CS315: Principles of 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, 2013-14

Tue, Fri 1530-1700 at CS101

Query

- When a database query in a high-level language comes, first it is parsed and validated
- Next, a query planner outputs an equivalent expression in relational algebra
  - $\Pi_{bal}(\sigma_{bal<1000}(account))$
  - A query order tree is created
- Then, all equivalent evaluation plans are generated
  - $\Pi_{bal}(\sigma_{bal<1000}(account))$
  - $\sigma_{bal<1000}(\Pi_{bal}(account))$
- A query optimizer 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
  - CPU time
  - Network communication
- Disk access is generally 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 is more since data is verified
- For s seeks and b block transfers, simply estimated as $s \times t_s + b \times t_b$
- Ignores CPU time and buffer management issues

Selection

- LINEAR SEARCH
- *Always* applicable

Selection

- Linear search
- *Always* applicable
- Scan all file blocks and test each record

Selection

- 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, then $b/2$ transfers on average

Selection (contd.)

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

Selection (contd.)

- 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 (contd.)

- 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*
  - Add required number of additional transfers
- *Lesser than*

Selection (contd.)

- 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*
  - Add required number of additional transfers
- *Lesser than*
  - Scan from beginning till matching record

Selection (contd.)

- INDEX SEARCH
- Applicable for comparison on the indexed attribute
- *Equality* using primary index

Selection (contd.)

- INDEX SEARCH
- 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 (contd.)

- INDEX SEARCH
- 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
  - $h + n$ 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 (contd.)

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

Selection (contd.)

- INDEX SEARCH
- *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 (contd.)

- Index search
- *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 (contd.)

- INDEX SEARCH
- *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 (contd.)

- INDEX SEARCH
- *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

Selection on multiple attributes

- *Conjunction*: AND

Selection on multiple attributes

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

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
- 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
- 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
- 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
- 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
- *Negation* of equality

Selection on multiple attributes

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

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

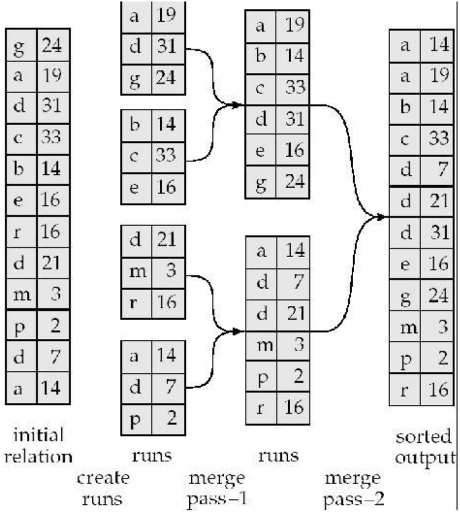
External mergesort

- 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

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


g 24
a 19
d 31
c 33
b 14
e 16
r 16
d 21
m 3
p 2
d 7
a 14
initial relation
a 19
d 31
g 24
b 14
c 33
e 16
d 21
m 3
r 16
a 14
d 7
p 2
runs
create runs
a 19
b 14
c 33
d 31
e 16
g 24
a 14
d 7
d 21
m 3
p 2
r 16
runs
merge pass–1
a 14
a 19
b 14
c 33
d 7
d 21
d 31
e 16
g 24
m 3
p 2
r 16
sorted output
merge pass–2

Cost of external mergesort

- Total number of blocks is b
- Initial number of sorted runs is

Cost of external mergesort

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

Cost of external mergesort

- Total number of blocks is b
- Initial number of sorted runs is $\lceil b/M \rceil$
- In each merge pass, $M - 1$ runs are sorted
- Therefore, total number of merge passes required is $\lceil \log_{M-1} \lceil b/M \rceil \rceil$
- 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 $\lceil b/M \rceil$
- In each merge pass, $M - 1$ runs are sorted
- Therefore, total number of merge passes required is $\lceil \log_{M-1} \lceil b/M \rceil \rceil$
- 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 $2b(\lceil \log_{M-1} \lceil b/M \rceil \rceil + 1)$

Cost of external mergesort (contd.)

