A Foundation for the Replacement of Pipelined Physical Join Operators in Adaptive Query Processing

Kwanchai Eurviriyanukul, Alvaro A.A. Fernandes, and Norman W. Paton

University of Manchester, Manchester M1 9PL, United Kingdom
{eurvirik,alvaro,norm}@cs.man.ac.uk

Abstract. Adaptive query processors make decisions as to the most effective evaluation strategy for a query based on feedback received while the query is being evaluated. In essence, any of the decisions made by the optimizer (e.g., on operator order or on which operators to use) may be revisited in an adaptive query processor. This paper focuses on changes to physical operators (e.g., the specific join operators used, such as hash-join or merge-join) in pipelined query evaluators. In so doing, the paper characterizes the runtime properties of pipelined operators in a way that makes explicit when specific operators may be replaced, and that allows the validity of operator replacements to be proved. This is illustrated with reference to the substitution of join operators during their evaluation.

1 Introduction

The execution plan of a query describes how the query is to be evaluated. The plan makes explicit the decisions made by the query optimizer, e.g., with respect to the order of evaluation of operators, the algorithms and auxiliary data structures to be used, the allocation of plan fragments to resources, and the level of partitioned parallelism. An adaptive query processor (e.g., [5]) may revise any of these different kinds of decision at query runtime.

Many proposals have been made for adaptive query processing techniques (see [2] for a recent survey). However, few of the proposals provide a formal characterization of the adaptations undertaken, and thus the validity of the runtime changes proposed is rarely addressed in a rigorous manner. We cannot point to any cases in which published adaptive strategies have subsequently been shown to produce incorrect results, but contend that certain forms of adaptation may benefit from a more formal approach. We observe that adaptive strategies may be associated with complex protocols for halting, revising and resuming execution plans (e.g., [14]), and that certain categories of runtime change may only be fully explored when the safety net of a formal foundation is in place. One such category, which is the focus of this paper, is the replacement of operators in pipelined query plans.

In pipelined plans, which have been shown to be effective for increasing resource usage and reducing response times in parallel and distributed settings, many operators may be being evaluated simultaneously. As such, if a particular operator is ineffective in a specific context, the replacement of that operator without disrupting its suppliers or consumers may provide effective and focused adaptation. However, during evaluation an operator may maintain internal data structures, and at any point in time may have partially processed some of its inputs. This paper presents an approach to the description of partially-evaluated operators that makes explicit the issues associated with in-flight operator replacement, and enables the validity of specific transformations to be proved. This is illustrated with reference to the substitution of physical join operators.

We observe that operator replacement for pipelined evaluation has not been extensively investigated to date. Several strategies that may lead to changes in the physical operators used by a plan do not adapt during operator evaluation (e.g., [15, 9, 8]). POP [10] explores several approaches to adaptation that materialize the results of complete sub-plans, but when an operator is replaced during its evaluation, the replacement operator starts evaluating from scratch, thus repeating work that was done by its predecessor. Rio [3] tests the suitability of an operator at a place in a plan by sampling and caching its inputs, and, like POP, when an algorithm is replaced it is rerun from scratch over its input buffers. This paper complements existing work by investigating finer-grained operator replacement and by providing a formal characterization of the changes made.

The remainder of the paper is structured as follows. Section 2 describes the technical context for the material that follows. Section 3 provides a notation for describing partially evaluated operators, characterizes the states in which an operator can safely be replaced, and illustrates the overall approach by considering the replacement of physical join operators, including a proof of validity for an example transformation. Section 4 presents some conclusions.

2 Technical Context

This section defines some terms and notions that are used later in the paper.

A query plan P can be represented by a tree consisting of a set of query operator nodes and a set of edges representing data that flows from child nodes to parent nodes. Given a node N, P^N denotes the sub-plan of P rooted at N. Given a query plan P, $[\![P]\!]$ denotes the result obtained by evaluating P.

