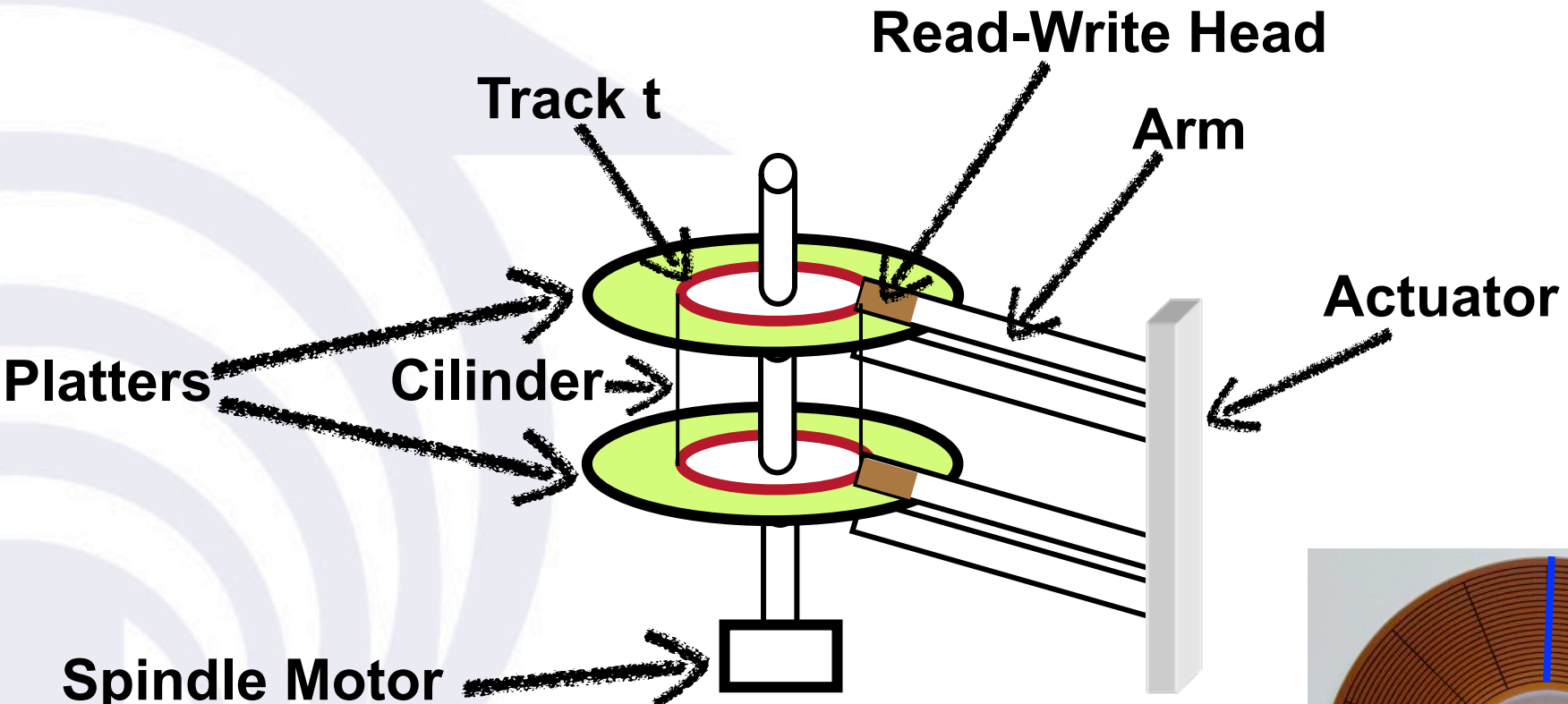
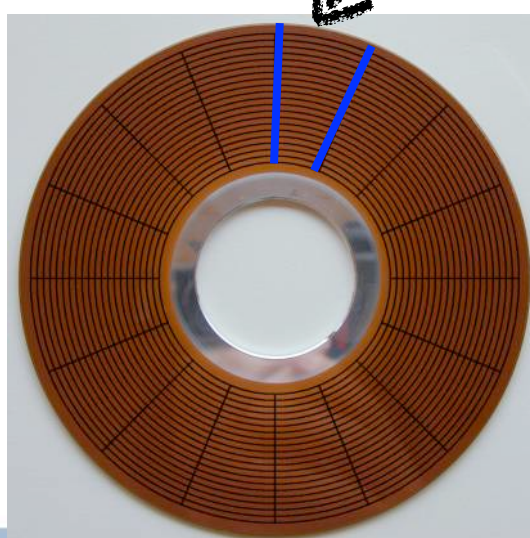
Solid-State Drives: How Do They Change the DBMS Game ?

Angelo Brayner
Universidade de Fortaleza, Brasil

Mario A. Nascimento
University of Alberta, Canada

Anatomy of an HD Drive





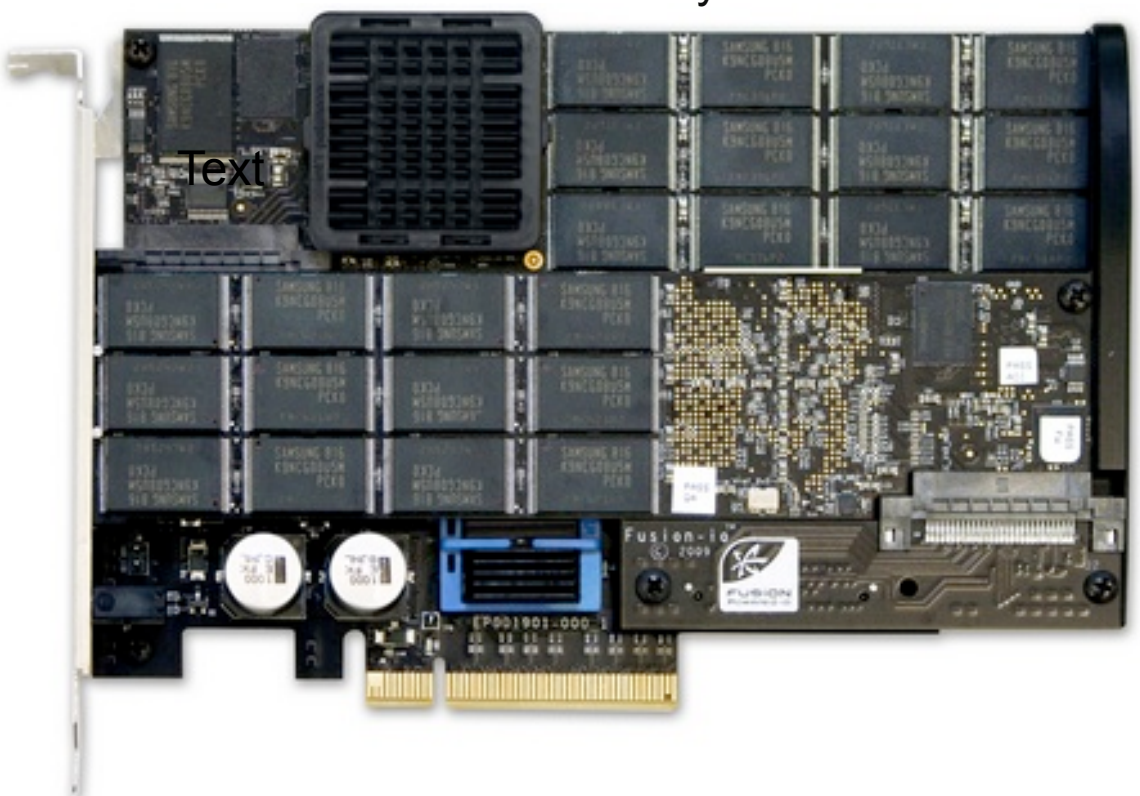
Time to access a sector:

$T_A = T_S + T_R + T_T$

Solid-State Drive

- Storage device made of silicon chips instead of spinning metal platters
  - No mechanical part

Fusion-io's ioMemory Module of 1.28TB



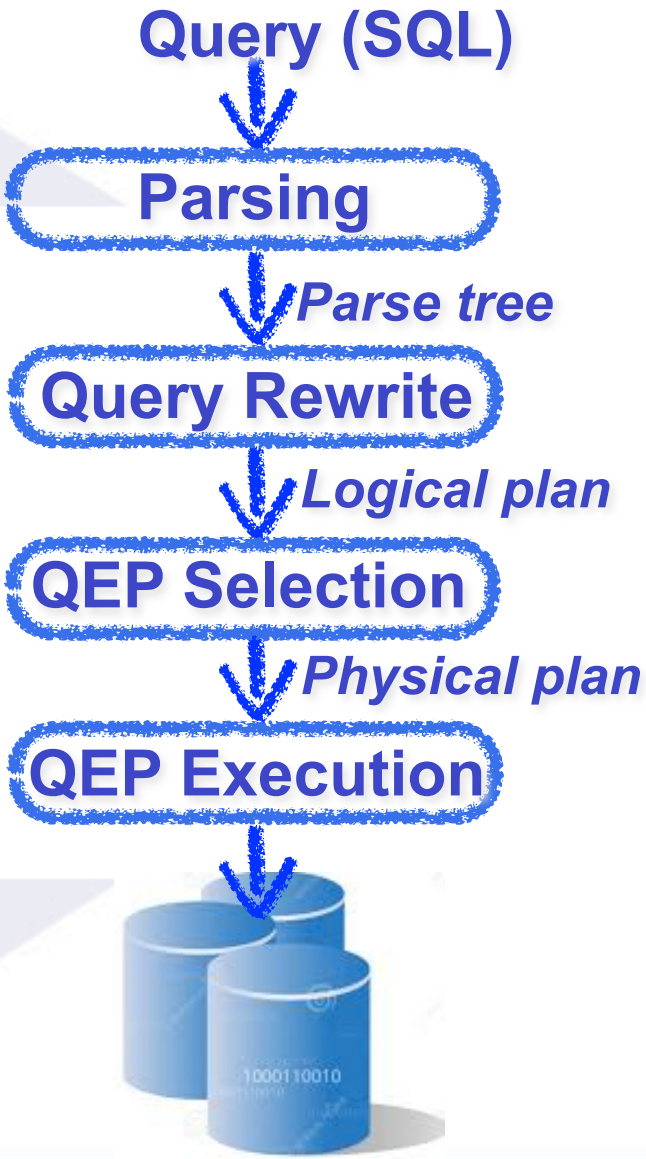
Solid-State Drive

- Flash
  - Computer chip which can be electrically reprogrammed and erased
  - Stores data in a floating-gate transistor (cell)
    - Array of cells
    - Data are represented by the voltage level in a cell
      - High voltage (>5v) - 1
        - Default state
      - Low voltage - 0
  - Block/page addressable
    - Read on a page
    - Write on a block

Outline - Part 2

- Revisiting Fundamental DBMS techniques and Algorithms
  - Indexing
  - Query Optimization
  - Join Algorithms
  - Caching
  - Logging

Query Processing



Query Processing

Query Optimization

- Query Rewrite
  - Several relational algebra expressions for a given SQL query
  - Operator graph
    - Logical query execution plan
  - Transformation rules
    - $\sigma_{\theta_1 \wedge \theta_2}(r \bowtie_{\theta_3} s) \Leftrightarrow \sigma_{\theta_1}(r) \bowtie_{\theta_3} \sigma_{\theta_2}(s)$
  - Heuristics
    - To push down selective operations
      - Selection and projection

Query Optimization

- Query Execution Plan Selection
  - Criterion
    - The lowest execution cost
      - Execution cost is approximated well by **# disk I/O's**
  - Algebraic operators in logical plan are substituted by physical operators
    - Join: nested-loop, merge, hash join
  - Each physical operator has a different execution cost
    - Nested-loop join: $P_r + n_r \cdot P_s$
    - Merge join: $P_r + P_s$
    - Hash join: $3(P_r + P_s)$

Query Optimization

- Query Execution Plan Selection
  - Join ordering problem
    - n-way joins executed as a sequence of 2-way joins
      - The best order to execute 2-way joins
    - The number of possible joins for n tables
      - 2n!/(n+1)!n!
    - Dynamic programming
      - $O(n \cdot 2^{(n-1)})$
      - Heuristics
        - Left deep tree
        - **Join selectivity**
    - Example

Query Optimization

- Query Execution Plan Execution
  - Stop-and-go way
    - Physical operators
  - Pipelining x Materialization (temporary results)

SSD-aware Query Optimization

- To be or not to be??
- Where to be SSD-aware??
  - QEP Selection
  - QEP Execution

SSD-aware Query Optimization

- Where to be SSD-aware??
  - QEP Selection
    - Cost model
      - Higher weight for write operations
    - Physical operators - Join
      - For reducing the # of write operations
    - Join ordering problem
      - Heuristics
        - Selectivity
        - Amount of write operations
  - QEP Execution
    - Pipelining execution of physical operators

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i = f(t_r[JoinAttrib])$
$PR_i = PR_i \cup \{t_r\}$

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i=f(t_r[JoinAttrib])$
$PR_i=PR_i \cup \{t_r\}$

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i = f(t_r[JoinAttrib])$
$PR_i = PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i = f(t_s[JoinAttrib])$
$PS_i = PS_i \cup \{t_s\}$

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i=f(t_r[JoinAttrib])$
$PR_i=PR_i \cup \{t_r\}$

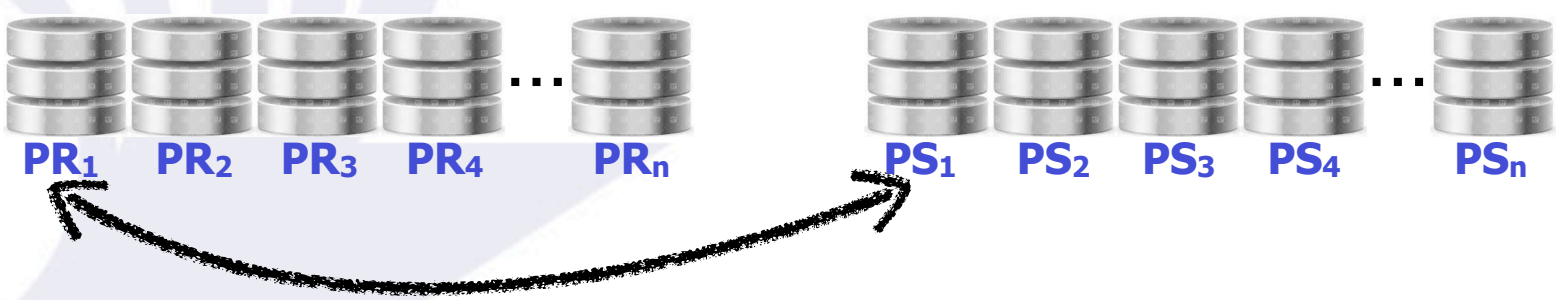
For each tuple $t_s \in S$ do
$i=f(t_s[JoinAttrib])$
$PS_i=PS_i \cup \{t_s\}$

Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i=f(t_r[JoinAttrib])$
$PR_i=PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i=f(t_s[JoinAttrib])$
$PS_i=PS_i \cup \{t_s\}$

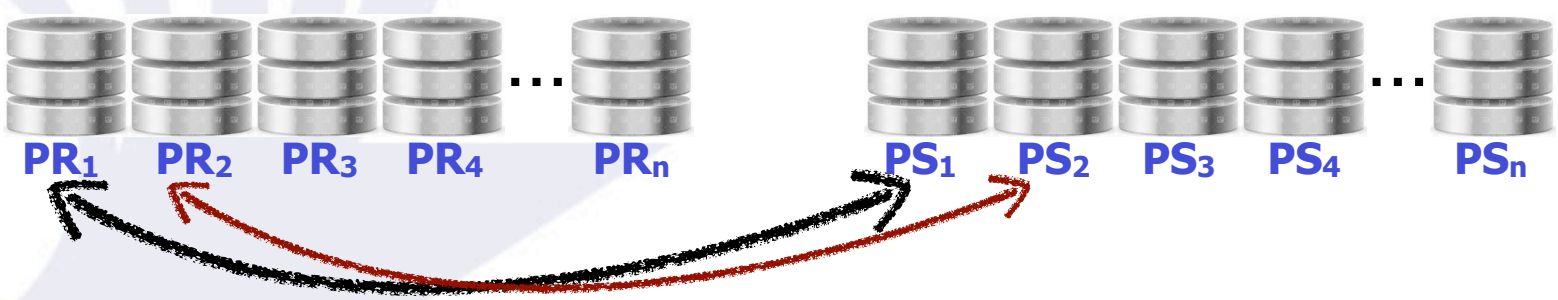


Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i = f(t_r[JoinAttrib])$
$PR_i = PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i = f(t_s[JoinAttrib])$
$PS_i = PS_i \cup \{t_s\}$

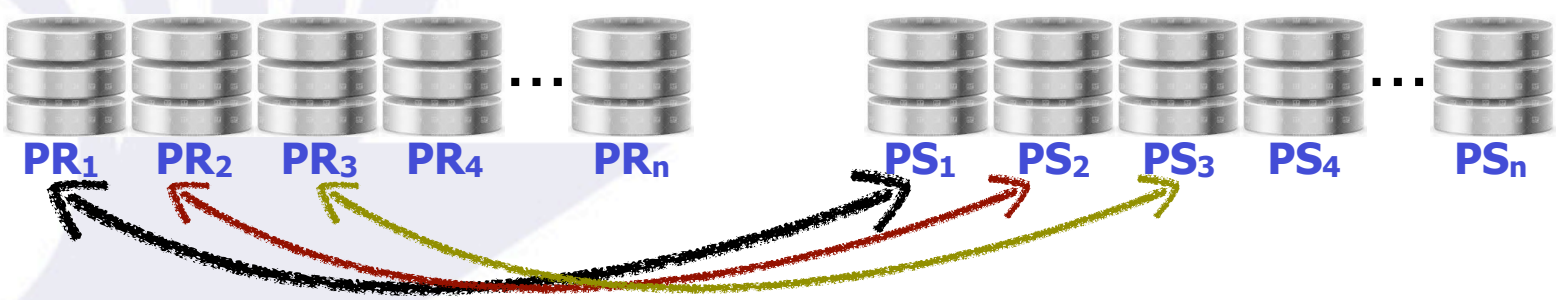


Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i = f(t_r[JoinAttrib])$
$PR_i = PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i = f(t_s[JoinAttrib])$
$PS_i = PS_i \cup \{t_s\}$

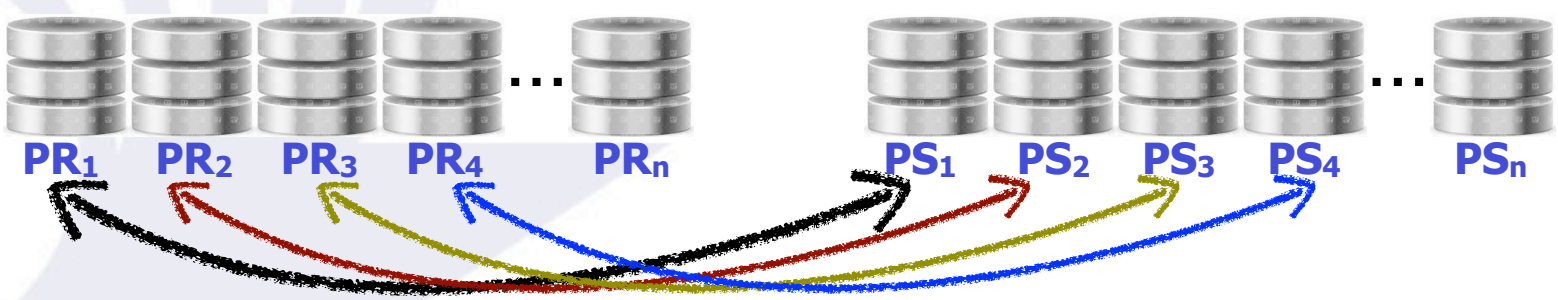


Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i=f(t_r[JoinAttrib])$
$PR_i=PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i=f(t_s[JoinAttrib])$
$PS_i=PS_i \cup \{t_s\}$

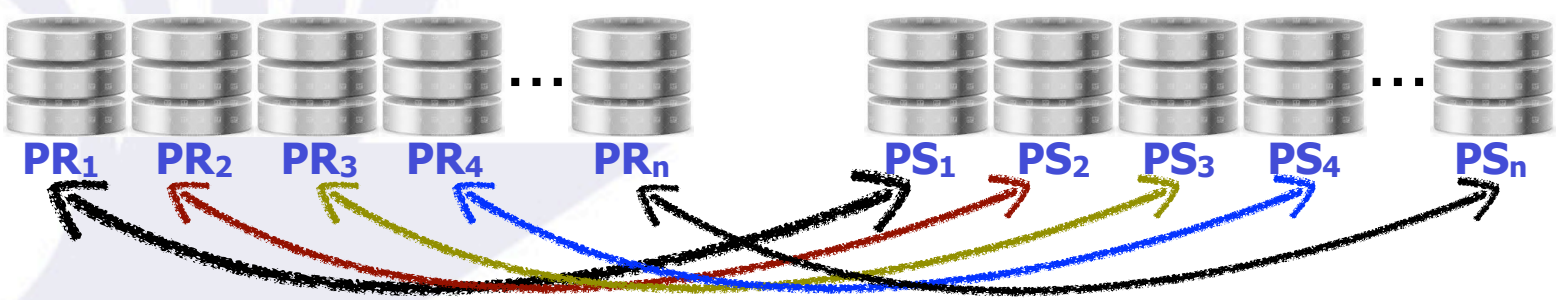


Join Algorithms

- Grace Hash Join (R⋈S)
  - 1st Phase
    - To Partition R and S applying a hash function f

For each tuple $t_r \in R$ do
$i=f(t_r[JoinAttrib])$
$PR_i=PR_i \cup \{t_r\}$

For each tuple $t_s \in S$ do
$i=f(t_s[JoinAttrib])$
$PS_i=PS_i \cup \{t_s\}$

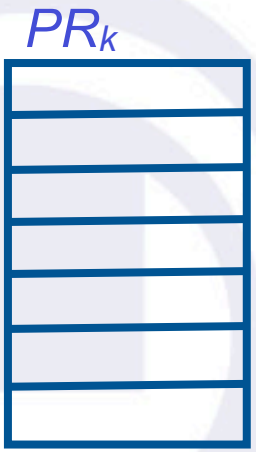


Join Algorithms

- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address

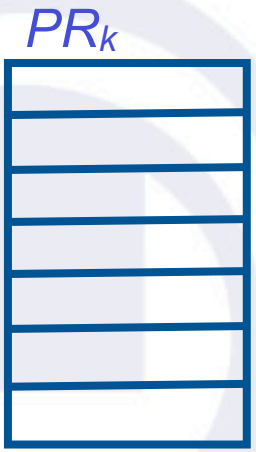
Join Algorithms

- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



Join Algorithms

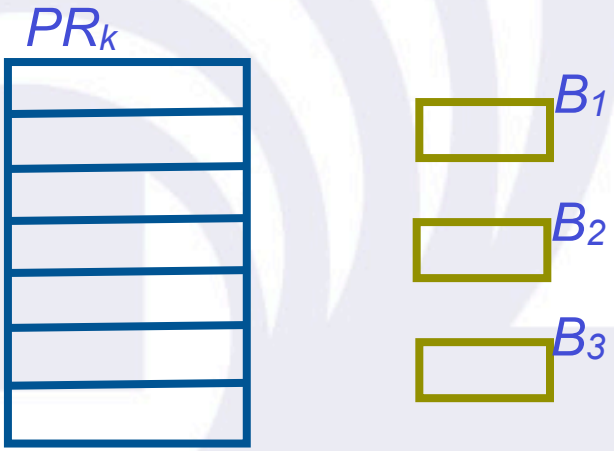
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

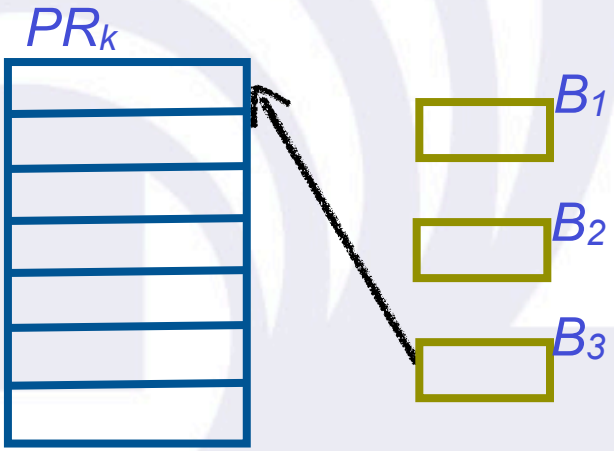
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i = h(t_r[JoinAttrib])$
$B_i = B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

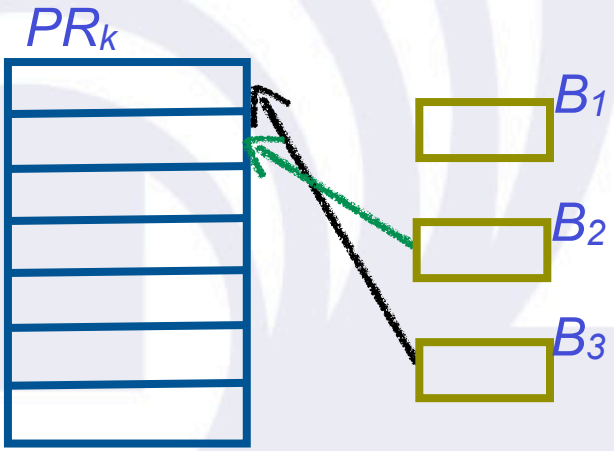
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

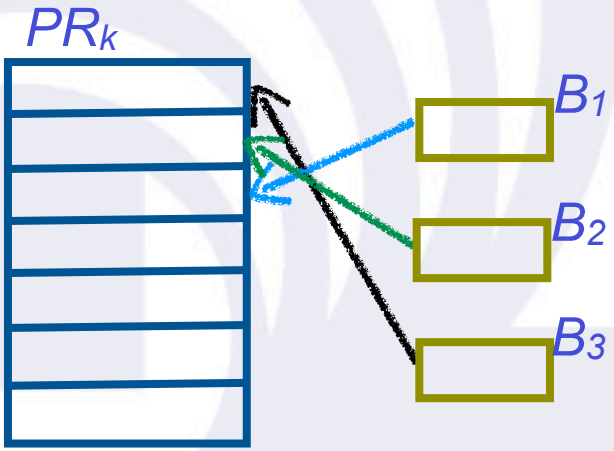
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

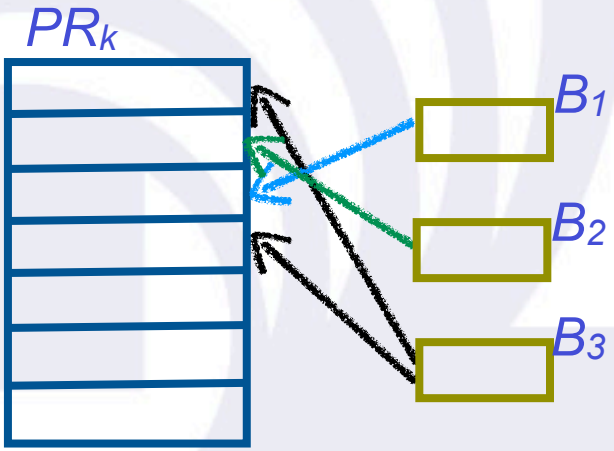
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

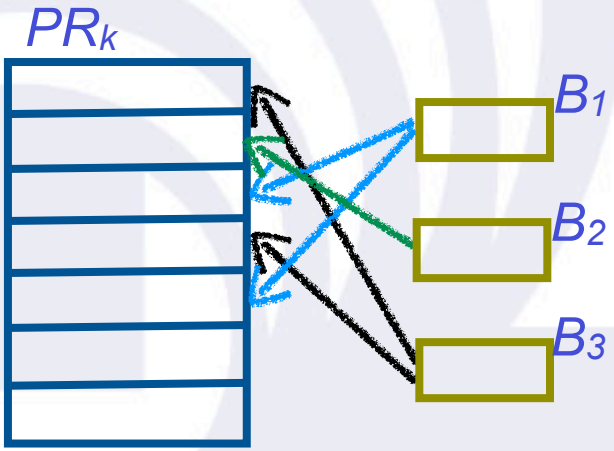
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

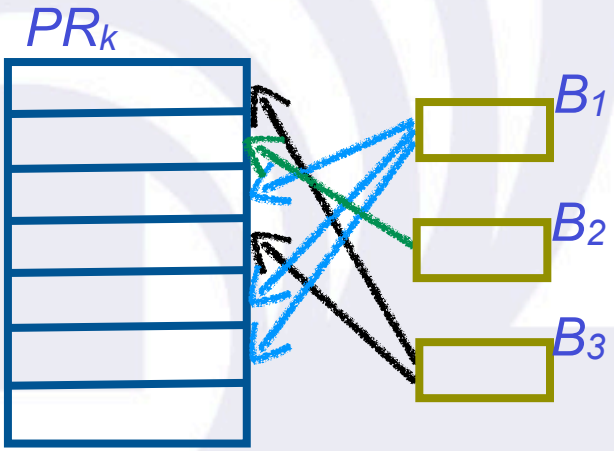
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

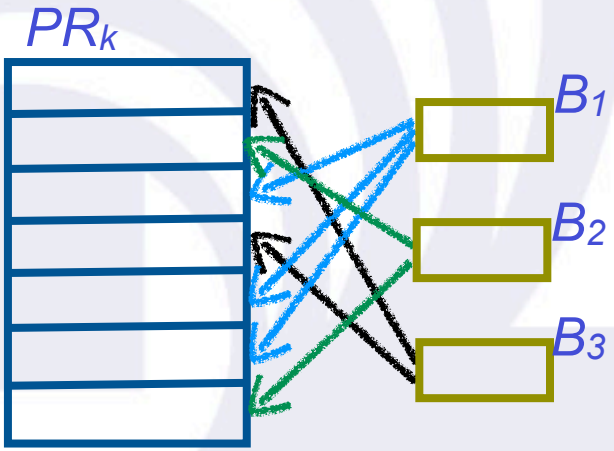
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

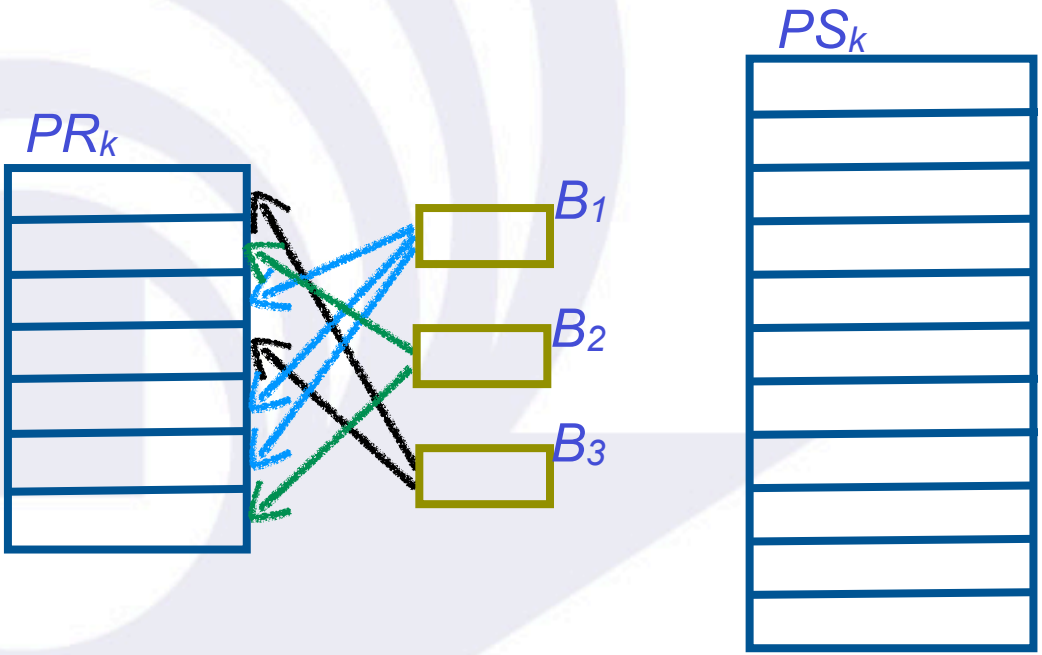
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

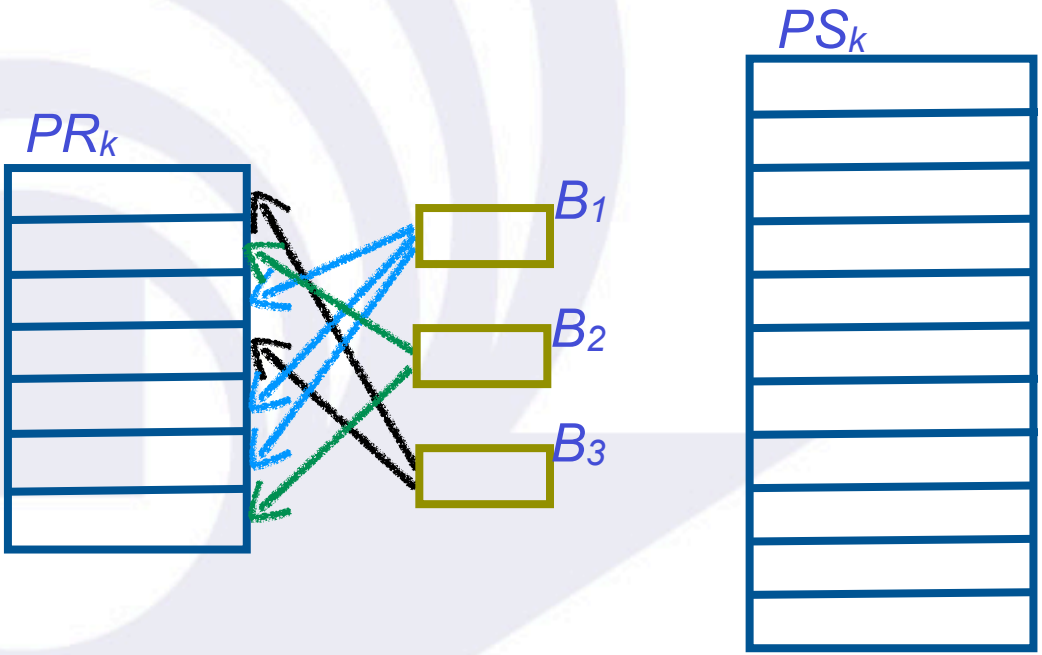
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;

Join Algorithms

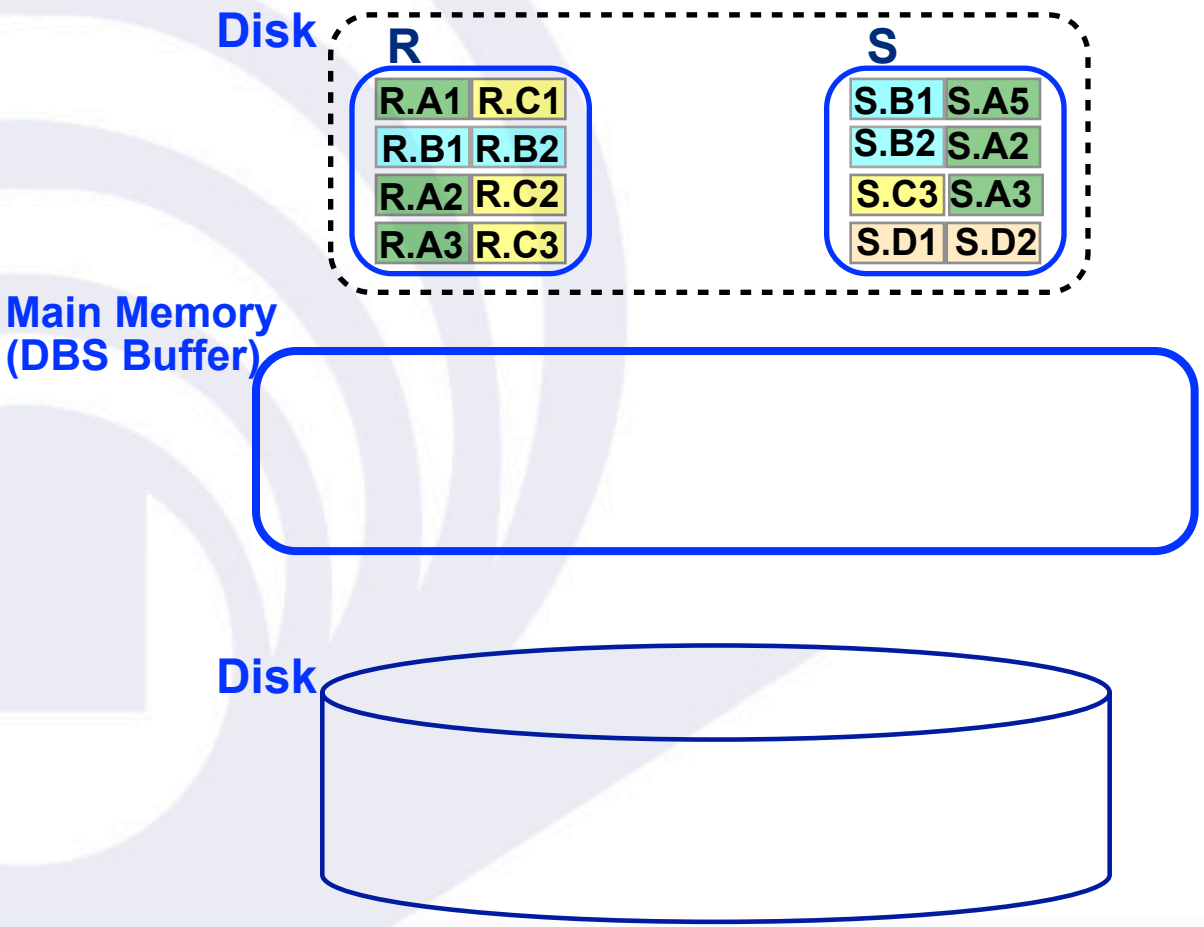
- Grace Hash Join (R⋈S)
  - 2nd Phase
    - Executes an index nested-loop join on each pair of partition with same hash address



For k=0 to max
For each tuple $t_r \in PR_k$ do
$i=h(t_r[JoinAttrib])$
$B_i=B_i \cup \{(t_r[JoinAttrib], Pointer)\}$
end;
For each tuple $t_s \in PS_k$ do
$i=h(t_s[JoinAttrib])$
case $t_s[JoinAttrib] \in B_i$
include t_r, t_s to result
end;
end;

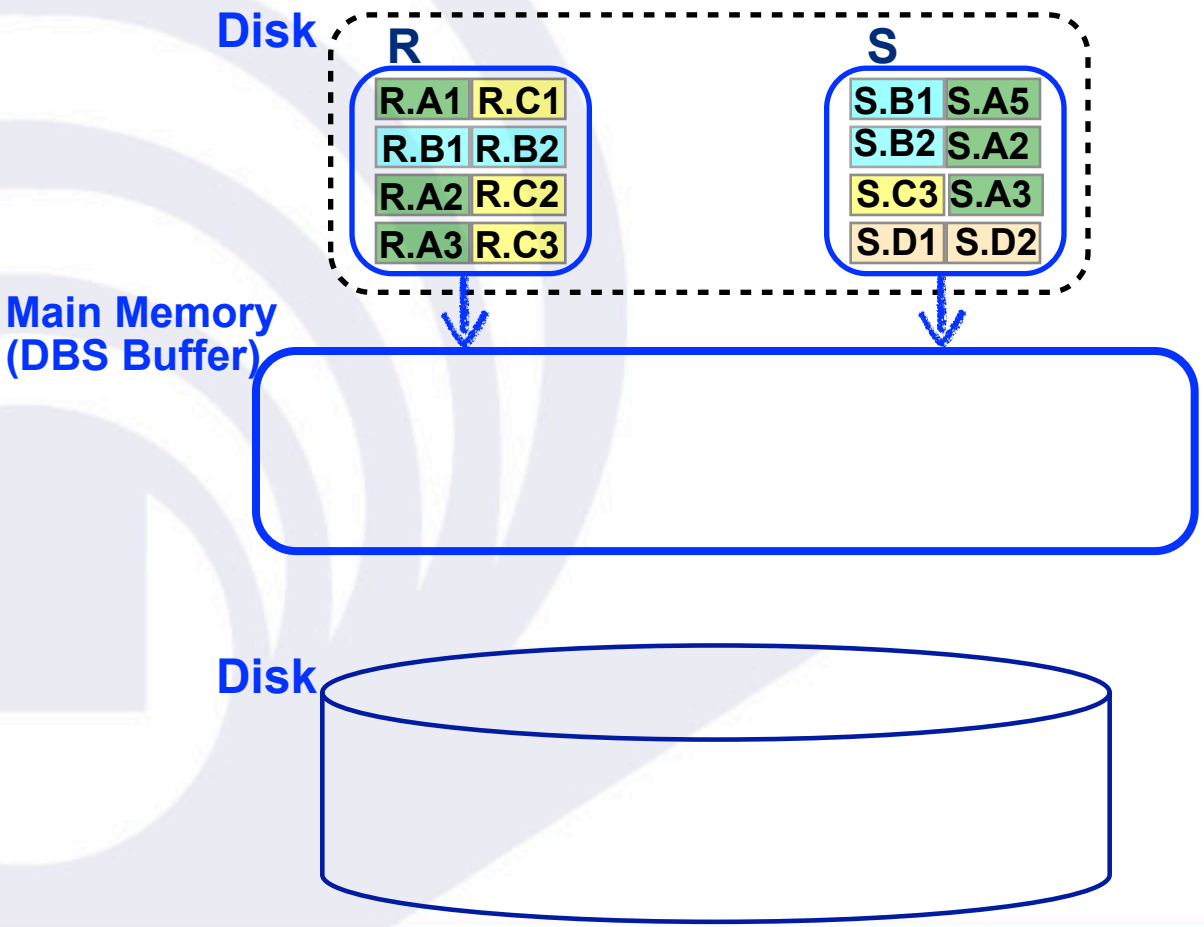
Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



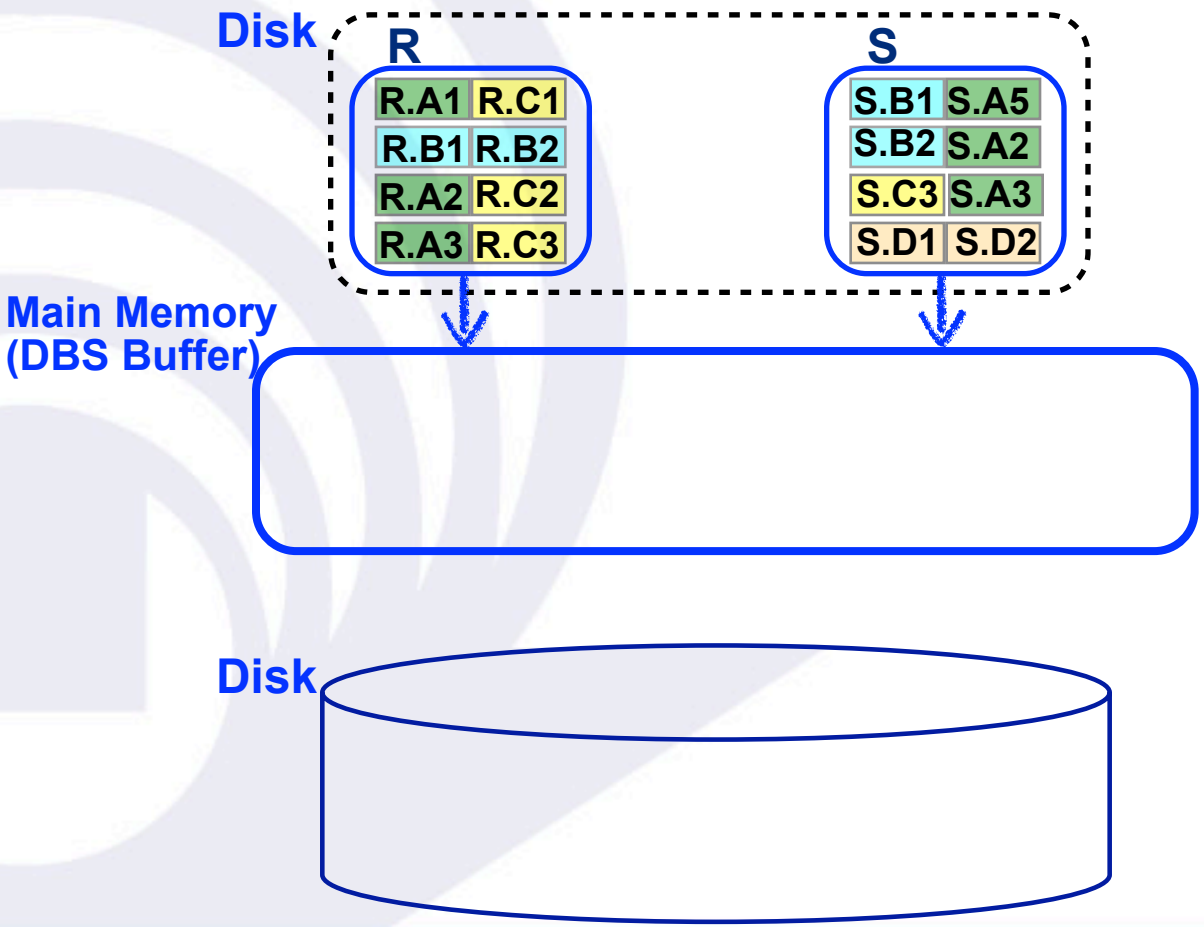
Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Join Algorithms

