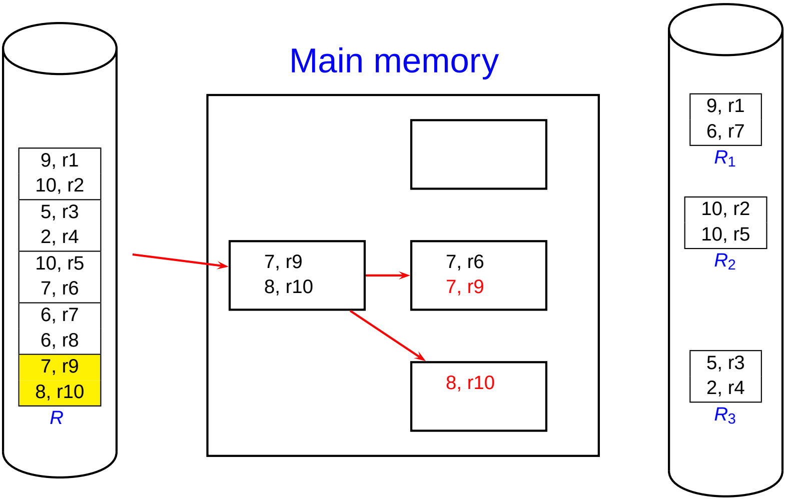
Grace Hash Join: Partitioning Relation R



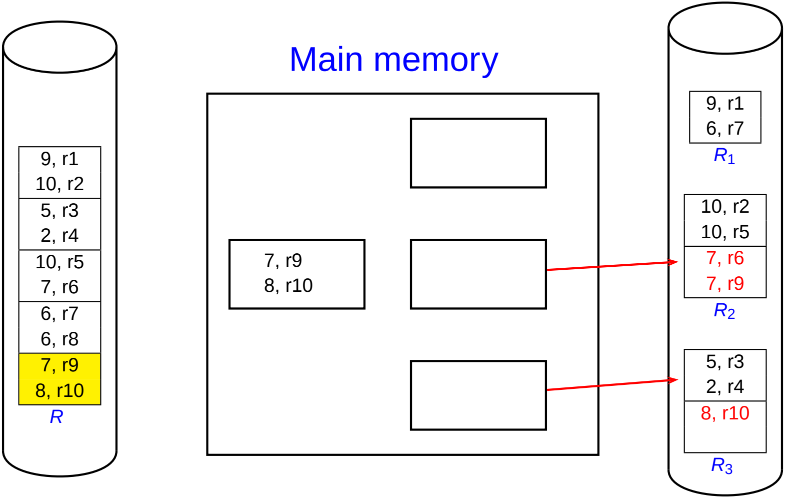
Grace Hash Join: Partitioning Relation R



