Internals of Database Systems: Query Execution

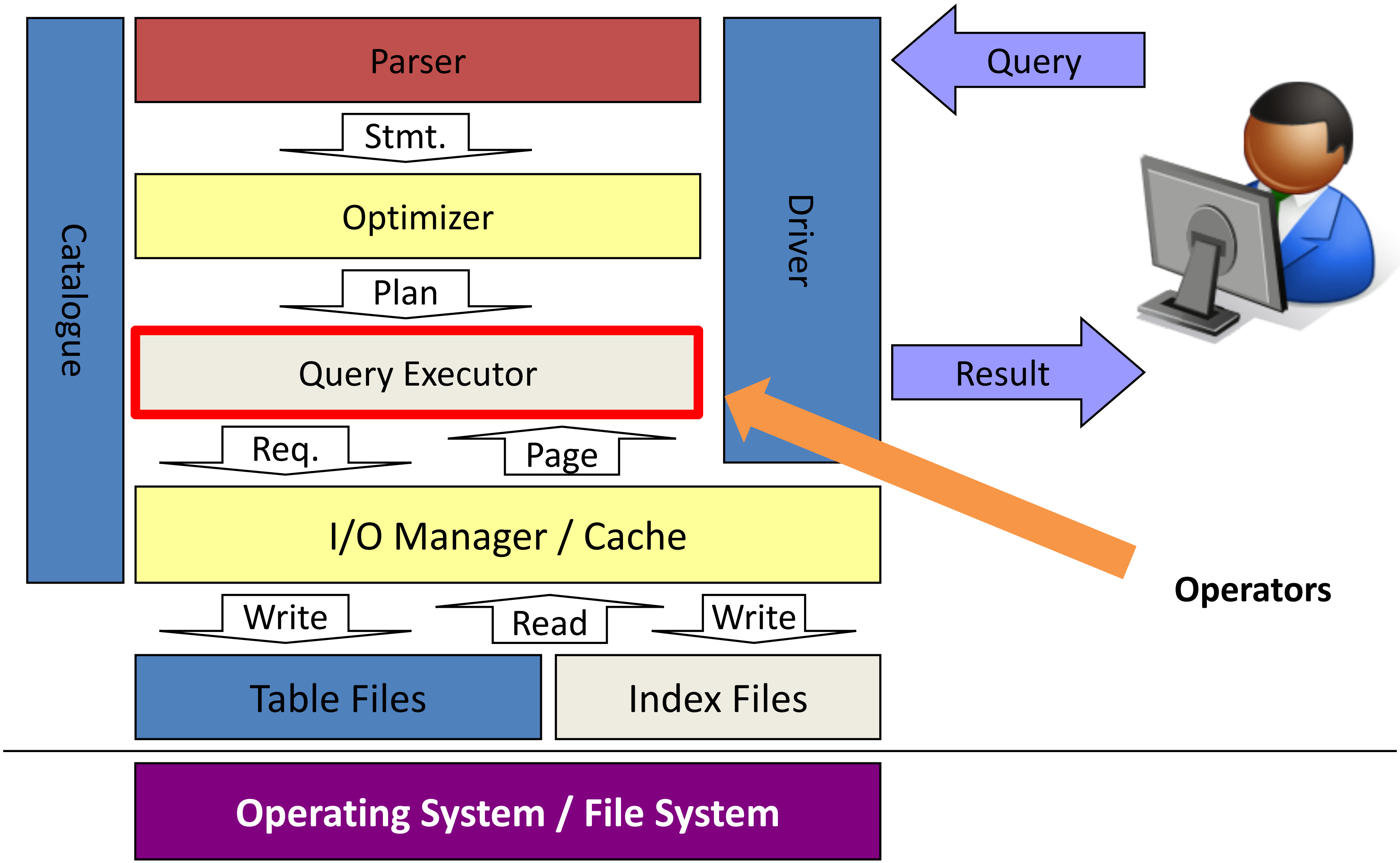
Dr. Asterios Katsifodimos

Material from Kostas Tzoumas, Volker Markl (recursively Felix Naumann, Len Shapiro), Stratis Viglas, Goetz Grafe

A note

- Changed the organization of this lecture
  - We will not follow the book
  - You should read the book nonetheless!
- Operator- and Algorithm driven
  - Sort-based, Hash-based, Index-based
- Feel free to interrupt with questions

Where we are…



Catalogue
Parser
Stmt.
Optimizer
Plan
Query Executor
Req.
Page
I/O Manager / Cache
Write
Read
Write
Table Files
Index Files
Driver
Query
Result
Operators
Operating System / File System

Outline

- Operators & Query Plans

- Iterator Model

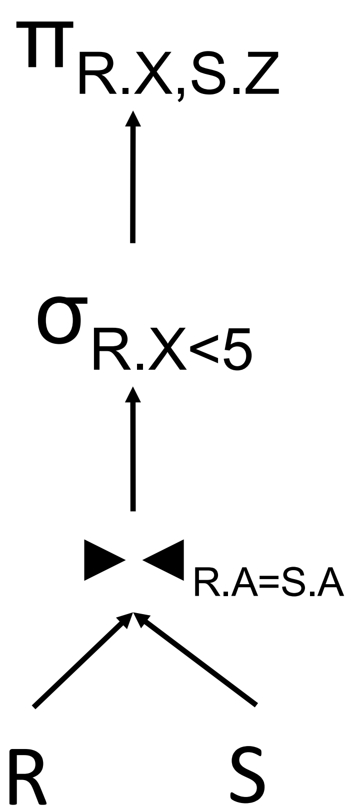
- Hashing and Sorting
- Join algorithms
- Aggregation
- Set operations

OPERATORS (LOGICAL & PHYSICAL)

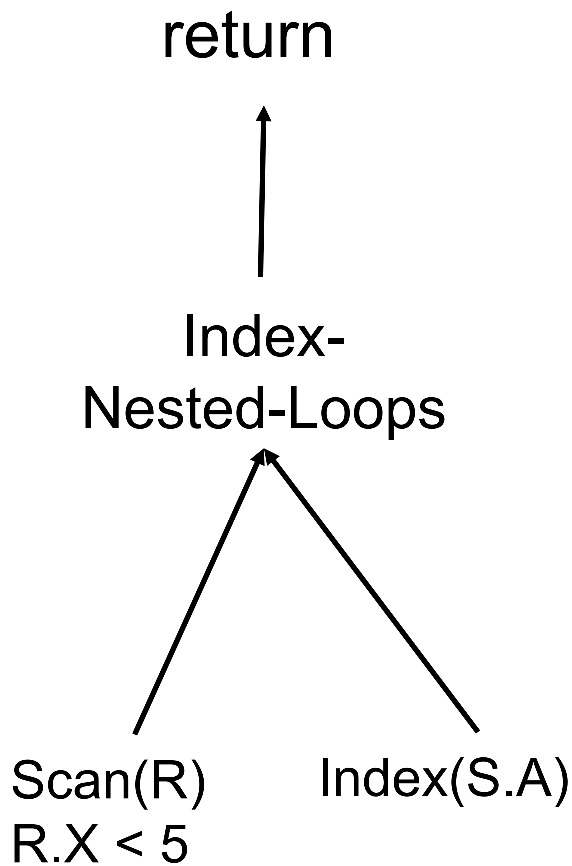
Operators

- Logical operators describe what happens
- What are the five relational algebra operators?
  - π, σ, x, $\cup$, -
  - γ, $\bowtie$, $\cap$
- Physical operators dictate how these are implemented exactly – aka algorithms
  - Several implementations of a logical op (examples?)
  - Several logical operators may use the same physical (examples?)

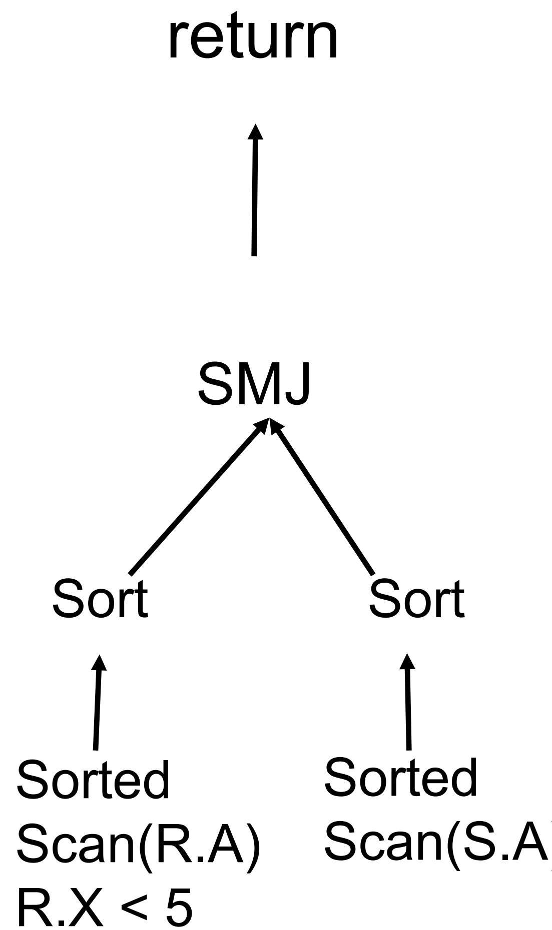
Logical and Physical Plans



Logical Plan



Physical Plan 1



Physical Plan 2

Why Different Physical Operators?

- Example: Table Scan
  - Read an entire table and apply selection predicate

- Physical implementations
  - Sequential scan No info (robust), Large selectivity, Time-to-first-tuple
    - Read disk blocks (known position) in order
  - Index scan Small selectivity
    - Index points to disk blocks (primary vs. secondary index)

- When would you choose what?

Logical-Physical Mapping

- Relation → Scan
- σ → filter, or Index-Access
- π (with duplicates) → Trivial
- x → Nested-Loops-Join
- ⋈ → Hash-, Sort-Merge-, Index-Nested-Loops-Join
- γ → Hash-aggregation, sorted-aggregation
- π (eliminating duplicates) → Special case of aggregation
- ∩ → Special case of a join
- - (difference) → Inverse case of a join (anti join)
- U → Union

Minimal set of Physical Algorithms

- Scan → Relation access
- Index-Lookup / Index Scan → σ
- Filter → σ
- Project → π *(keeping duplicates)*

- (Block)-Nested-Loops → x
- Index-Nested-Loops → ⋈, ∩, -

- Sort, merge, sorted-group → γ, ⋈, ∩, -

- Hash-Agg/Join → γ, ⋈, ∩, -, π

- Union → U

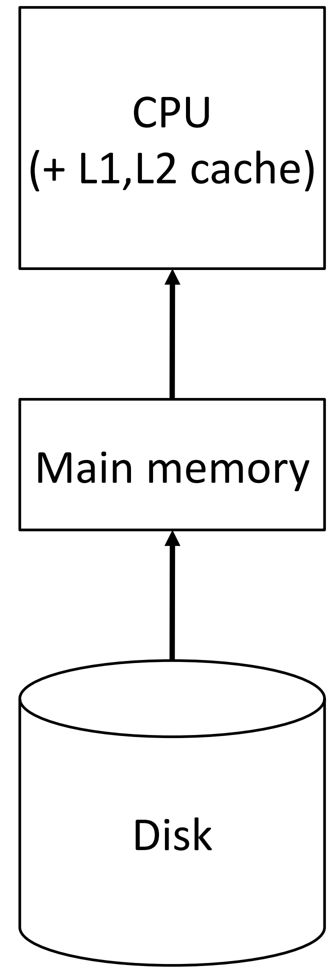
ALGORITHMS

Measuring Cost

- Cost = CPU, **I/O**, Network
- Assumption: Input read from disk → Cost dependent on input size
- Cost of physical operator depends on many parameters
  - Data distribution, available memory, caching effects, parallel operations, etc
  - We don't know the exact cost before executing – we use a highly simplified model to make rough predictions!
- Purpose of cost estimation: Choose which algorithm to use when

