Query Optimisation Tutorial

SWEN 304

Trimester 2, 2017

Lecturer: Dr Hui Ma

Engineering and Computer Science

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

Outline

- A query example with aggregate operations
- Cost function of DISTINCT project
- Cost functions of select operation
- Improving efficiency of join using more memory
- Cost of set theoretic operations
- Cost of aggregate functions

Query with Aggregate Operation: An Example (1)

- Consider the relational algebra statement:

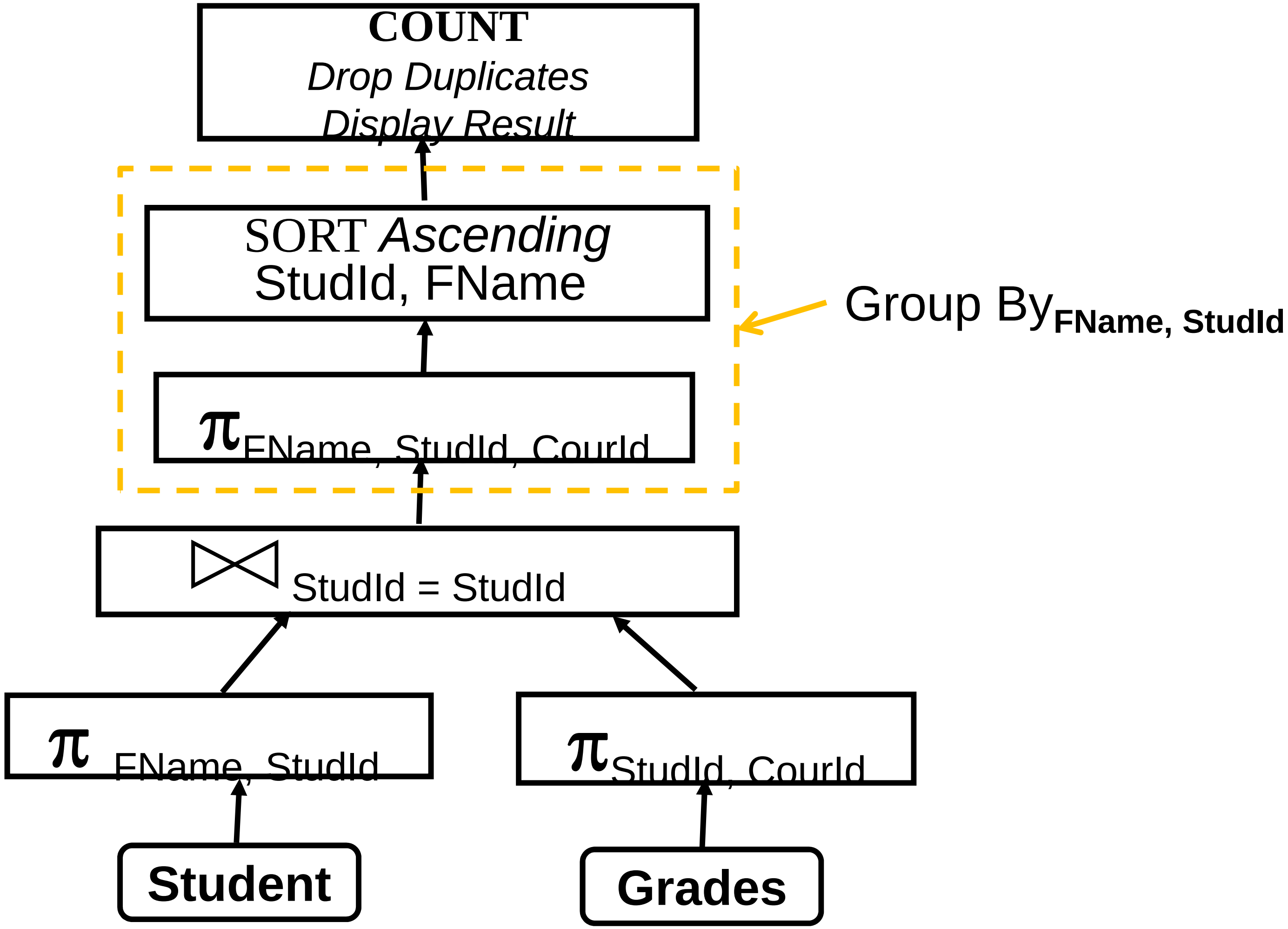
  $_{\text{FName, StudId,}}\mathcal{F}_{(\text{COUNT, }*)}(\text{Grades} \bowtie_{\text{StudId = StudId}} \text{Student})$

- For each student retrieve the number of papers **enrolled**

- A query processor would build the following binary query tree

Query with Aggregate Operation: An Example (2)

