Cost and feasibility estimations of an execution plan

Maarten Fokkinga

Version of December 9, 2011, 9:02

Abstract. The explanation by Kifer [1] for calculating the total cost and amount of buffers of a query execution plan (is elegant, clear, well-phrased, correct, instructive and so on, but) does not take care of pipelining and the available buffers in a systematic way, and, instead, needs *ad hoc* reasoning for these aspects. We show a method that is systematic, easily to be automated, and needs no *ad hoc* reasoning when applied.

1 Goal. An execution plan consists of a relational algebra expression, in tree form, together with an *access path* for each node in the tree, an estimation of the *total cost* of producing the end result, and the *amount of buffers* that are simultaneously used during the production of the end result. The total cost is often the main point of interest, the amount of buffers determines whether the plan is feasible in view of the number of available buffers. We shall show how to calculate total cost and amount of buffers in a *systematic* way, easily to be automated.

We follow the method of Kifer [1] as much as possible (and assume that the reader is familiar with his exposition) and expand his approach, and deviate from it, only in order to be systematic. In particular, *pipelining* and the *amount of buffers* is dealt with in the formulas and needs no *ad hoc* explanation anymore.

2 Problem and solution. Let us explain the problem and our solution by means of a single example: the block-nested loop with M input buffers for the *outer* argument. Denoting the size of a table or relation R by F_R (being a number of pages), Kifer [1] says that the entire outer argument is read in F_{outer}/M chunks of M pages at a total cost of F_{outer} page fetches, and for each chunk the entire *inner* argument is read at the cost of F_{inner} (putting the join of the chunk with *inner* in the output buffer); making together $F_{outer} + \lceil F_{outer}/M \rceil \times F_{inner}$ page fetches. However, if the outer argument is pipelined into the block-nested loop, then in the cost formula the first occurrence of F_{outer} should be replaced by 0 whereas the second one stays unaffected. In addition, to express the total, overall cost of the join result, the costs of producing the arguments should also be taken into account. To avoid *ad hoc* manipulation of the cost formula, we formulate the cost of the block-nested loop using I_{outer} and F_{outer}, where I_{outer} stands for the cost of "input" of the outer argument and F_{outer} for its size. The usage of the formula in an execution plan can then be specialized to "pipelining the outer argument" by substituting 'the cost of producing the outer argument' for all all occurrences of I_{outer}, and be specialized to "reading outer from disk" by *both* substituting F_{outer} for all occurrences of I_{outer} *and* adding 'the cost of producing and materializing the outer argument'.

Pipelining is a technique that may save a lot of page transfers (in comparison to writing intermediate results to disk and later reading them from disk). However, it has its price too: the usage of buffers—of which there is only a limited amount. So, we must also carefully formulate the usage of buffers, and take account of the fact that pipelining an argument to an operation has as consequence that both the production of the argument and the operation are executed simultaneously, using the same pool of buffers.

Before we can describe this method in detail, in §4, first a careful definition of the relevant concepts is needed.

3 Basic concepts. Following Kifer [1], we measure the *cost* of executing a plan in the number of page transfers, without distinguishing between sequential page access and random page access. The *size* of a table or relation R is measured in number of tuples, T_R, and in number of pages, F_R. Recall that *tables* may be base tables, already stored on disk, and computed tables, such as the result of nodes in the execution plan. Further, the following notions are crucial for the systematization.

- *Production* of a table is:-

  > a computation that puts the table rows *into the output buffer*; the page transfers for flushing the output buffer to disk are not part of this computation and not counted in the production cost. In fact, sometimes the output buffer is not flushed to disk, namely when its content is pipelined to the parent node.

  Materialization of a table is:-

  > a computation such that upon completion the table rows are stored on disk. For a table that is already stored on disk, this computation may be a noop. For a computed table this computation is: producing the table and writing it to disk; the page transfers for writing-to-disk are included in the computation and are counted in the materialization cost.

  For a computed table, the cost of writing it eventually to disk is the same for all execution plans for a given query and therefore it is not needed for determining the cheapest plan. So primarily we are interested in production costs. However, in subcomputations of intermediate results both production and materialisations may play a role.

- *Pipelining* of a child node to its parent node is:-

  > taking the "child's output buffer" and the "parent's buffer for inputing the child's result" to be *one and the same* buffer. The operation of "flushing the output buffer" in the child is synchronized with "emptying the corresponding input buffer" at the parent.

  Thus pipelining avoids materialization and may save page transfers. Pipelining has as consequence that the production of the child's result is done *simultaneously* with the operation of the parent thus affecting 'the buffers simultaneously in use'.

  > **Footnote.** If an access path uses *several* buffers to input the result of a child, C say, then pipelining child C to its parent means that the output buffer of C is identified successively with each of the input buffers in the parent. This successive identification might be done over and over again, if the access path uses an iteration to input the C's result; see node (5) in Example 3 below.

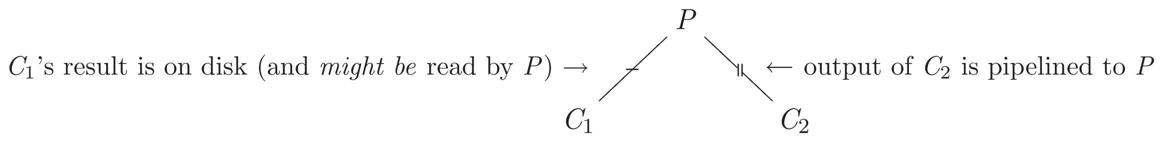
- *The amount of buffers simultaneously used* in a computation is:-

the minimum number of buffers with which the computation can be done, excluding the single output buffer – if any.

It is not formally necessary to exclude the output buffer, but it is handy for two reasons: first, every access paths has exactly one output buffer (so there is nothing wrong with excluding it systematically), and second, in the case of *pipelining* the output buffer need not be counted separately since it is at the same time an input buffer.

- *Buffer.* We consistently use the word *buffer* as a synonym for *buffer page.* So, instead of talking about "an input buffer of n pages" we'll talk about "n input buffers".