External Memory Model

- M=size of available main memory in #disk blocks (approximate)
- Disk → Memory → CPU cache → CPU (what do we measure?)
- Parameters
  - B(R) = #relation blocks
  - T(R) = #relation tuples
  - V(R,A) = #values of attribute A in R
  - f(R,A) = distribution of attribute A in R
- Sequential vs. random I/O



Example: Table Scan

- Cost(*SeqScan(R)*) = B(R)

- Blocks of R sequentially stored on disk or not
  - B(R) sequential accesses or B(R) random accesses

  - In practice: Databases and Filesystems try to sequentialize the blocks layout on disk as much as possible, so we typically assume sequential I/O

Example: Index Scan

- Here: Index is B-Tree
- Let *s* = # of tuples in *σ(R)*

- Cost(*IndexScan(R,σ)*) = 1 + ceil(s * B(R) / T(R))

Table Clustered by Index

- Cost(*IndexScan(R,σ)*) = 1 + s

Table Unclustered

Assume one page lookup in index (simplified, may be more, but always a small number)

Algorithms discussed here:

- (Block) Nested Loops
- Index Access and Index Nested Loops
- Sorting / Hashing
  - Serve same purpose
    - "Match" or "Group" items that are alike
  - Require techniques to handle case where data is larger than available main memory

Nested Loop Join

$Cost_{NLJ}(R,S) = B(R) + T(R)B(S)$

One scan of outer, T(R) scans of inner

```
for (r in R)
    for (s in S)
        if (r.A==s.A) return rs
```

Assume:

T(R) = 100,000

T(S) = 1,000,000

100 tuples per page

10 ms per I/O (random)

0,02 ms (sequential)

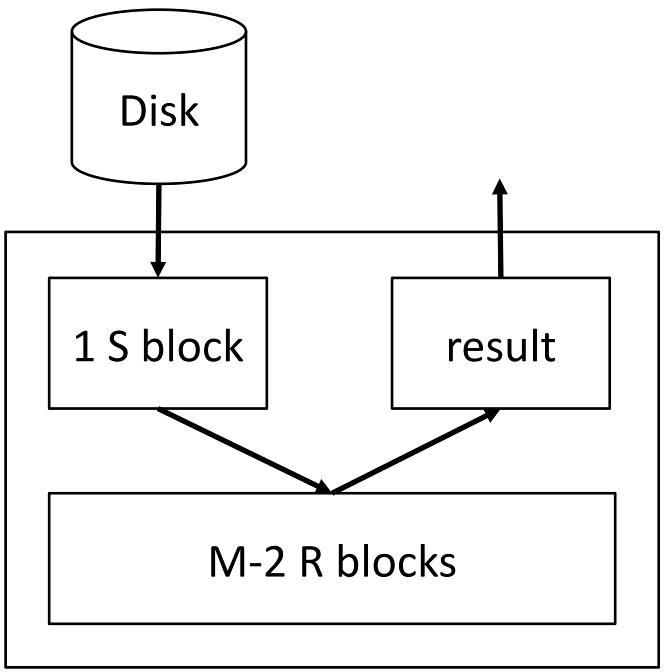
How much does it cost?

115 days (random), 5.8 hours (sequential)

How to improve this?

Block Nested Loop Join

- Assume we have M blocks in main memory
- Fetch as many as possible R blocks into memory (how many?)



while (R tuples left)
  R_M = fetch M-2 R blocks
  for each (page of S)
    emit all matching tuples $R_M S$

$Cost_{BNLJ}(R,S) = B(R)+B(S)B(R)/(M-2)$

Assume 100 disk blocks fit in memory (M=100). What is the cost for the previous example?

2.3 seconds (sequential I/O)

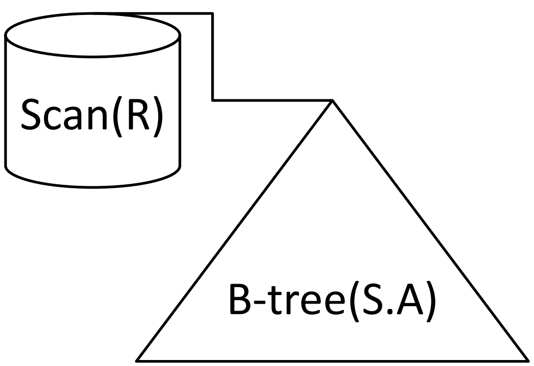
Which relation should be the inner?

The bigger one. Inner is scanned #times
Depending on size of outer.
Attention: Cost(RS) != Cost(SR)

Index Nested Loop Join

Assume index in S

```
for (r in R)
  a = r.A
  for (s in S where s.A==a)
    return rs
```



f = selectivity of join
(% of Cartesian product)
i = average I/Os for index lookup

Secondary index:
$Cost_{INLJ}(R,S) = T(R)*i + T(R)T(S)*f$

Assume result size 500, i = 1
How much does it cost for the example?

16 seconds, if index I/O is random

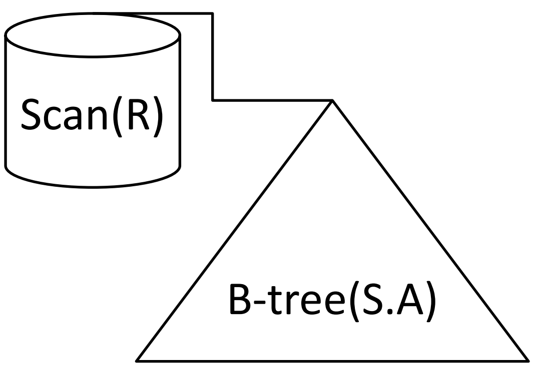
Index Nested Loop Join

Assume index in S

Assume R is SORTED
→ Index I/O largely sequential

f = selectivity of join
(% of Cartesian product)

Secondary index:
$Cost_{INLJ}(R,S) = T(R) * i + T(R)T(S)*f$



Assume result size 500, i = 1
How much does it cost for the example?

2 seconds

Sorting

Use a main-memory sort algorithm

- Quicksort, Mergesort, Heapsort, Radixsort, …

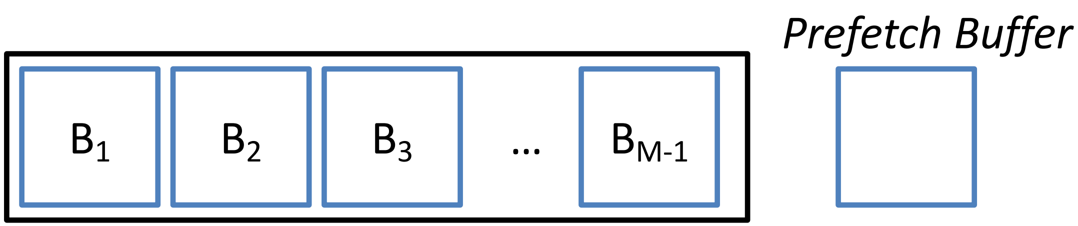
If main memory is not enough: External Sort (Merge Sort)

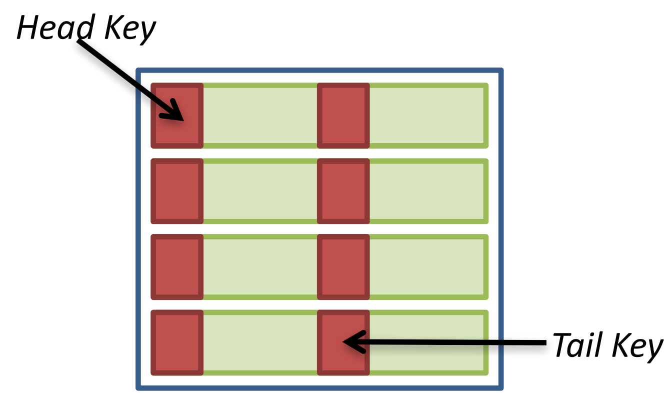
- Creating initial sorted runs
  - Using main memory sort algorithm to create : B(R)/M runs

- Merging of runs
  - Always keep one buffer for output
  - Remainder of buffers to read inputs (= F, merge fan-in)

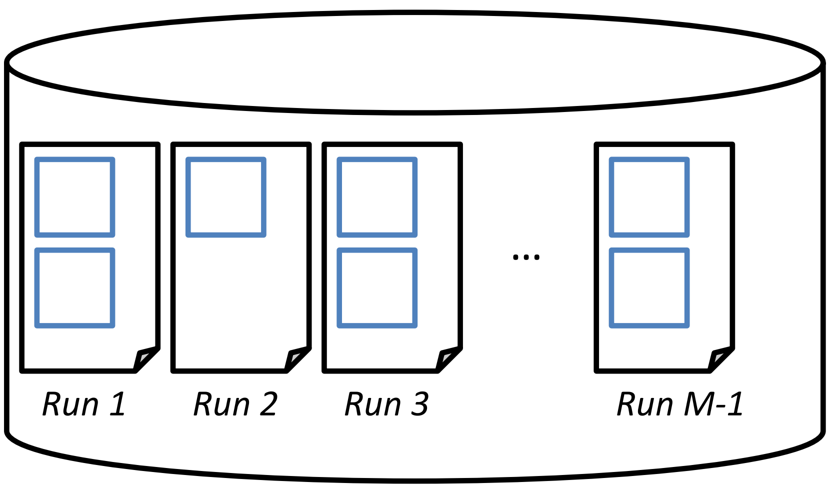
Merge Phase – Data Structures

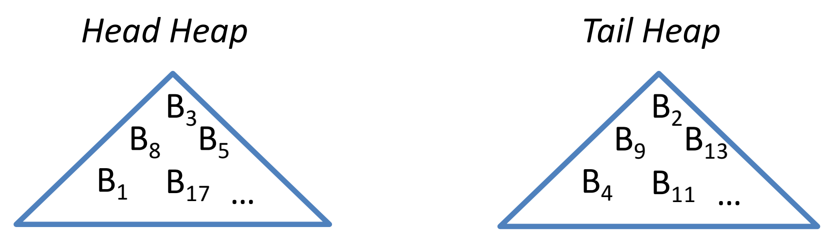
- Uses *M* memory buffers, merges *M-1* streams
- Head block of each run is in memory





