SQLServerFast.com Execution Plan Video Training

Block 3: Combining data

Level: Advanced

Chapter 3: Hash Match (advanced)

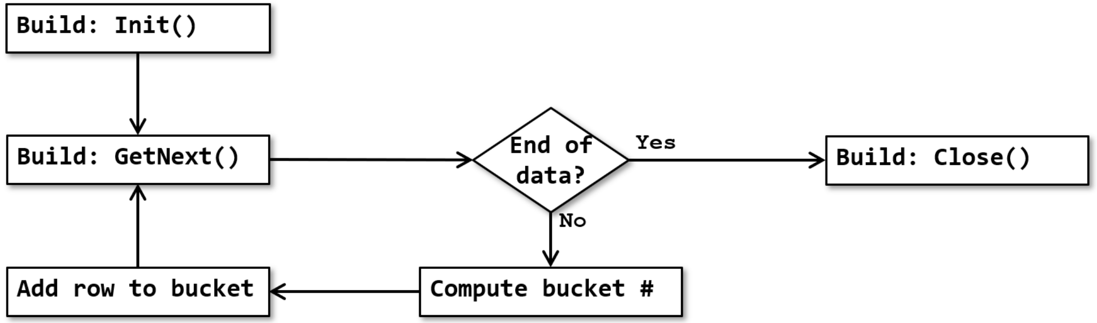
Hash Match

- Hash Match
  - Build phase
  - Probe phase

Hash Match

Hash Match

Build phase



Build: Init()

Build: GetNext()

End of data?

Yes

No

Build: Close()

Compute bucket #

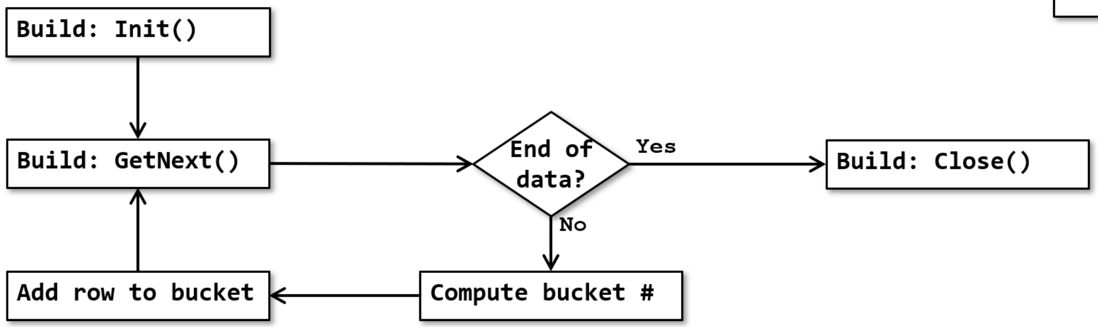
Add row to bucket

Hash Match (Inner Join)

Hash Match

Hash Match

Build phase



Build: Init()

Build: GetNext()

End of data?

Yes

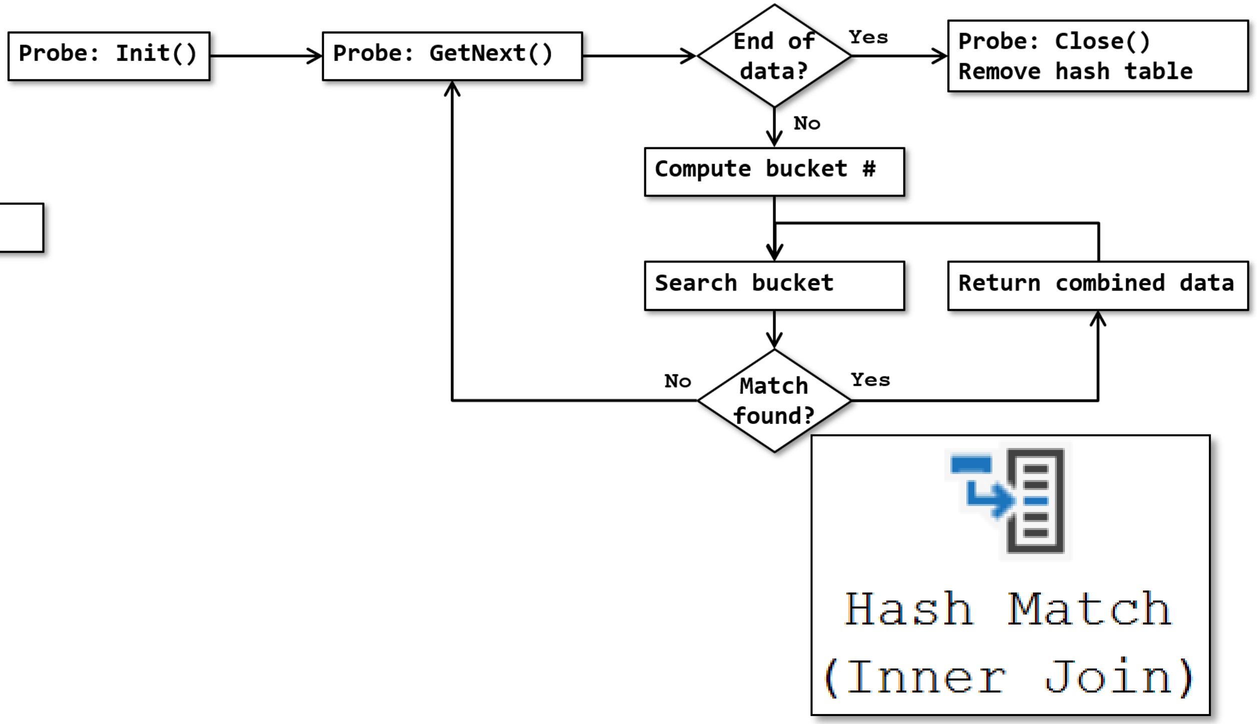
No

Build: Close()

Compute bucket #

Add row to bucket

Probe phase



Probe: Init()

Probe: GetNext()

End of data?

Yes

No

Probe: Close()
Remove hash table

Compute bucket #

Search bucket

Return combined data

Match found?

No

Yes

Hash Match
(Inner Join)

Hash Match (inner to left outer)

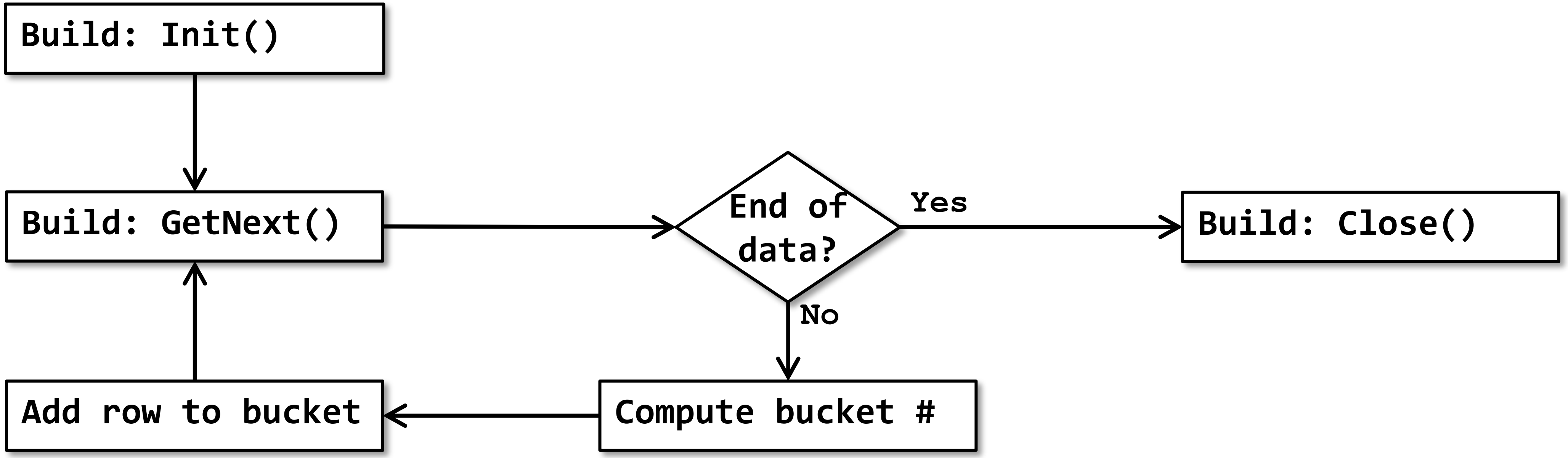
- From inner join to left outer join
  - Add unmatched left rows
    - These are from the build input
    - Can be found in the hash table
      - But which of these rows to return?
      - Use a Boolean to track when there are matches

Hash Match (inner to left outer)

- From inner join to left outer join
  - Add unmatched left rows
    - These are from the build input
    - Can be found in the hash table
      - But which of these rows to return?
      - Use a Boolean ***for each row in the hash table*** to track when there are matches

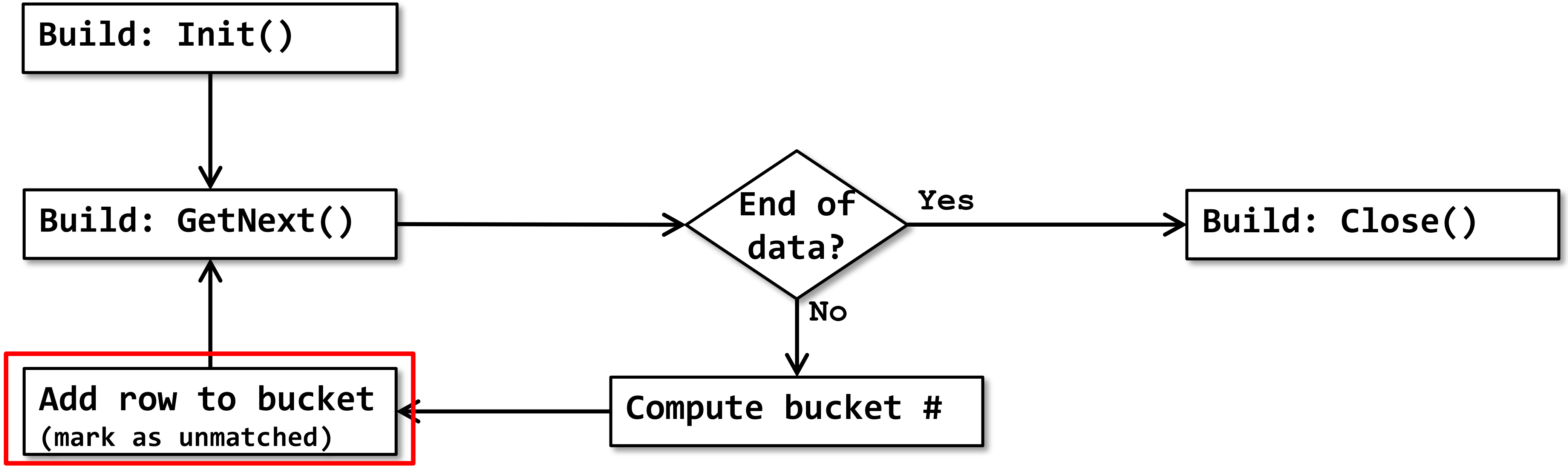
Hash Match (inner to left outer)

Build phase



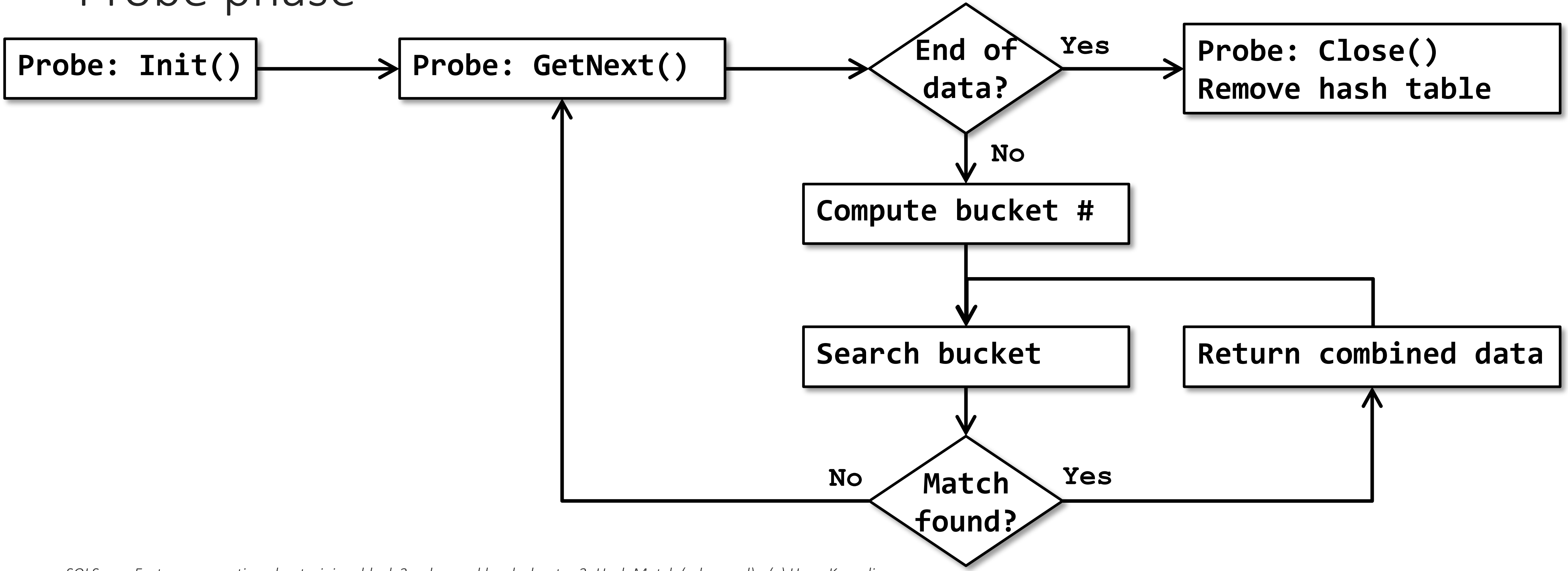
Hash Match (left outer join, no spill)

Build phase



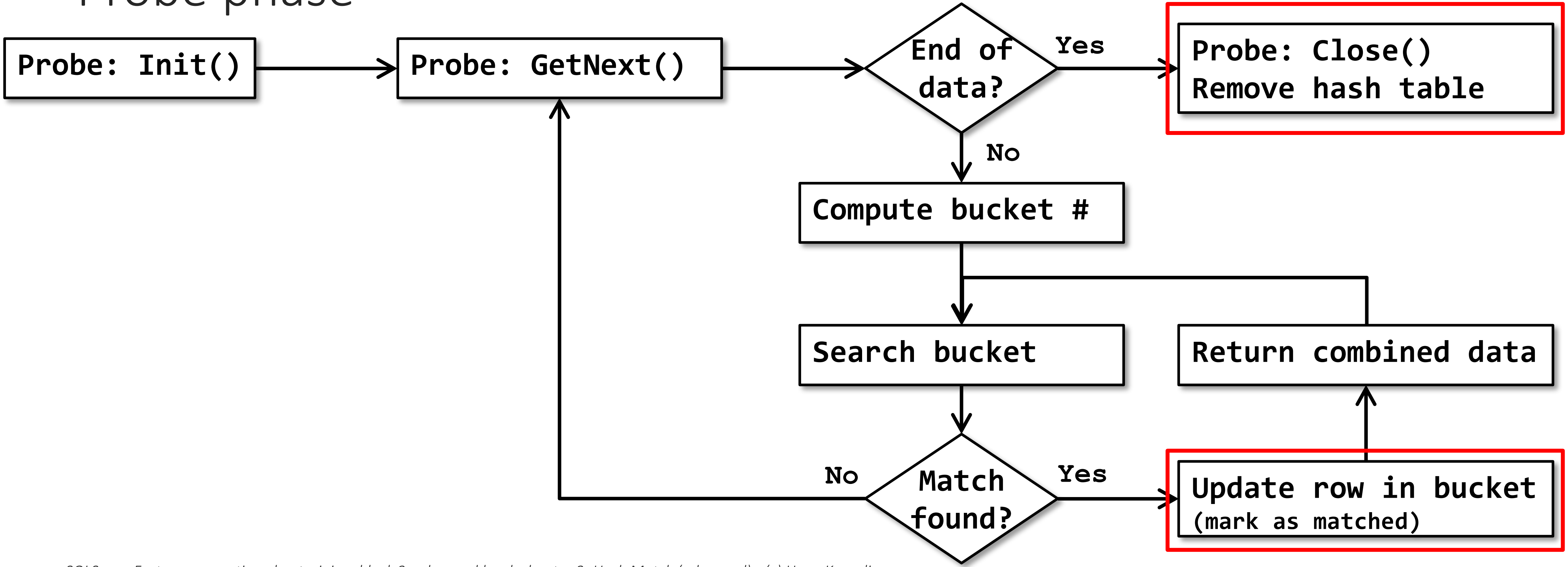
Hash Match (inner to left outer)

Probe phase



Hash Match (inner to left outer)

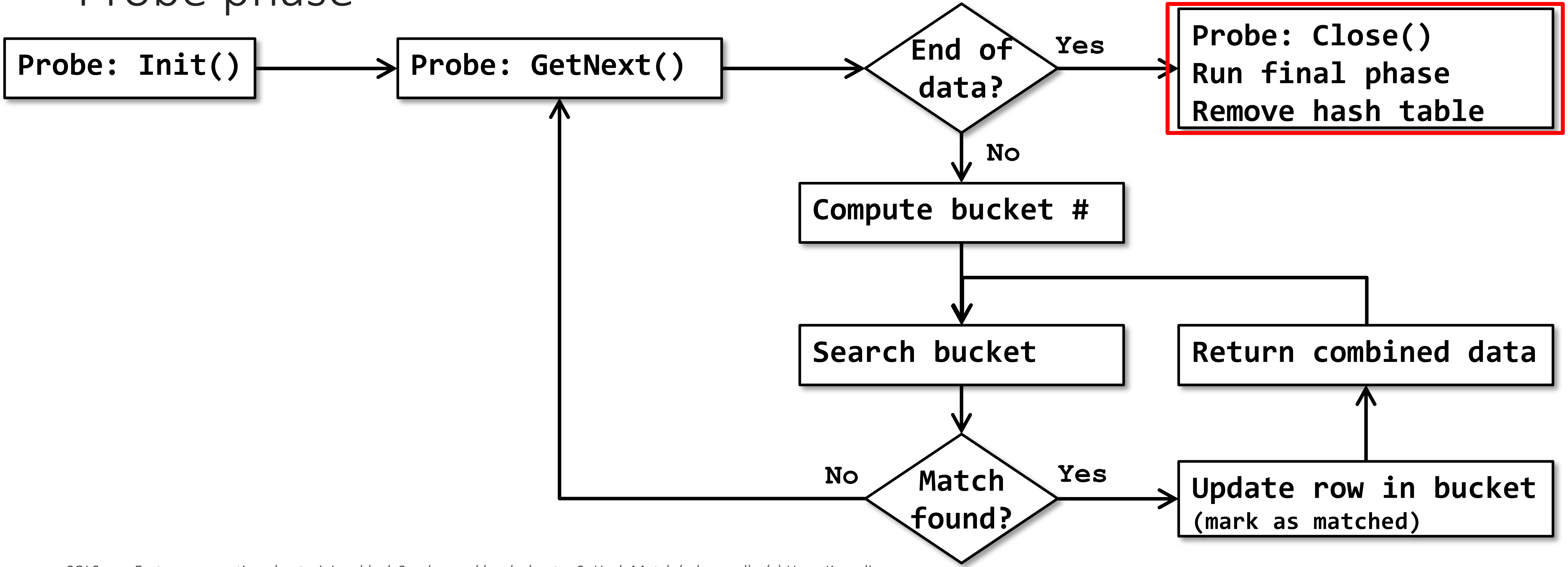
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

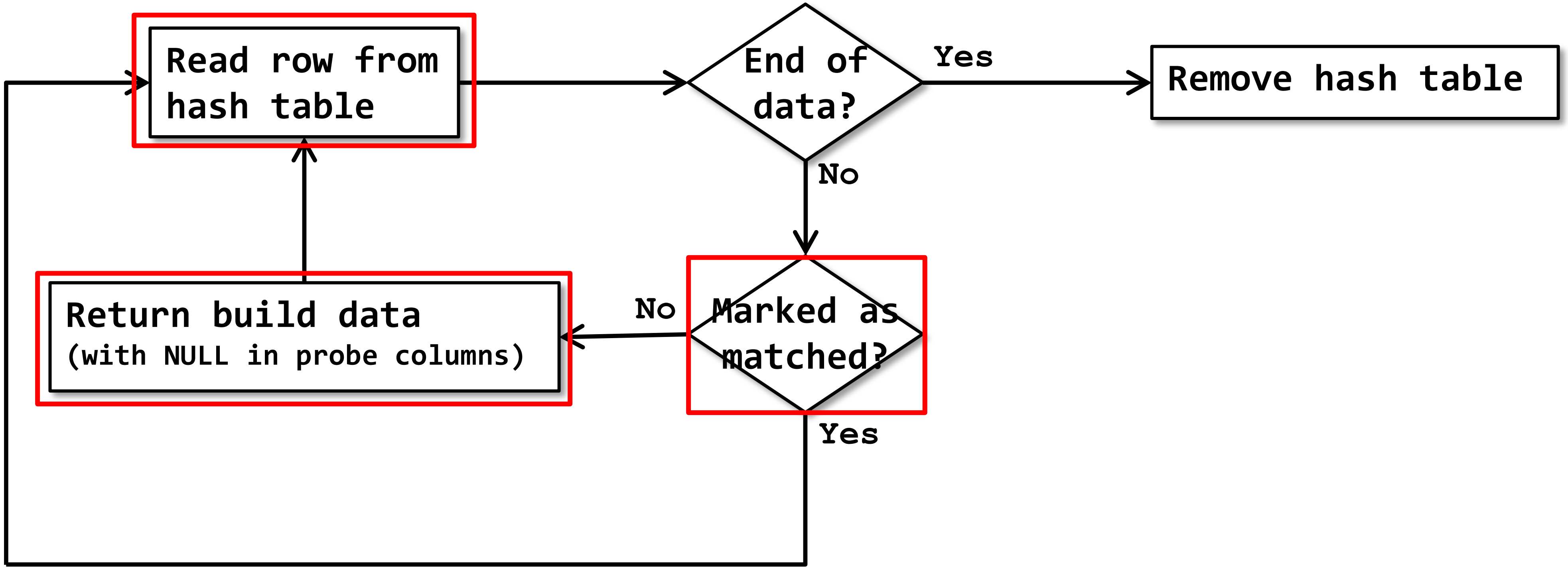
Hash Match (left outer join, no spills)