- Initial pass to created sorted runs reads M blocks at a time
- Therefore, number of seeks is $2\lceil b/M \rceil$ 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 moves the disk head away, thereby requiring a seek every time
- Hence, number of seeks for these passes is $2b\lceil \log_{M-1} \lceil b/M \rceil \rceil$
- Therefore, total number of seeks is

Cost of external mergesort (contd.)

- Initial pass to created sorted runs reads M blocks at a time
- Therefore, number of seeks is $2\lceil b/M \rceil$ 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 moves the disk head away, thereby requiring a seek every time
- Hence, number of seeks for these passes is $2b\lceil \log_{M-1}\lceil b/M \rceil \rceil$
- Therefore, total number of seeks is $2\lceil b/M \rceil + 2b\lceil \log_{M-1}\lceil b/M \rceil \rceil$

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

Cost of nested-loop join

- There are n_i records in b_i blocks for relation i
- Assumption is that only two blocks fit in memory
- Block transfers
  - b_s transfers every time a record t_r is read
  - b_r transfers for records in r
  - Therefore, total is $b_r + n_r \times b_s$ transfers

Cost of nested-loop join

- There are n_i records in b_i blocks for relation i
- Assumption is that only two blocks fit in memory
- Block transfers
  - b_s transfers every time a record t_r is read
  - b_r transfers for records in r
  - Therefore, total is $b_r + n_r \times b_s$ transfers
- Seeks
  - 1 seek for records in s every time a record t_r is read
  - Thus, 2 seeks per record in r
  - Therefore, total is $2n_r$ seeks

Cost of nested-loop join

- There are n_i records in b_i blocks for relation i
- Assumption is that only two blocks fit in memory
- Block transfers
  - b_s transfers every time a record t_r is read
  - b_r transfers for records in r
  - Therefore, total is $b_r + n_r \times b_s$ transfers
- Seeks
  - 1 seek for records in s every time a record t_r is read
  - Thus, 2 seeks per record in r
  - Therefore, total is $2n_r$ seeks
  - However, a block from r can be stored in memory
  - Then, $b_r + n_r$ seeks
- Smaller relation should be

Cost of nested-loop join

- There are n_i records in b_i blocks for relation i
- Assumption is that only two blocks fit in memory
- Block transfers
    - b_s transfers every time a record t_r is read
    - b_r transfers for records in r
    - Therefore, total is $b_r + n_r \times b_s$ transfers
- Seeks
    - 1 seek for records in s every time a record t_r is read
    - Thus, 2 seeks per record in r
    - Therefore, total is $2n_r$ seeks
    - However, a block from r can be stored in memory
    - Then, $b_r + n_r$ seeks
- Smaller relation should be outer

Cost of nested-loop join (contd.)

- If both relations fit in memory

Cost of nested-loop join (contd.)

- If both relations fit in memory
  - $b_r + b_s$ transfers and 2 seeks
- If only one relation fits in memory

Cost of nested-loop join (contd.)

- If both relations fit in memory
  - $b_r + b_s$ transfers and 2 seeks
- If only one relation fits in memory
  - Smaller relation is made inner and read first
  - Same cost: $b_r + b_s$ transfers and 2 seeks

Block nested-loop join

- Applicable for any kind of join
- Block version of the nested-loop algorithm
- 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

Cost of block nested-loop join

- Assumption is that only two 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

- Assumption is that only two 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 $b_r + b_r$ seeks
- Smaller relation should be

Cost of block nested-loop join

- Assumption is that only two 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 $b_r + b_r$ seeks
- Smaller relation should be outer (like nested-loop join)

Cost of block nested-loop join (contd.)

- If both relations fit in memory
  - $b_r + b_s$ transfers and 2 seeks

Cost of block nested-loop join (contd.)

- If both relations fit in memory
  - $b_r + b_s$ transfers and 2 seeks
- If only one relation fits in memory
  - Smaller relation is made inner and read first
  - Same cost: $b_r + b_s$ transfers and 2 seeks

Cost of block nested-loop join (contd.)