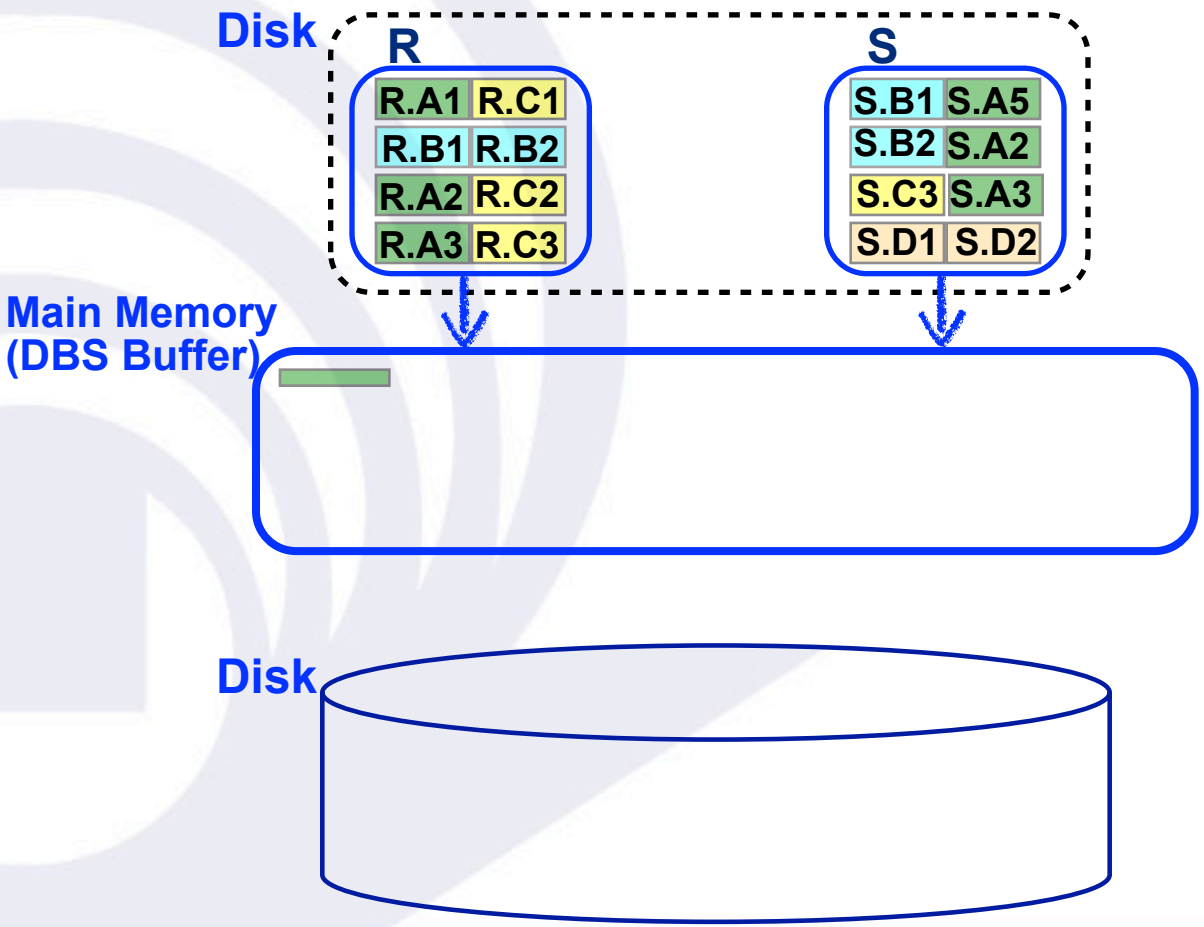
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

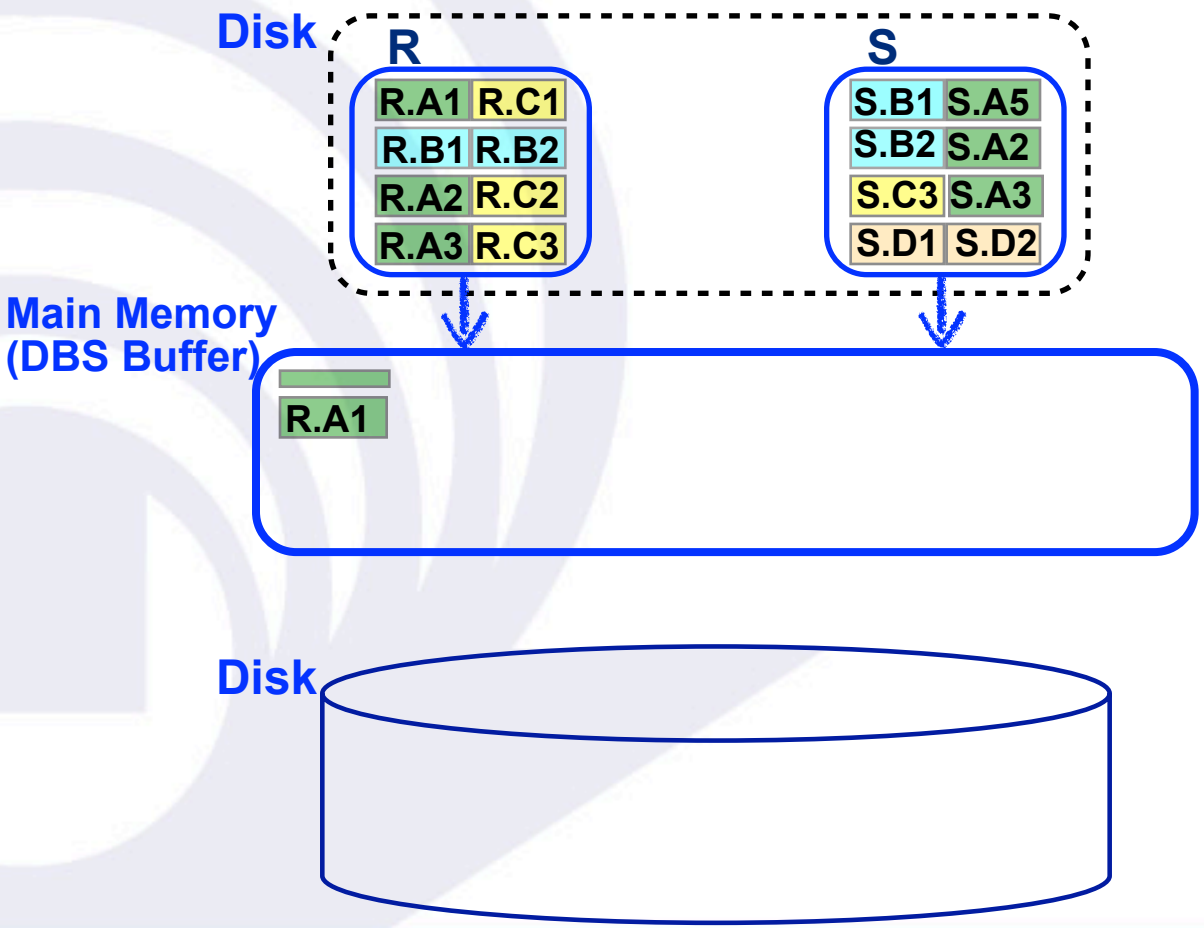
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

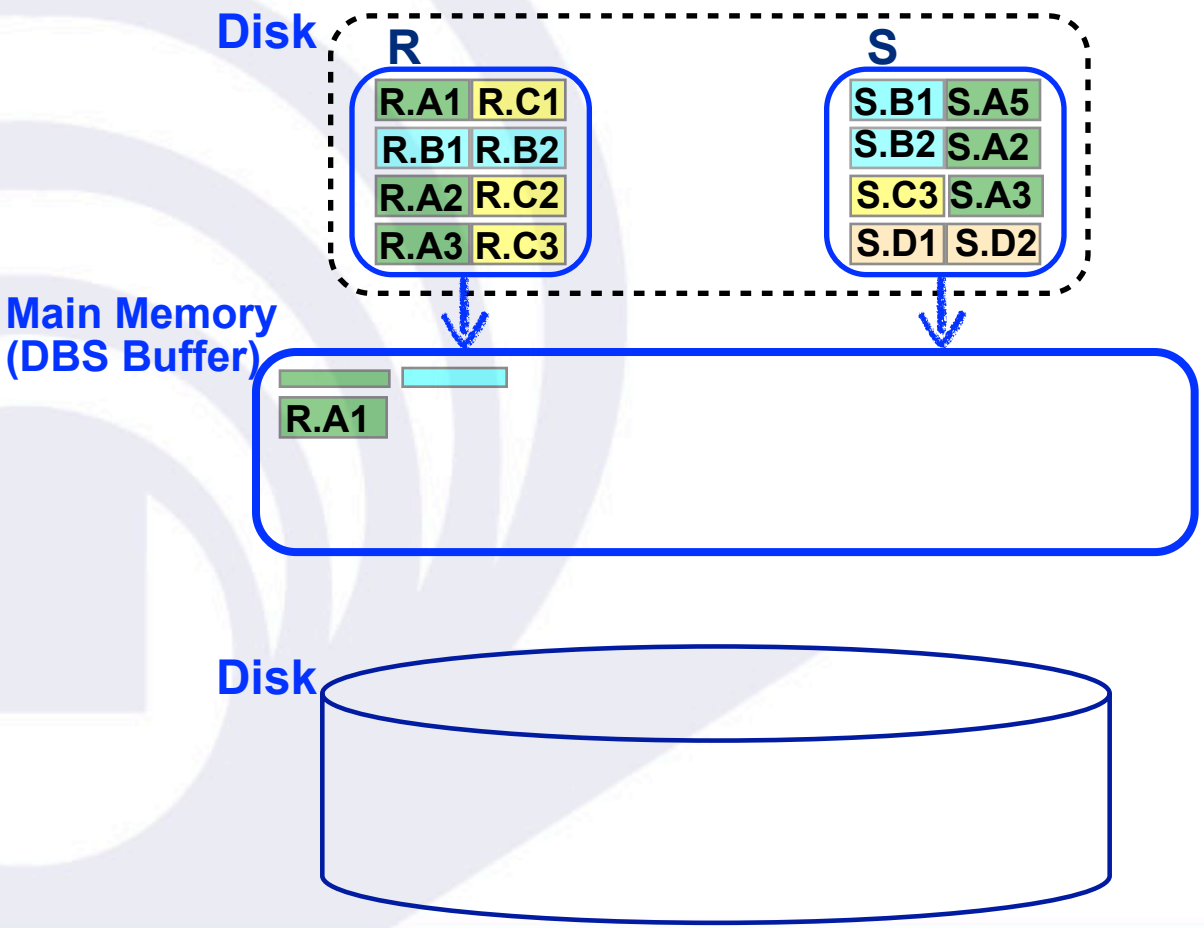
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

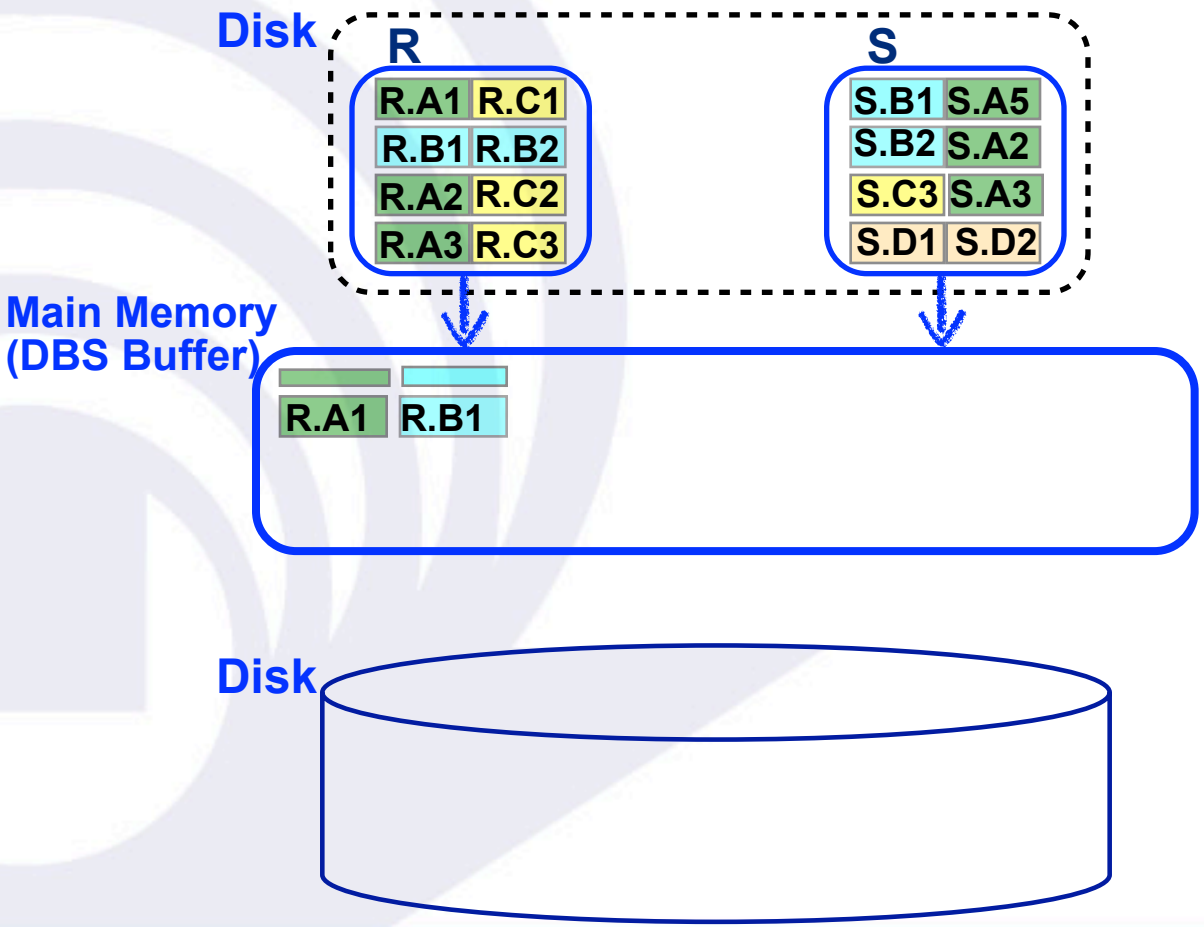
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

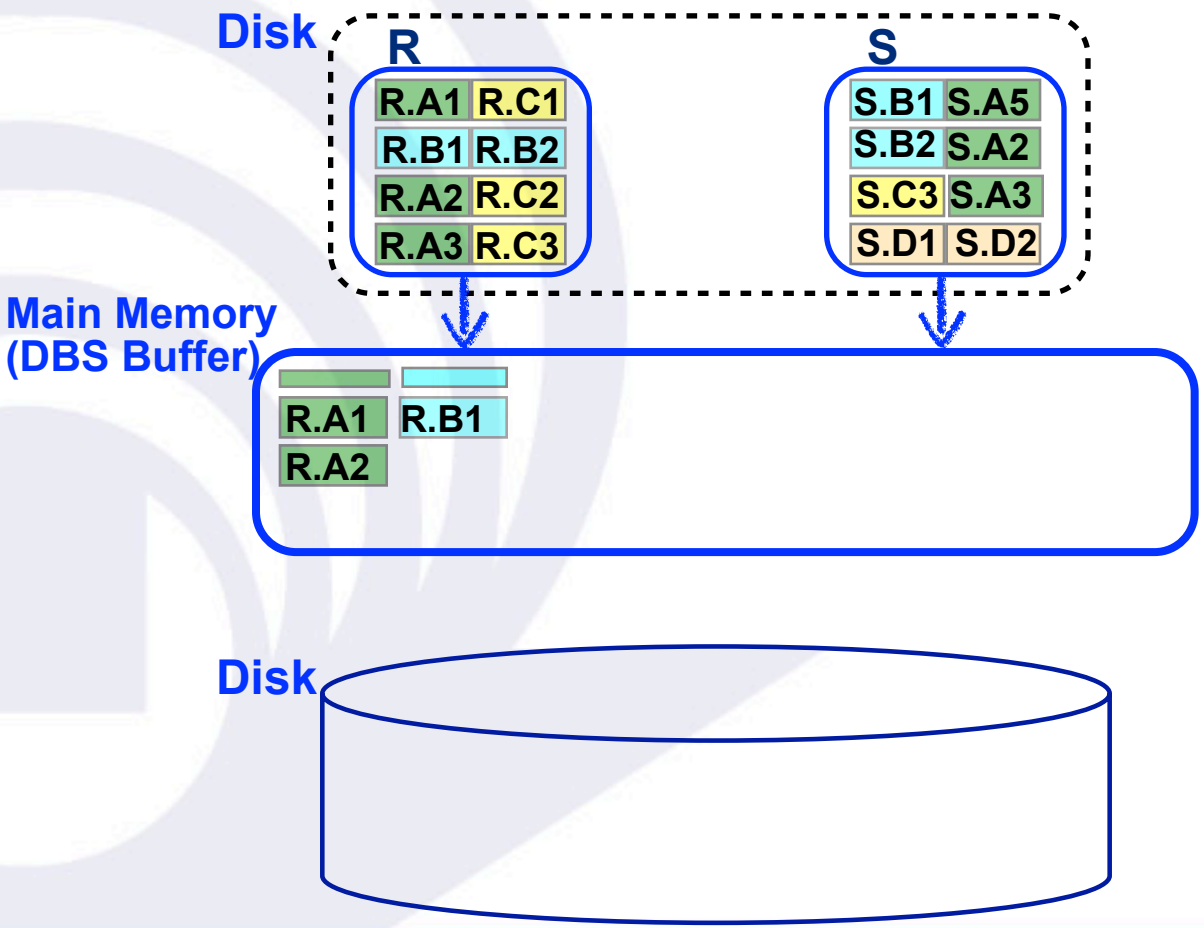
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

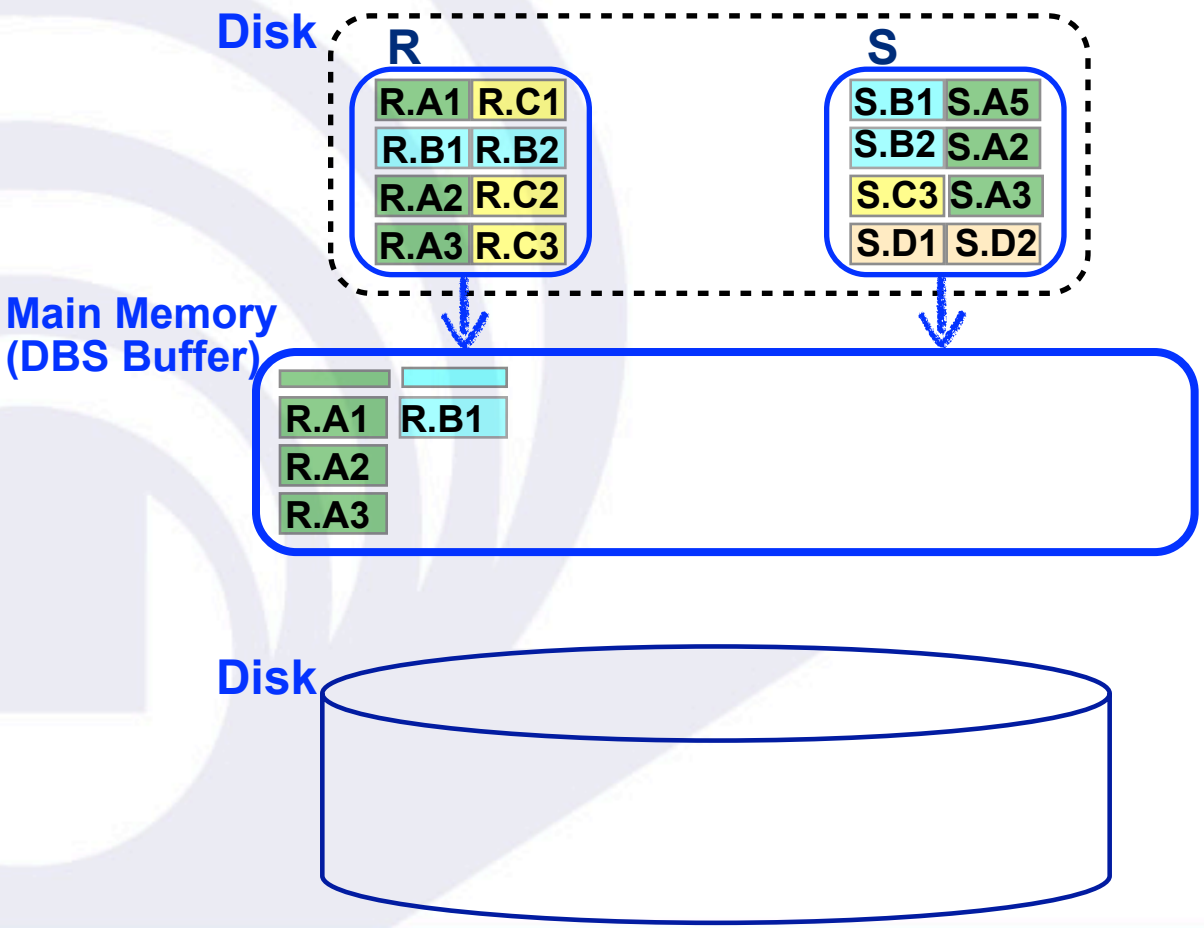
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

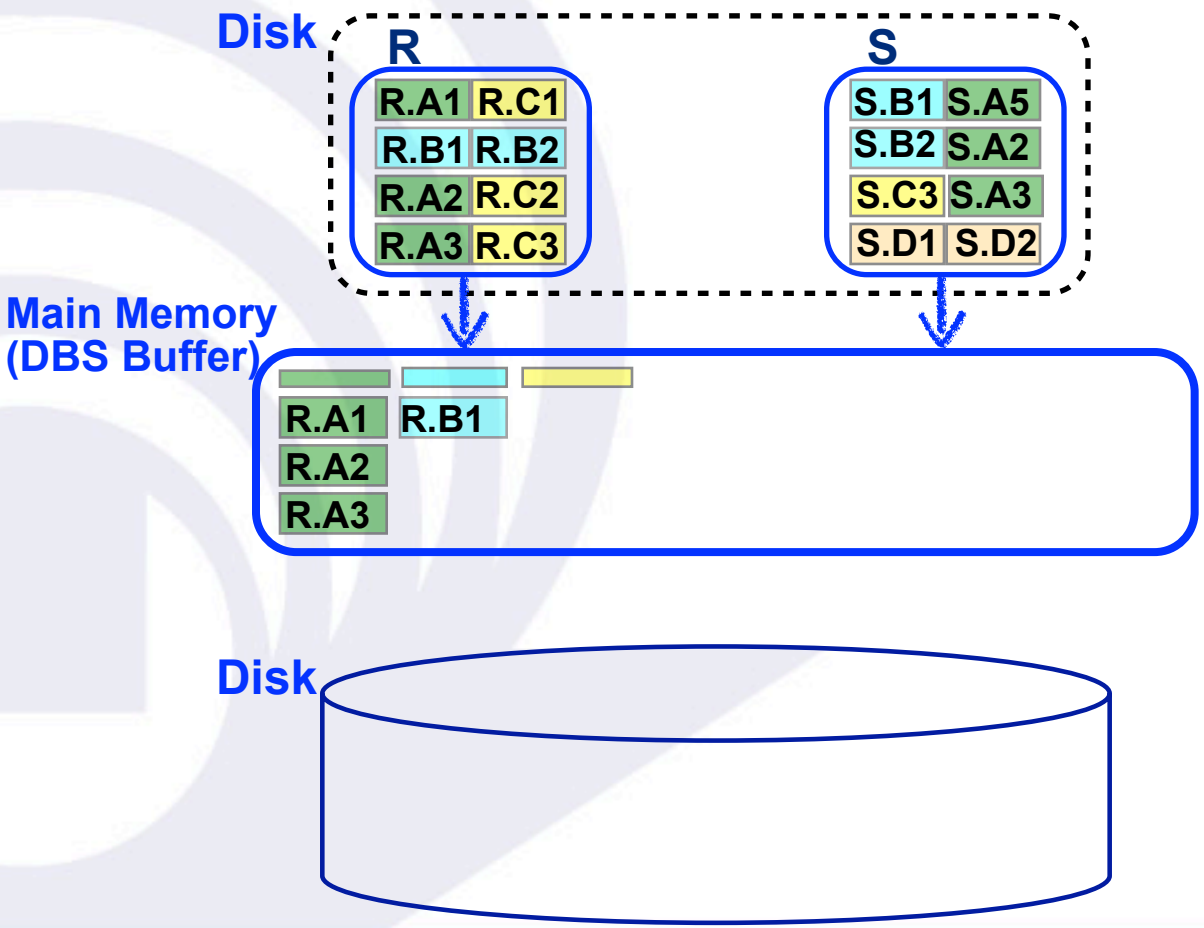
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

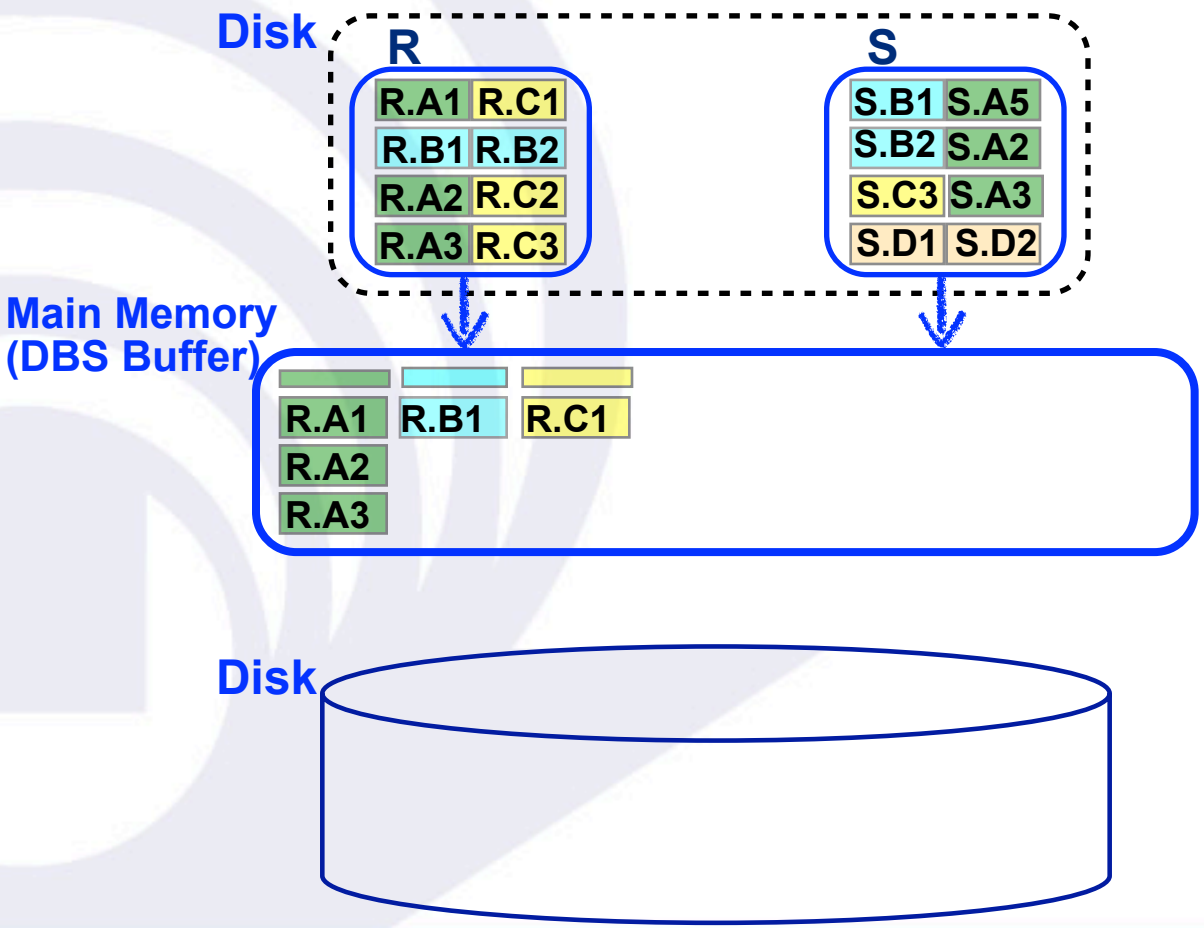
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

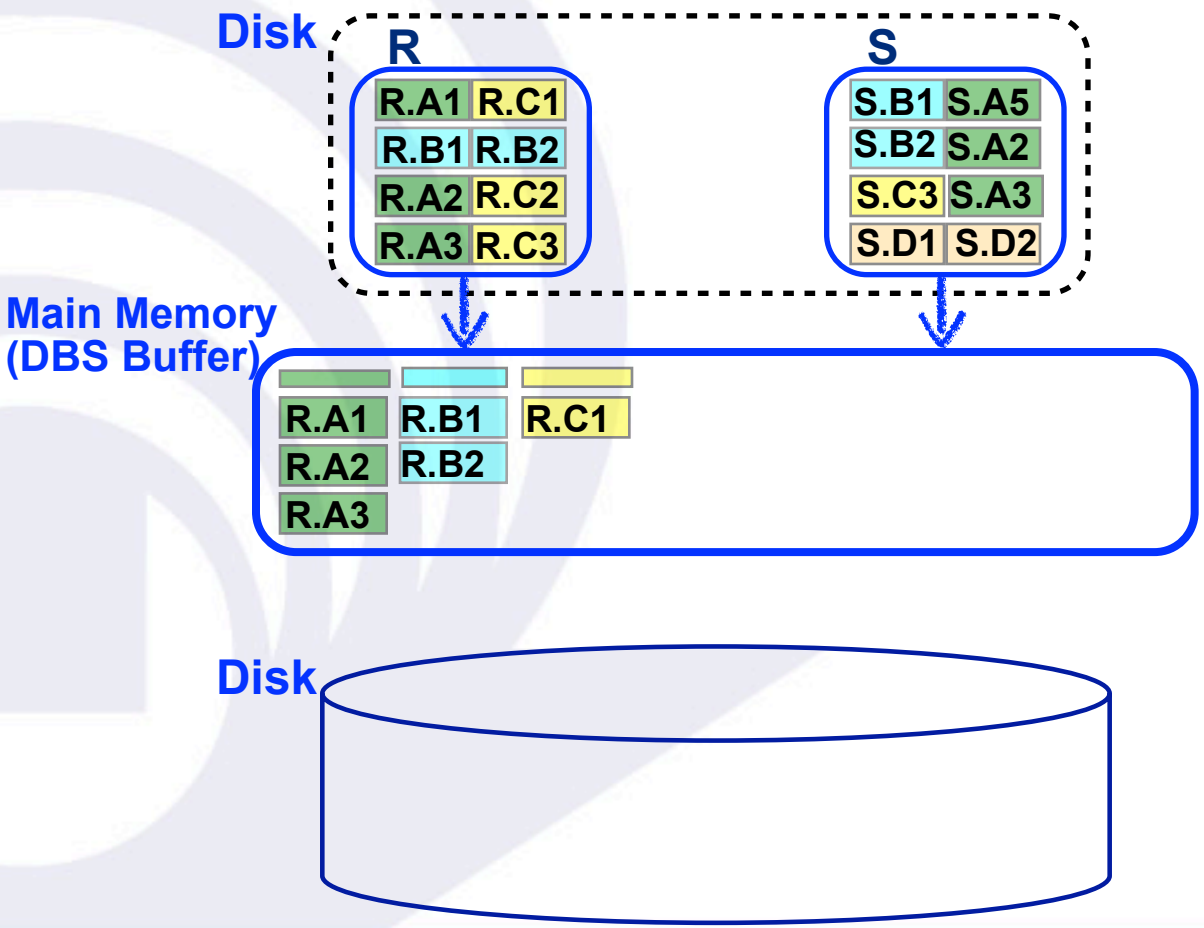
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

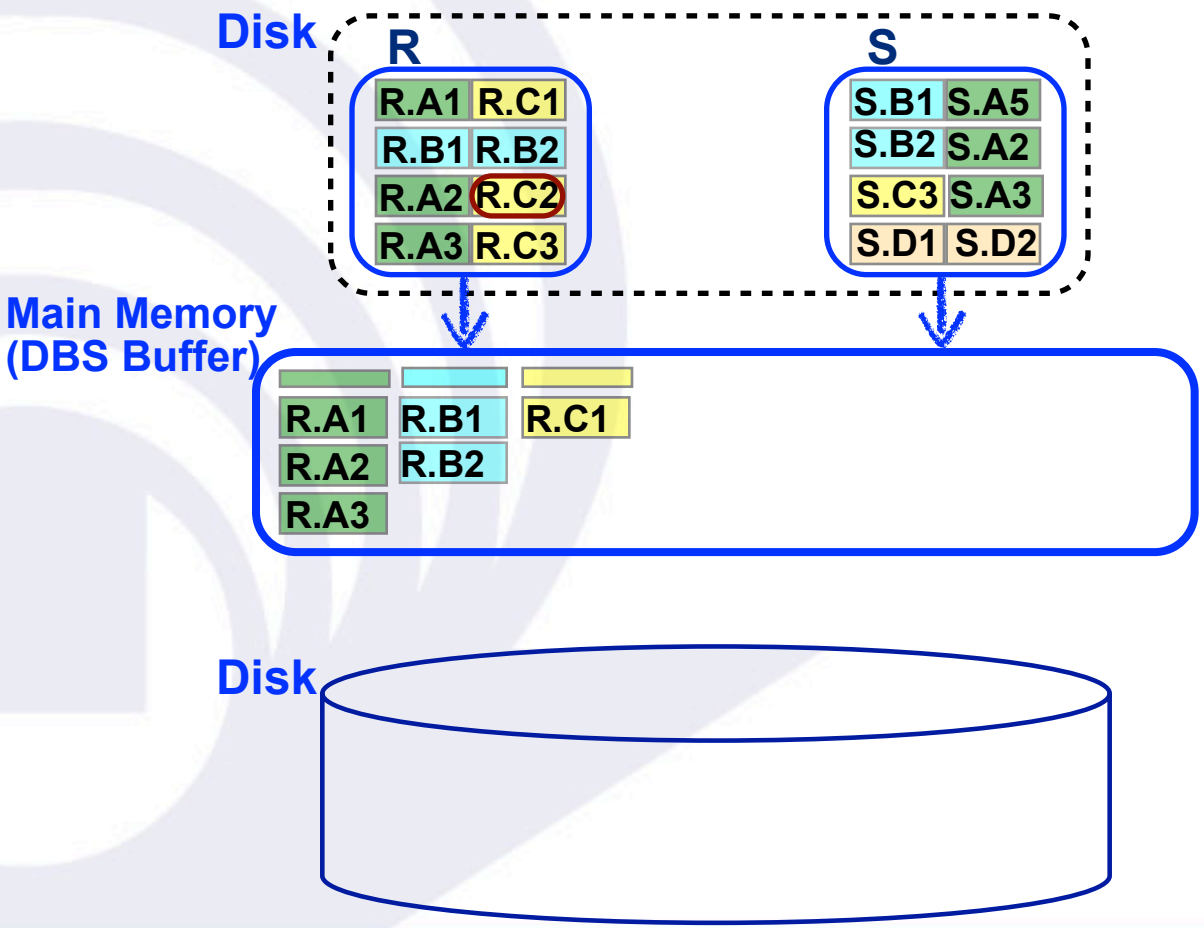
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

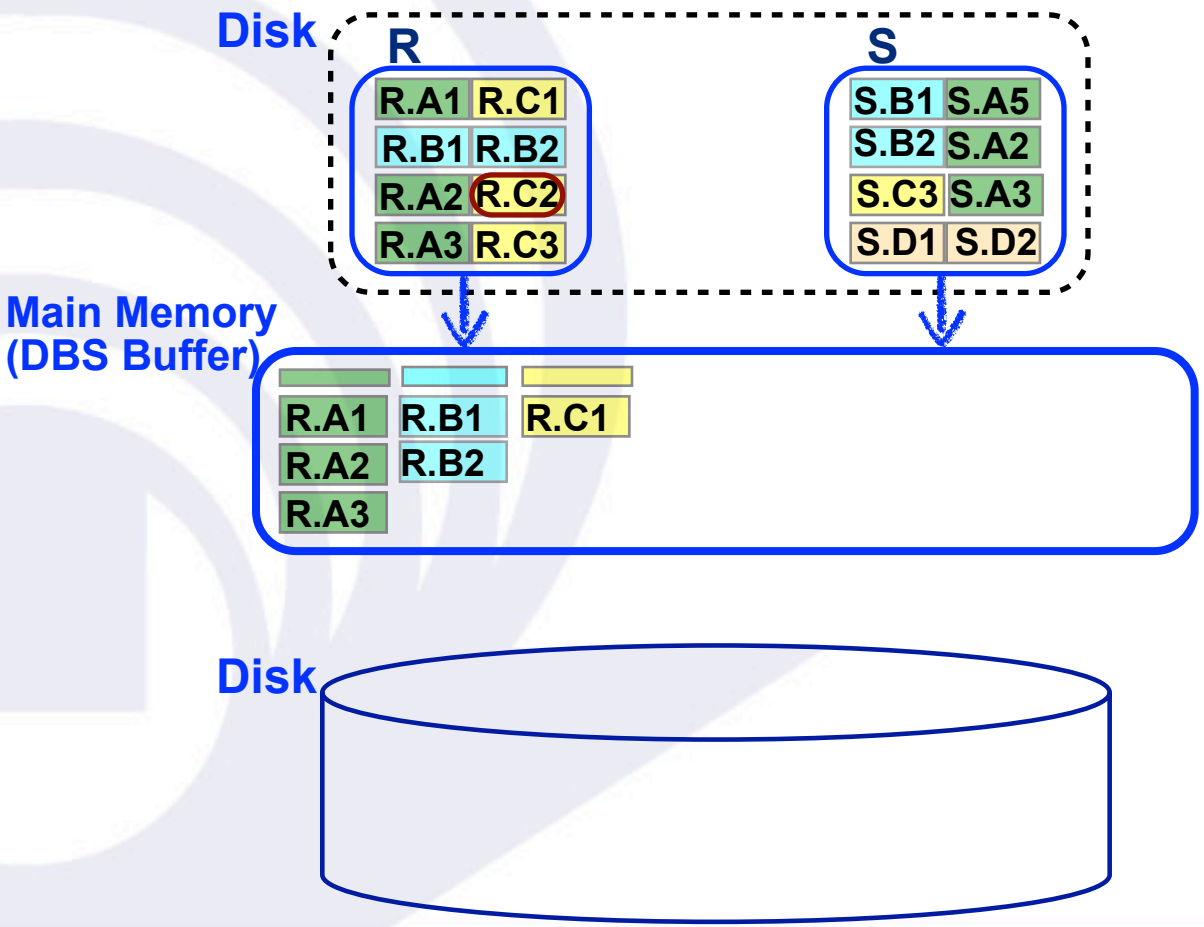
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer

Join Algorithms

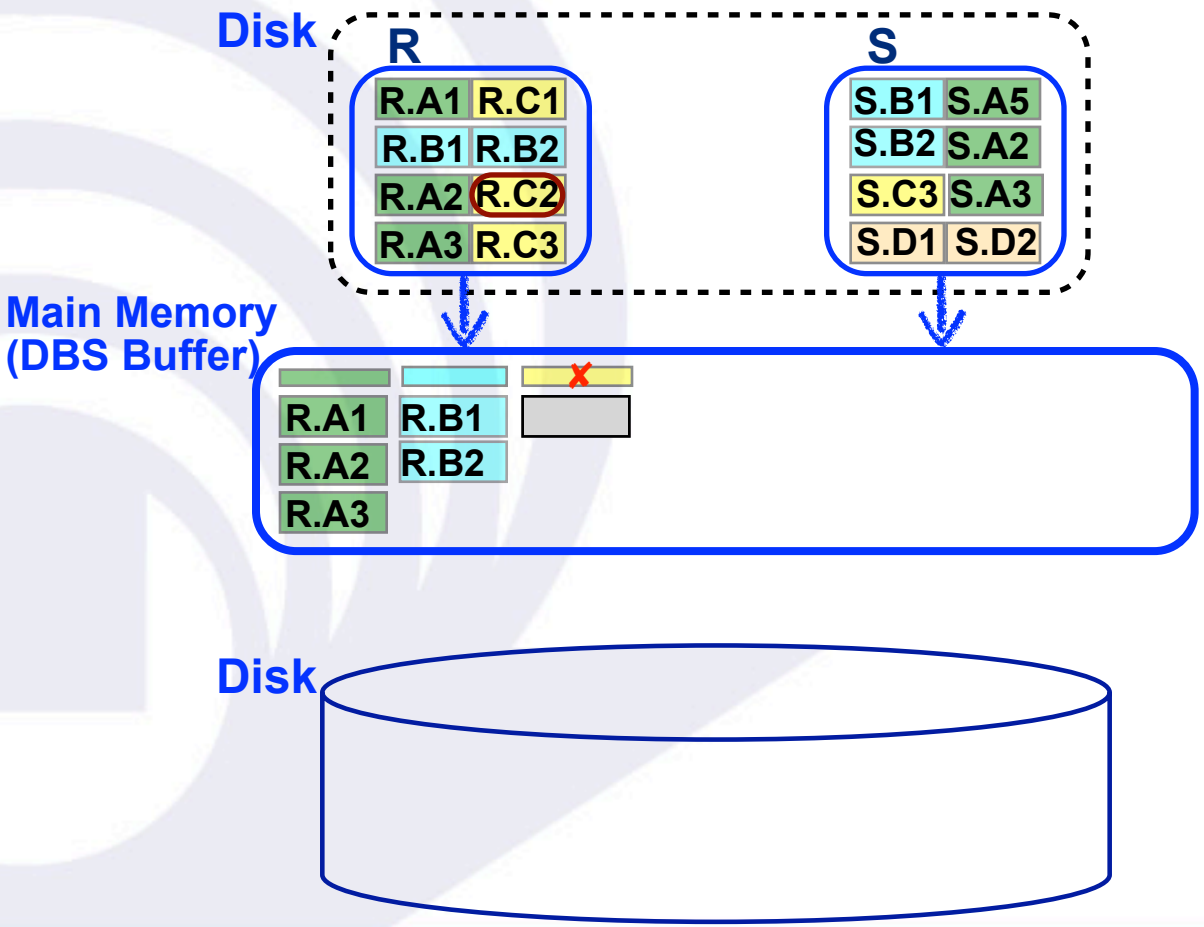
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition

Join Algorithms

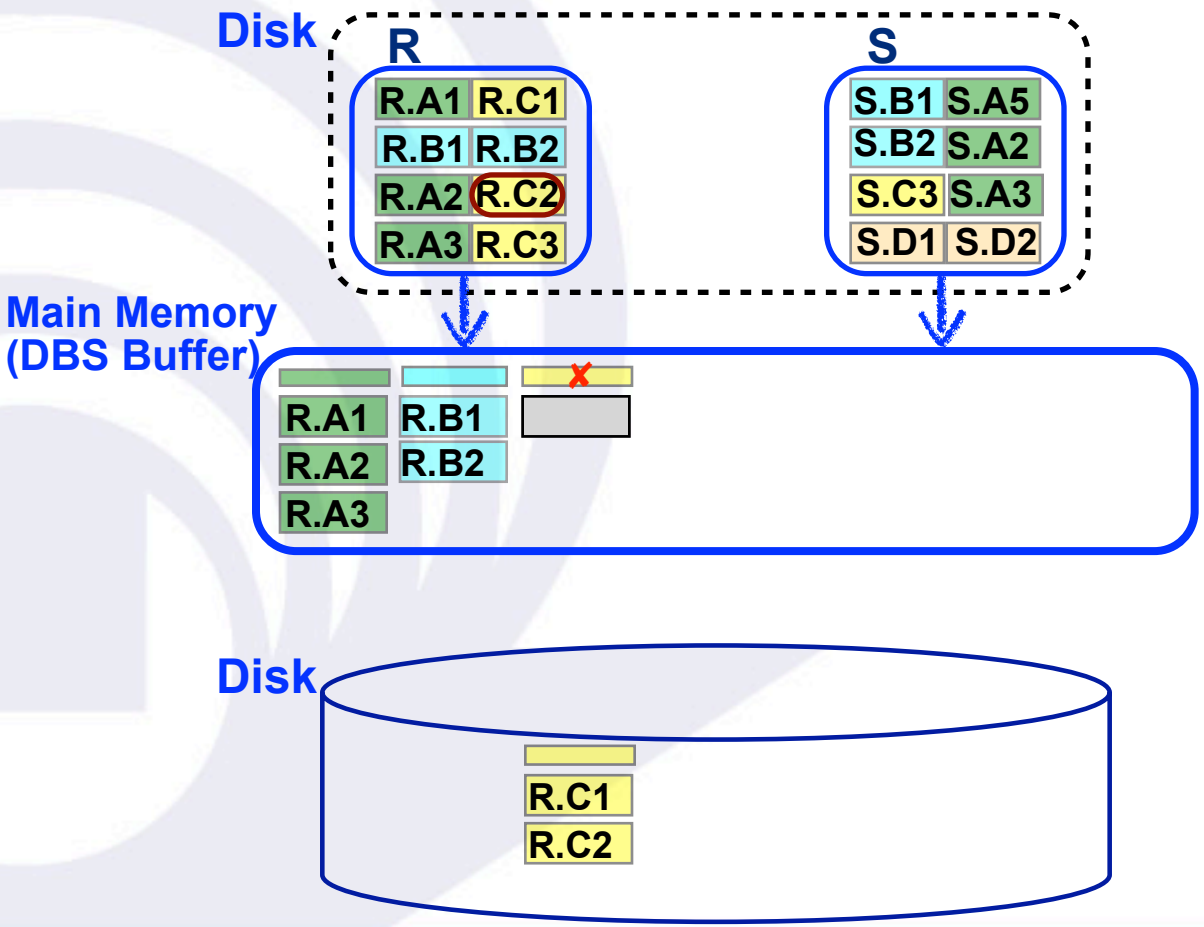
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition

Join Algorithms

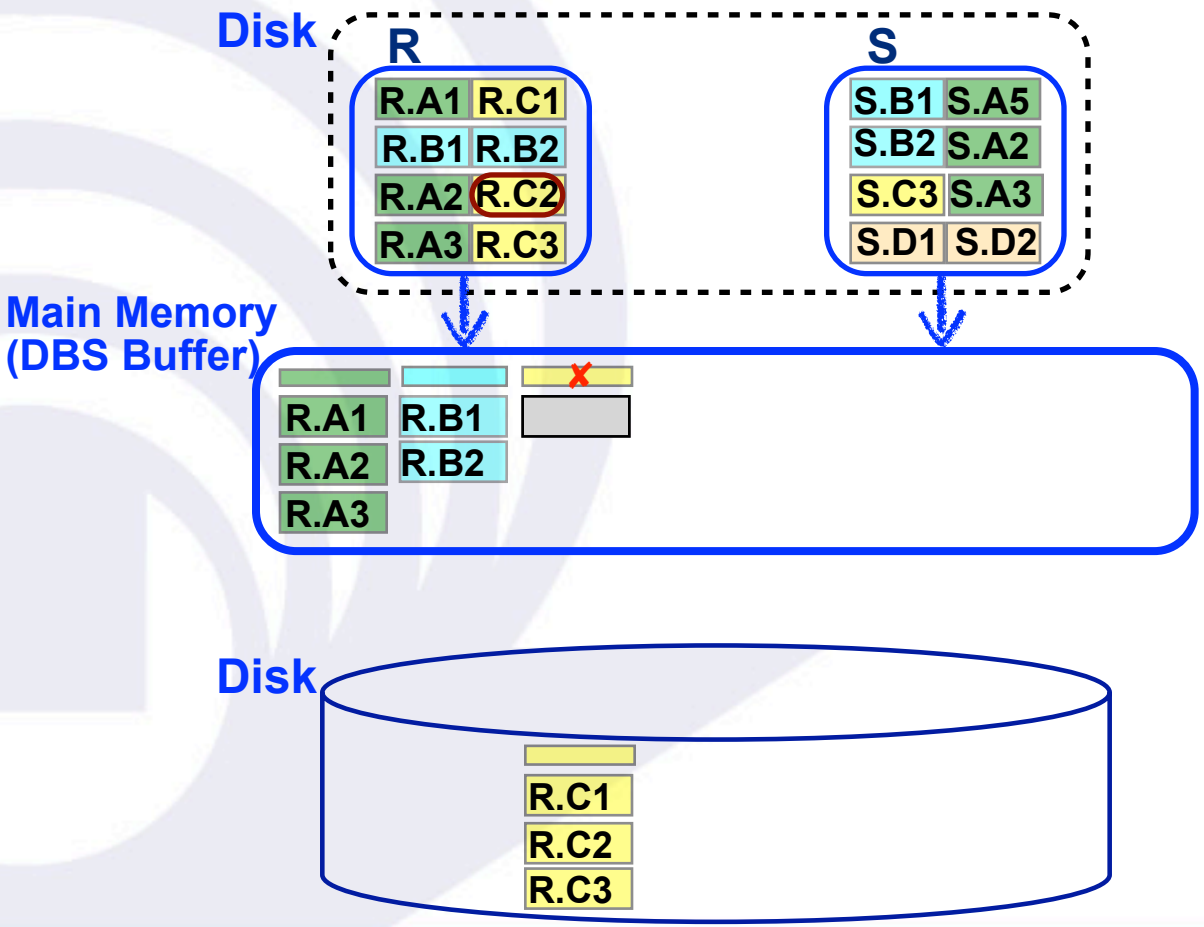
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition

Join Algorithms

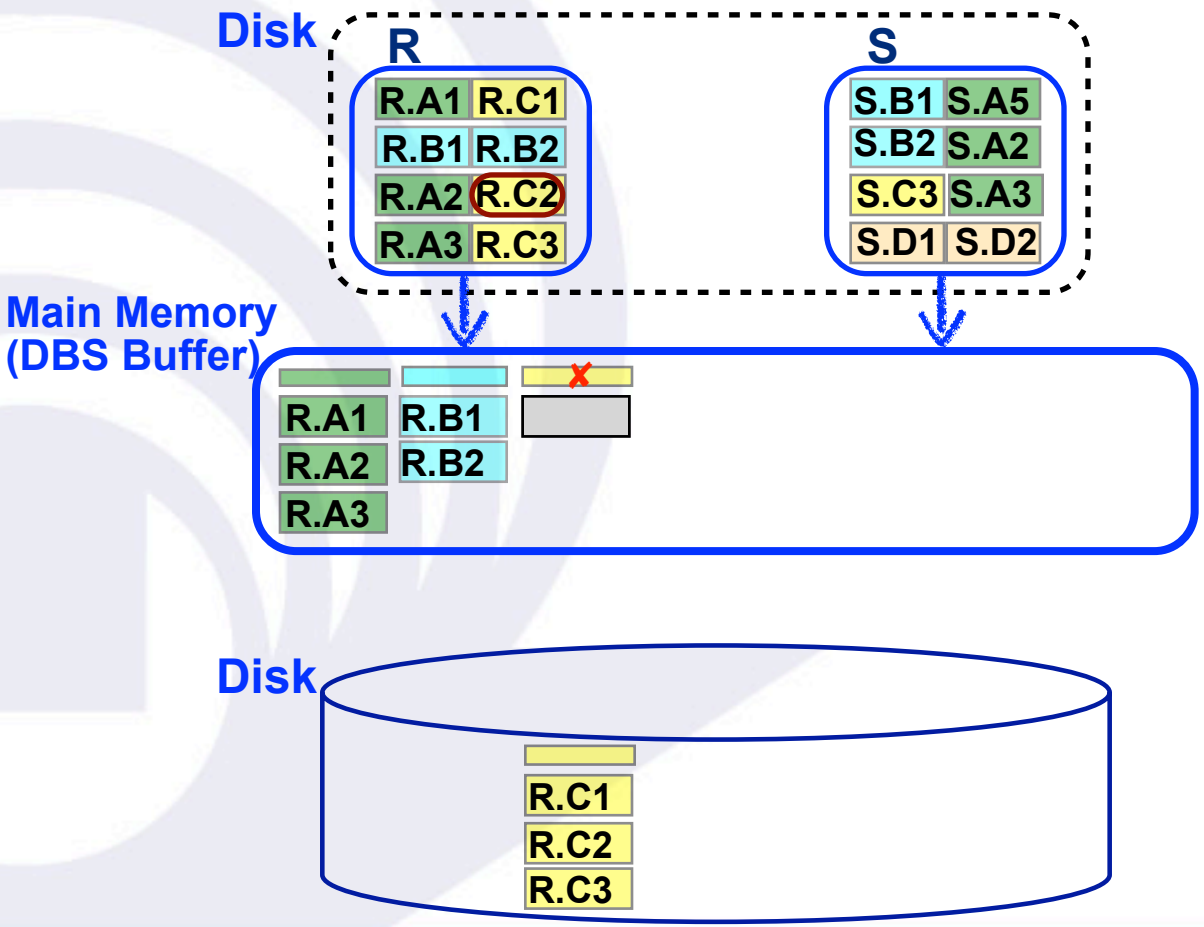
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition

Join Algorithms

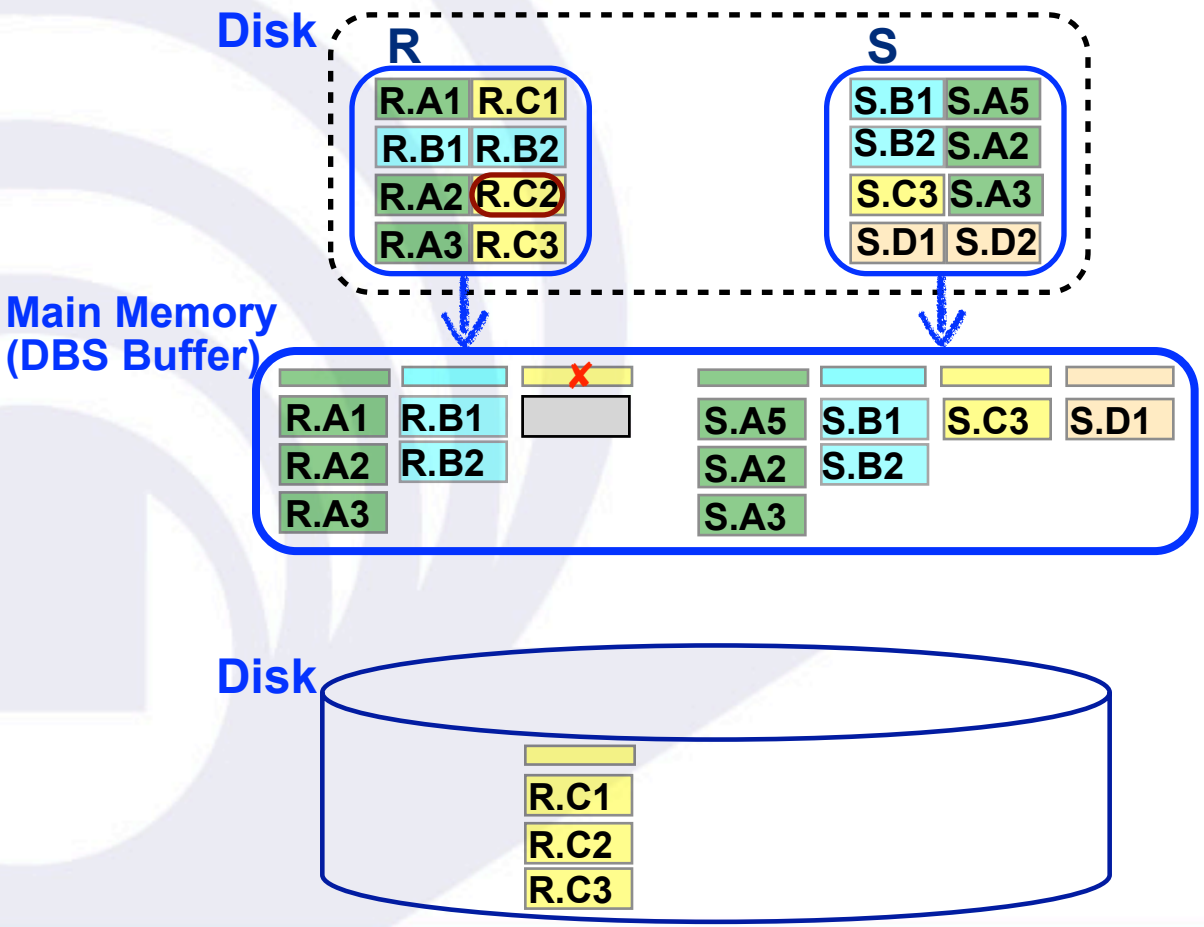
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer

Join Algorithms

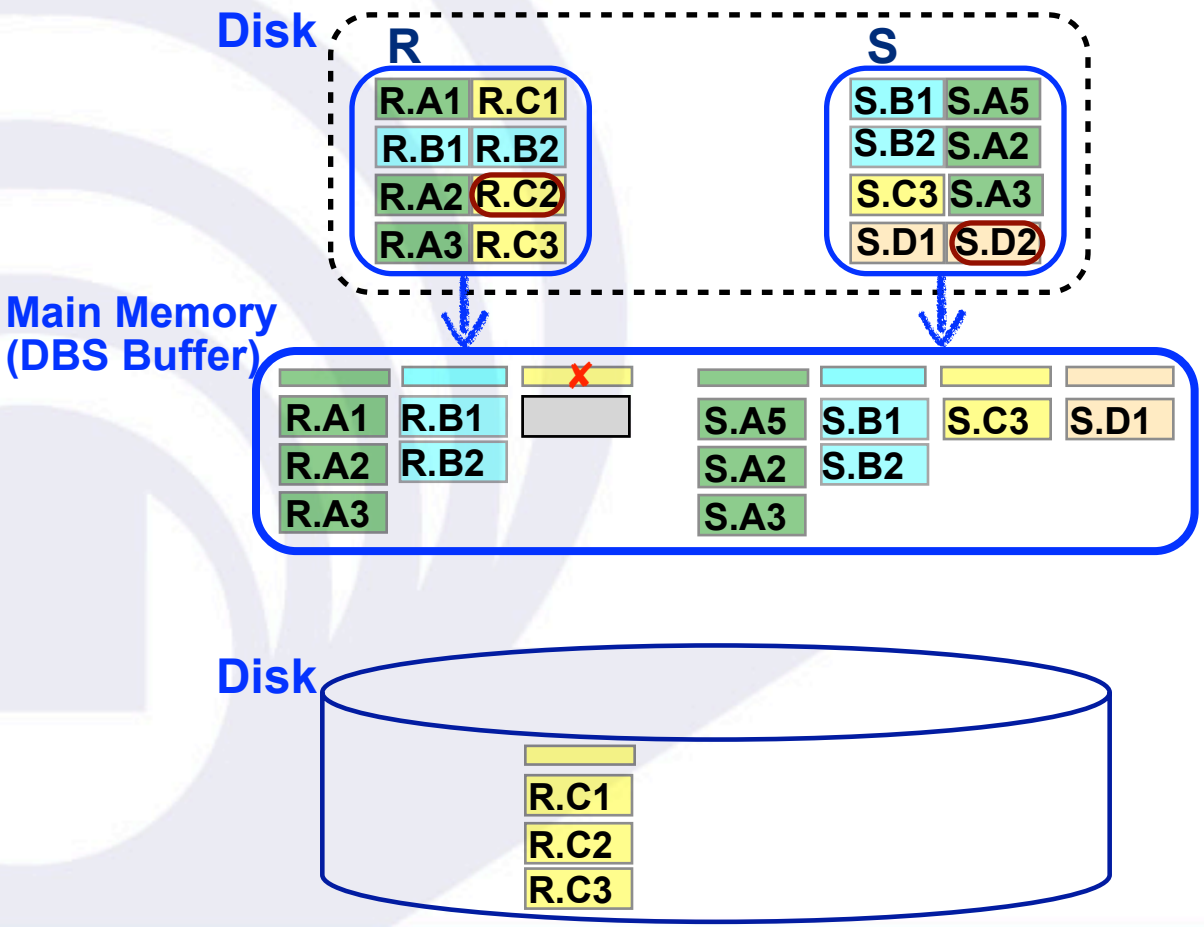
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer

Join Algorithms

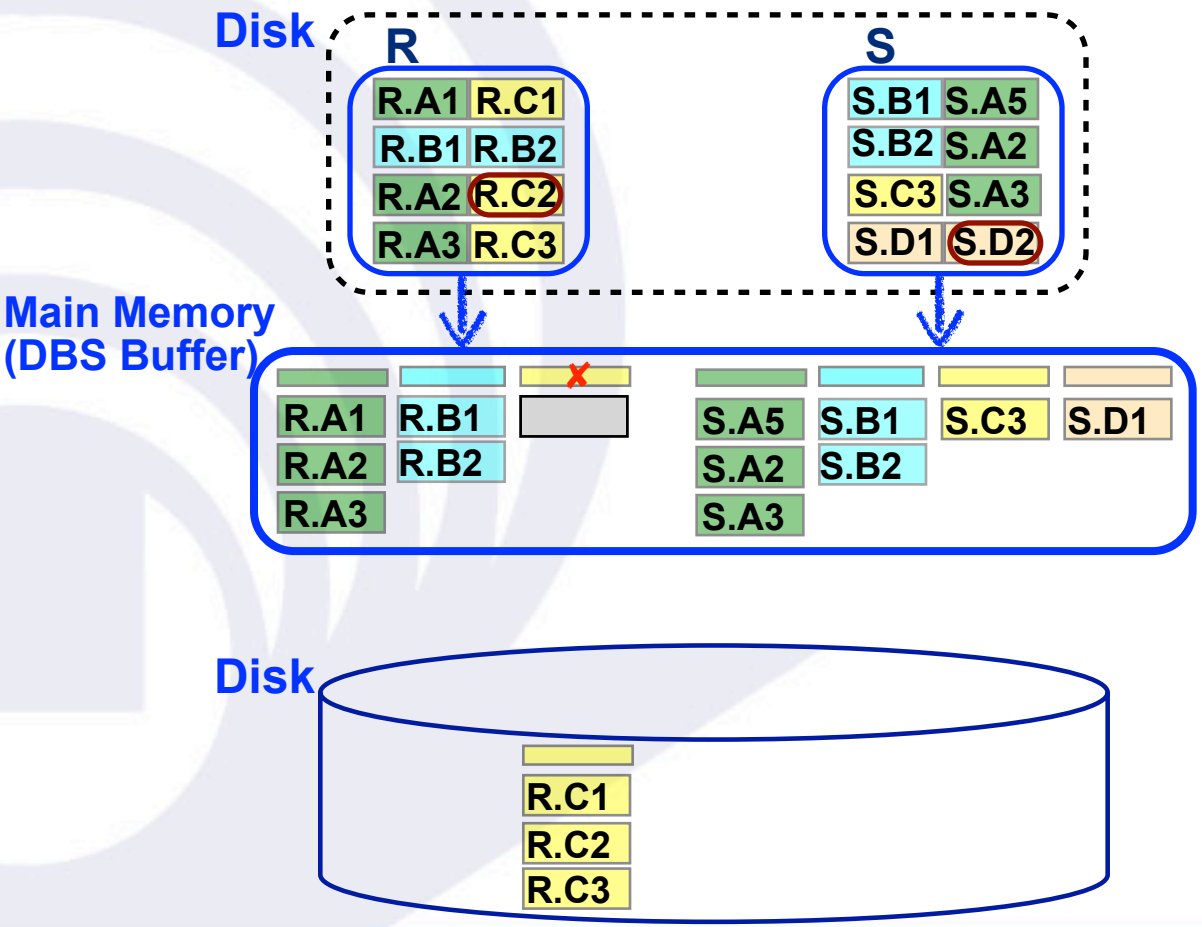
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer

Join Algorithms

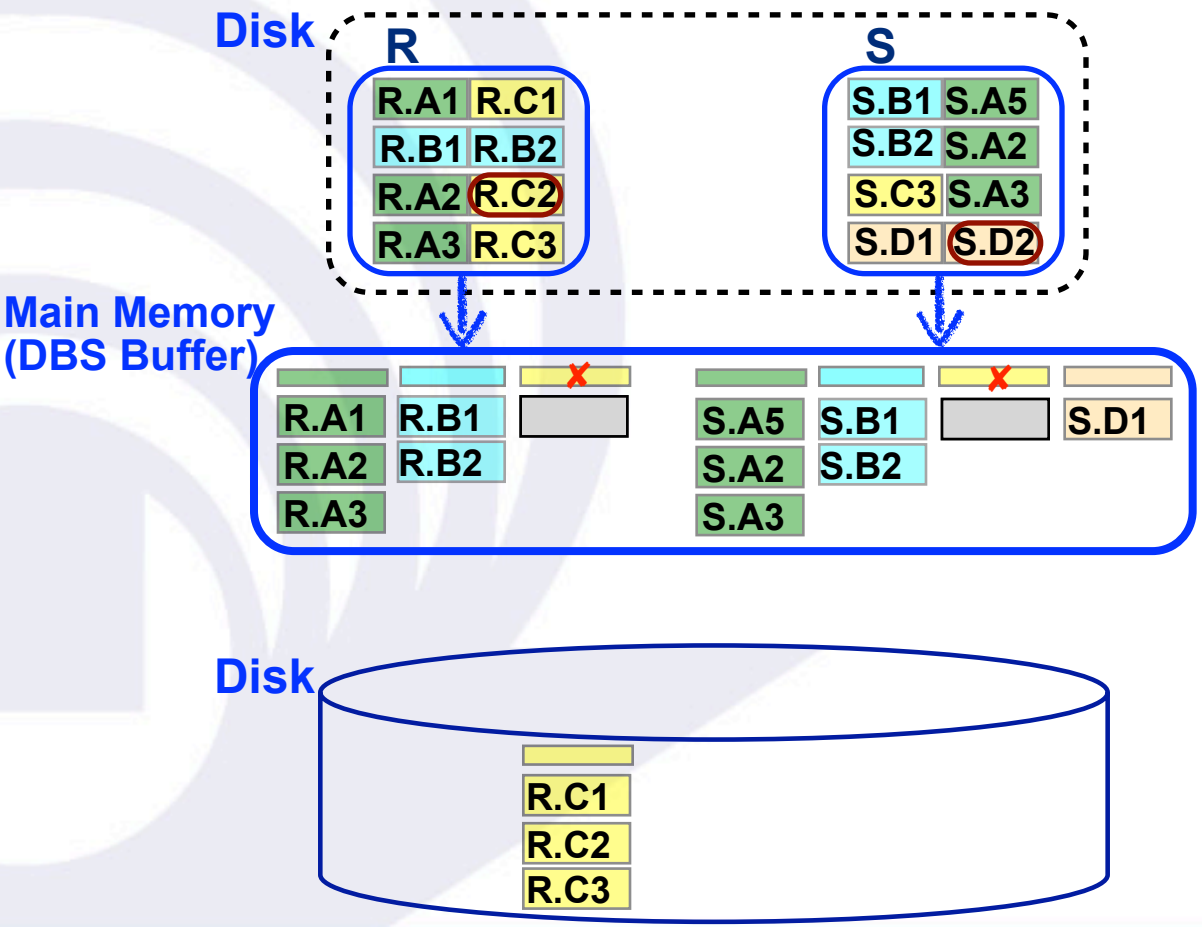
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk

Join Algorithms

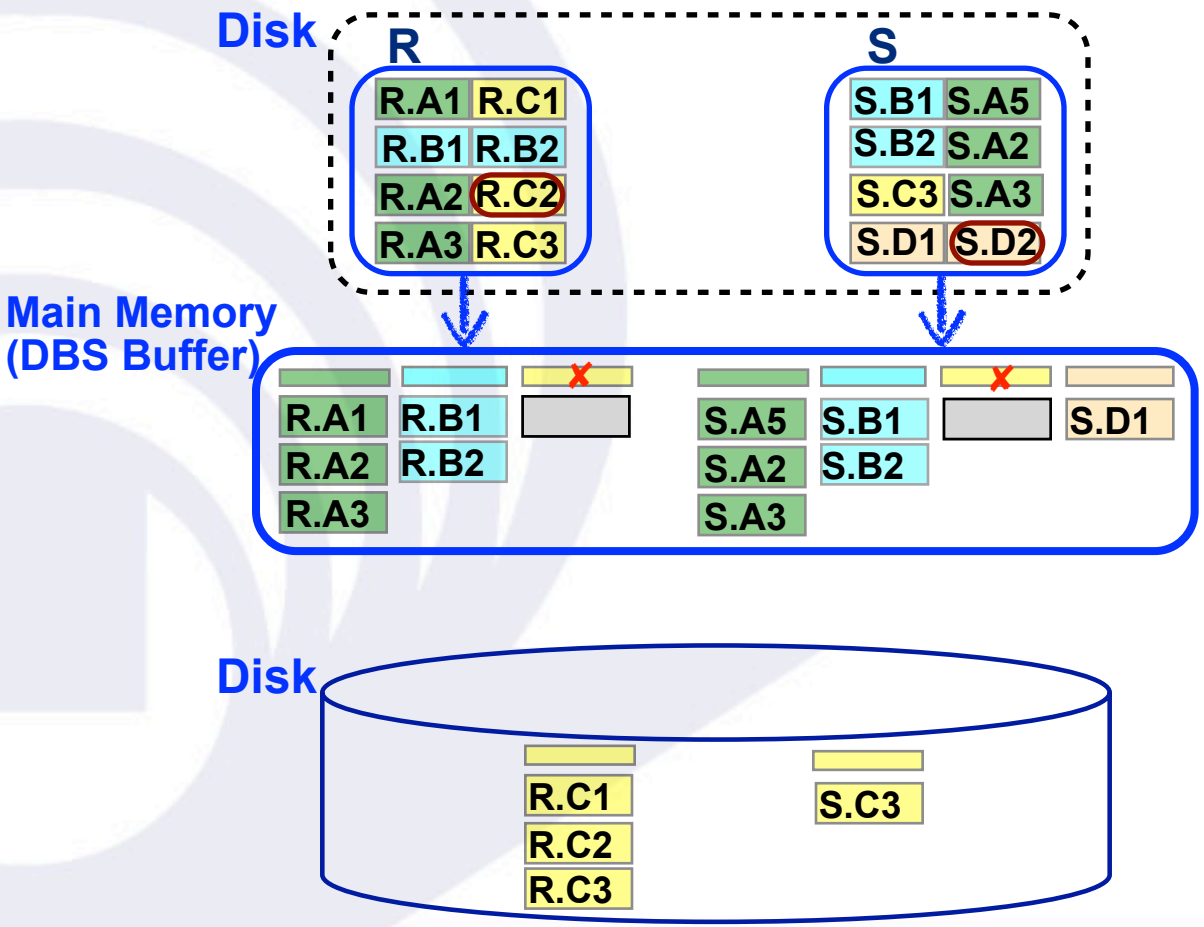
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk

Join Algorithms

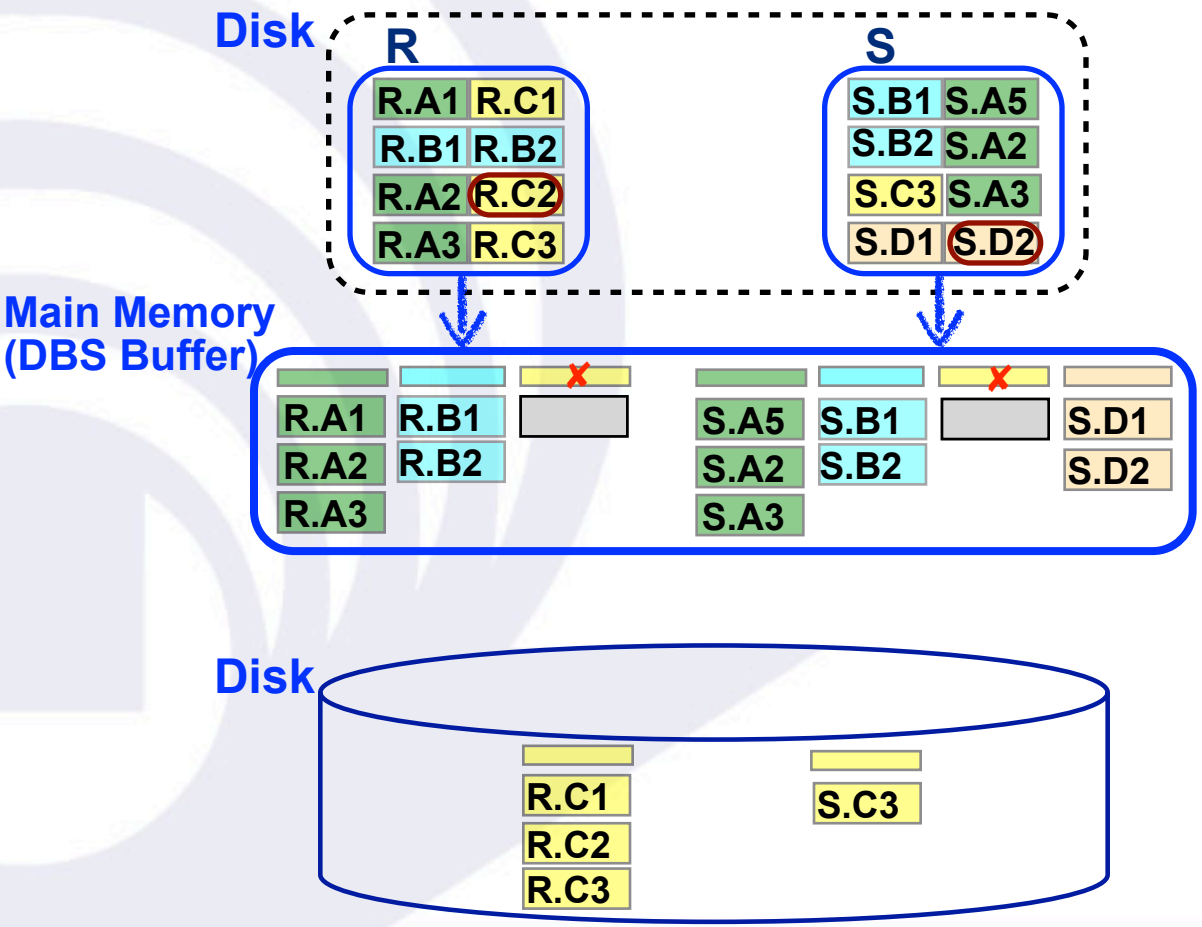
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk

Join Algorithms

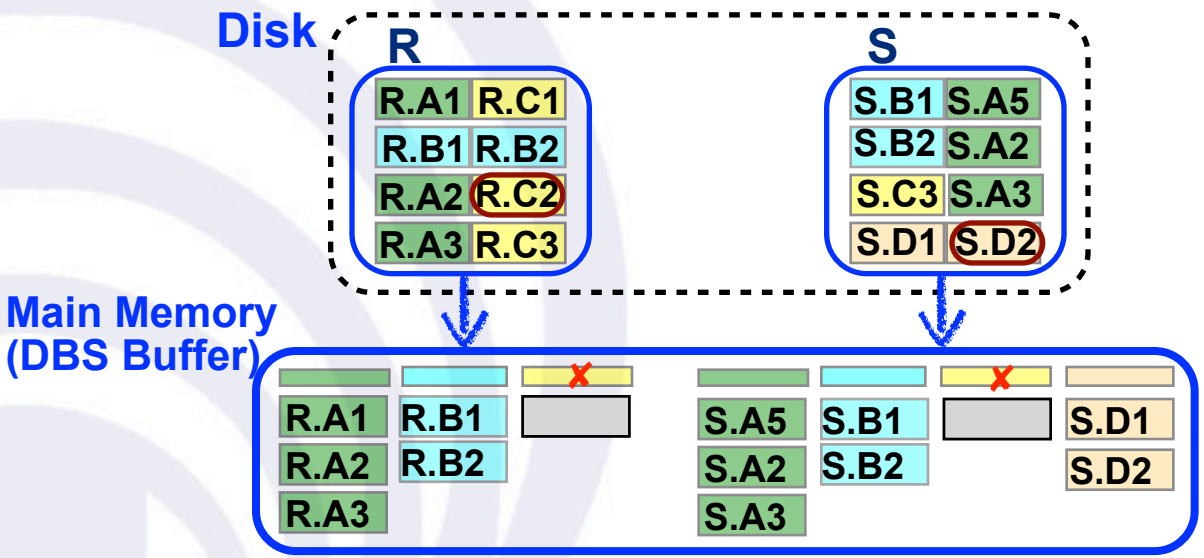
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



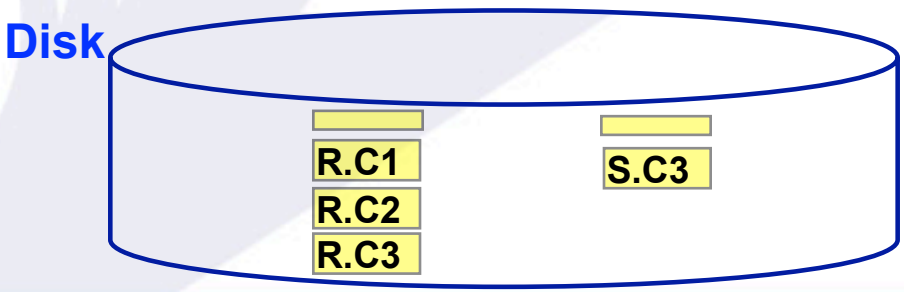
Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk

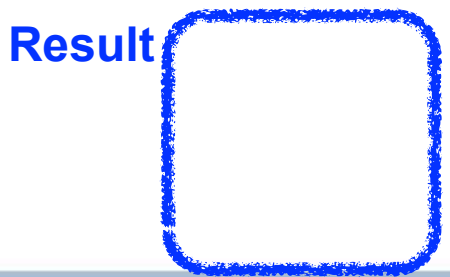
Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



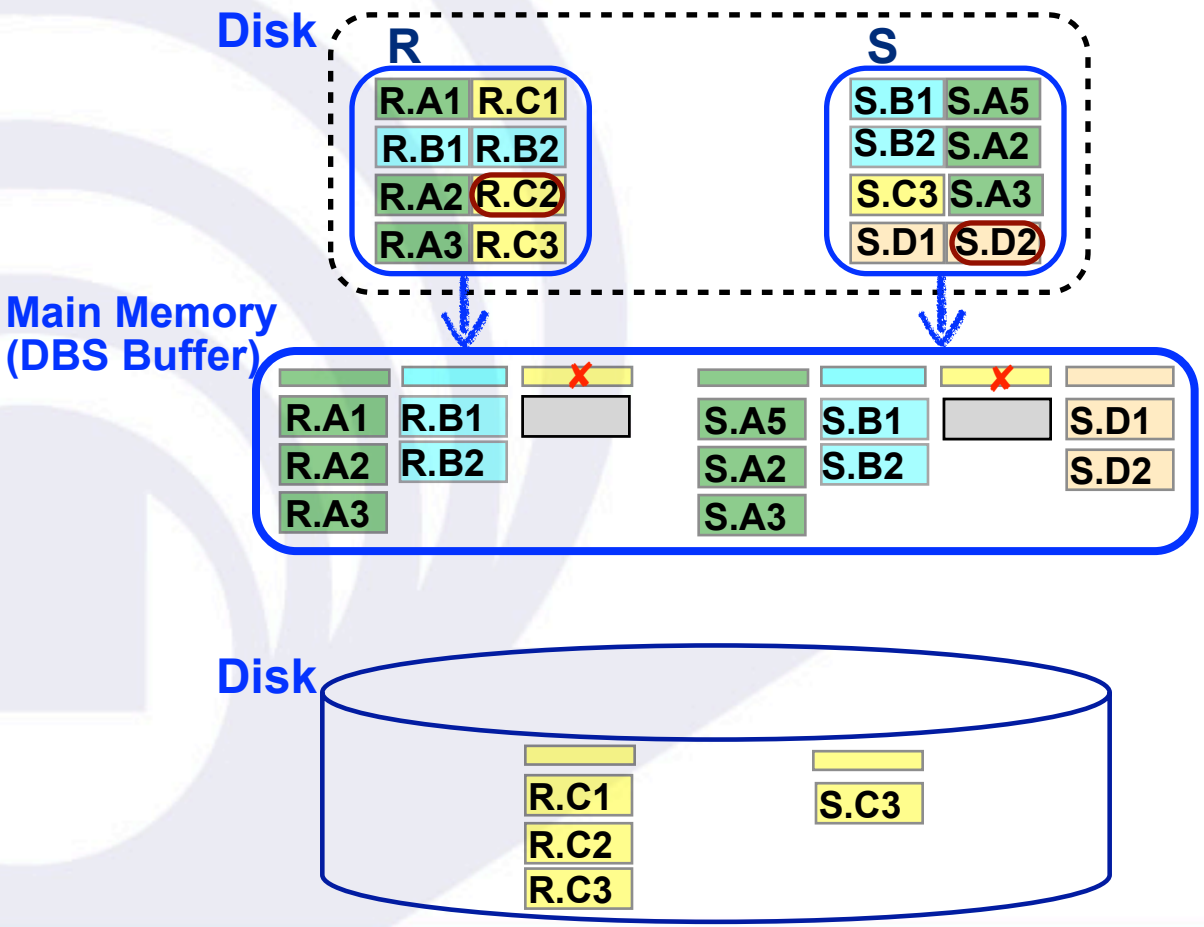
Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk





Join Algorithms

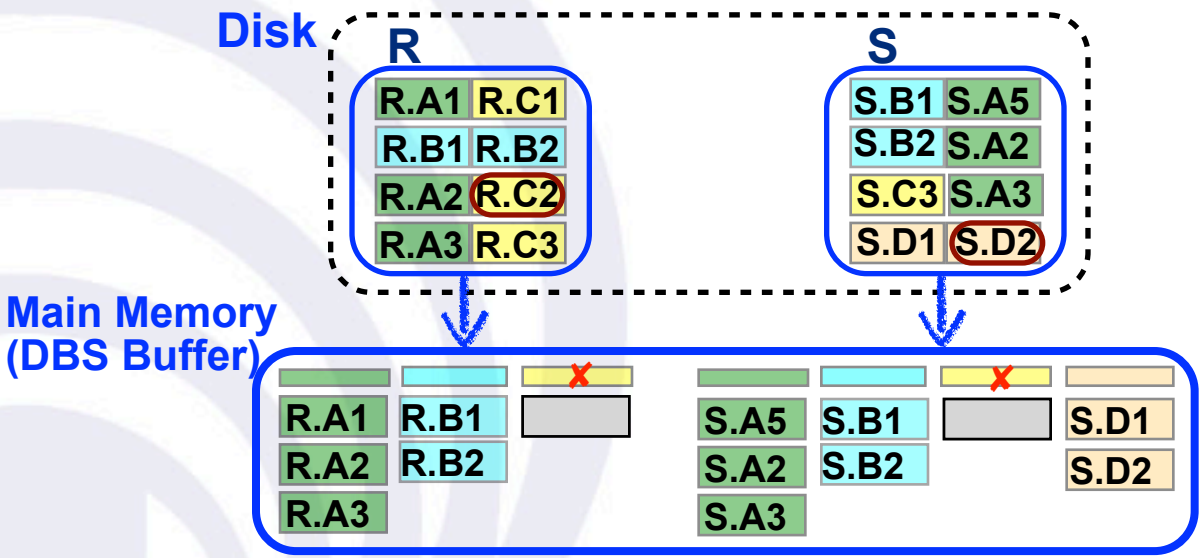
- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



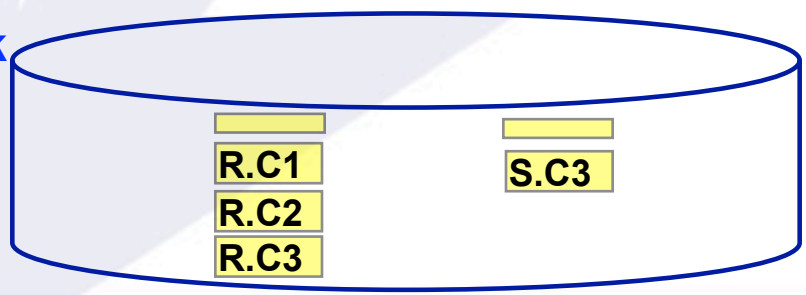
Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk
Execute nested-loop join on each pair of in-memory partitions

Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk
Execute nested-loop join on each pair of in-memory partitions

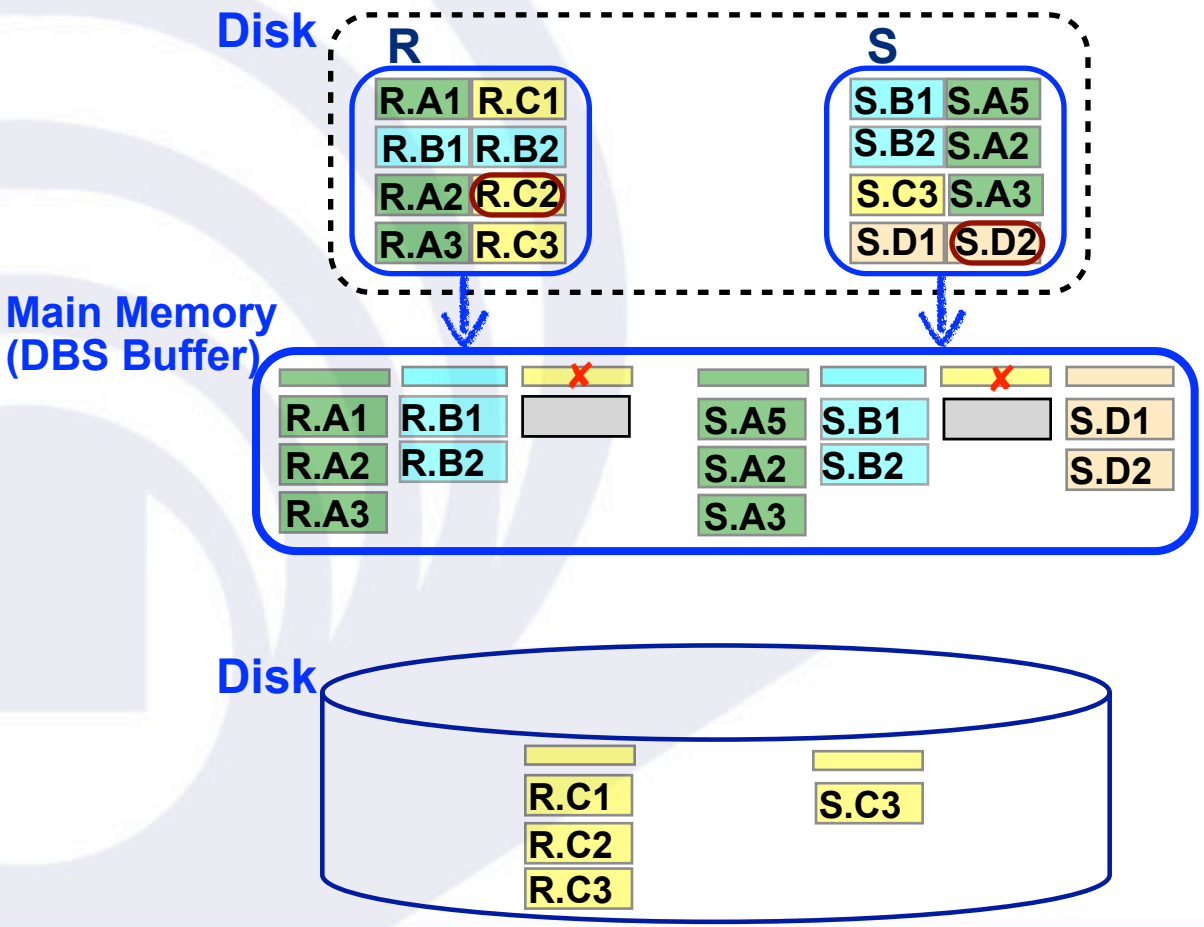


Result

R.A2	S.A2
R.A3	S.A3
R.B1	S.B1
R.B2	S.B2

Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



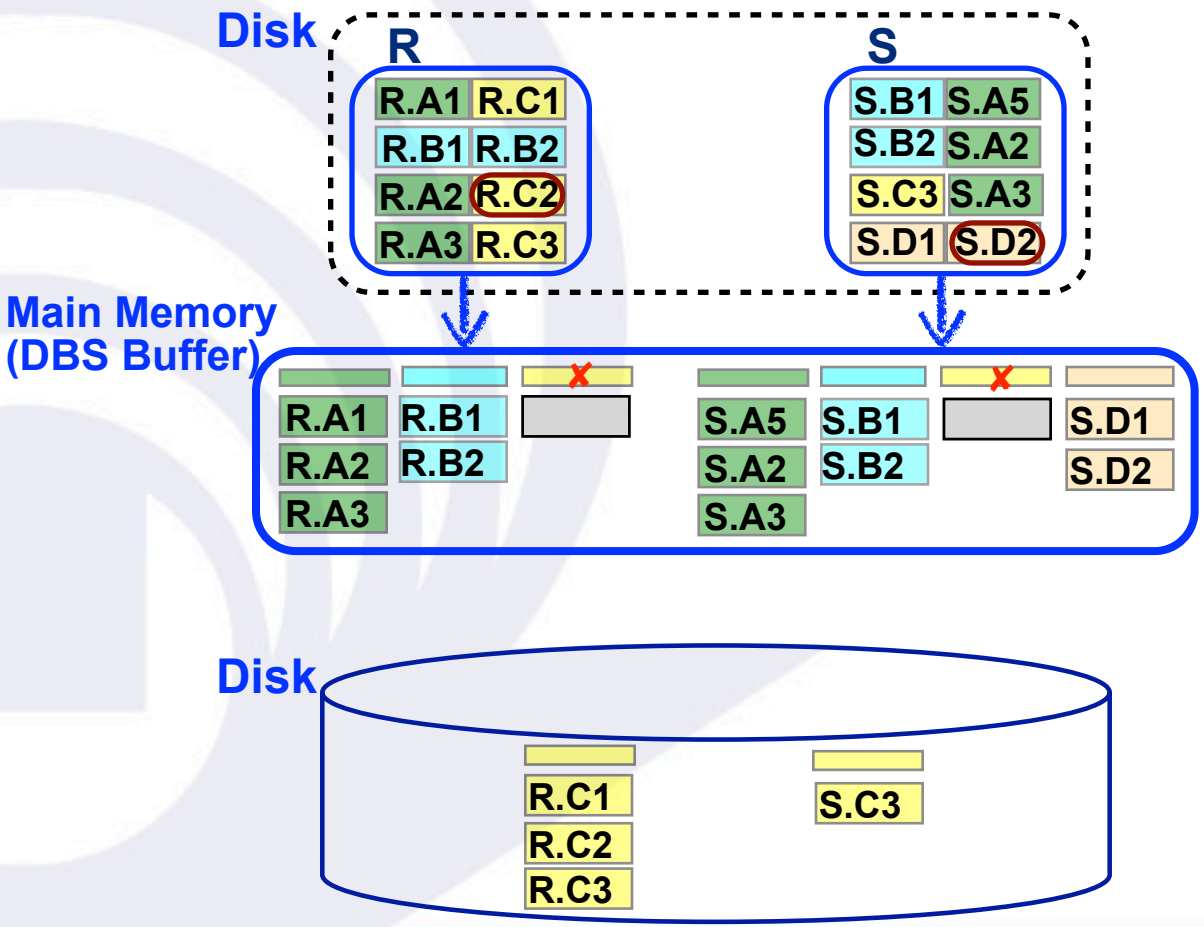
Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk
Execute nested-loop join on each pair of in-memory partitions
Execute nested-loop join on each pair of in-disk partitions

Result

R.A2	S.A2
R.A3	S.A3
R.B1	S.B1
R.B2	S.B2

Join Algorithms

- Hybrid Hash Join (R⋈S) [DeWitt et al. 1984]



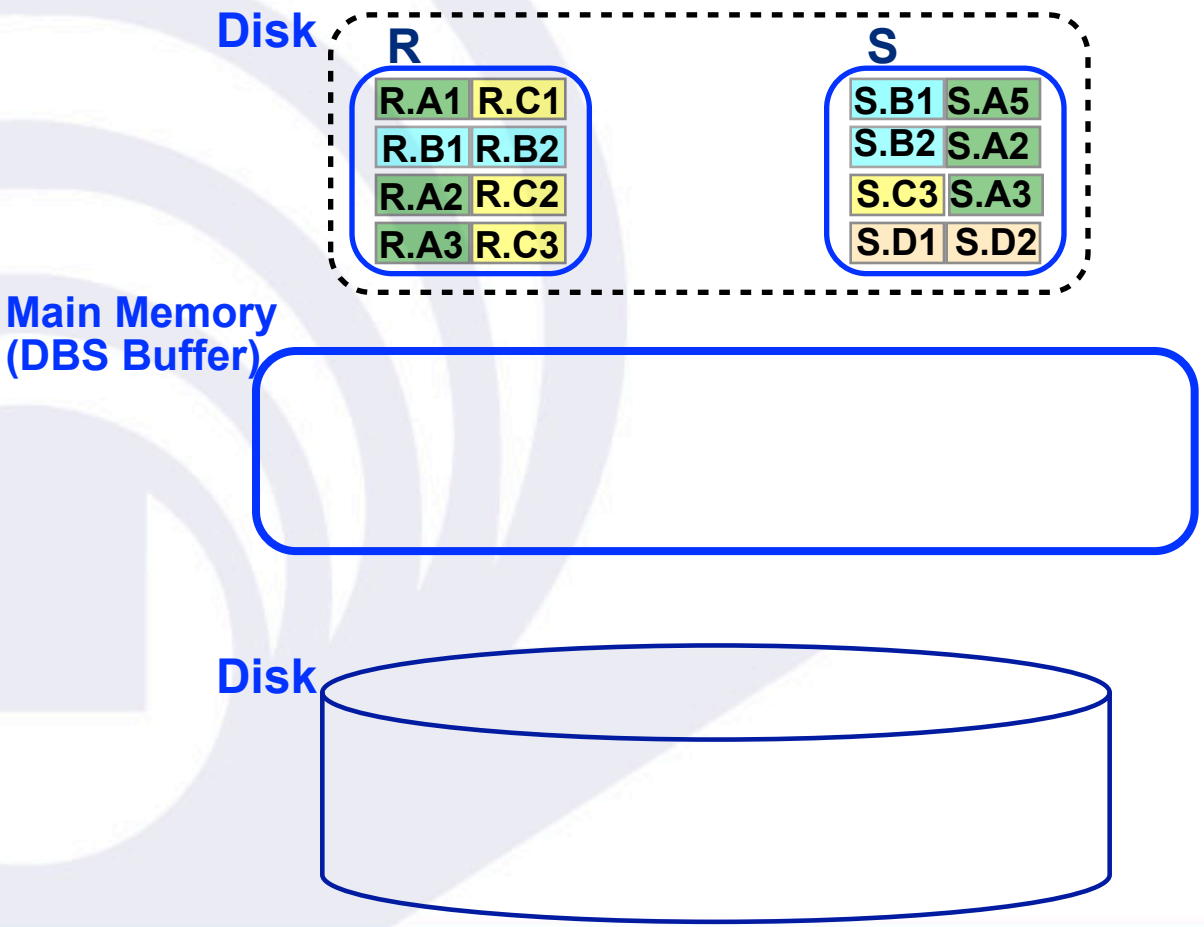
Partition R into buffer
Case Memory Overflow
Flush to Disk last referenced partition
Partition S into buffer
Case Memory Overflow
Flush to Disk partition with the same address of R's partition already in Disk
Execute nested-loop join on each pair of in-memory partitions
Execute nested-loop join on each pair of in-disk partitions