Probe phase



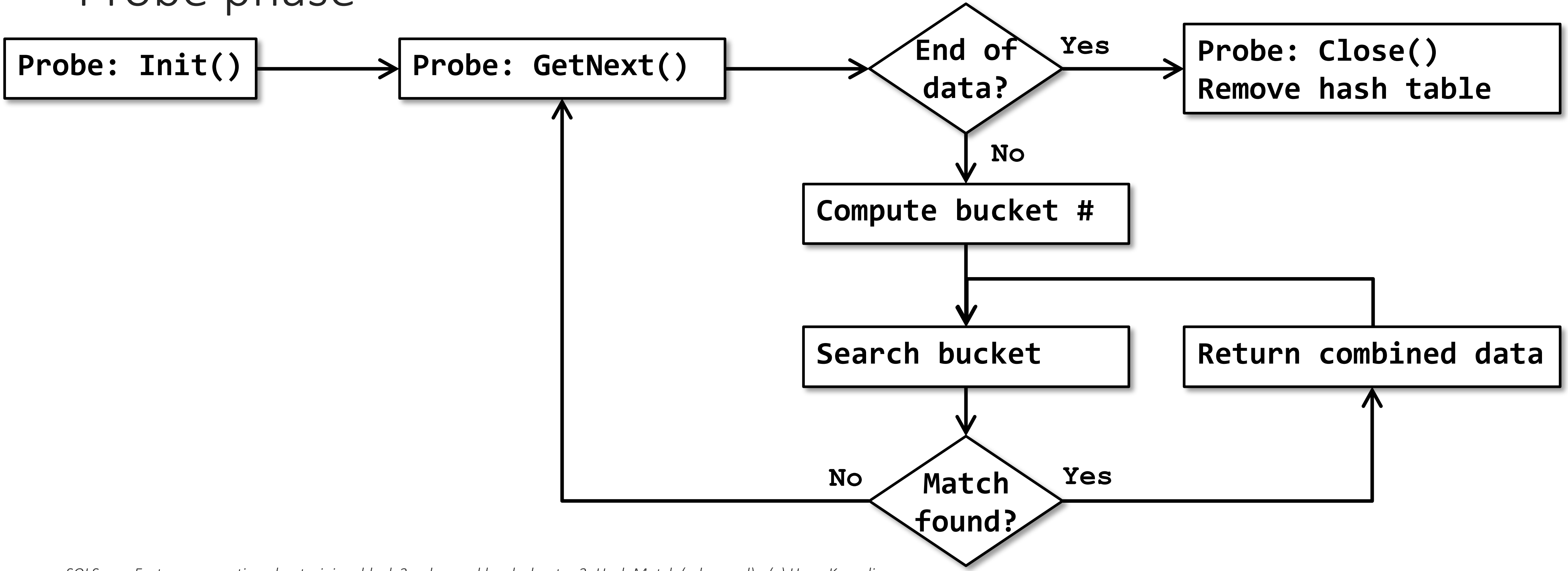
Hash Match (left outer join, no spills)

Final phase (officially part of the probe phase)



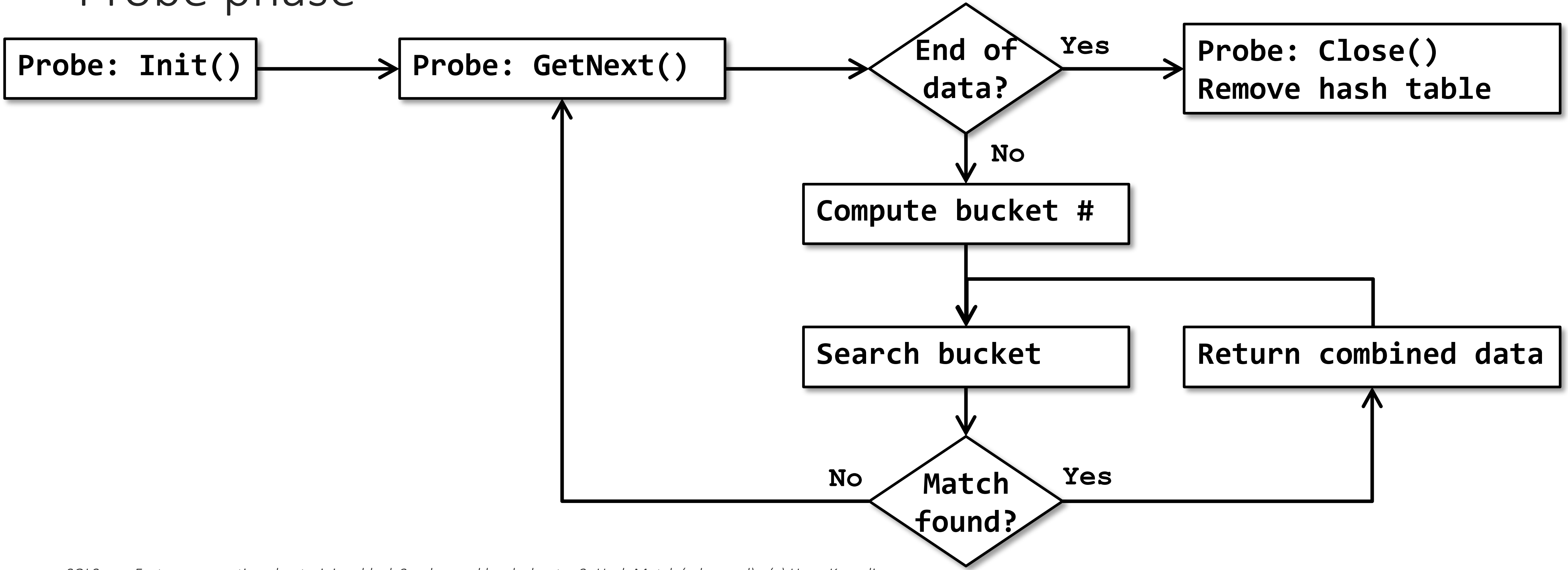
Hash Match (inner join, no spills)

Probe phase



Hash Match (inner to right outer)

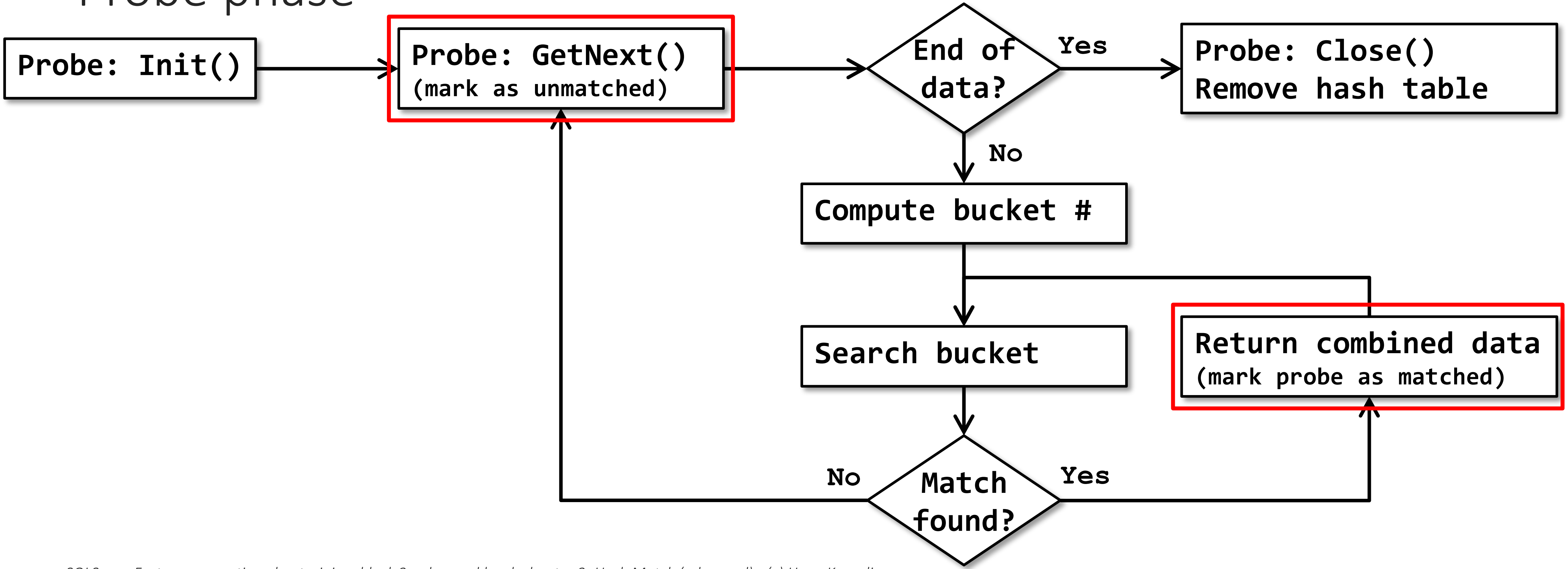
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

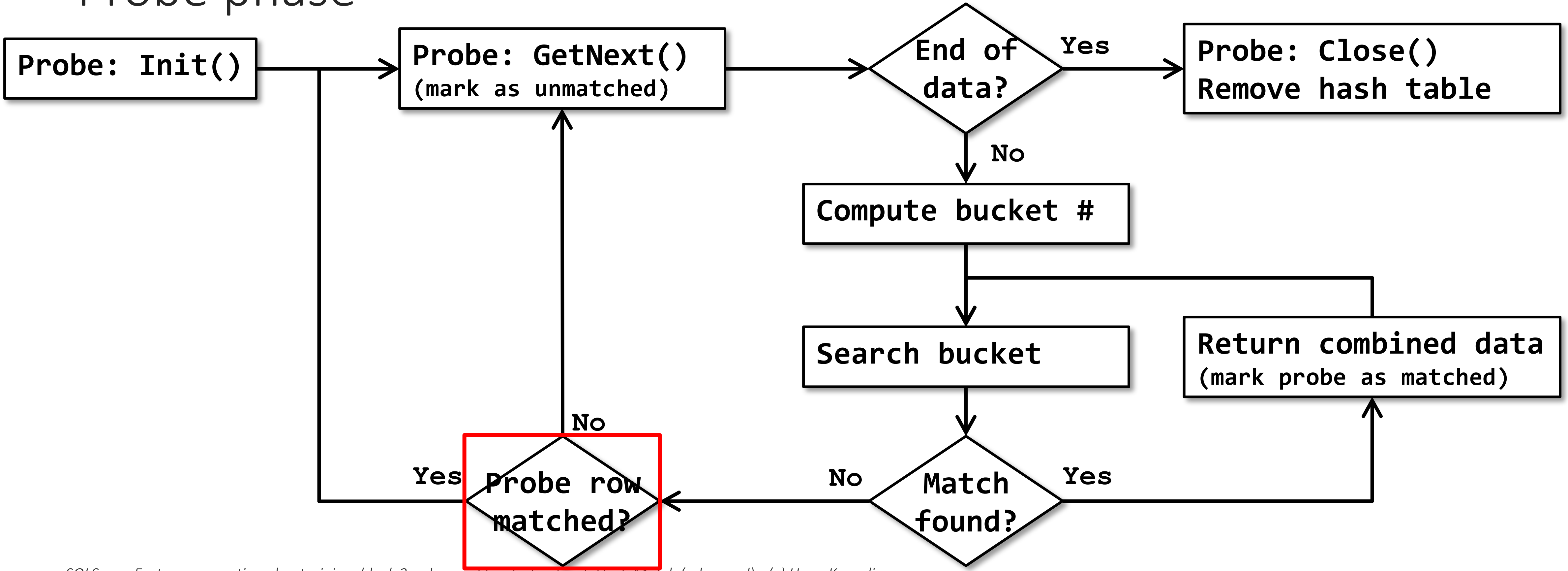
Hash Match (inner to right outer)

Probe phase



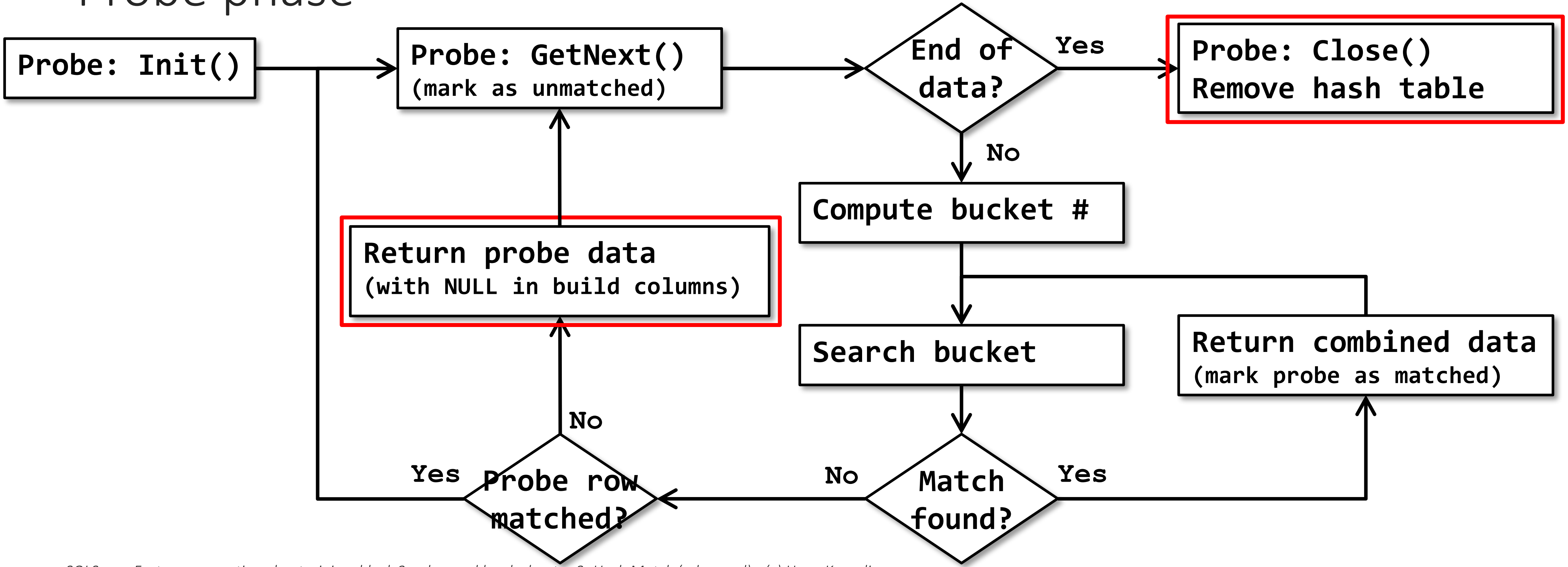
Hash Match (inner to right outer)

Probe phase



Hash Match (right outer join, no spills)

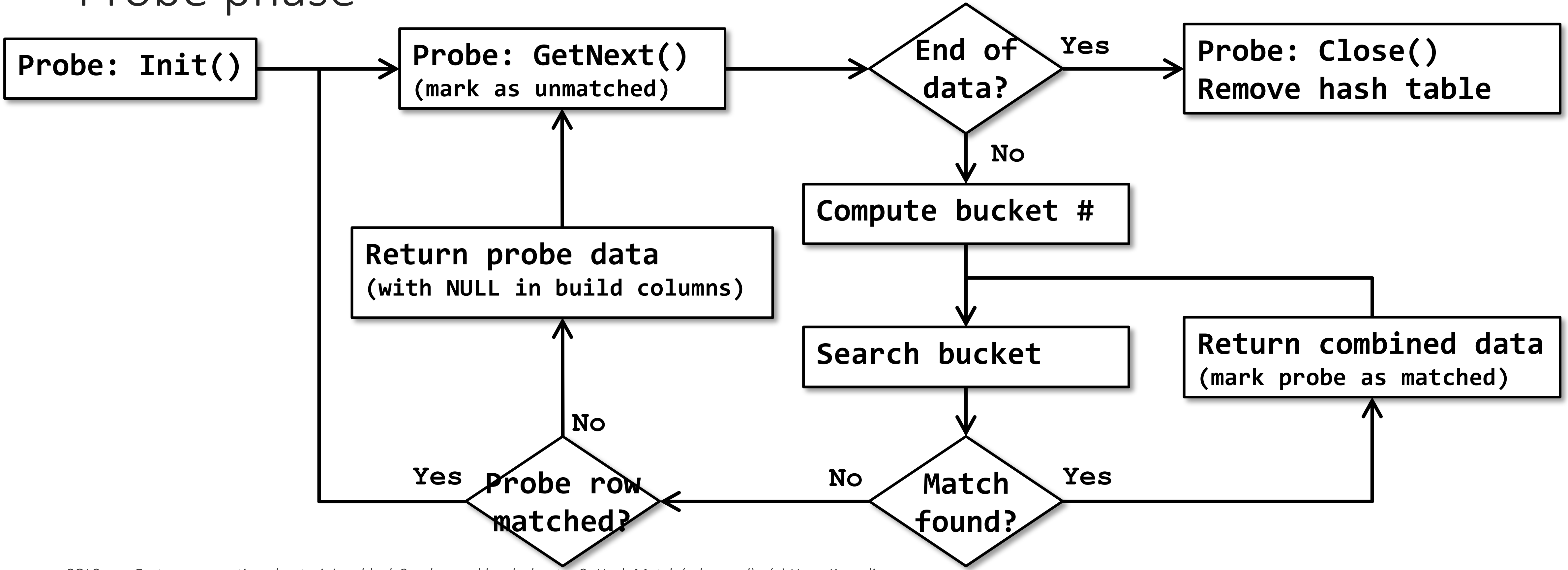
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

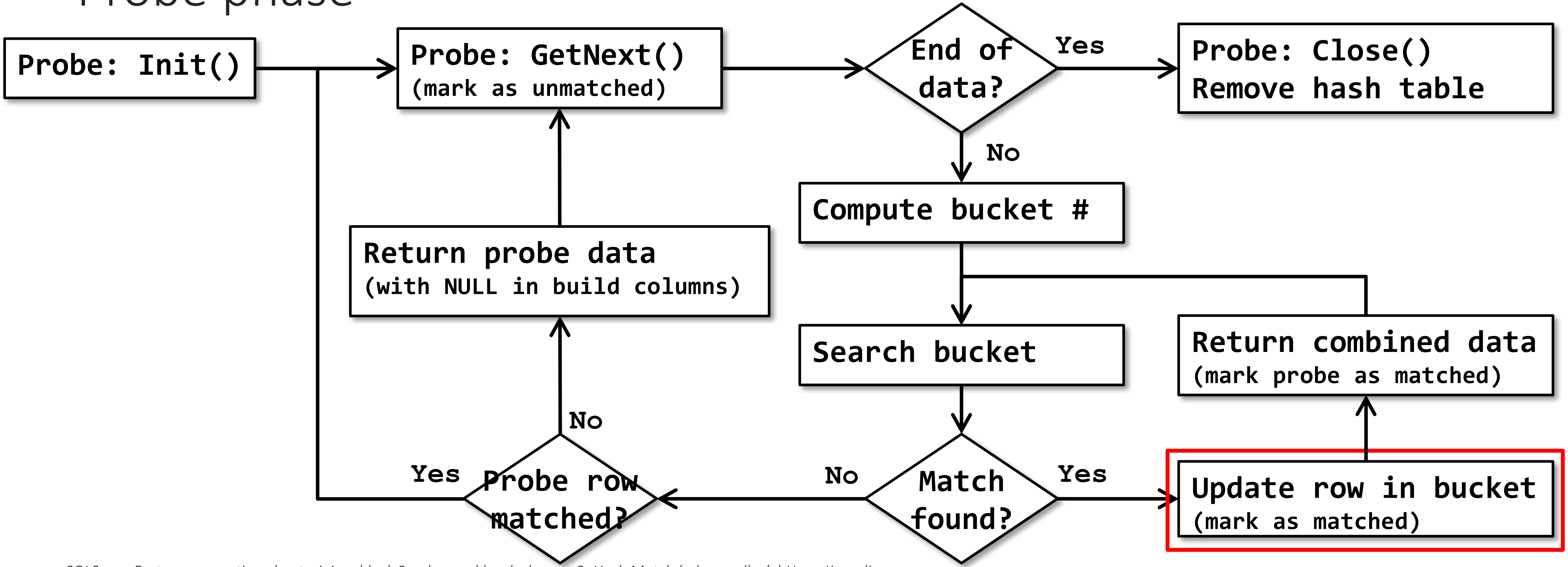
Hash Match (right outer to full outer)

Probe phase



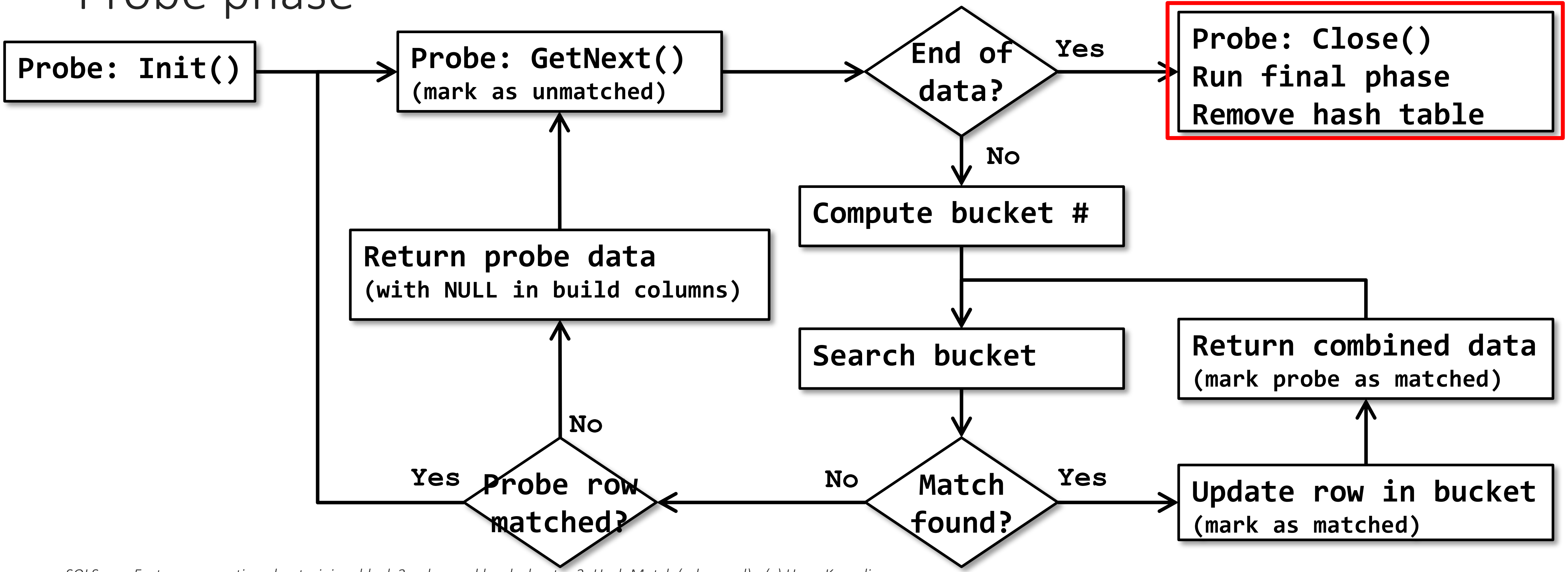
Hash Match (right outer to full outer)

Probe phase



Hash Match (full outer join, no spills)

Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

Hash Match (full outer join, no spills)

- Typical output order ***(when there are no spills!)***
  - All probe rows, matched and unmatched, in original order
  - All unmatched build rows, in “semi random” order
- This order is not guaranteed
  - But you should still be aware of it, in case it matters

Hash Match (inner to left semi)