- If both relations fit in memory
  - $b_r + b_s$ transfers and 2 seeks
- If only one relation fits in memory
  - Smaller relation is made inner and read first
  - Same cost: $b_r + b_s$ transfers and 2 seeks
- No difference between nested-loop join and block nested loop-join

Indexed nested-loop join

- Applicable when inner relation has an index on the joining attribute
- Indexed version of the block nested-loop algorithm
- 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 only two blocks fit in memory
- Size of index for relation i is c_i blocks
- Cost is c_s seeks and transfers for selection on index on s for every record in r
- b_r seeks and transfers for blocks in r
- Therefore, total is $b_r + n_r \times c_s$ seeks and $b_r + n_r \times c_s$ transfers

Cost of indexed nested-loop join

- Assumption is that only two blocks fit in memory
- Size of index for relation i is c_i blocks
- Cost is c_s seeks and transfers for selection on index on s for every record in r
- b_r seeks and transfers for blocks in r
- Therefore, total is $b_r + n_r \times c_s$ seeks and $b_r + n_r \times c_s$ 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 only two blocks fit in memory
- Size of index for relation i is c_i blocks
- Cost is c_s seeks and transfers for selection on index on s for every record in r
- b_r seeks and transfers for blocks in r
- Therefore, total is $b_r + n_r \times c_s$ seeks and $b_r + n_r \times c_s$ transfers
- If index is available on both relations, like block nested-loop, smaller relation should be outer

Merge join or Sort merge join

- Applicable only when the join condition is *equality*
- If necessary, sort the relations 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

Cost of merge join or sort merge join

- Each record is read only once
- Consequently, each block is read only once
- Blocks may be read in an interleaved manner
- Therefore, total cost is $b_r + b_s$ seeks and $b_r + b_s$ transfers

Cost of merge join or sort merge join

- Each record is read only once
- Consequently, each block is read only once
- Blocks may be read in an interleaved manner
- Therefore, total cost is $b_r + b_s$ seeks and $b_r + b_s$ transfers
- If b_b blocks of each relation can be buffered in memory, then cost is $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ seeks plus transfers

Cost of merge join or sort merge join

- Each record is read only once
- Consequently, each block is read only once
- Blocks may be read in an interleaved manner
- Therefore, total cost is $b_r + b_s$ seeks and $b_r + b_s$ transfers
- If b_b blocks of each relation can be buffered in memory, then cost is $\lceil b_r/b_b \rceil + \lceil b_s/b_b \rceil$ seeks plus transfers
- 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

Hash join

- Applicable only when the join condition is *equality*
- Idea: 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
- Hash function to partition records of *both* relations: PARTITION HASH JOIN
- Algorithm
  - Choose $h_{r,s} : record_{r,s} \rightarrow \{0, \ldots, n-1\}$
  - Output each record of r and s to partition into H_{r_i} and H_{s_i} where $h(t_r) = i$ and $h(t_s) = i$
  - Read partition H_{s_i} and build an in-memory hash index
  - For every record $t_{r_i} \in H_{r_i}$, probe hash index of H_{s_i} to locate matching records
- s is called build input
- r is called probe input
- Each partition of build relation must be stored in memory
- Build relation should be

Hash join

- Applicable only when the join condition is *equality*
- Idea: 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
- Hash function to partition records of *both* relations: PARTITION HASH JOIN
- Algorithm
  - Choose $h_{r,s} : record_{r,s} \rightarrow \{0, \dots, n-1\}$
  - Output each record of r and s to partition into H_{r_i} and H_{s_i} where $h(t_r) = i$ and $h(t_s) = i$
  - Read partition H_{s_i} and build an in-memory hash index
  - For every record $t_{r_i} \in H_{r_i}$, probe hash index of H_{s_i} to locate matching records
- s is called build input
- r is called probe input
- Each partition of build relation must be stored in memory
- Build relation should be smaller

Recursive partitioning

- Each of the n partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, n

Recursive partitioning

- Each of the n partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, $n \geq \lceil b_s / M \rceil$
- For partitioning to be done in one pass, each of n partitions must be stored in memory, i.e.,

Recursive partitioning

- Each of the n partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, $n \geq \lceil b_s/M \rceil$
- For partitioning to be done in one pass, each of n partitions must be stored in memory, i.e., $M > n$