Pipelining notation. In an expression tree, a line marked with – denotes that the child's result eventually resides on disk, whereas a line marked with ॥ denotes pipelining of the child's output to the parent's input (equating the child's output buffer with the parent's input buffers):


P
C_1's result is on disk (and *might be* read by P) $\rightarrow$
$\leftarrow$ output of C_2 is pipelined to P
C_1
C_2

4 The method. In the tree we enumerate the children of a node in a fixed (left to right) order by $1, 2, \ldots$. We assume that for each node N the following is known:

- *piped/stored*:- the numbers of the children of which the result is pipelined into N's operation and stored on disk, respectively.
- C:- the *operation* cost of producing N's result *expressed in terms of* the argument sizes T_i, F_i and the costs I_i of inputing the arguments, for various child nodes i.
- B:- the number of buffers used internally for N's operation and input of arguments.

This information is, in fact, part of the access path at node N. Section §5 elaborates these data for several access paths.

Method. The method is to calculate in a bottom-up way, or to verify in arbitrary order, for each node N these three quantities:

- T_N, F_N:- an estimation of the size of the result, measured in rows and pages, respectively,
- $\begin{cases} P_N\text{:- an estimation of the cost of producing } N\text{'s result, if } N \text{ is pipelined to its parent,} \\ M_N\text{:- an estimation of the cost of materializing } N\text{'s result, if } N \text{ is not pipelined} \end{cases}$
- B_N:- a lowerbound for the amount of buffers that are simultaneously used during the production of N's result (excluding the output buffer).

The calculation and verification of the result sizes is independent of the access paths and is beyond the scope of this note. The other quantities are fully determined, for a node N with children $N_1, N_2, \ldots$ and access path info C and B, by these equations:

- For a leaf node N:

$$\begin{aligned} B_N &= B \\ P_N &= F_N \\ M_N &= 0 \end{aligned}$$

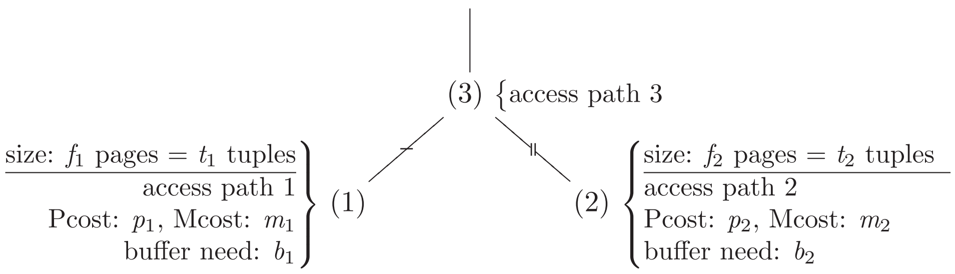
- For a non-leaf node N:

$$\begin{aligned}
B_N &= \text{the maximum of } B + \textstyle\sum_{i \in piped} B_{N_i} \text{ and all } B_{N_i} \text{ (for } i \in \mathit{stored}\text{)} \\
&= \max\left(\{B + \textstyle\sum_{i \in piped} B_{N_i}\} \cup \{i \in \mathit{stored} \bullet B_{N_i}\}\right) \\
P_N &= C' + \textstyle\sum_{i \in stored} M_i \\
M_N &= P_N + F_N
\end{aligned}$$

where C' is obtained from expression C by the following substitution, for all children i:

$$I_i \;:=\; \begin{cases} P_i & \text{if } i \in \mathit{piped} \\ F_i & \text{if } i \in \mathit{stored} \end{cases}$$

Example. Here is a typical application of "the method" for a non-leaf node. Consider the following fragment of an execution plan, in which the result of (1) resides on disk (whether because it is a base table or because it is produced and explicitly stored) and the result of (2) (whether residing on disk or explicitly produced) is pipelined to (3):


(3) {access path 3
size: f_1 pages = t_1 tuples
access path 1
Pcost: p_1, Mcost: m_1
buffer need: b_1
(1)
(2)
size: f_2 pages = t_2 tuples
access path 2
Pcost: p_2, Mcost: m_2
buffer need: b_2

Calculation and verification of size estimations are independent of the access paths and beyond the scope of the method, so we just assume that sizes $T_{(3)}$ and $F_{(3)}$ are given. Now suppose that node (3) is computed by an access path with local buffer need B and local cost estimation formula C. The overall costs $P_{(3)}$ and $M_{(3)}$ for *producing* and *materializing*, respectively, the result of (3), and the total amount $B_{(3)}$ of buffers needed during the *production* (thus excluding the overall output buffer) of the result of node (3), are then calculated as follows:

$$\begin{aligned}
B_{(3)} &= \max\{B + b_2,\ b_1\} \\
P_{(3)} &= C[I_1 := f_1,\ I_2 := p_2] + m_1 \\
M_{(3)} &= P_{(3)} + f_3
\end{aligned}$$

Proof. A formal proof of the correctness of the proposed method would be based on concrete algorithms that execute the plan. The formulation of these algorithms and, hence, a formal correctness proof is beyond the scope of our exposition. Instead, we provide an informal proof (which may be skipped without loss of continuity), in which we suggest to some degree how the algorithms look like, and in which we count on some imagination of the reader.

- **Leaf node**. The equations for a leaf node are obvious.
- **Non-leaf node, eqn for B_N.** A child whose result is pipelined into node N is executed simultaneously with the operation of N (because the output buffer in the child is identified with an input buffer in N's operation, and "flushing the output buffer" in a child and "emptying the corresponding input buffer" in N's operation are synchronized). The access path says that the operation in node N itself

uses B buffers. So, in order to produce N's result, its pipelined children need to produce their result simultaneously, and hence $B + \sum_{i \in piped} B_{N_i}$ might be the amount of buffers used at some time during the overall production. But this is not the whole story. The execution of a child whose result is stored on disk, need be completed before N's operation ends, and *can* be executed entirely before N's operation. If such a child needs more buffers than $B + \sum_{i \in piped} B_{N_i}$, then certainly that amount of buffers is needed in order to produce N's result.

- **Non-leaf node, eqn for P_N.** The cost estimation C of the access path is expressed, amongst others, in terms of "the cost of inputing the result of child i", denoted I_i. For a child i whose result resides on disk, this input costs F_i page fetches, hence I_i is replaced in C by F_i in order to form P_N; in addition the cost of materializing child i's result has to be added, as done in the term $\sum_{i \in stored} M_i$. For a child i that is pipelined to N, the input of the child's result from its output buffer to N's input buffer comes for free, but for each such input the child has to produce its result; this is reflected in the substitution of I_i by P_i in C in order to form P_N.