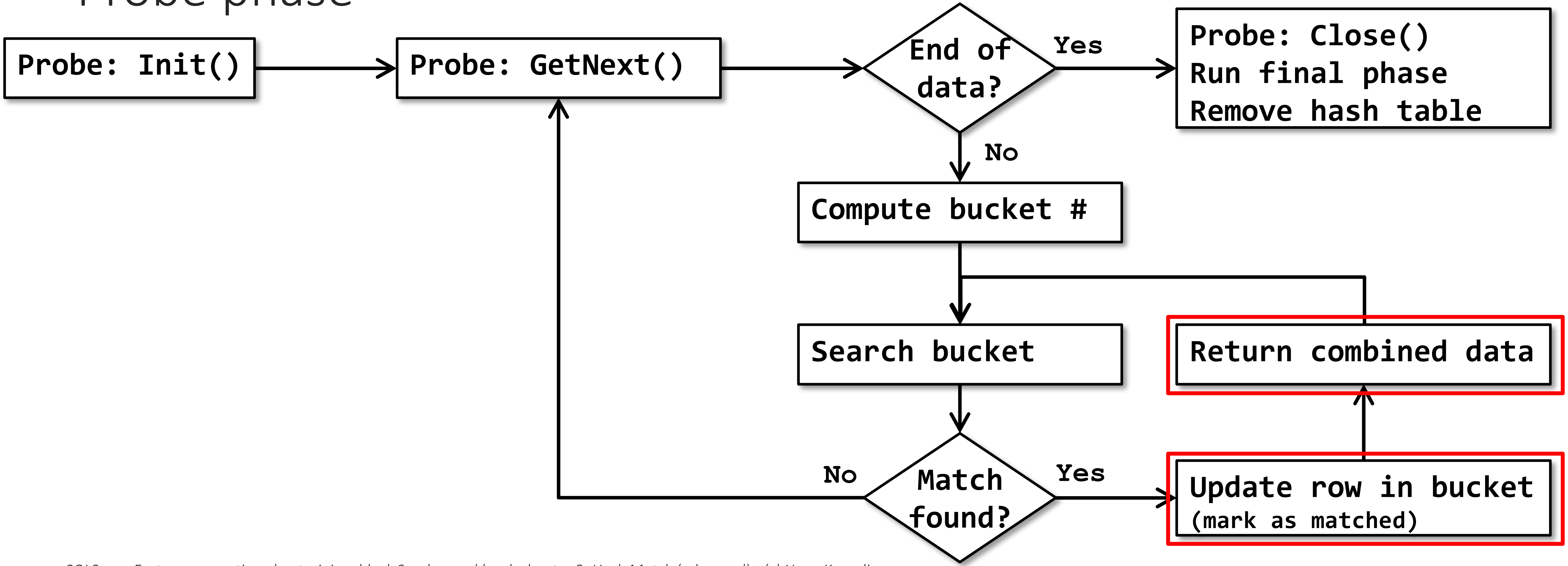
- From inner join to left semi join
  - Only return matched build rows
  - Return matched build rows once, regardless of number of matches
  - Two options
    - Return when first match found
      - Mark as matched
      - Skip on next match
    - Mark as matched when any match found
      - Return matched rows in final phase

Hash Match (left semi join, no spills)

- From inner join to left semi join
  - Only return matched build rows
  - Return matched build rows once, regardless of number of matches
  - Marks as matched during probe phase
  - Returns matched rows during final phase
  - Operator is now *fully* blocking
    - No data returned during build phase
    - No data returned during probe phase
    - All data returned during final phase

Hash Match (left outer to left semi)

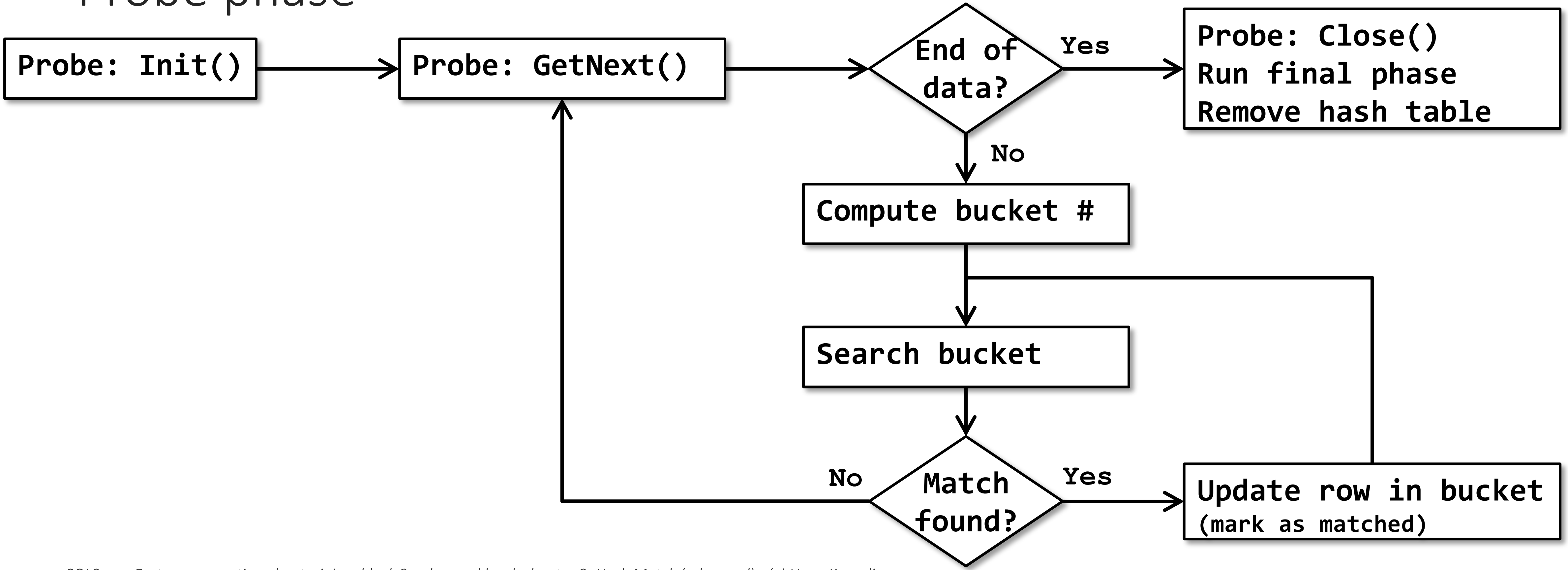
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

Hash Match (left semi join, no spills)

Probe phase



Probe: Init()

Probe: GetNext()

End of data?

Yes

Probe: Close()
Run final phase
Remove hash table

No

Compute bucket #

Search bucket

Match found?

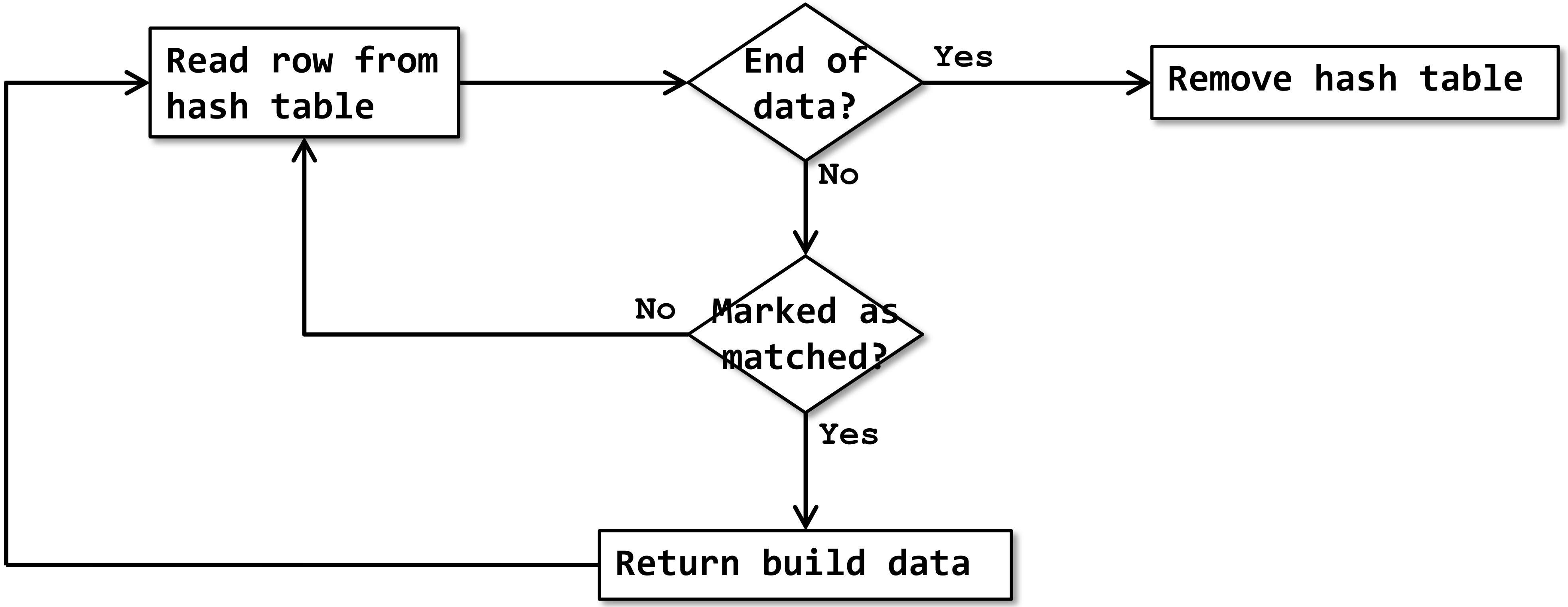
No

Yes

Update row in bucket
(mark as matched)

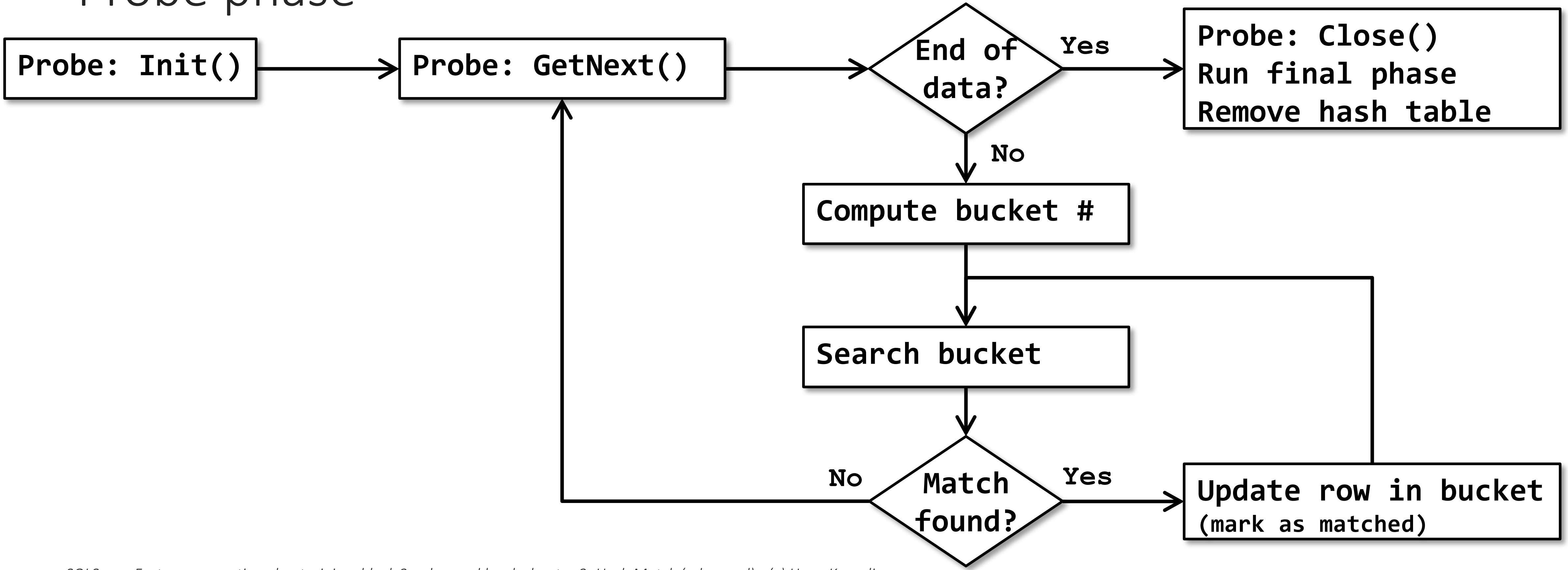
Hash Match (left semi join, no spills)

Final phase



Hash Match (left anti semi join, no spills)

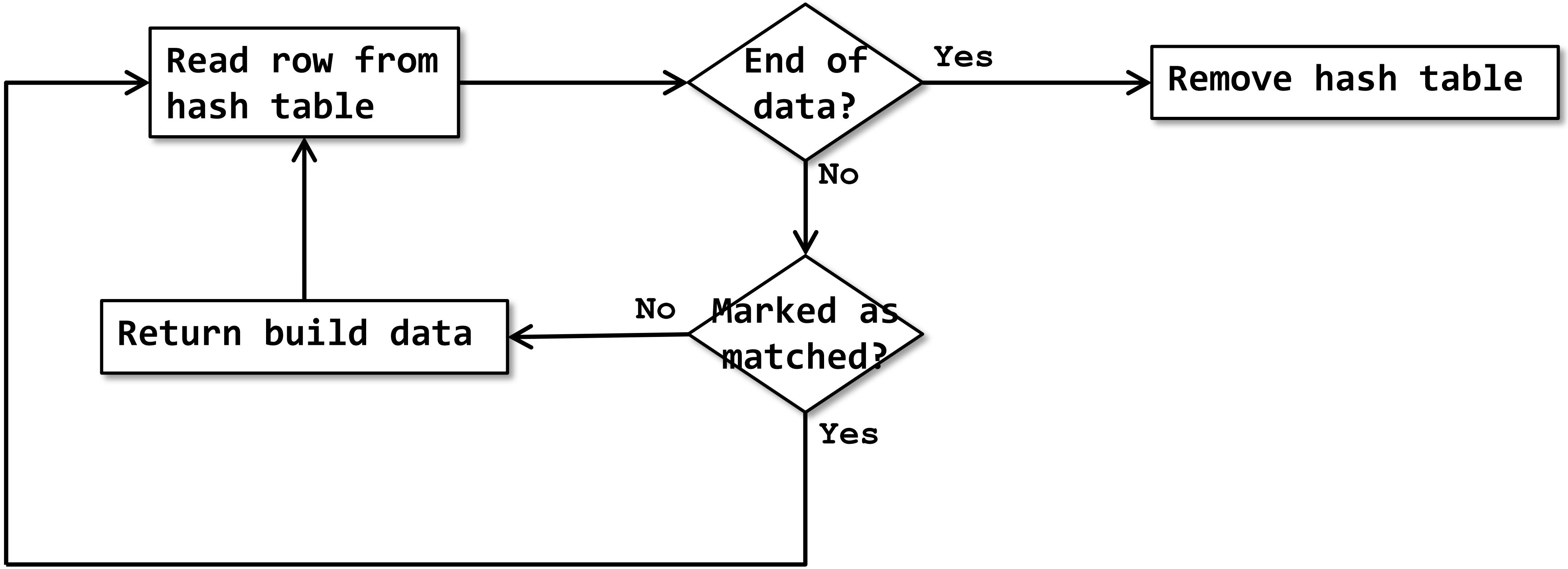
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

Hash Match (left anti semi join, no spills)

Final phase

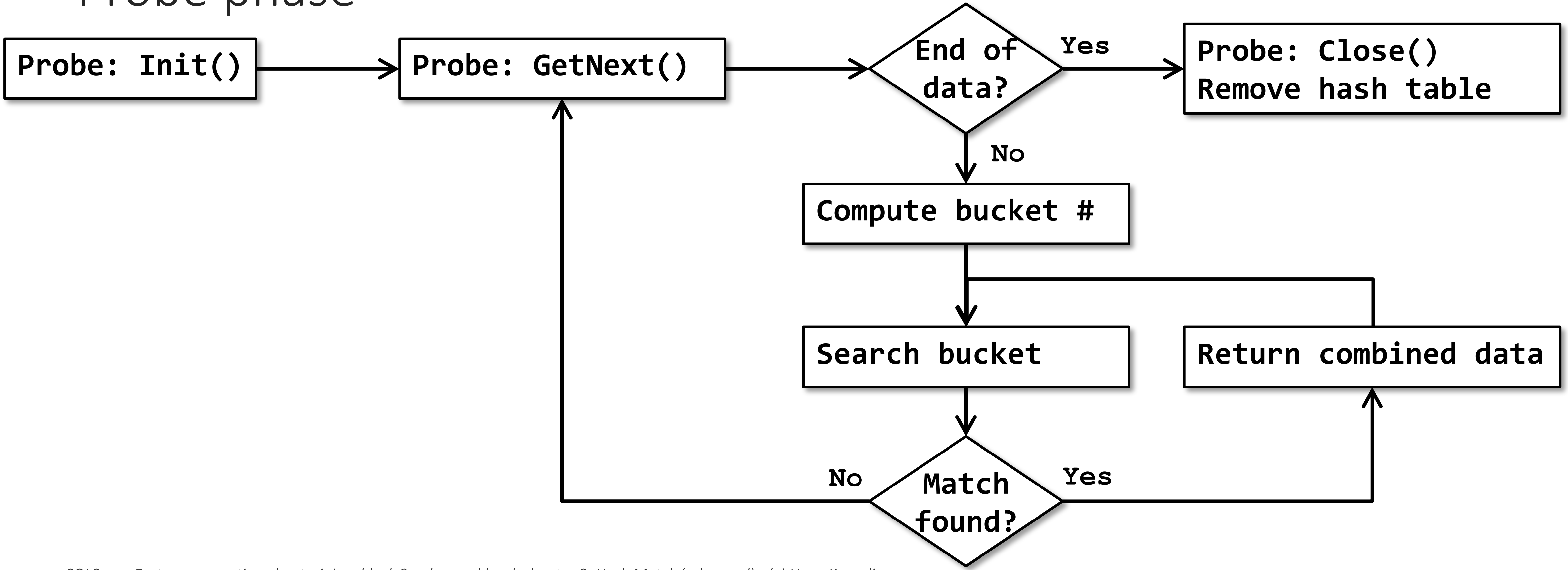


Hash Match (probed left semi join)

- Probed left semi join
  - Not supported
    - Not clear why
    - Would be relatively easy to build
    - Never encountered, never been able to repro

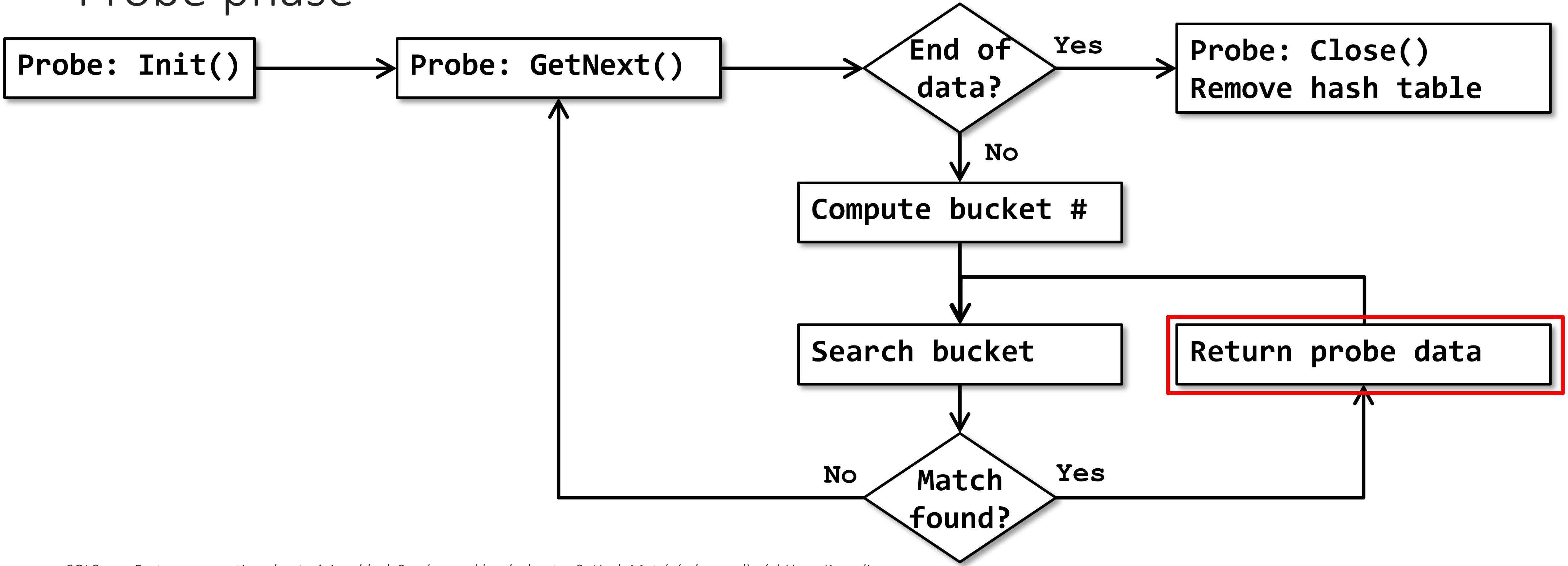
Hash Match (inner to right semi)

Probe phase



Hash Match (inner to right semi)

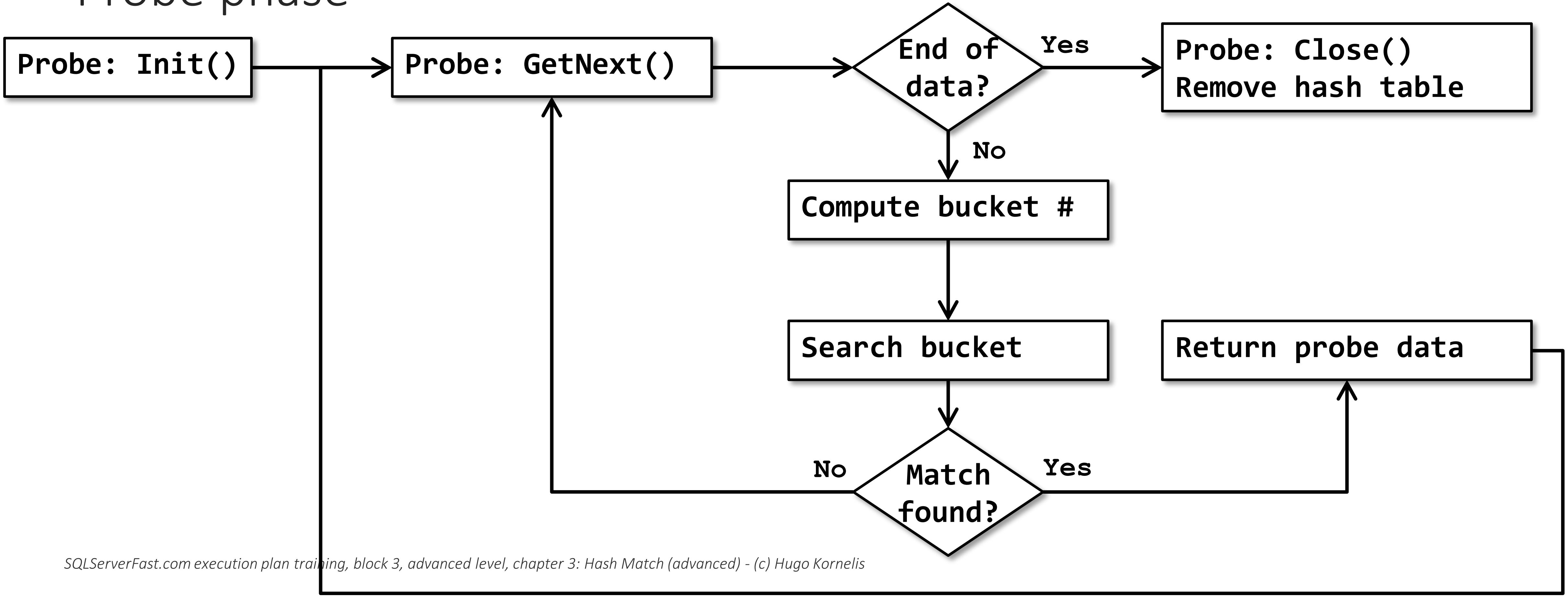
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