- Optimized Query Tree of Logical Operator



Query with Aggregate Function: An Example (3)

- Suppose nested loop join and blocking factor *f = 2*

$\pi_{\text{Fname, StudId}}$

FName	StudId
Susan	131313
James	007007
Susan	555555
John	010101

$\bowtie_{\text{StudId = StudId}}$

$\pi_{\text{StudId, CourId}}$

StudId	CourId
007007	C302
555555	C302
007007	C301
007007	M214
131313	C201
555555	C201
131313	C302
007007	C201
010101	C201

=

Grades $\bowtie_{\text{StudId = StudId}}$ Student

Fname	StudId	StudId	CourId
James	007007	007007	C302
Susan	555555	555555	C302
James	007007	007007	C301
James	007007	007007	M214
Susan	131313	131313	C201
Susan	555555	555555	C201
Susan	131313	131313	C302
James	007007	007007	C201
John	010101	010101	C201

Query with Aggregate Function: An Example (4)

- Sort and Display Result

SORT

Fname	StudId	CourId
James	007007	C302
James	007007	C301
James	007007	M214
James	007007	C201
John	010101	C201
Susan	131313	C201
Susan	131313	C302
Susan	555555	C302
Susan	555555	C201

Display Result

Fname	StudId	NoOfPap
James	007007	4
John	010101	1
Susan	131313	2
Susan	555555	2

Cost Function of a Project Operation

- The <attribute_list> of a project operation
  - contains a relation schema key, or
  - DISTINCT is not used in the SQL SELECT command,
  - The cost function is:

$$C = b_1 + b_2$$

  - Complexity O(r)
- DISTINCT is used in the SQL SELECT command,
  - With index and index being implemented as a B^+-tree
  - Complexity is O(d) or O(r)
  - The cost function is:

$$C = \lceil d(Y) / m \rceil + \lceil d(Y) / f \rceil$$

With $d(Y)$ ($\leq r$) as the number of different Y = <attr_list> values, and m the number of node entries

Projection: Evaluation of DISTINCT

- The <attribute_list> of the SELECT clause does not contain a relation schema key:
  - Without sorting and index: complexity is $O(r^2)$
  - With sorting: complexity is $O(r * log r)$
  - With index and index is implemented as a B^+-tree complexity is $O(d)$ or $O(r)$

Relationship Between b, r, f

- Let $(a_1,\ldots, a_n)$ be a tuple, and l_i the size of a_i in bytes, then

$$L = \sum_{i=1}^{n} l_i$$

  is the storage capacity needed to store a tuple
- Let B be the size (capacity) of a block on disk
  - B is a constant, defined during the disk initialization
- Let f be the number of tuples that fit into a block, then

$$f = \lfloor B / L \rfloor$$

- The relationship between the number of blocks b, number of records r, and blocking factor f is

$$b = \lceil r / f \rceil$$

Projection: Size Calculation

- The storage capacity needed to store one block of tuples is $B \geq L * f$
- Let $B_1 = L_1 * f_1$, and $B_2 = L_2 * f_2$ be given
- If $L_1 > L_2$, and $B_1 = B_2$, then

  $f_2 \geq f_1$ (more L_2 tuples fit into a block)

- Let r_1, L_1, f_1, and r_2, L_2, f_2 be given
- If $r_1 = r_2$, $L_1 > L_2$, and $B_1 = B_2$, then

  $b_1 \geq b_2$ (less blocks needed to store L_2 tuples)

- Since a project operation drops some fields in tuples, there will be less blocks after projection

Selection: Attribute Selection Cardinality

- If an attribute A of a relation schema N has $d(A)$ actual distinct values, then its selection cardinality $s(A)$ is

$$s(A) = r / d(A)$$

- For a key K, $d(K) = r$, and $s(K) = 1$
- If an attribute A is not a key, then

$$s(A) = (r / d(A)) \geq 1$$

- Selection cardinality $s(A)$ of the attribute A, allows us to compute how many tuples is expected to contain a given value

$$a \in \pi_A(N)$$

- *We always assume a uniform distribution*

Selection: Attribute Selection Cardinality Examples

- Consider relation *Student* having *1,000* tuples, then
  - $d(StudID) = 1{,}000$, and $s(StudID) = 1$
  - $d(Major) = 4$ and $s(Major) = 200$
  - $d(LName) = 800$, and $s(LName) = 1.25$
- Consider relation *Grades*, having *10,000* tuples, suppose:
  - $d(CourId) = 20$, $d(StudId) = 1000$
  - $Grade \in \{A+, A, A-, B+, B, B-, C+, C\}$,
- then:
  - $s(StudId) = 10$
  - $s(CourId) = 500$
  - $s(Grade) = 1{,}250$
  - $s(StudId, CourId) = 1$

