CS315: Database Systems
Query Processing

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2019-20

Tue, Wed 12:00-13:15

Query

- Query in a high-level language is first parsed and validated

```
select cpi from students where cpi > 9;
```

Query

- Query in a high-level language is first parsed and validated

```
select cpi from students where cpi > 9;
```

- Query planner next outputs an equivalent expression in relational algebra
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - A query order tree is created

Query

- Query in a high-level language is first parsed and validated

```
select cpi from students where cpi > 9;
```

- Query planner next outputs an equivalent expression in relational algebra
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - A query order tree is created
- All equivalent evaluation plans are then generated
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - $\sigma_{cpi>9}(\Pi_{cpi}(students))$

Query

- Query in a high-level language is first parsed and validated

```
select cpi from students where cpi > 9;
```

- Query planner next outputs an equivalent expression in relational algebra
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - A query order tree is created
- All equivalent evaluation plans are then generated
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - $\sigma_{cpi>9}(\Pi_{cpi}(students))$
- Query optimizer next decides on the best evaluation plan among all equivalent plans
  - Uses statistics of data
  - Actual algorithms also influence the choice

Query

- Query in a high-level language is first parsed and validated

  select cpi **from** students **where** cpi > 9;

- Query planner next outputs an equivalent expression in relational algebra
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - A query order tree is created
- All equivalent evaluation plans are then generated
  - $\Pi_{cpi}(\sigma_{cpi>9}(students))$
  - $\sigma_{cpi>9}(\Pi_{cpi}(students))$
- Query optimizer next decides on the best evaluation plan among all equivalent plans
  - Uses statistics of data
  - Actual algorithms also influence the choice
- Query code is finally generated and processed

Query Cost

- Factors that affect the runtime of the query
  - Disk accesses (weakest factor)
  - CPU time
  - Network communication
- In a non-distributed setting, disk access is the most dominant factor

Query Cost

- Factors that affect the runtime of the query
  - Disk accesses
  - CPU time
  - Network communication
- In a non-distributed setting, disk access is the most dominant factor
- It can be estimated as a total of
  - Number of seeks times average seek cost → disk seeks
  - Number of blocks read times average block read cost
  - Number of blocks written times average block write cost
    - Write cost can be more since data is verified

Query Cost

- Factors that affect the runtime of the query
  - Disk accesses
  - CPU time
  - Network communication
- In a non-distributed setting, disk access is the most dominant factor
- It can be estimated as a total of
  - Number of seeks times average seek cost
  - Number of blocks read times average block read cost
  - Number of blocks written times average block write cost
    - Write cost can be more since data is verified
- For s seeks and b block transfers, simply estimated as $s \times t_s + b \times t_b$

  t_s: avg seek time
  t_b: avg block transfer time
- Ignores CPU time and buffer management issues
  - ↳ CPU time
  - ↳ Buffering
  - ↳ Write verification
  - ↳ Network times.

Selection: Linear Search [Relational algebra not SQL]

- LINEAR SEARCH
- *Always* applicable

Selection: Linear Search

- LINEAR SEARCH
- *Always* applicable
- Scan all file blocks and test each record

Selection: Linear Search

- LINEAR SEARCH
- *Always* applicable
- Scan all file blocks and test each record
- Cost for a relation containing b blocks
  - 1 seek
  - b transfers

Selection: Linear Search

Relation r
→ total size: b blocks
→ stored contiguously

- LINEAR SEARCH
- *Always* applicable
- Scan all file blocks and test each record
- Cost for a relation containing b blocks
  - 1 seek
  - b transfers
- (*) If equality on key or unique attribute, then $b/2$ transfers on average

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute [predicate]
- *Equality*

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers
- If one block does not contain all records, add required number of additional transfers to the cost

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers
- If one block does not contain all records, add required number of additional transfers to the cost
- *Greater than*

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers
- If one block does not contain all records, add required number of additional transfers to the cost
- *Greater than*
  - Traverse forward and add required number of additional transfers

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers
- If one block does not contain all records, add required number of additional transfers to the cost
- *Greater than*
  - Traverse forward and add required number of additional transfers
- *Lesser than*

Selection: Binary Search

- BINARY SEARCH
- Applicable for comparison on the ordering attribute
- *Equality*
- Cost for locating the first record is
  - $\lceil \log_2 b \rceil$ seeks
  - $\lceil \log_2 b \rceil$ transfers
- If one block does not contain all records, add required number of additional transfers to the cost
- *Greater than* : 1st block is identified. Everything else is scanned after that.
  - Traverse forward and add required number of additional transfers
- *Lesser than*
  - Scan from beginning till matching record
  - Reduces to *linear search*

Selection: Index Search

- INDEX SEARCH
- B+-tree of height h
- Applicable for comparison on the indexed attribute

Selection: Index Search

- INDEX SEARCH
- B+-tree of height h
- Applicable for comparison on the indexed attribute
- *Equality* using primary index

Selection: Index Search

- INDEX SEARCH
- B+-tree of height h
- Applicable for comparison on the indexed attribute
- *Equality* using primary index
  - $h + 1$ seeks, where h is the height of B+-tree
  - $h + m$ transfers

  where m is the total number of blocks containing matching records
- For key, $m = 1$