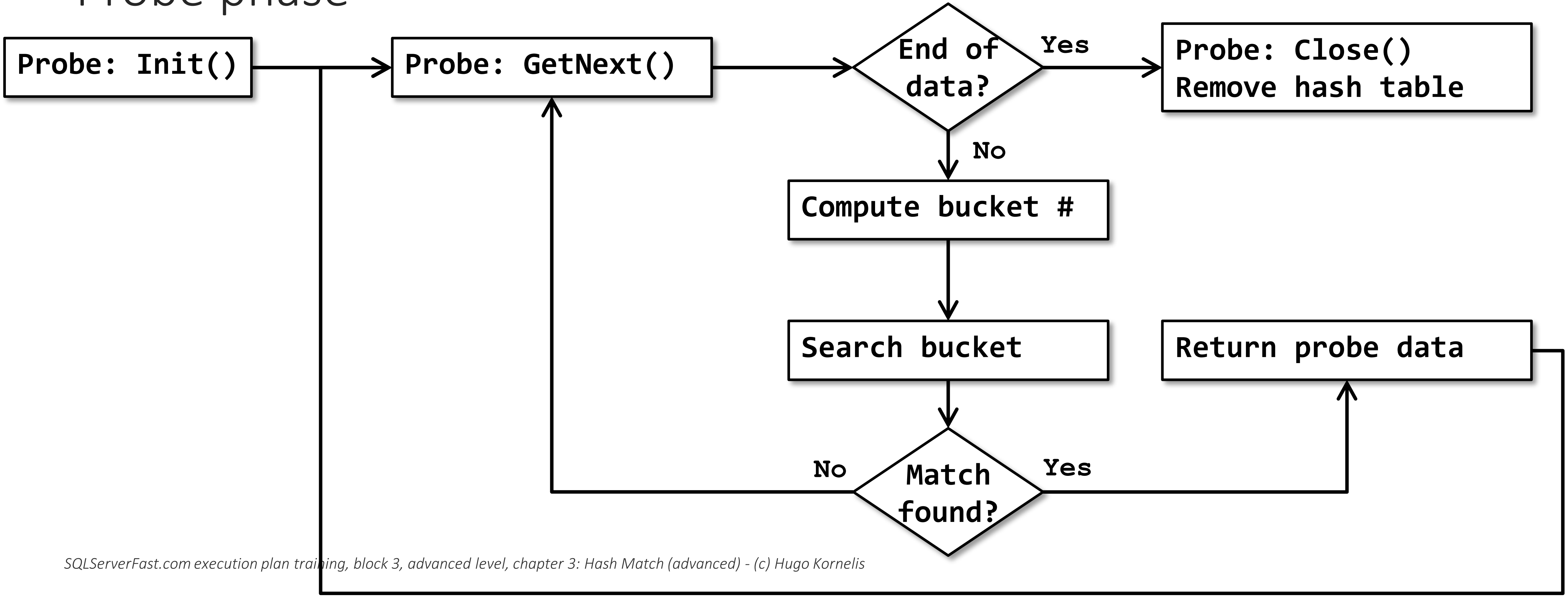
Hash Match (right semi join, no spills)

Probe phase



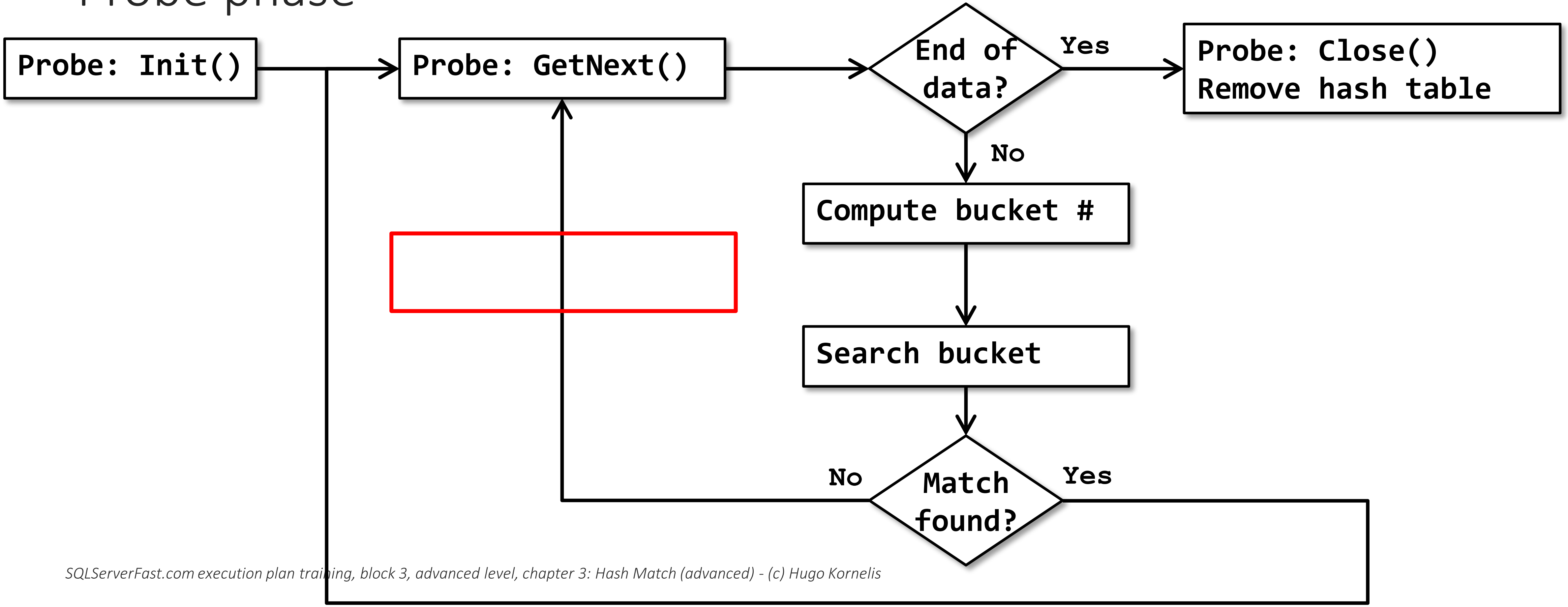
Hash Match (right semi to right anti semi)

Probe phase



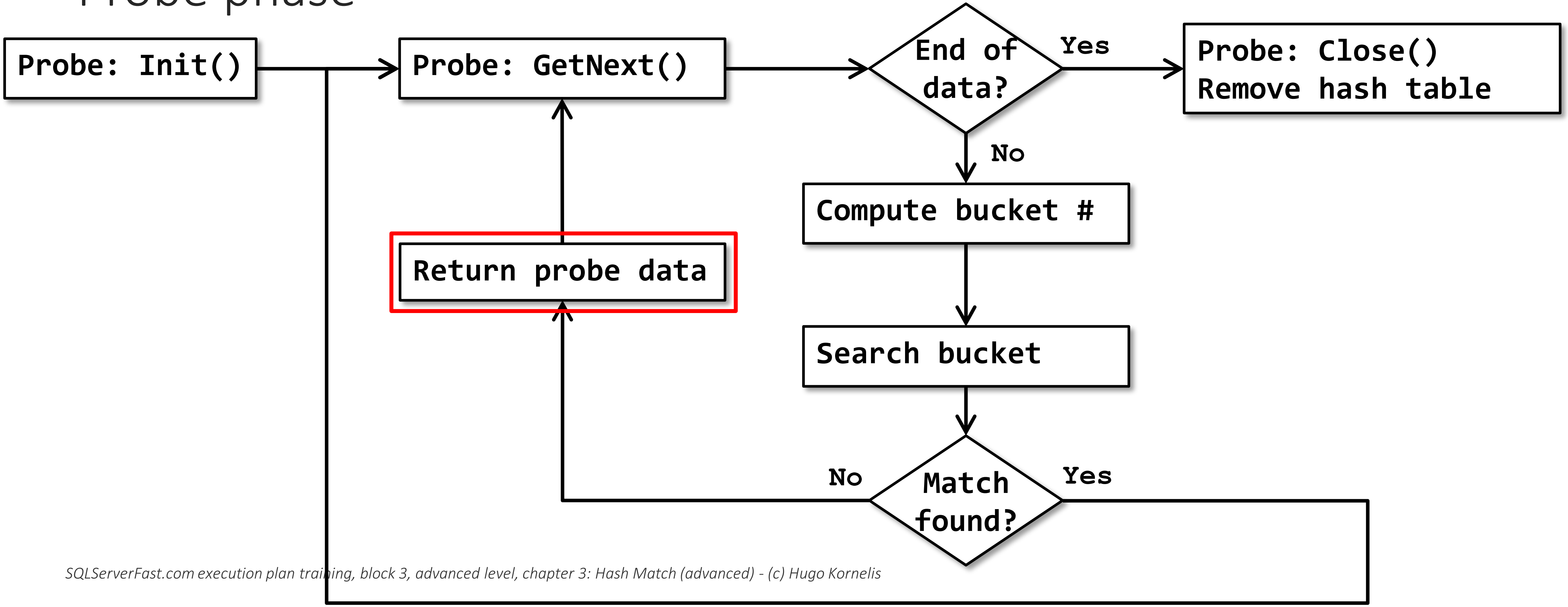
Hash Match (right semi to right anti semi)

Probe phase



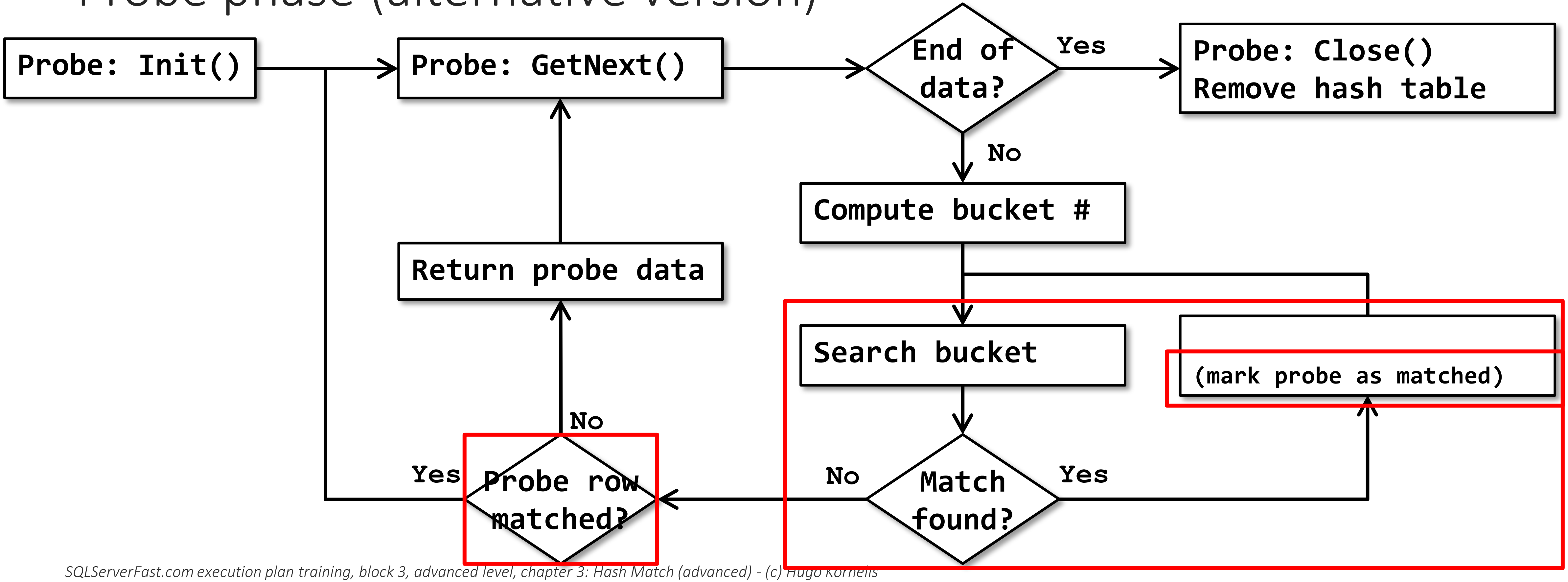
Hash Match (right anti semi join, no spills)

Probe phase



Hash Match (right anti semi join, no spills)

Probe phase (alternative version)



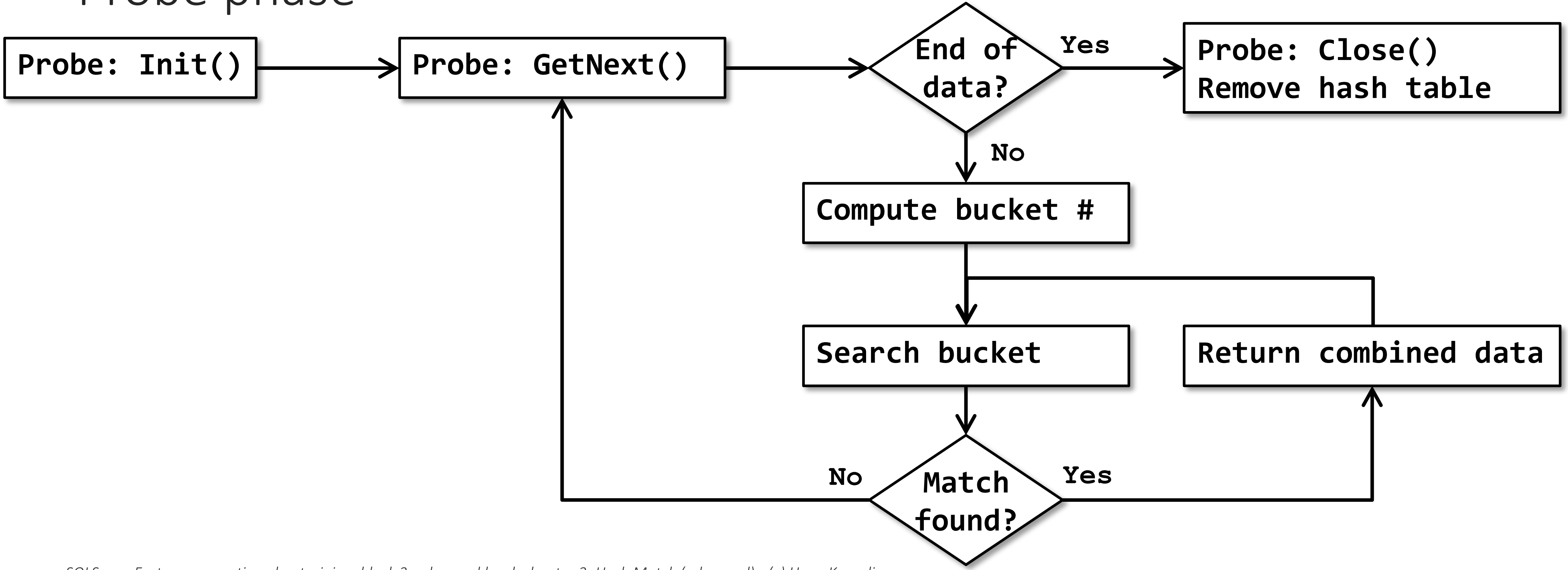
Hash Match (union)

- Union
  - Merge Join (Union)
    - Both inputs have to be free of duplicates already
    - Avoids new duplicates after combining the two inputs
  - Hash Match (Union)
    - Probe input has to be free of duplicates already
    - Duplicates in build input are removed by the operator
    - Avoids new duplicates after combining the two inputs

Hash Match
(Inner Join)

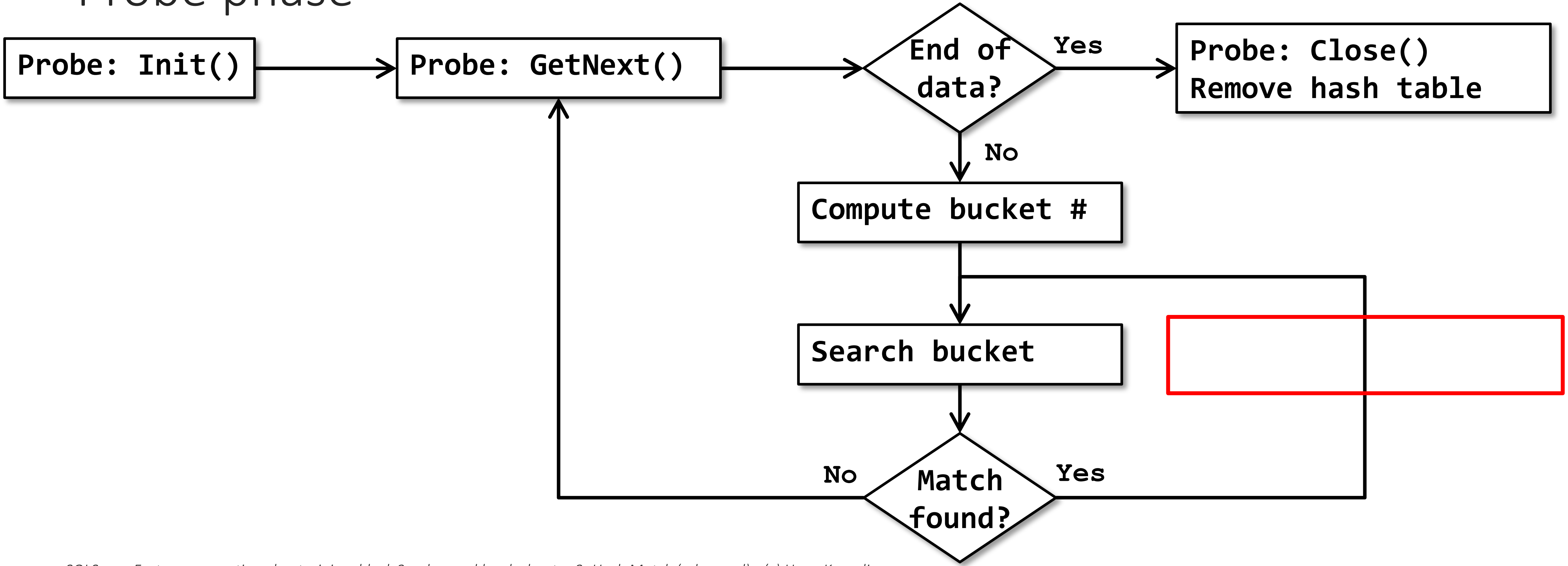
Hash Match (inner join to union)

Probe phase



Hash Match (inner join to union)

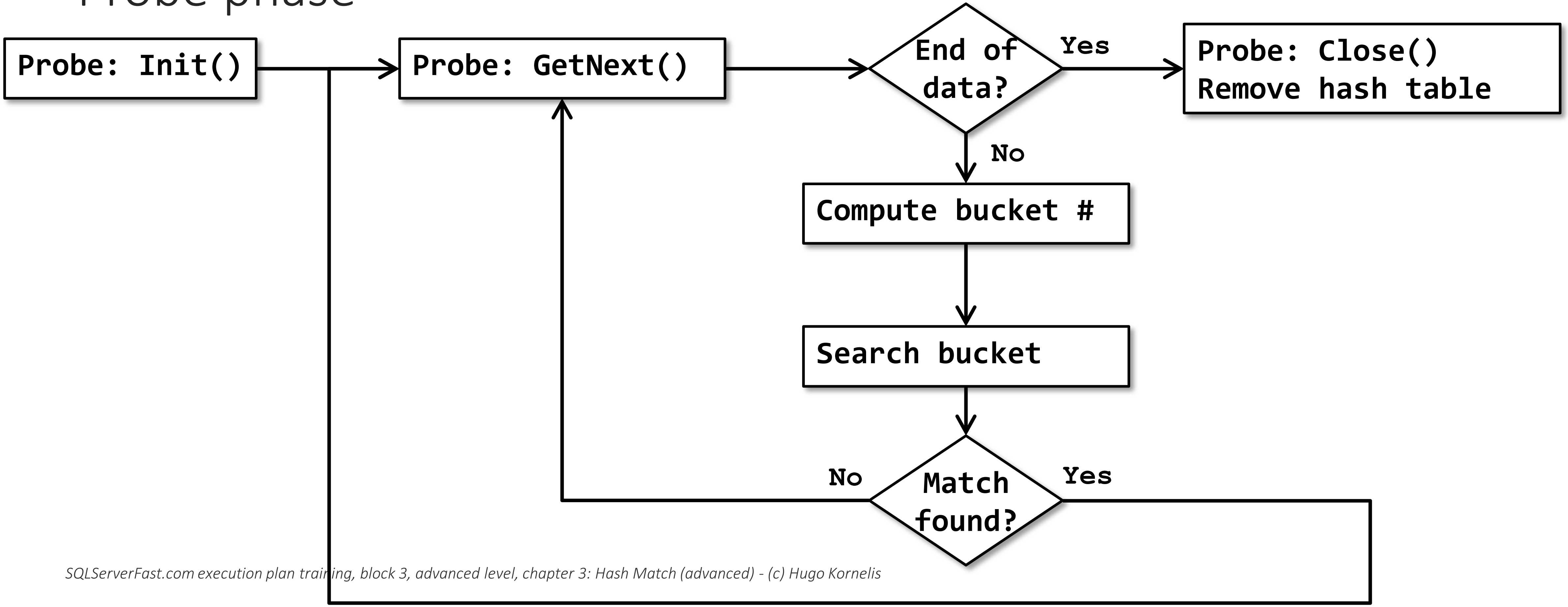
Probe phase



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

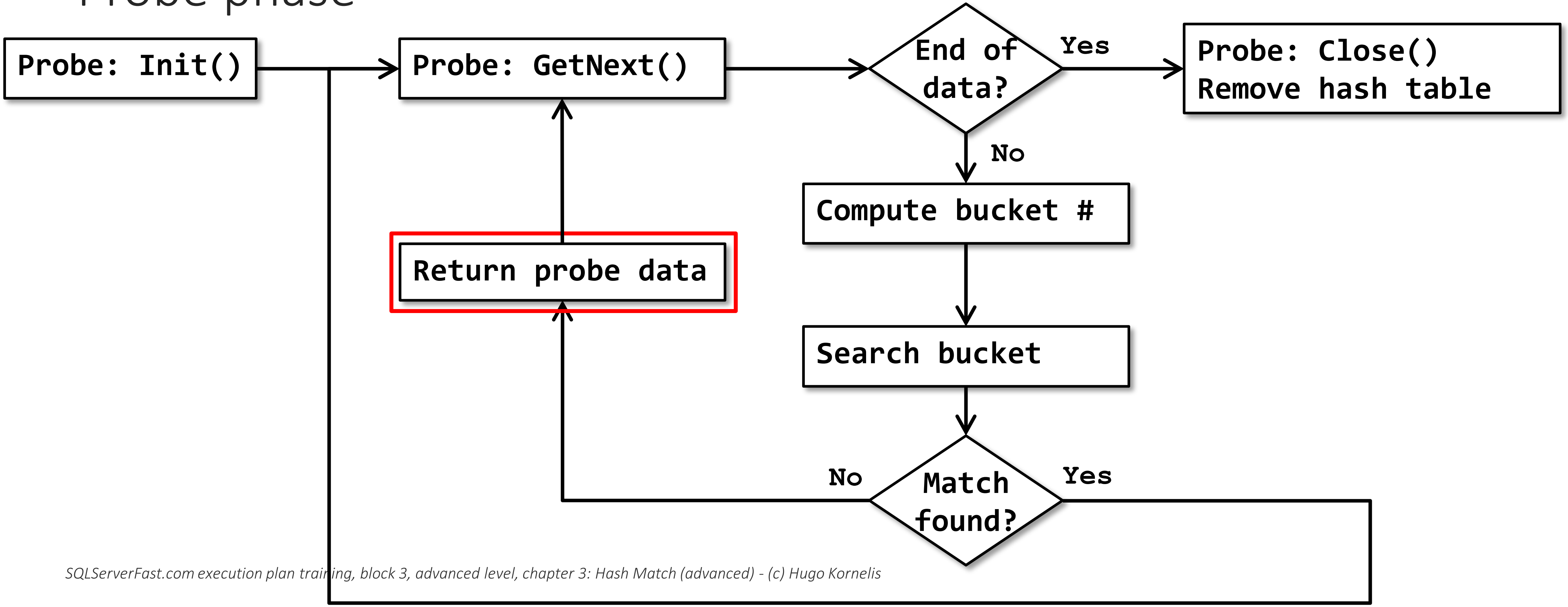
Hash Match (inner join to union)