Result

R.A2	S.A2
R.A3	S.A3
R.B1	S.B1
R.B2	S.B2
R.C3	S.C3

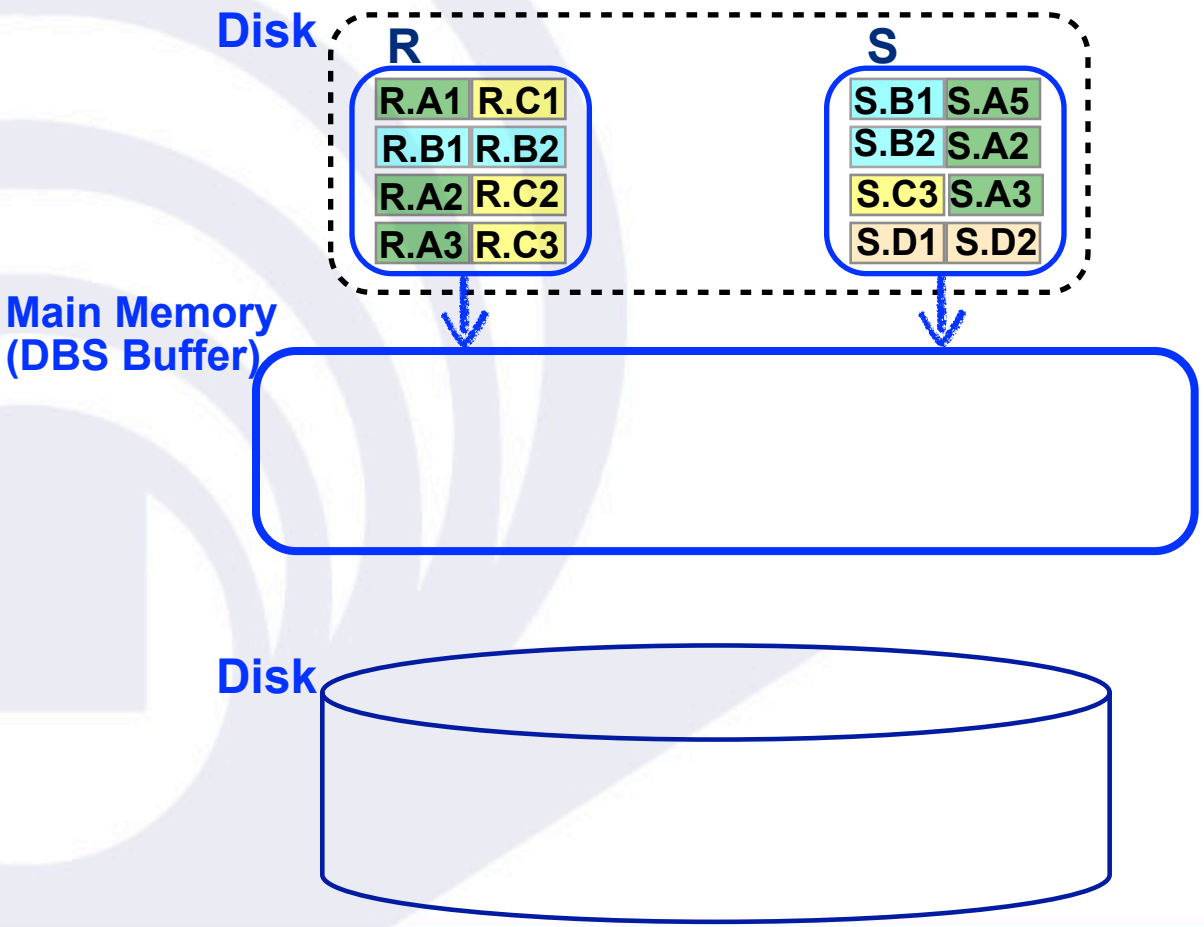
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



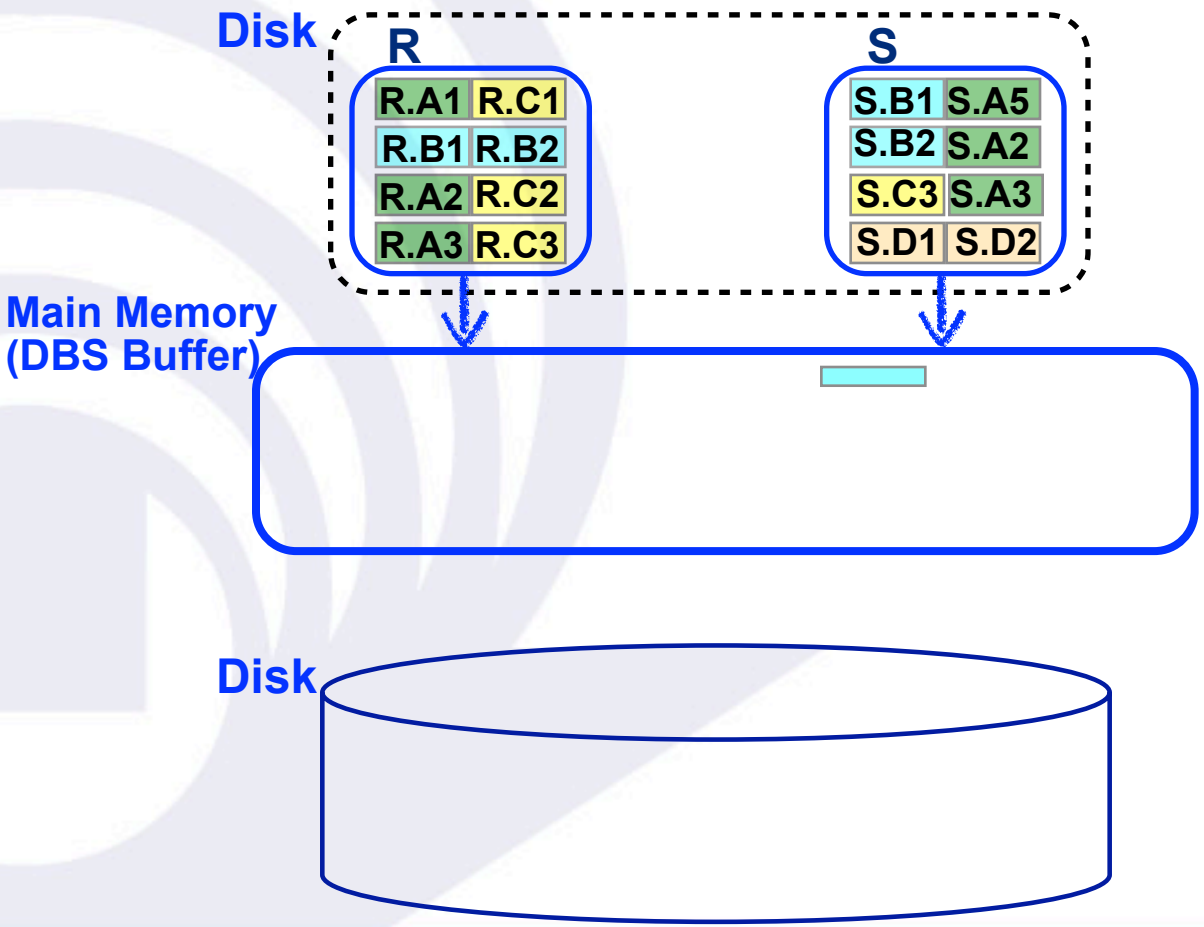
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



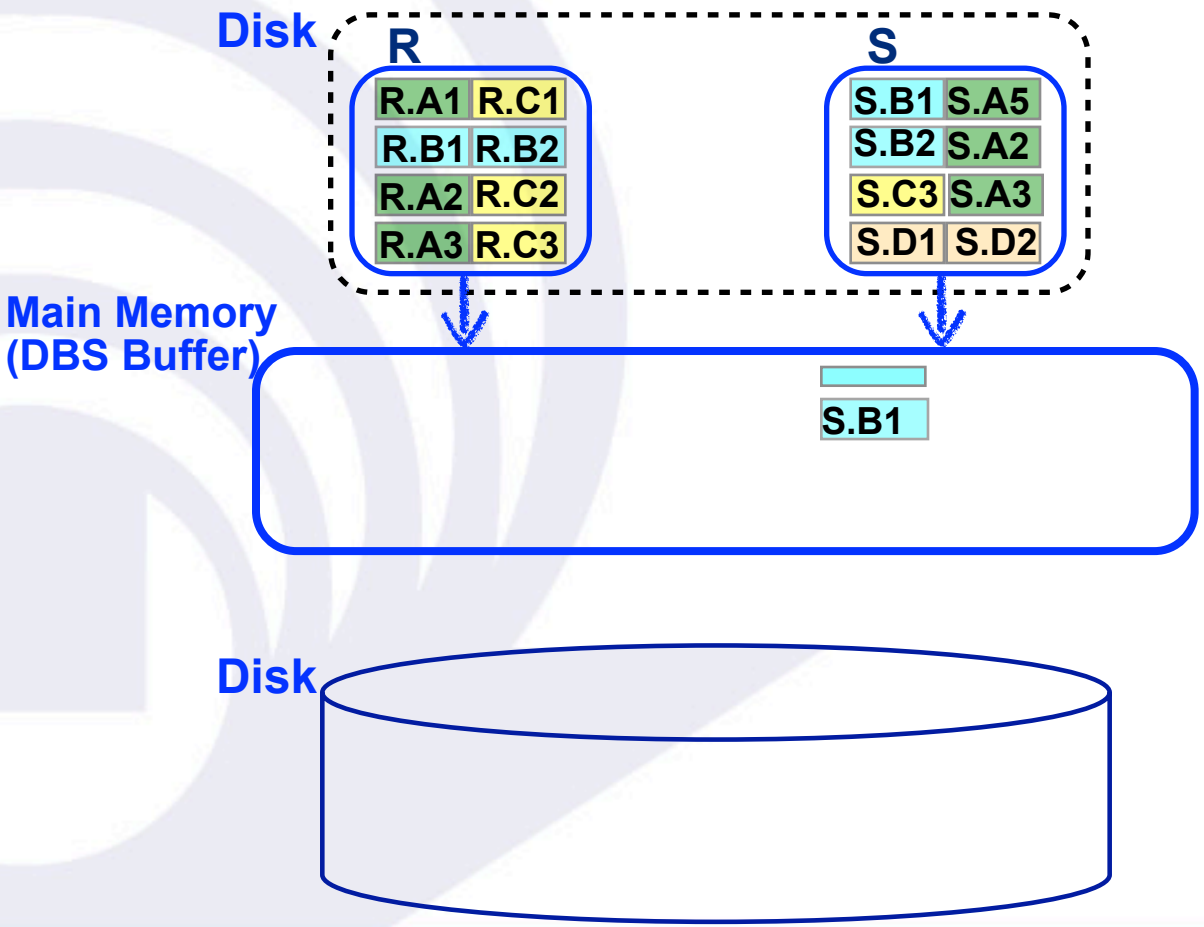
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



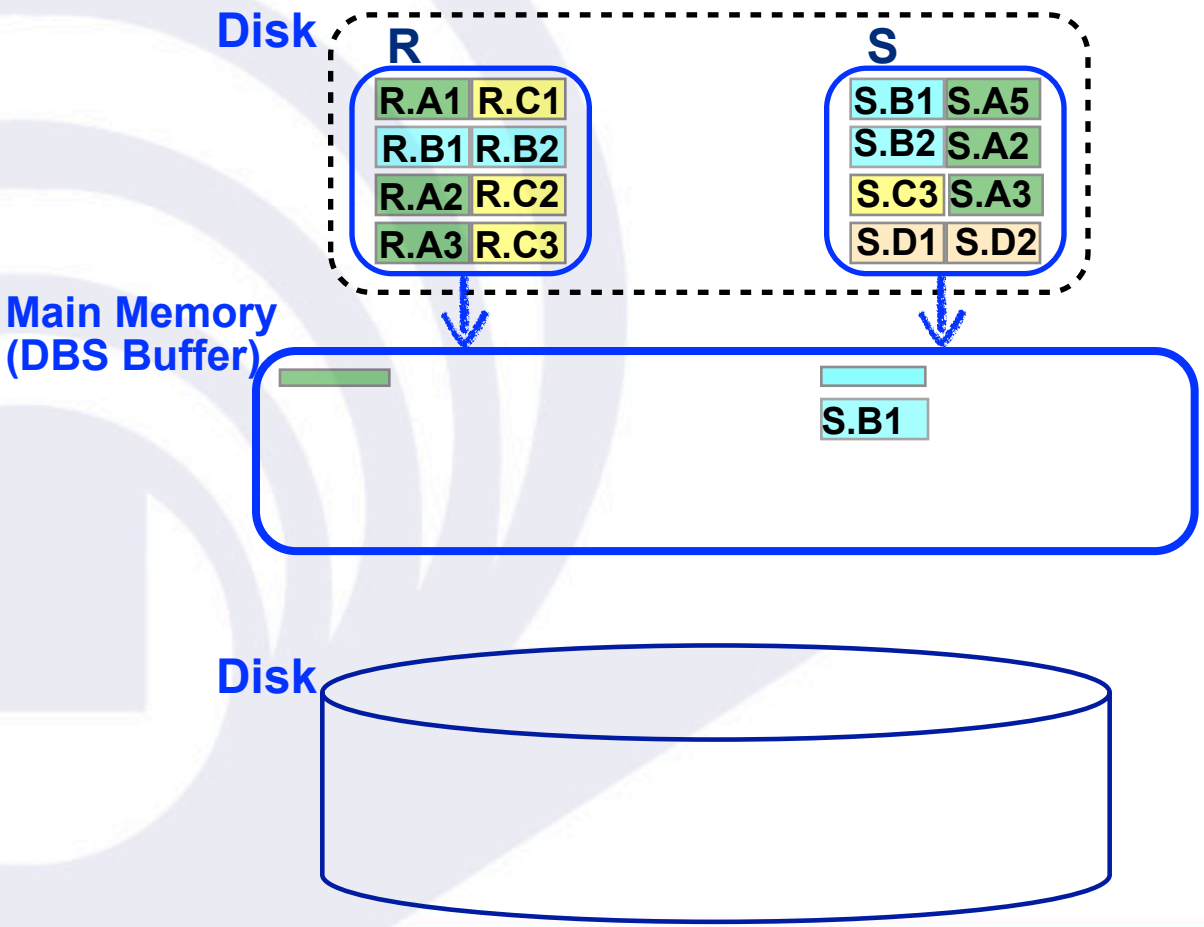
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



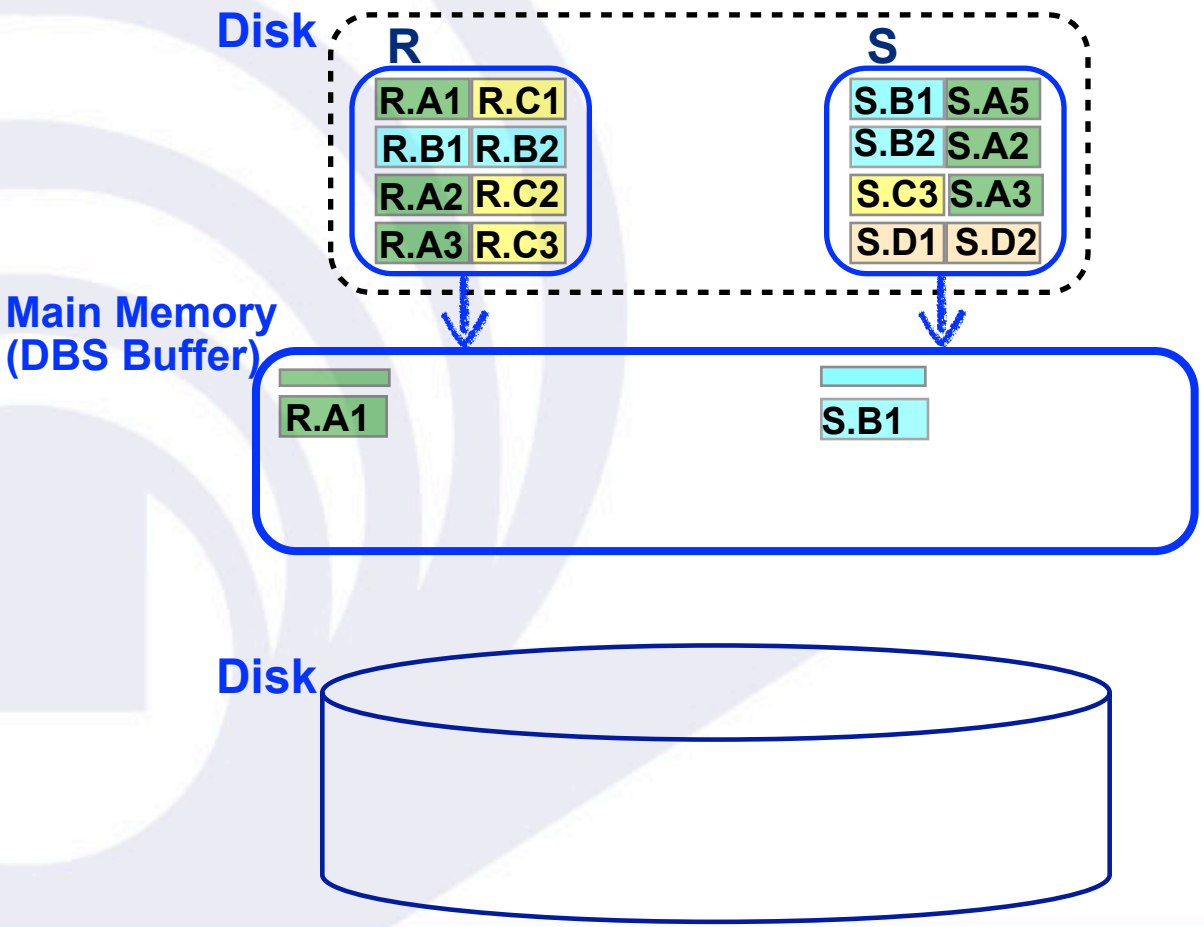
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



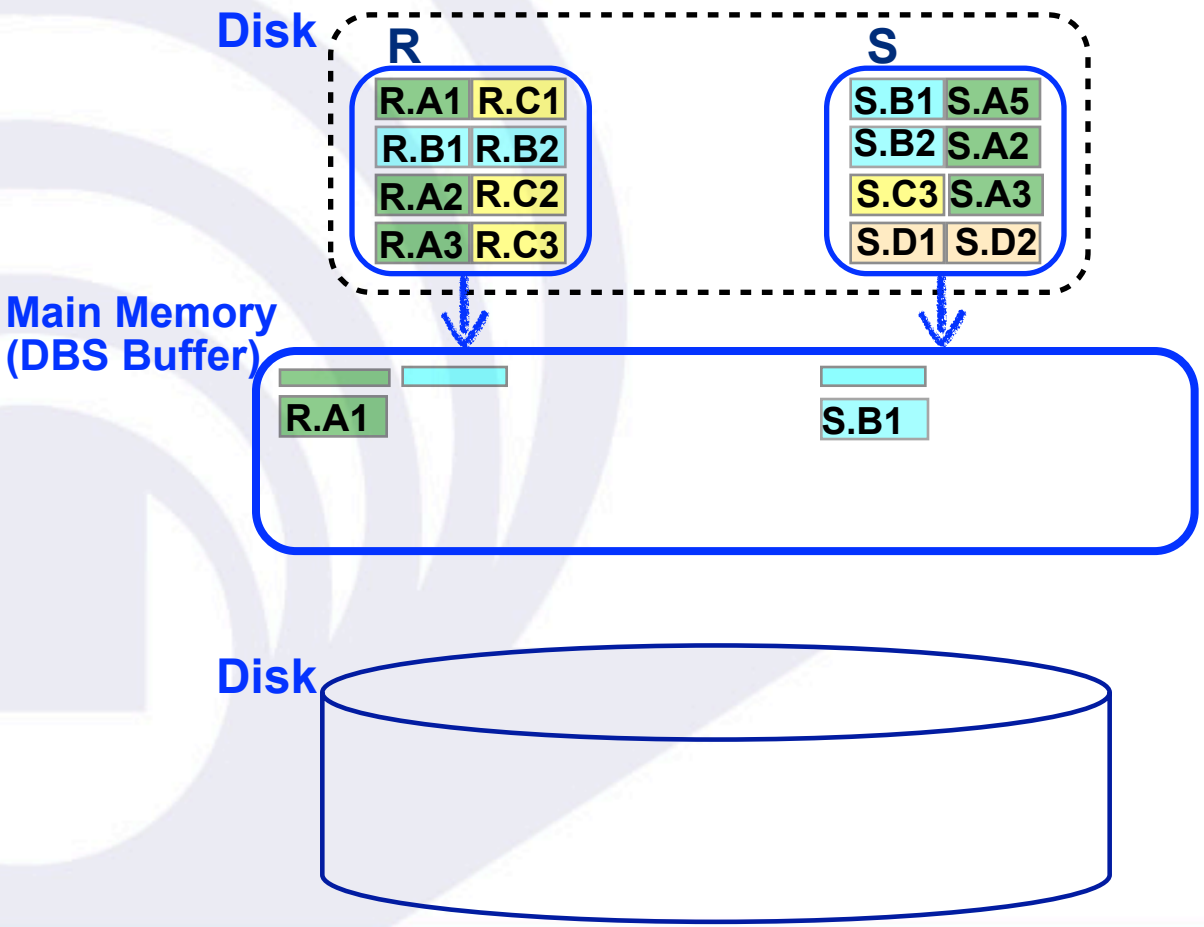
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



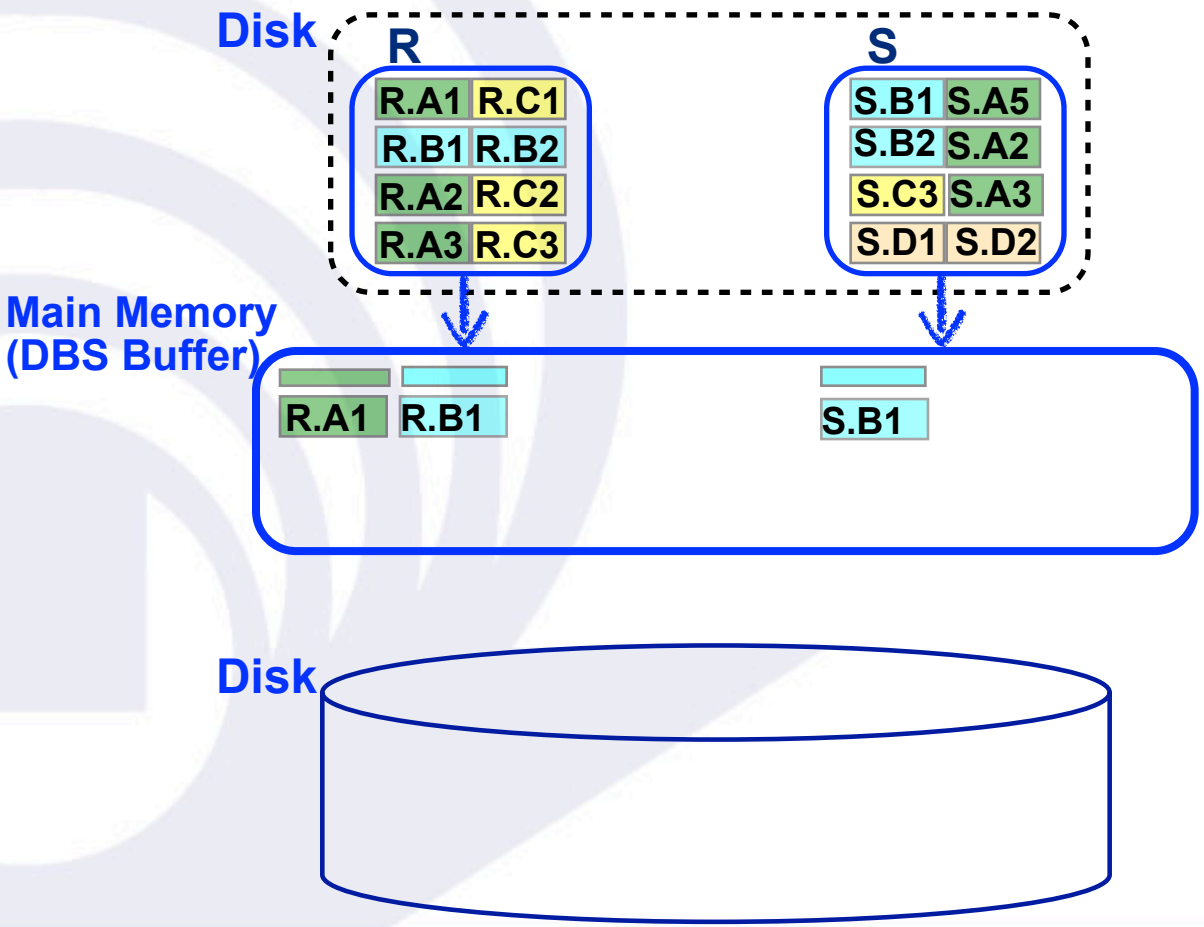
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



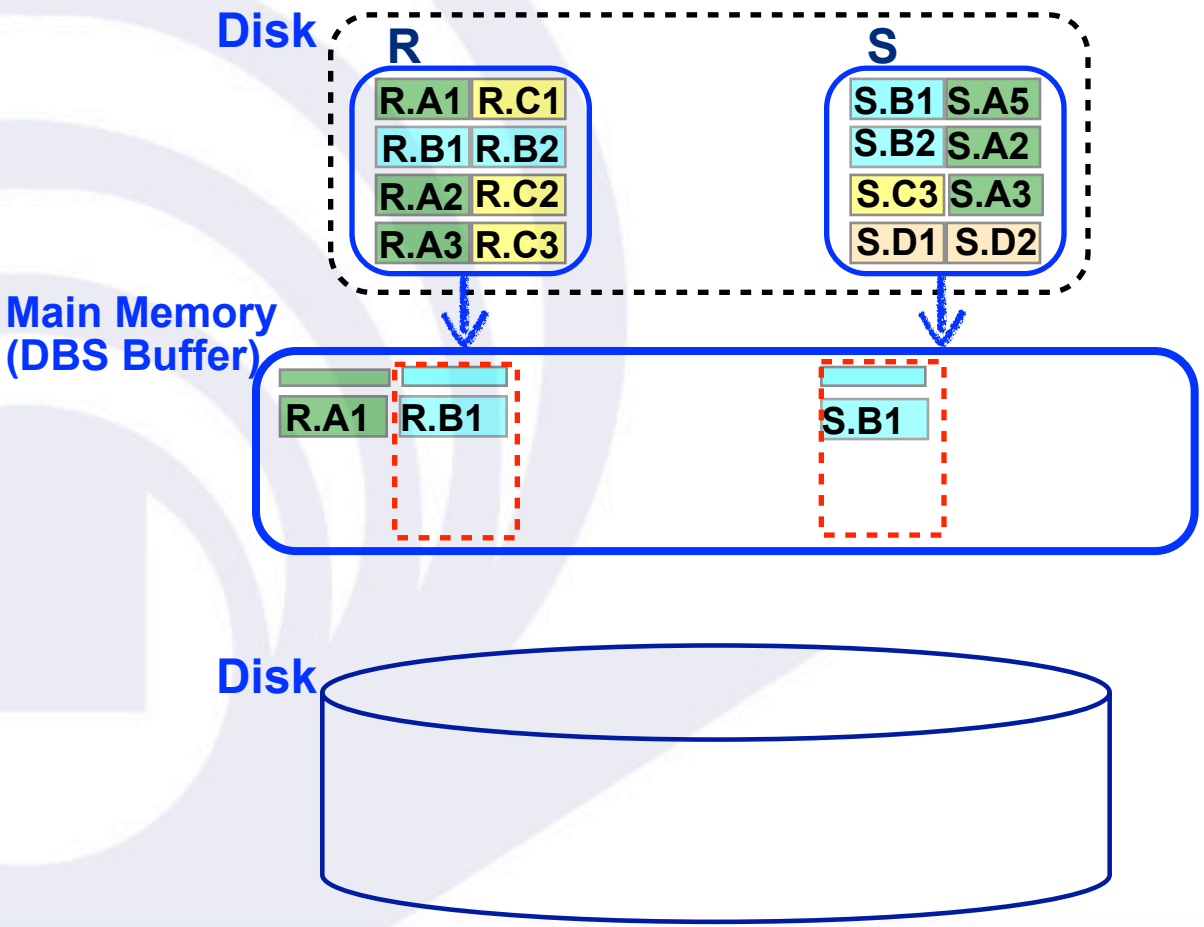
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



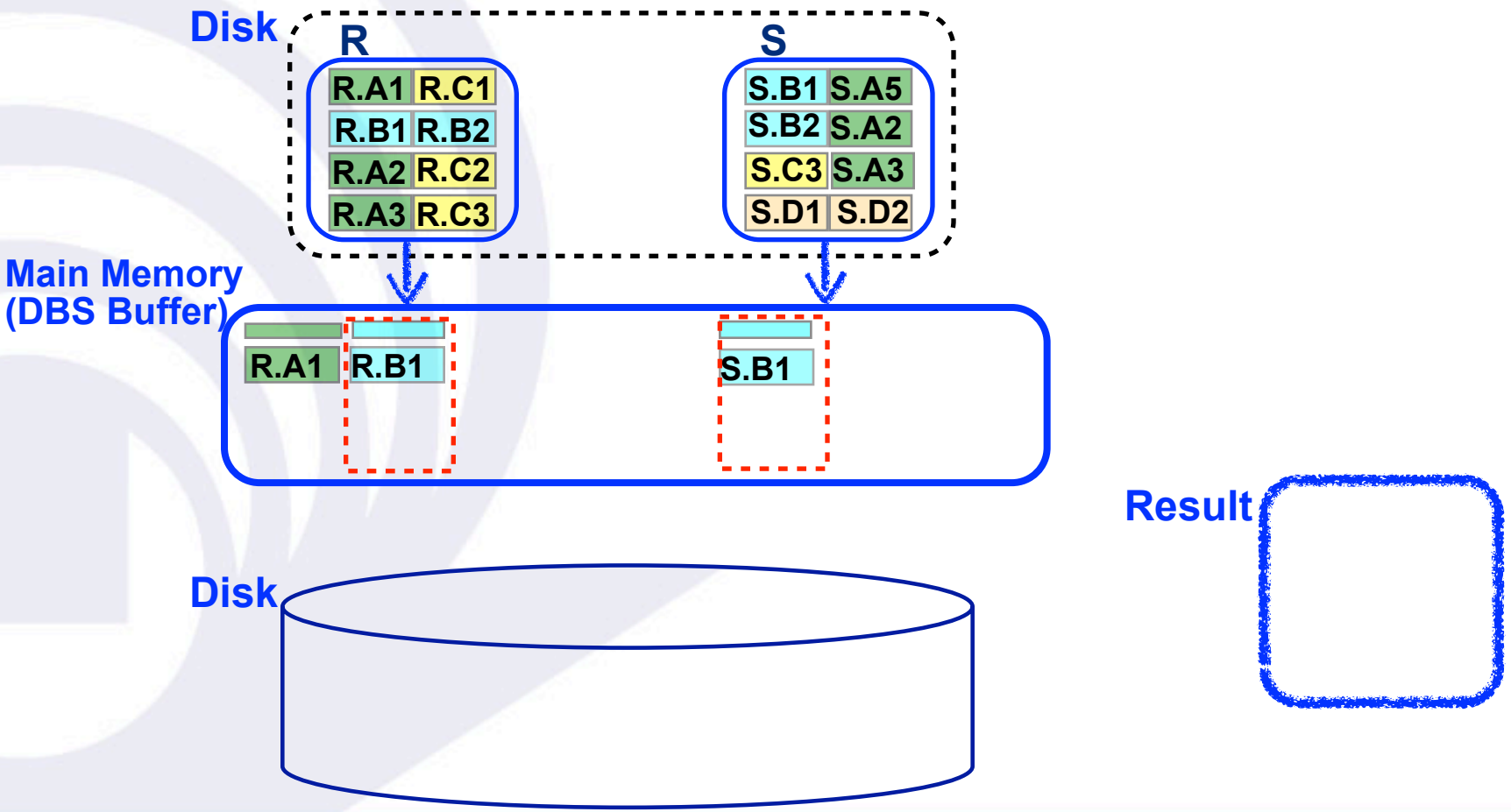
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



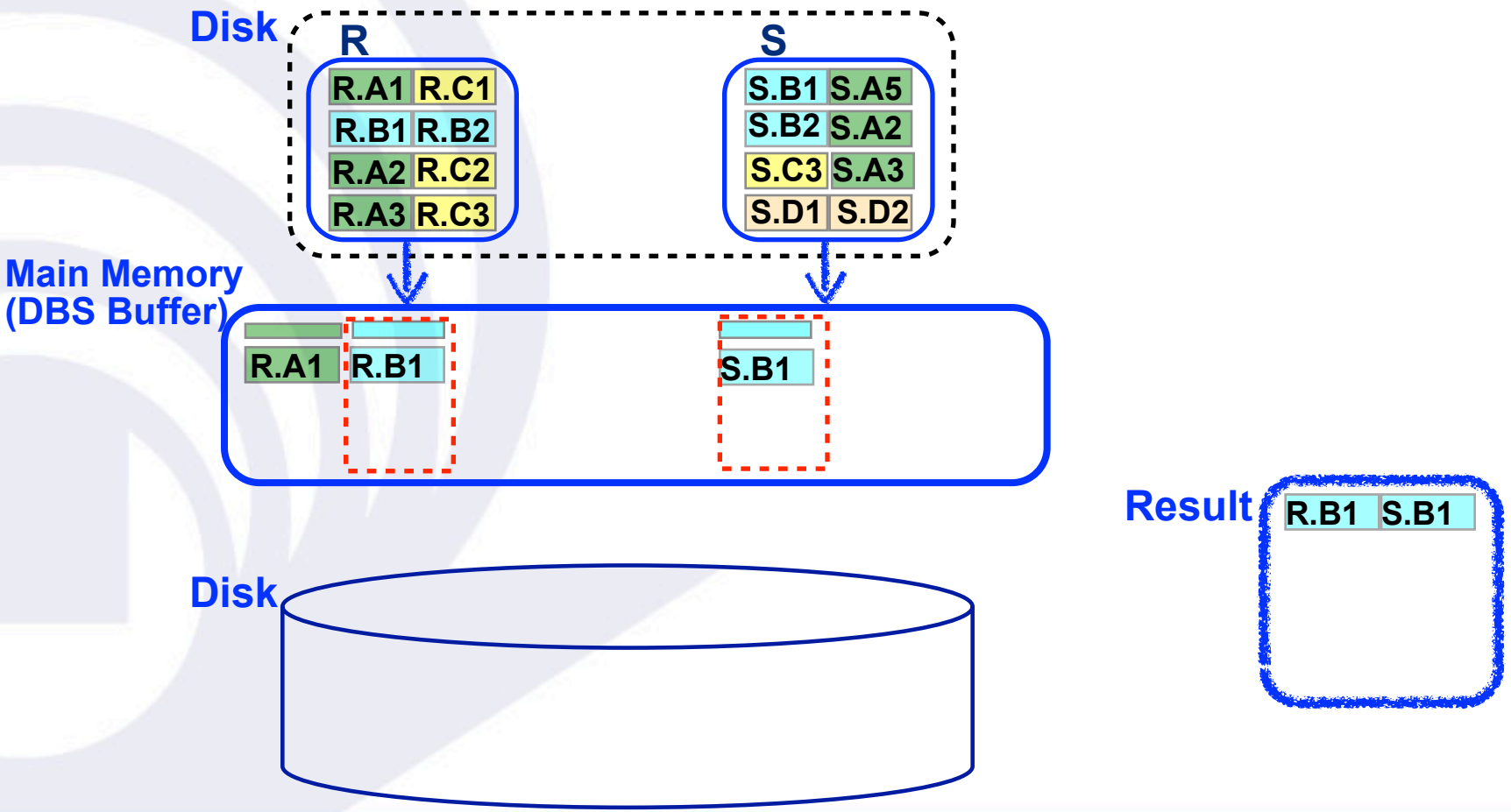
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



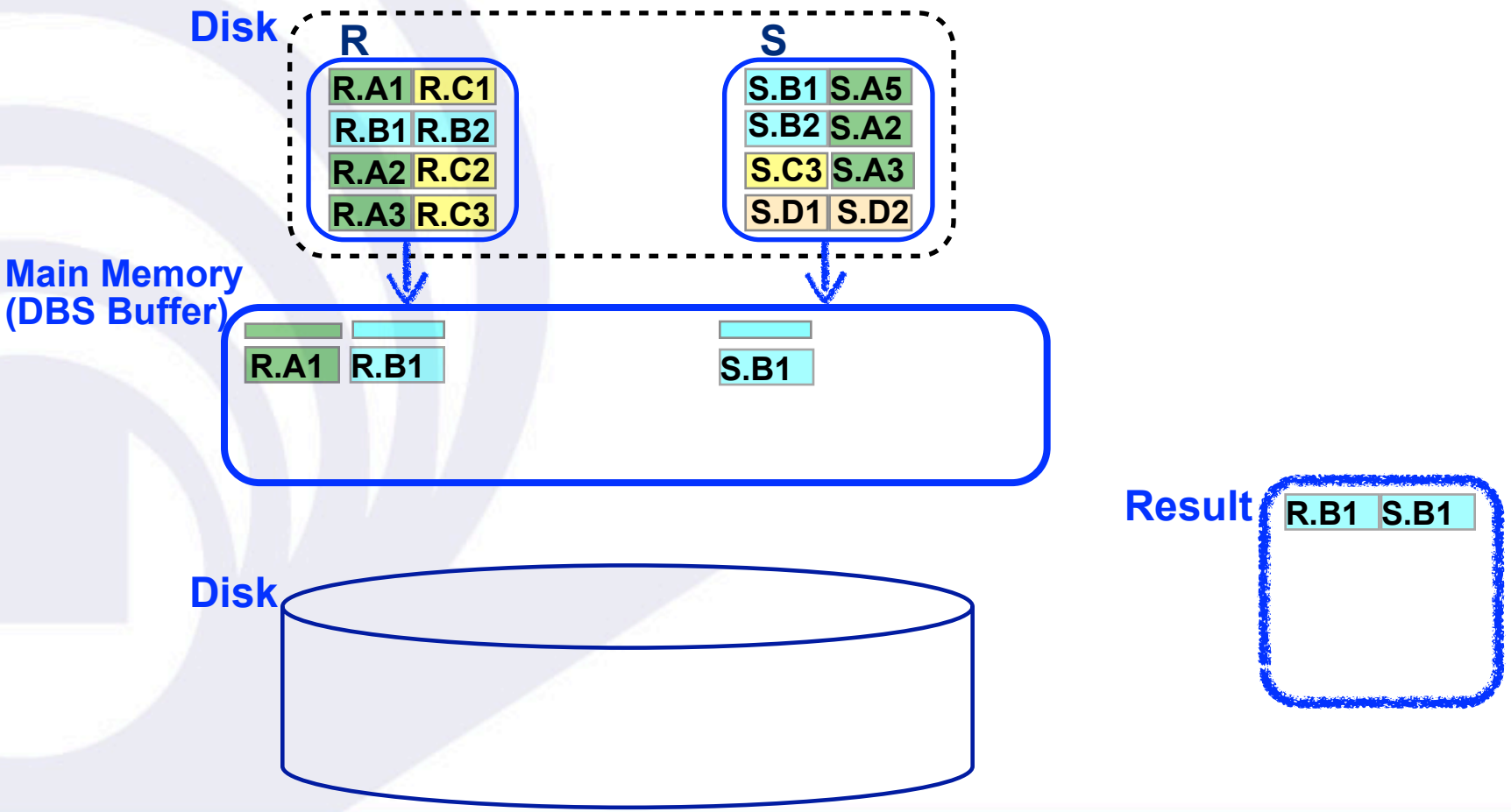
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



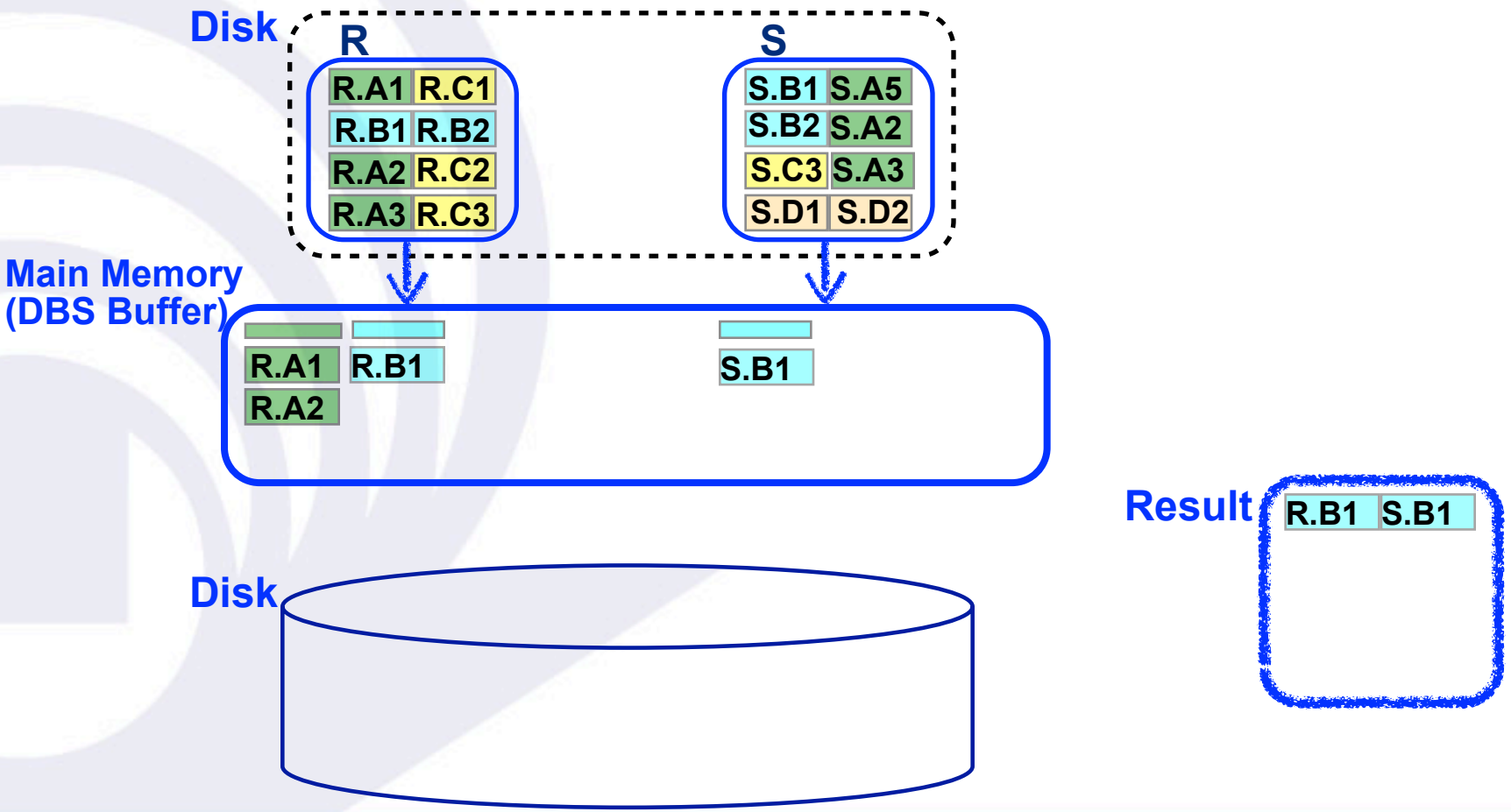
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



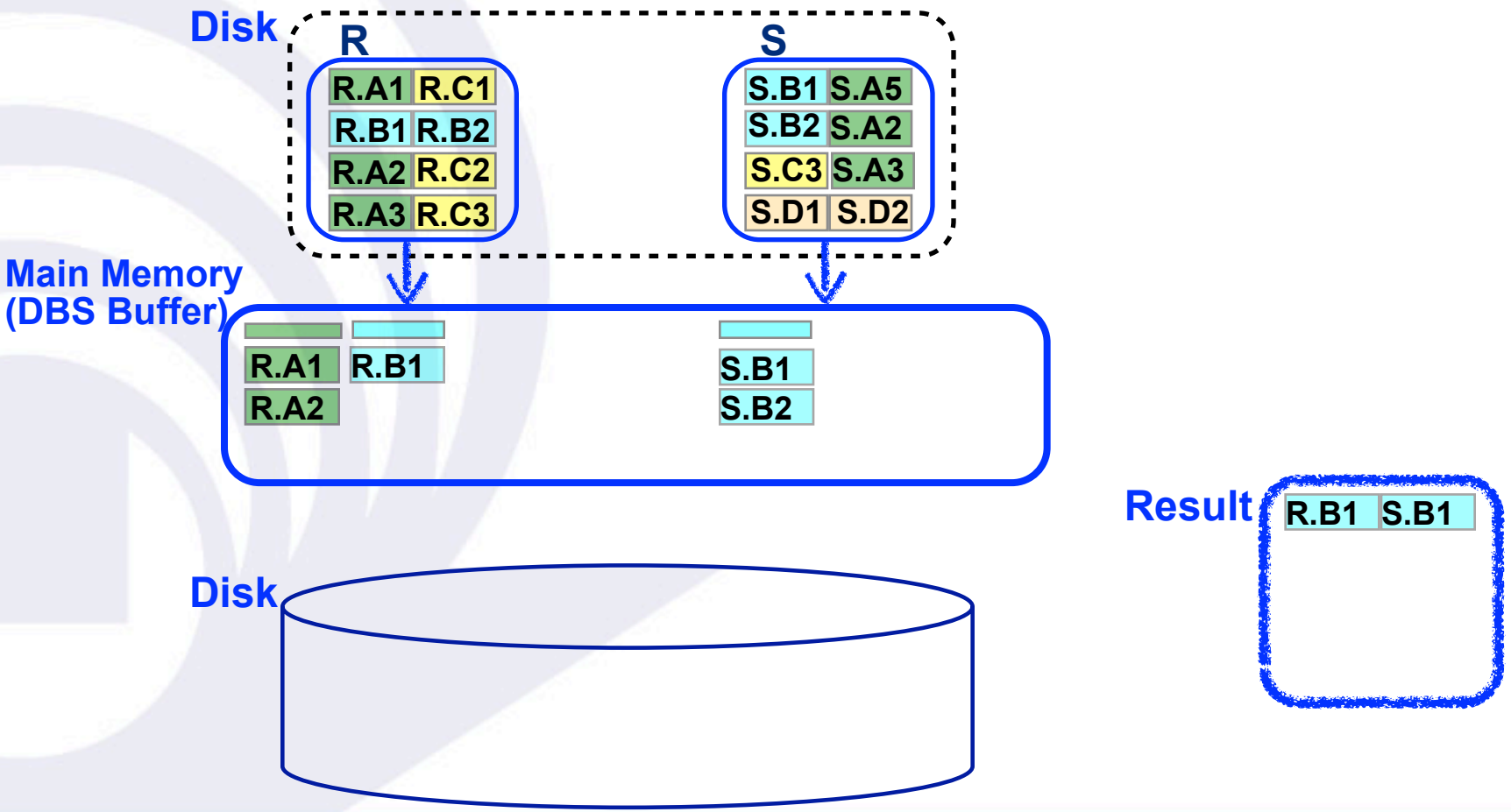
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