Memory Buffer Layout

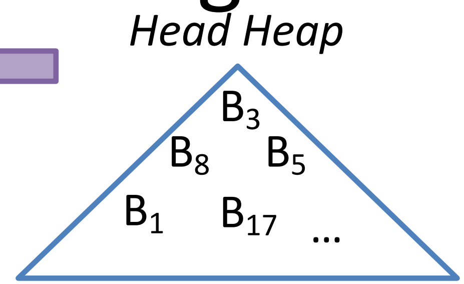




Two priority queues (aka. heaps)
One on the head key, one on the tail key

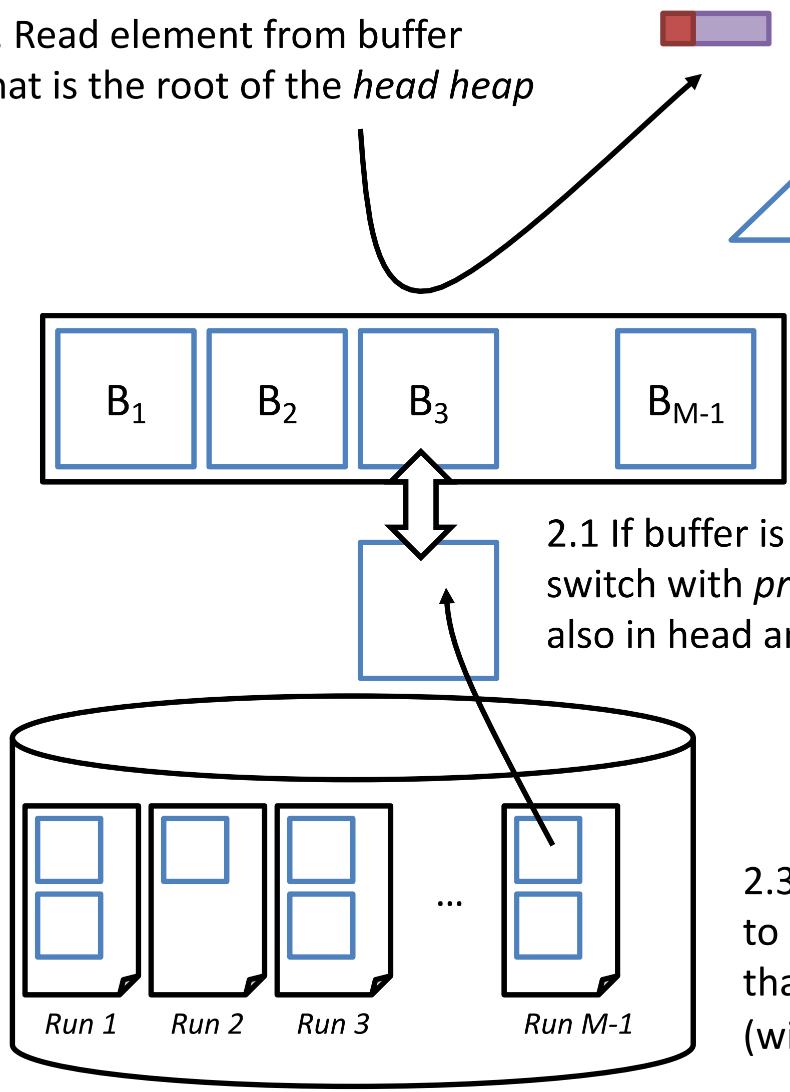
Merge Phase – Reading the next Element

1. Read element from buffer that is the root of the *head heap*

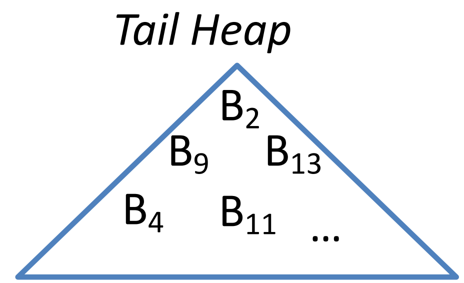


- $\log_2(n)$ each time for head-heap update
- $\log_2(M-1)$ for each buffer replacement

3. Update head-heap



2.1 If buffer is now consumed, switch with *prefetch buffer,* also in head and tail heap.



2.2 If prefetch buffer was switched, update *tail heap*

2.3 Use new prefetch buffer to load next block from stream that is root of *tail heap* (will be the next needed block)

Sort-Merge Join

- Sort both relations on join key
- Merge sorted relations
- Focus on merge phase
  - Complication: Groups in sorted relations with same join key. Need to generate all matches

Merge Join Cost

- *3*B(R) to sort R*, 3*B(S) to sort S
  - In practice most relations can be sorted in two passes
- Merging
  - In practice, one pass *B(R) + B(S)*
  - Worst case *O(B(R)*B(S))*
    - When s.A=a1. Very rare
  - Large groups can be problematic.
- $Cost_{SMJ}(R,S)=5B(R) + 5B(S)$
- *read, write sorted runs, read and merge runs, write, read and join*

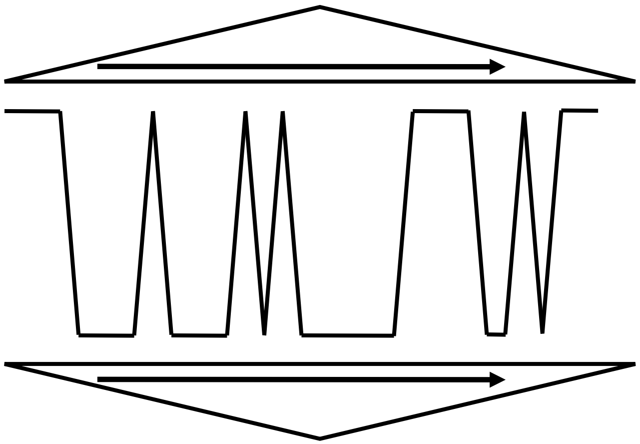
Refinement

- Idea: Combine merge phase of join with merge phase of sorting
- Create sorted runs $R_1,\ldots,R_m$ and $S_1,\ldots,S_m$ of size > $sqrt(B(S))$. Then $m < sqrt(B(S))$
- Merge R pages, merge S pages, join in one step in main memory. Need 2*$sqrt(B(S))$ buffers
- Cost $3(B(R)+B(S))$
  - read, write, read & join

0.6 seconds, if all I/O considered sequential
In practice, how sequential is sort-merge phase?

Joining with Sorted Index

- Sorted, dense index, e.g. B-Tree
- Idea 1: Sort-Merge-Join, but only one relation has to be sorted before
- Idea 2: If both relations have a sorted index on Y: Just only Merge-Phase
  - „Zig-Zag-Join“
  - Record of R without join partner of S will never be read(and vice versa)



Hashing

- Equality operations only!

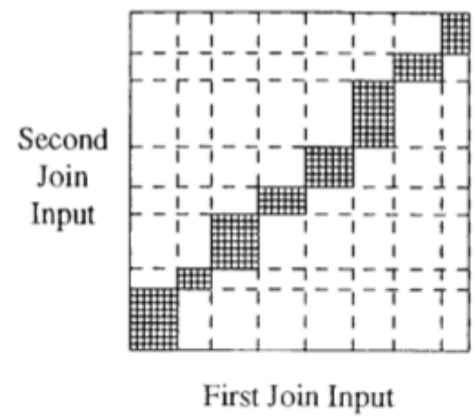
- Basic idea: Build a hash table in main memory

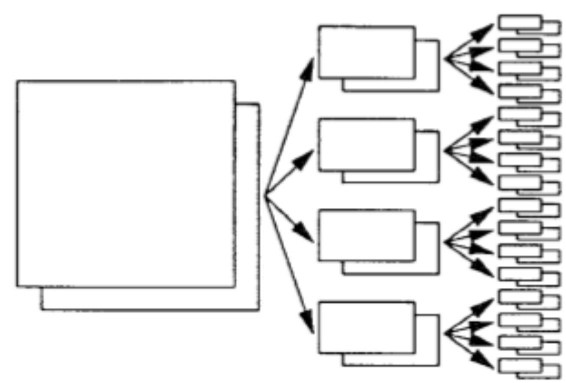
- If input fits into Main Memory, very fast
  - Exploits cardinality difference in binary operators
- Hash table overflow if Main Memory full
  - Solution: Partition input into subsets, processed independently
  - Fan-out F = number of partitions = M/C-1

- Two possibilities
  - Overflow avoidance (pessimistic): Partition first – Grace Hash Join
  - Overflow resolution (optimistic): Spill as needed - Hybrid Hash Join

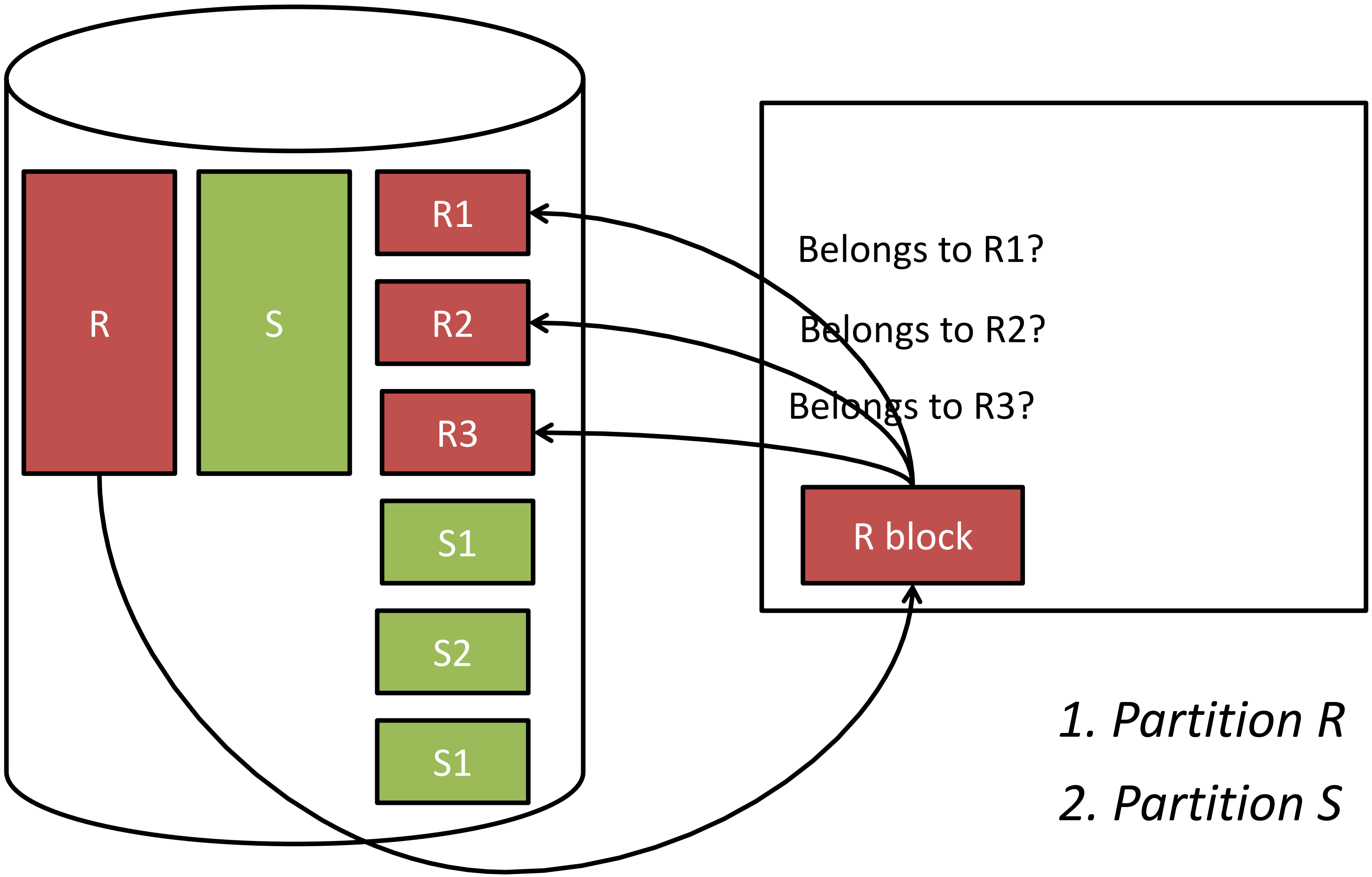
Hash Join in External Memory (Grace Hash Join)

- Idea: Partition *R,S* into *{R$_i$},{S$_i$}* such that the partitions fit in main memory
  - Either second hash function, or collections of hashes
- Build hash tables on *{R$_i$}*, probe *{S$_i$}*
  - When does this work?
  - Hash function must be on join key *R.A,S.A*
- May need to partition recursively until we have partitions that fit in memory