Probe phase



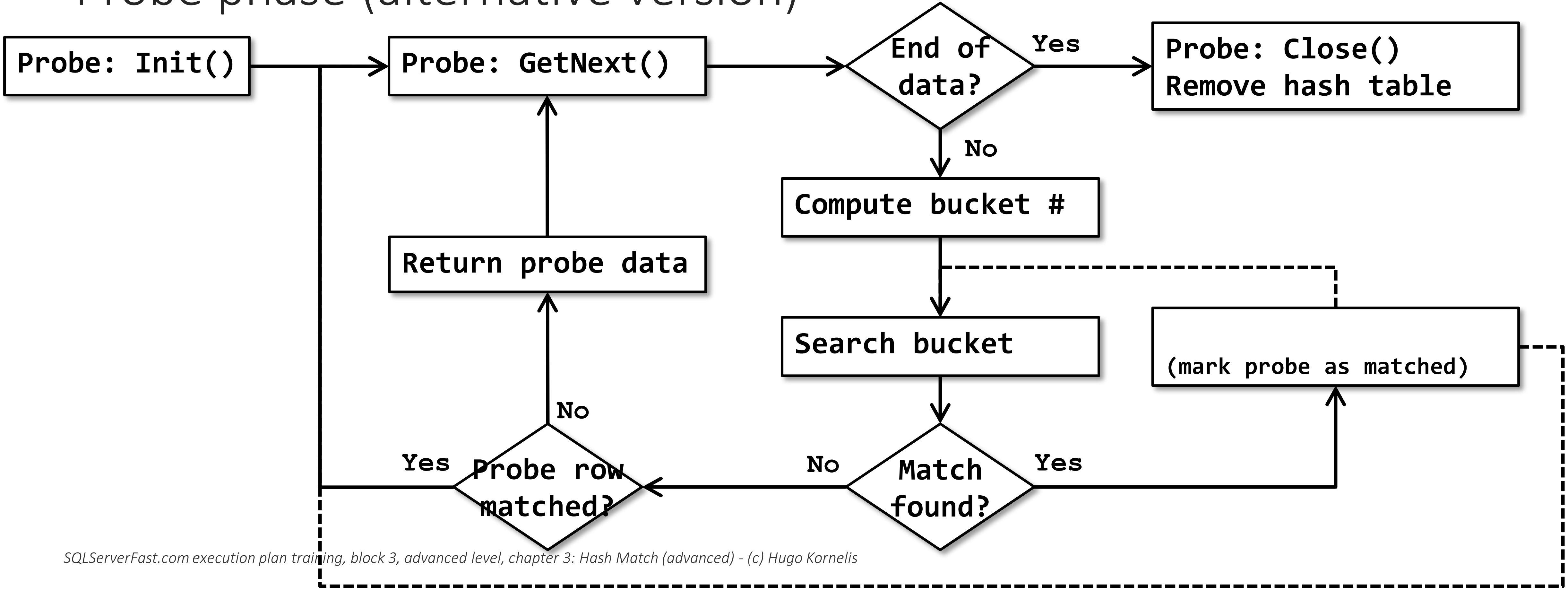
Hash Match (union, no spills)

Probe phase



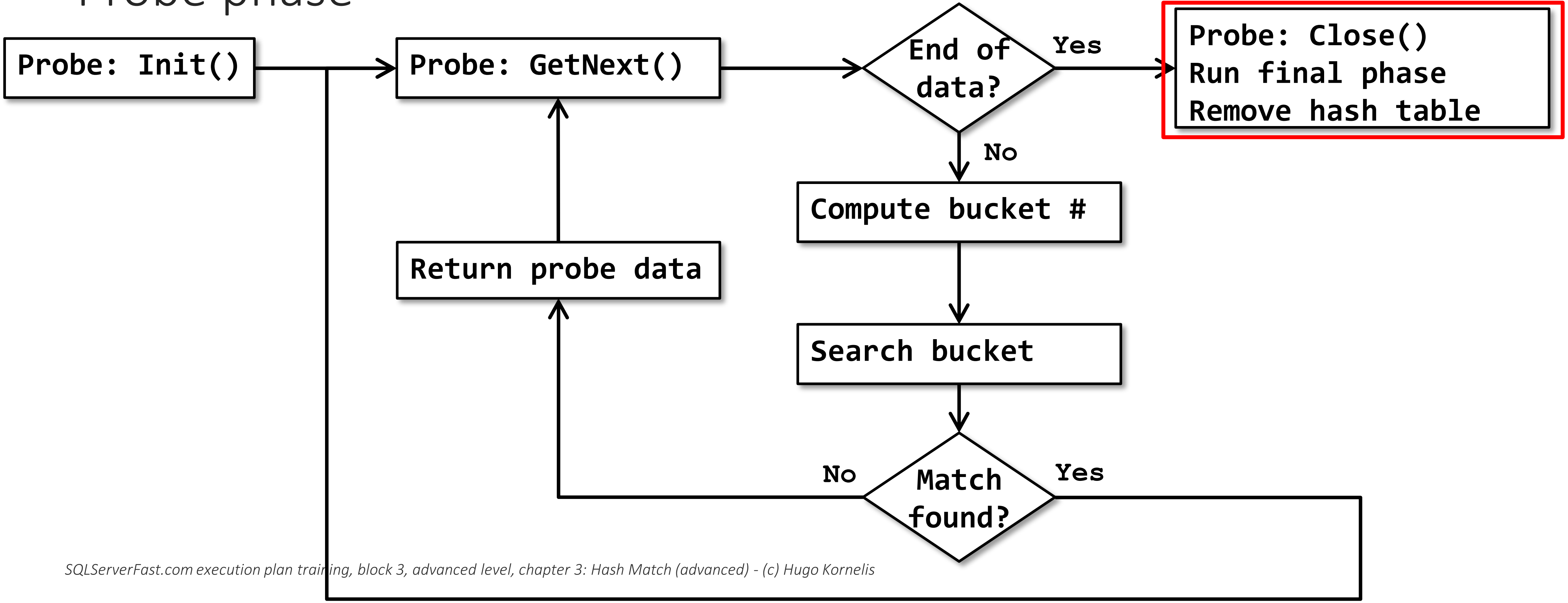
Hash Match (union, no spills)

Probe phase (alternative version)



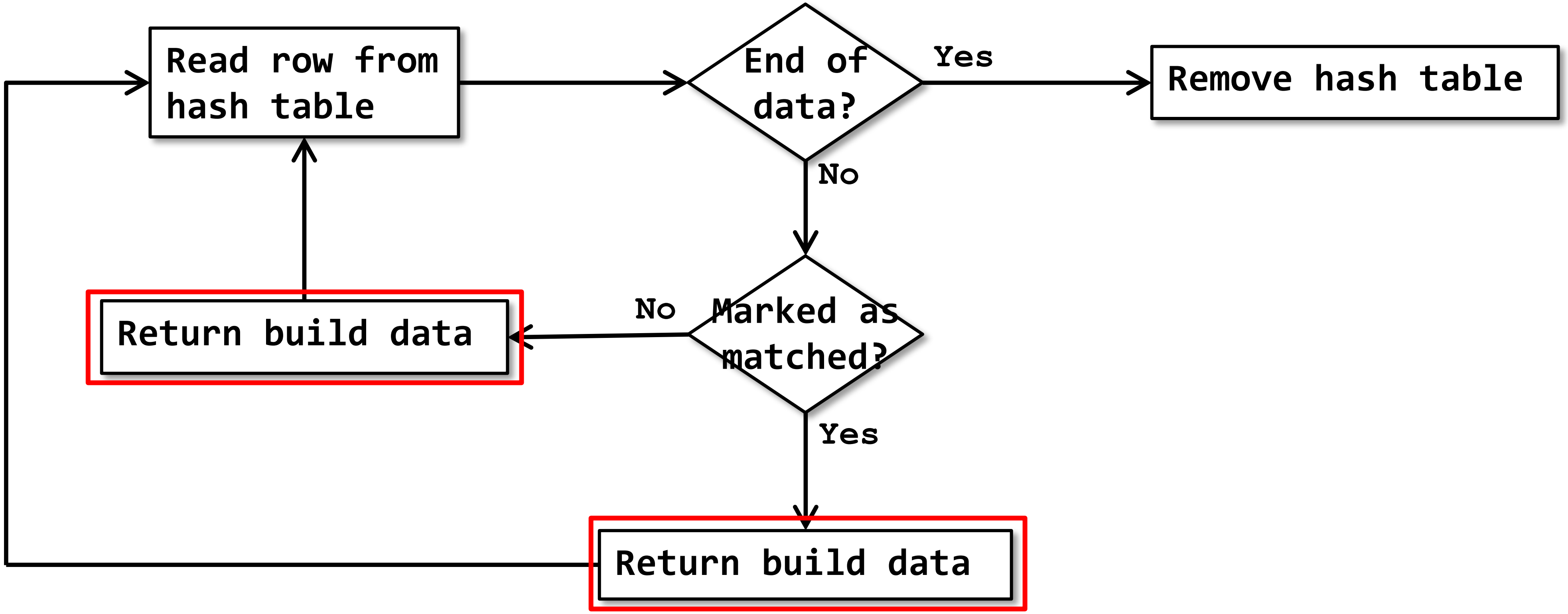
Hash Match (union, no spills)

Probe phase



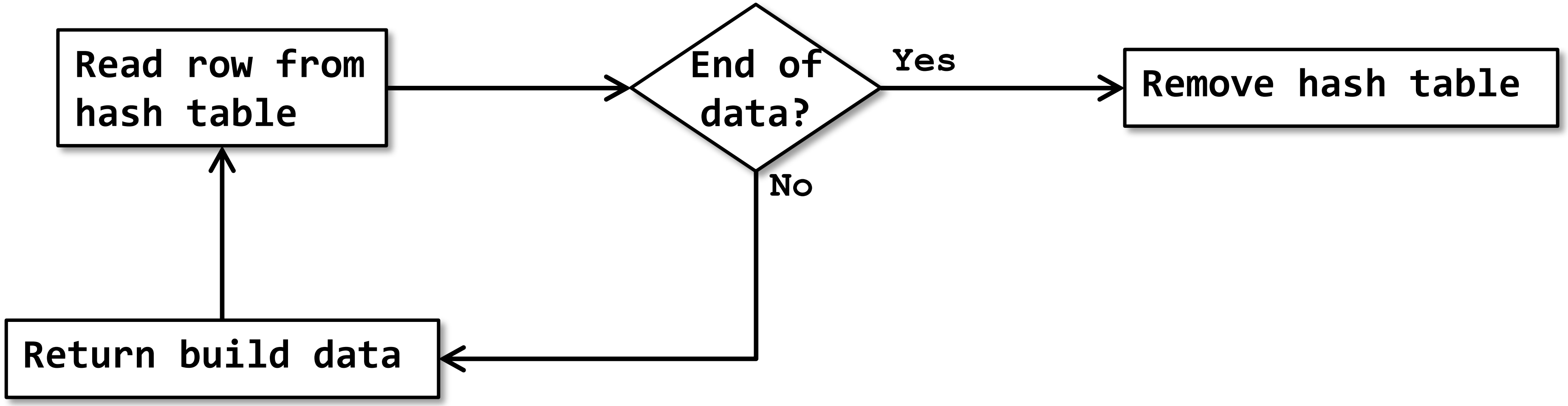
Hash Match (union, no spills)

Final phase



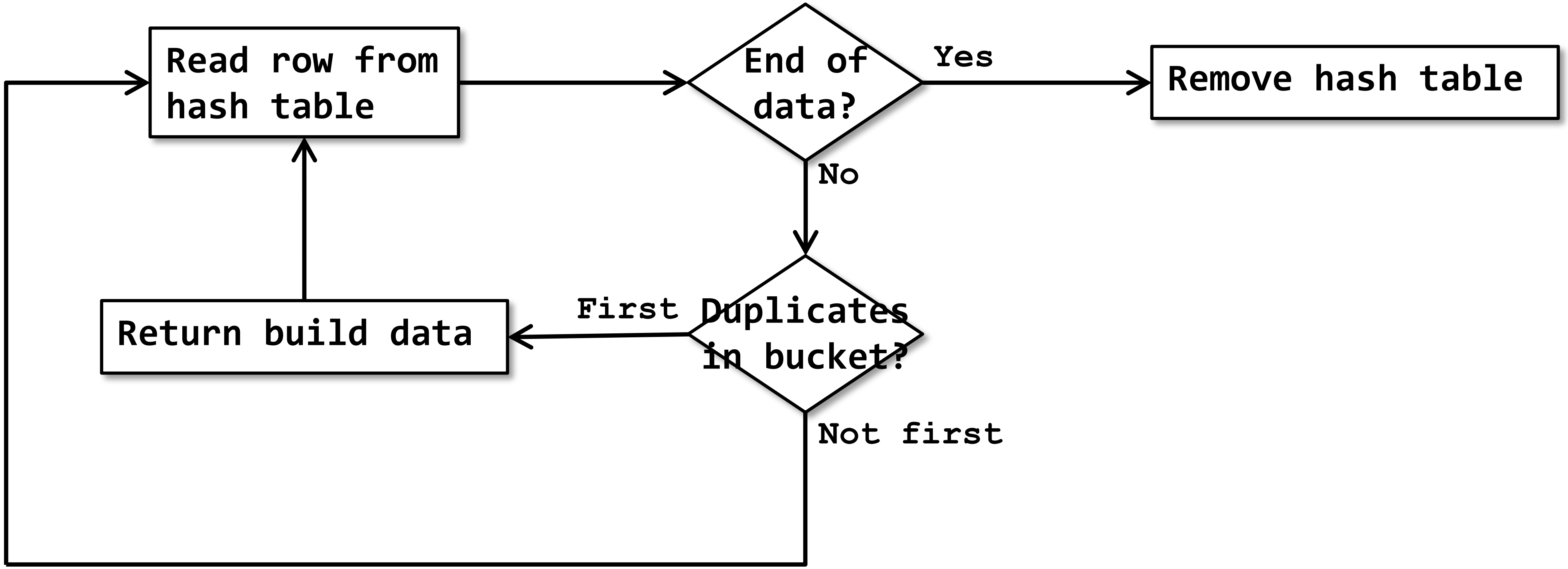
Hash Match (union, no spills)

Final phase



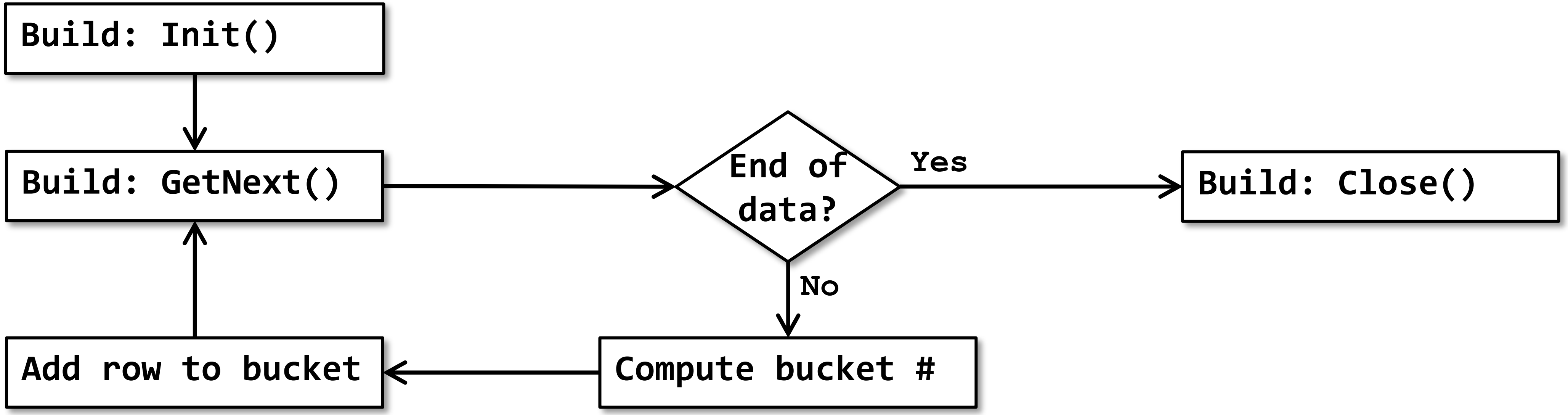
Hash Match (union, no spills)

Final phase (bad alternative, not used!!!)



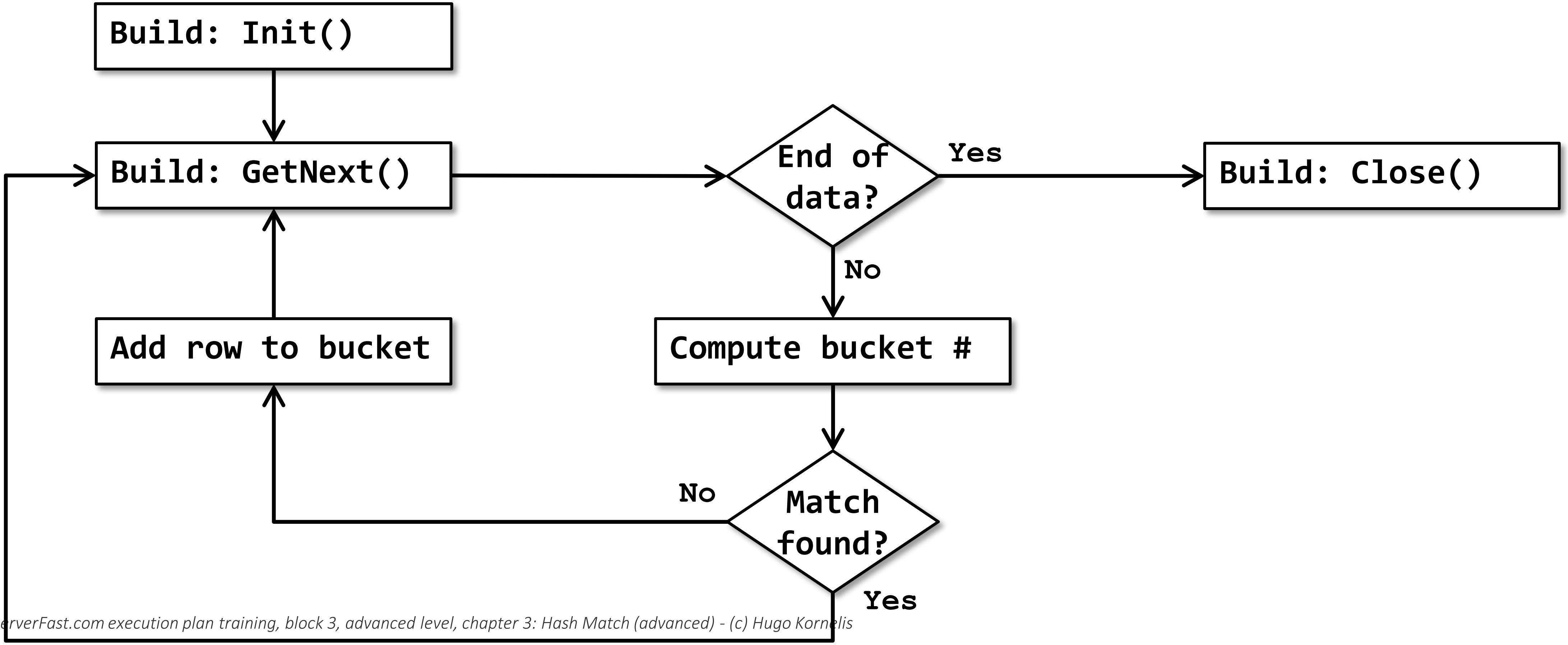
Hash Match (inner to union)

Build phase



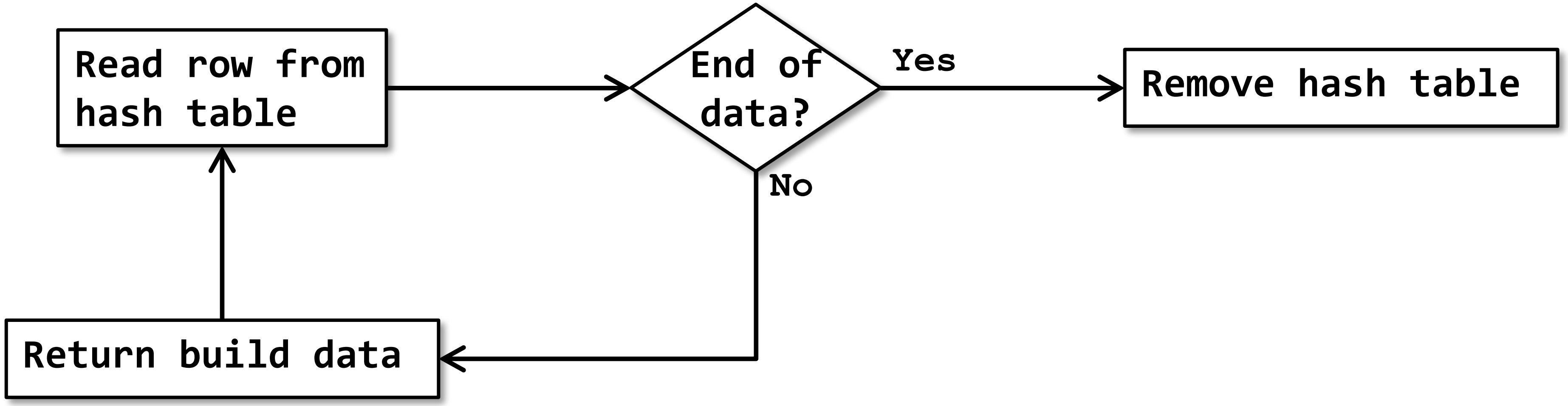
Hash Match (inner to union)