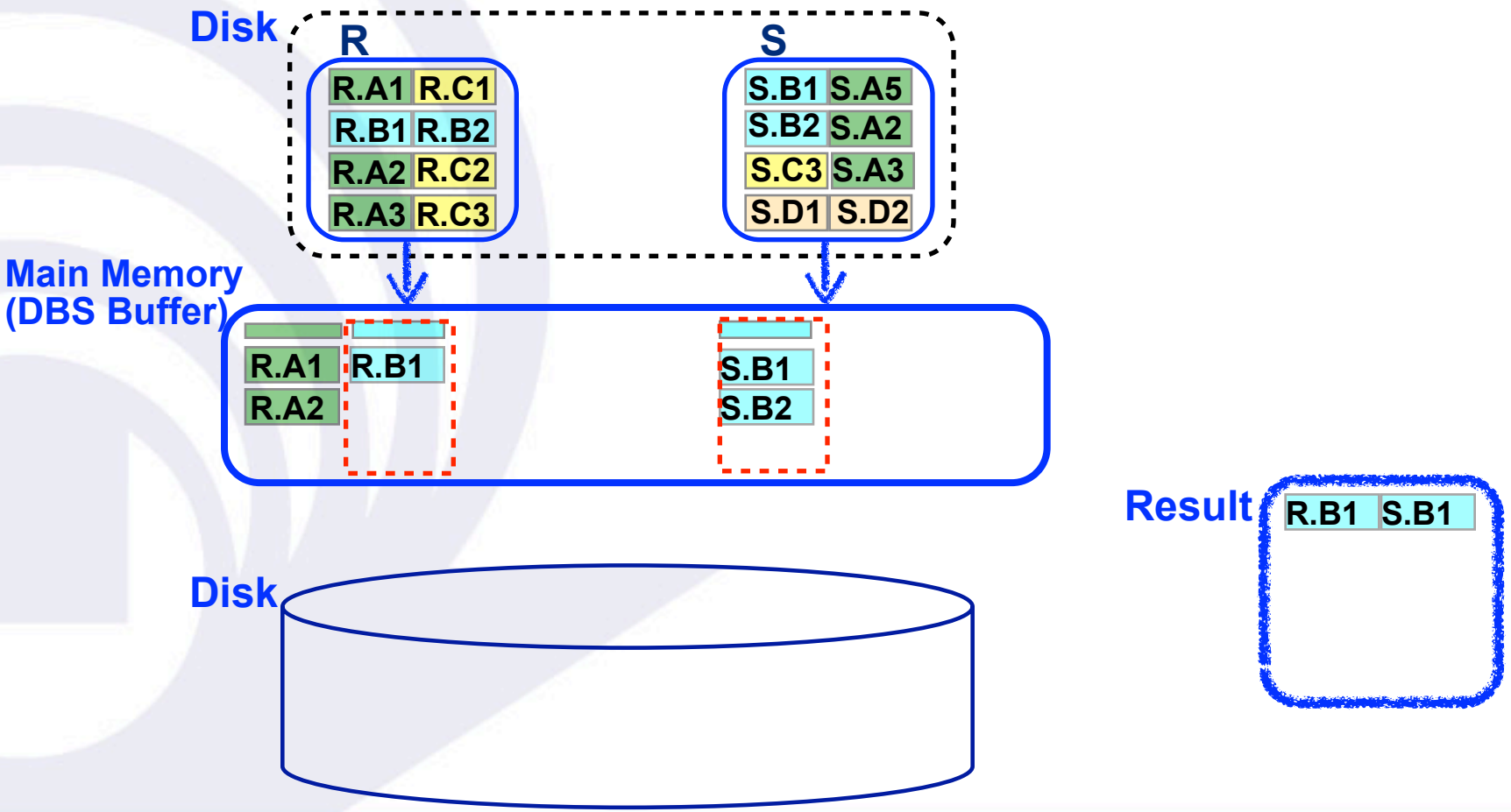
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



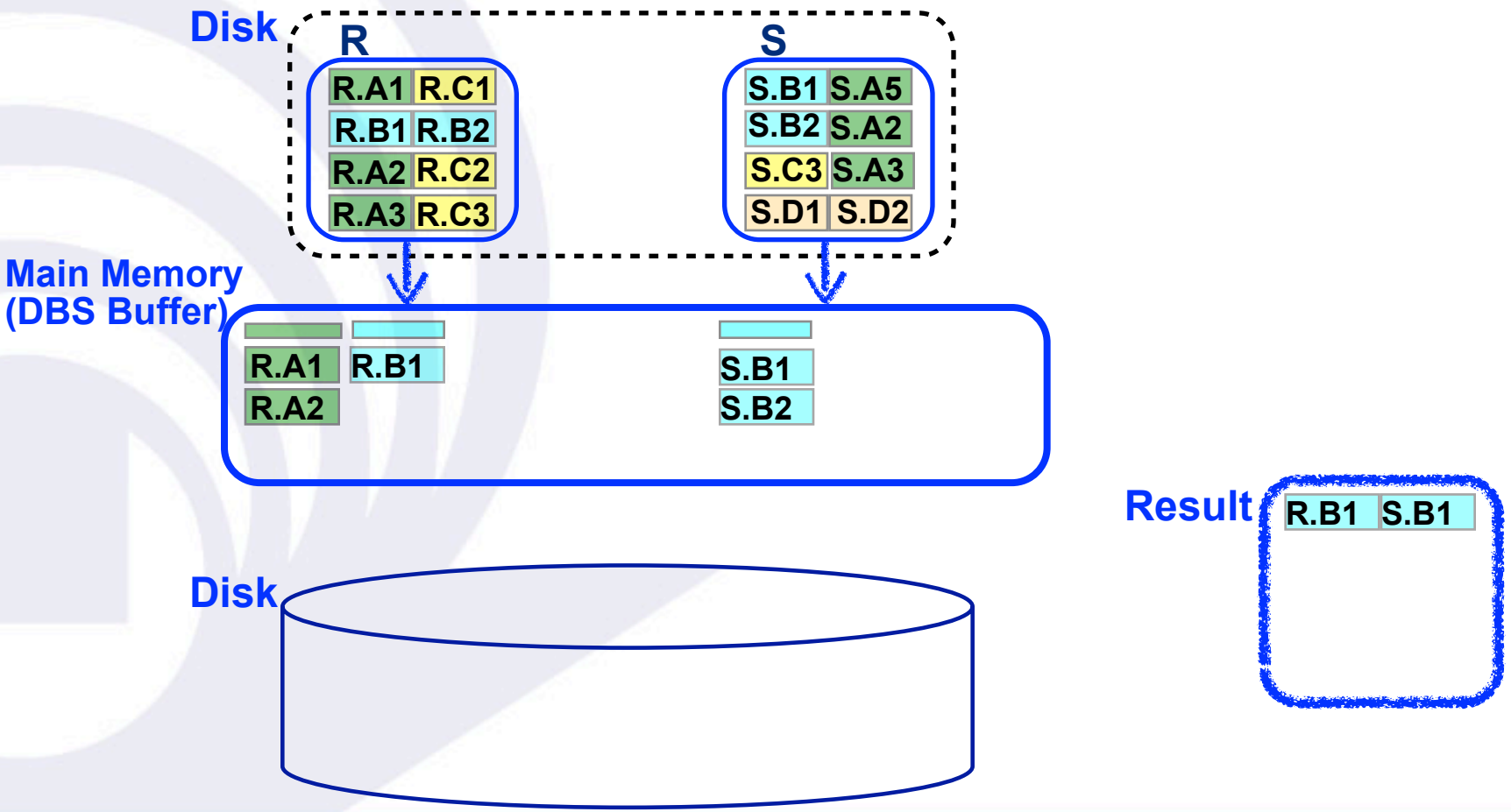
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



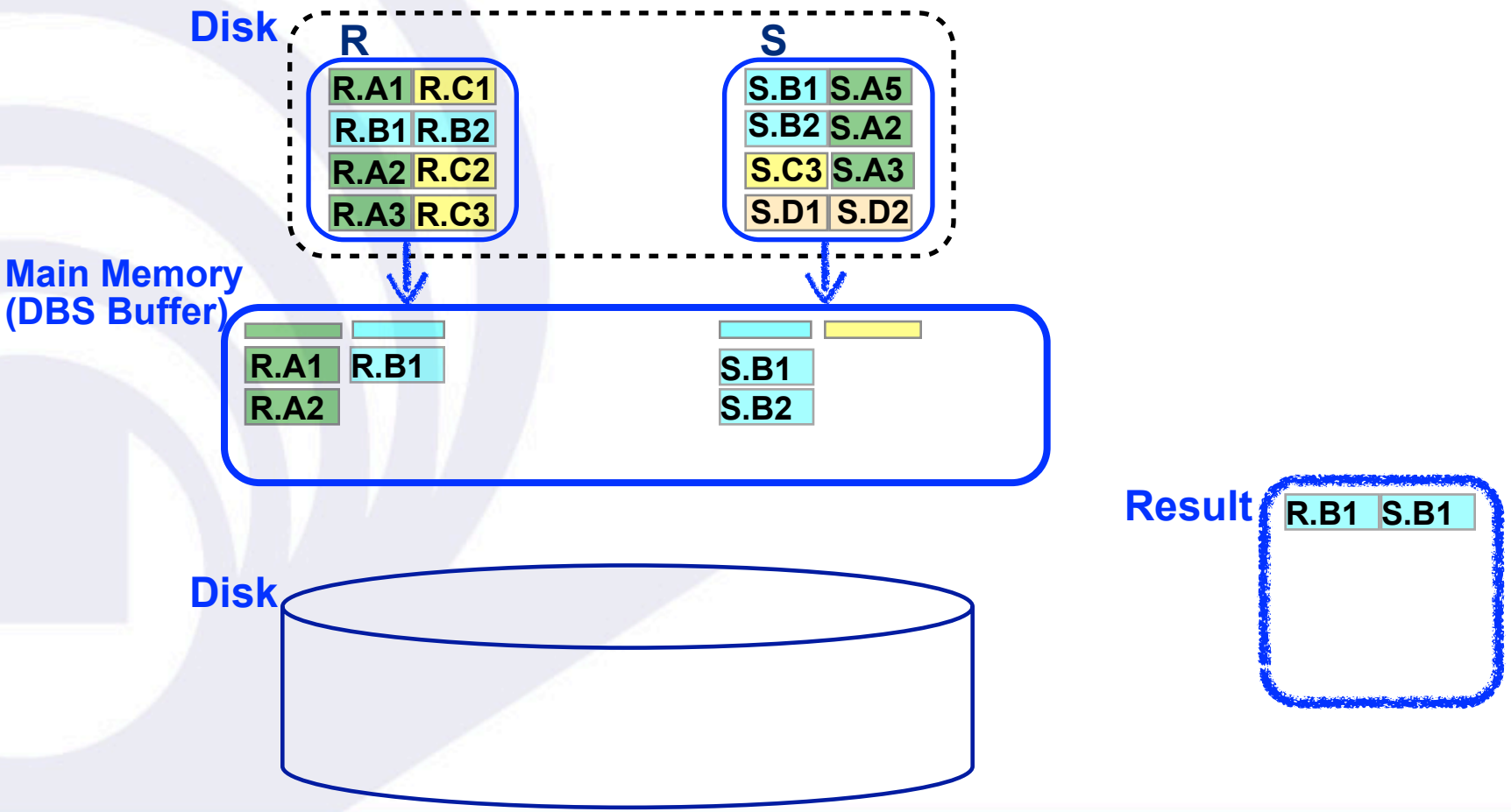
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



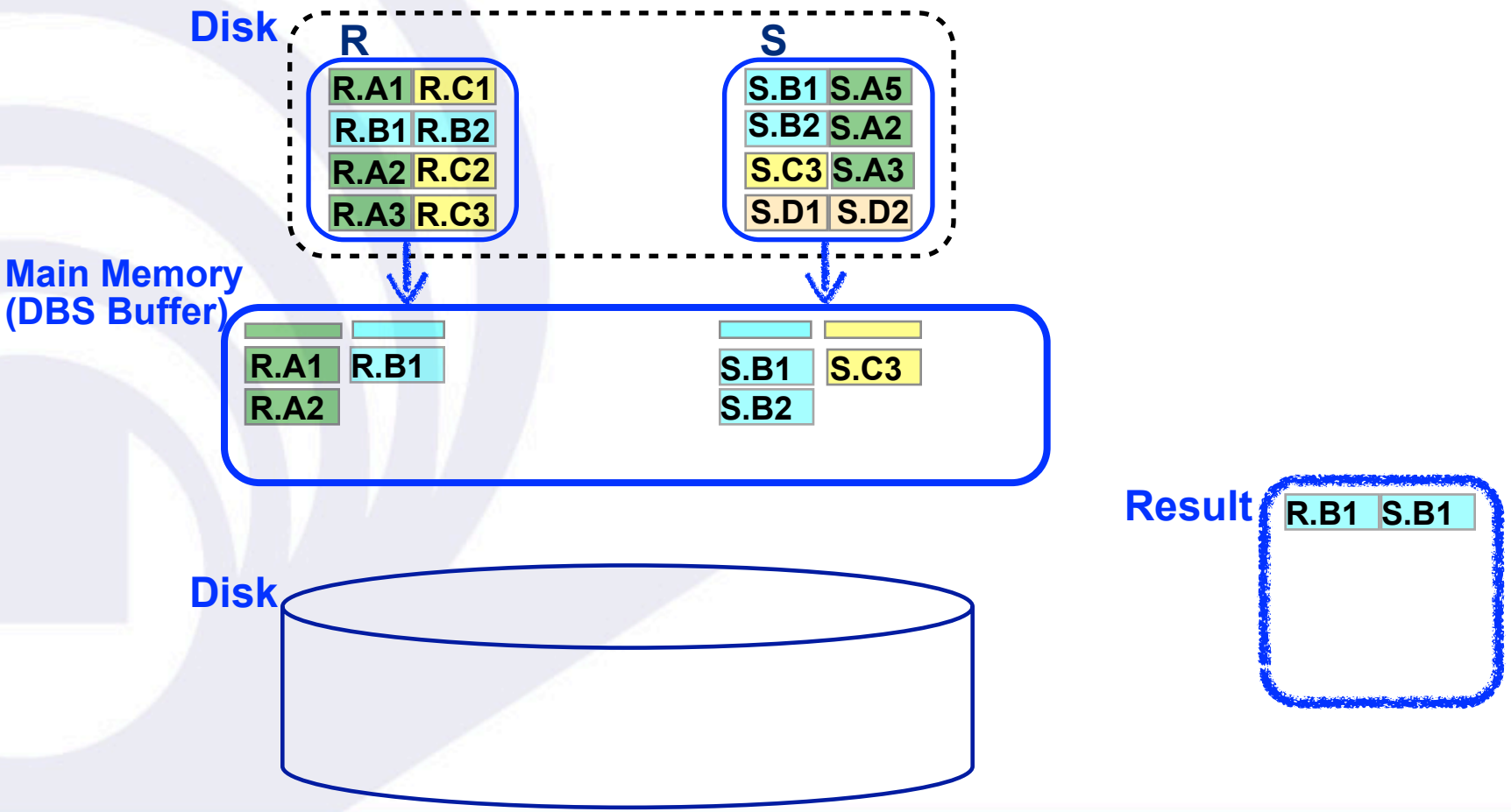
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



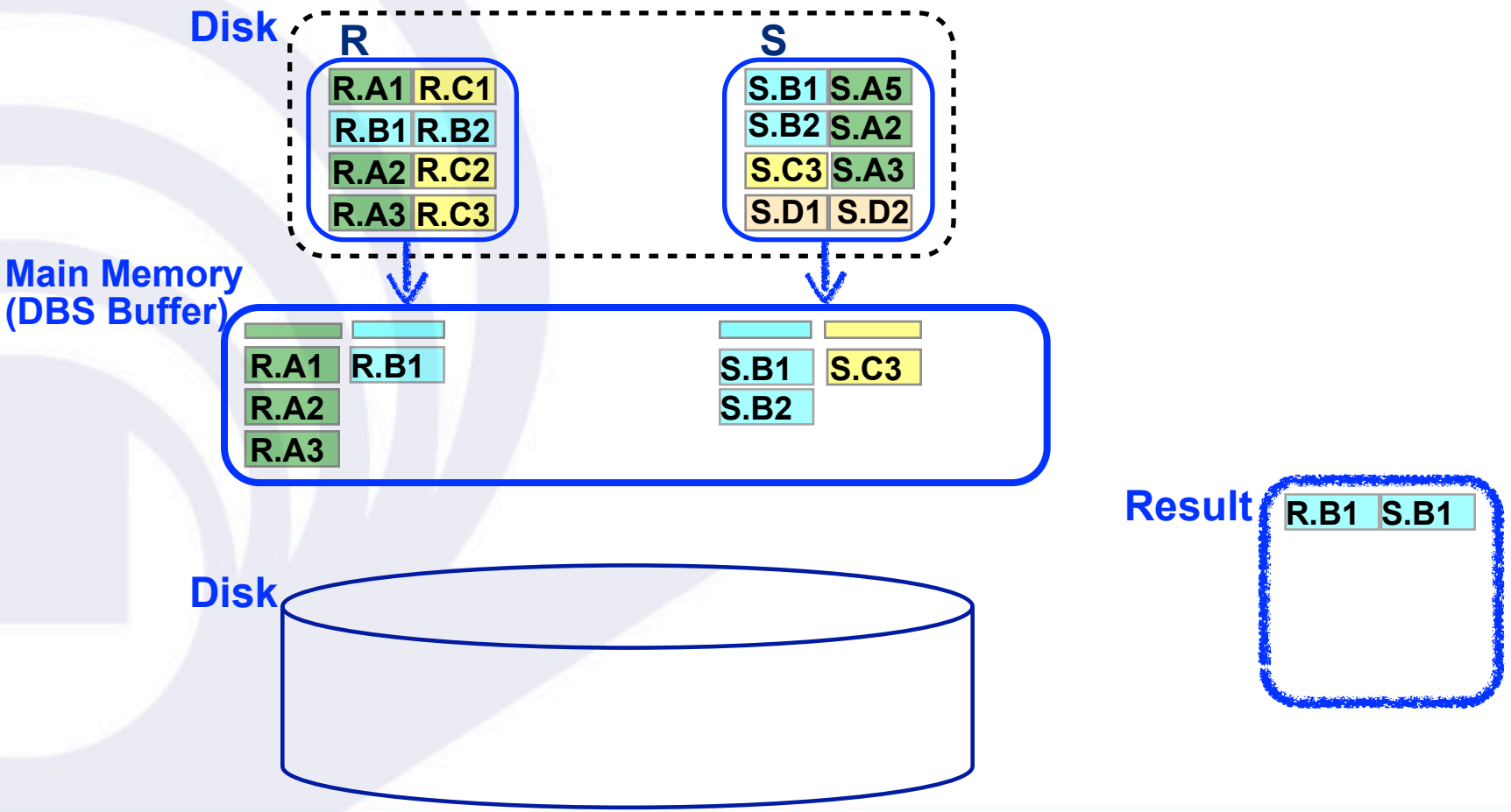
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



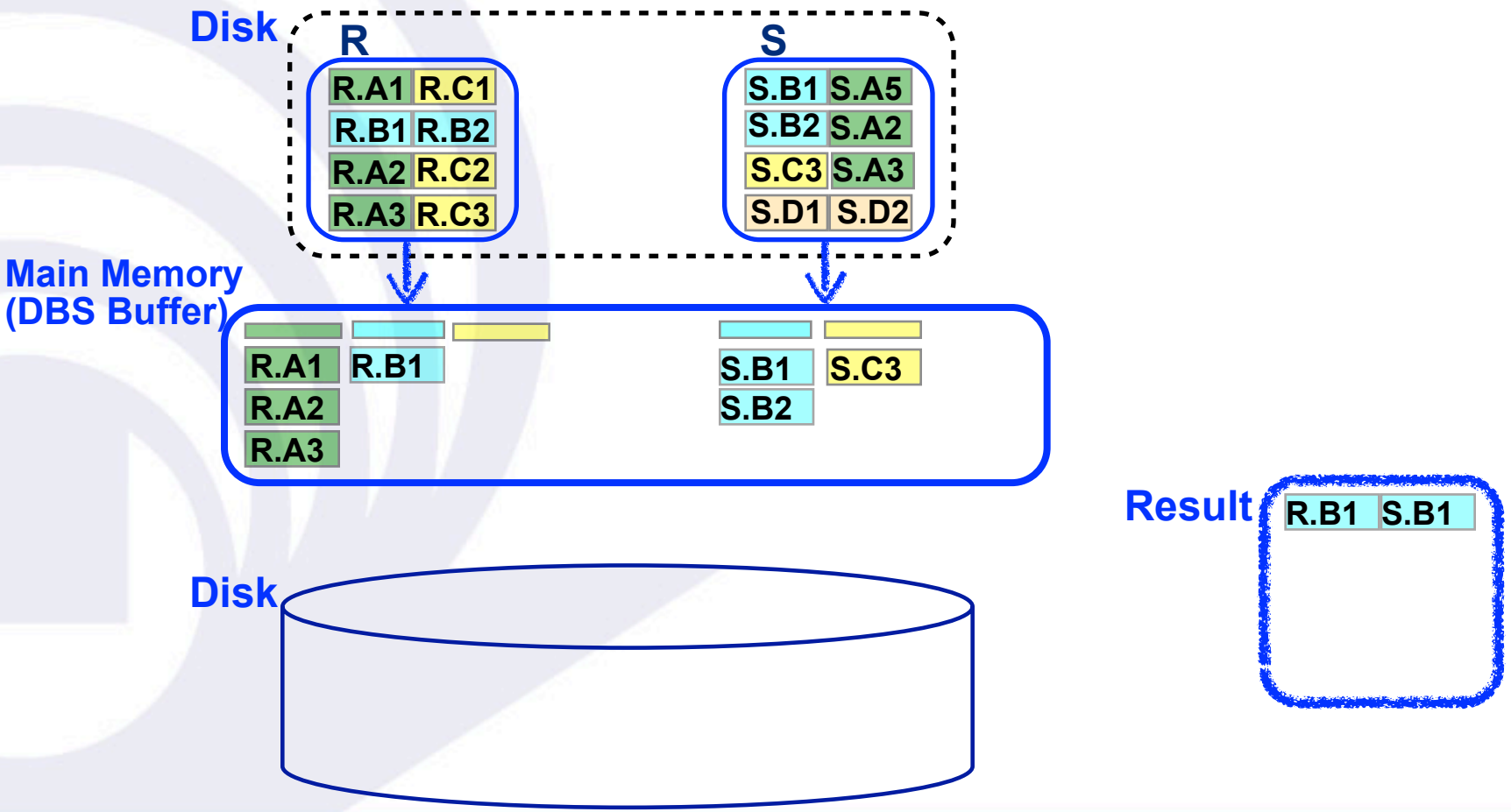
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



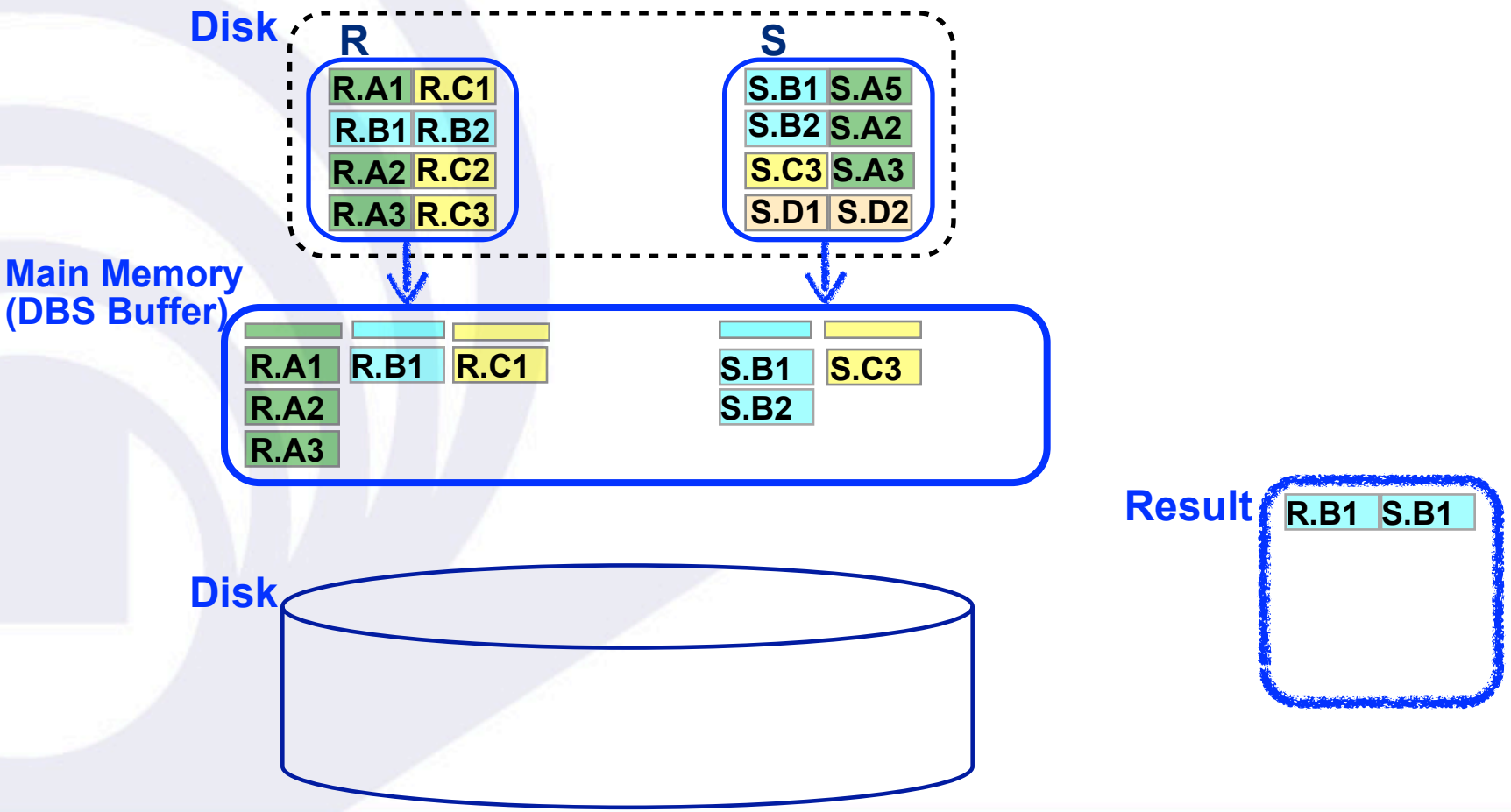
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



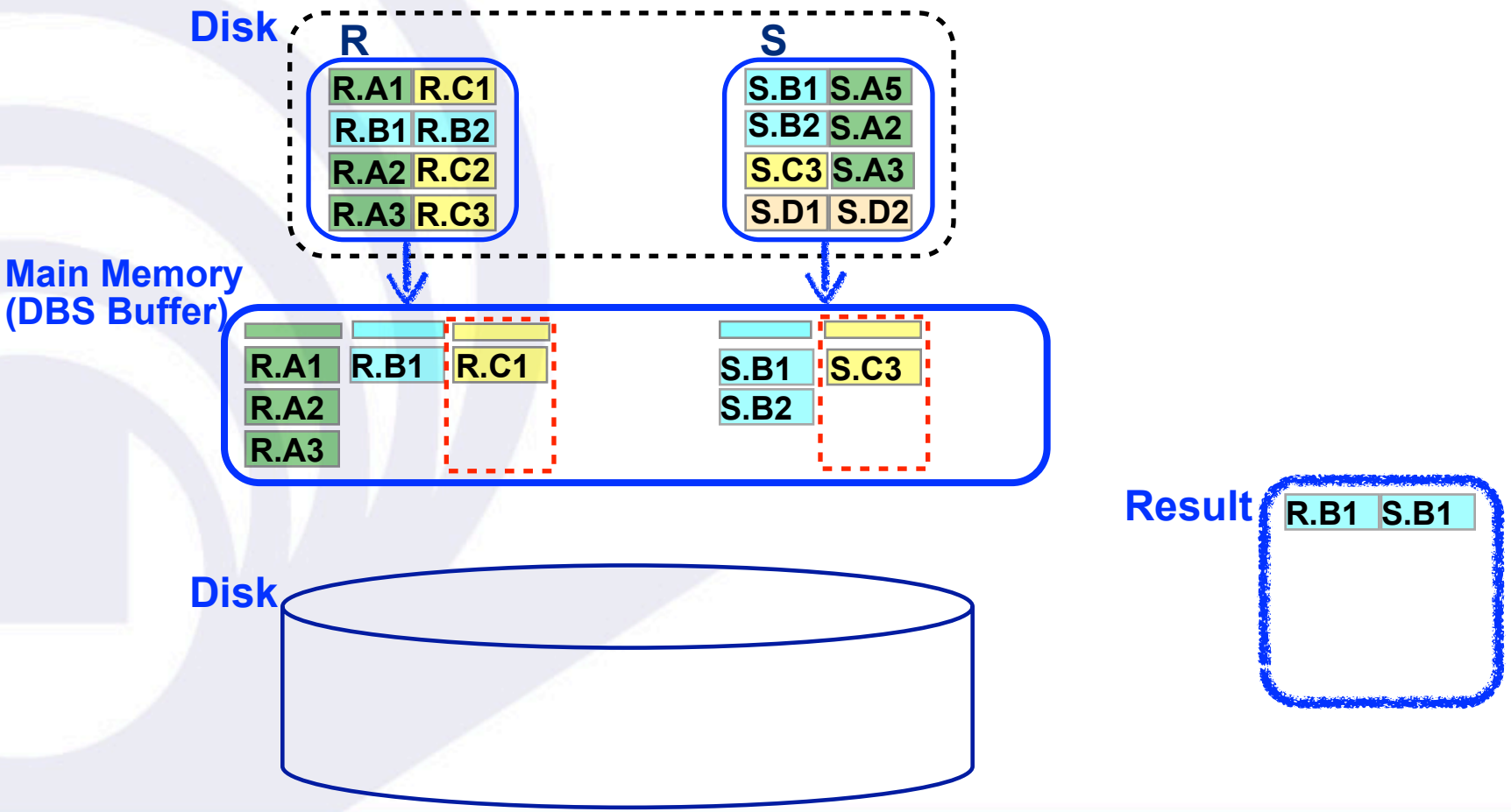
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



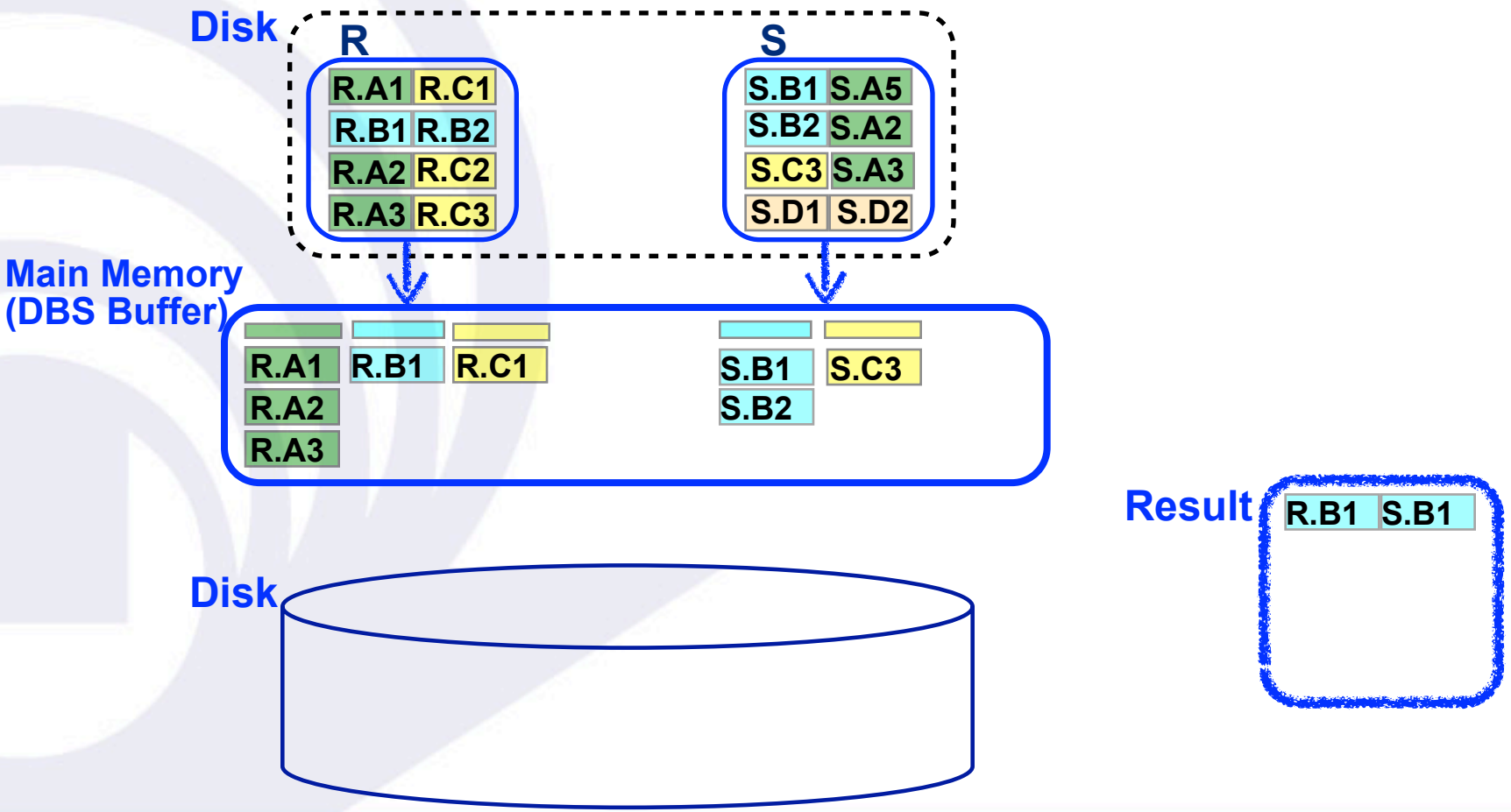
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



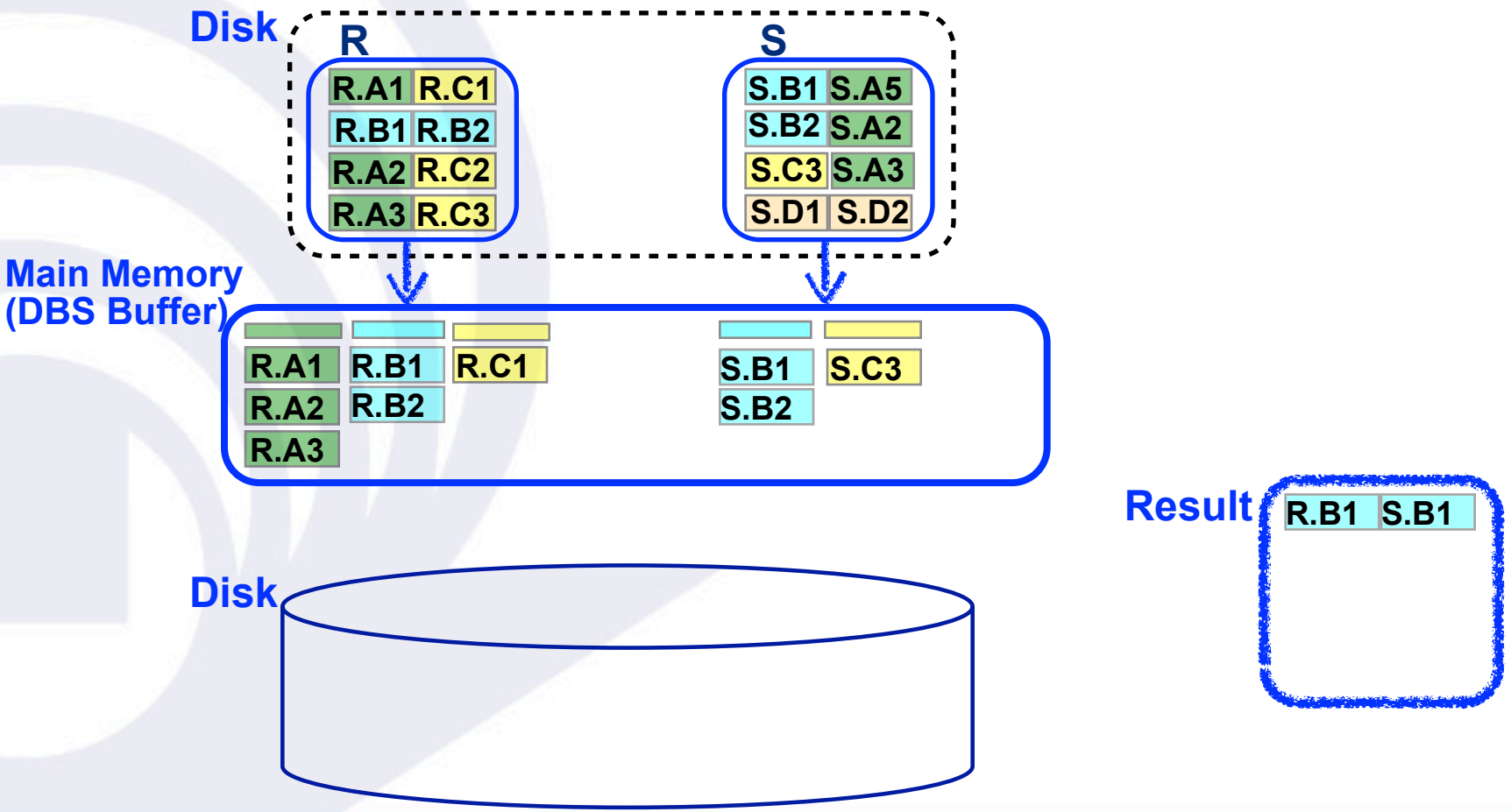
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



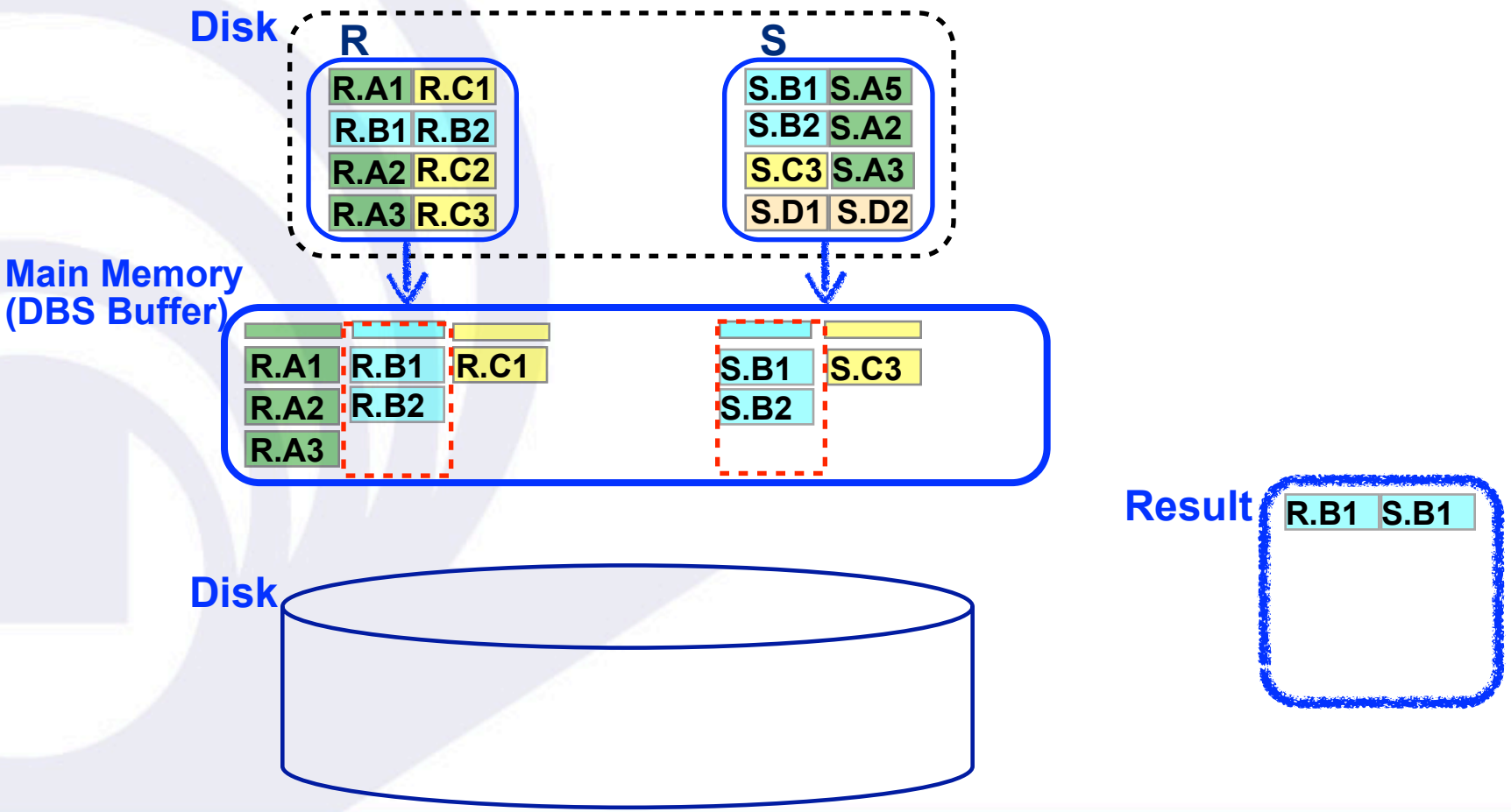
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



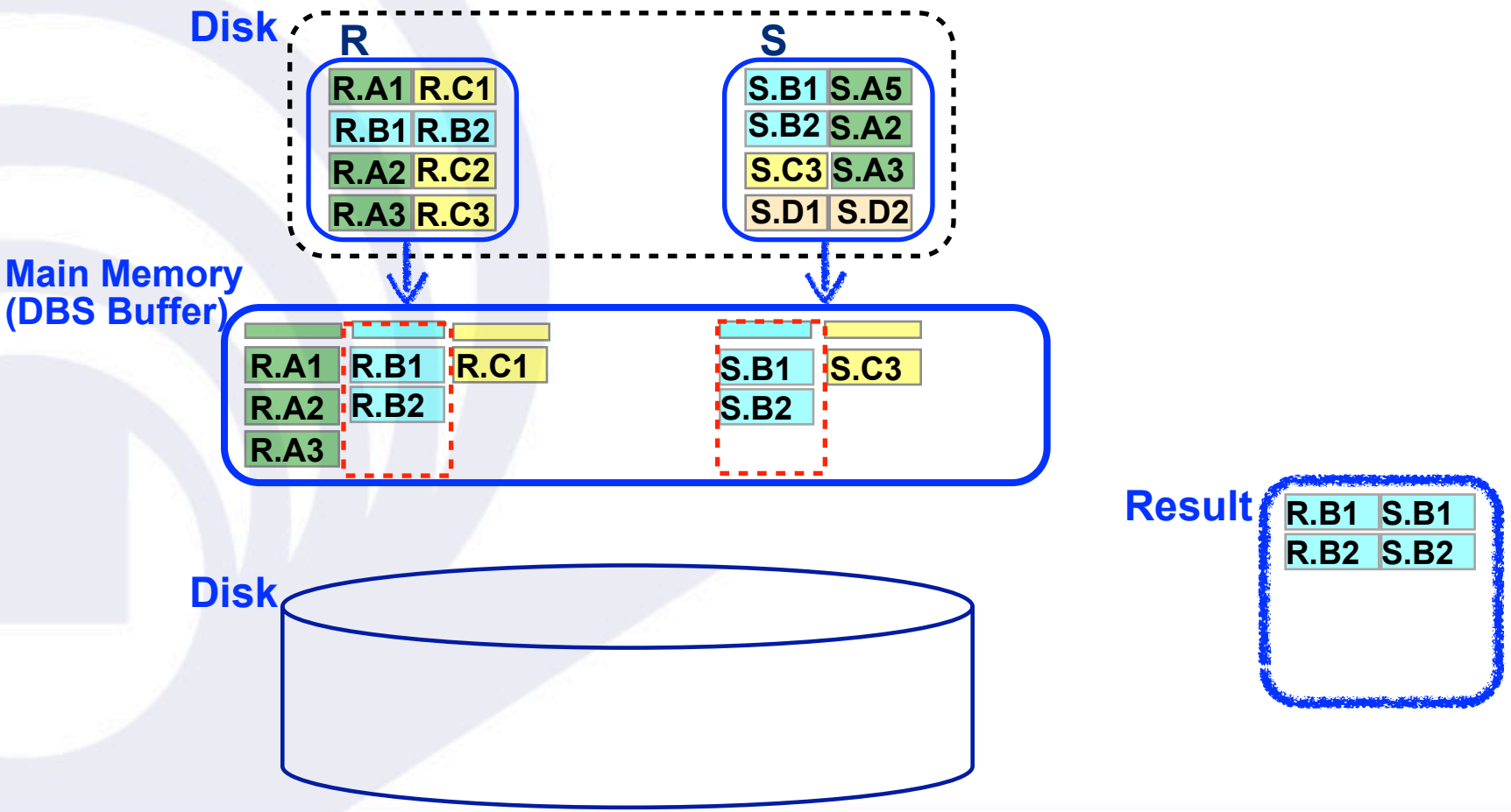
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