Selection: Index Search

- INDEX SEARCH
- B+-tree of height h
- Applicable for comparison on the indexed attribute
- *Equality* using primary index
  - $h + 1$ seeks, where h is the height of B+-tree
  - $h + m$ transfers

  where m is the total number of blocks containing matching records
- For key, $m = 1$
- *Equality* using secondary index

Selection: Index Search

- INDEX SEARCH
- B+-tree of height h [h+1 seeks] [h+1 transfers] + m transfers
- Applicable for comparison on the indexed attribute
- *Equality* using primary index
  - $h + 1$ seeks, where h is the height of B+-tree
  - $h + m$ transfers (h+1): leaf has to be searched

  where m is the total number of blocks containing matching records
- For key, $m = 1$
- *Equality* using secondary index
  - $h + n$ seeks, where h is the height of the index tree
  - $h + n$ transfers

  where n is the total number of matching records, each in a separate block
- For key, $n = 1$

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers

B+ trees are generally doubly linked.

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index
  - 1 seek to the first block
  - n transfers till v is located

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index
  - 1 seek to the first block
  - n transfers till v is located
- *Greater than* ($A \geq v$) using secondary index

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index
  - 1 seek to the first block
  - n transfers till v is located
- *Greater than* ($A \geq v$) using secondary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Use sibling leaf pointers to locate all other matching records resulting in n more seeks and n more transfers

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index
  - 1 seek to the first block
  - n transfers till v is located
- *Greater than* ($A \geq v$) using secondary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Use sibling leaf pointers to locate all other matching records resulting in n more seeks and n more transfers
- *Lesser than* ($A \leq v$) using secondary index

Selection: Index Search (contd.)

- *Greater than* ($A \geq v$) using primary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Scan for the rest of the relation resulting in n more transfers
- *Lesser than* ($A \leq v$) using primary index
  - 1 seek to the first block
  - n transfers till v is located
- *Greater than* ($A \geq v$) using secondary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Use sibling leaf pointers to locate all other matching records resulting in n more seeks and n more transfers
- *Lesser than* ($A \leq v$) using secondary index
  - $h + 1$ seeks and $h + 1$ transfers to locate v
  - Use sibling leaf pointers to locate all other matching records resulting in n more seeks and n more transfers

secondary index: good only for few record selection

Selection on Multiple Attributes

- *Conjunction*: AND

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes

⇒ Everything that matches first attr. is tested to match on other attr.

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)

– K-D tree / R-tree

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan
  - Negation of conjunction
    - May be very inefficient

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan
  - Negation of conjunction
    - May be very inefficient
- *Negation* of equality

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan
  - Negation of conjunction
    - May be very inefficient
- *Negation* of equality
  - Linear scan
  - Index selects leaf pointers for which corresponding records will not be retrieved

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly
- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan
  - Negation of conjunction
    - May be very inefficient
- *Negation* of equality
  - Linear scan
  - Index selects leaf pointers for which corresponding records will not be retrieved
- Negation of comparison

Selection on Multiple Attributes

- *Conjunction*: AND
  - Single index
    - For each record satisfying it, test the other attributes
  - Multiple attribute index
    - Use one likely to produce lesser tuples (e.g., equality)
  - Intersection
    - If some attribute does not have an index, test explicitly

cost: sum of cost of individual index scans + cost of retreiving records

- *Disjunction*: OR
  - Union
    - If some attribute does not have an index, then linear scan
  - Negation of conjunction
    - May be very inefficient
- *Negation* of equality
  - Linear scan
  - Index selects leaf pointers for which corresponding records will not be retrieved
- Negation of comparison is just another comparison

Sorting

Sorting

- Display purposes
- Certain operations such as join can be implemented efficiently on sorted relations

Sorting

- Display purposes
- Certain operations such as join can be implemented efficiently on sorted relations
- Index provides a *logical* sorted view
- Tuples need to be *physically* sorted

Sorting

- Display purposes
- Certain operations such as join can be implemented efficiently on sorted relations
- Index provides a *logical* sorted view
- Tuples need to be *physically* sorted
- When the relation fits in memory, QUICKSORT can be used
- When it does not, external sorting algorithms are used

Sorting

- Display purposes
- Certain operations such as join can be implemented efficiently on sorted relations
- Index provides a *logical* sorted view
- Tuples need to be *physically* sorted
- When the relation fits in memory, QUICKSORT can be used
- When it does not, external sorting algorithms are used
- EXTERNAL MERGESORT or EXTERNAL SORT-MERGE is the most used

External Mergesort

- Assume only *m* blocks can be put into memory
- Size of relation is more than *m* blocks

External Mergesort

- Assume only *m* blocks can be put into memory
- Size of relation is more than *m* blocks
- Create sorted runs
  - Read *m* blocks at a time
  - Sort them in-memory using any algorithm such as quicksort
  - Write them back to disk