Build phase



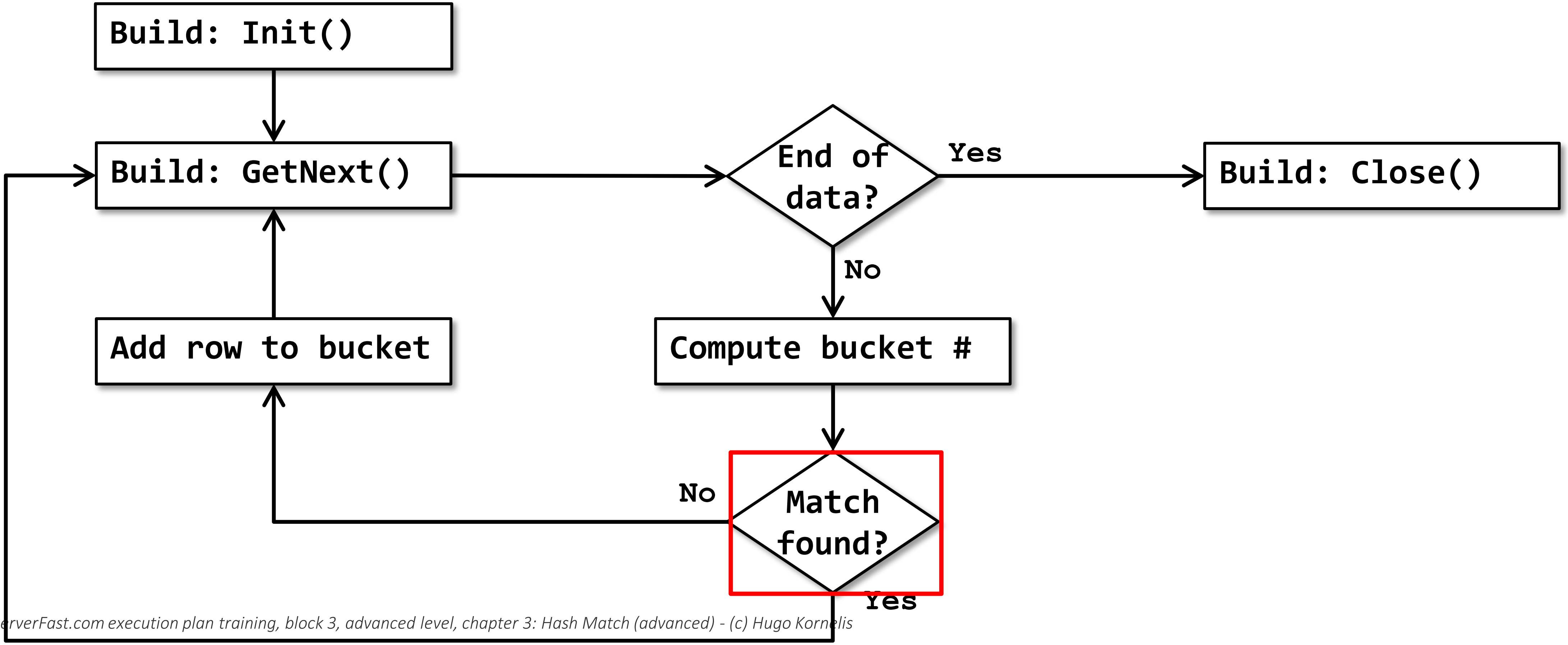
Hash Match (union, no spills)

Final phase



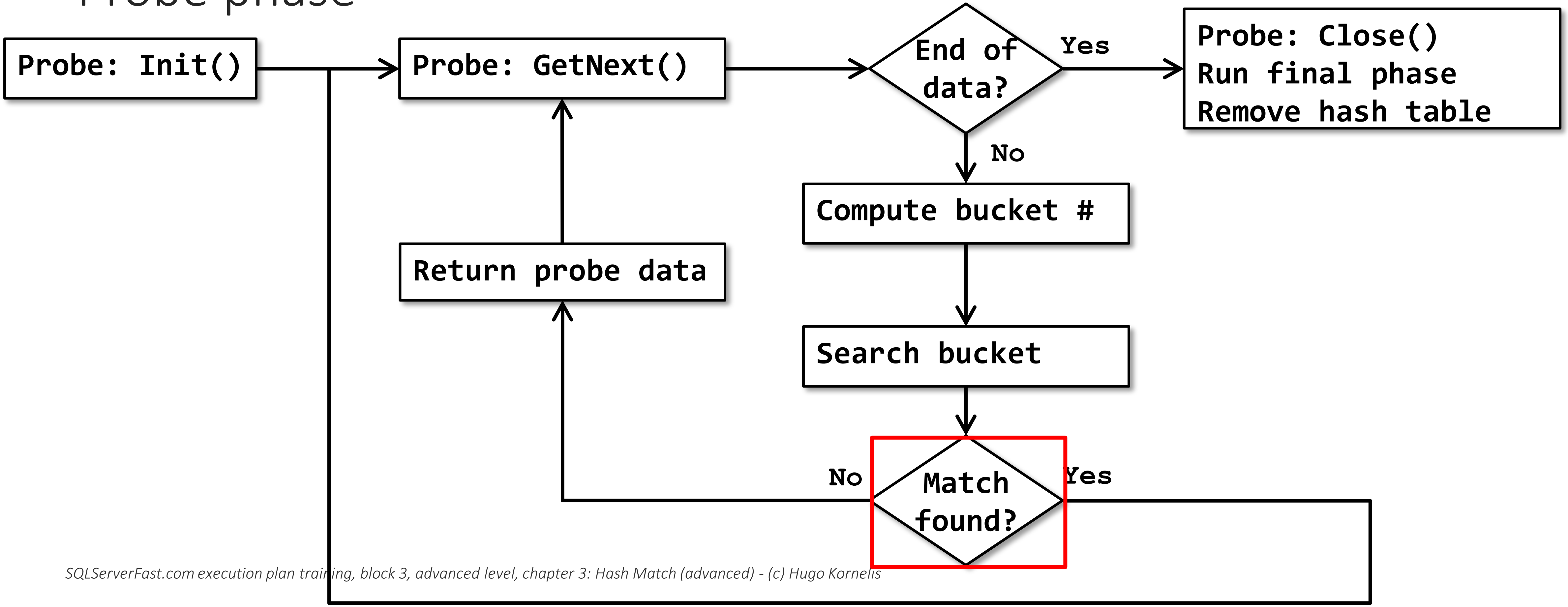
Hash Match (union, no spills)

Build phase



Hash Match (union, no spills)

Probe phase



Probe: Init()

Probe: GetNext()

End of data?

Yes

Probe: Close()
Run final phase
Remove hash table

No

Compute bucket #

Search bucket

Match found?

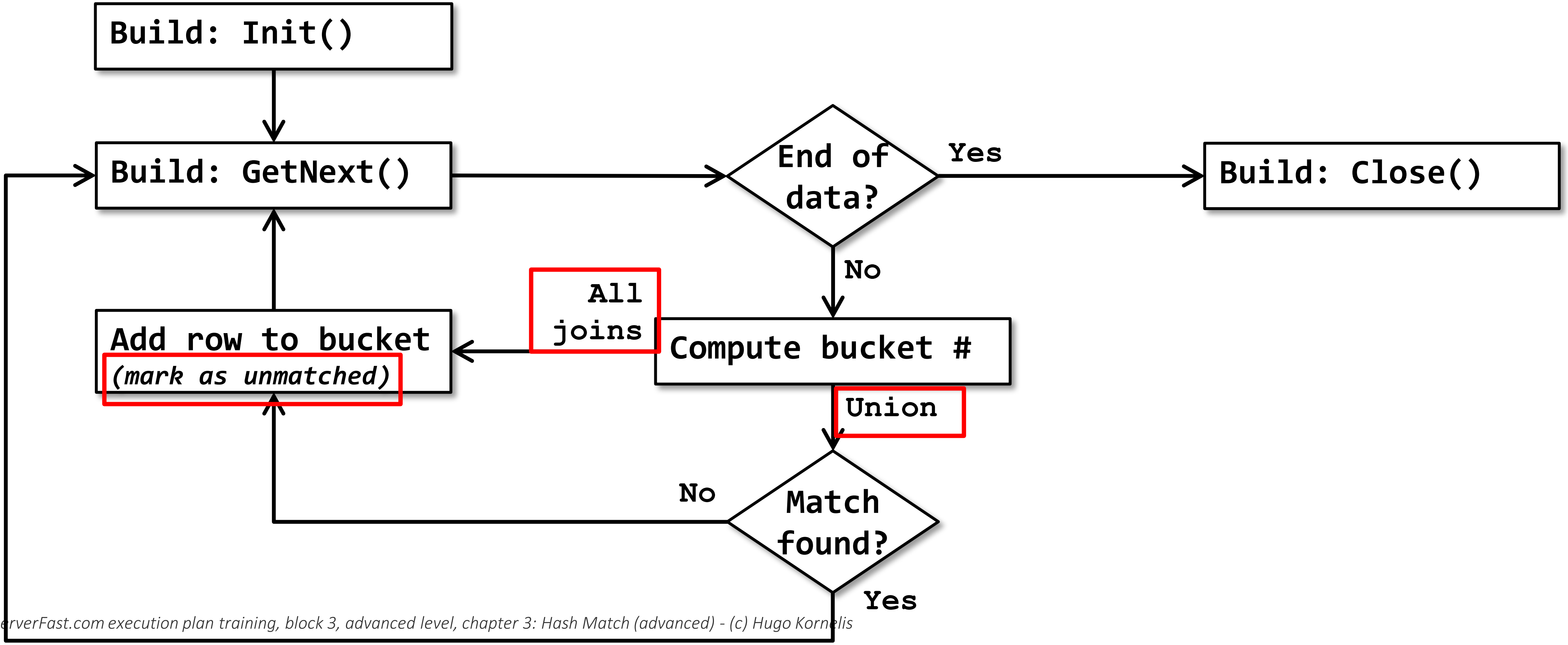
No

Yes

Return probe data

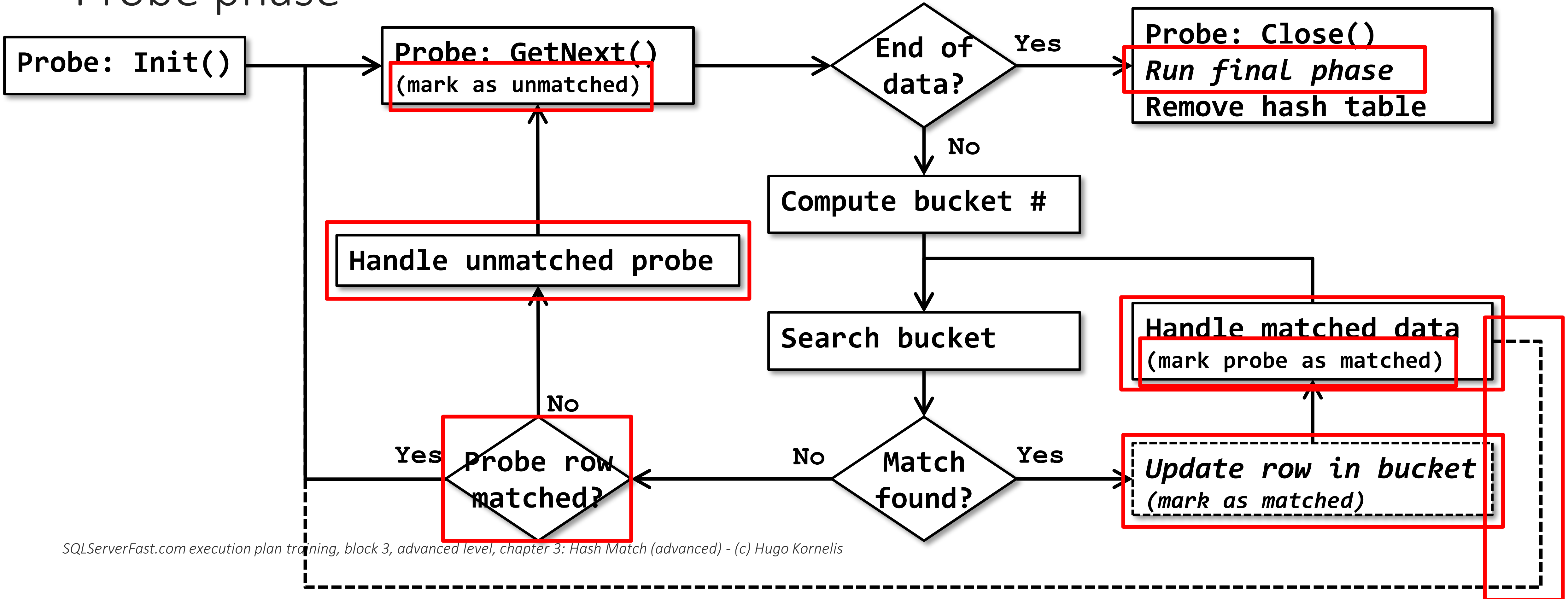
Hash Match (all operations, no spills)

Build phase



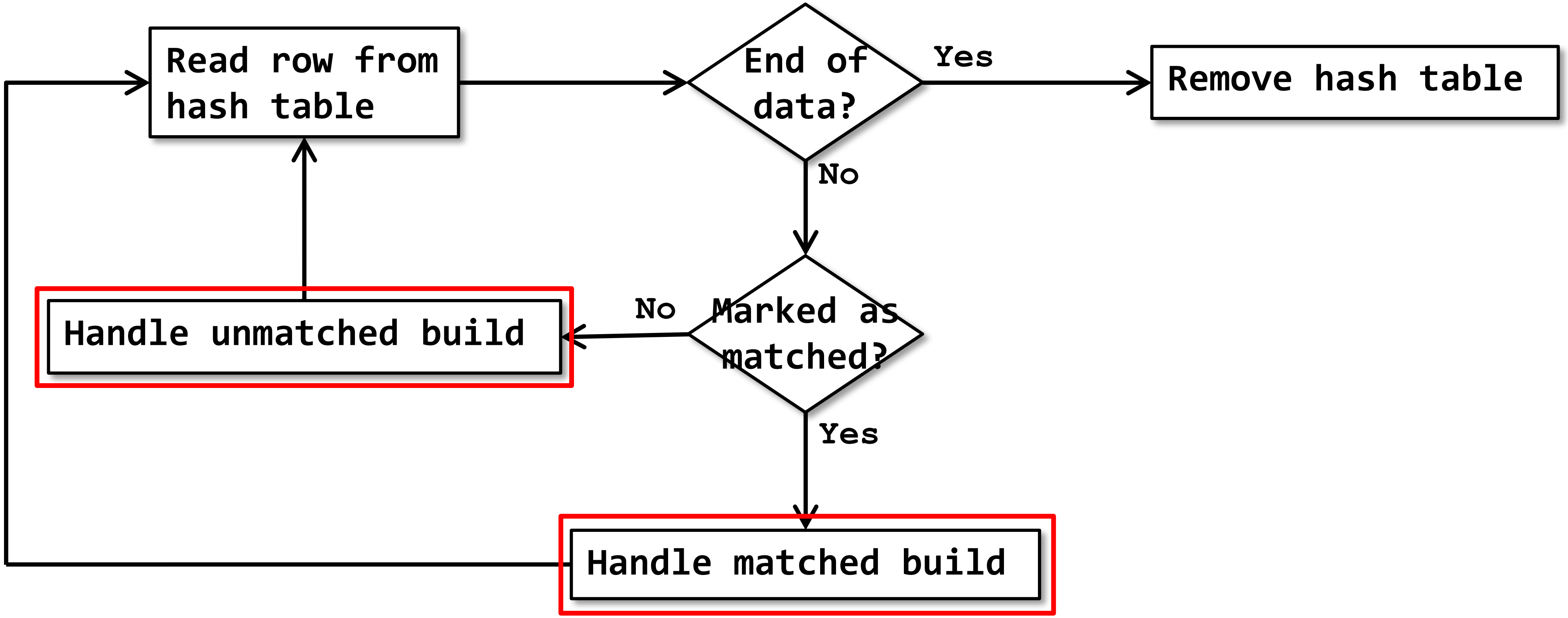
Hash Match (all operations, no spills)

Probe phase



Hash Match (all operations, no spills)

Final phase



Hash Match

- Memory
  - Hash table stored in memory
  - Memory Grant
    - Determined by optimizer
    - Based on estimates
    - May be adjusted for later executions by Memory Grant Feedback
      - Since SQL Server 2017 for batch mode
      - Since SQL Server 2019 for row mode
    - No additional memory allocations once query runs
      - (there are some rare exceptions to this in batch mode plans only)

Hash Match
(Inner Join)

Hash Match

- Memory
  - Hash table stored in memory
  - Memory Grant
  - What if the build input is larger than the available memory?
    - Fail with run-time error
    - Different version of algorithm
      - Still returns correct results
      - Performance suffers

Hash Match

- Memory
  - Hash table stored in memory
  - Memory Grant
  - What if the build input is larger than the available memory?
    - ~~Fail with run-time error~~
    - Different version of algorithm
      - Still returns correct results
      - Performance suffers
      - Uses tempdb to store data that doesn’t fit in memory
        - This data is then read back and processed later
        - Called “spilling” to tempdb
        - Indicated in execution plan plus run-time statistics

Hash Match

- Memory
  - Hash table stored in memory
  - Memory Grant
  - What if the build input is larger than the available memory?
    - ~~Fail with run-time error~~
    - Different version of algorithm
      - Still returns correct results
      - Performance suffers
      - Uses tempdb to store data that doesn’t fit in memory
        - This data is then read back and processed later
        - Called “spilling” to tempdb
        - Indicated in execution plan plus run-time statistics, Extended Events, and SQL Trace

Hash Match

- Hash spill in detail
  - Can only occur during the build phase
    - Build row read, but no memory available to store it
      - Algorithm changes from “in-memory hash join” to “dynamic hash join” or “grace hash join”

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
    - Hash table divided into several partitions
      - Partition number determined by hashing the bucket number
    - One “active” partition remains in memory
      - (For grace hash join, zero partitions remain in memory)
    - All inactive partitions spill to tempdb
      - Data already in hash table moved to “files” in tempdb
    - Rest of build phase stores data in memory or in appropriate file
    - End of build phase: active partition in memory, rest in files

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
    - Build phase: Active partition in memory, inactive partitions in tempdb files
    - Probe phase:
      - Rows in active partition can be regularly processed
      - Rows in inactive partitions stored in yet more tempdb files
      - This finds matches and non-matches for active partition only
        - Execute final phase (if needed) to complete final results for active partition
      - No results at all for inactive partitions
        - But build and probe data for these partitions is now in separate files in tempdb

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
    - Build phase: Active partition in memory, inactive partitions in tempdb files
    - Probe phase: Results for active partition, inactive partitions in tempdb files
    - Process inactive partitions
      - Make active
      - Read build file, store data in hash table
      - Read probe file, handle matches and non-matches as they are found
      - Execute final phase (if needed)
    - Repeat for each inactive partition

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
    - Multiple iterations of build, probe, and final phase
    - First iteration
      - Reads build input, stores in memory or in tempdb
      - Reads probe input, produces partial results or stores in tempdb
    - Later iterations
      - Reads build data from tempdb, stores in memory
      - Reads probe data from tempdb, produces partial results

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
  - Recursive hash join
    - Initial partitions too large
    - Partitions are split into new, smaller partitions

Hash Match
(Inner Join)

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
  - Recursive hash join
  - Bail-out
    - When recursive hash join fails
    - Merge Join or Nested Loops for only those partition(s)
      - Input from files in tempdb

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
  - Recursive hash join
  - Bail-out
  - Bit-vector filtering
    - Probe rows that match empty bucket are handled immediately
    - This reduces size of probe files for inactive partitions

Hash Match
(Inner Join)

Hash Match

- Hash spill in detail
  - Dynamic hash join / grace hash join
  - Recursive hash join
  - Bail-out
  - Bit-vector filtering
  - Dynamic role reversal
    - Partitions that start inactive
    - Reverse build and probe if probe file has less rows

Hash Match

- Hash spill
  - Data split into multiple partitions
  - Data for partitions written to tempdb
  - Partitions processed one by one
- Output
  - Start with first partition
    (may be mixed with some unmatched probe rows)
  - Then second partition, etc
  - Effectively (semi) random

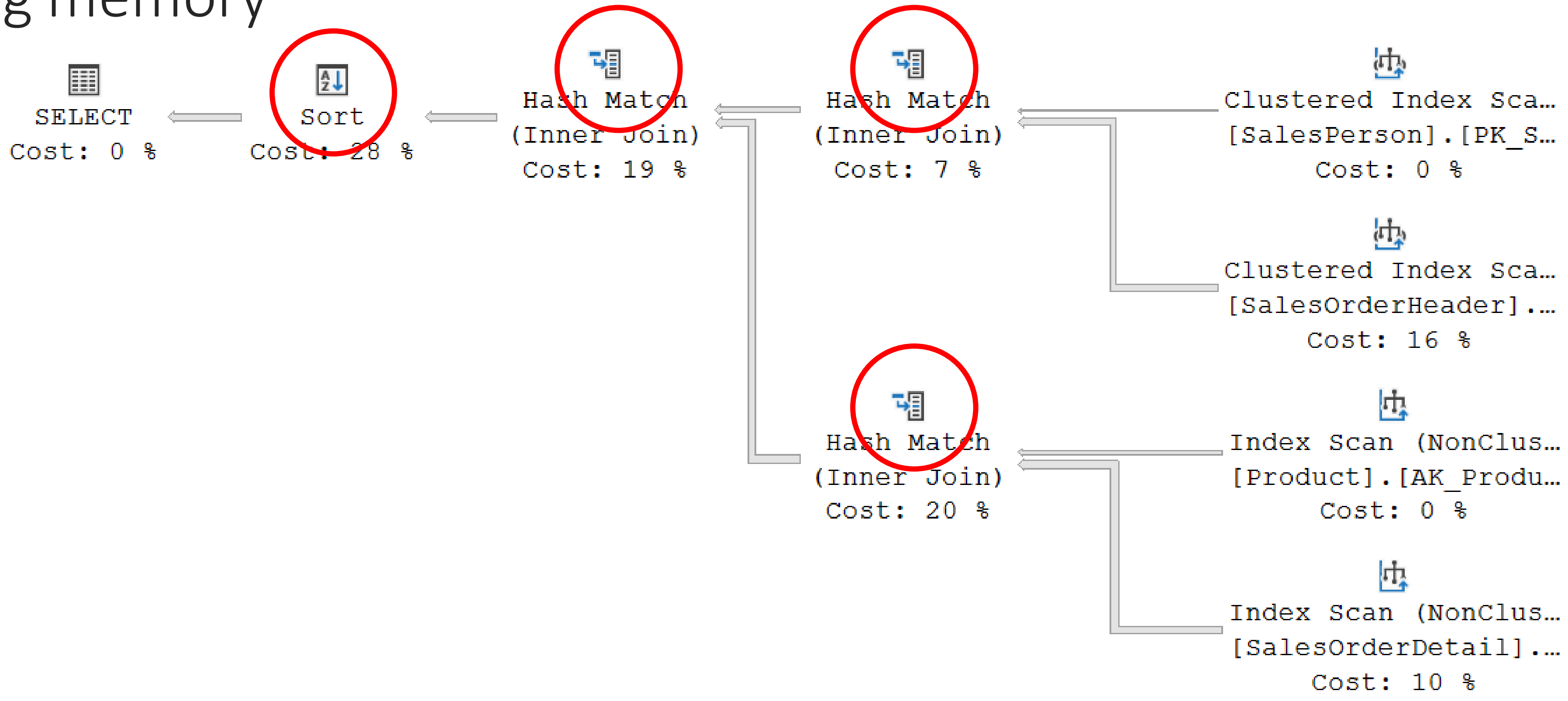
Hash Match

- Multiple Hash Match operators in a single execution plan
  - Reusing memory
  - Hash teams