Cost Functions of Select Operation

- **Linear search** (neither indexes nor hash functions provided)

  $C = b + \lceil s / f \rceil$, hence $O(r)$

- **Unique key index** (B^+-tree):
  - If selection condition is $K = k$:

    $C = h + 1 + \lceil 1 / f \rceil$

    Hence $O(\log r)$ – index height h is proportional to $\log r$
  - If selection condition is $k_1 \leq K \leq k_2$ and suppose $s \leq r$ tuples satisfy the condition:

    $C = h + \lceil s / m \rceil + s + \lceil s / f \rceil$

    where m is the number of entries in a tree node

    $\lceil s / m \rceil$ is the number of tree leaves containing key values $k_1 \leq K \leq k_2$

    hence $O(\max\{\log r, s\})$

Cost Functions of Select Operation

- **Secondary index** (B^+-tree) on secondary key Y
  - $s \leq d(Y)$ random tuples satisfy condition $Y = y$
  - each Y value has a pointer to a sequence of blocks containing up to p pointers to tuples in the data area
  - the height h of the tree is proportional to $log(d(Y))$

$$C = h + \lceil s / p \rceil + s + \lceil s / f \rceil,$$

- Hence O(s)

Join: An Example of the Size of a Join Result

N

A	*B*
1	1
0	2
5	3

$\bowtie$

M

B	*C*
1	1
1	2
2	3
2	4
3	5
3	6

=

$N \bowtie M$

A	*B*	*B*	*C*
1	1	1	1
1	1	1	2
0	2	2	3
0	2	2	4
5	3	3	5
5	3	3	6

$\boldsymbol{r_N = 3, r_M = 6,}$

Since relation *N* key is *B*, and referential integrity $M[B] \subseteq N[B]$ is satisfied, $|N \bowtie M| = r_M = 6$

Join: Cost Function

- The result of an equijoin is a set of tuples over a concatenation of relation *N* and relation *M* attribute sets
- So, if the relation *N* tuple size is L_N and the relation *M* tuple size is L_M, then the size *L* of an equijoin result tuple is:

$$L = L_N + L_M$$

- Since the size of a block is always given, the result blocking factor is

$$f = \lfloor B / L \rfloor$$

- The size of the join result is:

$$\lceil r_M / f \rceil$$

Join: Cost Function of Nest Loop Join

- Cost of a nested loop join is

$$C = b_N + b_M * \lceil b_N / (n - 2) \rceil + \lceil r_M / f \rceil$$

- Use more memory can improve join efficiency
- Examples:
  - If the number of memory buffers is $n = 3$, then