Grace Hash Join: Partitioning Relation R

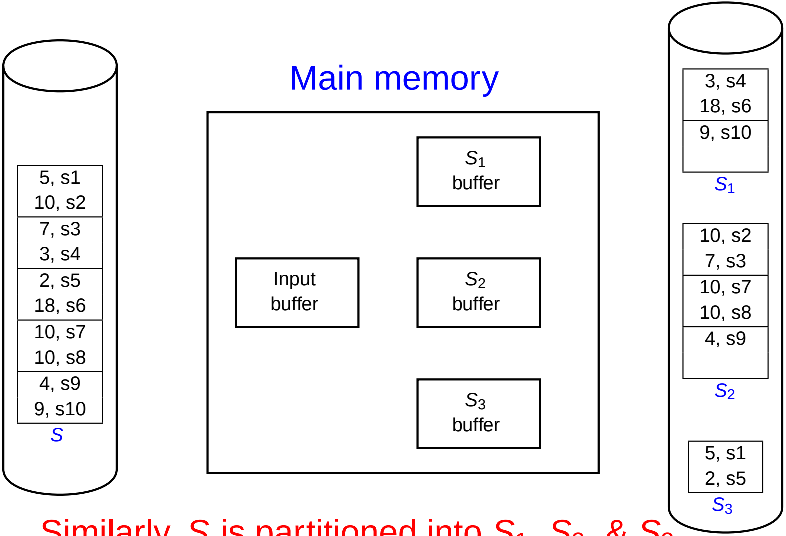


Grace Hash Join: Partitioning Relation R



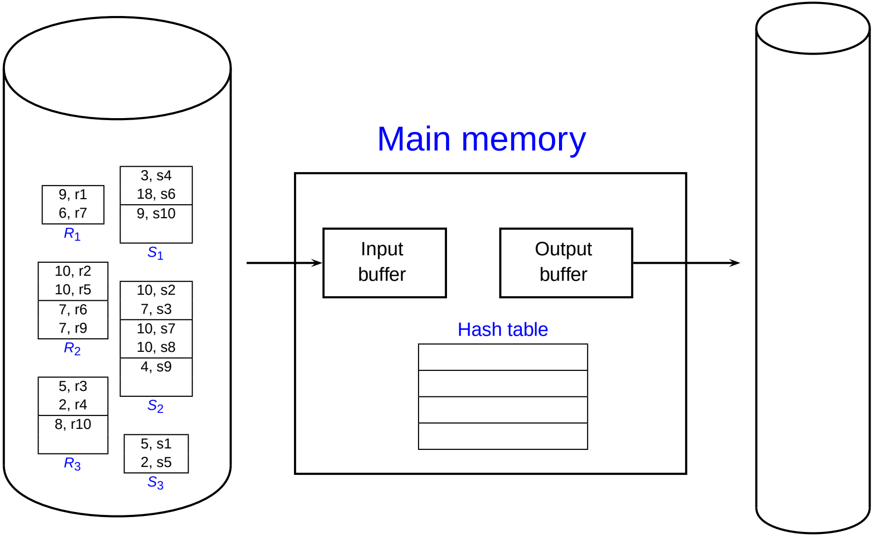
R is partitioned into R_1, R_2, & R_3

Grace Hash Join: Partitioning Relation S

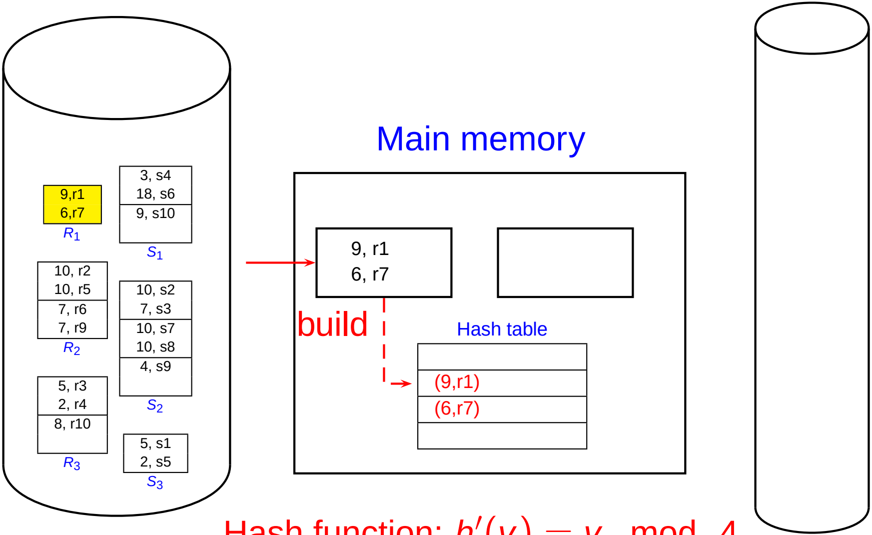


Similarly, S is partitioned into S_1, S_2, & S_3

Grace Hash Join: Probing Phase

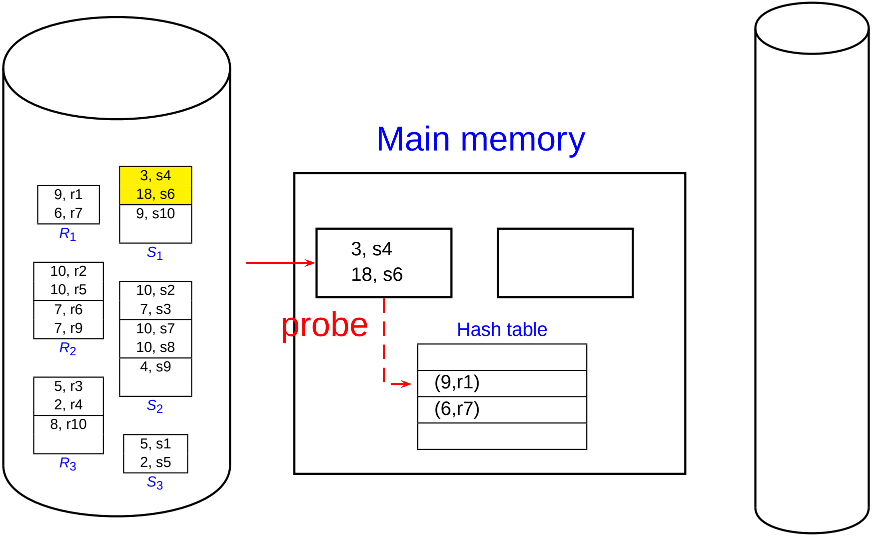


Grace Hash Join: Probing Phase

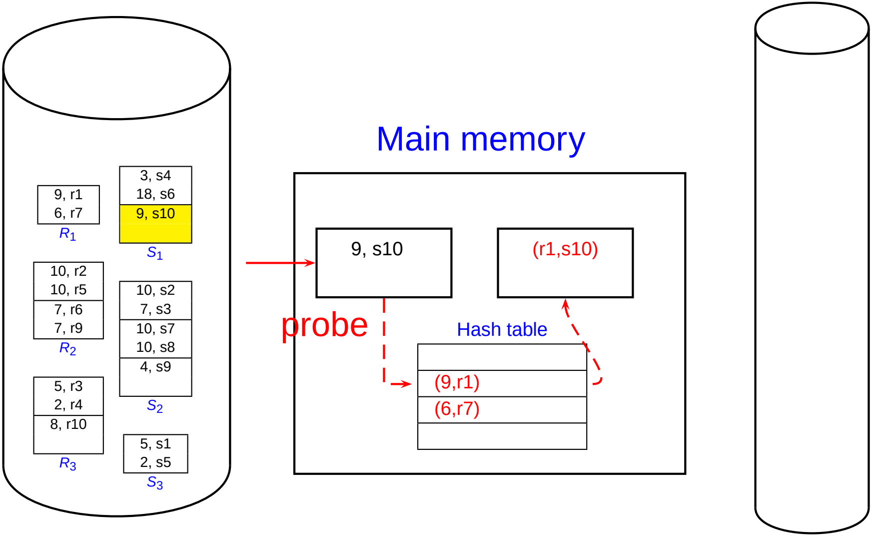


Hash function: $h'(v) = v \mod 4$

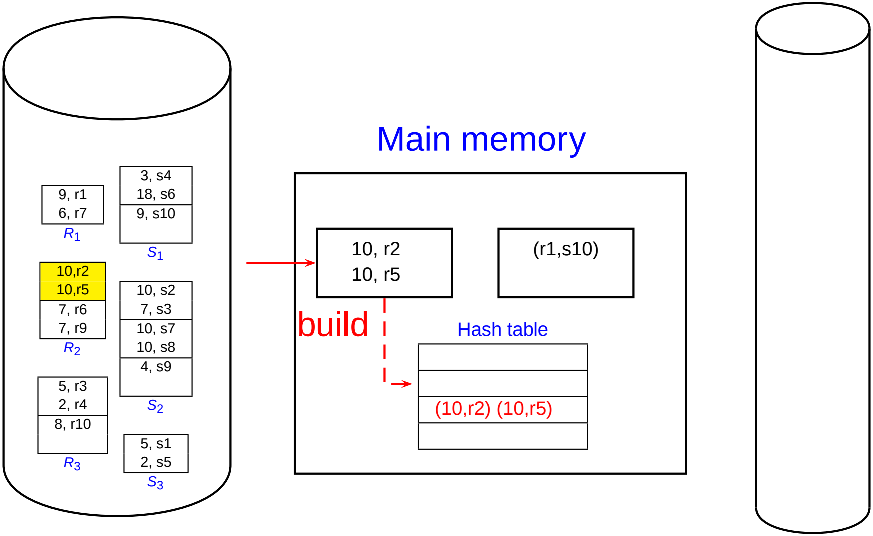
Grace Hash Join: Probing Phase



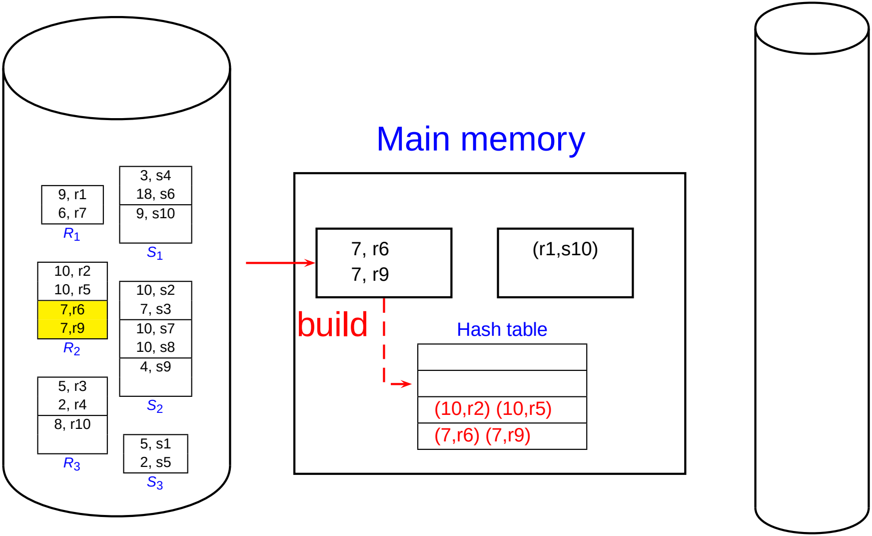
Grace Hash Join: Probing Phase



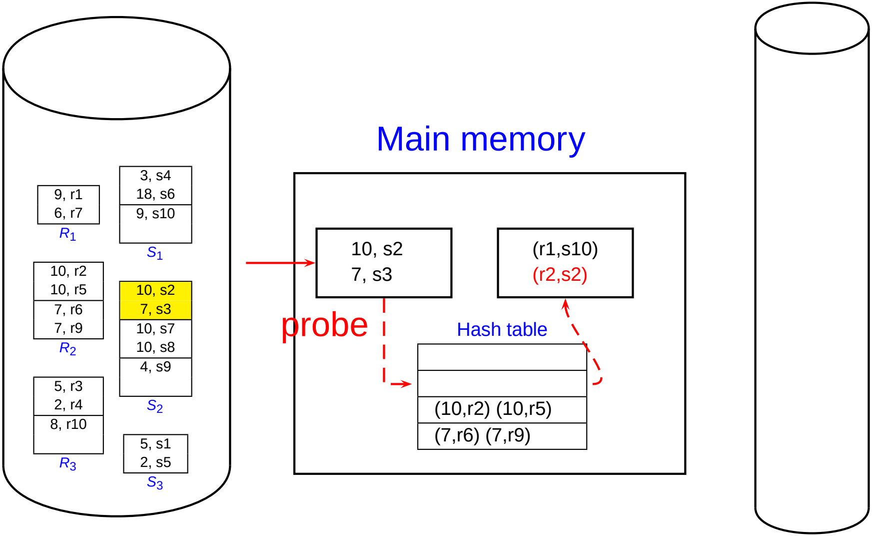
Grace Hash Join: Probing Phase