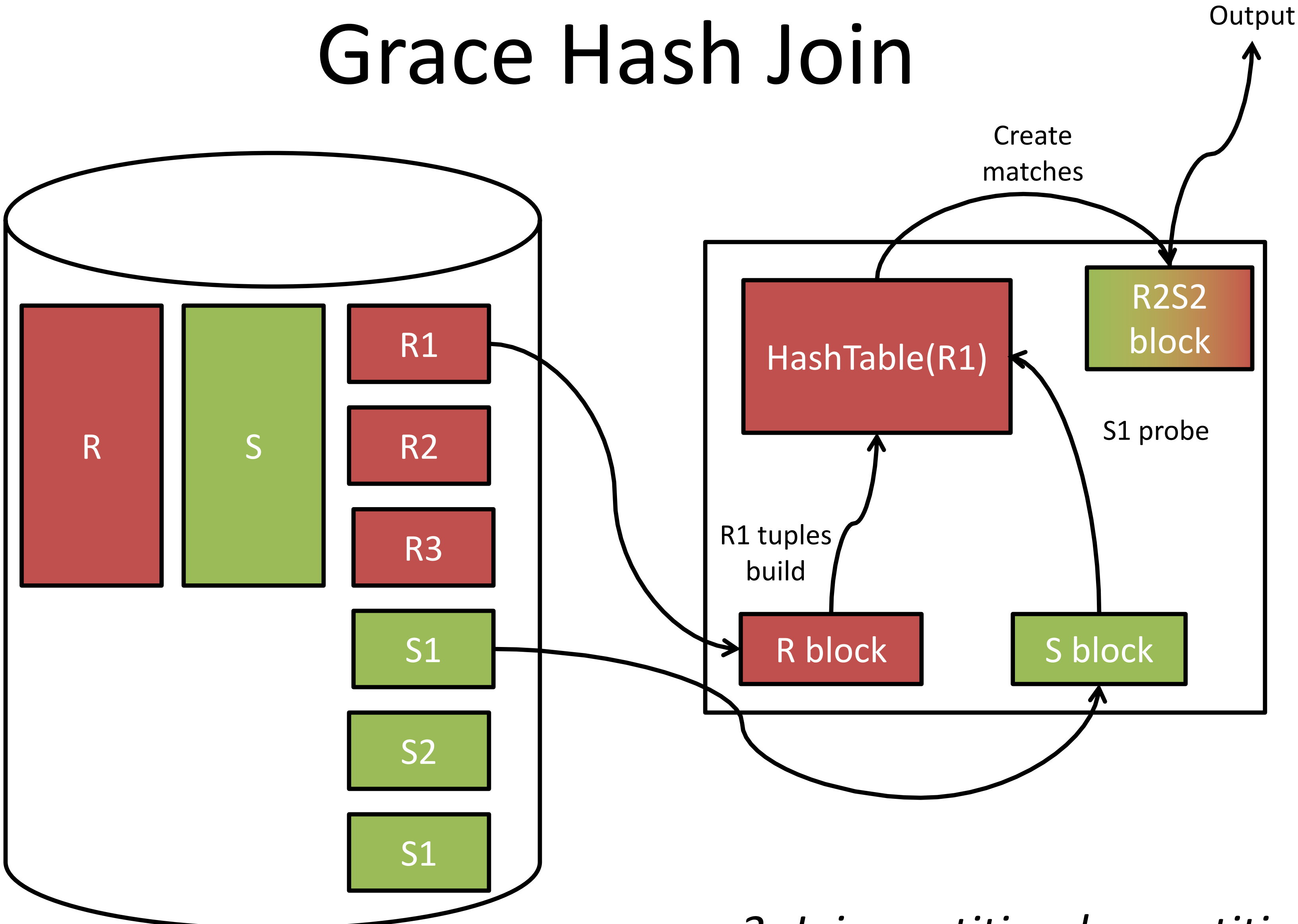
Grace Hash Join



1. Partition R

2. Partition S

Grace Hash Join



3. Join partition-by-partition

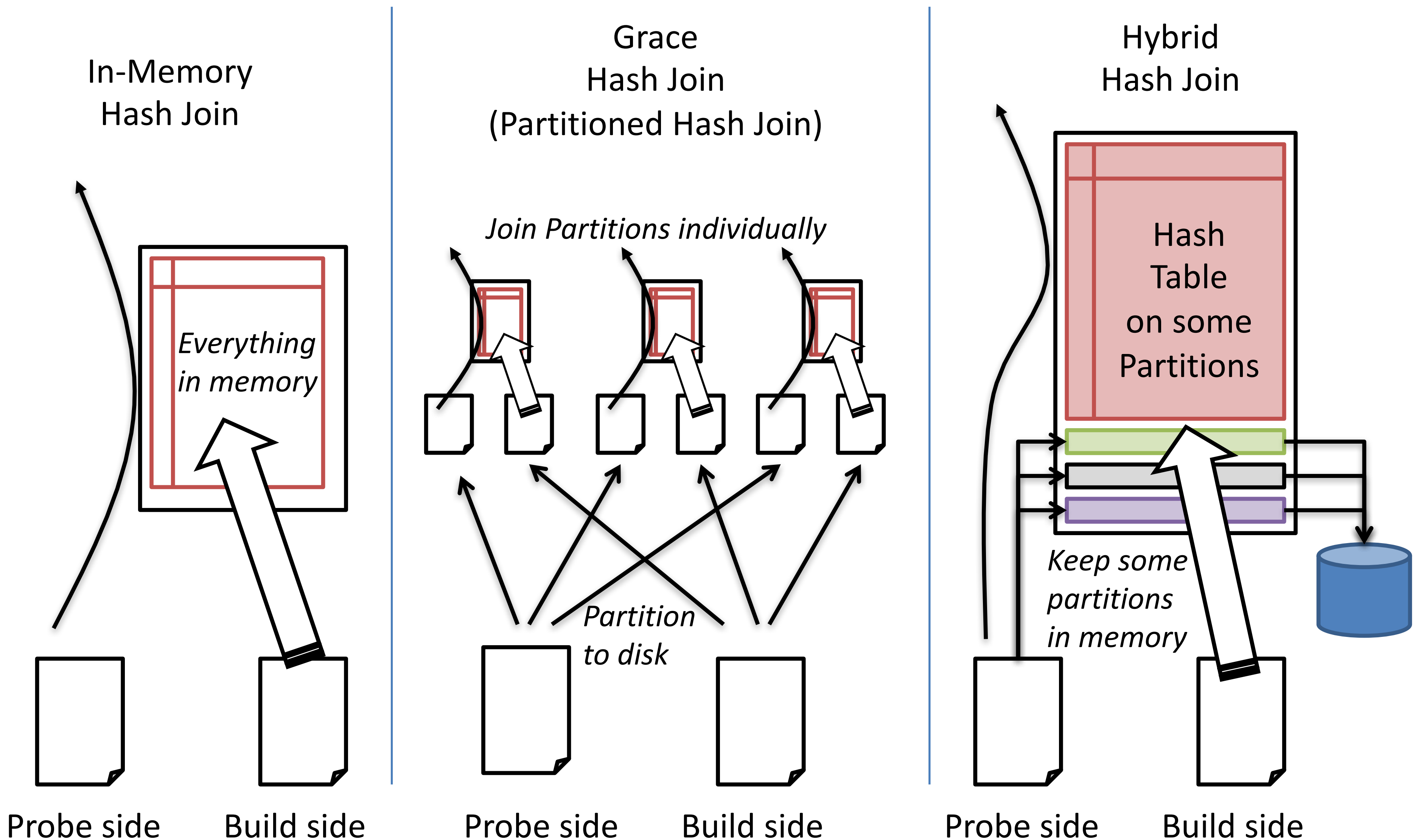
Grace Hash Join Cost

- Partition R
  - read and write: 2B(R)
- Partition S
  - read and write: 2B(S)
- Read R partition by partition
  - B(R)
- Read S partition by partition and probe
  - B(S)
- $Cost_{GHJ}(R,S)=3(B(R)+B(S))$

Grace Hash Join Memory Requirements

- The hash table for R_i must fit in memory
  - Minimize partition size
  - Maximize #partitions
- M buffer pages → m=M-1 partitions
  - Partitioning phase
- Size of partition: B(R)/m
- Size of hash table: B(R)F/m
  - "fudge factor"
- Probing phase: hash table + 1 page to read + 1 for output
  - → $M > fB(R)/(M-1) + 2$ → $M > \sqrt{fB(R)}$

Comparing Ideas



Hybrid Hash Join

- Goal is to exploit large memory
- Assume M > fB(R)/k, for k integer
  - We can build a HT for k R partitions of size M/k
  - To be able to partition R into k partitions we need k+1 pages
  - This leaves us with M-(k+1) free pages during partitioning phase. How to use these?
- Assume M-(k+1) > fB(R)/k
  - This means that we have enough memory for a hash table on an R partition during the partitioning phase
  - Build this hash table during partitioning and keep it in memory
  - While partitioning S directly probe the HT

Hybrid Hash Join Cost Savings

- Assume R is 500 pages, S is 1000 pages
- Assume M = 300 pages
- Partition R into R1,R2 of 250 pages each
  - Keep R1 in memory while partitioning
  - 500 + 250 IOs
- Scan S and write out one partition
  - S1 probes HT on R1
  - 1000 + 500 IOs
- Probing phase
  - Scan R2,S2
  - 250 + 500 IOs
- <u>Total cost: 3000 IOs</u>
- <u>Grace HJ cost: 4500 IOs</u>

Hash – Sort Duality

- For small datasets, use quicksort or in-memory hashing
- For large datasets, divide-and-conquer partitioning is used for both
- Sorting: Partitions determined by amount of main memory (physical partitioning) and combined by merging (logical combining)
- Hashing: Partitions determined by hash function (logical partitioning) and combined physically
- I/O pattern
  - Writing initial runs after sorting: sequential I/O
  - Merging: random reads for many files
  - Partitioning: random I/O
  - Reading a partition: sequential I/O

Hash-Sort Differences

- Both cost 3(B(R)+B(S)) given large enough memory
  - M > sqrt(B(R)) for hash, M > sqrt(B(S)) for sort
- Hash join sensitive to data skew (hash collisions)
- Hash join exploits cardinality difference
  - Main memory requirement depends on smaller relation
- Sort produces sorted result
  - May be needed/useful later (interesting orders)
- Non-equijoins:
  - In general resort to nested loops

Grouping and Aggregation

- One-pass Hash Algorithm
  - Scan input, build hash-table on group (hash on grouping key)
  - One entry per group in main memory + computed aggregate
  - With every scanned element, update existing entry or create a new one

- Hash table needs to fit into main memory

  → Cost = B(R)

Grouping and Aggregation

- If Hash Table does not fit into memory
  - Similar approach as Grace Hash Join and Hybrid Hash Join are possible
  - Grace: First partition your input such that the individual Hash Tables would, then aggregate as usual → Cost: 3*B(R)
  - Hybrid: Start with an in-memory hash table and spill partitions to disk if table grows too large → Savings as with Hybrid Hash Join

Grouping and Aggregation

- Nested loops
  - Iterate for each input record over temporary output file and either aggregate or create new group
- Sorting
  - On grouping attributes (or PK/RID for dupl. Elim)
  - Early aggregation (when writing back run files during sorting) provides significant speed-up

Grouping by Sorting

- Phase 1
  - Sort elements. When writing back, apply aggregation (write only one element per group). Works only with distributive aggregation functions.
- Phase 2
  - Merge as in mergesort
  - Process smallest key (create new group)
  - Aggregate all tuples of this key
    - May need to load more blocks
  - Output one tuple
  - Proceed to next smallest key
- Cost: 3B(R) for external sort, B(R) if fits in memory

PIPELINING, ITERATORS

Pipelined versus Blocking Operators

Two methods for composing operators

1) Execute one operator at a time over its input.
   Store the result (memory buffers or on disk)
   Next Operator afterwards

2) Interleave the operators → Pipelining
   Operator consumes data as previous operator produces them

- Pipelining has advantages
  - No buffering of huge intermediate results
  - May return first results before query completion (and early out from operations)
- Pipelining disadvantages?