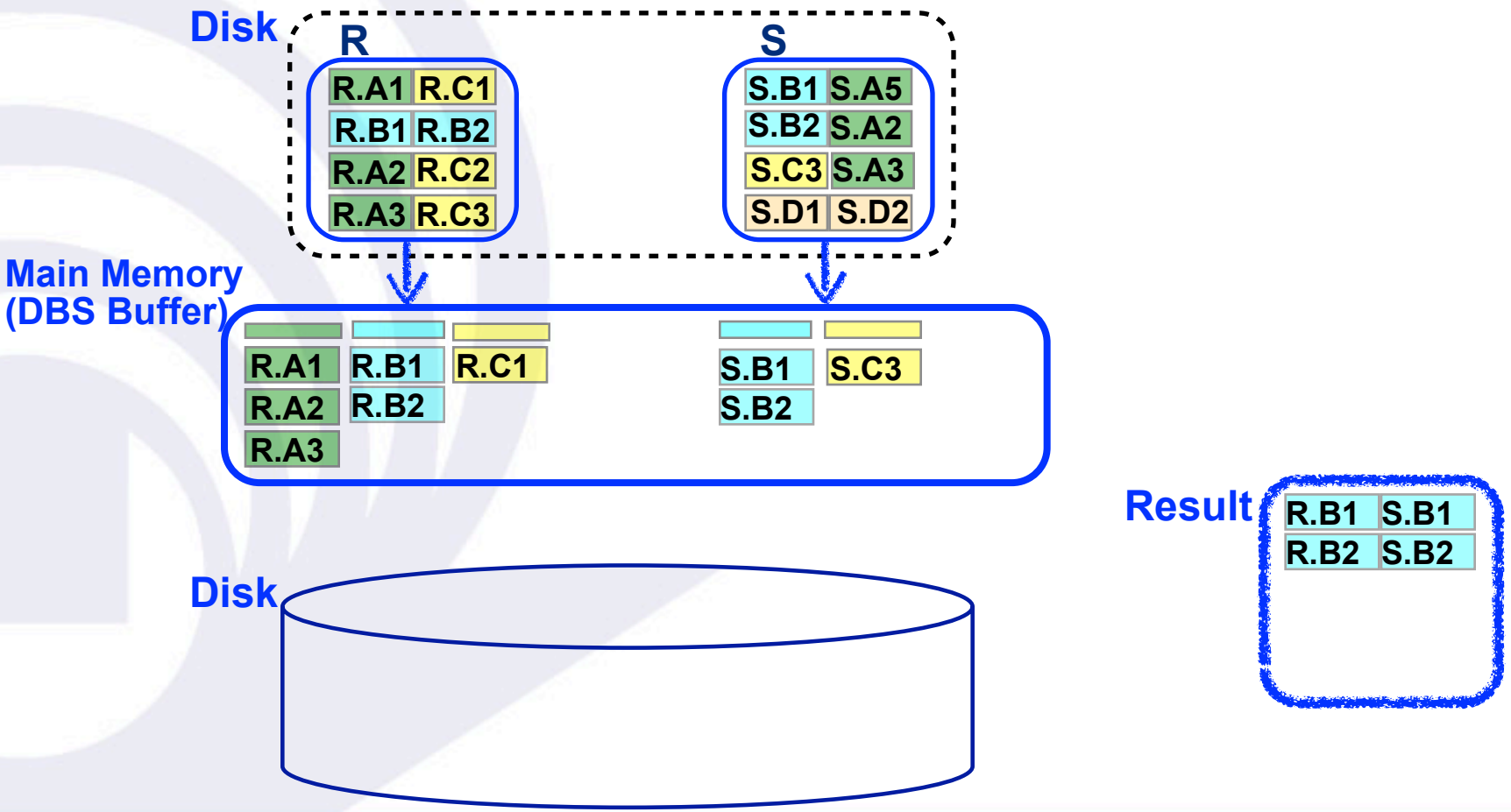
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



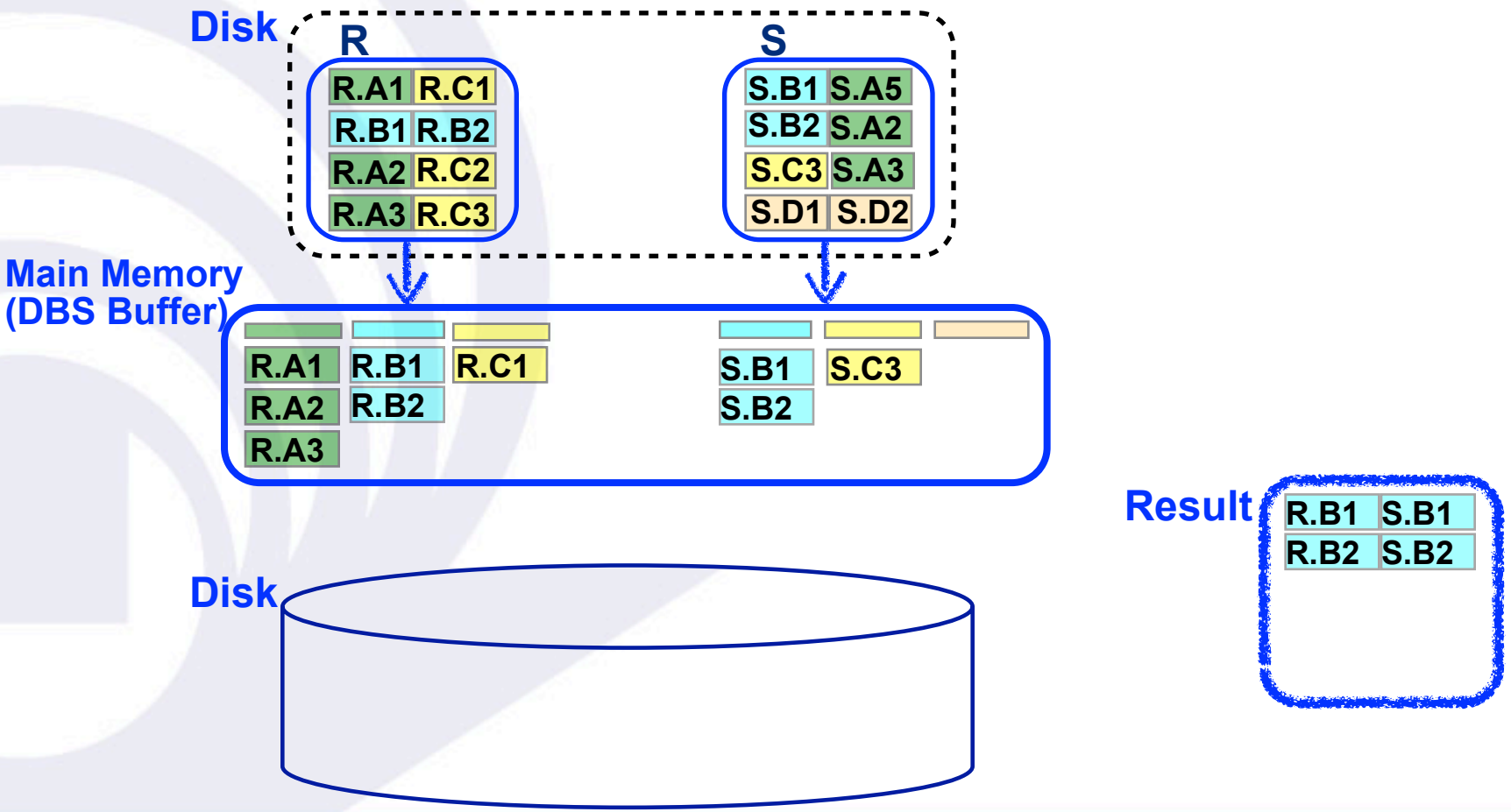
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



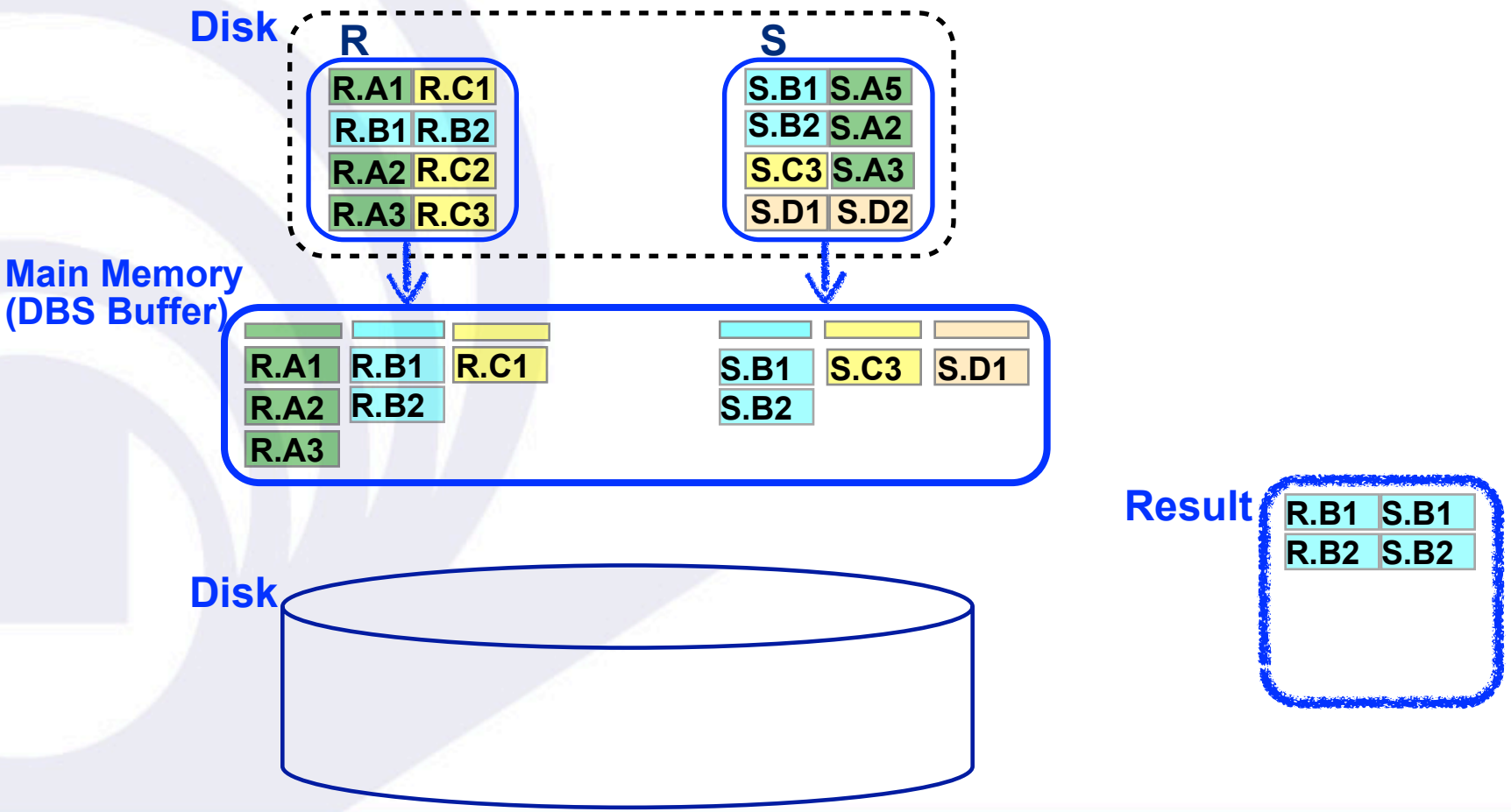
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



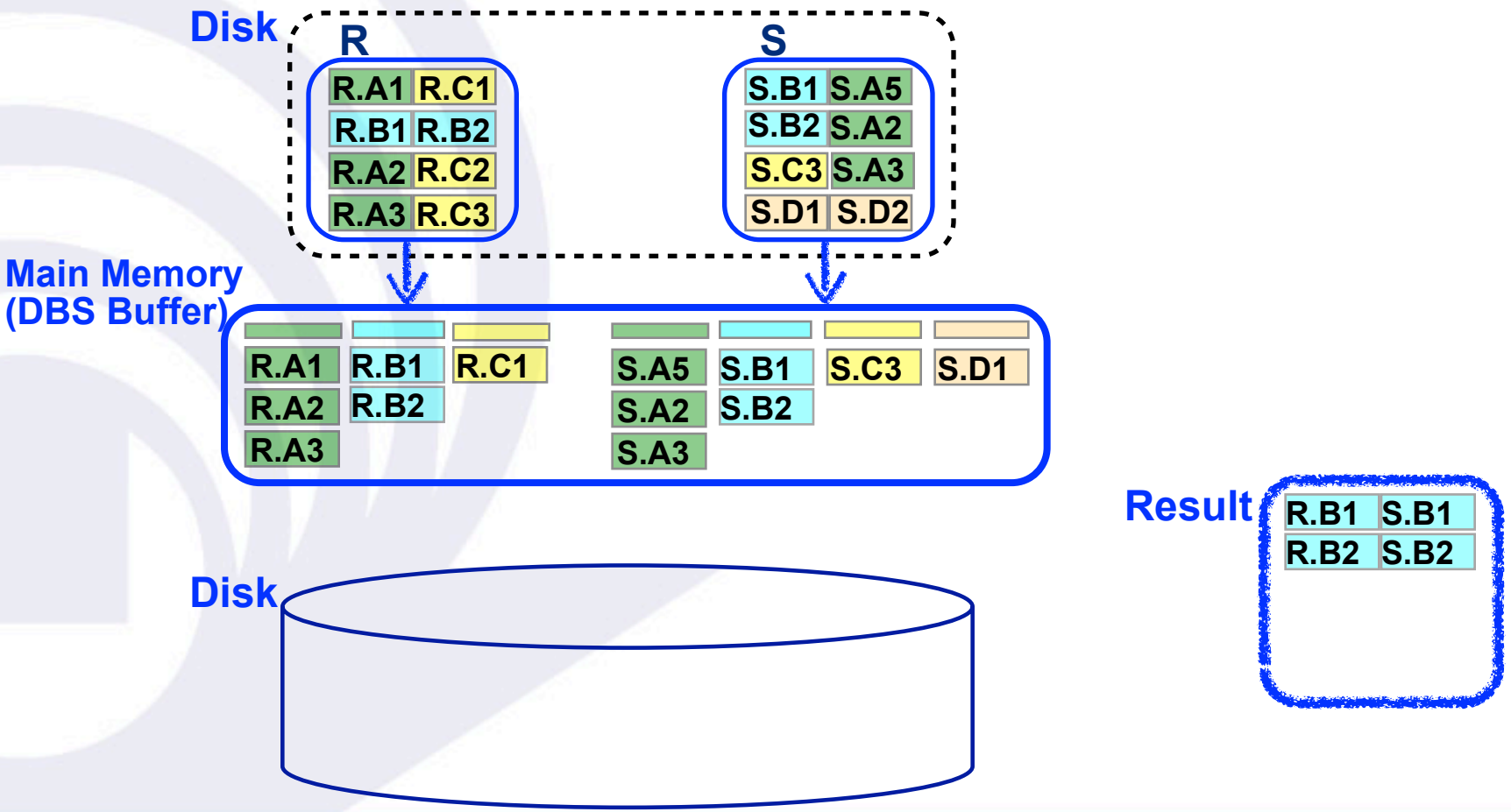
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



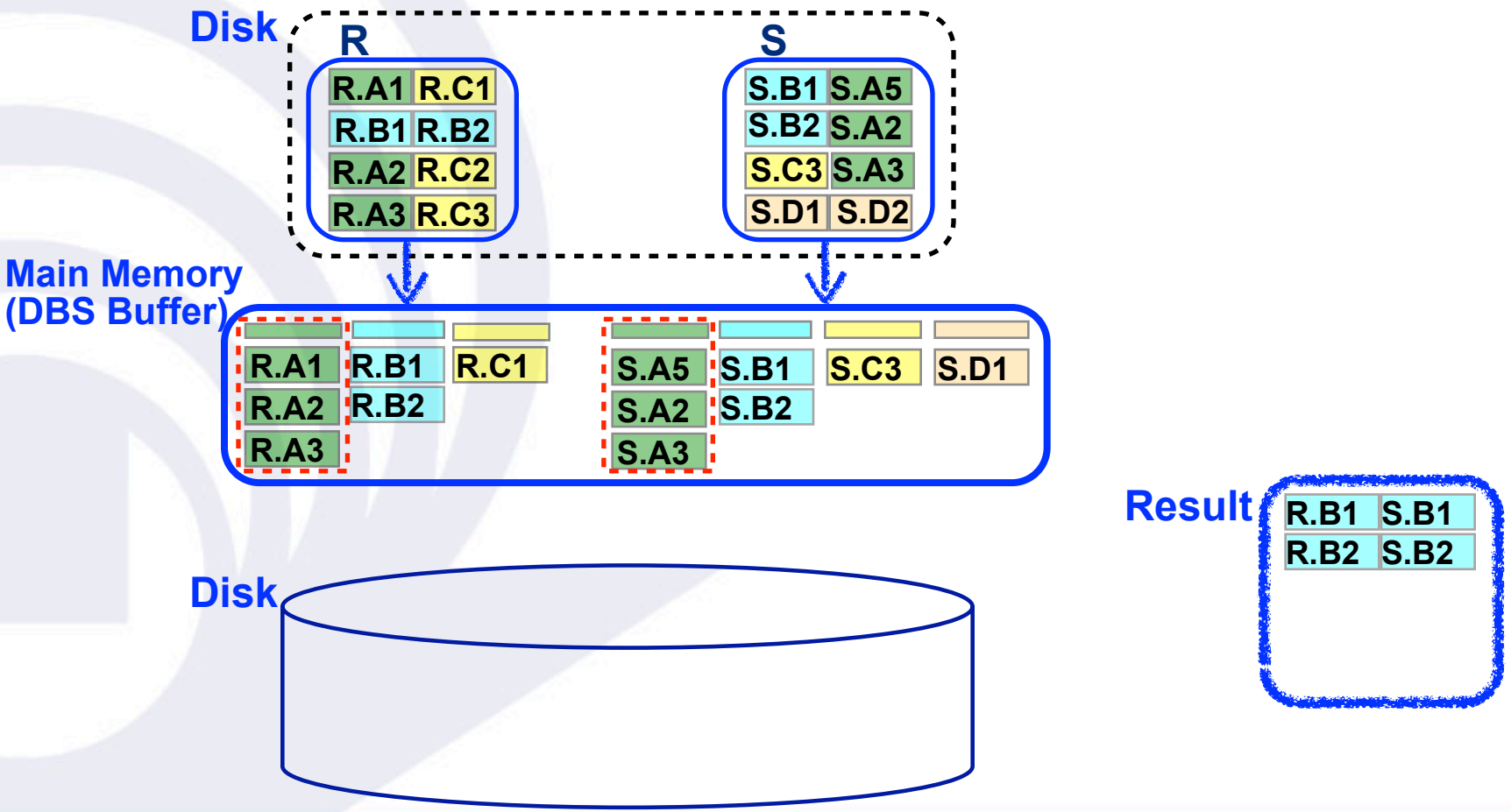
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



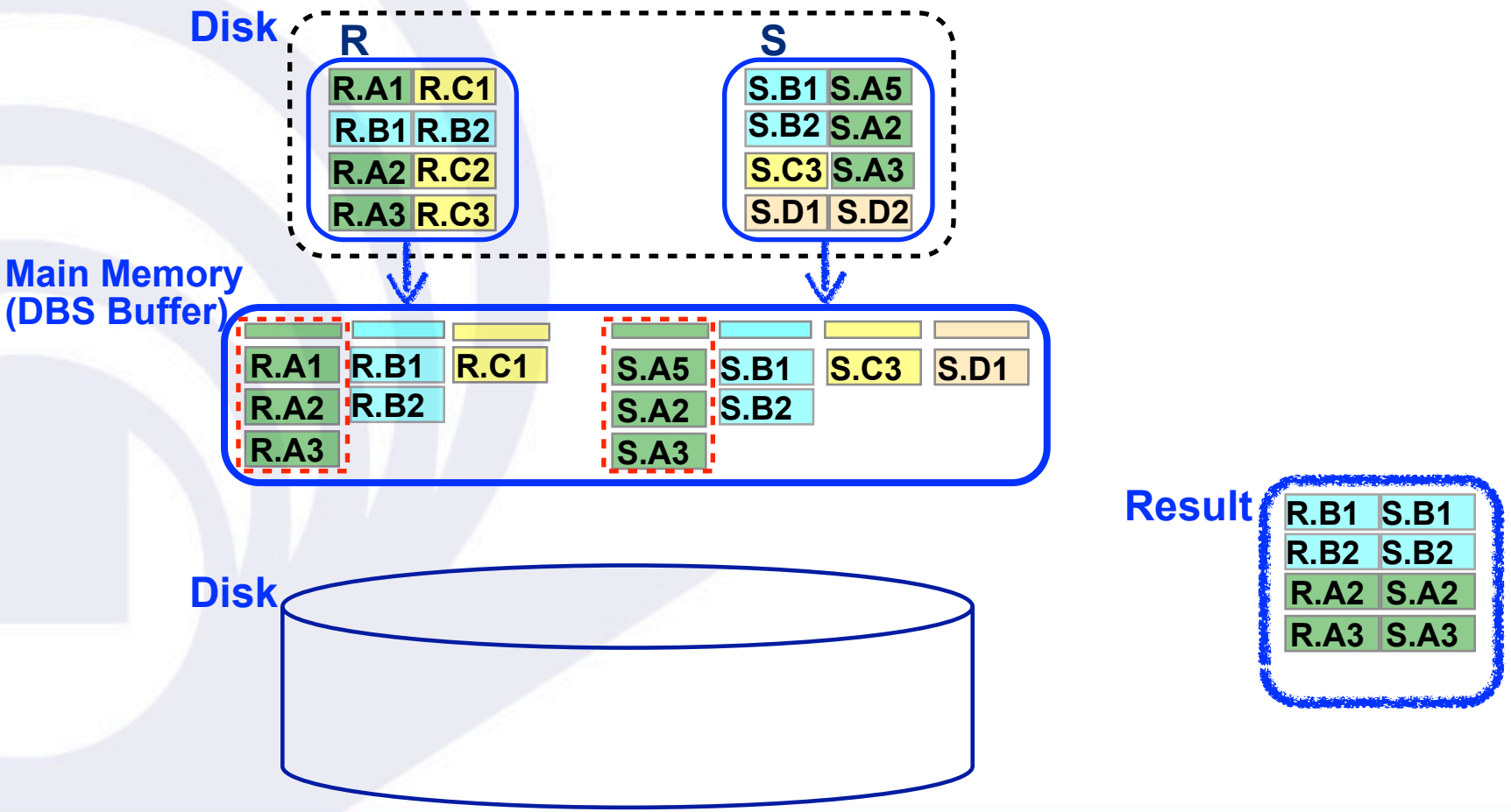
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



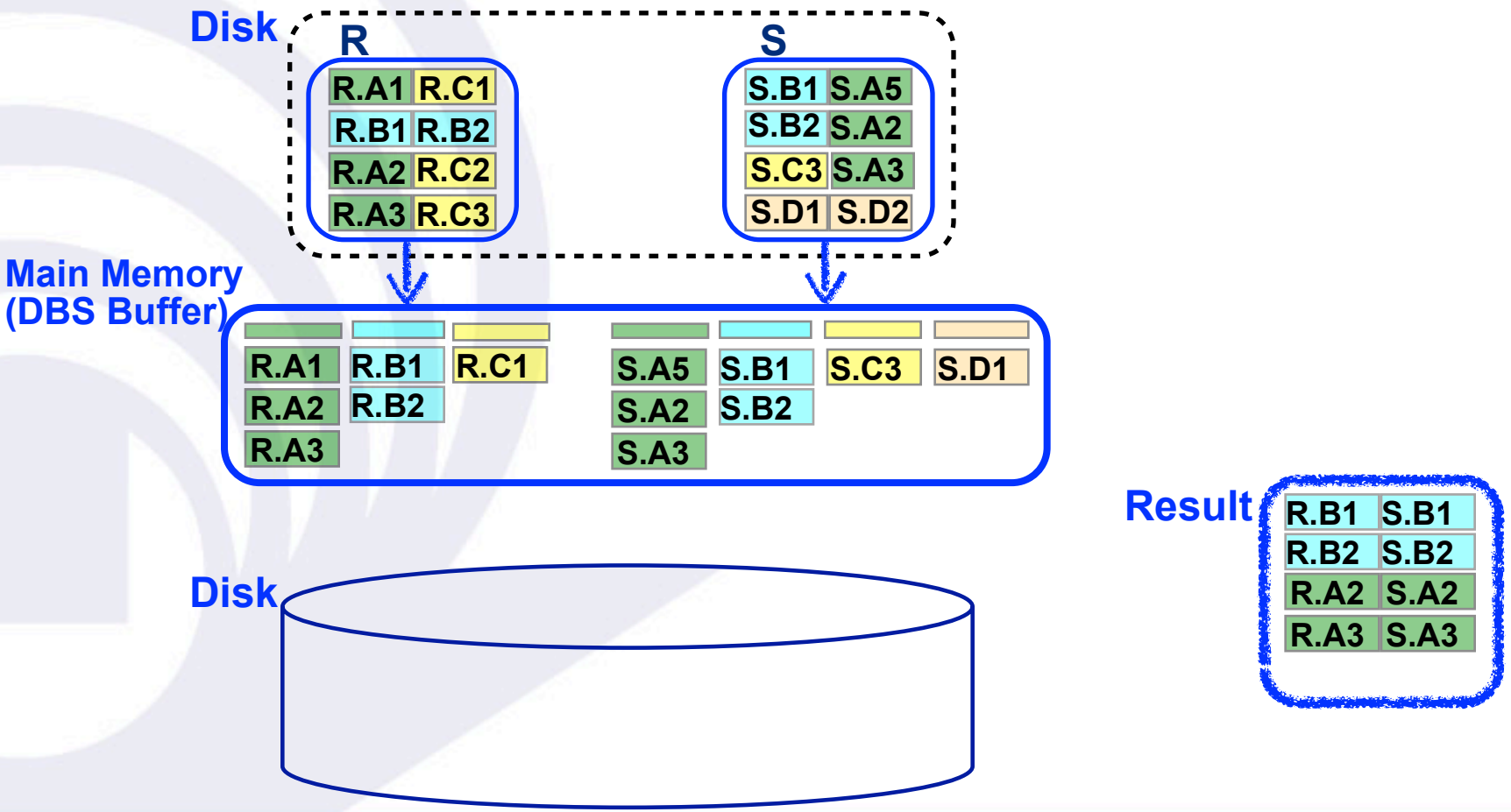
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



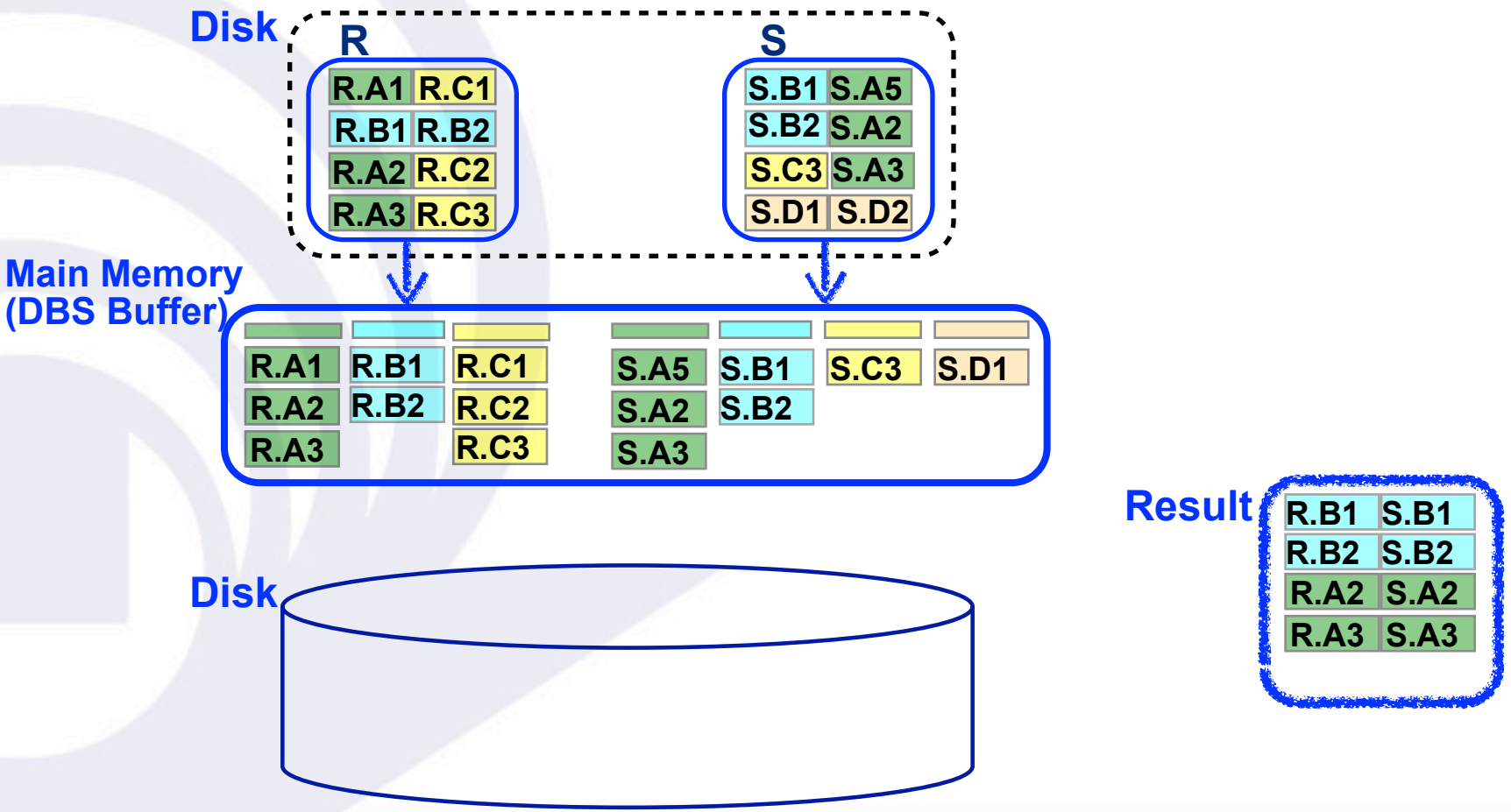
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



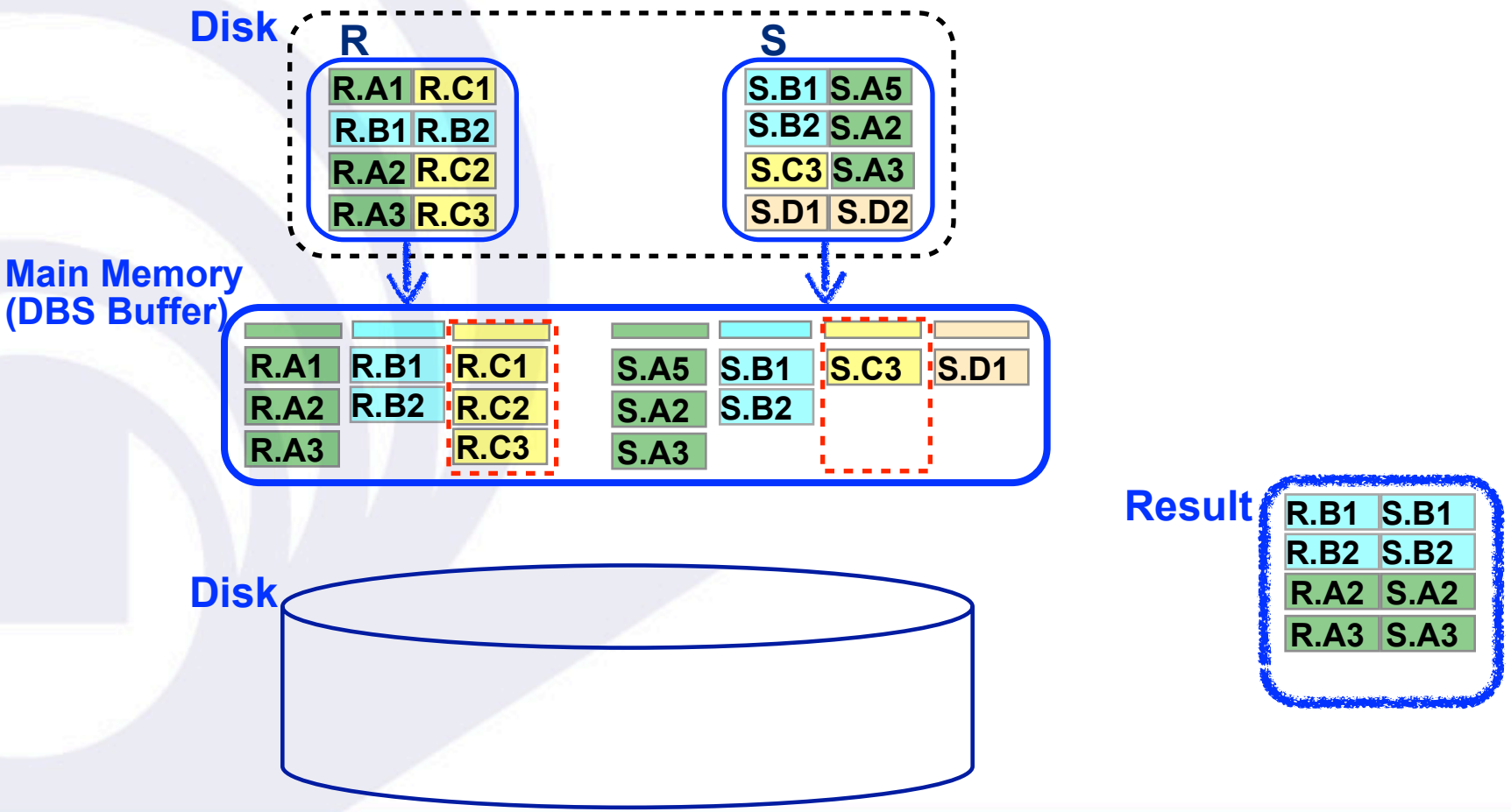
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



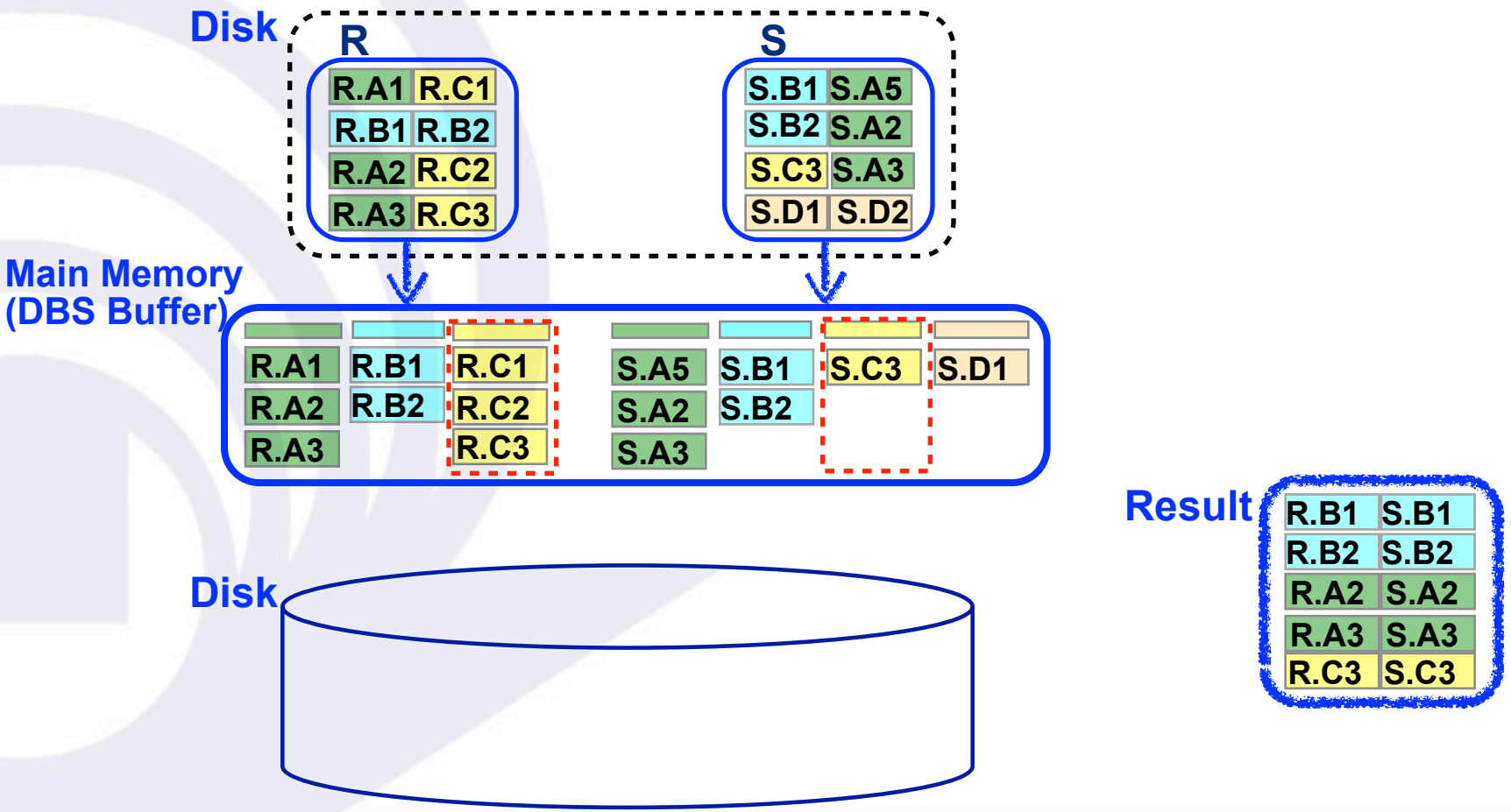
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



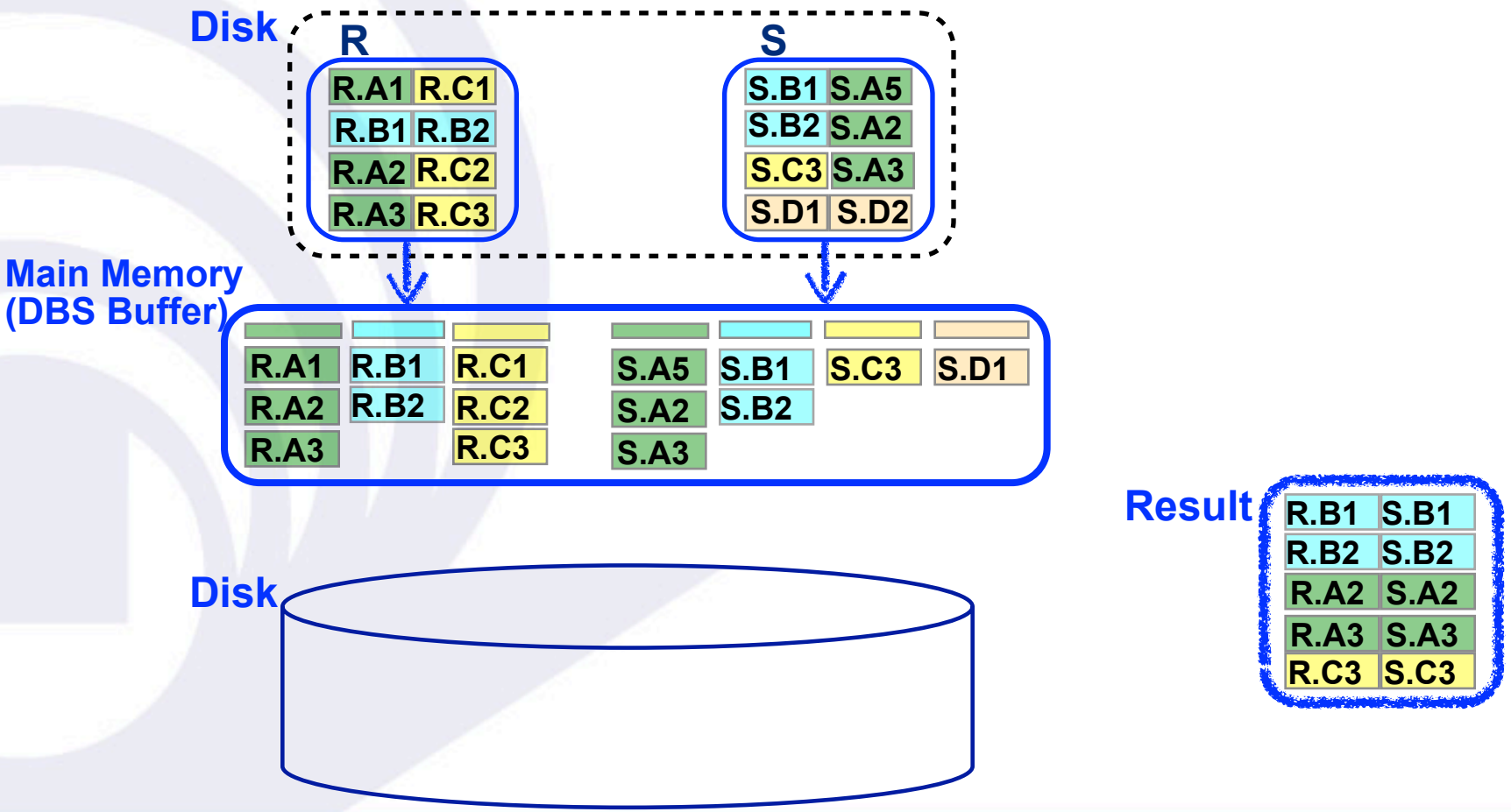
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



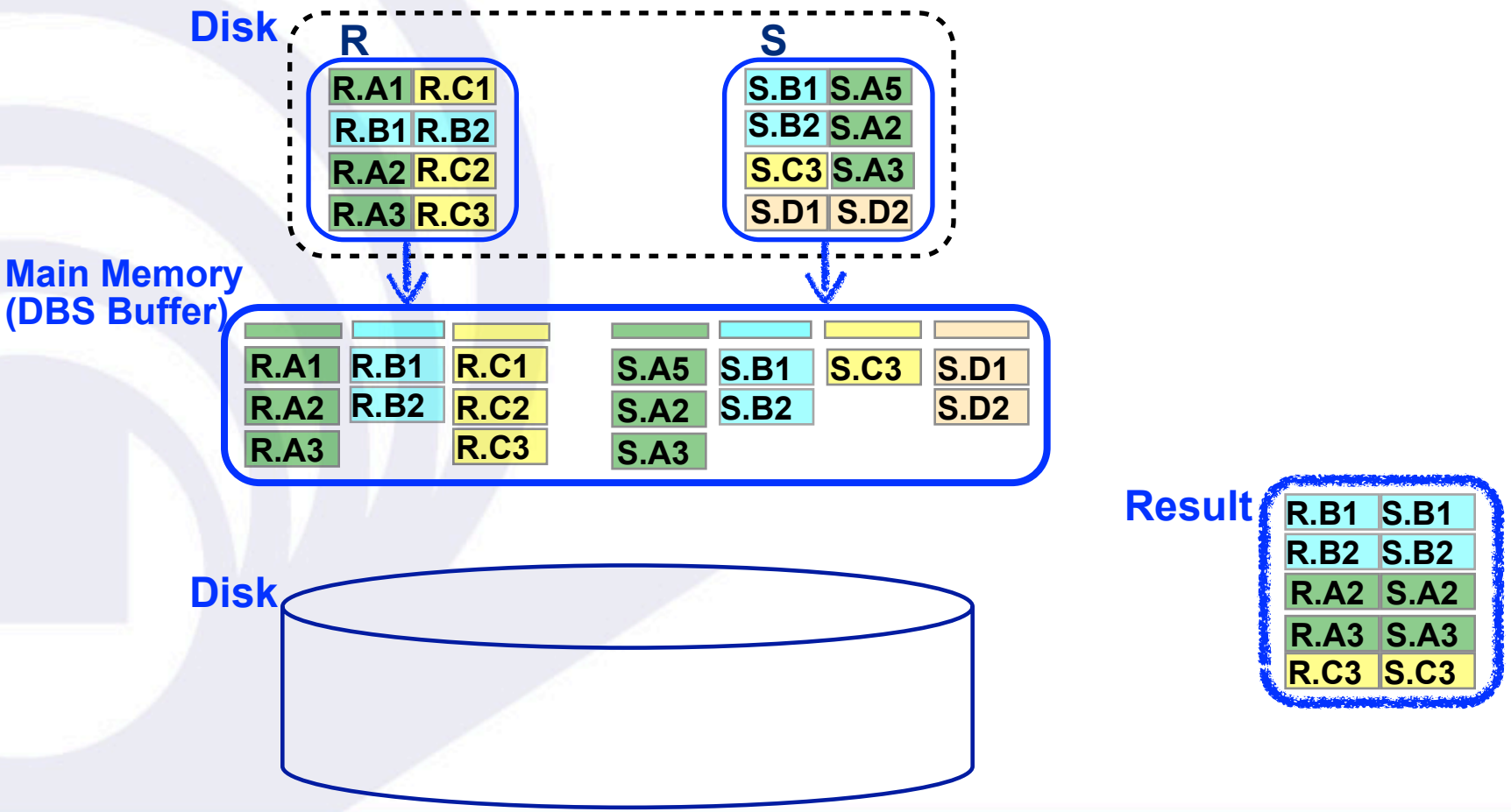
Join Algorithms

- Symmetric Hash Join [Graefe 1993]



Join Algorithms

- Symmetric Hash Join [Graefe 1993]



Join Algorithms

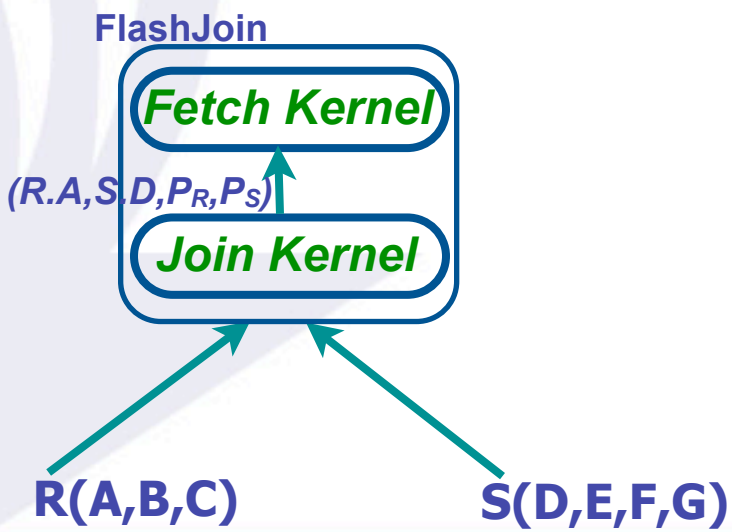
- Flash Join (R⋈S) [Graefe et al. 2009]
  - Join kernel
    - Computes the join algebraic operator
      - May implement any join algorithm
      - Hybrid Hash Join
        - **Partitions contain tuples: *(JoinAttrib, P_R)***
    - Yields a join index (JI)
      - (JoinAttrib, P_R, P_S)
  - Fetch Kernel
    - Uses JI to materialize the attributes of the join result from R and S

Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L

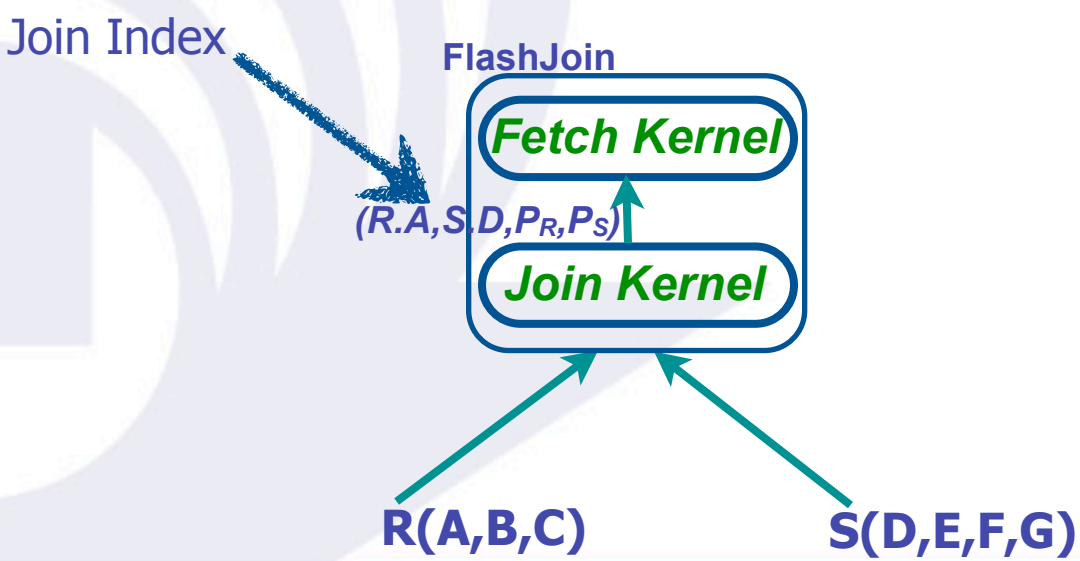
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