- **Non-leaf node, eqn for M_N.** Materialization differs from production only in that the final result is flushed from the output buffer to disk, thus costing an additional amount of F_N page transfers.

The method is illustrated concretely in a series of elaborated examples in §6. First we need to formally define the info of the access paths.

5 Access paths.

The access paths are characterized by the following formula's.

- *Noop.* A noop does nothing; it makes sense and may be applied only if its argument is already stored on disk. Pipelining the result cannot be done, since the result is not produced in the output buffer.
  $B = 0, \quad C = 0$.

- *Sequential scan* with optional filter. It needs only one buffer: this one is the input buffer and the output buffer at the same time, so that "the buffers in use during the production" is zero (the output buffer is not counted!). The cost is: the cost of inputing the argument. The optional filter determines which rows are removed from the buffer.
  $B = 0, \quad C = I_1$.

- *Merge join* with $N_1 + N_2 \geq 2$ input buffers. A merge join expects both arguments to be sorted on the join key. The merge operation inputs the arguments in N_1 and N_2 buffers, respectively, and, for each filling of the buffers, looks for matching pairs of rows and, for each matching pair, it puts their join into the output buffer. This approach assumes that for each value v of the join key, the entire set $\sigma_{\text{join key}=v}\, argument_i$ fits in N_i pages.
  $B = N_1 + N_2, \quad C = I_1 + I_2$.

- *Explicit sorting* with N auxiliary buffers. (As a relational algebra operation, sorting is the identity since the sets of input tuples and output tuples are the same.) Sorting a file of size F with external sorting using N buffers, excluding the output buffer, costs about $2F\lceil \log_N F \rceil$ page transfers. Initially the file is read from disk and eventually the file sits on disk. So, to cater for pipelining the input from the child node and pipelining the output to the parent node, $2F$ page transfers must be subtracted and, instead, the term I (for inputing the file) must be added.
  $B = N, \quad C = I_1 + 2F_1\lceil \log_N F_1 \rceil - 2F_1$.

  It follows that if $N \geq F$, the sorting can be done entirely in-memory without page transfers.

- *Sort-merge join.* A sort-merge join can be expressed in the relational algebra tree as explicit sort operations on the children separately, followed by a merge join.
  It follows that if the sort nodes use N_1 and N_2 buffers (excluding the two output buffers), respectively, and are pipelined to the merge node having N input buffers, then the cost formula for these three nodes together is:
  $B = N + N_1 + N_2, \;\; C = I_1 + 2F_1\lceil\log_{N_1} F_1\rceil - 2F_1 \; + \; I_2 + 2F_2\lceil\log_{N_2} F_2\rceil - 2F_2.$

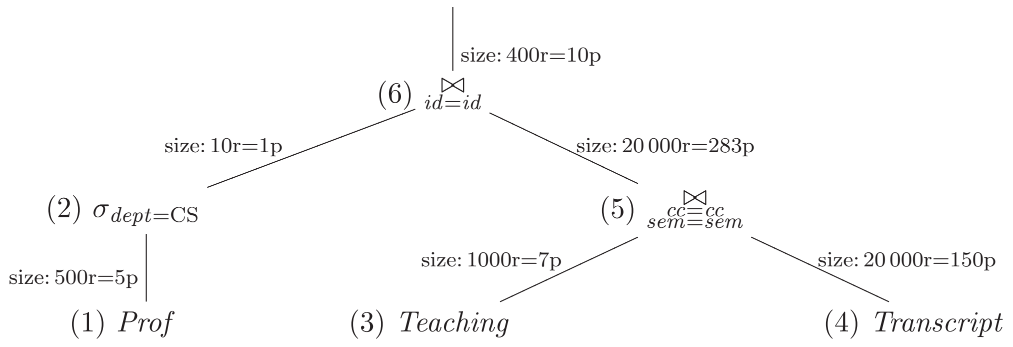
- *Optimized sort-merge join* using $N_1 + N_2 \geq 2$ buffers. According to Kifer's explanation [1, Section 10.5.2], this operation sorts the arguments in parallel, using N_1 and N_2 buffers, respectively, (excluding the output buffer), and combines the final merge step of the sorting with the merge step of the join. This approach assumes that for each value v of the join key, the entire set $\sigma_{\text{join key}=v}\, argument_i$ fits in N_i pages.
  $B = N_1 + N_2, \;\; C = I_1 + 2F_1\lceil\log_{N_1} F_1\rceil - 2F_1 \; + \; I_2 + 2F_2\lceil\log_{N_2} F_2\rceil - 2F_2.$

  Footnote. The 'sort-merge join with pipelining' and the optimized sort-merge join have exactly the same cost formula, but the latter needs less buffers. When they have the same amount of buffers, the optimized merge join can indeed do its job with less page transfers.

- *Block-nested loop* with N buffers for inputing *outer*. Per iteration, it reads N pages of the *outer* argument into N buffers, and looks for matching rows in the *inner* argument by inputing it page by page; for each pair of matching rows the join is put into the output buffer.
  $B = N + 1, \;\; C = I_{outer} + \lceil F_{outer}/N\rceil \times I_{inner}.$