$$C = b_N + b_M * b_N + \lceil r_M / f \rceil$$

  - If the number of memory buffers is $n = 12$, then

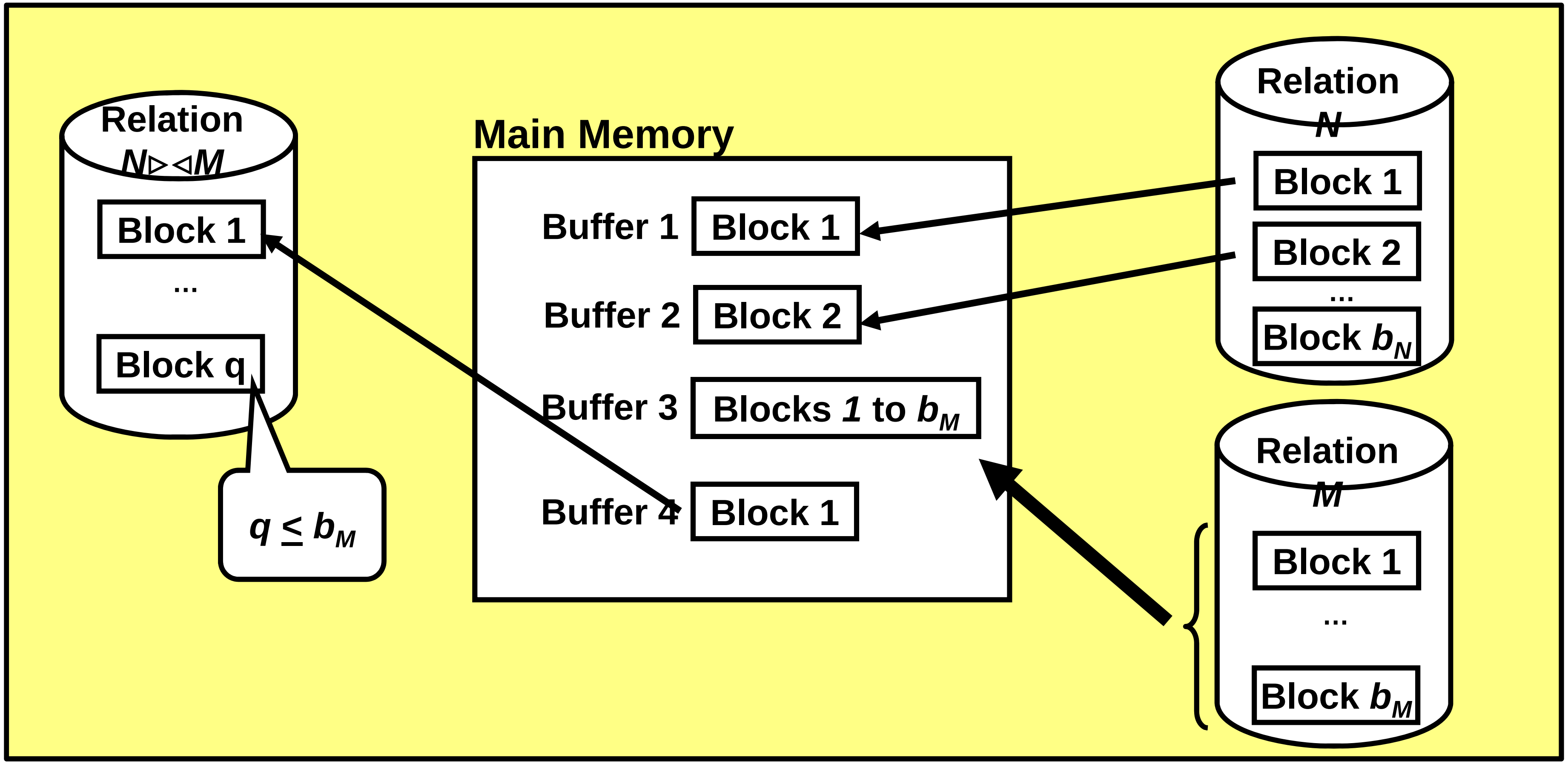
$$C = b_N + b_M * \lceil b_N / 10 \rceil + \lceil r_M / f \rceil$$

Nested Loop Join – Four Buffers (n = 4)

For each two (n-2) successive blocks of the relation *N* transferred in the main memory,

All of *M* blocks are transferred into the main memory

Hence: $b_N / 2 * b_M$ accesses



Cost of Nested Loop Join: An Example

- Cost of a nested loop join is

$$C = b_N + b_M * \lceil b_N / (n - 2) \rceil + \lceil r_M / f \rceil$$

- Let:
  - $r_N = 600$, $b_N = 60$,
  - $r_M = 5000$, $b_M = 1000$,
  - $n = 5$,
  - blocking factor of join result $f = 10$,
  - join condition attribute Y be the key of N, and
  - referential integrity $M[Y] \subseteq N[Y]$ satisfied
- Then, the cost of $N \bowtie M$ is:

$$60 + 1000 \lceil 60 / 3 \rceil + \lceil 5000 / 10 \rceil = 20{,}560$$

The cost of $M \bowtie N$ is:

$$1000 + 60 \lceil 1000 / 3 \rceil + \lceil 5000 / 10 \rceil = 21{,}540$$

Cost of the Set Theoretic Operations

- Generally, union, set difference, and intersection require comparing each tuple of one relation with all tuples of the other relation
  - Hence, complexity is $O(r^2)$
- Sorting considerably improves efficiency
- Algorithm:
  - Sort relations N and M
  - Read blocks from relations N and M, one from each at a time
  - Compare tuples from the blocks read in
  - Output into the result relation those tuples that satisfy the condition of the operation at hand

Cost of Aggregate Functions

- Executing an aggregate function **without** grouping requires reading all b blocks into main memory and outputing just one result block, so

$$C = b + 1,$$

  - Hence O(r)
- Executing an aggregate function with **grouping** requires comparing each tuple with all other tuples and keeping partial aggregate results in main memory
  - So, complexity is $O(r^2)$
- Sorting greatly improves efficiency

Improving Cost of Aggregate Functions

- Algorithm:
  - Sort the relation according to the <grouping_list>,
    - All tuples of a group come next to each other
  - Read successive blocks in
  - When a complete group is read in
    - Compute aggregate
    - Output the result tuple
- Complexity is O($r * log\ r$), the complexity of the sort operation
- If there exists an appropriate access structure on the whole <grouping_list> (like a cluster index), sorting is unnecessary
  - Then, the complexity becomes O($log\ r$) (**height** of the index)