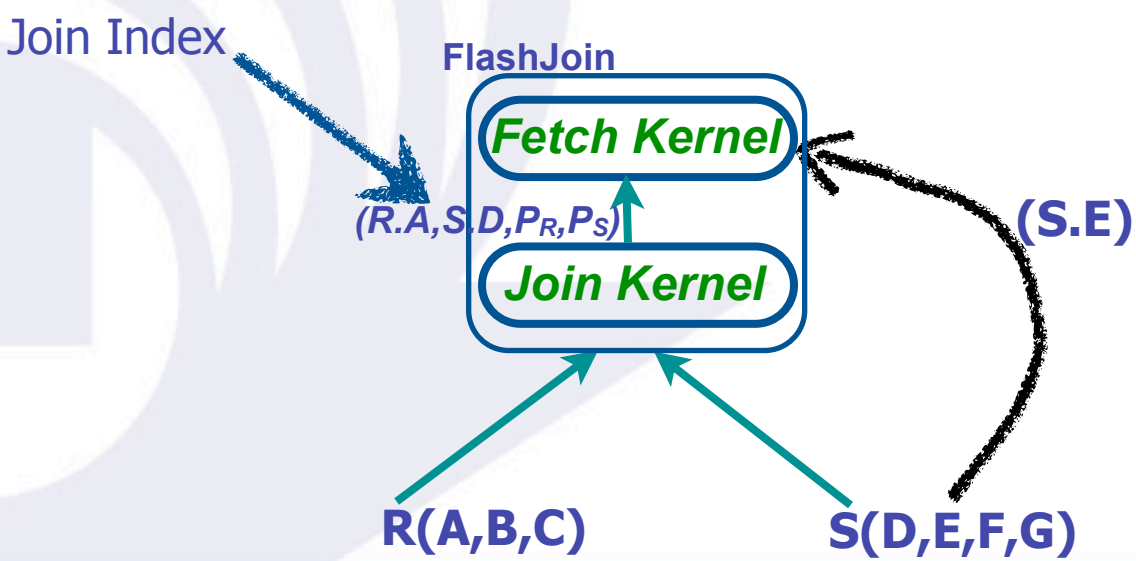
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



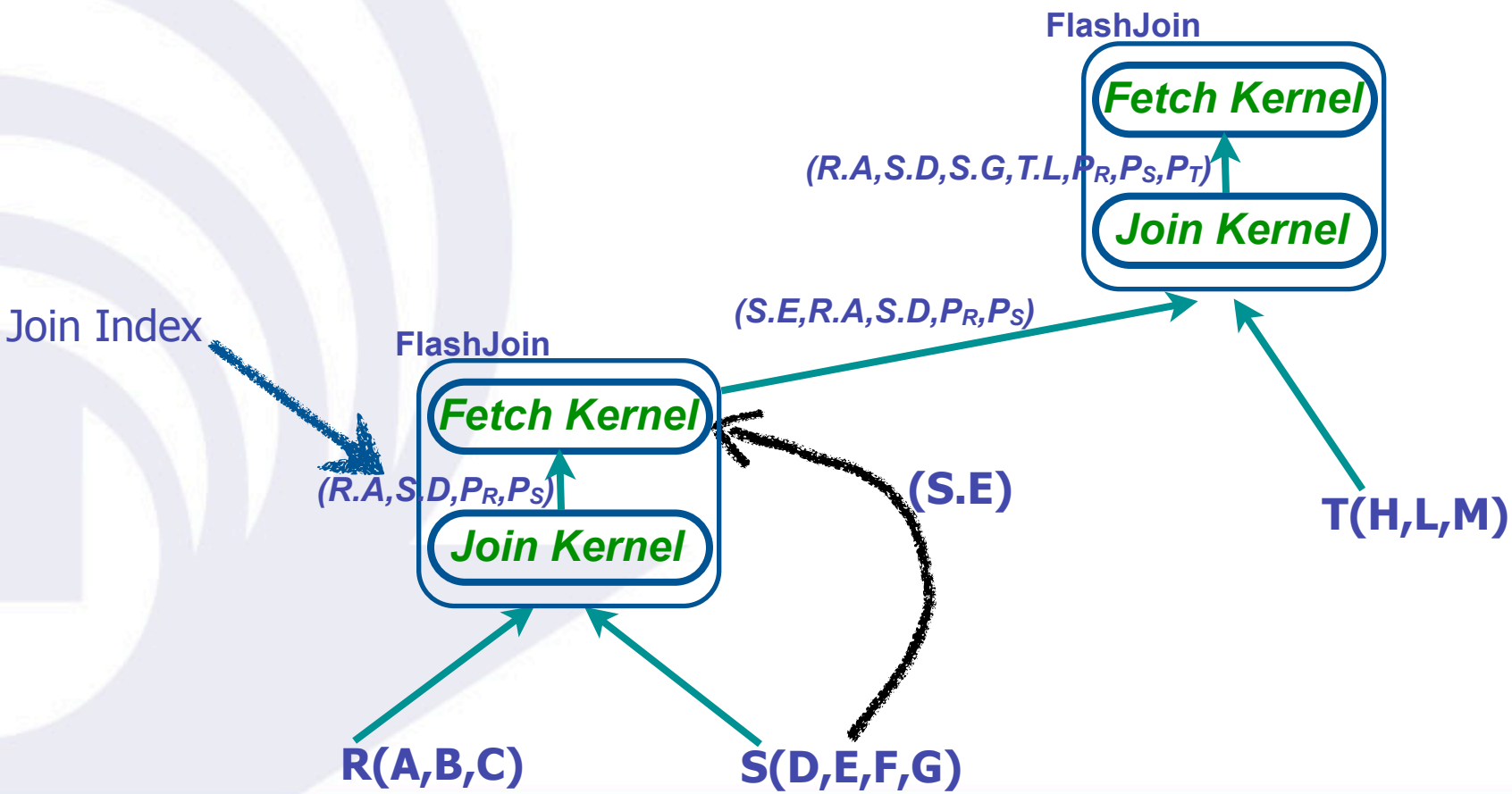
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



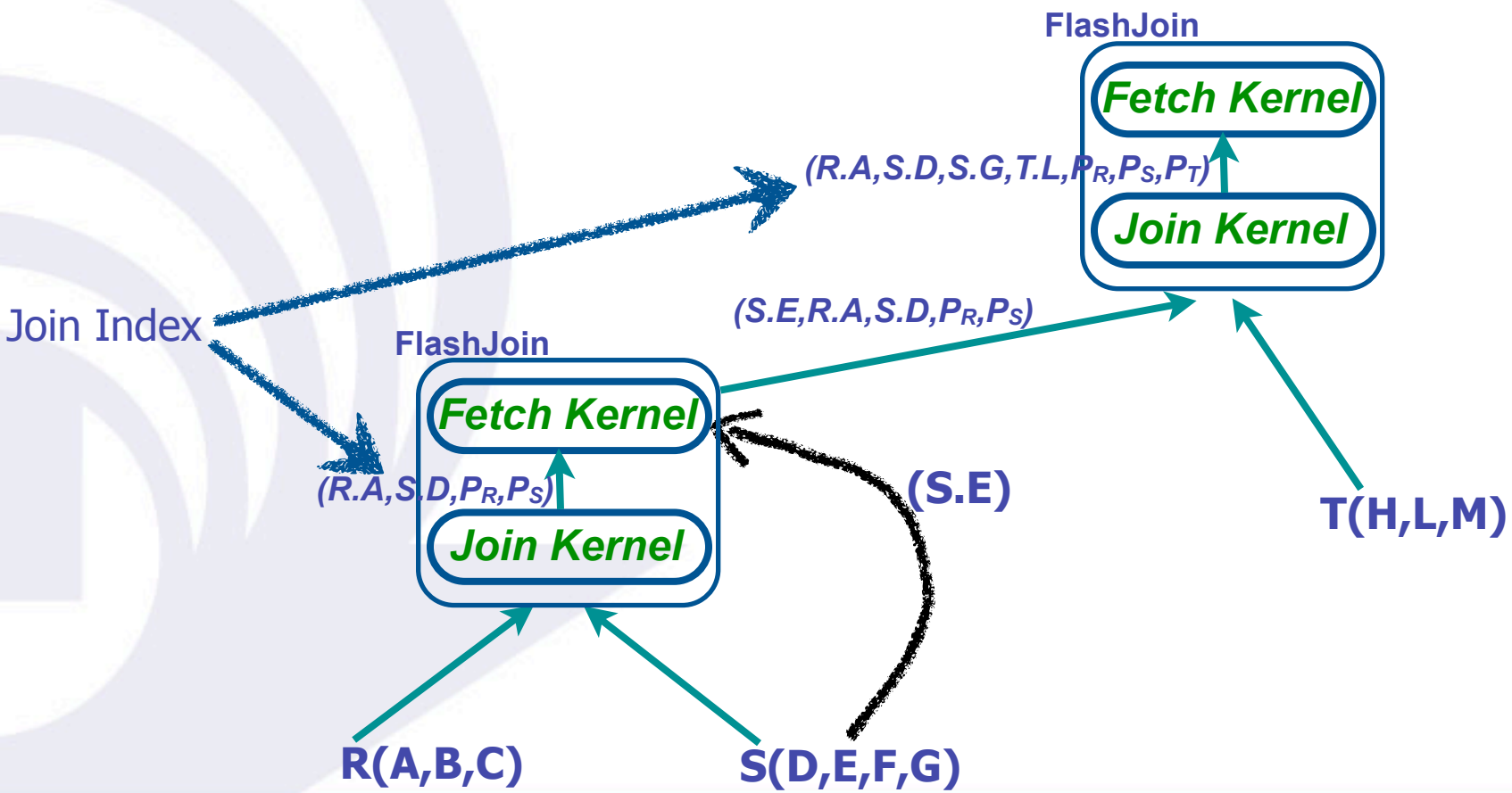
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



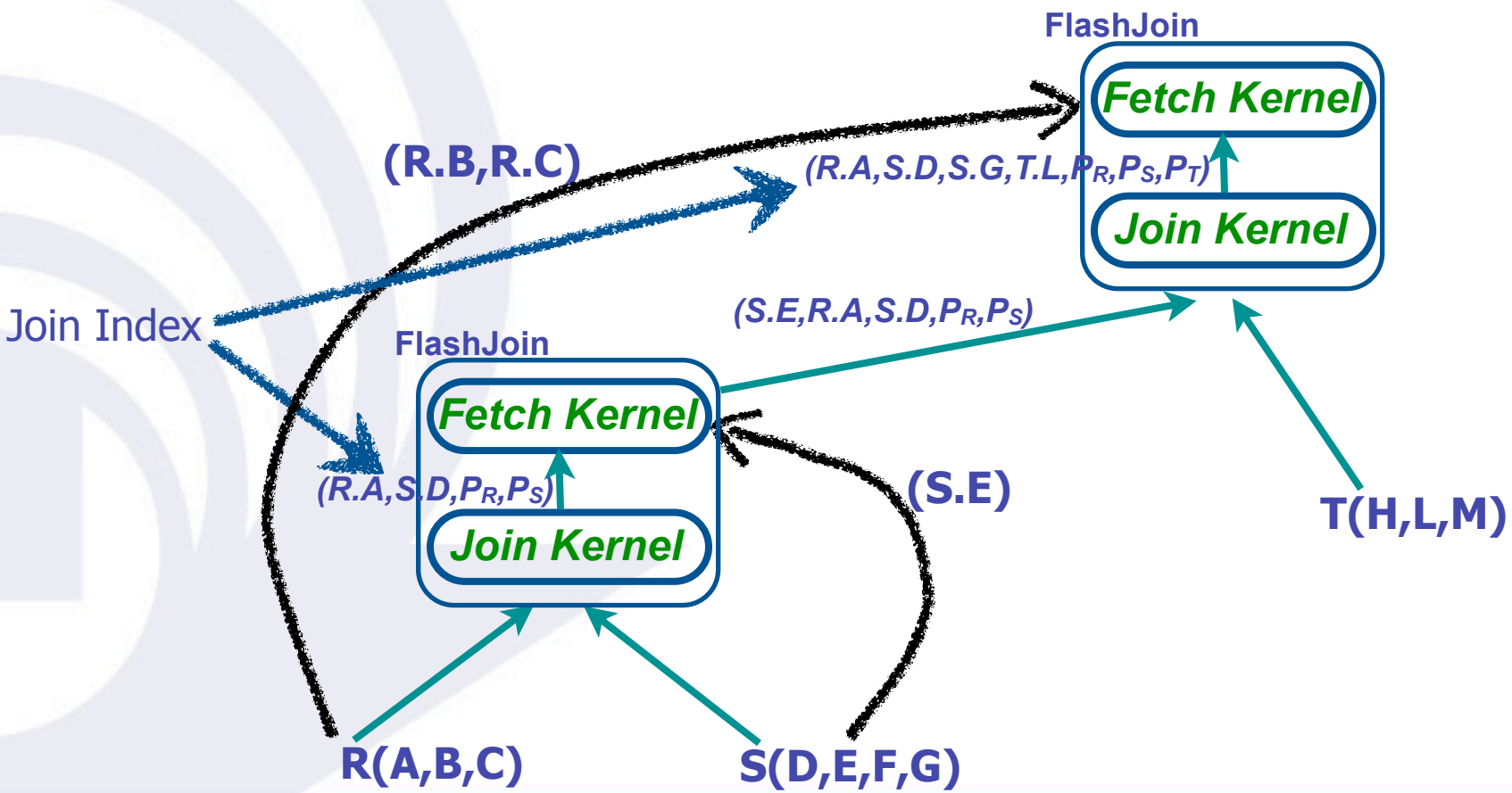
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



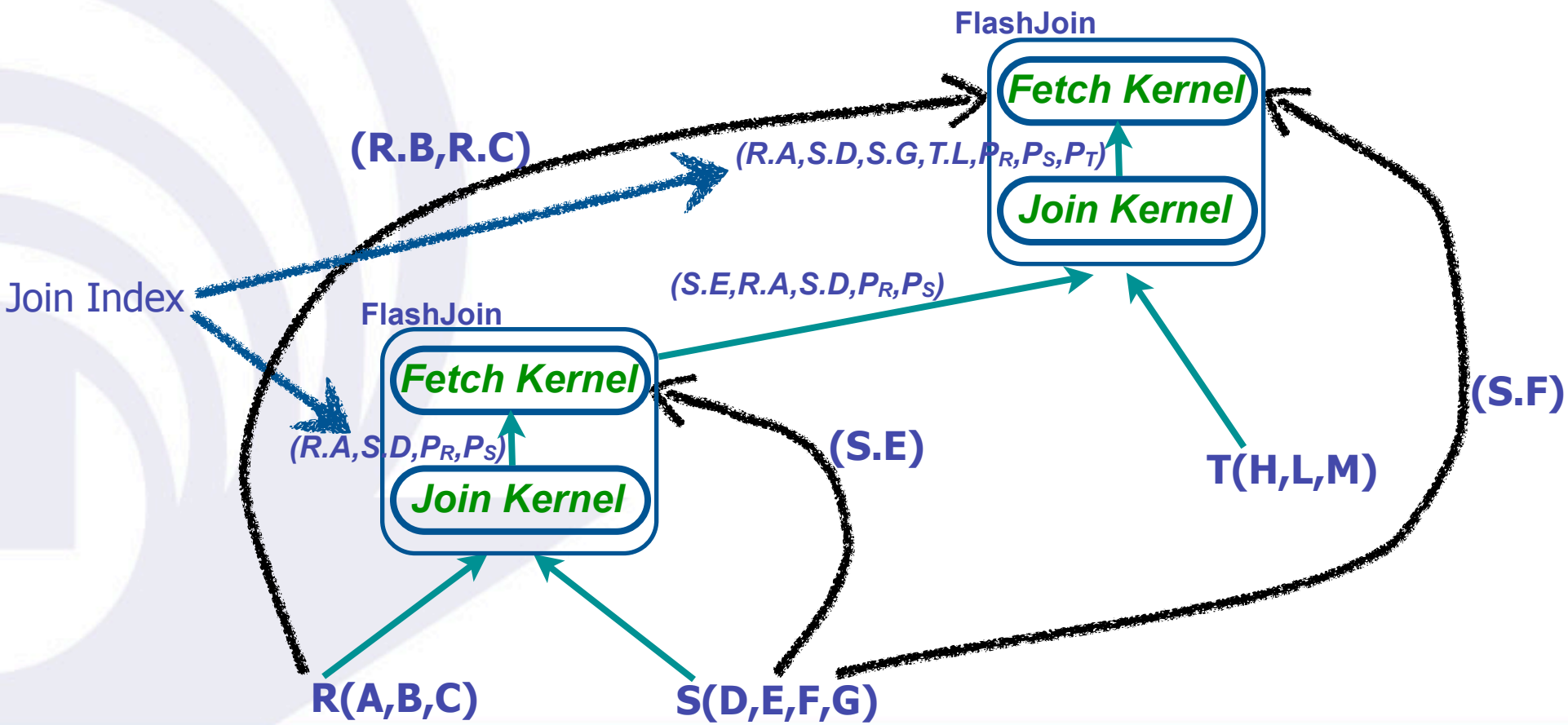
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



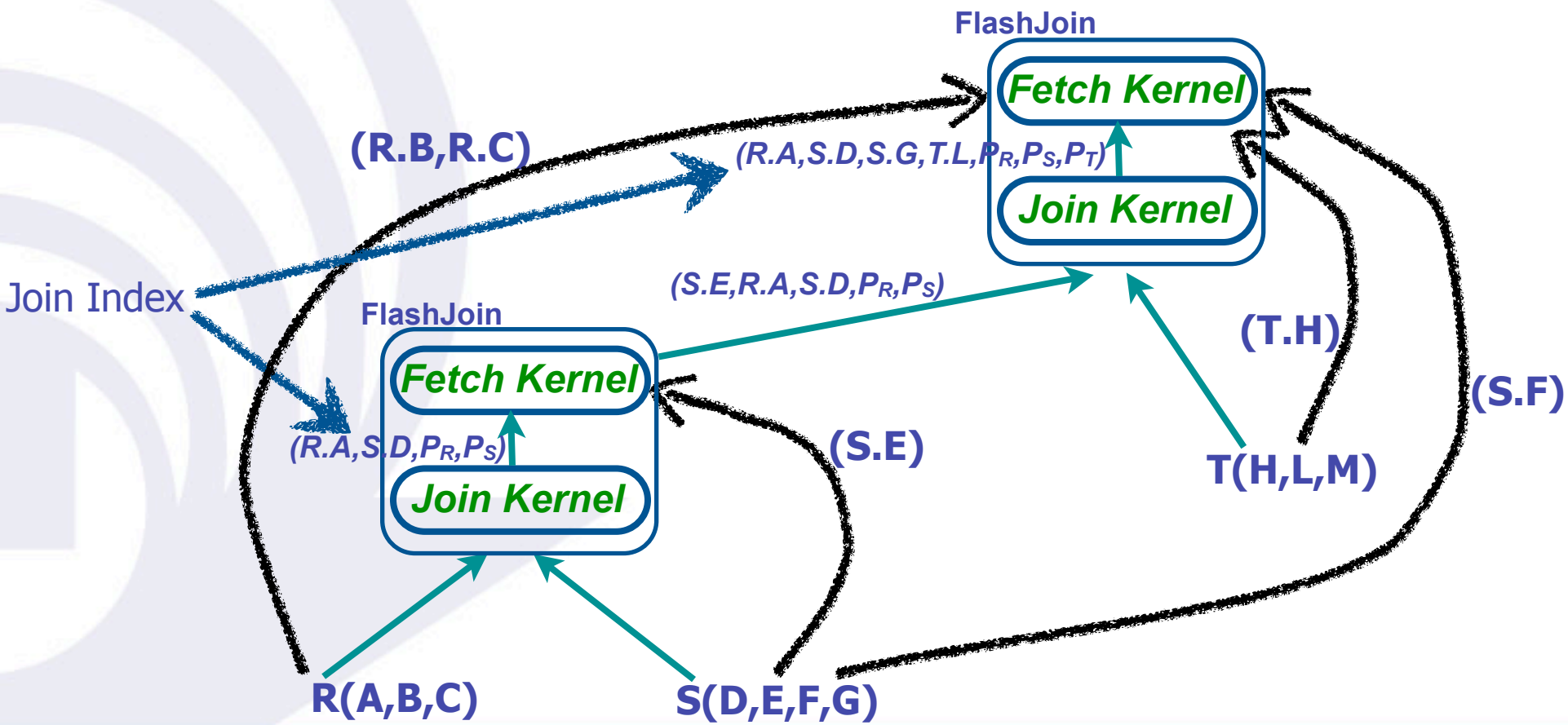
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



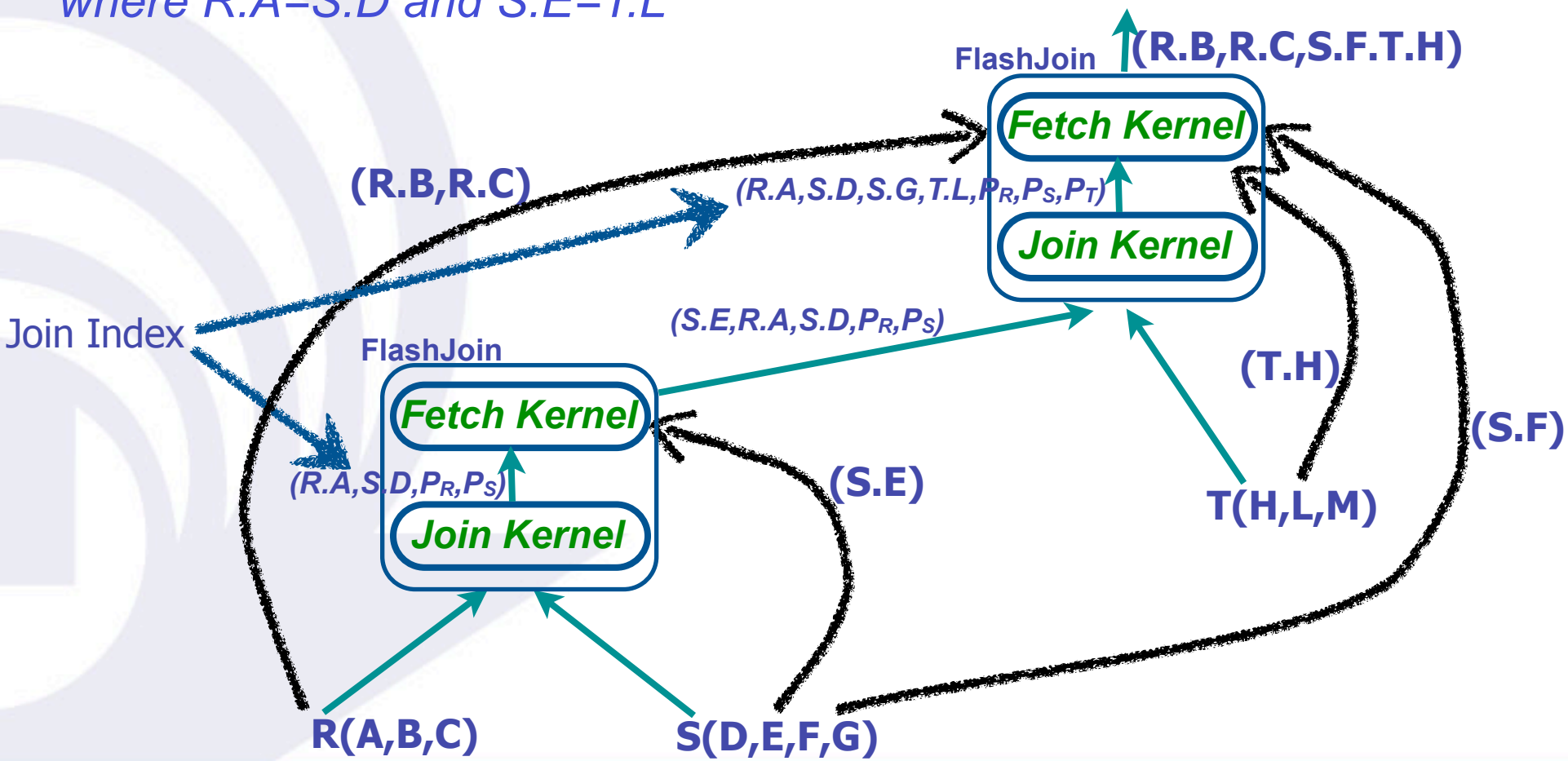
Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



Join Algorithms

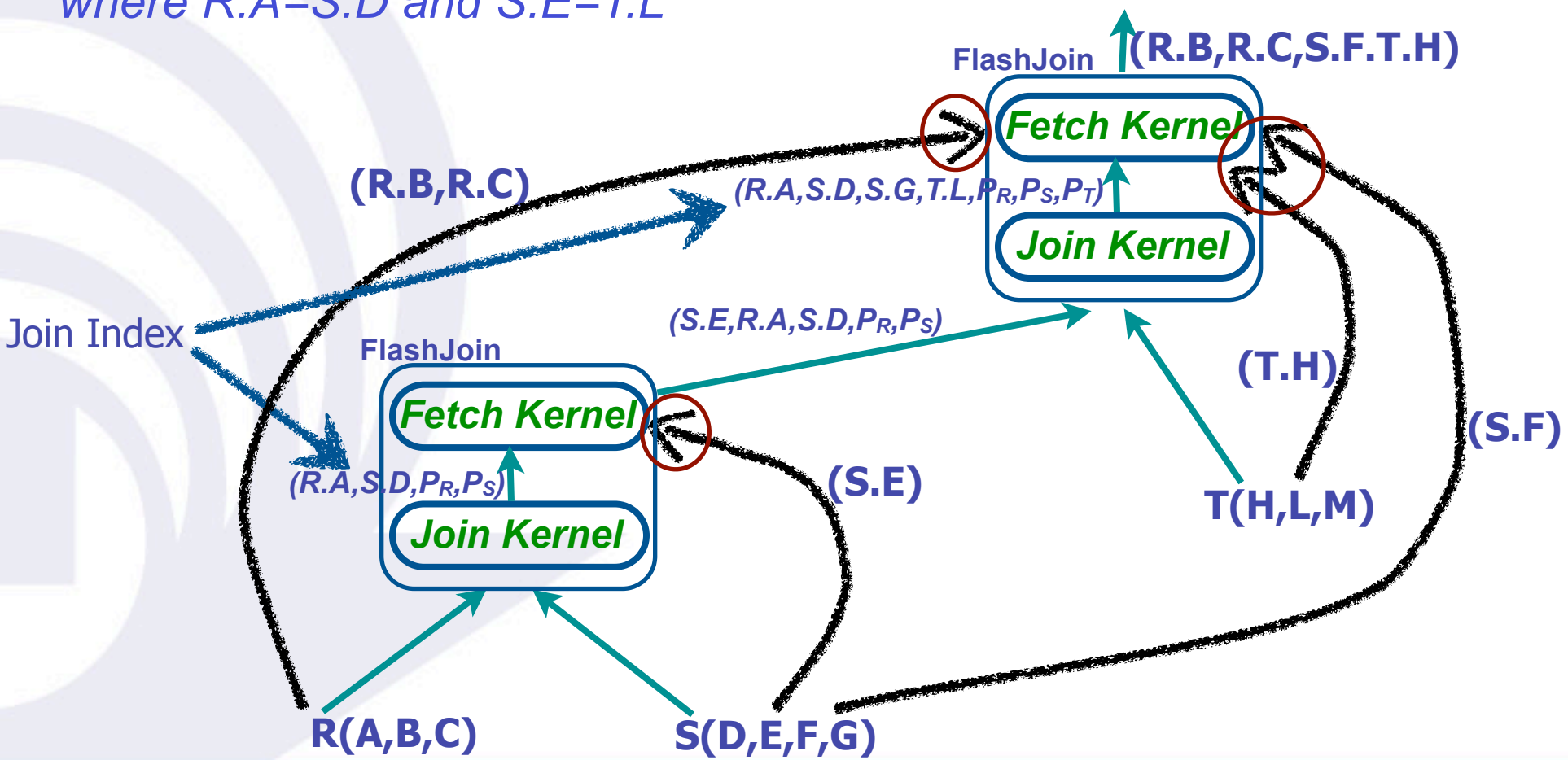
- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L



Join Algorithms

- Flash Join (R⋈S⋈T)
  Select R.B, R.C, S.F, T.H from R,S,T
  where R.A=S.D and S.E=T.L

Late Materialization



Join Algorithms

- SCM Join
  - Implements a Hybrid Hash Join on compressed tuples
  - Late materialization
    - Without cost of
      - Random reads for accessing join attributes and other projected attributes
      - Random write to include join attributes and other projected attributes into partitions

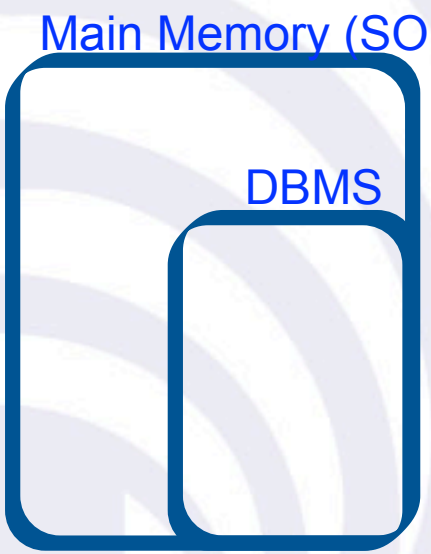
Caching

- Goal
  - To hold on main-memory (buffer pool) the most referenced pages
    - **Reduces the amount of disk access**
- Database Buffer Manager
  - Buffer replacement policy
  - Efficiency metric
    - Hit Ratio

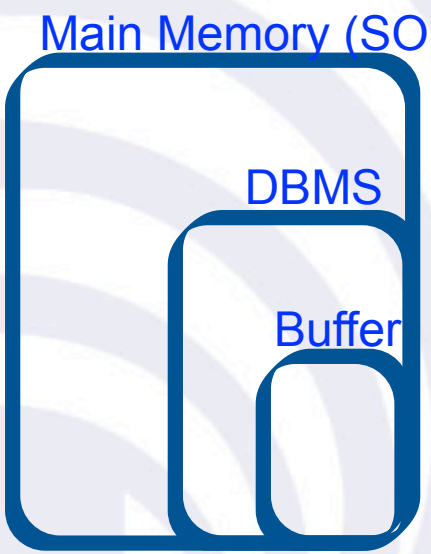
Database Buffer Management

Database Buffer Management

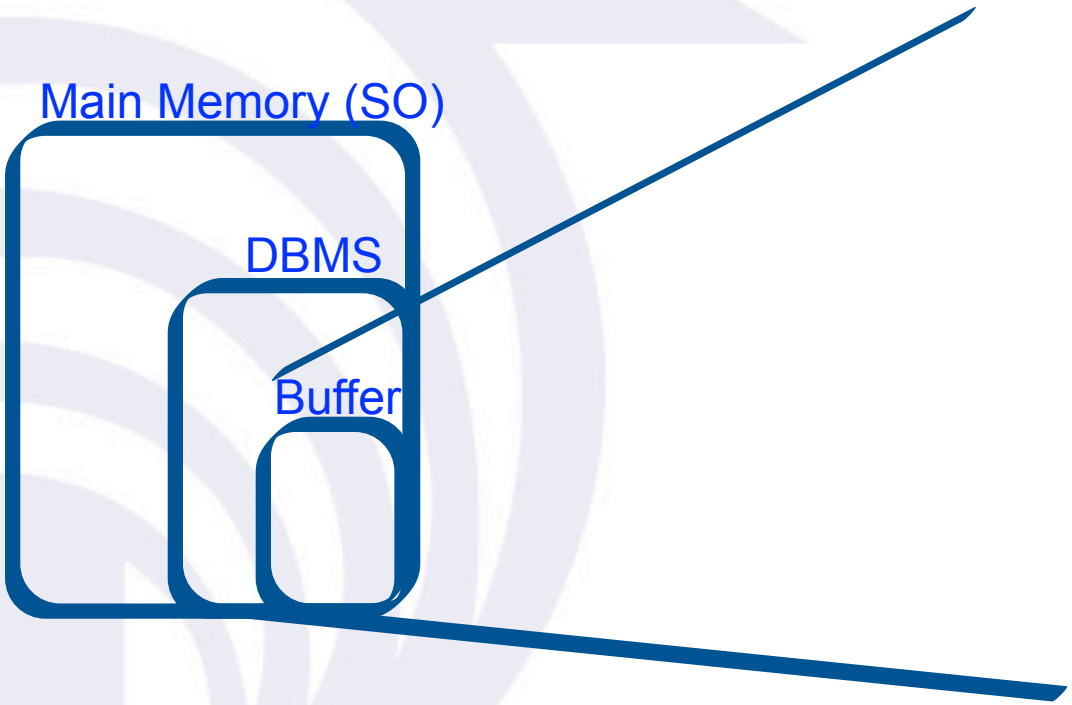
Database Buffer Management



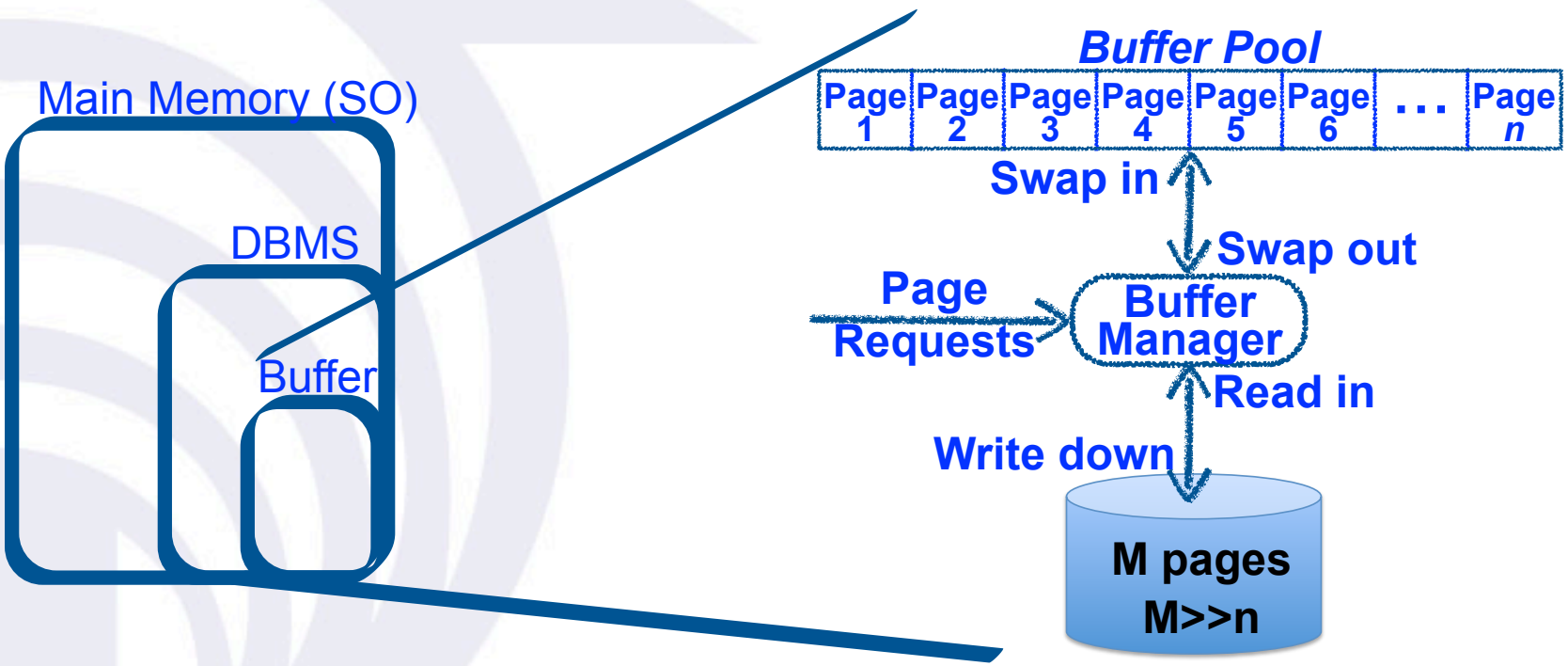
Database Buffer Management



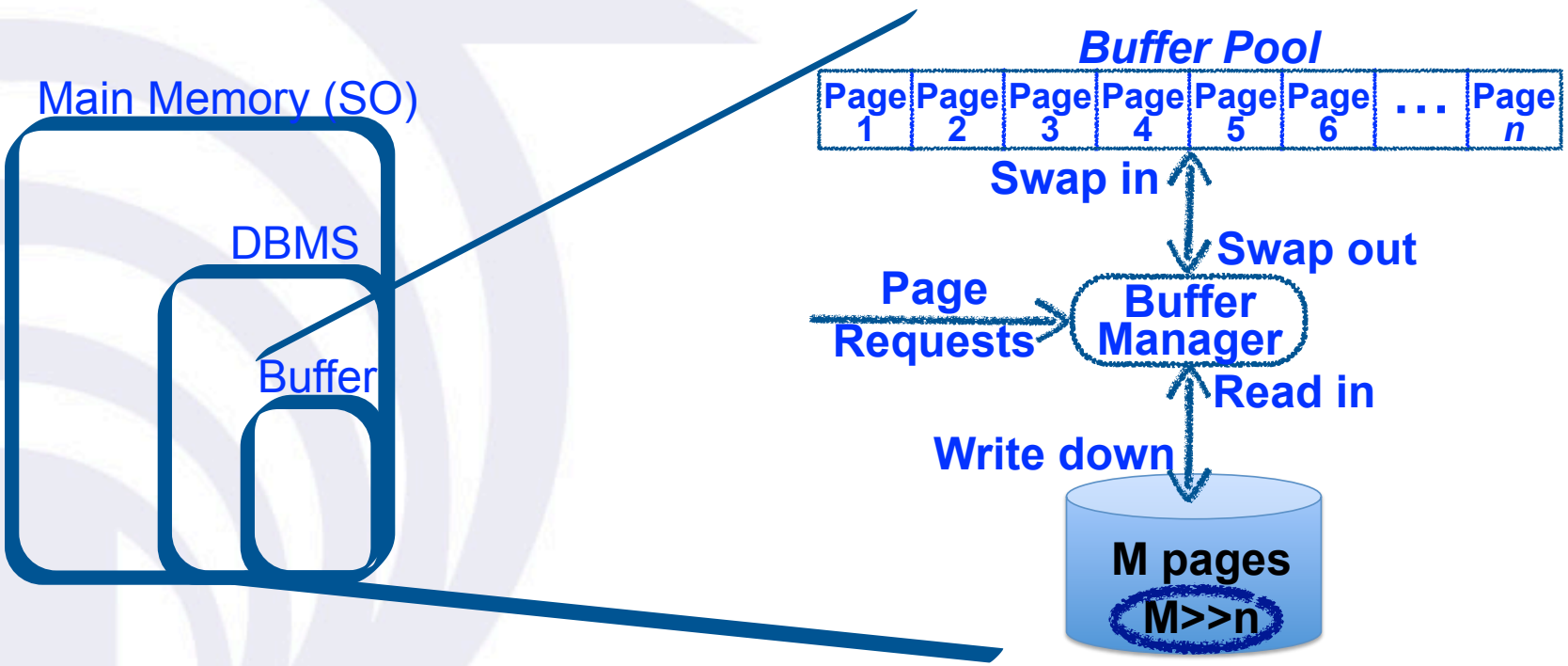
Database Buffer Management



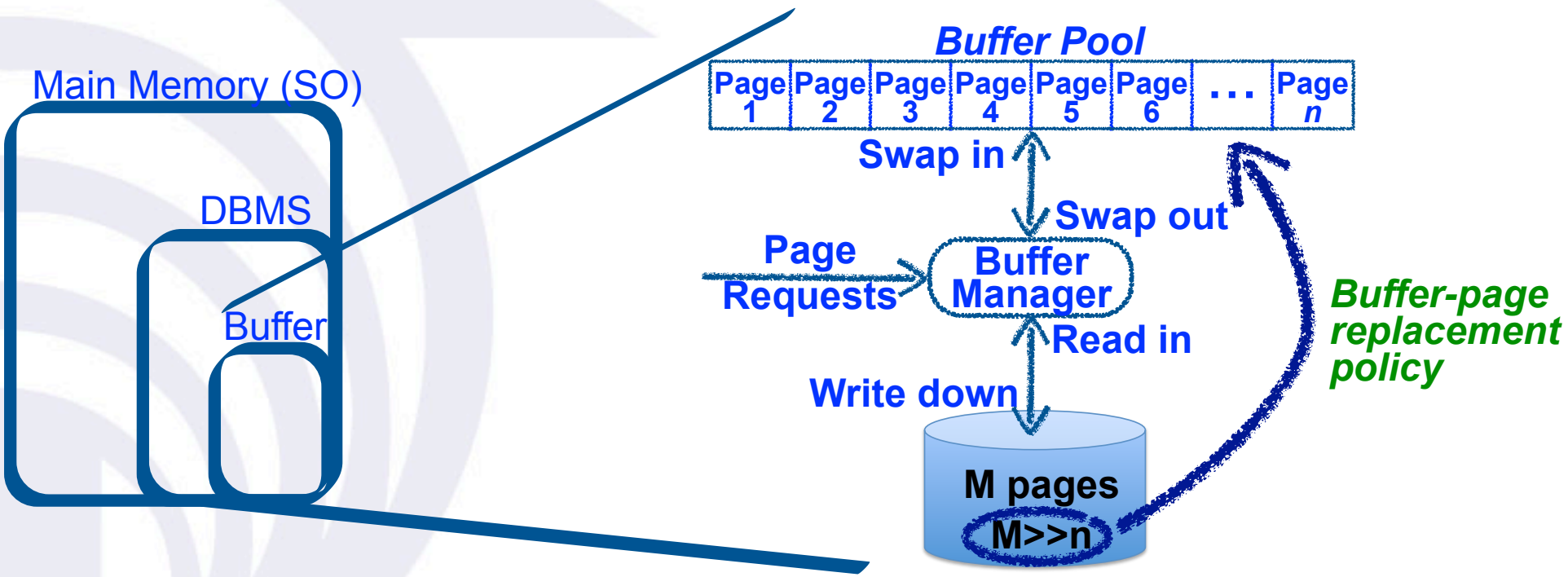
Database Buffer Management



Database Buffer Management



Database Buffer Management



Buffer Replacement Policy

- LRU (Least Recently Used)
  - Replace the page which has not been referenced for the longest period of time
  - Counter implementation
    - Every buffer page has a counter
    - Whenever a page is referenced, its counter is incremented
    - The page with the lowest counter is selected to be evicted

Buffer Replacement Policy

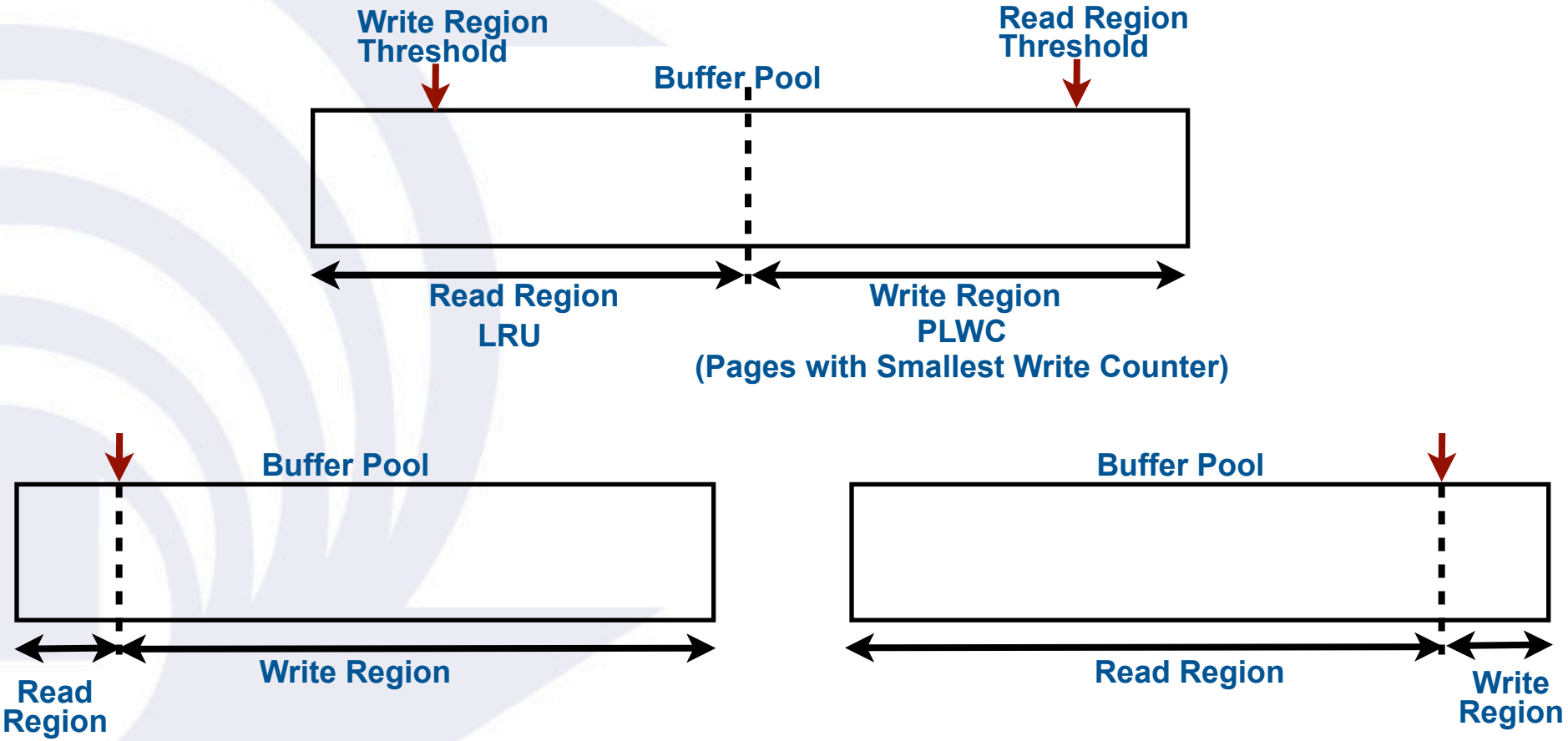
- LRU-WSR [Jung et al. 2008]
  - LRU - Write Sequence Reordering
  - Buffer page state
    - Clean
      - There is no write operation
    - Cold-dirty
      - There are write operations
      - Not referenced in a given time window
    - Hot-dirty
      - There are write operations
      - Referenced in a given time window

Buffer Replacement Policy

- LRU-WSR's replacement criteria
  - clean page >> cold-dirty page >> hot-dirty page
- CFDC [Ou et al. 2009]
  - Clean-First Dirty-Clustered
  - Clean page queue and dirty page queue
  - Dirty page queue
    - pages are clustered
      - Locality principle