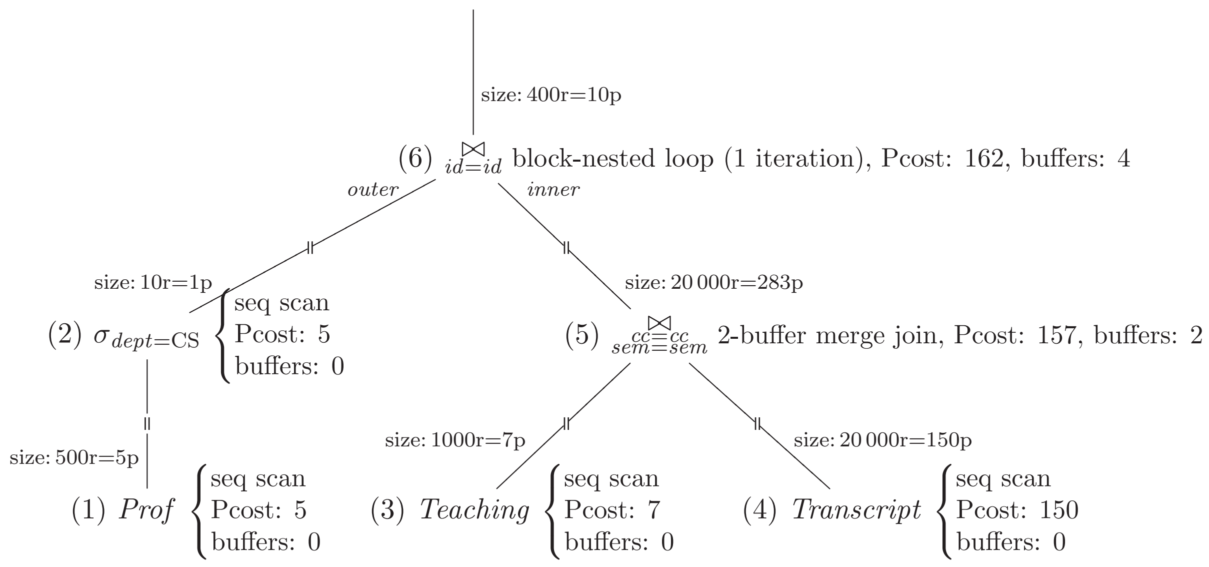
- *Index-nested loop.* An index-nested loop inputs its outer argument page by page (using one buffer), and for each of the rows it uses the index on the inner argument to look up the matching rows; for this latter look-up one buffer is needed, since reads from disk can only be done page-wise. For each matching pair it immediately produces a join row in the output buffer.
  $B = 2, \quad C = I_{outer} + T_{outer} \times (\rho + \mu)$
  where
  ρ is 2–4 for a B^+ tree and 1.2 for a hash index
  (i.e., the average, per tuple of *outer*, of the cost to get the *first* index entry),
  and, if the index is unclustered:
  μ is the average, per tuple of *outer*, of the number of matching tuples in *inner* but no more than F_{inner},
  but, if the index is clustered:
  μ is the average, per tuple of *outer*, of the number of (adjacent!) *inner* pages containing matching tuples (in practice, 1 or 2 pages).

6 Concrete examples. The examples use table *Prof*, *Teaching*, *Transcript*, and concern this relational algebra expression, in tree form:



Calculation and verification of size estimations are independent of the access paths and beyond the scope of the method, so the figures in the tree should be taken for granted. We present a series of examples, all having the tree above but with varying access paths at the nodes.

Example 1. A 1-buffer block-nested loop for (6) and a 2-buffer merge join for (5), assuming both *Teaching* and *Transcript* are sorted on (cc, sem), and all results are pipelined to their parent nodes. Here is the plan:


size: 400r=10p
(6) $\bowtie_{id=id}$ block-nested loop (1 iteration), Pcost: 162, buffers: 4
outer
inner
size: 10r=1p
size: 20 000r=283p
(2) $\sigma_{dept=\text{CS}}$ seq scan Pcost: 5 buffers: 0
(5) $\bowtie_{cc=cc, sem=sem}$ 2-buffer merge join, Pcost: 157, buffers: 2
size: 500r=5p
size: 1000r=7p
size: 20 000r=150p
(1) *Prof* seq scan Pcost: 5 buffers: 0
(3) *Teaching* seq scan Pcost: 7 buffers: 0
(4) *Transcript* seq scan Pcost: 150 buffers: 0

Verification of the cost estimations and buffer counts:

(1) For a sequential scan, we have $B = 0$ and $C = I_1$.
This node is a leaf node, so:
$B_{(1)} \;=\; B \;=\; 0.$
$P_{(1)} \;=\; F_{(1)} \;=\; 5.$

(2) For a sequential scan, we have $B = 0$ and $C = I_1$.
For this node, child 1 is (1), and $\mathit{piped}_{(2)} = \{(1)\}$ and $\mathit{stored}_{(2)} = \varnothing$, so:
$B_{(2)} \;=\; \max\{B + B_{(1)}\} \;=\; 0 + 0 \;=\; 0.$
$P_{(2)} \;=\; C[I_1 := P_{(1)}] \;=\; P_{(1)} \;=\; 5.$

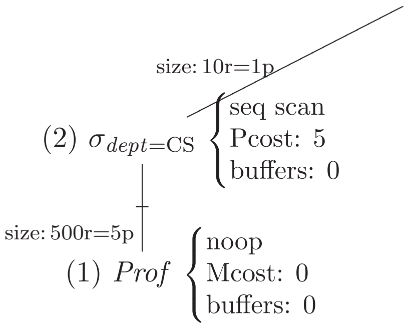
(3,4) Similarly to (1).

(5) For a 2-buffer merge join, we have $B = 2$ and $C = I_1 + I_2$.
Here, child 1 is (3), child 2 is (4), and $\mathit{piped}_{(5)} = \{(3), (4)\}$ and $\mathit{stored}_{(5)} = \varnothing$, so:
$B_{(5)} \;=\; \max\{B + B_{(3)} + B_{(4)}\} \;=\; 2 + 0 + 0 \;=\; 2.$
$P_{(5)} \;=\; C[I_1 := P_{(3)},\, I_2 := P_{(4)}] \;=\; 7 + 150 \;=\; 157.$

(6) For a block-nested loop with N=1 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, *outer* = (2) and *inner* = (5) and $\mathit{piped}_{(6)} = \{(2), (5)\}$ and $\mathit{stored}_{(6)} = \varnothing$, so:

$$\begin{aligned}
B_{(6)} &= \max\{B + B_{(2)} + B_{(5)}\} \;=\; 2 + 0 + 2 \;=\; 4.\\
P_{(6)} &= C[I_1{:=}P_{(2)},\, I_2{:=}P_{(5)}]\\
&= P_{(2)} + \lceil F_{(2)}/N \rceil \times P_{(5)}\\
&= 5 + \lceil 1/1 \rceil \times 157\\
&= 5 + 157\\
&= 162.
\end{aligned}$$