External Mergesort

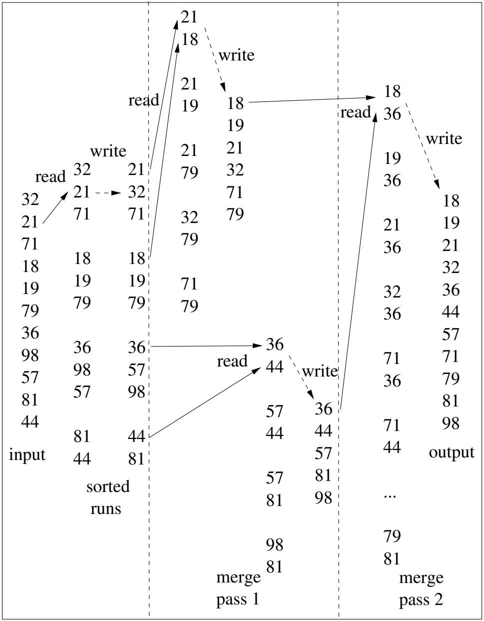
- Assume only m blocks can be put into memory
- Size of relation is more than m blocks
- Create sorted runs
  - Read m blocks at a time
  - Sort them in-memory using any algorithm such as quicksort
  - Write them back to disk
- Merge $m-1$ runs ($(m-1)$-way merge)
  - Read in first block of $m-1$ runs
  - Output the first record to *buffer* block (m-th block in memory)
  - Continue till buffer block is full
  - Write buffer block to disk
  - When a block of a particular run is exhausted, read in the next block of the run

This may not finish in 2nd merge as m-1 runs may not be enough. There may be more than m-1 blocks produced in the 1st stage.

External Mergesort

- Assume only m blocks can be put into memory
- Size of relation is more than m blocks
- Create sorted runs
  - Read m blocks at a time
  - Sort them in-memory using any algorithm such as quicksort
  - Write them back to disk
- Merge $m-1$ runs (($m-1$)-way merge)
  - Read in first block of $m-1$ runs
  - Output the first record to *buffer* block (m-th block in memory)
  - Continue till buffer block is full
  - Write buffer block to disk
  - When a block of a particular run is exhausted, read in the next block of the run
- Continue with ($m-1$)-way merge till the number of sorted runs is *less than m*
- The last ($m-1$)-way merge sorts the relation

Example



read
write
read
write
read
write
read
write
32 21 71 18 19 79 36 98 57 81 44
input
32 21 71
18 19 79
36 98 57
81 44
21 32 71
18 19 79
36 57 98
44 81
sorted runs
21 18
21 19
21 79
32 79
71 79
18 19 21 32 71 79
36 44
57 44
57 81
98 81
merge pass 1
36 44 57 81 98
18 36
19 36
21 36
32 36
71 36
71 44
...
79 81
merge pass 2
18 19 21 32 36 44 57 71 79 81 98
output

Cost of External Mergesort

- Total number of blocks is b $(b > M)$ Memory : M
- Initial number of sorted runs is

Cost of External Mergesort

- Total number of blocks is b
- Initial number of sorted runs is $n = \lceil b/m \rceil$
- In each merge pass, $m - 1$ runs are sorted
- Therefore, total number of merge passes required is

Cost of External Mergesort

m n

- Total number of blocks is b
- Initial number of sorted runs is $n = \lceil b/m \rceil$
- In each merge pass, $m - 1$ runs are sorted
- Therefore, total number of merge passes required is $r = \lceil \log_{m-1} n \rceil$
- During each of these passes and the first pass, all blocks are read and written
- There is an initial pass of creating runs
- Hence, total number of block transfers is

Cost of External Mergesort

- Total number of blocks is b
- Initial number of sorted runs is $n = \lceil b/m \rceil$
- In each merge pass, $m - 1$ runs are sorted
- Therefore, total number of merge passes required is $r = \lceil \log_{m-1} n \rceil$
- During each of these passes and the first pass, all blocks are read and written [b + b]
- There is an initial pass of creating runs
- Hence, total number of block transfers is $2br + 2b$

  $= 2b(r+1)$

Cost of External Mergesort (contd.)

and there is only 1 seek for m blocks

- Initial pass to created sorted runs reads *m* blocks at a time
- Therefore, number of seeks is $2n$ for reading and writing

$\therefore$ for $\left\lceil \frac{b}{M} \right\rceil$ we'll have $(\cancel{1} + 1)\underbrace{\left\lceil \frac{b}{M} \right\rceil}_{n}$ seeks.

read write

Cost of External Mergesort (contd.)

- Initial pass to created sorted runs reads *m* blocks at a time
- Therefore, number of seeks is $2n$ for reading and writing
- During the merge passes, blocks from different runs may not be read consecutively
- Consequently, each read and write for another run may move the disk head away, thereby requiring a seek every time
- Hence, number of seeks for these passes is $2br$
- Therefore, total number of seeks is

For 1 pass : $2b$
For r passes: $2br$

Cost of External Mergesort (contd.)