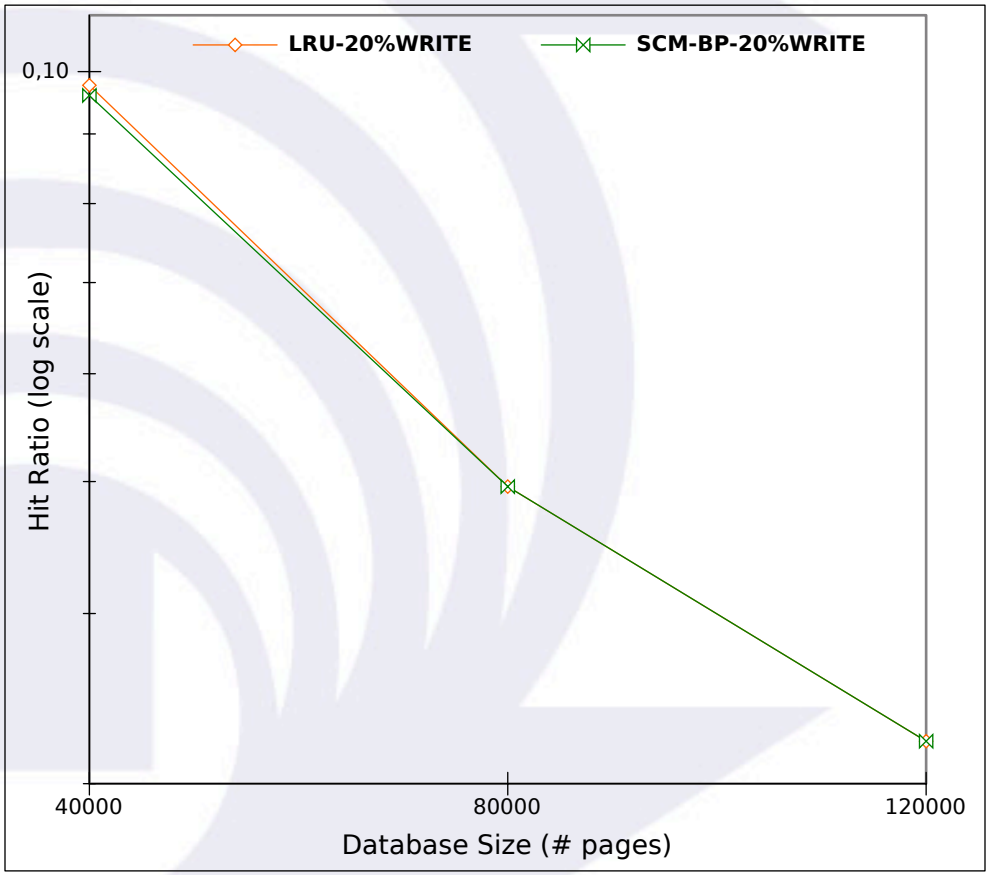
Buffer Replacement Policy

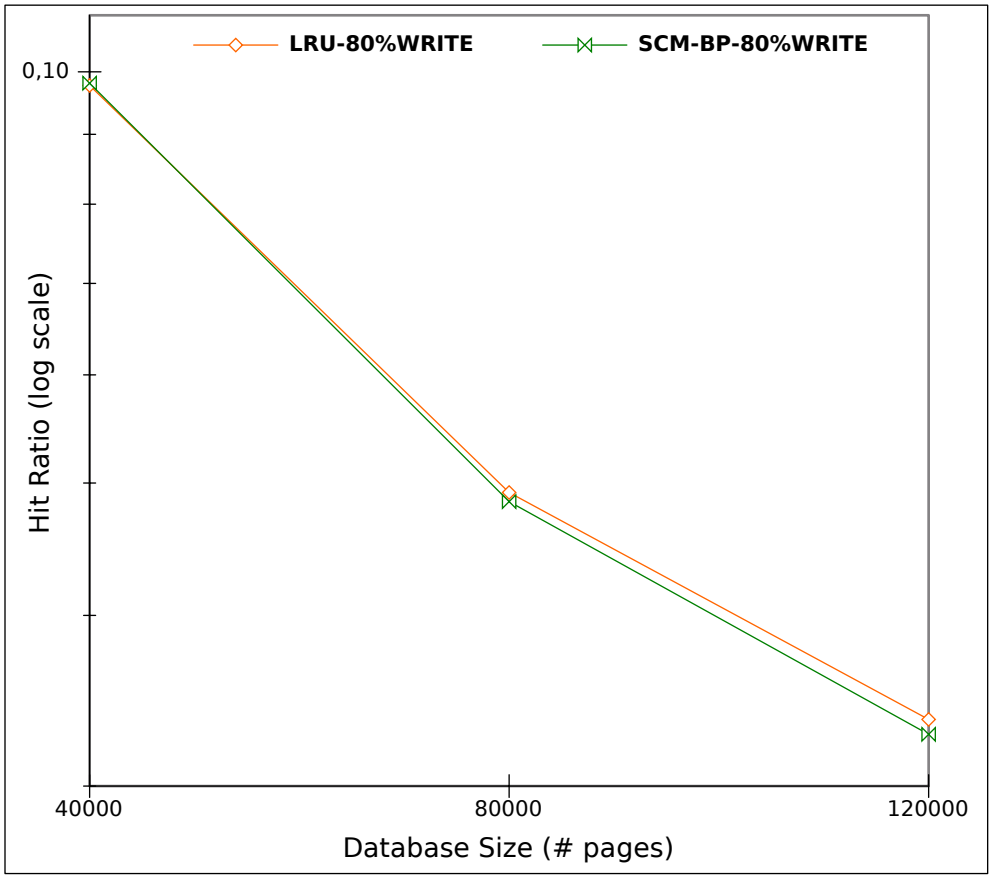
- SCM-BP



Buffer Replacement Policy

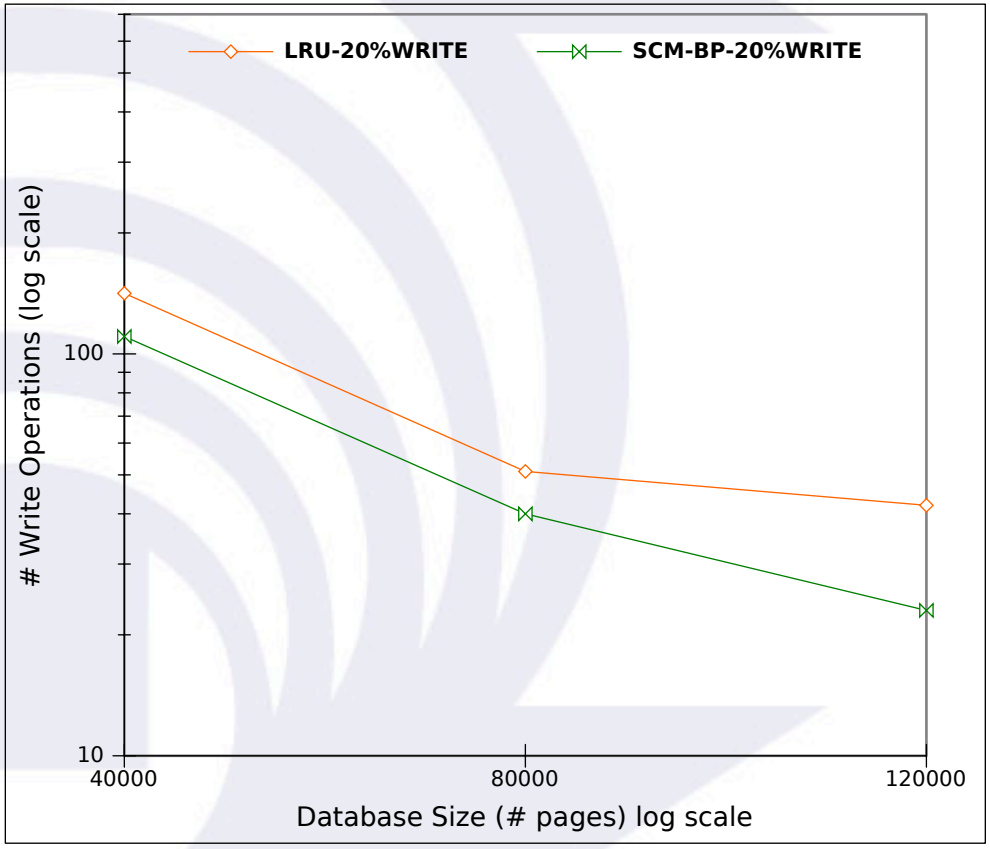
- SCM-BP

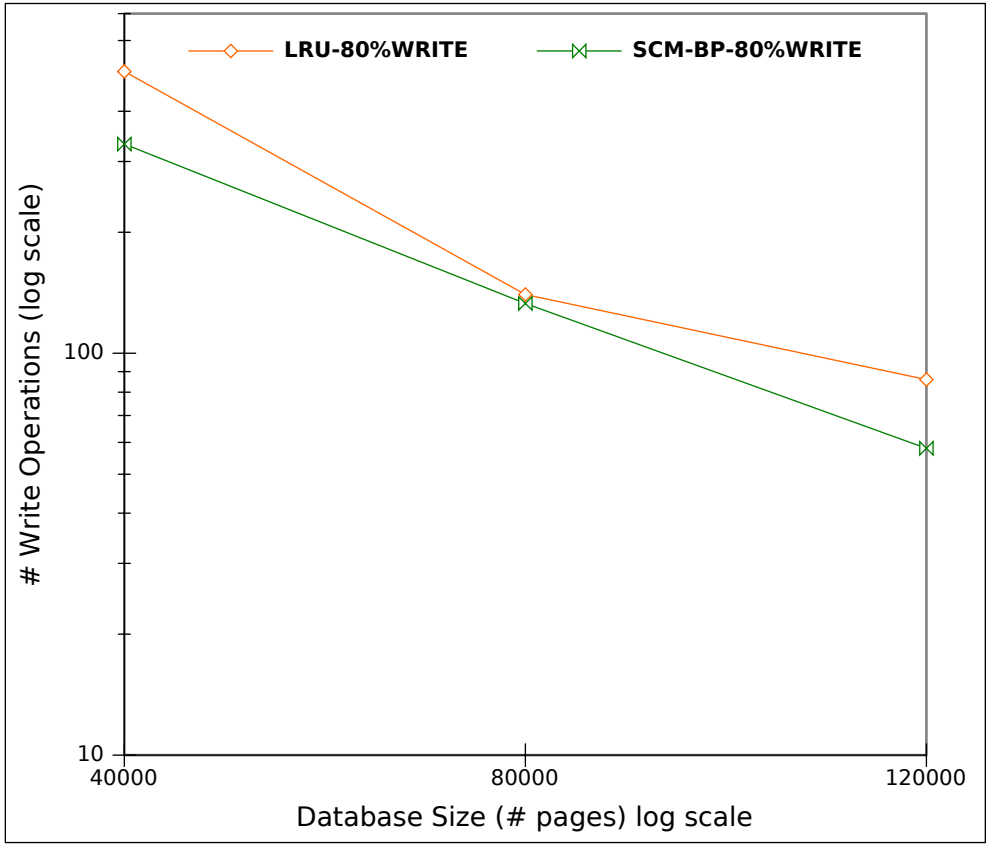




Buffer Replacement Policy

- SCM-BP

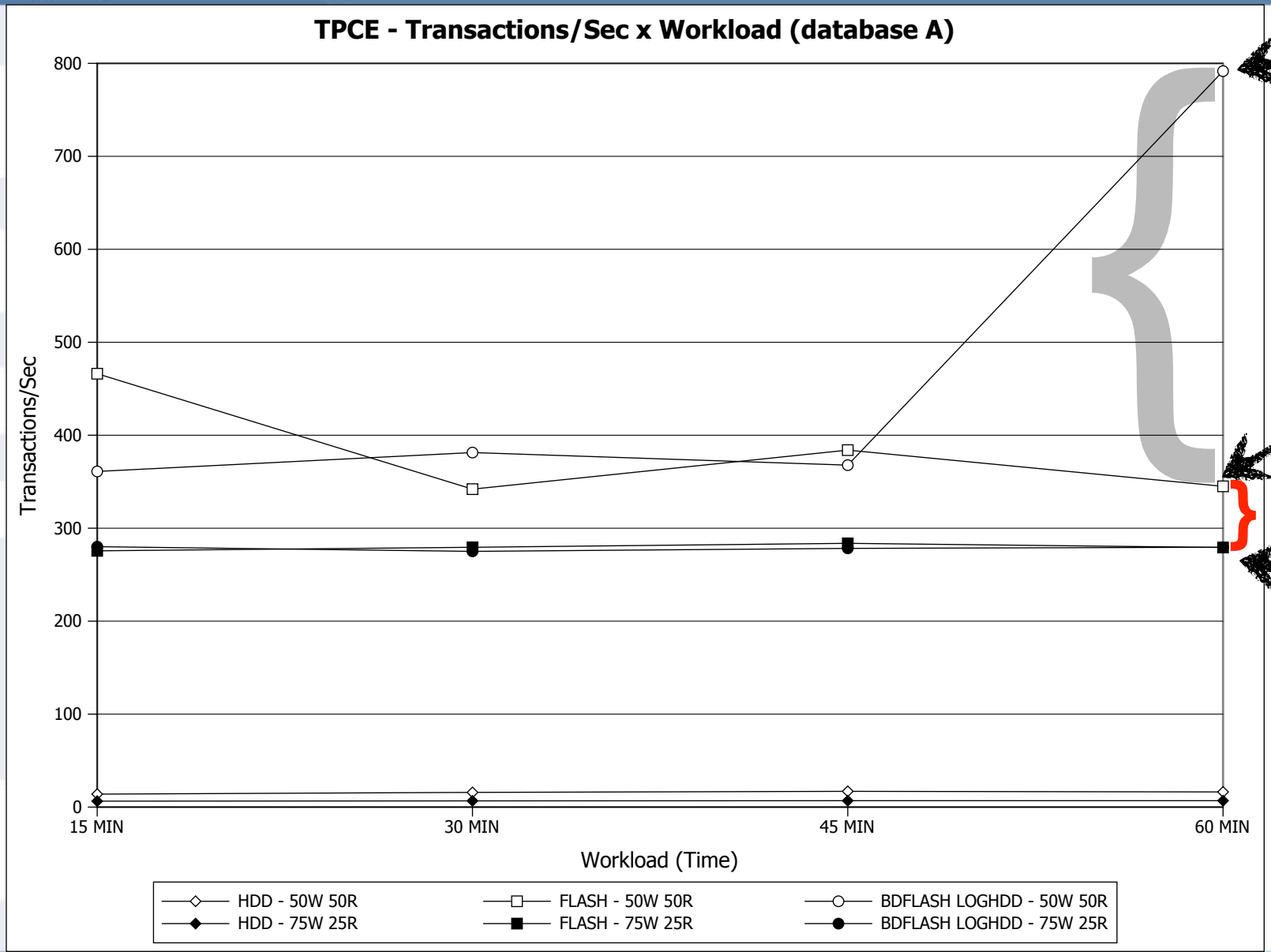




Logging

- Log File
- Log records for each transaction
  - BOT
  - For each write operation
  - EOT (commit/rollback)
- Writing into the Database Log File
  - Write ahead log
  - Force log at commit

Logging



5 wt + 5 rt
Log file on HD

5 wt + 5 rt

7 wt + 5 rt

Research Challenges

- Write-intensive components of DBSs
  - Negatively impact SSD's "write" bandwidth
- To take full advantage of SSDs
  - DBS components should be redesigned
- Query Engine
  - Query execution cost models
  - Heuristics for solving the join ordering problem
  - Physical operators
  - Pipelining query execution

Research Challenges

- Buffer Management
  - Hold in buffer pool write-intensive pages
- Recovery Management
  - Logging strategy
    - New recovery process

Ref

Research Challenges

- Buffer Management
  - Hold in buffer pool write-intensive pages
- Recovery Management
  - Logging strategy
    - New recovery process

- "The end of an architectural era: (it's time for a complete rewrite)"

Ref

Research Challenges

- Buffer Management
  - Hold in buffer pool write-intensive pages
- Recovery Management
  - Logging strategy
    - New recovery process

- "The end of an architectural era: (it's time for a complete rewrite)"

Write is expensive !

Ref

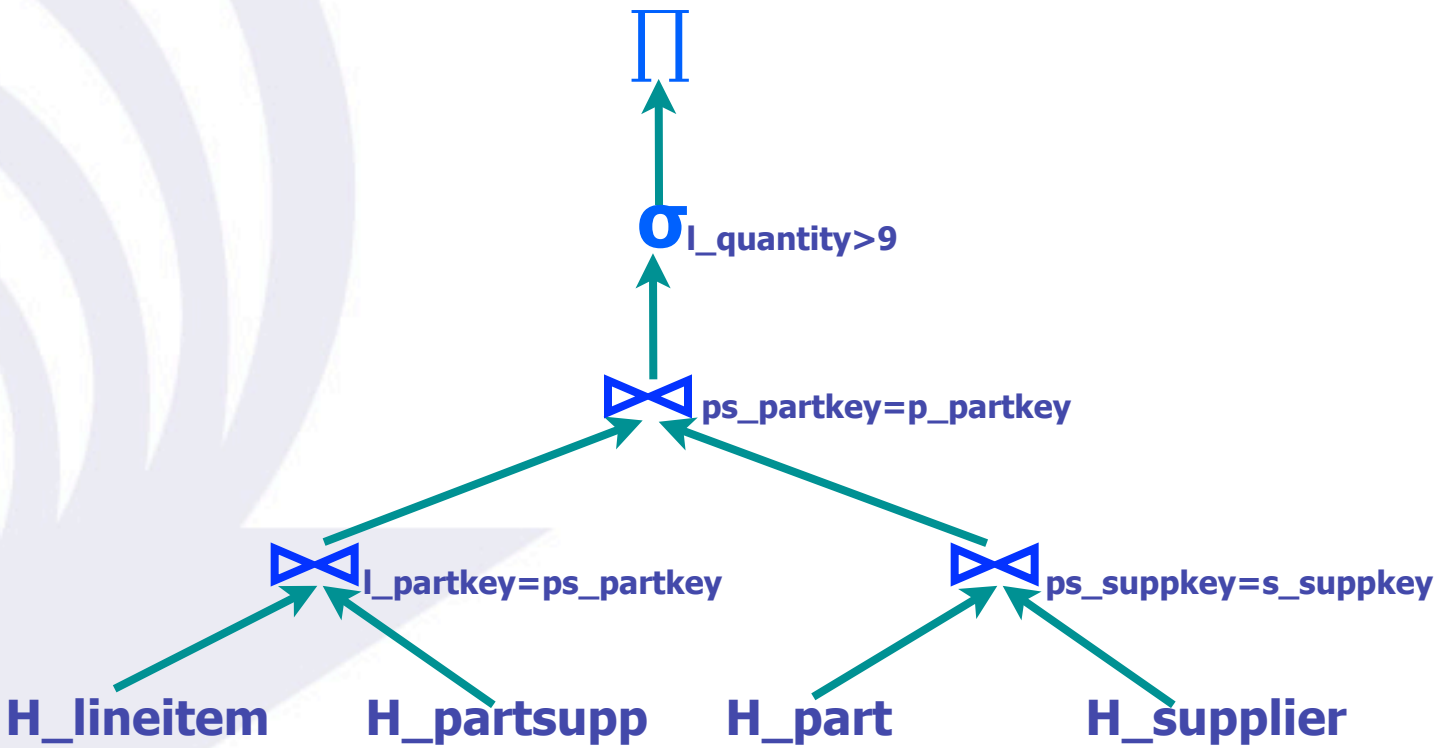
Query Optimization

```
select *
from H_lineitem, H_partsupp, H_part, H_supplier
where l_partkey=ps_partkey and
      p_partkey=ps_partkey and
      ps_suppkey=s_suppkey and
      l_quantity>9
```

Back

Query Optimization

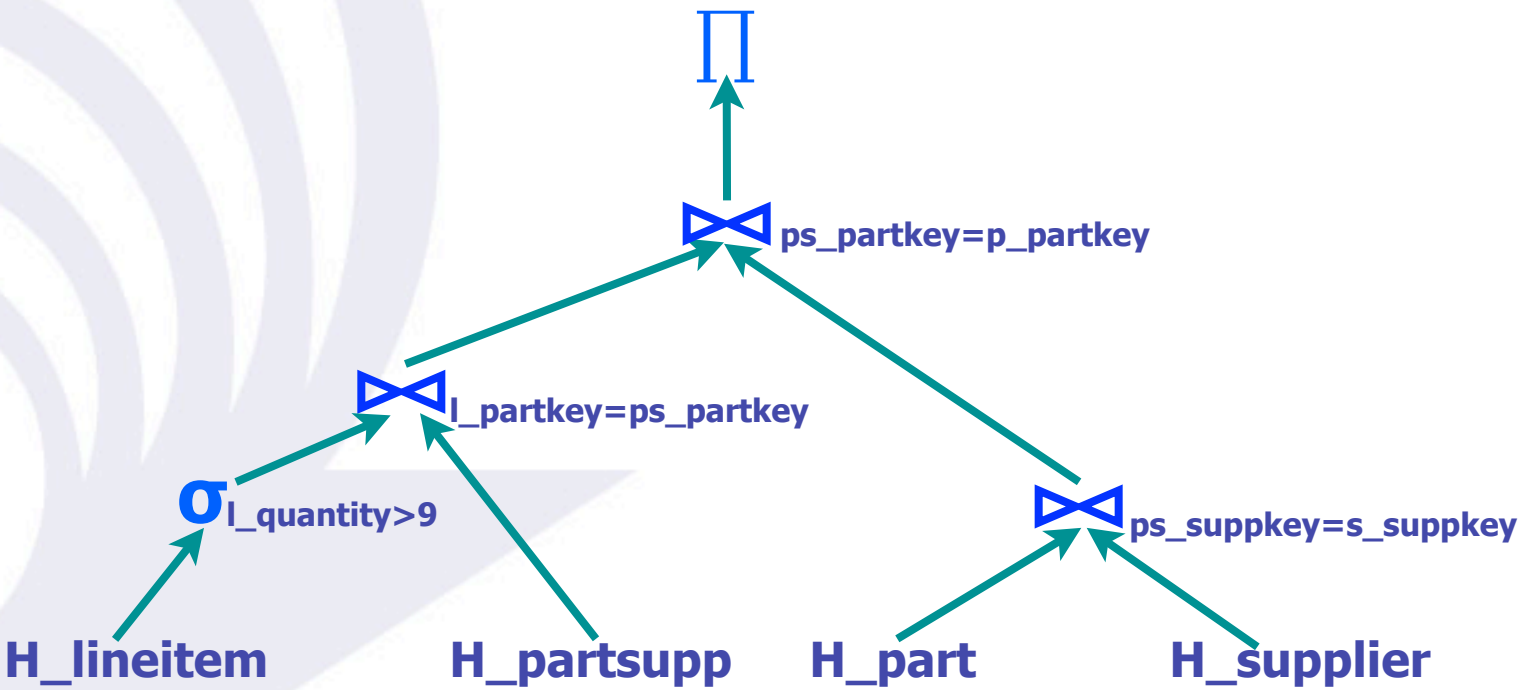
```
select *
from H_lineitem, H_partsupp, H_part, H_supplier
where l_partkey=ps_partkey and
      p_partkey=ps_partkey and
      ps_suppkey=s_suppkey and
      l_quantity>9
```



Back

Query Optimization

```
select *
from H_lineitem, H_partsupp, H_part, H_supplier
where l_partkey=ps_partkey and
      p_partkey=ps_partkey and
      ps_suppkey=s_suppkey and
      l_quantity>9
```



Back

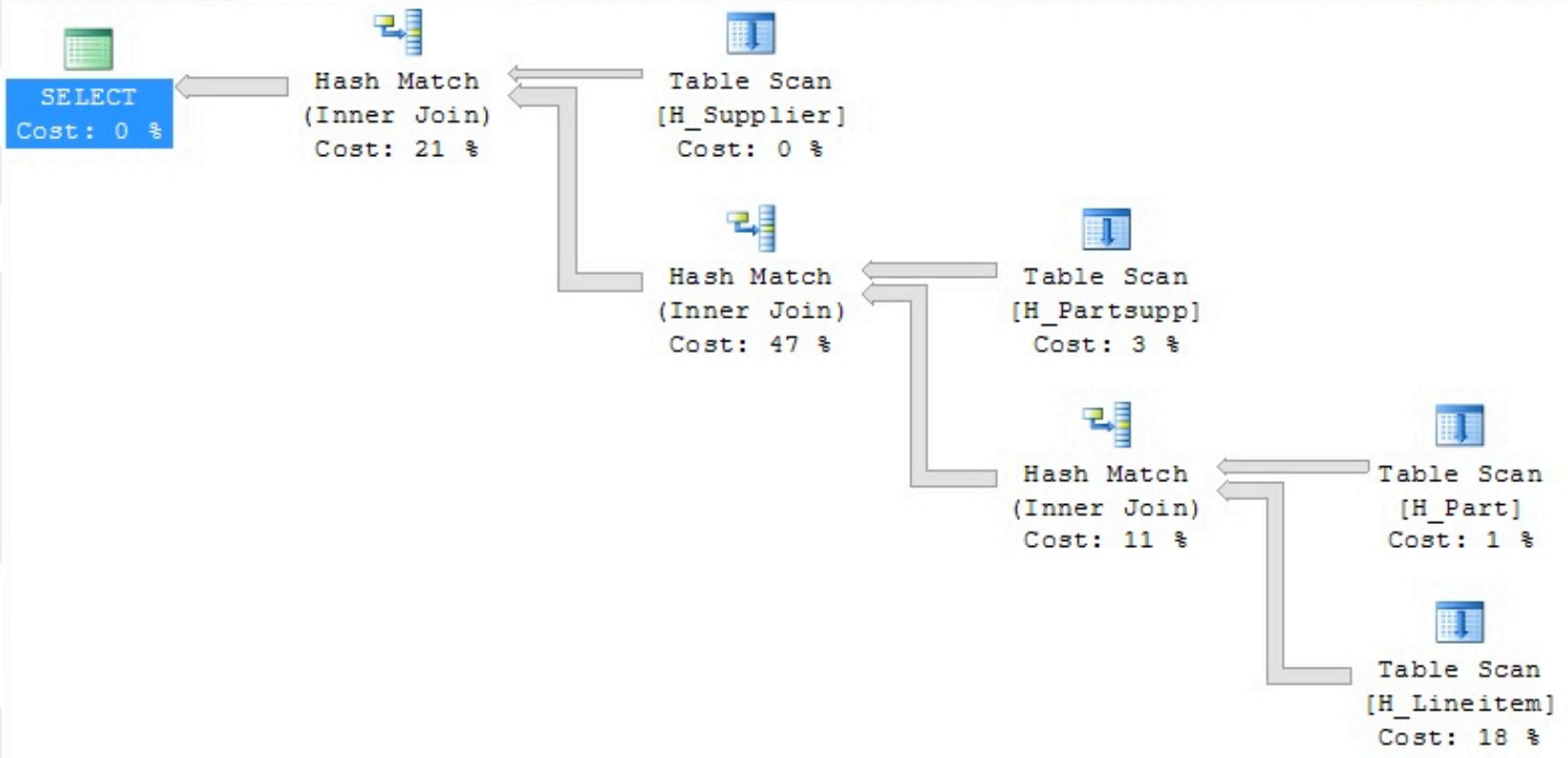
Query Optimization

SQLQuery2.sql - LAR...ares-PC\lares (52))* | Object Explorer

```
select * from H_Lineitem, H_Partsupp, h_part, H_Supplier
where l_partkey=ps_partkey and p_partkey=ps_partkey and ps_suppkey=s_suppkey
and l_quantity>9
```

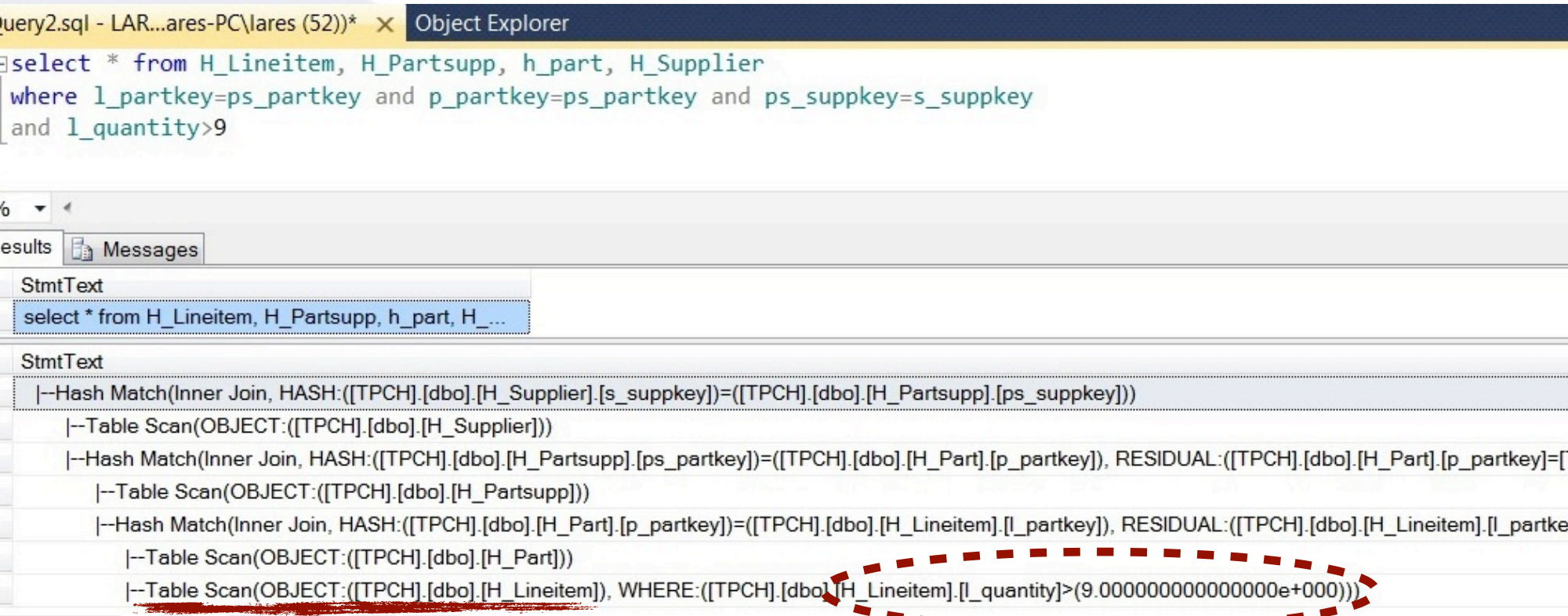
100 %

Messages | Execution plan



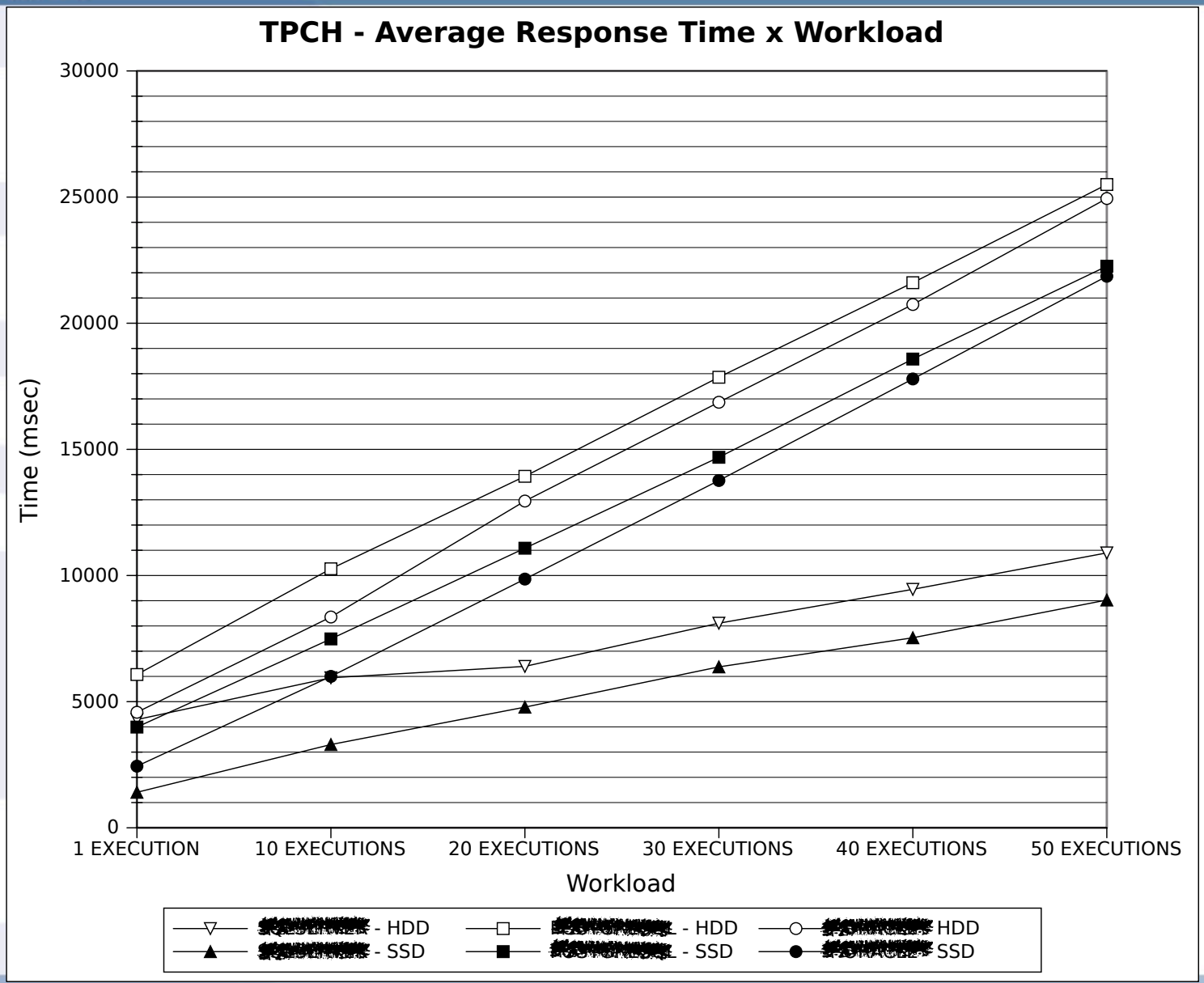
QEP exec

Query Optimization



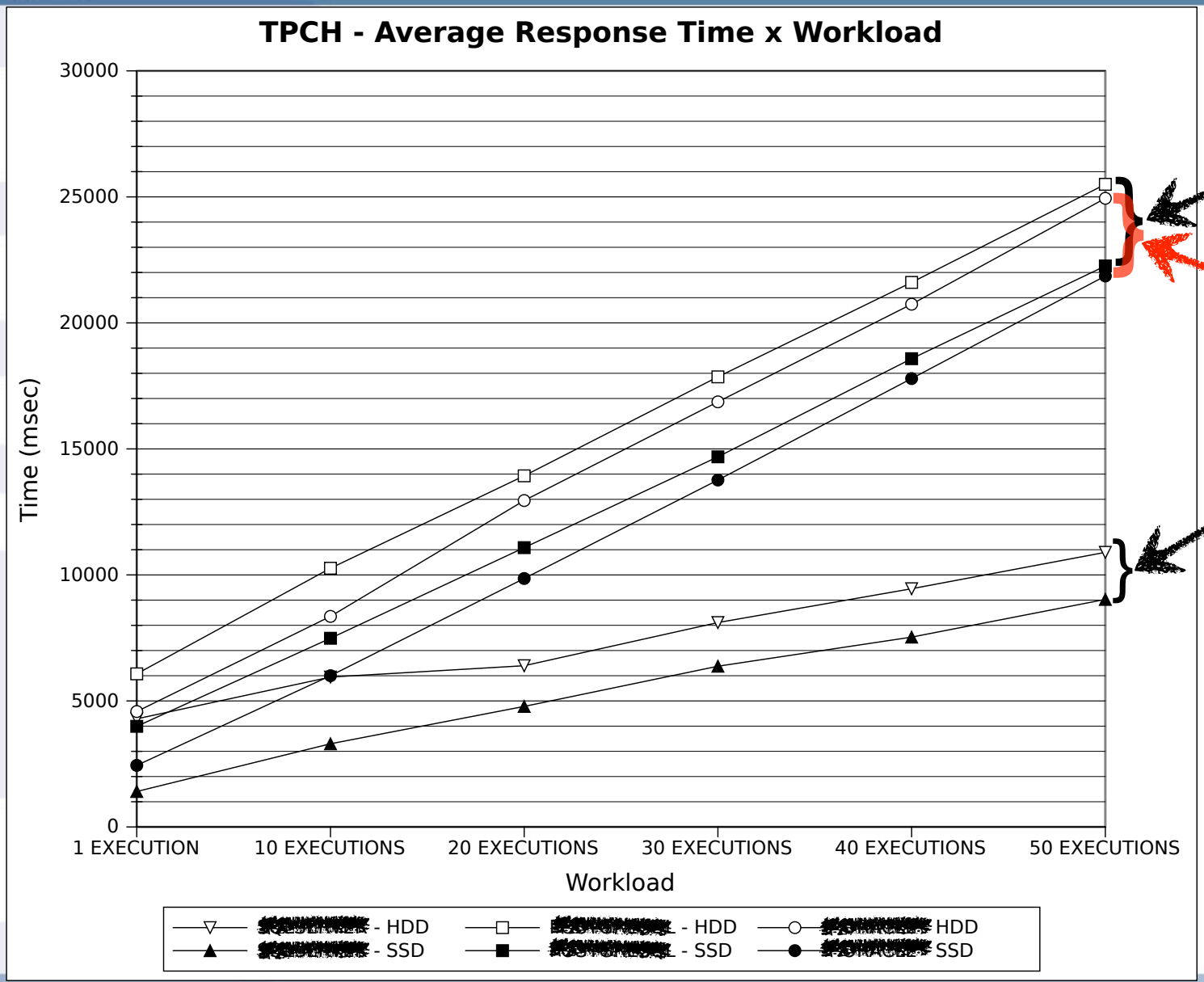
Back

SSD x HD: Query Optimization



Back

SSD x HD: Query Optimization



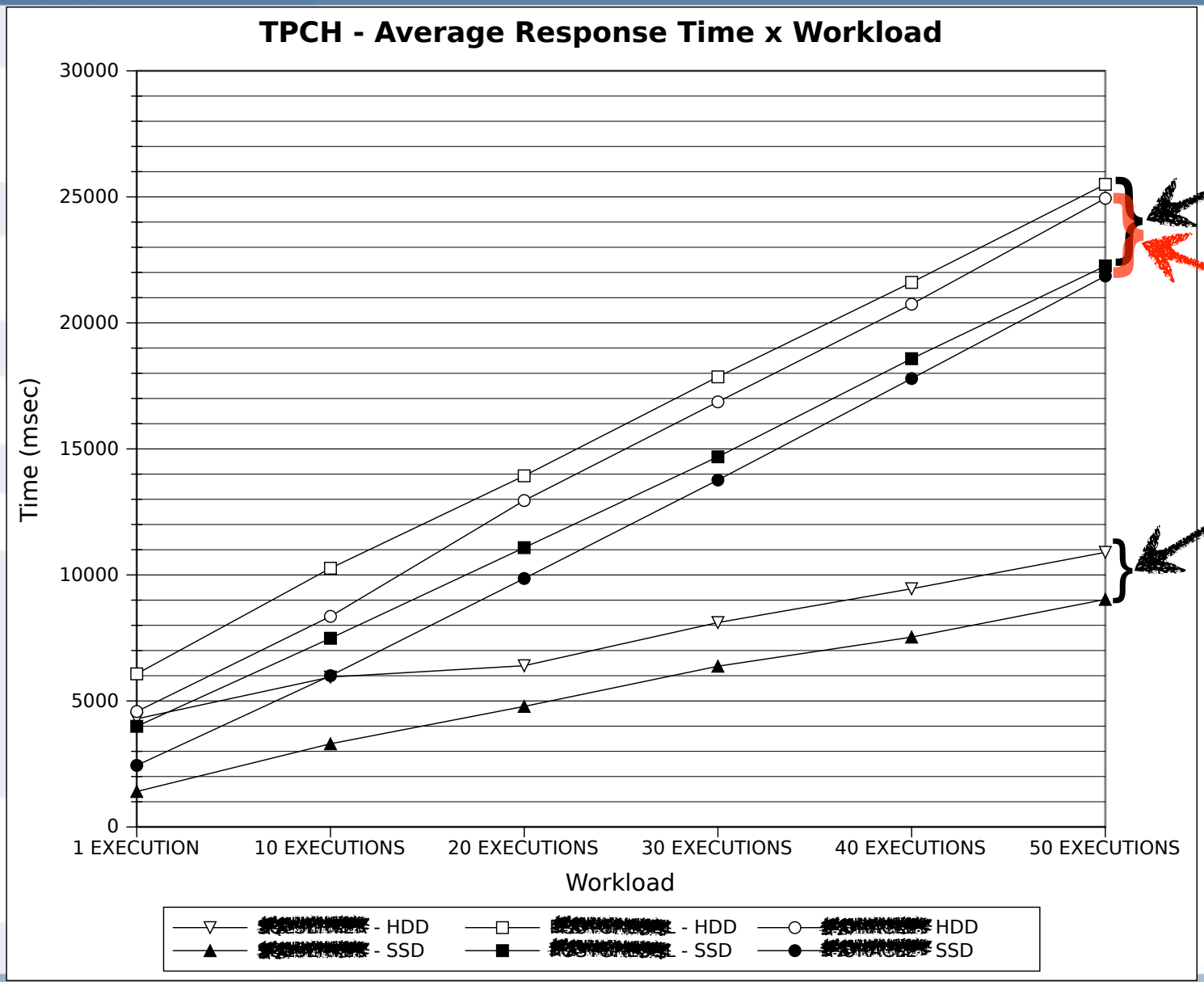
~ 3.5 sec

3 sec

2 sec

Back

SSD x HD: Query Optimization



~ 3.5 sec

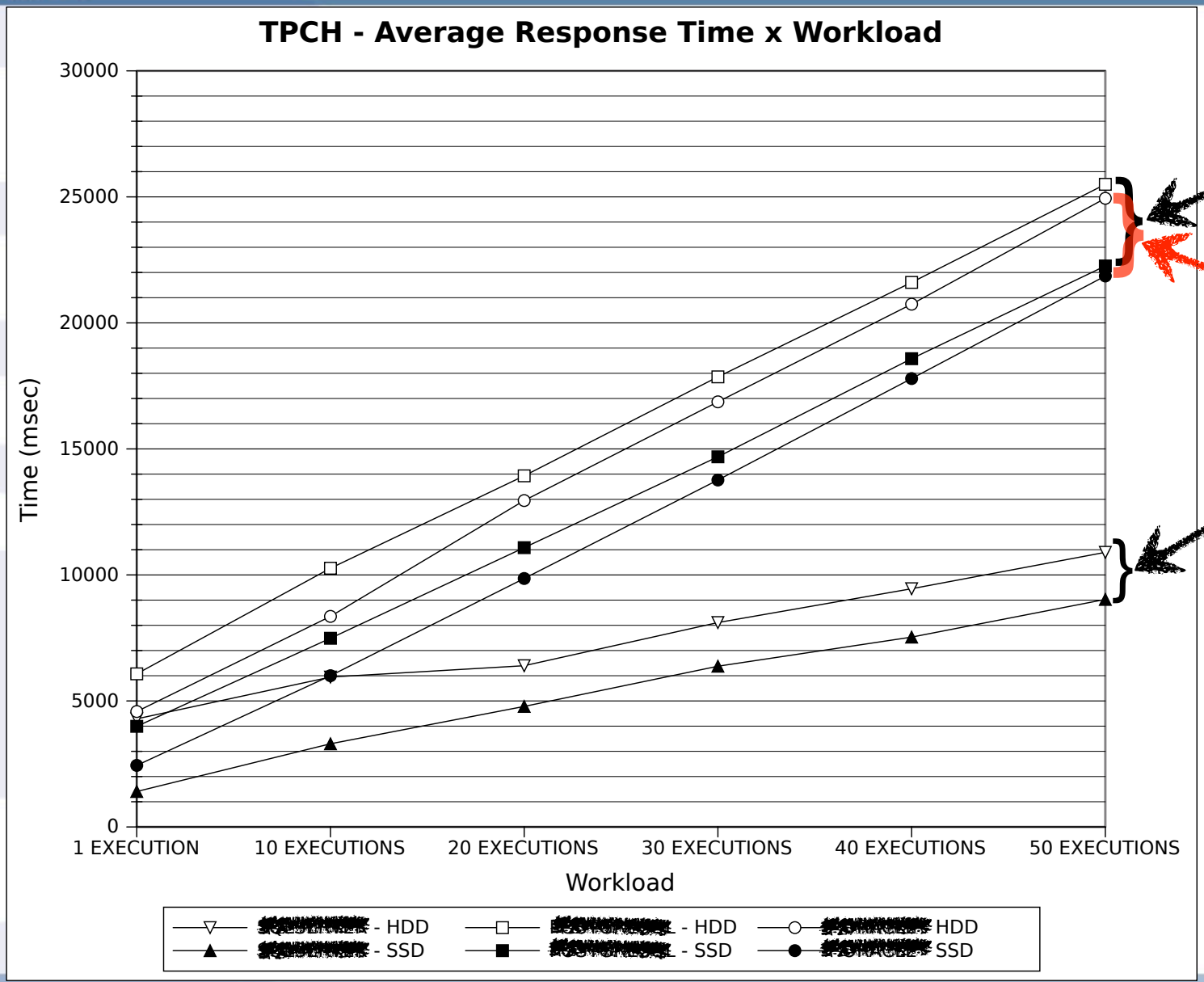
3 sec

2 sec

SSD: IOPS over 100 times greater than HDD !

Back

SSD x HD: Query Optimization



~ 3.5 sec

3 sec

2 sec

SSD: IOPS over 100 times greater than HDD !

A write is up to 2 orders of magnitude slower than a read

Back

References

- David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, David A. Wood. Implementation techniques for main memory database systems. In proceeding of: SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984
- G. Graefe. Query Evaluation Techniques for Large Databases. ACM Comput. Surveys, 25(2):73-170, June 1993
- Jung, H., Shim, H., Park, S., Kang, S., and Cha, J. LRU-WSR: Integration Of LRU And Writes Sequence Reordering For Flash Memory. IEEE Transactions on Consumer Electronics 54 (3): 1215–1223, 2008.
- Ou, Y., Härder, T., and Jin, P. CFDC: A Flash-Aware Replacement Policy For Database Buffer Management. In Proceedings of the International Workshop on Data Management on New Hardware. New York, NY, USA, pp. 15–20, 2009.
- Stonebraker, M., Madden, S., Abadi, D.J., Harizopoulos, S., Hachem, N., and Helland, P. The end of an architectural era: (it's time for a complete rewrite). In Proceedings of the 33rd international conference on Very large data bases, VLDB '07, pages 1150–1160. VLDB Endowment, 2007
- Tsirogiannis, D., Harizopoulos, S., Shah, M. A., Wiener, J. L., and Graefe, G. (2009). Query processing techniques for solid state drives. In Çetintemel, U., Zdonik, S. B., Kossmann, D., and Tatbul, N., editors, SIGMOD Conference, pages 59–72. ACM.

Obrigado!!