Example 2. A small change in the previous plan, without affecting the total cost: a noop on *Prof*, keeping it materialized on disk, followed by reading it from disk in (2). Here is the plan:



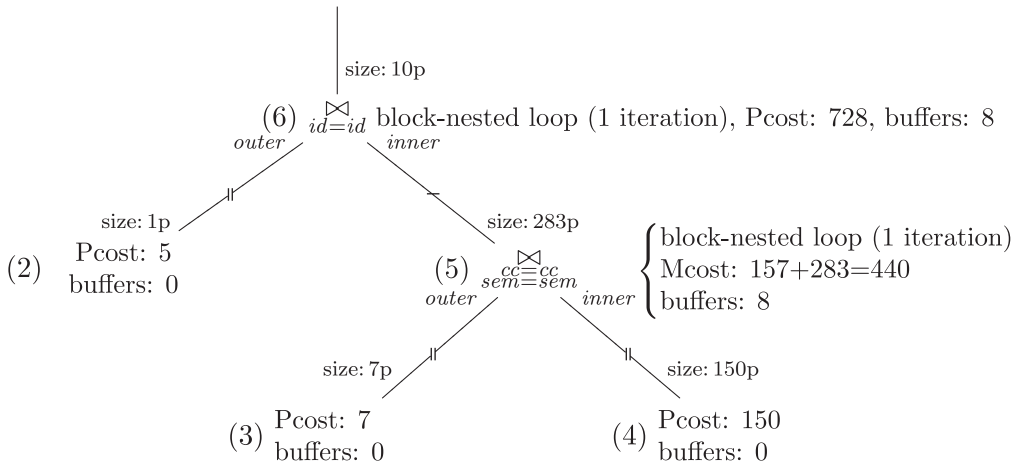
Verification:

(1) For a noop, we have $B = 0$ and $C = 0$, and pipelining the result is not allowed.
This node is a leaf node, so:
$B_{(1)} = B = 0.$
$M_{(1)} = 0.$

(2) For a sequential scan, we have $B = 0$ and $C = I_1$.
For this node, child 1 is (1), and $\mathit{piped}_{(2)} = \varnothing$ and $\mathit{stored}_{(2)} = \{(1)\}$, so:

$$\begin{aligned}
B_{(2)} &= \max\{B,\ B_{(1)}\} \;=\; \max\{0, 0\} \;=\; 0.\\
P_{(2)} &= C[I_1 := F_{(1)}] + M_{(1)}\\
&= F_{(1)} + M_{(1)}\\
&= 5 + 0\\
&= 5.
\end{aligned}$$

Example 3. A block-nested loop with one iteration at both node (5) (using 8 internal buffers) and node (6) (using 2 internal buffers). The result of (5) is stored on disk so that the total number of buffers simultaneously used during the production, including the output buffer, does not exceed 10 (in contrast to forthcoming cheaper plan in Example 4). Here is the plan:



Verification:

(5) For a block-nested loop with N=7 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, child *outer* is (3) and *inner* is (4), and $piped_{(5)} = \{(3), (4)\}$ and $stored_{(5)} = \varnothing$, so:

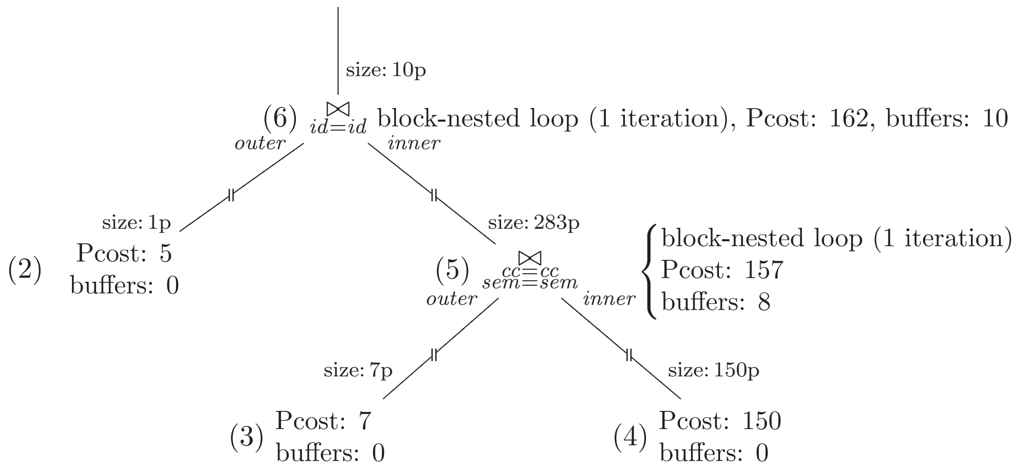
$$\begin{aligned} B_{(5)} &= \max\{B + B_{(3)} + B_{(4)}\} = (7+1)+0+0 = 8. \\ M_{(5)} &= C[I_1 := P_{(3)},\, I_2 := P_{(4)}] + F_{(5)} \\ &= (P_{(3)} + \lceil F_{(3)}/7 \rceil \times P_{(4)}) + F_{(5)} \\ &= (7 + \lceil 7/7 \rceil \times 150) + 283 \\ &= 440. \end{aligned}$$

Note. Pipelining (3) into (5) means that the output buffer of (3) is identified successively with the seven left-input buffers of (5).