- Initial pass to created sorted runs reads m blocks at a time
- Therefore, number of seeks is $2n$ for reading and writing
- During the merge passes, blocks from different runs may not be read consecutively
- Consequently, each read and write for another run may move the disk head away, thereby requiring a seek every time
- Hence, number of seeks for these passes is $2br$
- Therefore, total number of seeks is $2n + 2br$

Join

- Different join algorithms
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge join
  - Hash join
- Choice depends on cost estimates

Nested-Loop Join

- Applicable for any kind of join
- For each record $t_r \in r$ and for each record $t_s \in s$, if $t_r \bowtie t_s$ satisfies the join condition, add it to result
- Outer relation r: outer loop; inner relation s: inner loop

relation i:
- n_i: records
- b_i: blocks

$r \bowtie s$

Block transfers.

(1) t_r : b_s

[For some record $t_r \in r$, b_s blocks from s are fetched]

n_r total records, : $\boxed{b_r + n_r \times b_s}$

Seeks

t_r : $1 + 1$

$\boxed{2n_r}$

Block Nested-Loop Join

- Applicable for any kind of join
- *Disk block* aware version of nested-loop
- For each block $l_r \in r$ and for each block $l_s \in s$, test if every record $t_r \in l_r$ and $t_s \in l_s$ satisfies the join condition; if so, add to the result

Block by block join is done

block r   block s

t_r t_s

Cost of Block Nested-Loop Join: Minimal

- Minimal assumption that only 2 blocks fit in memory

Cost of Block Nested-Loop Join: Minimal

- Minimal assumption that only 2 blocks fit in memory
- Block transfers
  - b_s transfers every time a block l_r is read
  - b_r transfers for blocks in r
  - Therefore, total is $b_r + b_r \times b_s$ transfers

Cost of Block Nested-Loop Join: Minimal

- Minimal assumption that only 2 blocks fit in memory
- Block transfers
  - b_s transfers every time a block l_r is read
  - b_r transfers for blocks in r
  - Therefore, total is $b_r + b_r \times b_s$ transfers
- Seeks
  - 1 seek for records in s every time a block l_r is read
  - b_r seeks for records in r
  - Therefore, total is $b_r + b_r = 2b_r$ seeks
- Smaller relation should be

Cost of Block Nested-Loop Join: Minimal

- Minimal assumption that only 2 blocks fit in memory
- Block transfers
  - b_s transfers every time a block l_r is read
  - b_r transfers for blocks in r
  - Therefore, total is $\underbrace{b_r + b_r \times b_s}$ transfers

$\{ b_r + n_r \times b_s$ for NLJ $\}$

- Seeks
  - 1 seek for records in s every time a block l_r is read
  - b_r seeks for records in r
  - Therefore, total is $b_r + b_r = 2b_r$ seeks
- Smaller relation should be outer

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available
- 1 left for inner relation s
- $m - 1$ blocks read at a time from outer relation r

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available
- 1 left for inner relation s
- $m - 1$ blocks read at a time from outer relation r
- Number of such runs is $n = \lceil b_r / (m - 1) \rceil$

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available
- 1 left for inner relation s
- $m - 1$ blocks read at a time from outer relation r
- Number of such runs is $n = \lceil b_r/(m-1) \rceil$
- Block transfers
  - For r: b_r
  - For s: $n \times b_s$

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available
- 1 left for inner relation s
- $m - 1$ blocks read at a time from outer relation r
- Number of such runs is $n = \lceil b_r/(m-1) \rceil$
- Block transfers
  - For r: b_r
  - For s: $n \times b_s$
- Seeks
  - For r: n = no. of runs
  - For s: n = no. of runs
- Smaller relation should be

Cost of Block Nested-Loop Join

- Assume m blocks of memory to be available
- 1 left for inner relation s
- $m - 1$ blocks read at a time from outer relation r
- Number of such runs is $n = \lceil b_r/(m-1) \rceil$
- Block transfers
  - For r: b_r
  - For s: $n \times b_s$
- Seeks
  - For r: n
  - For s: n
- Smaller relation should be outer

Example

- r: 40000 tuples at 200 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block:

Example

b_r

- *r*: 40000 tuples at 200 per block: 200 blocks
- *s*: 25000 tuples at 100 per block: 250 blocks

b_s

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer $\underbrace{(+1 \text{ for inner})}$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$ $\left\lceil \frac{b_r}{m} \right\rceil$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$
- In each run, s requires 1 seek and 250 transfers
- Total is 8 seeks and 2000 transfers

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$
- In each run, s requires 1 seek and 250 transfers
- Total is 8 seeks and 2000 transfers
- r requires 8 seeks and 200 transfers

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$
- In each run, s requires 1 seek and 250 transfers
- Total is 8 seeks and 2000 transfers
- r requires 8 seeks and 200 transfers
- Total is 16 seeks and 2200 transfers

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$
- In each run, s requires 1 seek and 250 transfers
- Total is 8 seeks and 2000 transfers
- r requires 8 seeks and 200 transfers
- Total is 16 seeks and 2200 transfers
- If s made outer,

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer ($+1$ for inner)
- Outer should be r
- Number of runs is $n = \lceil 200/25 \rceil = 8$
- In each run, s requires 1 seek and 250 transfers
- Total is 8 seeks and 2000 transfers
- r requires 8 seeks and 200 transfers
- Total is 16 seeks and 2200 transfers
- If s made outer, 20 seeks and 2250 transfers