In this paper, the nodes of a plan are considered to be drawn from a physical algebra, i.e., one in which the node identifies the algorithm to be used. Examples are presented later in the paper of adaptations where one physical join operator replaces another. Five representative join algorithms, viz., hash join, nested-loop join, merge join, index nested-loop join and symmetric hash join are considered. An equality join condition and bag semantics are assumed throughout. Assume that R is the left, and S the right, input of a join. Let $\overset{H}{\bowtie}$, $\overset{NL}{\bowtie}$, $\overset{M}{\bowtie}$, $\overset{IN}{\bowtie}$ and $\overset{SH}{\bowtie}$, respectively, denote the above physical join operators, the algorithms for which are described briefly below:

- **Hash Join** [7]: All R-tuples are read and stored in a hash table, indexed on the join attribute(s). Then, each S-tuple is read in turn and used to probe the hash table. Any matching R-tuples are retrieved.
- **Nested-Loop Join** [11]: Given inputs sorted on the join attribute(s), the tuples are read from either R or S in turn and to find which ones is S or R (respectively) match.
- **Merge Join** [11]: Given inputs sorted on the join attribute(s), the tuples are read from either R or S in turn and to find which ones is S or R (respectively) match.
- **Index Nested-Loop Join** [4]: Each R-tuple is read in turn and its joint attribute(s) are then used to search an index on S and retrieve the tuples that match.
- **Symmetric Hash Join (Pipelined Hash Join)** [16]: Each tuple from either R or S is read in turn and is both stored in the hash table for R (or S, respectively) and used to to probe the hash table for S (or R, respectively). Any matching R- (or S-) tuples (respectively) are retrieved.

In this paper, pipelined evaluation is assumed to be implemented using the the iterator model [7], which has three principal functions: Open, Next and Close. The Open function prepares the operator for result production. The Next function produces the results one at a time, and the Close function performs cleaning up. When combined with communication operators such as *exchange* [6], the iterator model supports pipelined parallelism.

A state-transition diagram can be used to capture the evaluation trace of operators implemented using the iterator model. These states are labelled as $\mathbb{I}, \mathbb{O}, \mathbb{O}', \mathbb{N}, \mathbb{N}', \mathbb{C}$ and $\mathbb{C}'$ in Figure 1.

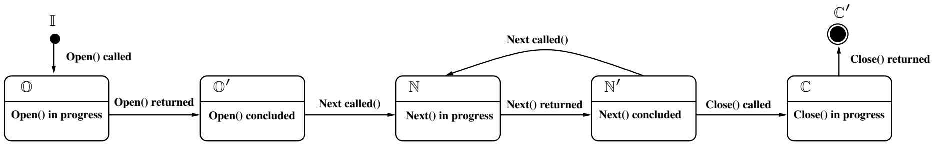

Fig. 1. The state-transition diagram of an iterator.

In this paper, the emphasis is on adapting query plans in the state $\mathbb{N}'$, as other states are either in-progress states or else are only reached before or after the operator as a whole has been evaluated. In state $\mathbb{N}'$, a call to the Next function has been evaluated and its result returned. If the operator has not returned all its results, the Next function will be called again, and the operator returns to state $\mathbb{N}$. On the other hand, if the operator has returned all its results, the Close function is called and the operator moves to state $\mathbb{C}$ and then $\mathbb{C}'$.

3 Replacing Physical Operators

This section presents an approach to the description and validation of operator replacement for pipelined query plans. The following issues are addressed:

1. The provision of a notation for describing partially evaluated query plans.
2. The identification of points during operator evaluation, referred to as *quiescent* states, in which the results produced by an operator in state $\mathbb{N}'$ can be defined precisely in terms of the inputs read by the operator up to then.
3. The description, for different physical operators, of the data produced and the results that remain to be produced, at quiescent states.
4. The provision of an approach to proving that specific operator replacements are result-neutral, i.e., that they have no effect on the result that is output.

3.1 A Notation for Partially Evaluated Query Plans

Expressions in logical or physical algebras, such as $(R \bowtie S)$, describe query plans, but provide no way of describing the runtime properties of the plan, such as the data produced by an operator at a point in time. To describe not only the plan, but also its evaluation status, some additional notation is introduced.

Given a sub-plan P^N, let I be a child node of N.