(6) For a block-nested loop with N=1 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, child *outer* is (2) and *inner* is (5), and $piped = \{(2)\}$ and $stored = \{(5)\}$, so:

$$\begin{aligned} B_{(6)} &= \max\{B + B_{(2)},\ B_{(5)}\} = \max\{(N+1)+0,\ 8\} = \max\{2, 8\} = 8. \\ P_{(6)} &= C[I_1 := P_{(2)},\, I_2 := F_{(5)}] + M_{(5)} \\ &= (P_{(2)} + \lceil F_{(2)}/N \rceil \times F_{(5)}) + M_{(5)} \\ &= (5 + \lceil 1/1 \rceil \times 283) + 440 \\ &= 728. \end{aligned}$$

Example 4. In the previous plan the access path at node (5) uses 8 buffers, and this node can only be pipelined to (6) (with a decrease of 2×283 in the cost) if the number of available buffers *including the overall output buffer* is at least 11. Here is the plan:



size: 10p
(6) $\bowtie_{id=id}$ block-nested loop (1 iteration), Pcost: 162, buffers: 10
outer
inner
size: 1p
(2) Pcost: 5 buffers: 0
size: 283p
(5) $\bowtie_{cc=cc, sem=sem}$
block-nested loop (1 iteration) Pcost: 157 buffers: 8
outer
inner
size: 7p
size: 150p
(3) Pcost: 7 buffers: 0
(4) Pcost: 150 buffers: 0

Verification:

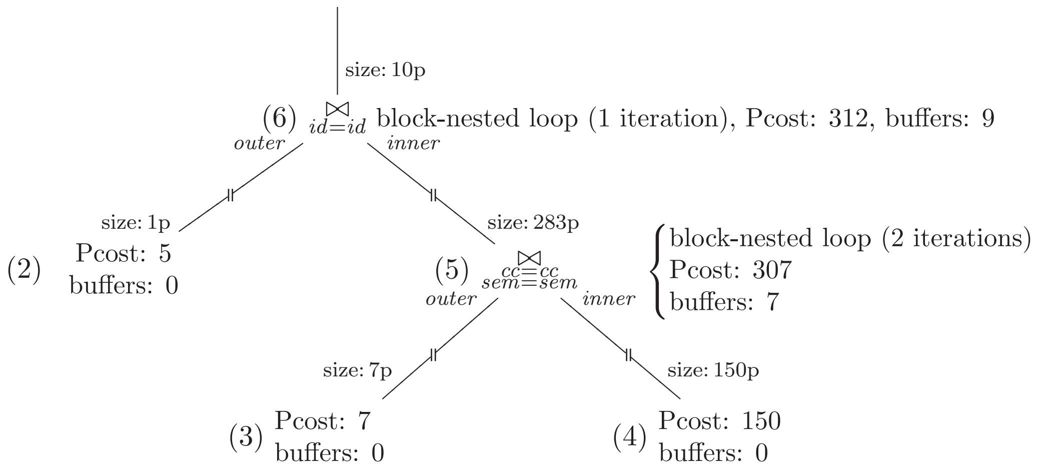
(5) For a block-nested loop with N=7 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, *outer* is (3) and *inner* is (4), and $piped_{(5)} = \{(3), (4)\}$ and $stored_{(5)} = \varnothing$, so:

$$\begin{array}{lcl} B_{(5)} & = & \max\{B + B_{(3)} + B_{(4)}\} \;=\; (7+1)+0+0 \;=\; 8. \\ P_{(5)} & = & C[I_1 := P_{(3)},\, I_2 := P_{(4)}] \\ & = & P_{(3)} + \lceil F_{(3)}/N \rceil \times P_{(4)} \\ & = & 7 + \lceil 7/7 \rceil \times 150 \\ & = & 157. \end{array}$$

(6) For a block-nested loop with N=1 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, *outer* is (2) and *inner* is (5), and $piped_{(6)} = \{(2), (5)\}$ and $stored_{(6)} = \varnothing$, so:

$$\begin{array}{lcl} B_{(6)} & = & \max\{B + B_{(2)} + B_{(5)}\} \;=\; (N+1)+0+8 \;=\; 10. \\ P_{(6)} & = & C[I_1{:=}P_{(2)},\, I_2{:=}P_{(5)}] \\ & = & P_{(2)} + \lceil F_{(2)}/N \rceil \times P_{(5)} \\ & = & 5 + \lceil 1/1 \rceil \times 157 \\ & = & 162. \end{array}$$

Example 5. If the number of available buffers *including the overall output buffer* is 10, and we still want to pipeline (5) to (6), then the block-nested loop of (5) can use only 7 buffers which is too few in order to do its work in one iteration:


size: 10p
(6) $\bowtie_{id=id}$ block-nested loop (1 iteration), Pcost: 312, buffers: 9
outer
inner
size: 1p
(2) Pcost: 5 buffers: 0
size: 283p
(5) $\bowtie_{cc=cc, sem=sem}$
block-nested loop (2 iterations) Pcost: 307 buffers: 7
outer
inner
size: 7p
size: 150p
(3) Pcost: 7 buffers: 0
(4) Pcost: 150 buffers: 0

Verification:

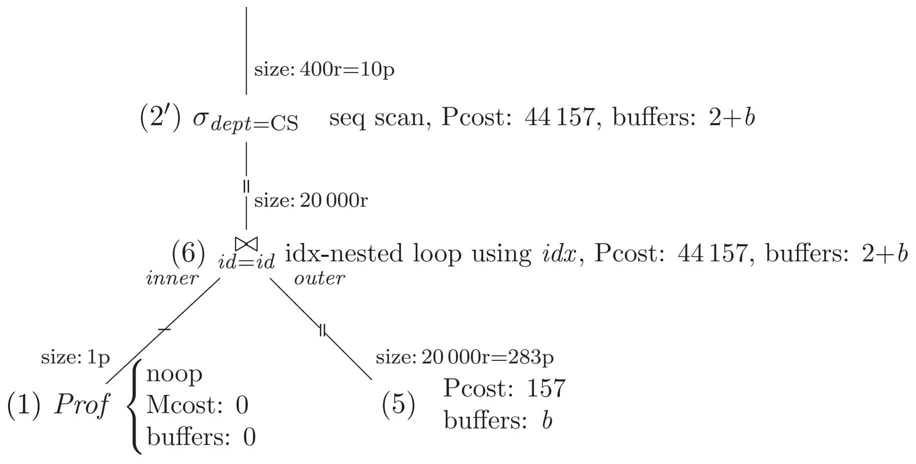
(5) For a block-nested loop with N=6 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, child *outer* is (3) and *inner* is (4), and $piped_{(5)} = \{(3), (4)\}$ and $stored_{(5)} = \varnothing$, so:

$$\begin{aligned} B_{(5)} &= \max\{B + B_{(3)} + B_{(4)}\} = (6+1)+0+0 = 7. \\ P_{(5)} &= C[I_1{:=}P_{(3)},\ I_2{:=}P_{(4)}] \\ &= P_{(3)} + \lceil F_{(3)}/6 \rceil \times P_{(4)} \\ &= 7 + \lceil 7/6 \rceil \times 150 \\ &= 7 + 2 \times 150 \\ &= 307. \end{aligned}$$