Indexed Nested-Loop Join

- Indexed version of the block nested-loop algorithm
- Applicable when inner relation has an index on the joining attribute
- For each block $l_r \in r$ and for each record $t_r \in l_r$, use index on s to locate records $t_s \in s$ that satisfies the join condition

Indexed Nested-Loop Join

- Indexed version of the block nested-loop algorithm
- Applicable when inner relation has an index on the joining attribute
- For each block $l_r \in r$ and for each record $t_r \in l_r$, use index on s to locate records $t_s \in s$ that satisfies the join condition
- Most effective when the join condition is equality

Cost of Indexed Nested-Loop Join

- Assumption is that *m* blocks are available in memory

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time
- Assume cost of searching index on s is c_s seeks and c_t transfers
  - Typically, c_s and c_t are height h of B+-tree

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time
- Assume cost of searching index on s is c_s seeks and c_t transfers
  - Typically, c_s and c_t are height h of B+-tree
- Cost for r is $n = \lceil b_r/(m-1) \rceil$ seeks and b_r transfers
- Therefore, total is $\underbrace{n + n_r \times c_s}$ seeks and $b_r + n_r \times c_t$ transfers

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time
- Assume cost of searching index on s is c_s seeks and c_t transfers
    - Typically, c_s and c_t are height h of B+-tree
- Cost for r is $n = \lceil b_r/(m - 1) \rceil$ seeks and b_r transfers
- Therefore, total is $n + n_r \times c_s$ seeks and $b_r + n_r \times c_t$ transfers
- If index is available on both relations, like block nested-loop, smaller relation should be

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time
- Assume cost of searching index on s is c_s seeks and c_t transfers
  - Typically, c_s and c_t are height h of B+-tree
- Cost for r is $n = \lceil b_r/(m-1) \rceil$ seeks and b_r transfers
- Therefore, total is $n + n_r \times c_s$ seeks and $b_r + n_r \times c_t$ transfers
- If index is available on both relations, like block nested-loop, smaller relation should be outer

Cost of Indexed Nested-Loop Join

- Assumption is that m blocks are available in memory
- $m - 1$ blocks from outer relation r can be read at a time
- Assume cost of searching index on s is c_s seeks and c_t transfers
  - Typically, c_s and c_t are height h of B+-tree
- Cost for r is $n = \lceil b_r/(m-1) \rceil$ seeks and b_r transfers
- Therefore, total is $n + n_r \times c_s$ seeks and $b_r + n_r \times c_t$ transfers
- If index is available on both relations, like block nested-loop, smaller relation should be outer
- Generally, all levels of B+-tree are held in memory except the last
  - Then, cost of index search falls to $c_s = c_t = 1$

Example

- r: 40000 tuples at 200 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s
- Number of runs is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s
- Number of runs is $n = \lceil 250/25 \rceil = 10$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s
- Number of runs is $n = \lceil 250/25 \rceil = 10$
- In each run, r requires 1 seek and 1 transfer
- Total is 25000 seeks and 25000 transfers

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s
- Number of runs is $n = \lceil 250/25 \rceil = 10$
- In each run, r requires $\underbrace{\text{1 seek}}_{cs}$ and $\underbrace{\text{1 transfer}}_{ct}$
- Total is 25000 seeks and 25000 transfers
- s requires $\underbrace{10}_{n}$ seeks and $\underbrace{250}_{bf}$ transfers

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 25$ for outer and enough for inner
- Outer should be s
- Number of runs is $n = \lceil 250/25 \rceil = 10$
- In each run, r requires 1 seek and 1 transfer
- Total is 25000 seeks and 25000 transfers
- s requires 10 seeks and 250 transfers
- Total is 25010 seeks and 25250 transfers

Merge Join or Sort-Merge Join [only for equality join]

- Applicable only when the join condition is *equality*
- Relations need to be sorted according to the joining attribute

Merge Join or Sort-Merge Join

- Applicable only when the join condition is *equality*
- Relations need to be sorted according to the joining attribute
- Proceed in sorted order on two relations
- If records match, output; otherwise, advance to next record
- Join step is similar to merge step in mergesort

Merge Join or Sort-Merge Join

- Applicable only when the join condition is *equality*
- Relations need to be sorted according to the joining attribute
- Proceed in sorted order on two relations
- If records match, output; otherwise, advance to next record
- Join step is similar to merge step in mergesort
- If relations are not sorted, secondary index on attributes can be used
- HYBRID MERGE JOIN algorithm merges sorted records in one relation with B+-tree leaves of other relation

* one relation is sorted
* other is indexed (B + tree)

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once
- However, blocks may be read in an interleaved manner

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once
- However, blocks may be read in an interleaved manner
- If only 2 blocks of memory are available (1 for each), cost is

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once
- However, blocks may be read in an interleaved manner
- If only 2 blocks of memory are available (1 for each), cost is
  - $b_r + b_s$ transfers
  - $b_r + b_s$ seeks