Hash Match

Multiple ***memory using*** operators in a single execution plan

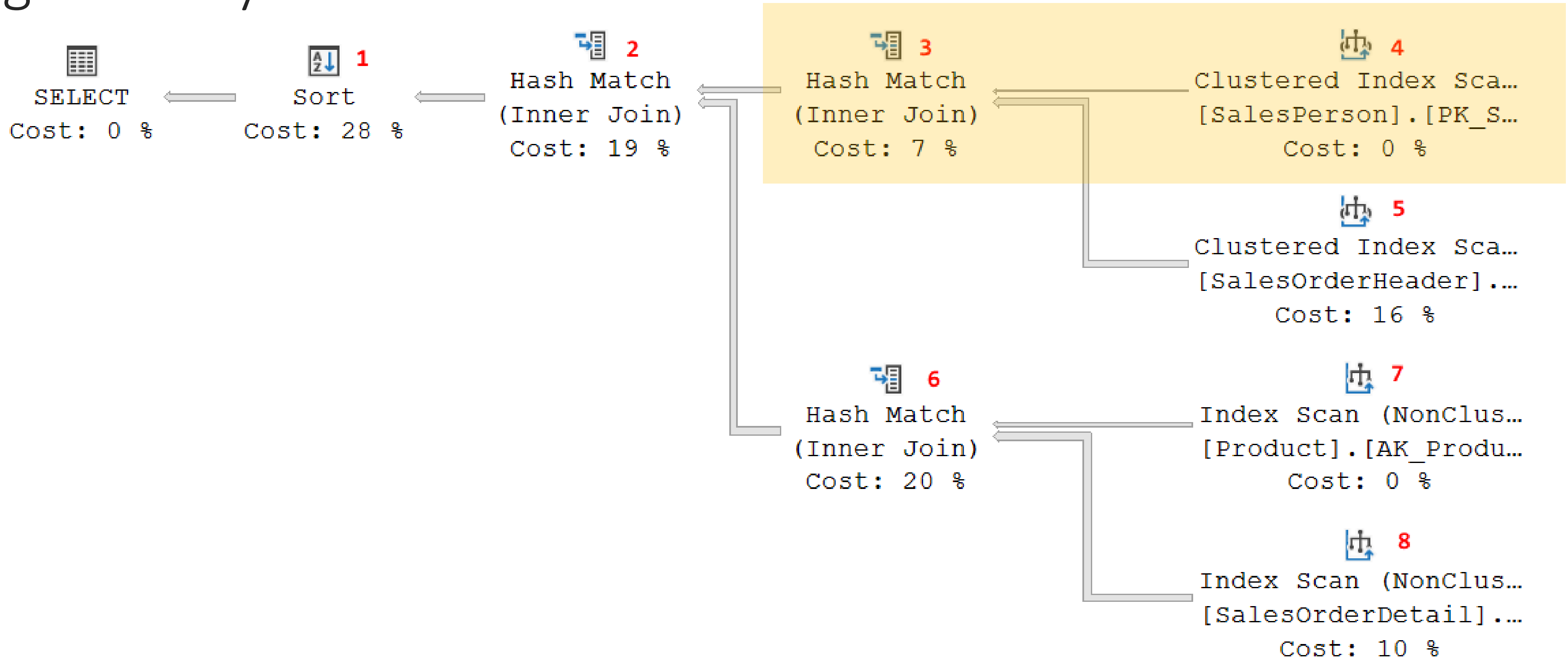
- Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

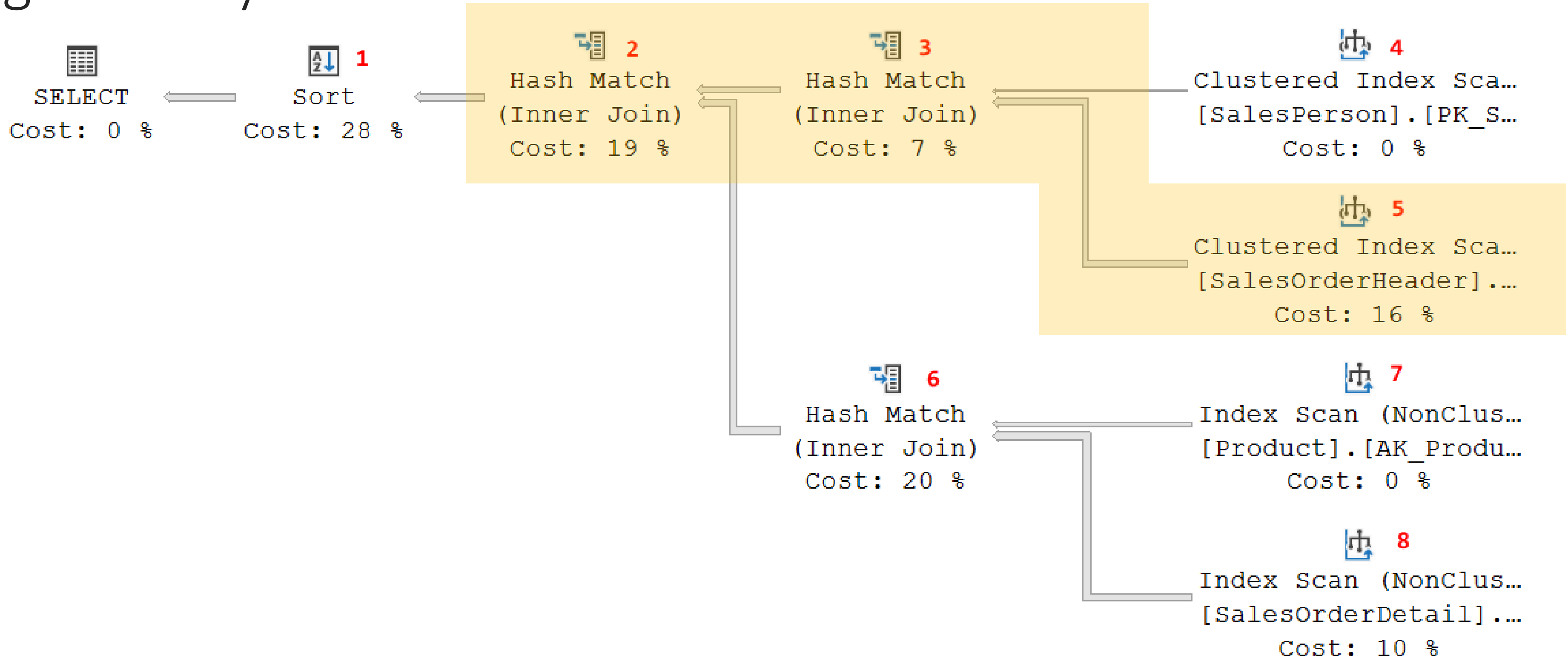
- Reusing memory



SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

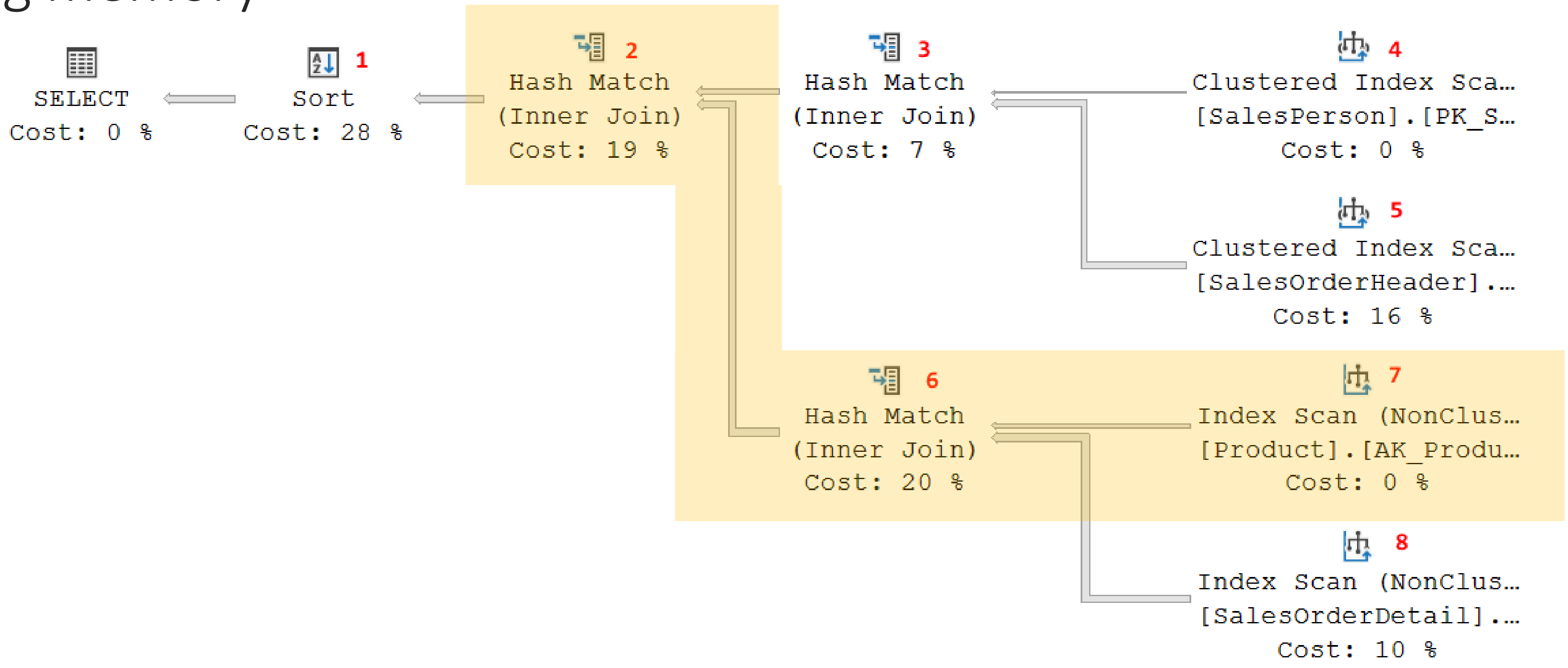
Hash Match

- Multiple ***memory using*** operators in a single execution plan
  - Reusing memory



Hash Match

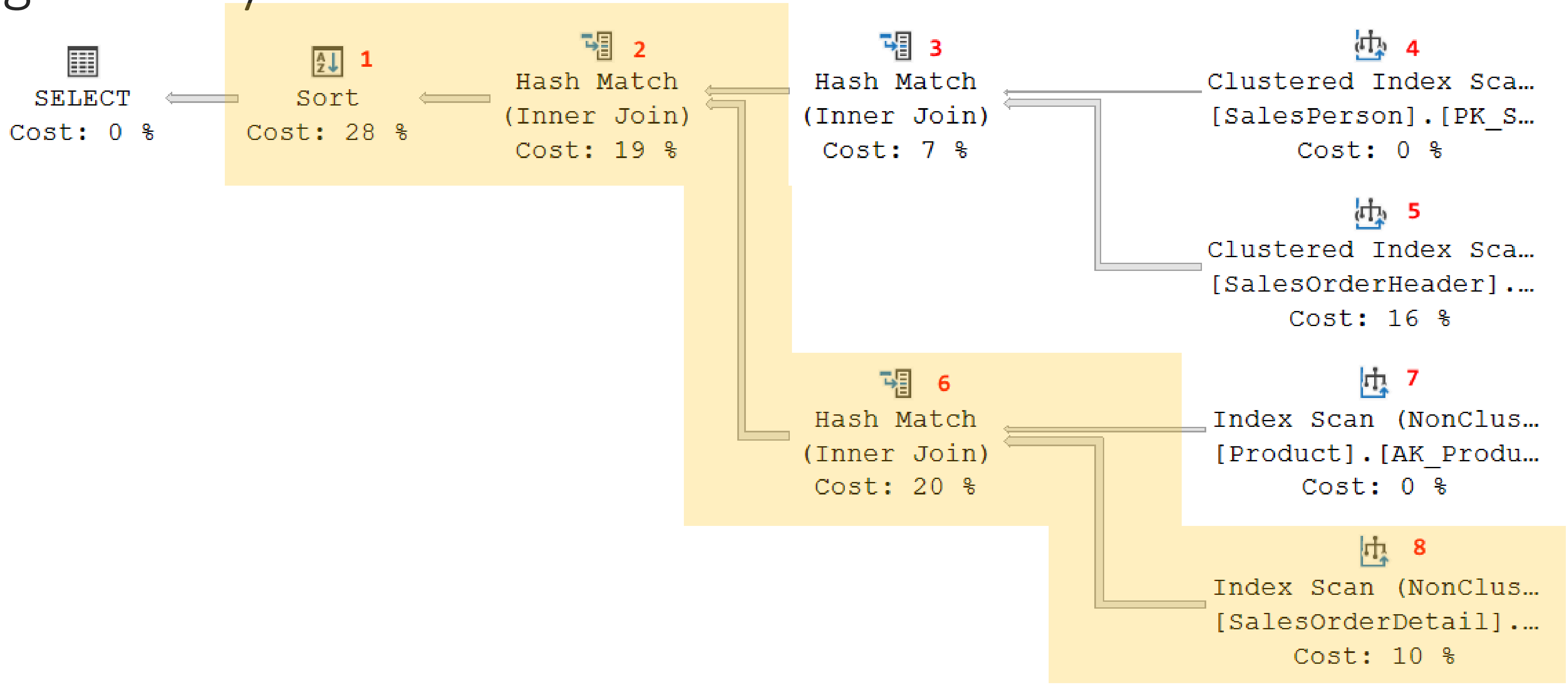
- Multiple ***memory using*** operators in a single execution plan
  - Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

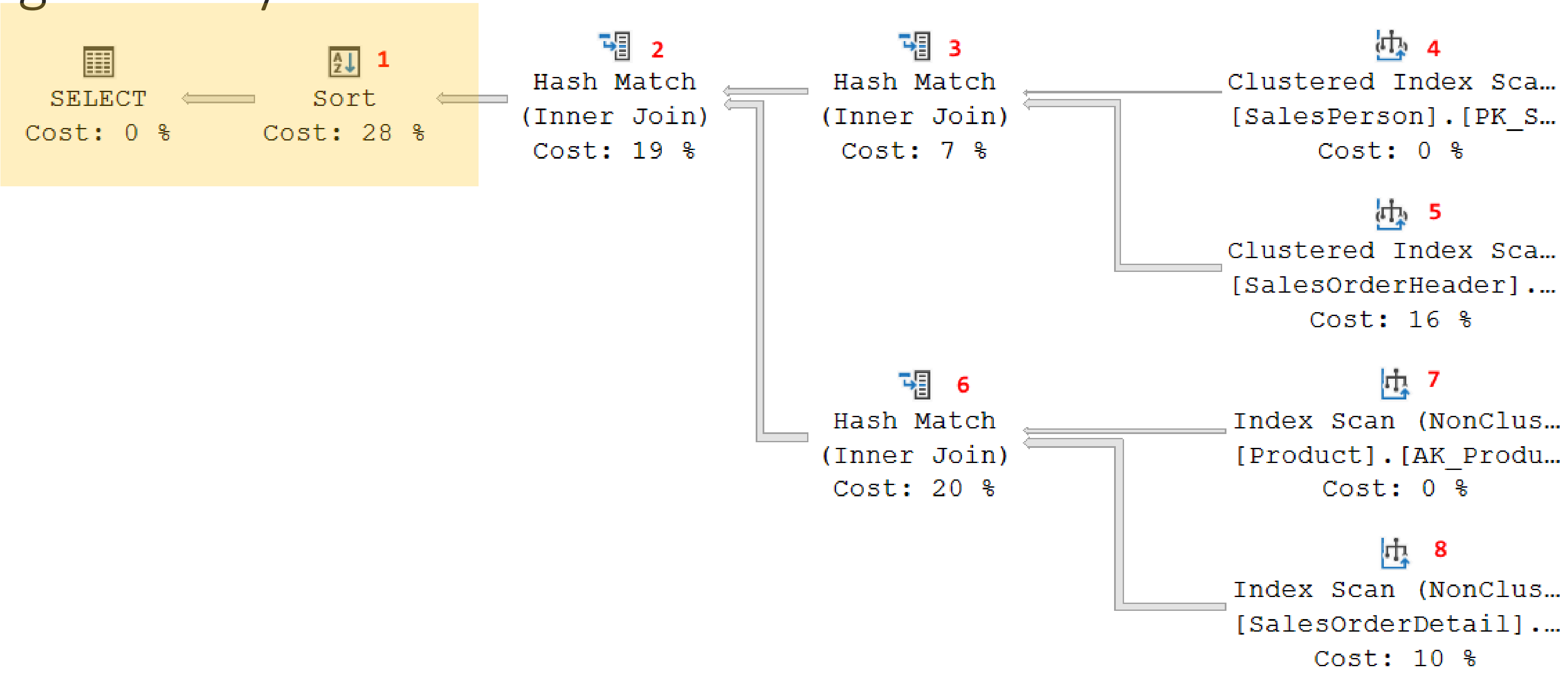
- Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

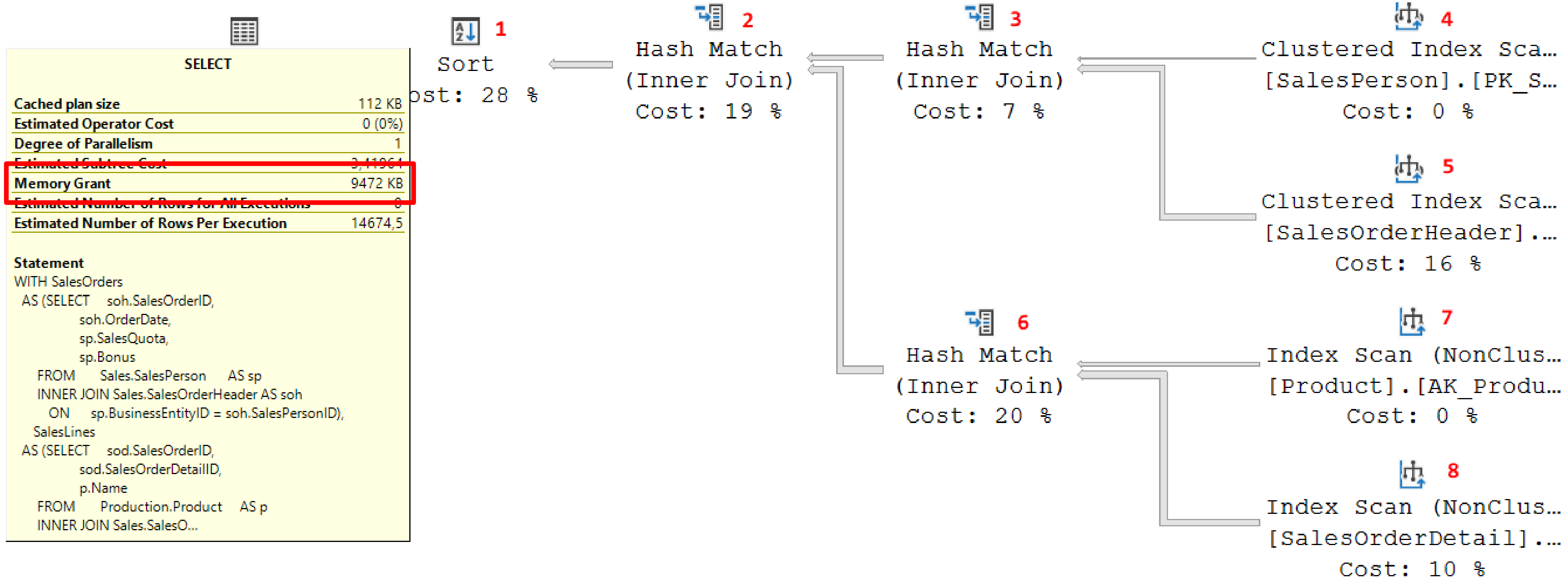
- Reusing memory



Hash Match

Multiple *memory using* operators in a single execution plan

Reusing memory



SELECT

Cached plan size	112 KB
Estimated Operator Cost	0 (0%)
Degree of Parallelism	1
Estimated Subtree Cost	3,41964
Memory Grant	9472 KB
Estimated Number of Rows for All Executions	0
Estimated Number of Rows Per Execution	14674,5