(6) For a block-nested loop with N=1 buffers for inputing *outer*, we have $B = N + 1$ and $C = I_{outer} + \lceil F_{outer}/N \rceil \times I_{inner}$.
Here, child *outer* is (2) and *inner* is (5), and $piped_{(6)} = \{(2), (5)\}$ and $stored_{(6)} = \varnothing$, so:

$$\begin{aligned} B_{(6)} &= \max\{B + B_{(2)} + B_{(5)}\} = (1+1)+0+7 = 9. \\ P_{(6)} &= C[I_1{:=}P_{(2)},\ I_2{:=}P_{(5)}] \\ &= P_{(2)} + \lceil F_{(2)}/1 \rceil \times P_{(5)} \\ &= 5 + \lceil 1/1 \rceil \times 307 \\ &= 5 + 1 \times 307 \\ &= 312. \end{aligned}$$

Example 6. Here is a quite different plan: an *index-nested loop* for node (6), taking node (5) as *outer* thus retrieving rows of node (1) by index look-up using a hash index *idx* on *Prof* for search key (*id*). Note that in fact we are changing the tree, because the index is not defined on the result of $\sigma_{dept=\mathrm{CS}} Prof$ but only on *Prof* itself; the selection is pushed up to work on the result of node (6). Pipelining (6) into the new selection node (2′) means that *still* the selection itself is for free! Here is the plan:


size: 400r=10p
(2′) $\sigma_{dept=\text{CS}}$ seq scan, Pcost: 44 157, buffers: 2+b
size: 20 000r
(6) $\bowtie_{id=id}$ idx-nested loop using *idx*, Pcost: 44 157, buffers: 2+b
inner
outer
size: 1p
size: 20 000r=283p
(1) *Prof*
noop
Mcost: 0
buffers: 0
(5)
Pcost: 157
buffers: b

Verification:

(1) For a noop, we have $B = 0$ and $C = 0$, and pipelining the result is not allowed.
This node is a leaf node, so:
$B_{(1)} = B = 0.$
$M_{(1)} = 0.$

(5) See Example 1 with $b = 2$, or Example 4 with $b = 8$.

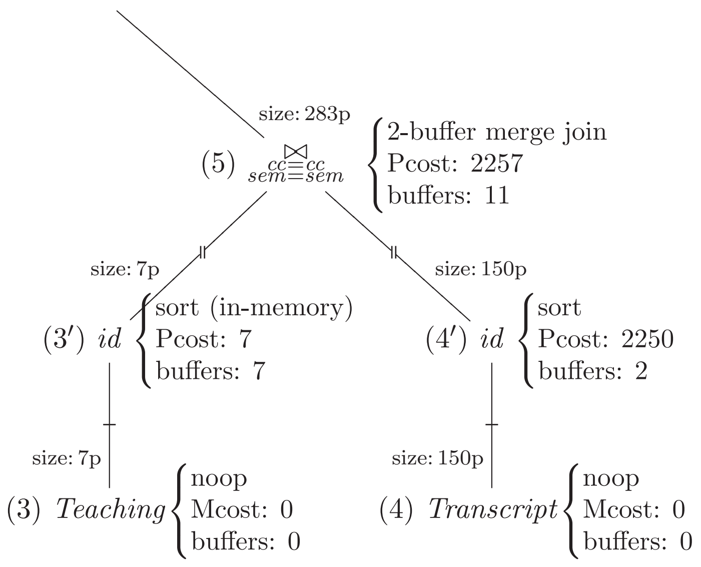
(6) For an index-nested loop we have $B = 2$ and $C = I_{outer} + T_{outer} \times (\rho + \mu)$.
Here, child *outer* is (5), *inner* is (1), and $piped_{(6)} = \{(5)\}$, and $stored_{(6)} = \{(1)\}$.
The index is a hash, so $\rho = 1.2$; the index is unclustered and on average just one *Prof* tuple matches a given tuple in the result of (5), so $\mu = 1$.

$$\begin{aligned}
B_{(6)} &= \max\{B + B_{(5)},\ B_{(1)}\} \;=\; \max\{2 + b,\ 0\} \;=\; 2{+}b.\\
P_{(6)} &= C[I_1{:=}F_{(1)},\ I_2{:=}P_{(5)}] + M_{(1)}\\
&= P_{(5)} + T_{(5)} \times (1.2 + 1) + 0\\
&= 157 + 20\,000 \times 2.2\\
&= 44\,157.
\end{aligned}$$

(2′) For a sequential scan, we have $B = 0$ and $C = I_1$.
Here, child 1 is (6), and $piped_{(2')} = \{(6)\}$ and $stored_{(2')} = \varnothing$, so:

$$\begin{aligned}
B_{(2')} &= \max\{B + B_{(6)}\} \;=\; 0 + 2{+}b \;=\; 2{+}b.\\
P_{(2')} &= C[I_1 := P_{(6)}] \;=\; P_{(6)} \;=\; 44\,157.
\end{aligned}$$

Example 7. A 2-buffer merge join at node (5) and explicit sort nodes for the arguments; the presence of explicit sort nodes is actually a change of the tree. We aim at sorting *Teaching* in main memory, while at the same time keeping the overall number of buffers as low as possible. Here is the plan:



Verification:

(3′) For sorting with N buffers we have $B = N$ and $C = I_1 + 2F_1\lceil\log_N F_1\rceil - 2F_1$.
Here, child 1 is (3), and $\mathit{piped}_{(3')} = \varnothing$ and $\mathit{stored}_{(3')} = \{(3)\}$.
Take $N = F_{(3)} = 7$, so that sorting can be done in main memory.

$$\begin{aligned}
B_{(3')} &= \max\{B,\ B_{(3)}\} \ = \ \max\{N, 0\} \ = \ N \ = \ 7.\\
P_{(3')} &= C[I_1{:=}F_{(3)}] + M_{(3)}\\
&= F_{(3)} + 2F_{(3)}\lceil\log_N F_{(3)}\rceil - 2F_{(3)} + M_{(3)}\\
&= 7 + 2\times 7\times\lceil\log_7 7\rceil - 2\times 7 + 0\\
&= 7.
\end{aligned}$$