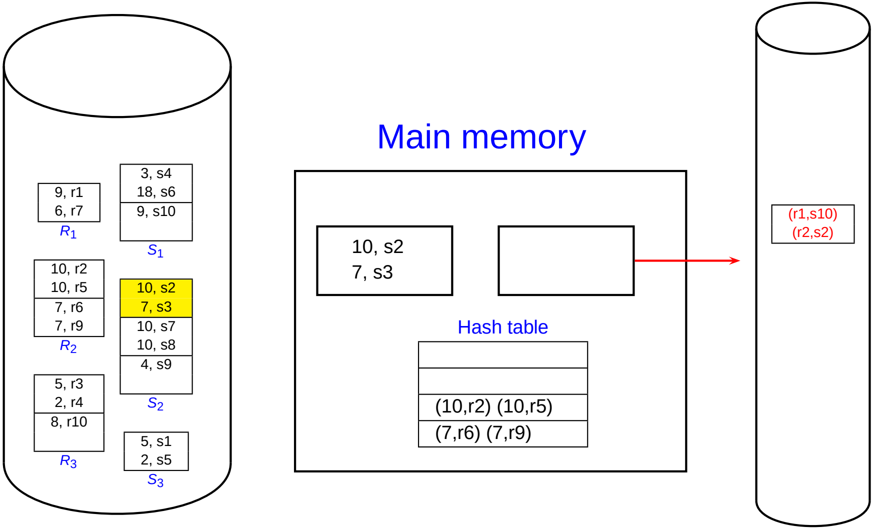
Grace Hash Join: Probing Phase



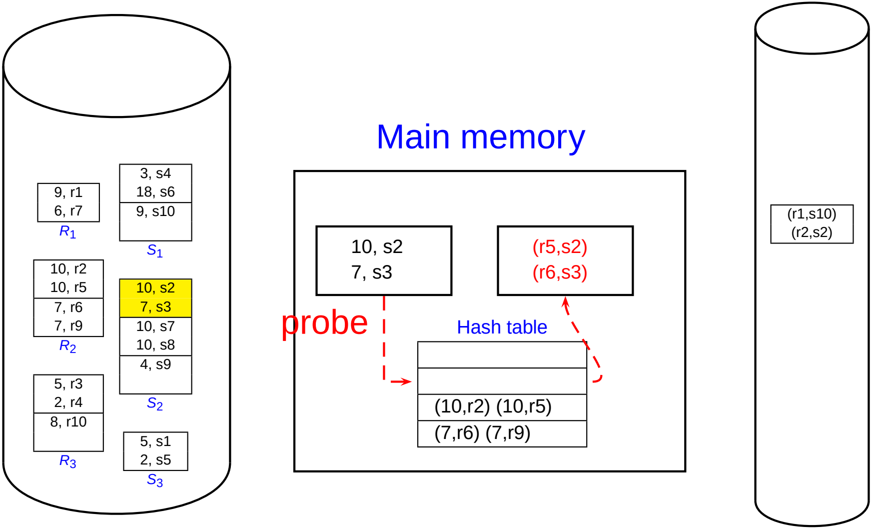
Grace Hash Join: Probing Phase



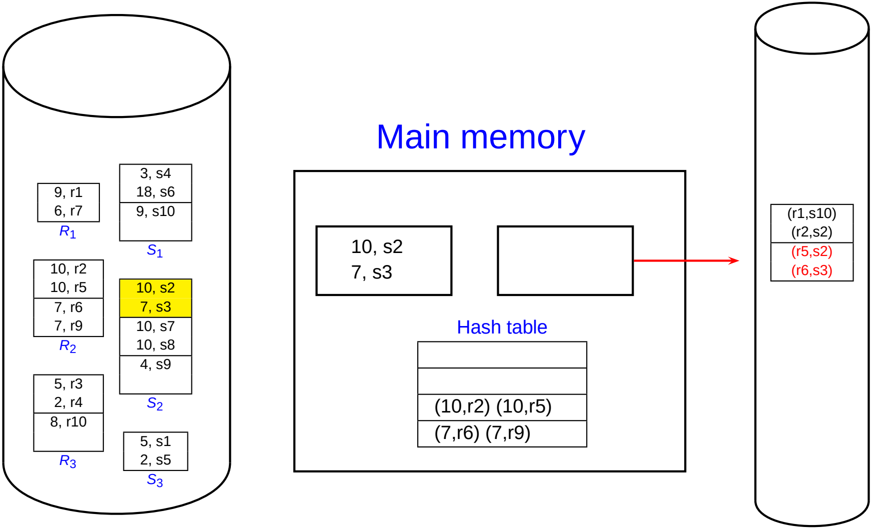
Grace Hash Join: Probing Phase



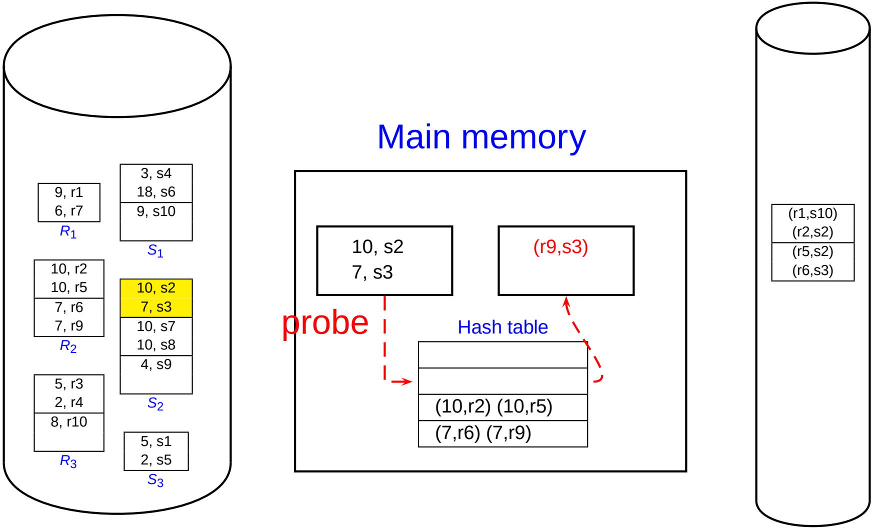
Grace Hash Join: Probing Phase



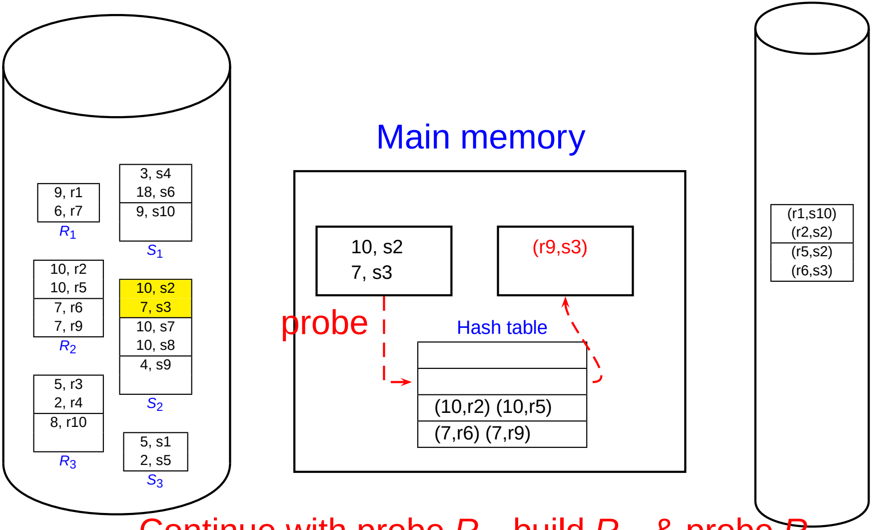
Grace Hash Join: Probing Phase



Grace Hash Join: Probing Phase



Grace Hash Join: Probing Phase



Continue with probe R_2, build R_3, & probe R_3

Review: Hybrid Hash Join (HHJ), $R \bowtie S$

- Improvement of Grace Hash Join
- Enables some build partitions R_i to be resident in memory at the end of partitioning build relation R
  - Each R_i is either a resident or spilled partition
  - Memory usage
    - ★ hash tables for resident partitions