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once
- However, blocks may be read in an interleaved manner
- If only 2 blocks of memory are available (1 for each), cost is
  - $b_r + b_s$ transfers
  - $b_r + b_s$ seeks
- If $m = m'/2$ blocks of memory are available for each relation, cost is

Cost of Merge Join

- Each record is read only once
- Consequently, each block is read only once
- However, blocks may be read in an interleaved manner
- If only 2 blocks of memory are available (1 for each), cost is
  - $b_r + b_s$ transfers
  - $b_r + b_s$ seeks
- If $m = m'/2$ blocks of memory are available for each relation, cost is $\lceil b_r/m \rceil + \lceil b_s/m \rceil$ seeks and $b_r + b_s$ transfers

Example

- r: 40000 tuples at 200 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each
- Number of seeks for r is

$$= \left\lceil \frac{200}{m} \right\rceil + \left\lceil \frac{250}{m} \right\rceil \text{ seeks}$$

$$200 + 250 \text{ transfers}$$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each
- Number of seeks for r is $\lceil 250/13 \rceil = 20$
- Number of seeks for s is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each
- Number of seeks for r is $\lceil 250/13 \rceil = 20$
- Number of seeks for s is $\lceil 200/13 \rceil = 16$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each
- Number of seeks for r is $\lceil 250/13 \rceil = 20$
- Number of seeks for s is $\lceil 200/13 \rceil = 16$
- Total is

Example

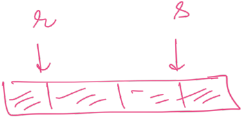
- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is $m = 13$ for each
- Number of seeks for r is $\lceil 250/13 \rceil = 20$
- Number of seeks for s is $\lceil 200/13 \rceil = 16$
- Total is 36 seeks and 450 transfers

Hash Join {only on equality}

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared

t_r (h)

t_s



t_r & t_s must be hashed to the same position (equality join)

Partition hash join

0, ... , n − 1

H_{r_i} partition H_{s_i}

$h(t_r) = i$ $h(t_s) = i$

read in-memory hash for H_{s_i}

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \ldots, n-1\}$

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \ldots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \ldots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
  - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
  - A finer-grained in-memory hash may be constructed on $P_i(s)$

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \to \{0, \ldots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
  - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
  - A finer-grained in-memory hash may be constructed on $P_i(s)$
  - Partition $P_i(r)$ is read block by block: $t_r \in P_i(r) \iff h(t_r) = i$
  - For each tuple in $P_i(r)$, in-memory hash of $P_i(s)$ is searched

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \dots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
  - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
  - A finer-grained in-memory hash may be constructed on $P_i(s)$
  - Partition $P_i(r)$ is read block by block: $t_r \in P_i(r) \iff h(t_r) = i$
  - For each tuple in $P_i(r)$, in-memory hash of $P_i(s)$ is searched
  - After partition i is done, new partitions are loaded

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \dots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
  - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
  - A finer-grained in-memory hash may be constructed on $P_i(s)$
  - Partition $P_i(r)$ is read block by block: $t_r \in P_i(r) \iff h(t_r) = i$
  - For each tuple in $P_i(r)$, in-memory hash of $P_i(s)$ is searched
  - After partition i is done, new partitions are loaded
- s is called build input
- r is called probe input

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \ldots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
    - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
    - A finer-grained in-memory hash may be constructed on $P_i(s)$
    - Partition $P_i(r)$ is read block by block: $t_r \in P_i(r) \iff h(t_r) = i$
    - For each tuple in $P_i(r)$, in-memory hash of $P_i(s)$ is searched
    - After partition i is done, new partitions are loaded
- s is called build input → in-memory hash
- r is called probe input → block-by-block
- Each partition of build relation must be stored in memory
- Smaller relation should be build relation

Hash Join

- Applicable only when the join condition is *equality*
- If record t_r and t_s match, they must hash to same value, and thus, only partitions with the same hash value need to be compared
- PARTITION HASH JOIN
- Hash function h to partition records of *both* relations into n partitions: $h : t \in (r \cup s) \rightarrow \{0, \ldots, n-1\}$
- Partitioning (or building) phase partitions the relations and writes them to disk
- Matching (or probing) phase
  - Partition $P_i(s)$ is read into memory: $t_s \in P_i(s) \iff h(t_s) = i$
  - A finer-grained in-memory hash may be constructed on $P_i(s)$
  - Partition $P_i(r)$ is read block by block: $t_r \in P_i(r) \iff h(t_r) = i$
  - For each tuple in $P_i(r)$, in-memory hash of $P_i(s)$ is searched
  - After partition i is done, new partitions are loaded
- s is called build input
- r is called probe input
- Each partition of build relation must be stored in memory
- Smaller relation should be build

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks

$n \geq \left\lceil \frac{b_s}{M} \right\rceil$ M: memory size

$M > n$

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks
- Assume an available memory of m blocks for build

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks
- Assume an available memory of m blocks for build
- If each partition fits, then $m > b_s/n$ or $n > b_s/m$

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks
- Assume an available memory of m blocks for build
- If each partition fits, then $m > b_s/n$ or $n > b_s/m$
- During partitioning, at least 1 block of each partition should fit in memory
- Thus, $m > n$