Pipelined versus Blocking Operators

- Not all operators can be pipelined
- Some need to see **all** the input before they can produce the output
  - For example sorting
- Some can produce output, but need to materialize a large portion of the input nonetheless

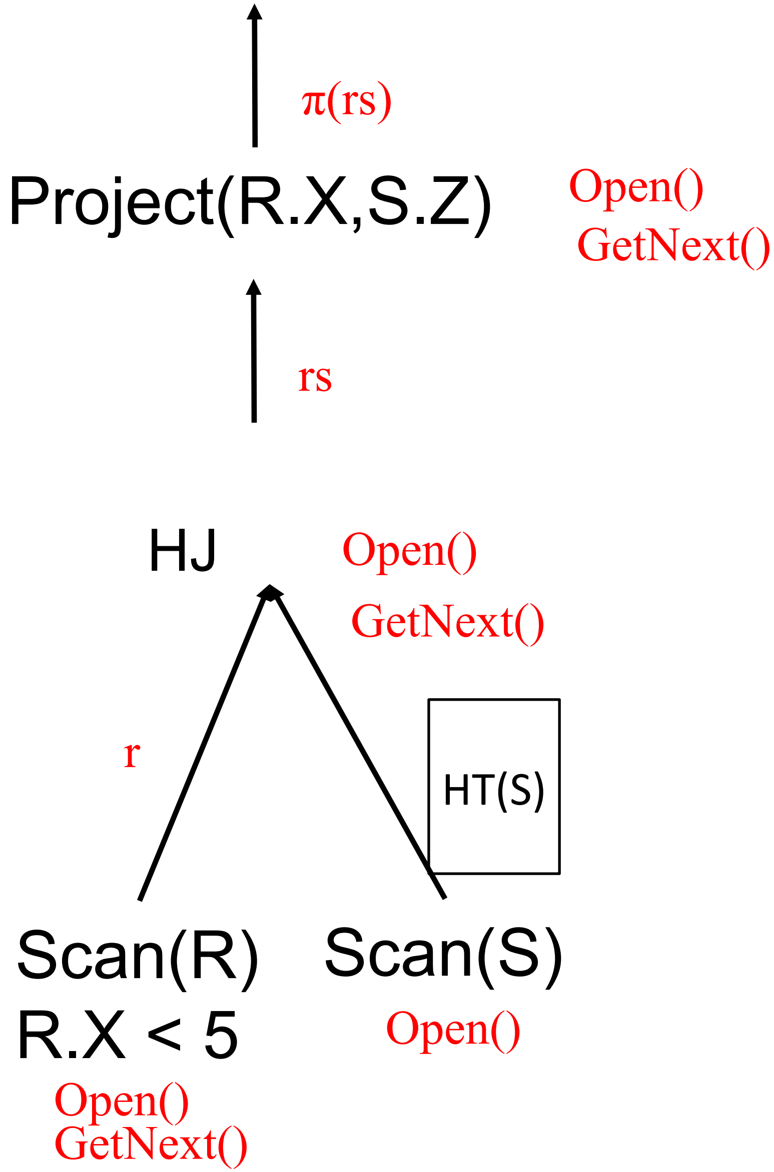
Iterator Model

- How to assemble operators into a pipelined data flow

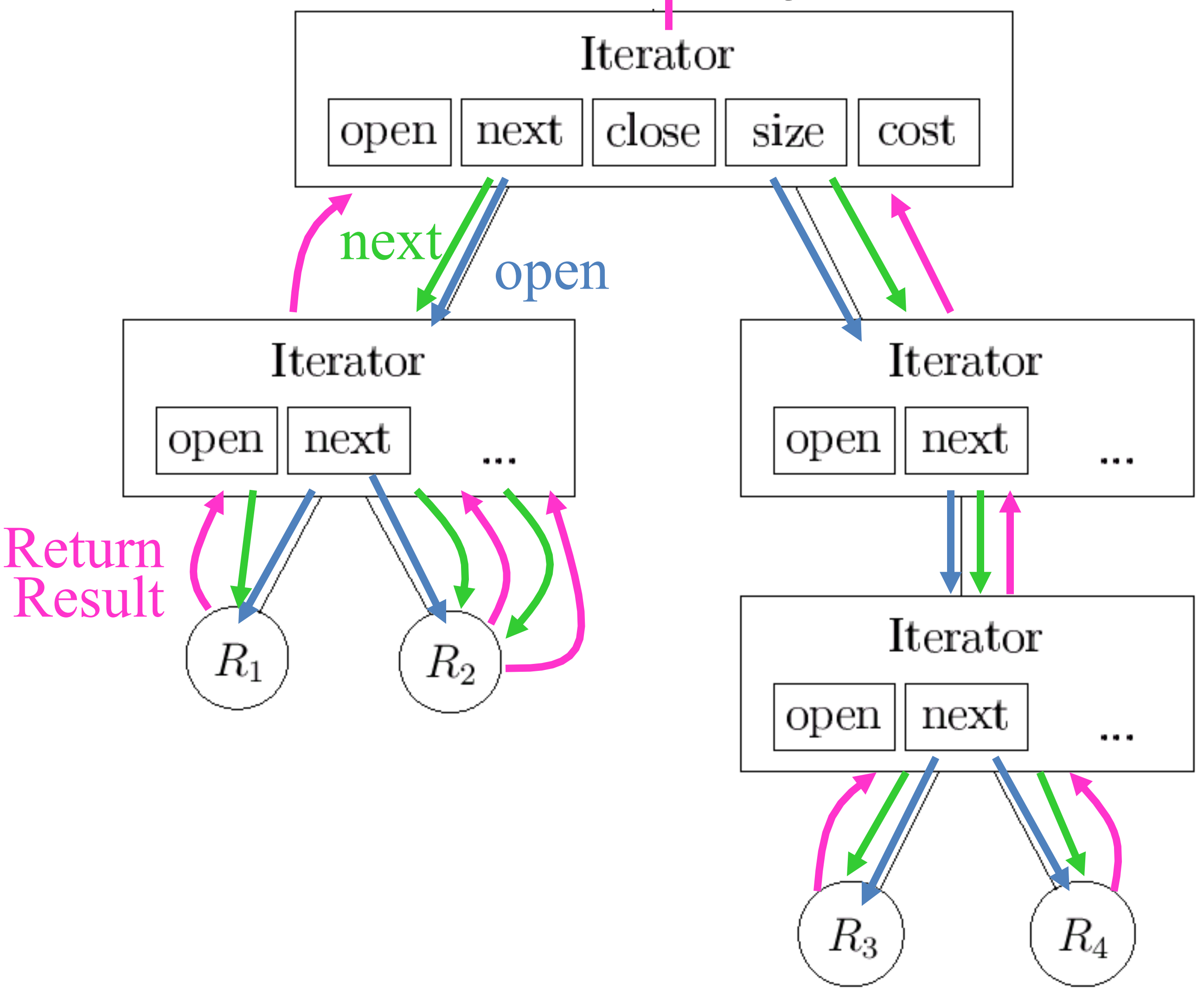
- Iterator model
  - `Open()`: init data structures, call `Open()` for input operators
  - `GetNext()`: call `GetNext()` for input operators, assemble next tuple
    - Or return tuple if here already
    - Or return `EndOfStream`
  - `Close()`: call `Close()` for input operators

- Pipelining between operators
- Internally, an operator may materialize

Iterator Example



Pull-based Query Execution



Iterator – Example

- **Open()**
  - **b := the first block of R;**
  - **t := the first tuple of block b;**
- **GetNext()**
  - **IF (t is past the last tuple on block b)**
    - **Increment b to the next block;**
    - **IF (there is no next block)**
      - » **RETURN NotFound;**
    - **ELSE**
      - » **t := first tuple on block b;**
  - **oldt := t;**
  - **Increment t to the next tuple of b;**
  - **RETURN oldt;**
- **Close()**
  - **Do Nothing**

Question: What is being implemented here?

Answer: Table scan

Iterator – Example

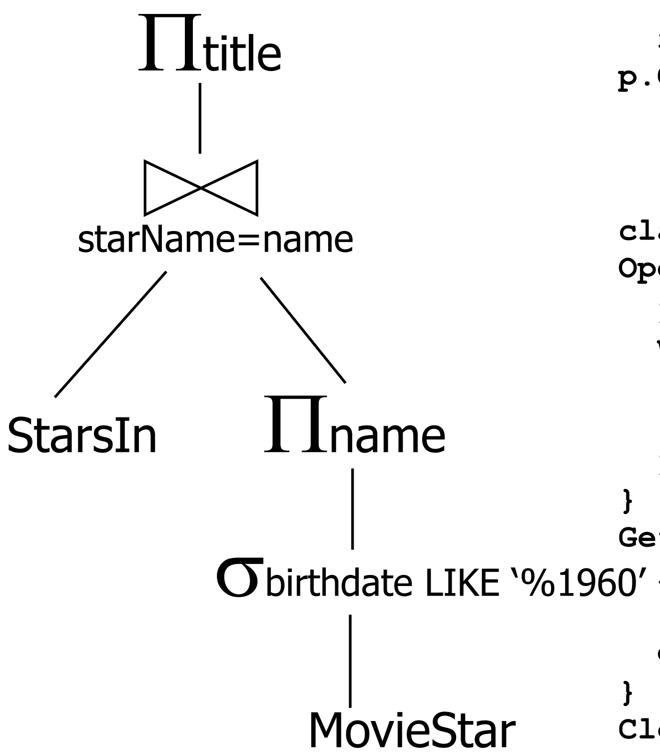
```
Open(R,S) {
    R.open();
    CurRel := R;
}
GetNext(R,S) {
    IF (CurRel = R) {
         t := R.GetNext();
         IF(t <> NotFound) /*R ist not empty*/
             RETURN t;
         ELSE /*R is empty */ {
             S.Open();
             CurRel := S;
    }}
    RETURN S.GetNext();
}
Close(R,S) {
    R.Close();
    S.Close()
}
```

Question: What is being implemented here?