Recursive partitioning

- Each of the n partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, $n \geq \lceil b_s/M \rceil$
- For partitioning to be done in one pass, each of n partitions must be stored in memory, i.e., $M > n$
- Combining, $M \geq \lceil b_s/M \rceil$ or $M > \sqrt{b_s}$
- Therefore, if $b_s \geq M^2$, partitioning cannot be done in one pass

Recursive partitioning

- Each of the *n* partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, $n \geq \lceil b_s / M \rceil$
- For partitioning to be done in one pass, each of n partitions must be stored in memory, i.e., $M > n$
- Combining, $M \geq \lceil b_s / M \rceil$ or $M > \sqrt{b_s}$
- Therefore, if $b_s \geq M^2$, partitioning cannot be done in one pass
- Recursive partitioning is employed
- Initially, n_s is chosen to less than or equal to M
- Each partition is then read and re-partitioned
- This is continued till each partition fits into memory

Recursive partitioning

- Each of the n partitions should fit into memory
- If build relation has b_s blocks, for a memory size of M blocks, $n \geq \lceil b_s/M \rceil$
- For partitioning to be done in one pass, each of n partitions must be stored in memory, i.e., $M > n$
- Combining, $M \geq \lceil b_s/M \rceil$ or $M > \sqrt{b_s}$
- Therefore, if $b_s \geq M^2$, partitioning cannot be done in one pass
- Recursive partitioning is employed
- Initially, n_s is chosen to less than or equal to M
- Each partition is then read and re-partitioned
- This is continued till each partition fits into memory
- Hybrid hash join: Retain the first partition of build relation in memory

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 and reading them again requires $2(b_i + n)$ transfers
- Therefore, total number of transfers is $3(b_r + b_s) + 4n$

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 and reading them again requires $2(b_i + n)$ transfers
- Therefore, total number of transfers is $3(b_r + b_s) + 4n$
- Reading relation i requires $\lceil b_i / b_b \rceil$ seeks where b_b is the input buffer size
- Writing relation i requires $\lceil (b_i + n) / b_b \rceil$ seeks where b_b is the output buffer size
- Reading n partitions of relation i requires n seeks
- Therefore, total number of seeks is $2\lceil (b_r + b_s + n) / b_b \rceil + 2n$

Cost of hash join (contd.)

- If recursive partitioning is used, number of passes required is

Cost of hash join (contd.)

- If recursive partitioning is used, number of passes required is $\lceil \log_M b_i \rceil$
- Ignoring partially-filled partitions
  - Total number of transfers is $2b_r\lceil \log_M b_r \rceil + 2b_s\lceil \log_M b_s \rceil + (b_r + b_s)$

Cost of hash join (contd.)

- If recursive partitioning is used, number of passes required is $\lceil \log_M b_i \rceil$
- Ignoring partially-filled partitions
  - Total number of transfers is $2b_r\lceil \log_M b_r \rceil + 2b_s\lceil \log_M b_s \rceil + (b_r + b_s)$
  - Total number of seeks is $2\lceil b_r/b_b \rceil\lceil \log_M b_r \rceil + 2\lceil b_s/b_b \rceil\lceil \log_M b_s \rceil + 2n$

Other operations

- Complex join

Other operations

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

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Aggregation with grouping

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Aggregation with grouping
  - Use hashing to organize into groups; then apply aggregation

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Aggregation with grouping
  - Use hashing to organize into groups; then apply aggregation
- Outer join

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Aggregation with grouping
  - Use hashing to organize into groups; then apply aggregation
- Outer join
  - Nested-loop algorithms require almost no modification
  - Select all records while scanning in merge join
  - If $r ⟕ s$, r should be the

Other operations

- Complex join
  - Use block nested-loop for conjunctive and/or disjunctive selection
  - If only equality, union/intersection of results of merge/hash join may be used
- Duplicate elimination
  - Use hashing or sorting
- Set operations
  - If relations are sorted, scan in order
  - Build hash index on one relation; test records from other relation
- Aggregation with grouping
  - Use hashing to organize into groups; then apply aggregation
- Outer join
  - Nested-loop algorithms require almost no modification
  - Select all records while scanning in merge 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