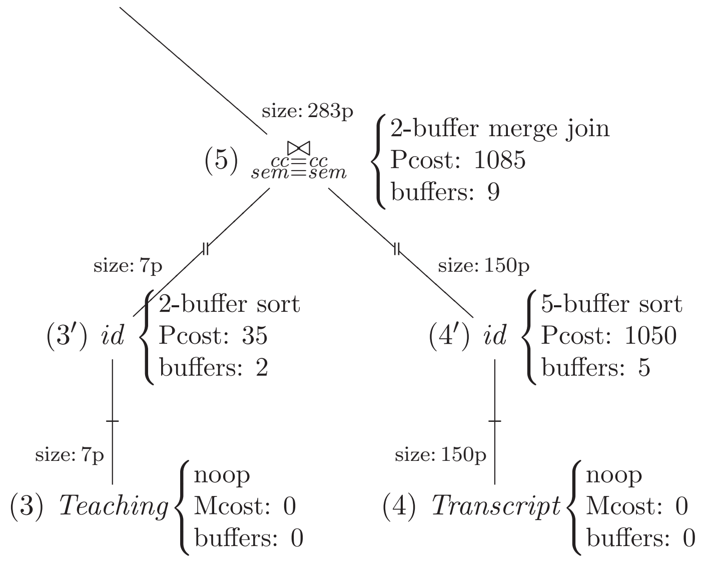
(4′) For sorting with N buffers we have $B = N$ and $C = I_1 + 2F_1\lceil\log_N F_1\rceil - 2F_1$.
Here, child 1 is (4), $\mathit{piped}_{(4')} = \varnothing$ and $\mathit{stored}_{(4')} = \{(4)\}$.
Take, rather arbitrarily, N as small as possible: $N = 2$.

$$\begin{aligned}
B_{(4')} &= \max\{B,\ B_{(4)}\} \ = \ \max\{N, 0\} \ = \ N \ = \ 2.\\
P_{(4')} &= C[I_1{:=}F_{(4)}] + M_{(4)}\\
&= F_{(4)} + 2F_{(4)}\lceil\log_2 F_{(4)}\rceil - 2F_{(4)} + M_{(4)}\\
&= 150 + 2\times 150\times\lceil\log_2 150\rceil - 2\times 150 + 0\\
&= 150 + 2\times 150\times \quad 8 \quad - 2\times 150\\
&= 2250.
\end{aligned}$$

(5) For a merge join with two buffers, we have $B = 2$ and $C = I_1 + I_2$.
Here, child 1 is (3′), child 2 is (4′), and $\mathit{piped}_{(5)} = \{(3'), (4')\}$ and $\mathit{stored}_{(5)} = \varnothing$, so:

$$\begin{aligned}
B_{(5)} &= \max\{B + B_{(3')} + B_{(4')}\} \ = \ 2 + 7 + 2 \ = \ 11.\\
P_{(5)} &= C[I_1 := P_{(3')},\ I_2 := P_{(4')}]\\
&= 7 + 2250\\
&= 2257.
\end{aligned}$$

Example 8. A 2-buffer merge join at node (5) and explicit sort nodes for the arguments; the presence of explicit sort nodes is actually a change of the tree. We aim at keeping the overall number of buffers, including the output buffer, at most 10, and reserve as much buffers as possible for the larger table: *Transcript*. Here is the plan:



Verification:

(3′) For sorting with N buffers we have $B = N$ and $C = I_1 + 2F_1\lceil\log_N F_1\rceil - 2F_1$.
Here, child 1 is (3), and $\mathit{piped}_{(3')} = \varnothing$ and $\mathit{stored}_{(3')} = \{(3)\}$.
Take N as small as possible, $N = 2$, so as to have as many buffers as possible for (4′).

$$\begin{aligned}
B_{(3')} &= \max\{B, B_{(3)}\} = \max\{N, 0\} = N = 2.\\
P_{(3')} &= C[I_1{:=}F_{(3)}] + M_{(3)}\\
&= F_{(3)} + 2F_{(3)}\lceil\log_N F_{(3)}\rceil - 2F_{(3)} + M_{(3)}\\
&= 7 + 2\times 7\times\lceil\log_2 7\rceil - 2\times 7 + 0\\
&= 35.
\end{aligned}$$

(4′) For sorting with N buffers we have $B = N$ and $C = I_1 + 2F_1\lceil\log_N F_1\rceil - 2F_1$.
Here, child 1 is (4), and $\mathit{piped}_{(4')} = \varnothing$ and $\mathit{stored}_{(4')} = \{(4)\}$.
Take N as large as possible in view of the *assumed* overall number of buffers: $N = 5$.

$$\begin{aligned}
B_{(4')} &= \max\{B, B_{\{4\}}\} = \max\{N, 0\} = N = 5.\\
P_{(4')} &= C[I_1{:=}F_{(4)}] + M_{(4)}\\
&= F_{(4)} + 2F_{(4)}\lceil\log_N F_{(4)}\rceil - 2F_{(4)} + M_{(4)}\\
&= 150 + 2\times 150\times\lceil\log_5 150\rceil - 2\times 150 + 0\\
&= 150 + 2\times 150\times 4 - 2\times 150\\
&= 1050.
\end{aligned}$$

(5) For a 2-buffer merge join, we have $B = 2$ and $C = I_1 + I_2$.
Here, child 1 is (3′), child 2 is (4′), and $\mathit{piped}_{(5)} = \{(3'), (4')\}$ and $\mathit{stored}_{(5)} = \varnothing$, so:

$$\begin{aligned}
B_{(5)} &= \max\{B + B_{(3)} + B_{(4)}\} = 2 + 2 + 5 = 9.\\
P_{(5)} &= C[I_1 := P_{(3')}, I_2 := P_{(4')}]\\
&= P_{(3')} + P_{(4')}\\
&= 35 + 1050\\
&= 1085.
\end{aligned}$$

References

[1] Michael Kifer, Arthur Bernstein, and Philip M. Lewis. *Database Systems: An Application-Oriented Approach, Complete Version, 2nd edition.* Addison-Wesley, 2006.