Answer: UNION ALL

Iterator – Example

```
p = projection.Open();
while (t <> NotFound)
      t = p.GetNext()
  return t;
p.Close();
```



Blocking

```
class projection {
Open() {
  j = join.Open();
  while (t <> NotFound)
    t:=j.GetNext()
    tmp[i++]=t.title;
  j.Close();
}
GetNext( ) {
  if (cnt < tmp.size())
    return tmp[cnt++];
  else return NotFound;
}
Close() {
  discard(tmp);
}
}
```

```
class join {
Open() {
  l = table.open();
  while (tl <> NotFound)
    t1 = l.GetNext();
    r = projection.Open();
    while (tr <> NotFound)
      tr = r.GetNext();
      if tl.starname=tr.name
        tmp[i++]=tl⋈tr;
    end while;
    l.Close();
  end while;
  r.Close();
}
GetNext( ) {
  if (cnt < tmp.size())
    return tmp[cnt++];
  else return NotFound;
}
Close() {
  discard(tmp);
  Close();
}
}
```

Quiz

Push or Pull Plans

- Iterators "pull" the tuples from the predecessor. How would a symmetric counterpart look where operators push tuples into the successor

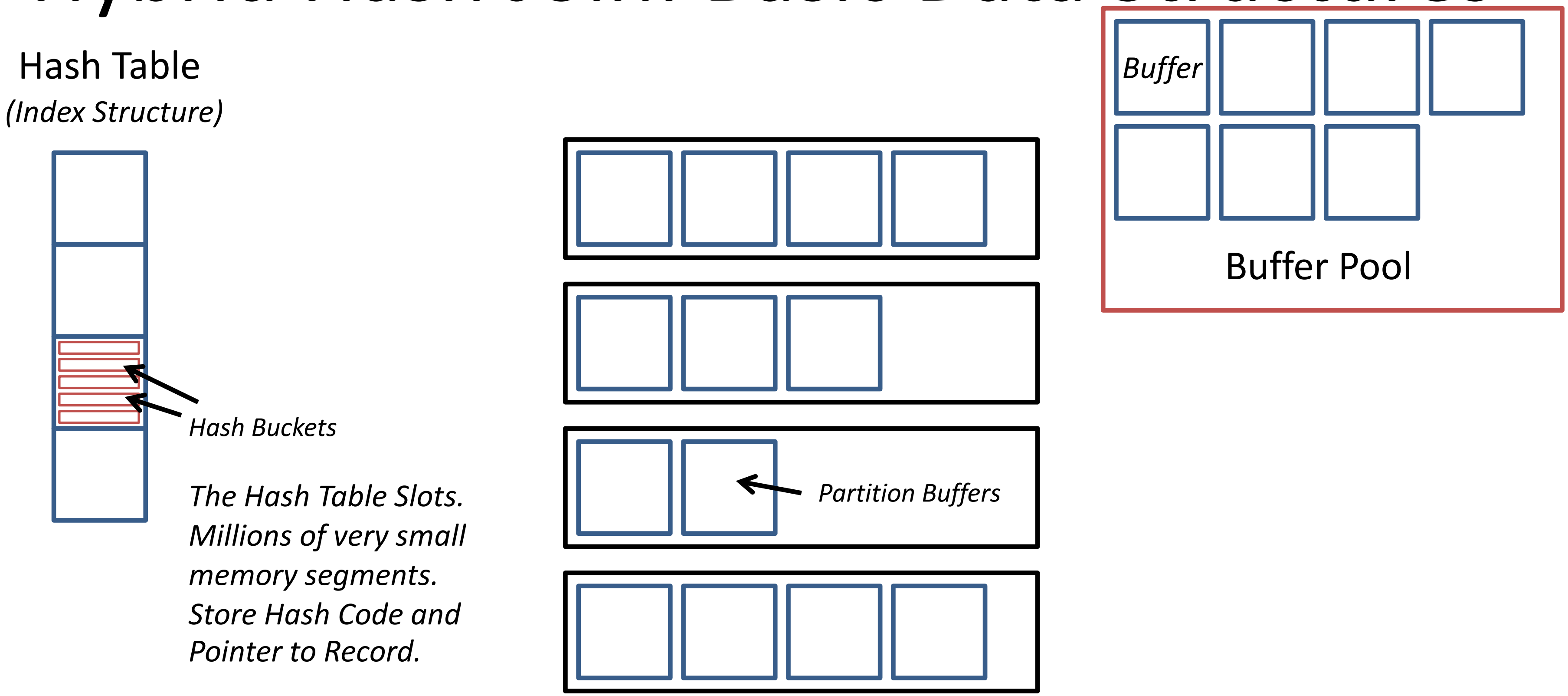
Granularity

- Is pulling (pushing) one tuple at a time the best thing to do?

Appendix A

HOW TO DO A HYBRID HASH JOIN

Hybrid Hash Join: Basic Data Structures

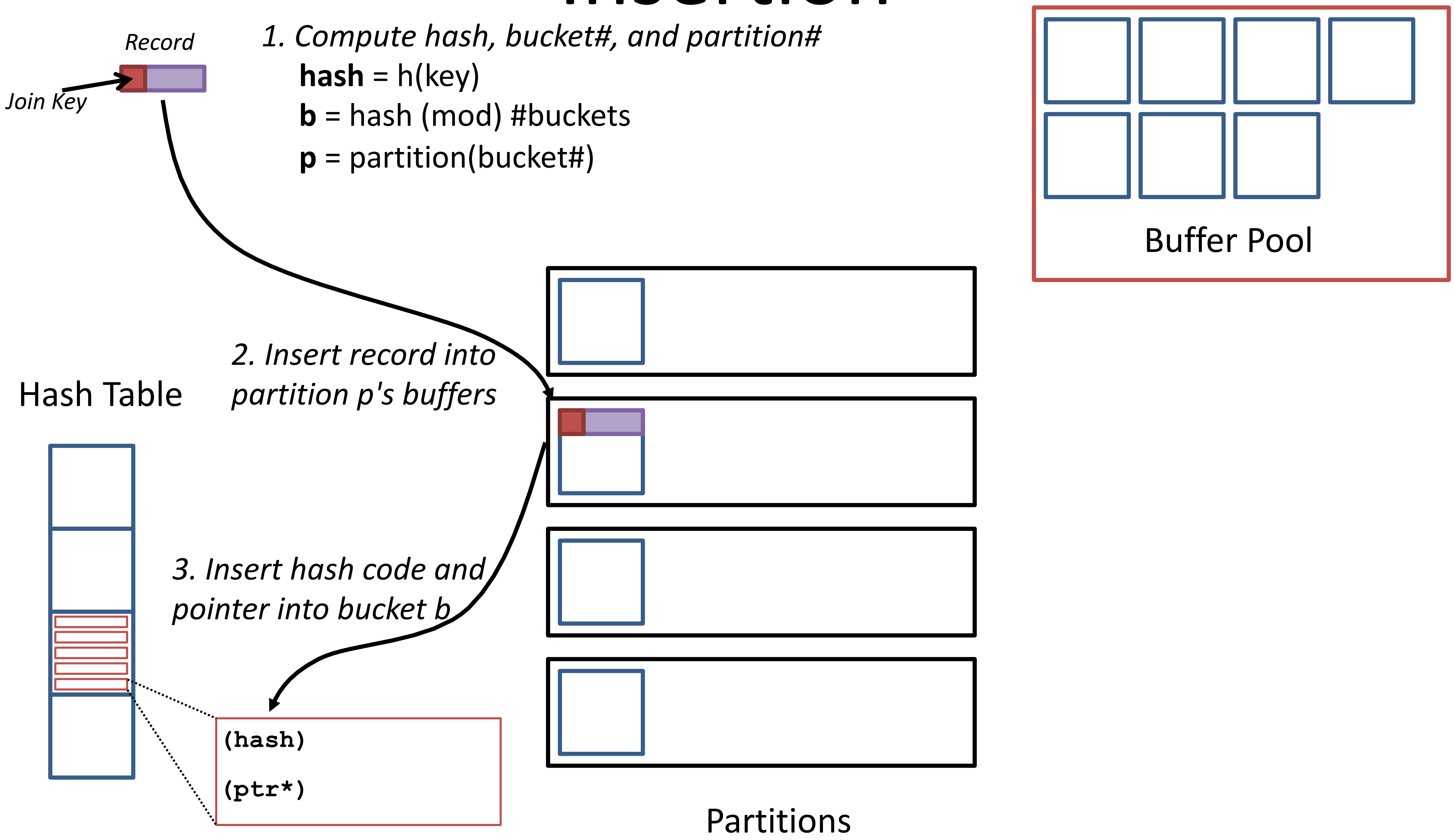


Partitions

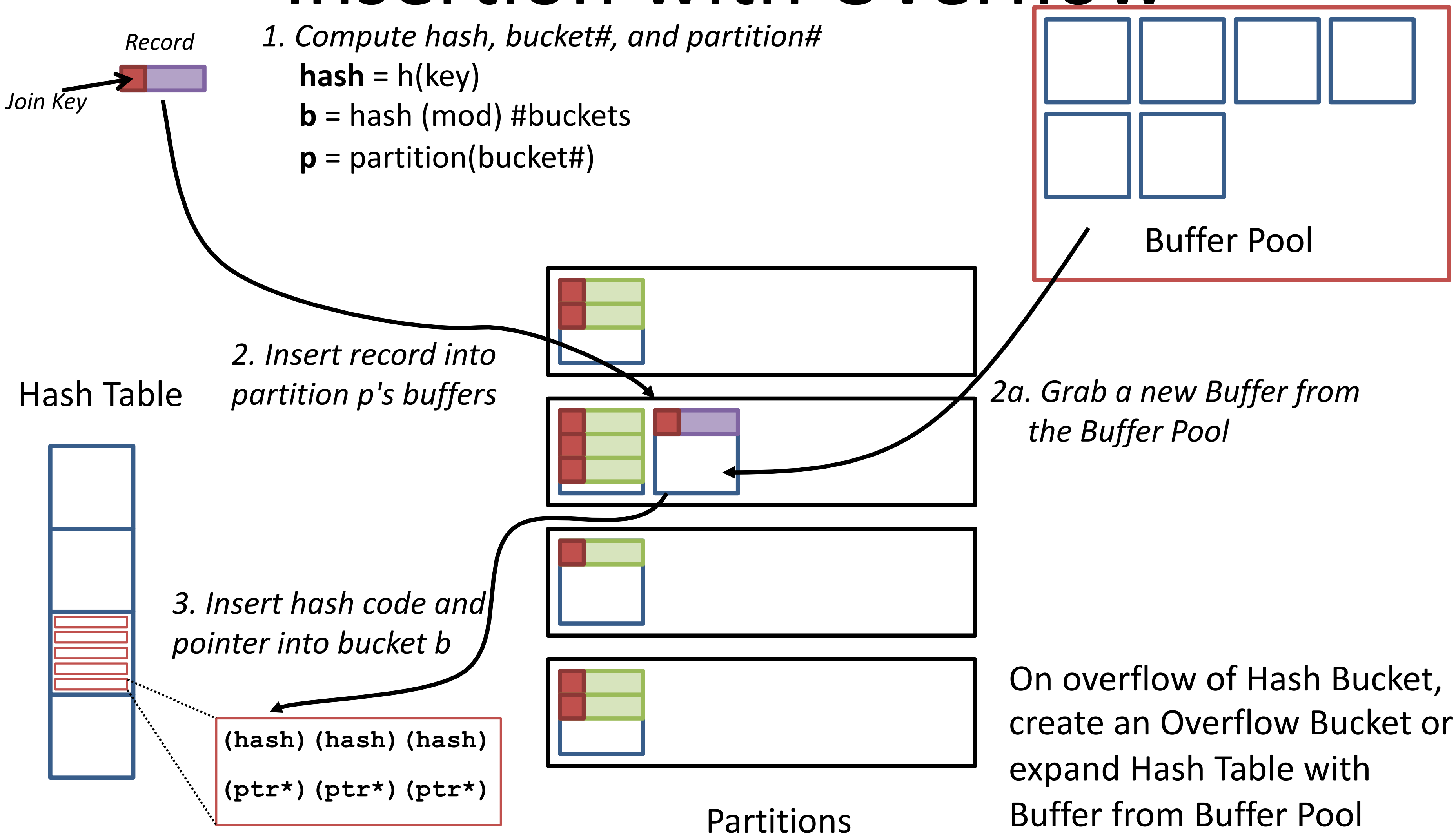
Few, for example about 100 - 200
Granularity of spilling data to disk.
Store the records for multiple buckets.