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks
- Assume an available memory of m blocks for build
- If each partition fits, then $m > b_s/n$ or $n > b_s/m$
- During partitioning, at least 1 block of each partition should fit in memory
- Thus, $m > n$
- Combining,

Single Pass

- Each of the n partitions of build input s should fit into memory
- If build relation has b_s blocks, each partition has roughly b_s/n blocks
- Assume an available memory of m blocks for build
- If each partition fits, then $m > b_s/n$ or $n > b_s/m$
- During partitioning, at least 1 block of each partition should fit in memory
- Thus, $m > n$
- Combining, $m > n > b_s/m$ or, $\boxed{m > \sqrt{b_s}}$
- If not so, then partitioning *cannot* be done in one pass

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed
- Initially, number of partitions n_s is chosen to be at most m
- However, each partition now does *not* fit into memory
- Thus, each partition is read and re-partitioned till each smaller partition fits into memory

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed
- Initially, number of partitions n_s is chosen to be at most m
- However, each partition now does *not* fit into memory
- Thus, each partition is read and re-partitioned till each smaller partition fits into memory
- Number of passes is

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed
- Initially, number of partitions n_s is chosen to be at most m
- However, each partition now does *not* fit into memory
- Thus, each partition is read and re-partitioned till each smaller partition fits into memory
- Number of passes is $p = \lfloor \log_m b_s \rfloor$

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed
- Initially, number of partitions n_s is chosen to be at most m
- However, each partition now does *not* fit into memory
- Thus, each partition is read and re-partitioned till each smaller partition fits into memory
- Number of passes is $p = \lfloor \log_m b_s \rfloor$
- HYBRID HASH JOIN when more memory is available

Recursive Partitioning

- If $m < \sqrt{b_s}$, then partitioning *cannot* be done in one pass
- Recursive partitioning is employed
- Initially, number of partitions n_s is chosen to be at most m
- However, each partition now does *not* fit into memory
- Thus, each partition is read and re-partitioned till each smaller partition fits into memory
- Number of passes is $p = \lfloor \log_m b_s \rfloor$
- HYBRID HASH JOIN when more memory is available
- Entire build relation s can be in memory
- Retain the first partition of build relation in memory

Example

- Suppose, build input has $b_s = 120$ blocks

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks
  - Partitioning produces $120/12 = \underbrace{10 \text{ blocks that fit}}$
  - Thus, only 1 pass is required

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks
  - Partitioning produces $120/12 = 10$ blocks that fit
  - Thus, only 1 pass is required
- Available memory is $m = 10$ blocks

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks
  - Partitioning produces $120/12 = 10$ blocks that fit
  - Thus, only 1 pass is required
- Available memory is $m = 10$ blocks
  - Partitioning produces $120/10 = 12$ blocks that do not fit
  - Thus, 12 blocks are partitioned again to $12/10 = 2$ blocks each
  - This now fits
  - Thus, 2 passes are required $\rightarrow \log_m b_s$

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks
  - Partitioning produces $120/12 = 10$ blocks that fit
  - Thus, only 1 pass is required
- Available memory is $m = 10$ blocks
  - Partitioning produces $120/10 = 12$ blocks that do not fit
  - Thus, 12 blocks are partitioned again to $12/10 = 2$ blocks each
  - This now fits
  - Thus, 2 passes are required
- Available memory is $m = 4$ blocks

Example

- Suppose, build input has $b_s = 120$ blocks
- Available memory is $m = 12$ blocks
  - Partitioning produces $120/12 = 10$ blocks that fit
  - Thus, only 1 pass is required
- Available memory is $m = 10$ blocks
  - Partitioning produces $120/10 = 12$ blocks that do not fit
  - Thus, 12 blocks are partitioned again to $12/10 = 2$ blocks each
  - This now fits
  - Thus, 2 passes are required
- Available memory is $m = 4$ blocks
  - Partitioning produces $\underbrace{120/4}_{1} = 30$ and then $\underbrace{30/4}_{2} = 8$ and then 3 $\underline{8/4} = 2$ blocks
  - Thus, 3 passes are required

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most $b_i + n$ as each of n partitions may be partially full
- Writing them back requires

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most $b_i + n$ as each of n partitions may be partially full
- Writing them back requires $(b_i + n)$ transfers
- Reading them again during matching requires

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most $b_i + n$ as each of n partitions may be partially full
- Writing them back requires $(b_i + n)$ transfers
- Reading them again during matching requires $(b_i + n)$ transfers
- Therefore, total number of transfers is $3(b_r + b_s) + 4n$

for i:
$\boxed{3b_i + 2n}$
transfers

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most $b_i + n$ as each of n partitions may be partially full
- Writing them back requires $(b_i + n)$ transfers
- Reading them again during matching requires $(b_i + n)$ transfers
- Therefore, total number of transfers is $3(b_r + b_s) + 4n$
- Assume memory buffer of m blocks
- Partitioning requires