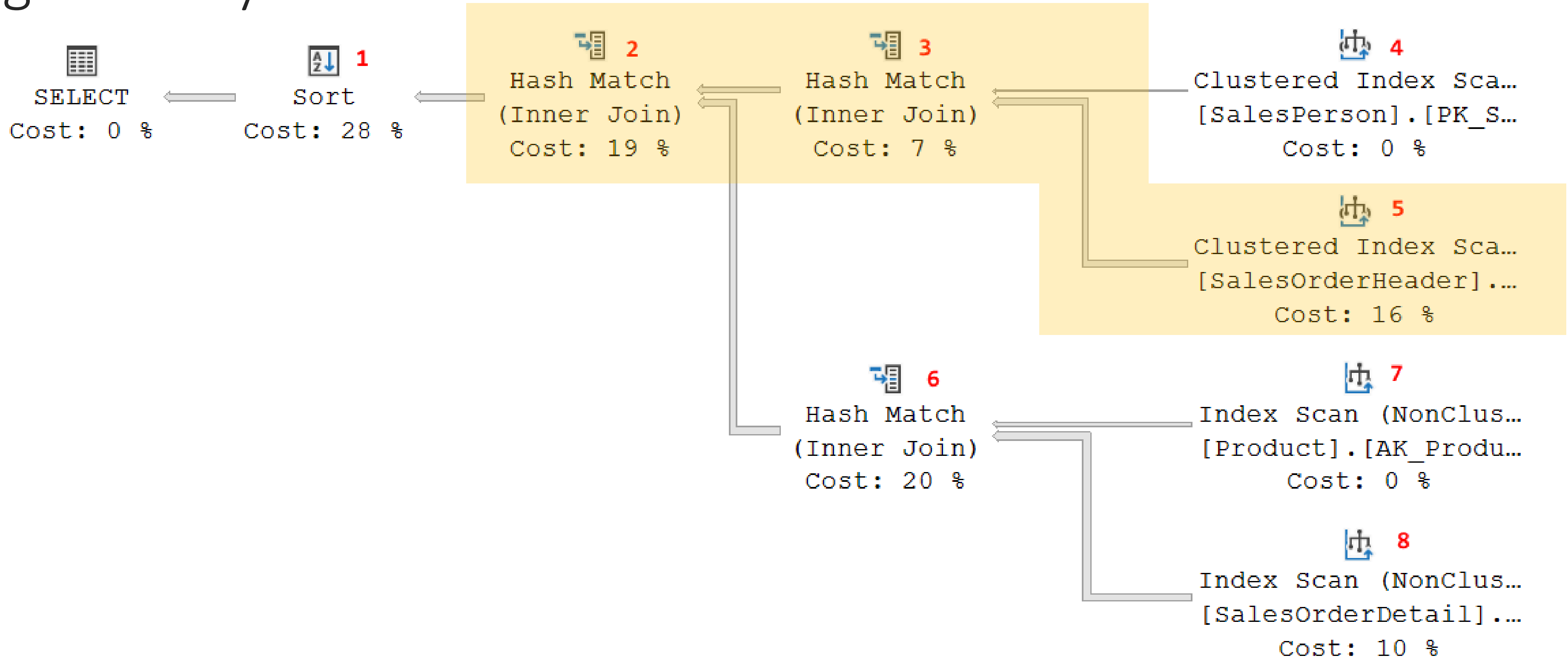
Statement

```
WITH SalesOrders
 AS (SELECT    soh.SalesOrderID,
              soh.OrderDate,
              sp.SalesQuota,
              sp.Bonus
     FROM     Sales.SalesPerson      AS sp
     INNER JOIN Sales.SalesOrderHeader AS soh
        ON     sp.BusinessEntityID = soh.SalesPersonID),
    SalesLines
 AS (SELECT    sod.SalesOrderID,
              sod.SalesOrderDetailID,
              p.Name
     FROM     Production.Product     AS p
     INNER JOIN Sales.SalesO...
```

Hash Match

Multiple *memory using* operators in a single execution plan

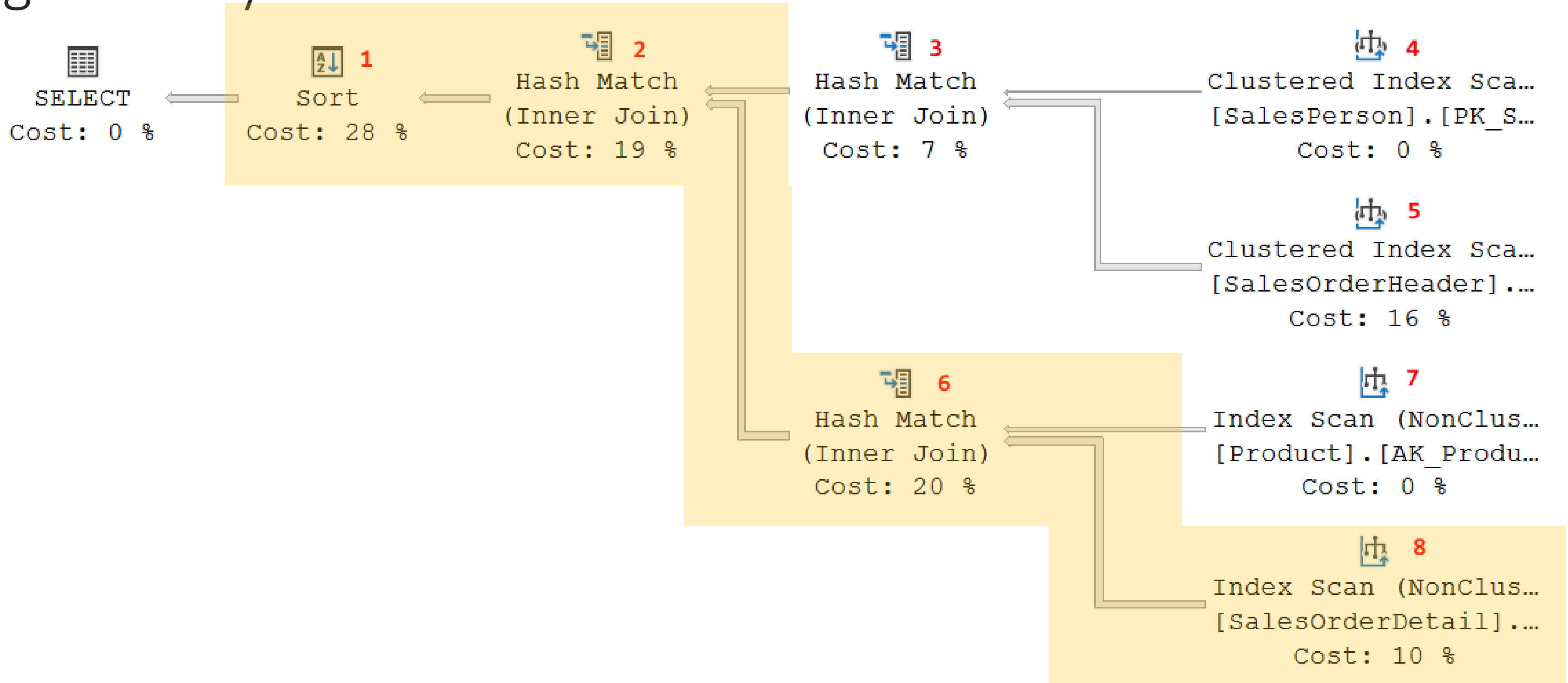
- Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

- Reusing memory

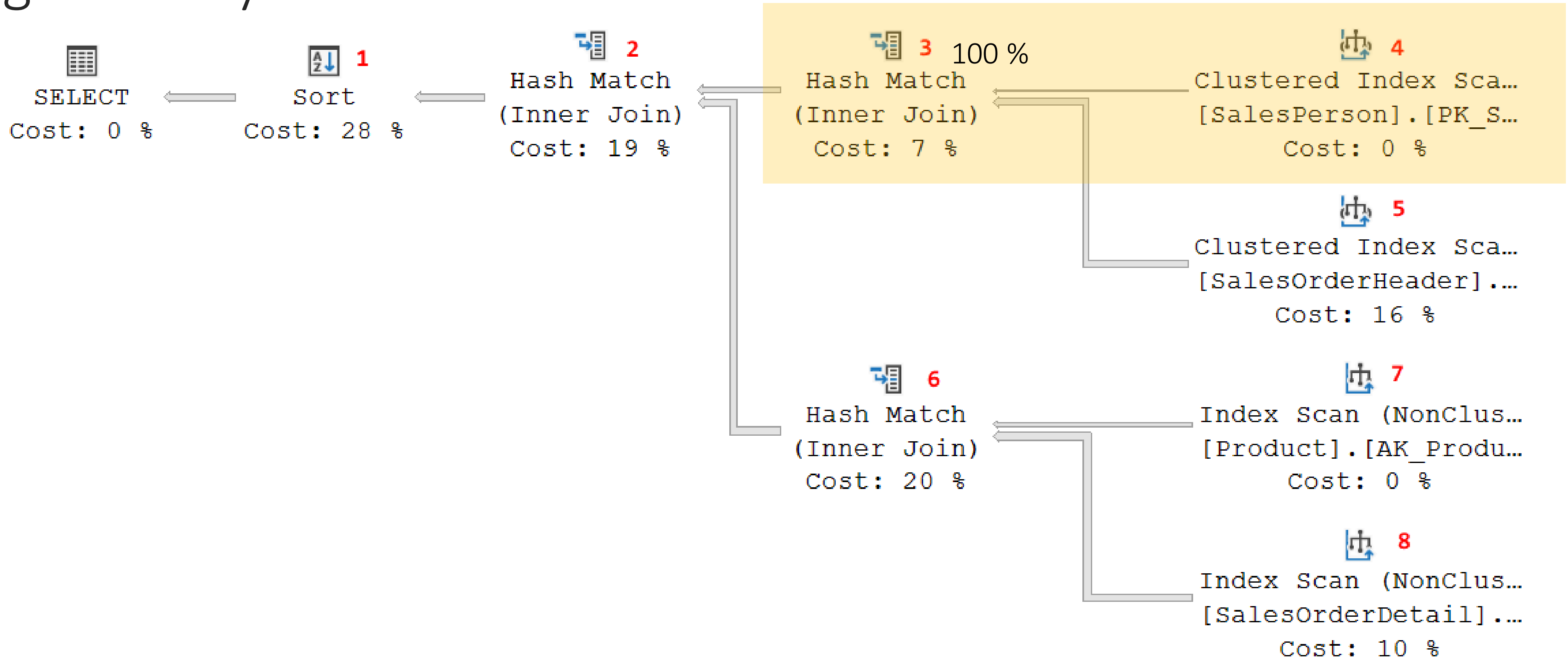


SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

Hash Match

Multiple ***memory using*** operators in a single execution plan

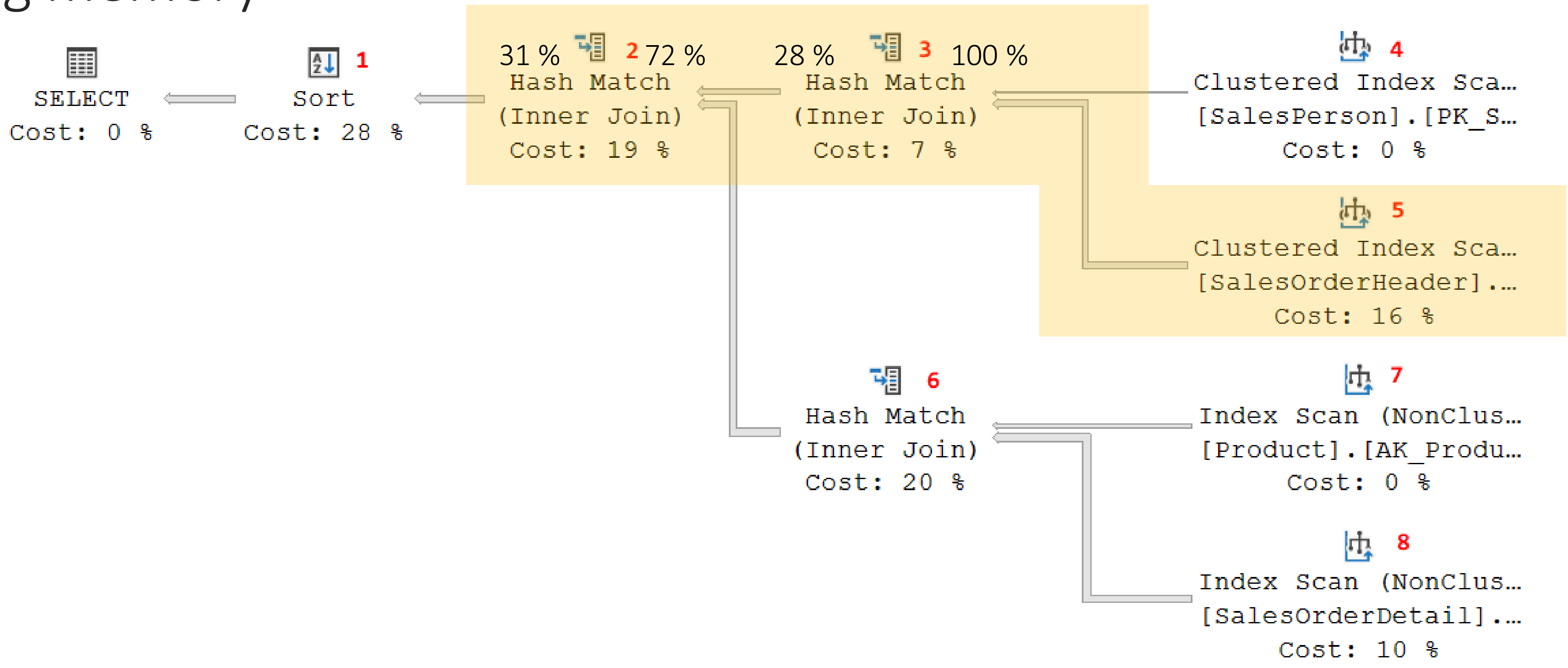
Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

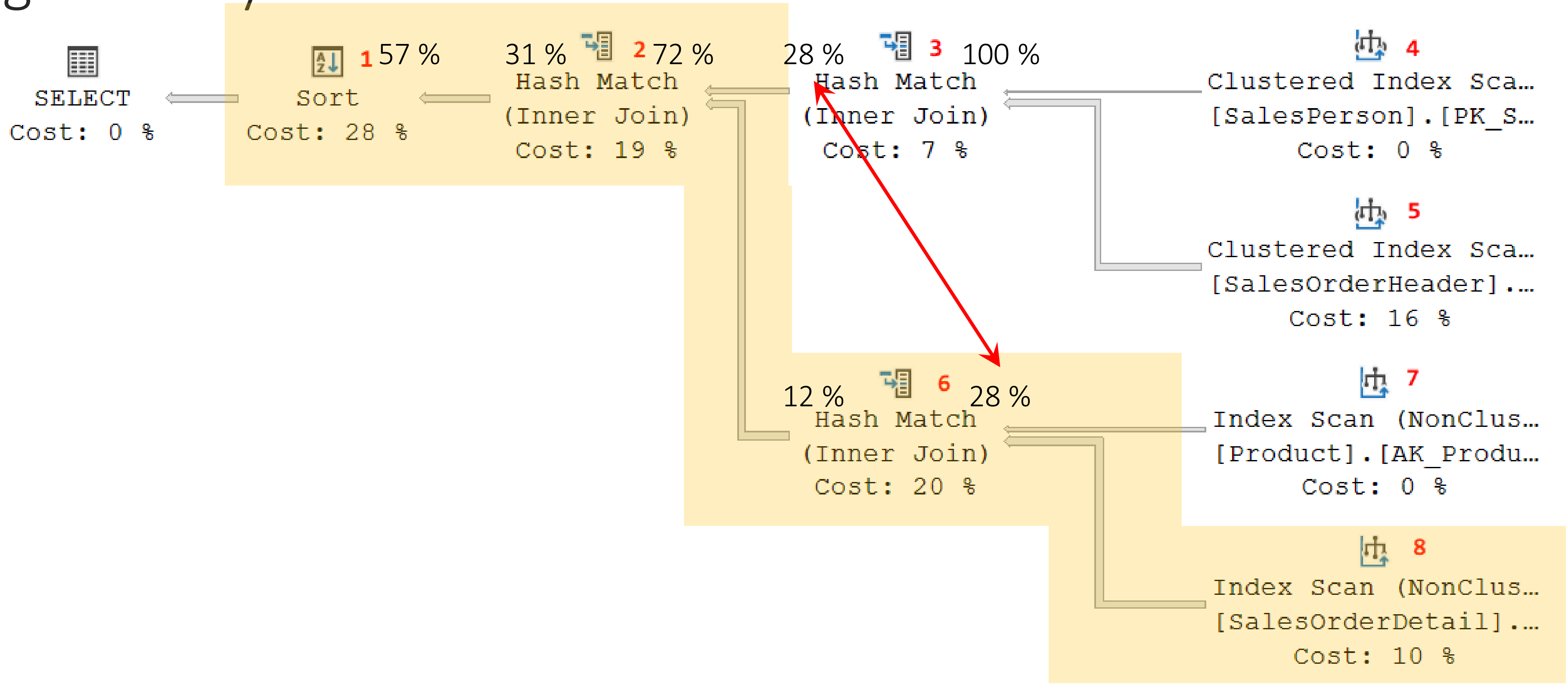
Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

- Reusing memory

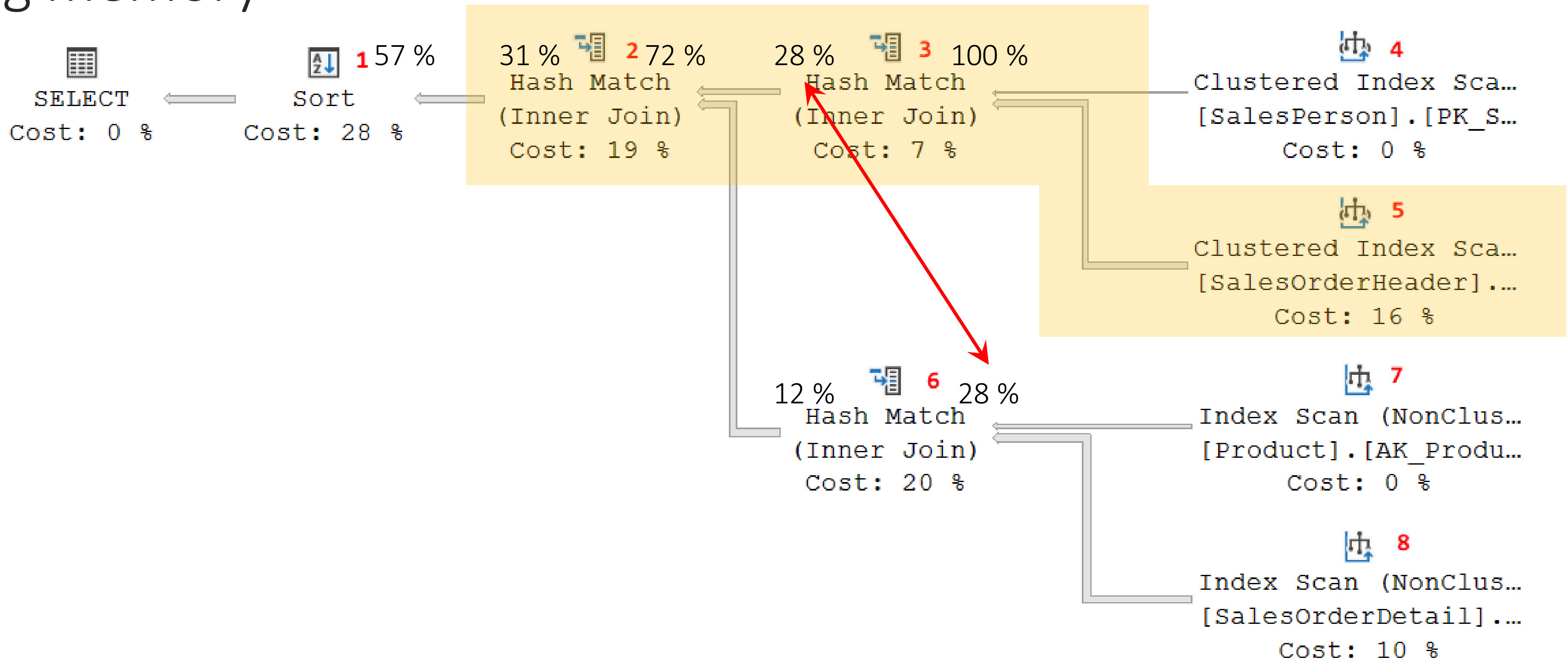


SQLServerFast.com execution plan training, block 3, advanced level, chapter 3: Hash Match (advanced) - (c) Hugo Kornelis

Hash Match

Multiple ***memory using*** operators in a single execution plan

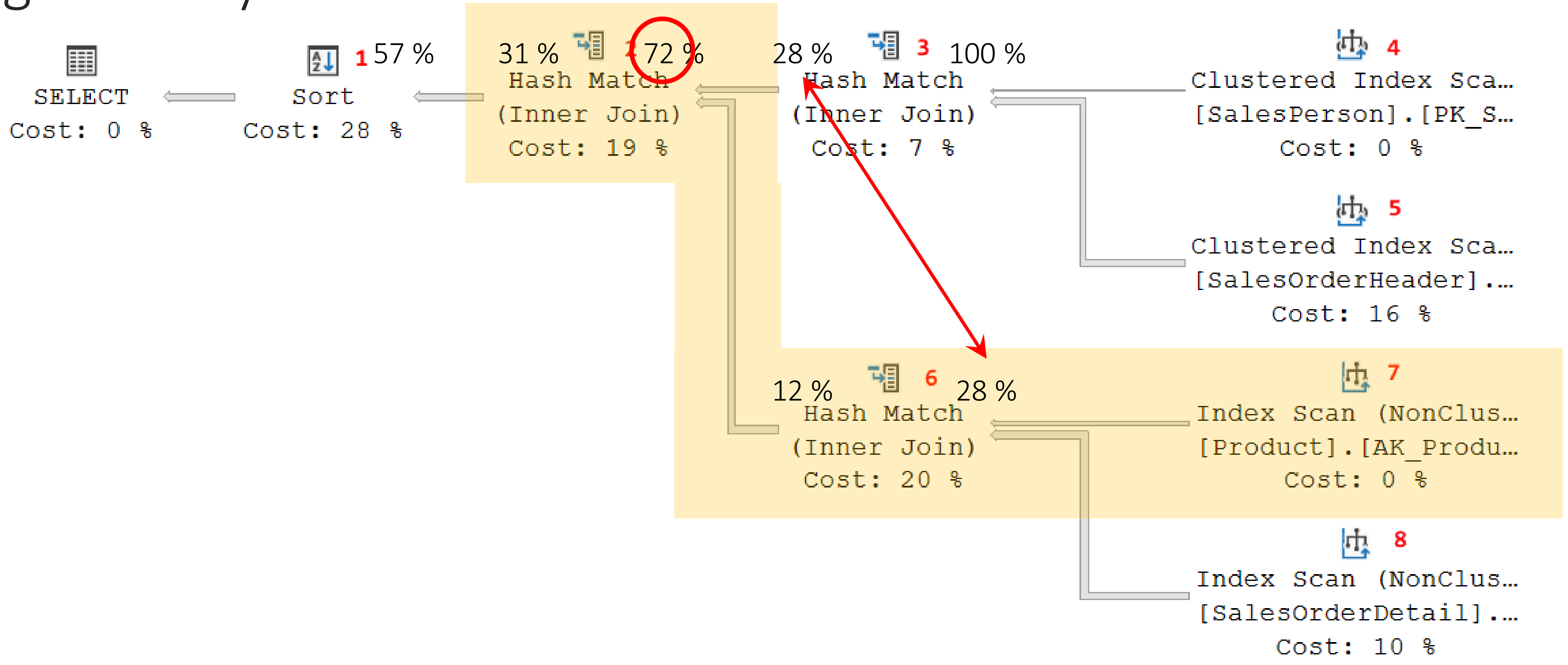
Reusing memory



Hash Match

Multiple ***memory using*** operators in a single execution plan

- Reusing memory



Hash Match

- Multiple ***memory using*** operators in a single execution plan
  - Reusing memory
    - Memory Fractions Input
    - Memory Fractions Output

Hash Match

- Multiple ***memory using*** operators in a single execution plan
  - Reusing memory
  - Hash teams
    - Directly adjacent Hash match operators
    - Using the same hash keys
    - Team manager
      - Mapping of values to buckets
      - Mapping of buckets to partitions
      - Memory management
      - Decisions to spill
        - Entire team spills at once

Summary

- Hash Match (advanced)
  - Logic for all supported join types
  - Hash spills
    - Effect on order
  - Memory fractions
  - Hash teams

Next chapters

Chapter 4: Adaptive Join (advanced)

- Logic for all supported join types
- Spills
- Considerations for choosing Adaptive Join