    - ★ output buffers for spilled partitions
    - ★ input buffer
    - ★ free buffer - spool area for pages flushed to disk
- During partitioning of probe table S, S_i can join with R_i if R_i is resident partition
- Number of partitions, k = $\sqrt{F|R|}$
  - $|R|$ = size of build relation R in pages
  - F = fudge factor

HHJ: Partition Build Relation

```
let k = sqrt(F|R|)
for i = 1 to k do
    mark R_i as a resident partition & allocate one page to R_i
allocate input buffer & assign remaining pages to free buffer
for each tuple t ∈ R do
    let t be hashed to R_i
    if (R_i is full) and (R_i is resident) then
        if (there's a free page) then
            add free page to R_i
        else
            mark R_i as spilled
            flush R_i to disk & free all pages (except one) of R_i
    else if (R_i is full) and (R_i is spilled) then
        flush R_i to disk
    insert t into R_i
flush & free all pages in spilled partitions
```

HHJ: Partition Probe Relation

```
for i = 1 to k do
    if (R_i is spilled) then
        allocate one page to S_i
for each tuple t ∈ S do
    let t be hashed to S_i
    if (R_i is resident) then
        use t to probe R_i for matches
    else
        insert t into S_i
        if (S_i is full) then
            flush S_i to disk
flush all pages in S partitions & free all pages
```

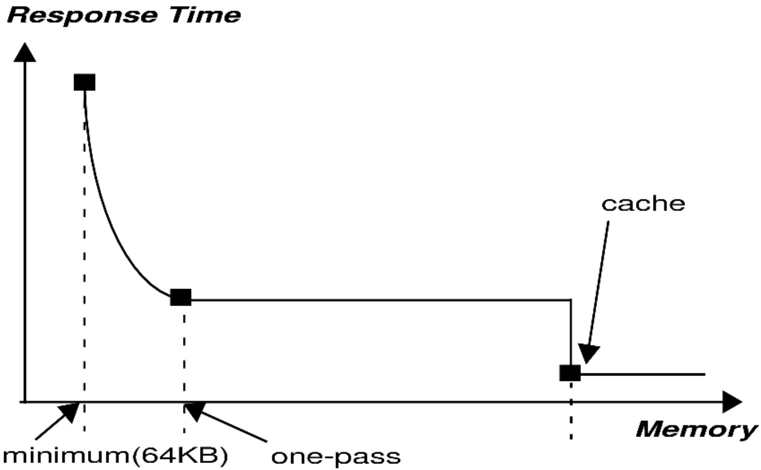
HHJ: Probing Phase

```
for i = 1 to k do
    if (R_i is spilled) then
        read in R_i & build a hash table