Cost of Hash Join

- Initial reading of a relation i requires b_i transfers
- Total number of blocks after partitioning is at most $b_i + n$ as each of n partitions may be partially full
- Writing them back requires $(b_i + n)$ transfers
- Reading them again during matching requires $(b_i + n)$ transfers
- Therefore, total number of transfers is $3(b_r + b_s) + 4n$
- Assume memory buffer of m blocks
- Partitioning requires $k_i = \lceil b_i/m \rceil$ seeks for reading and $k_i' = \lceil (b_i + n)/m \rceil$ seeks for writing
- Reading n partitions during matching requires n seeks per relation
- Therefore, total number of seeks is $k_r + k_s + k_r' + k_s' + 2n$

Reading seeks: $\overset{r}{n} + \overset{s}{n} = 2n$

For i: seeks. $\left\lceil \frac{2b_i + n}{m} \right\rceil$: (Partitioning + writing)

Total: $\left\lceil \frac{2(b_r + b_s + n)}{m} \right\rceil + 2n$

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is $2p(b_r + n + b_s + n)$

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is $2p(b_r + n + b_s + n)$
- Number of seeks during partitioning is

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is $2p(b_r + n + b_s + n)$
- Number of seeks during partitioning is $2p(k'_r + k'_s)$

$\left\lceil \frac{b_r}{m} \right\rceil \quad \left\lceil \frac{b_s}{m} \right\rceil$

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is $2p(b_r + n + b_s + n)$
- Number of seeks during partitioning is $2p(k_r' + k_s')$
- Matching requires

Cost of Recursive Partitioning

- Number of passes in recursive partitioning is $p = \lfloor \log_m b_s \rfloor$
- During partitioning, total number of block transfers is $2p(b_r + n + b_s + n)$
- Number of seeks during partitioning is $2p(k'_r + k'_s)$
- Matching requires $(b_r + n + b_s + n)$ transfers and $n + n$ seeks

Example

- r: 40000 tuples at 200 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires

Example

Doubt

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$
- Matching phase requires reading all blocks of s and r:

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$
- Matching phase requires reading all blocks of s and r: $215 + 265 = 480$
- Number of seeks in matching phase is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$
- Matching phase requires reading all blocks of s and r: $215 + 265 = 480$
- Number of seeks in matching phase is $15 + 15 = 30$

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$
- Matching phase requires reading all blocks of s and r: $215 + 265 = 480$
- Number of seeks in matching phase is $15 + 15 = 30$
- Total is

Example

- r: 40000 tuples at 200 per block: 200 blocks
- s: 25000 tuples at 100 per block: 250 blocks
- Available memory is 15 for partitions and $m = 10$ for buffering
- Number of partitions is $n = 15$
- Size of each partition of s is $\lceil 200/15 \rceil = 14$
- Size of each partition of r is $\lceil 250/15 \rceil = 17$
- Build input should be s
- Partitioning s requires $200 + (200 + 15) = 415$ transfers
- Number of seeks for s is $\lceil 200/10 \rceil + \lceil 215/10 \rceil = 42$
- Partitioning r requires 1 pass since it is probe input
- Number of transfers for r is $250 + (250 + 15) = 515$
- Number of seeks for r is $\lceil 250/10 \rceil + \lceil 265/10 \rceil = 52$
- Matching phase requires reading all blocks of s and r: $215 + 265 = 480$
- Number of seeks in matching phase is $15 + 15 = 30$
- Total is $15 + 15 + 42 + 52 = 124$ seeks and $415 + 515 + 480 = 1410$ transfers

Other Operations

- Complex join conditions

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning

left - outer join

$r ⟕ S$

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if $r ⟕ s$, r should be the ⇒ probe

s : should be build relation

Every tuple from r is probed. If it matches, we output

Even if it doesn't, the r side is output

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if r ⟕ s, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If r ⟗ s, use both techniques

Hash join is generally not used for full-join

What can be done is:

Use markers on S.
(build relation)

r → S

1 } Match

0
0
0 } Don't match

At the end a pass is taken to look for 0s and output (clumsy)

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if $r ⟕ s$, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If $r ⟗ s$, use both techniques
- Set operations

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if *r* ⟕ *s*, *r* should be the probe relation
  - If *r* is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If *r* ⟗ *s*, use both techniques
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if $r ⟕ s$, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If $r ⟗ s$, use both techniques
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Grouping and aggregation

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if r ⟕ s, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If r ⟗ s, use both techniques
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Grouping and aggregation
  - Use hashing to organize into groups; then apply aggregation

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if $r ⟕ s$, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If $r ⟗ s$, use both techniques
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Grouping and aggregation
  - Use hashing to organize into groups; then apply aggregation
- Duplicate detection and elimination

Other Operations

- Complex join conditions
  - Block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of index or merge or hash join may be used
- Outer join
  - Block nested-loop algorithm requires almost no modification
  - For merge join, select all (non-joining) records while scanning
  - For hash join, if r ⟕ s, r should be the probe relation
  - If r is the build relation, keep track of which records in hash index have been used; output all non-used records
  - If r ⟗ s, use both techniques
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Grouping and aggregation
  - Use hashing to organize into groups; then apply aggregation
- Duplicate detection and elimination
  - Use hashing or sorting