After Graefe et al. "Hash joins and hash teams in Microsoft SQL Server", 1998

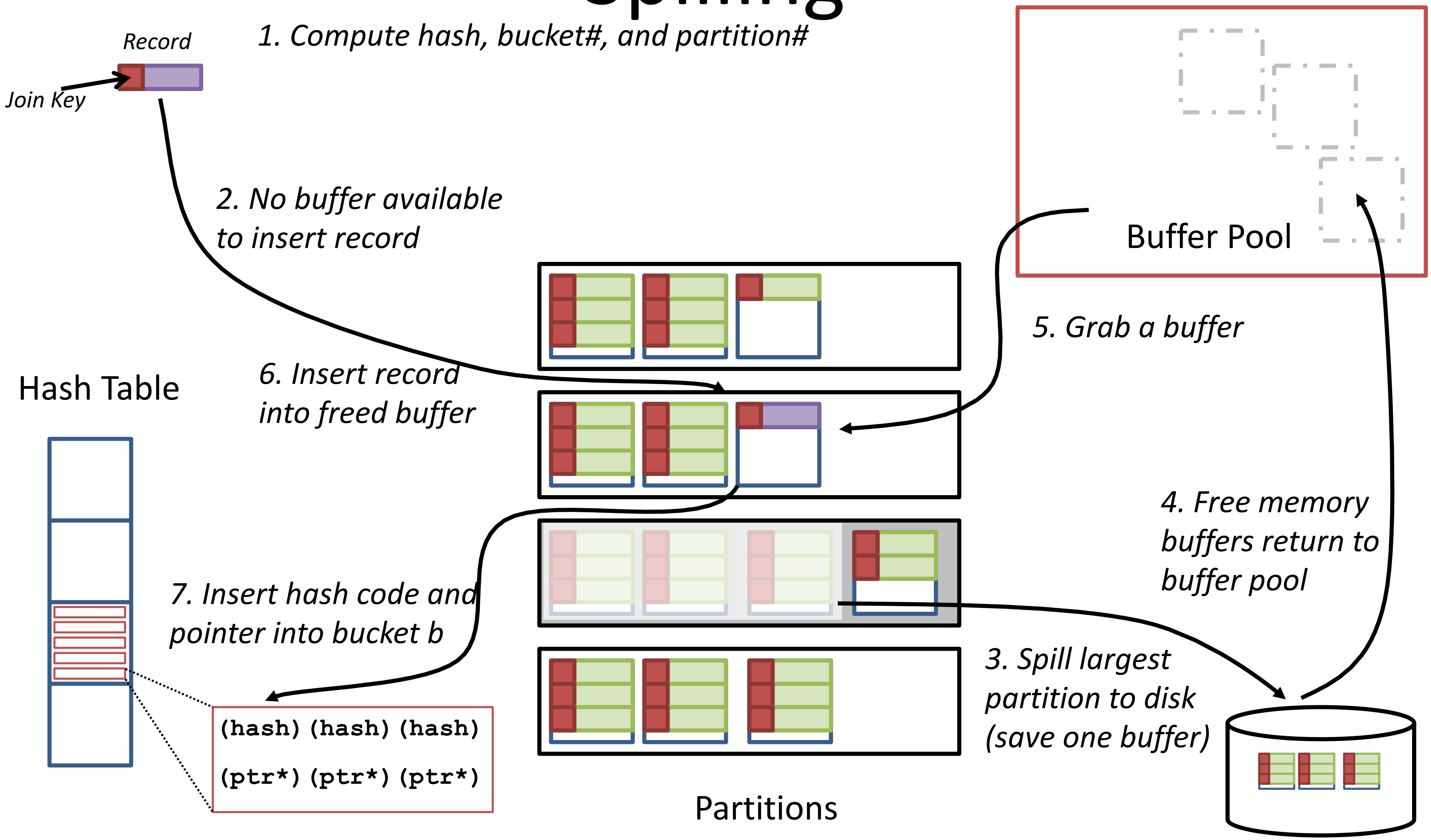
Insertion


Record
Join Key
1. Compute hash, bucket#, and partition#
hash = h(key)
b = hash (mod) #buckets
p = partition(bucket#)
Buffer Pool
2. Insert record into partition p's buffers
Hash Table
3. Insert hash code and pointer into bucket b
(hash)
(ptr*)
Partitions

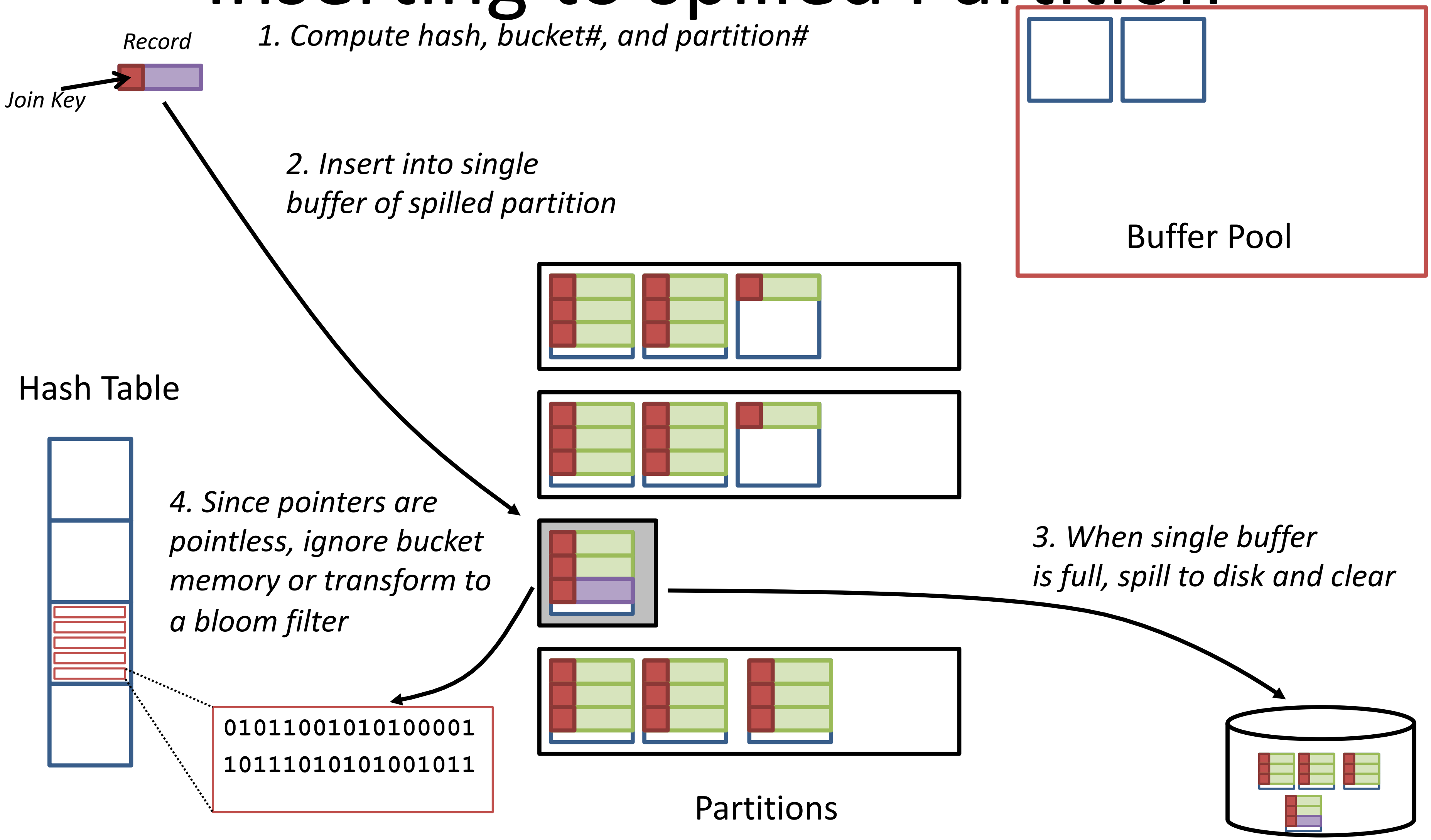
Insertion with Overflow



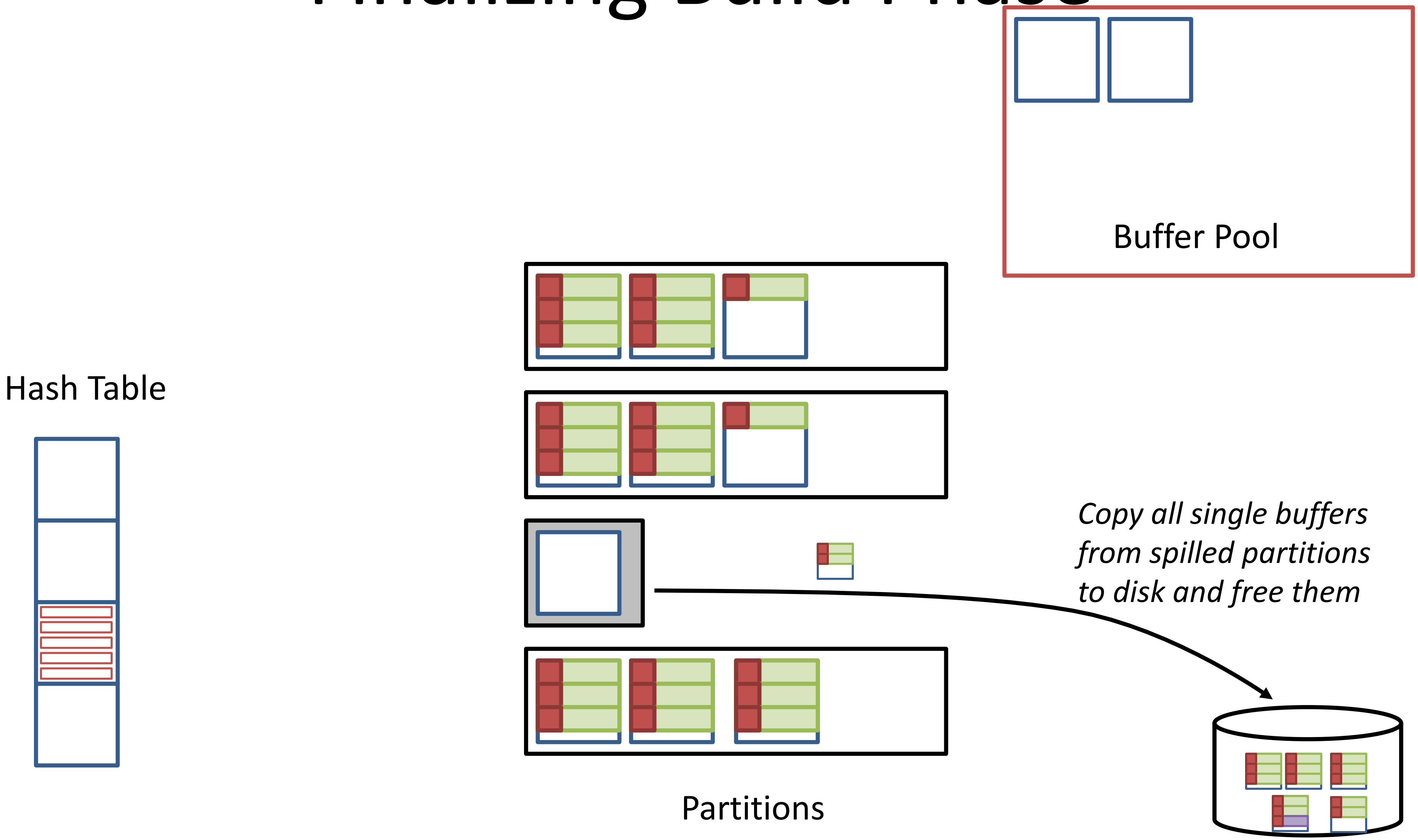
Spilling


Record
Join Key
1. Compute hash, bucket#, and partition#
2. No buffer available
to insert record
Buffer Pool
5. Grab a buffer
6. Insert record
into freed buffer
Hash Table
4. Free memory
buffers return to
buffer pool
7. Insert hash code and
pointer into bucket b
3. Spill largest
partition to disk
(save one buffer)
(hash) (hash) (hash)
(ptr*) (ptr*) (ptr*)
Partitions