        for each tuple t ∈ S_i do
            use t to probe R_i for matches
```

Oracle PGA Memory Management

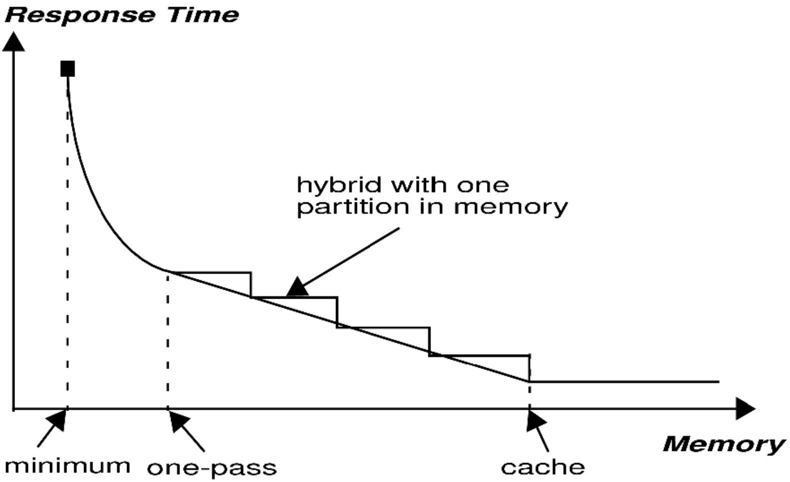
- **operator work area** = memory allocated for operator's execution
- Oracle's classification of work area size:
  - cache size = minimum work area size that enables evaluation to incur a single scan of input data
    - ★ cache size = size of input data + auxiliary memory structures
  - one-pass size = minimum work area size that enables evaluation to incur 2 scans of input data
  - multiple-pass size = work area size that is less than one-pass size
- cache size $>$ one-pass size $>$ multiple-pass size

Effect of Memory on Sort Performance



(Dageville & Zait, VLDB 2002)

Effect of Memory on Hash Join Perf.



(Dageville & Zait, VLDB 2002)

Memory Management Approaches

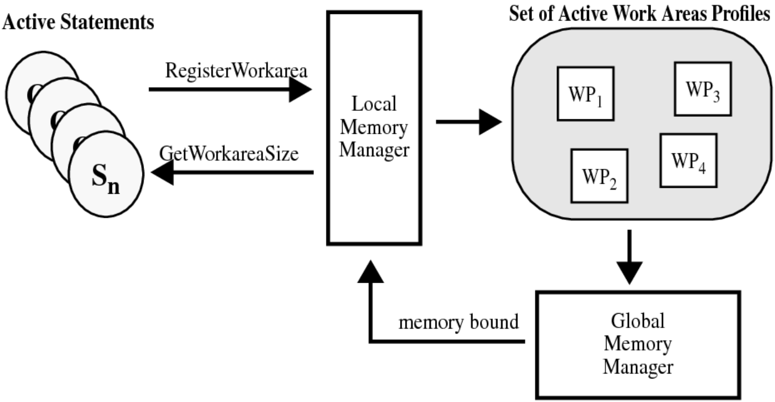
How much memory to allocate to SQL operators?

- Fixed amount of memory for each operator
- Amount based on size of input operands
- Amount based on size of input operands + current workload
- Memory usage adapts to memory demand of system

DBMS Approaches (circa 2002)

DBMS	Initial work area size	Size during execution	Response to memory pressure
Oracle 8i	static	static	none
Oracle 9i	**dynamic**	**adaptive**	**adaptive**
DB2/UDB 7.1	static	static	minimum
Informix 9.3	static	static	limit ops
SQL Server 7	dynamic	static	queueing
Teradata	dynamic	static	?

Memory Management Feedback Loop



(Dageville & Zait, VLDB 2002)

Work Area Profile

Work area profile = metadata for an operator work area:

- Operator type
  - Example: sort, hash-join
- Current memory requirement to run with minimum/one-pass/cache memory
- Number instances of work area
  - ie degree of parallelism of operator
- Current amount of PGA memory

Memory Management Feedback Loop

SQL Operator

1. Registers work area profile with local memory manager
5. Adjusts memory usage based on expected work area size (derived by LMM) & updates work area profile

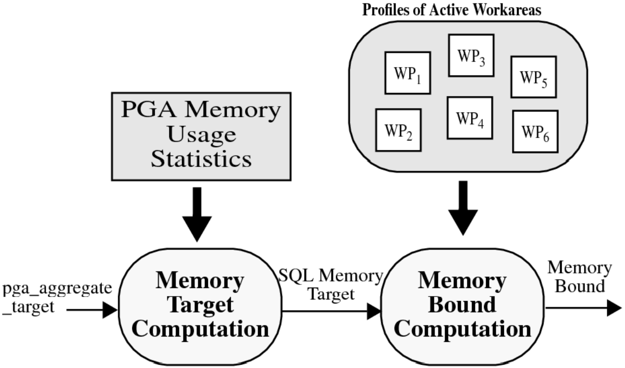
Local Memory Manager (LMM)

2. Maintains active work area profiles in SGA
4. Computes expected work area size for each operator based on
   - operator's work area profile
   - memory bound (derived by GMM)

Global Memory Manager (GMM)

3. Computes memory bound for work area based on
   - `pga_aggregated_target`
   - PGA memory usage statistics
   - active work area profiles

GMM: Memory Bound Computation



(Dageville & Zait, VLDB 2002)

- **GMM** computes memory bound for work area based on
  - `pga_aggregated_target`
  - PGA memory usage statistics
  - active work area profiles
- SQL memory target = amount of PGA allocated to work areas

Memory Bound Computation

- Find largest value of B such that

$$\sum_{i=1}^{N} ExpectedWorkAreaSize(WP_i, B) \leq SQL\ Memory\ Target$$

- N = number of work area profiles
- WP_i = memory profile of i^{th} work area, $i \in [1, N]$
- Work area profile
  - minimum, one-pass & cache memory thresholds of work area

Expected Work Area Size, E

- Computation of E depends on type of operator (sort or non-sort) & execution mode (serial or parallel)
- $E \in$ [minimum threshold, cache threshold]
- **Type of operator**:
  - Operator is sort

$$E = \begin{cases} \text{cache threshold} & \text{if } B \geq \text{ cache threshold,} \\ \text{one-pass threshold} & \text{if } B \geq \text{ one-pass threshold,} \\ \text{multi-pass threshold} & \text{otherwise.} \end{cases}$$