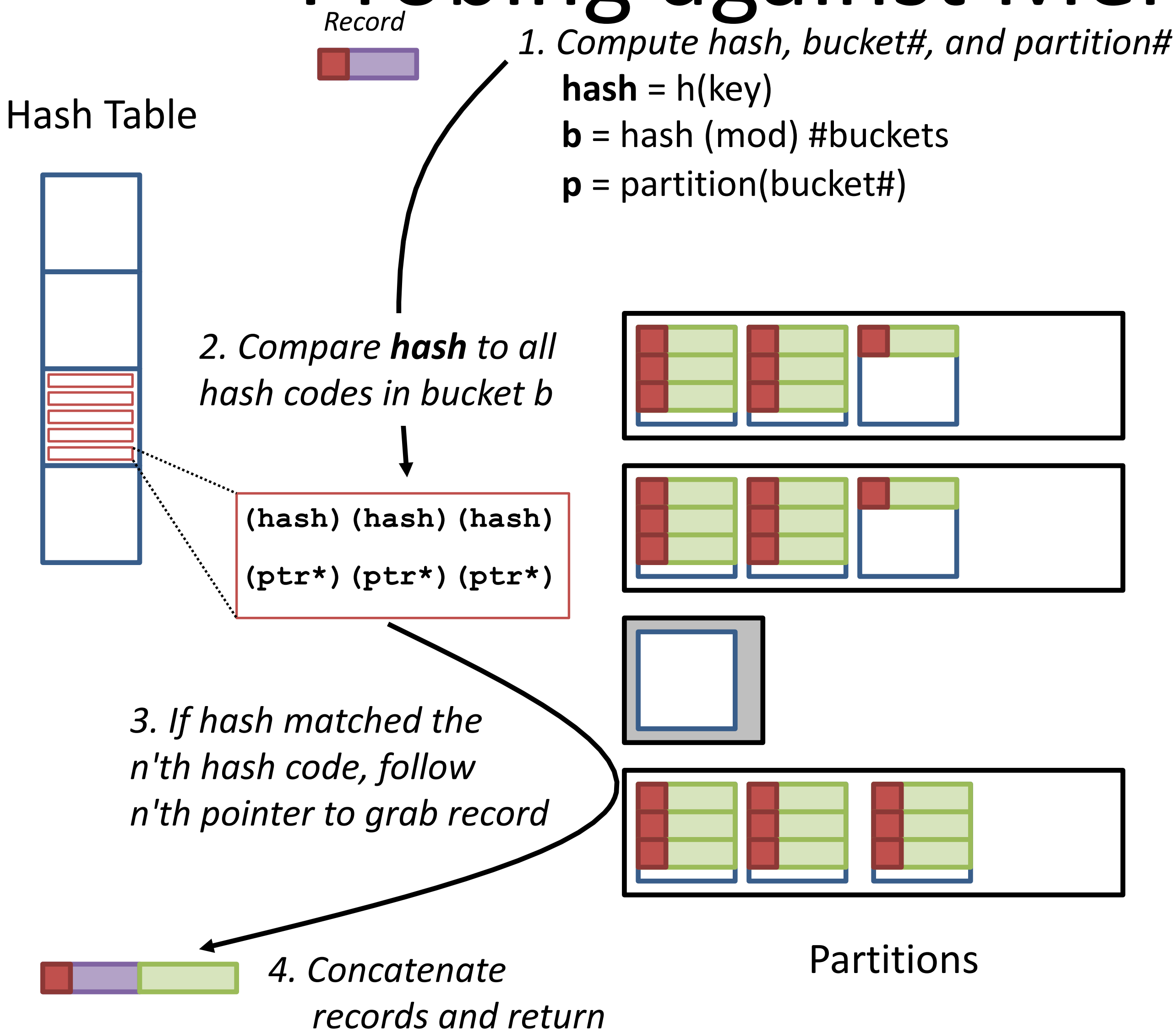
Inserting to spilled Partition



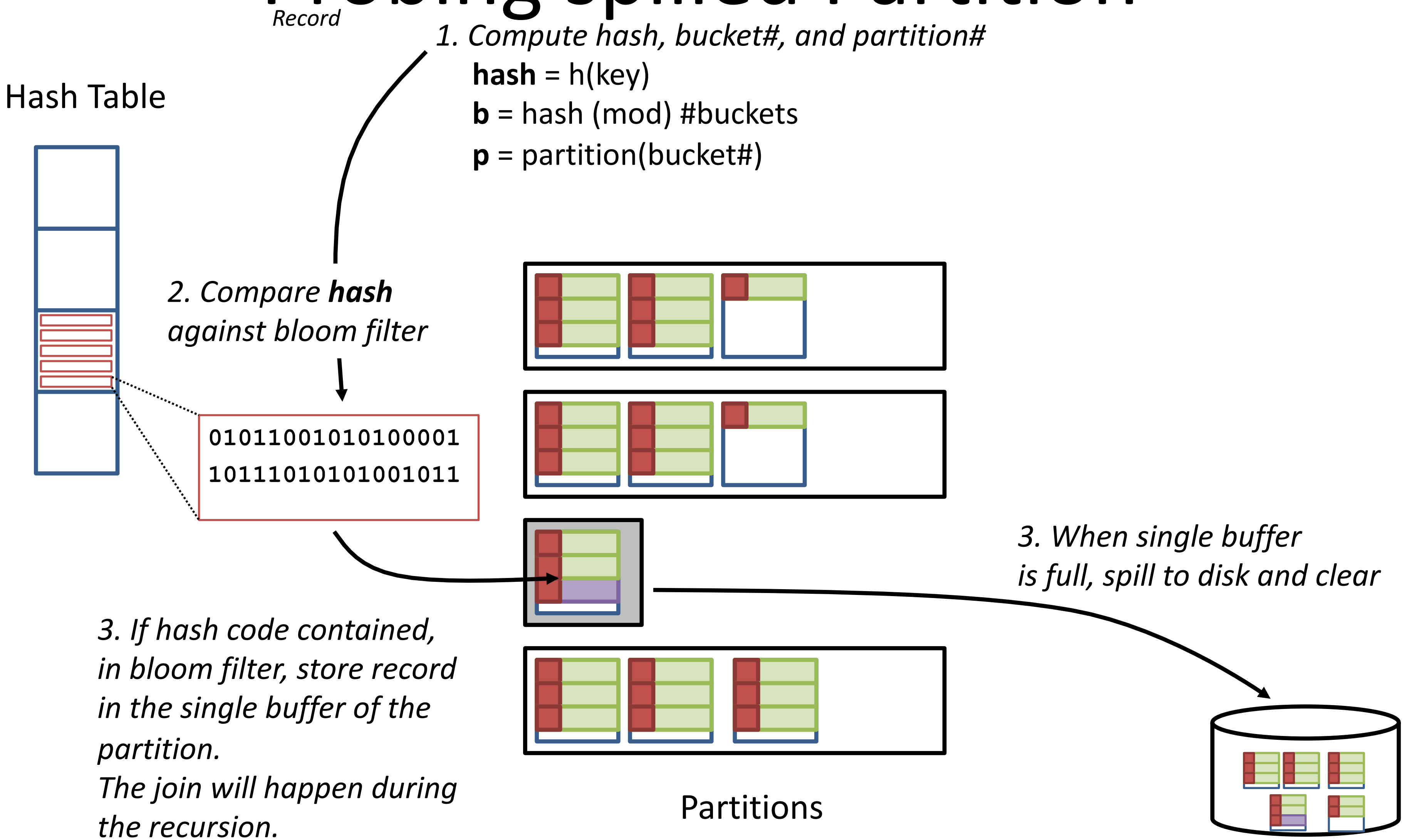
Finalizing Build Phase



Probing against Memory

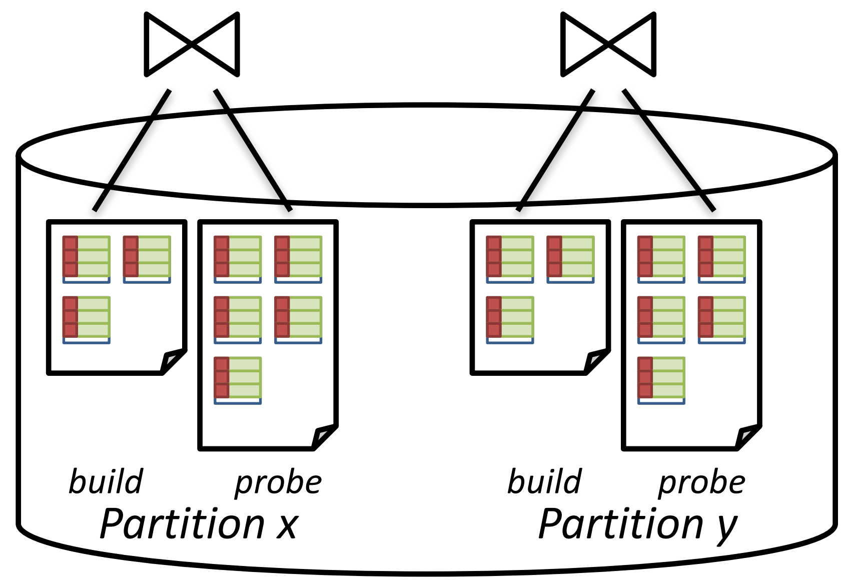


Probing spilled Partition



Hybrid Hash Join: Recursing

- After both inputs are consumed, some partitions may be spilled
- Two files (or collections of buffers) for each partition, one for build and probe side
- Recursively execute Hybrid Hash Join on the partitions
  - Possibly creates new spilled partitions (rarely in today's memory sizes)
- Recursive join must not use the same hash function
  - Elements in one partition came from a subset of the buckets.
  - Would cause majority of buckets to remain empty.



Multi Level Hash Function

- Ideal: Values that hash collide on one level have uniformly distributed hash values at the higher levels
- Initial join has level 0, a recursion level *L+1*, if *L* was the level of the join that produced that spilled partition.
- Example of Hash Function in Stratosphere Hash Join

```
public static final int hash(int key, int level)
{
        final int rotation = level * 7;

        key = (key << rotation) | (key >>> -rotation);

        key = (key + 0x7ed55d16) + (key << 12);
        key = (key ^ 0xc761c23c) ^ (key >>> 19);
        key = (key + 0x165667b1) + (key << 5);
        key = (key + 0xd3a2646c) ^ (key << 9);
        key = (key + 0xfd7046c5) + (key << 3);
        key = (key ^ 0xb55a4f09) ^ (key >>> 16);
        return key >= 0 ? key : -(key + 1);
}
```

Level-dependent rotation changes bit order

Jenkins Hash (has full avalanche property)

Improvements

- Partition Tuning
  - When finishing initial build phase, some memory buffers may be free from the last spill. Possibly exchange some in-memory partitions with some spilled partitions, to increase memory usage and hence the number of in-memory probes.

- Role Reversal
  - For spilled partition, switch probe and build side when probe side is actually smaller.

- Bail-out to Block-Nested-Loop Join
  - When recursing does not reduce partition sizes enough (high number of duplicate keys on both sides), process partition in a block-nested-loops fashion.