  - Operator is non-sort

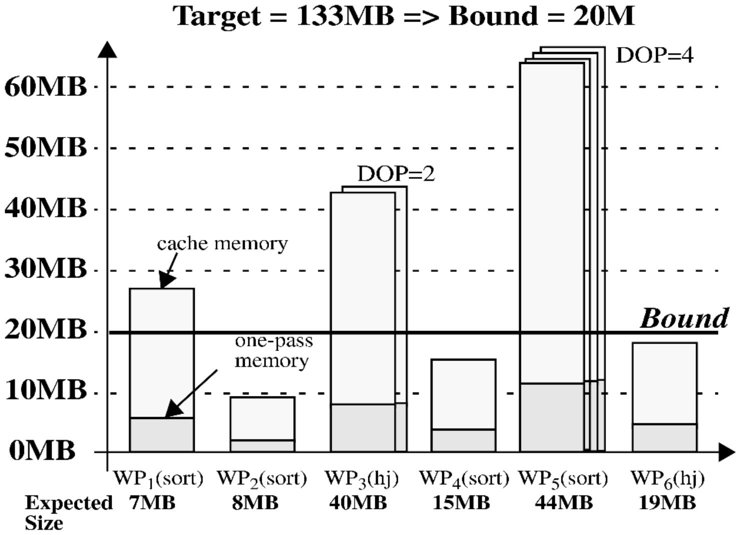
$$E = \min\{B, \text{ cache threshold}\}$$

- **Execution mode**:

$$E = \begin{cases} \min\{E, \ 0.05 \times \text{memory target}\} & \text{if serial,} \\ \min\{E \times DOP, \ 0.3 \times \text{memory target}\} & \text{otherwise.} \end{cases}$$

Example of Memory Bound Computation

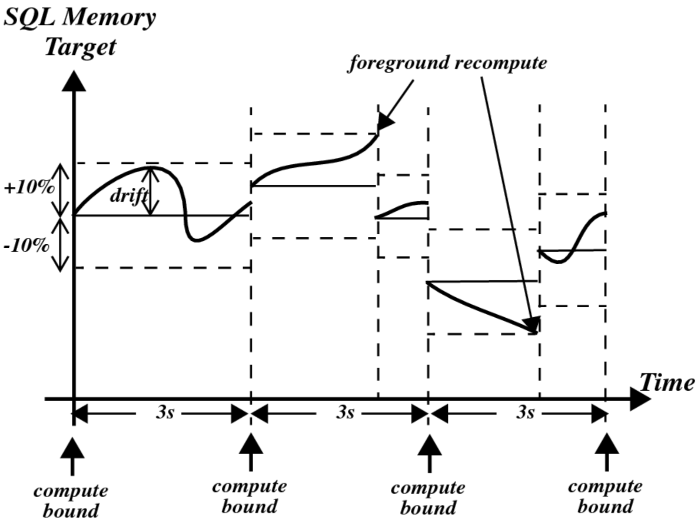
(Dageville & Zait, VLDB 2002)



Drift Management

- **GMM** is a background daemon
  - Recomputes memory bound periodically
- Limitation of background recomputation:
  - Slow to react to workload changes
- Memory drift
  - Measures staleness of memory bound
  - Expected amount of memory acquired/released by queries since last recomputation of memory bound
  - Drift can be positive or negative
- Triggers foreground recomputation when drift exceeds 10% of SQL memory target

Foreground Computation of Bound



(Dageville & Zait, VLDB 2002)

Memory Adaptive External Sorting

- **Creation of initial sorted runs**
  - Adapt size of sorted runs
- **Merging sorted runs**
  - Adapt merging fan-out
- Adapt size of input/output buffers

Memory Adaptive Hash Join

- **Partitioning of build relation**
  - Adapt size of free buffer
  - Convert resident partitions to spilled ones to release memory
- **Partitioning of probe relation**
  - Adapt size of free buffer
  - Convert resident partitions to spilled ones to release memory
  - Restore spilled partitions to resident ones using extra memory
- Adapt size of input/output buffers

References

Required Readings

- B. Dageville, M. Zait, SQL Memory Management in Oracle9i, VLDB 2002, 962-973.

Additional Readings

- W. Zhang, P.A. Larson, *Dynamic Memory Adjustment for External Mergesort*, VLDB 1997.
- H.H. Pang, M.J. Carey, M. Livny, *Partially preemptible hash joins*, SIGMOD 1993.
- A.J. Storm, C. Garcia-Arellano, S.S. Lightstone, Y. Diao, M. Surendra, *Adaptive self-tuning memory in DB2*, VLDB 2006.