- $I^+_{[N:\mathbb{S}]}$ is the portion of $[\![I]\!]$ that has been returned by a previously made call to the Next() function of I when node N is in state $\mathbb{S}$, where $\mathbb{S} \in \{\mathbb{I}, \mathbb{O}, \mathbb{O}', \mathbb{N}, \mathbb{N}', \mathbb{C}, \mathbb{C}'\}$. Therefore, $I^+_{[N:\mathbb{S}]} \subseteq [\![I]\!]$.
- $I^-_{[N:\mathbb{S}]}$ is the portion of $[\![\mathrm{I}]\!]$ that has yet to be returned to N by subsequent calls to the Next() function of I. As a result, $I = (I^+_{[N:\mathbb{S}]} \cup I^-_{[N:\mathbb{S}]})$.
- $last(I^+_{[N:\mathbb{S}]})$ is the set containing the last tuple added into $[\![I^+_{[N:\mathbb{S}]}]\!]$.

3.2 Quiescent States

Section 3.1 provided a notation for describing the state of the inputs to an operator during operator evaluation. However, the relationship between the input read by an operator and the output produced by an operator at a point in time may be different at different points in the trace, i.e., at different occurrences of $\mathbb{N}'$. Thus, during the evaluation of a hash join $R\overset{H}{\bowtie}S$, the operator is in state $\mathbb{N}'$, and the inputs read by the algorithm are $R^+_{[N:\mathbb{N}']}$ and $S^+_{[N:\mathbb{N}']}$, where N is the relevant instance of $\overset{H}{\bowtie}$ in the plan. Assuming that the left operand is used to populate the hash table, $R^+_{[N:\mathbb{N}']} = [\![R]\!]$. The tuples produced so far by the algorithm may not be denoted by $R^+_{[N:\mathbb{N}']} \bowtie S^+_{[N:\mathbb{N}']}$, because the last tuple read from S may join with many tuples in R. As such, the tuples produced by the operator will only be $R^+_{[N:\mathbb{N}']} \bowtie S^+_{[N:\mathbb{N}']}$ if $last(S^+_{[N:\mathbb{N}']})$ has been joined with every matching tuple in R.

A *quiescent* state for an operator is one in which the result produced by the operator in state $\mathbb{N}'$ can be precisely defined in terms the input to the operator at that point in time. The test as to whether or not an operator is in a quiescent state is operator-specific. To make possible a quiescence test, we extend the interface to an operator with an isQuiescent function, which determines from the internal state of the operator whether or not it is in a quiescent state.

Algorithm 3.1: Nested-Loop Join(*Operator* R, *Operator* S)

```
comment: R and S are the outer and the inner inputs, respectively
Tuple r, s, eof        comment: the state of the operator

boolean procedure Open()
  if (R.Open() and S.Open())
    then { r ← R.Next()    comment: set up the outer loop
         { return ( true )
    else return ( false )

Tuple procedure Next()
  while ( true )
    { s ← S.Next()
    { if (s ≠ nil and r ≠ nil)
    {   then { if (r.joinAtt() = s.joinAtt())
    {        {   then return (concat(r,s))
    {        {   else continue
    {   else { comment: no s ∈ S to match with r ∈ R
    {        { r ← R.Next()
    {        { if (r ≠ nil)
    {        {   then { comment: restart S
    {        {        { S.Close()
    {        {        { S.Open()
    {        {   else { comment: R was consumed
    {        {        { break ;
  return (eof ← nil)

boolean procedure Close()
  if (R.Close() and S.Close())
    then return ( true )
    else return ( false )

boolean procedure hasNext()
  return (eof ≠ nil)

boolean procedure IsQuiescent()
  comment: returns  true if the inner input been consumed
  return (¬(S.hasNext()))
```

The following are characterizations of the quiescent states for the example join operators, assuming that R is the left, and S the right, input of a join.

- **Hash Join**: The last S-tuple read has been joined with all matching R-tuples in the hash table.
- **Nested-Loop Join**: The last R-tuple read has been joined with all matching S-tuples.
- **Merge Join**: The last R-tuple read has been joined with all matching S-tuples and the join attribute value(s) of the next R-tuple is different from that of the last R-tuple read.
- **Index Nested-Loop Join**: The last R-tuple read has been joined with all matching S-tuples that were retrieved by a lookup on an index on S.
- **Symmetric Hash Join**: The last tuple read from either R or S has been joined with all matching tuples in the hash table for S or R, respectively.

Algorithm 3.1 defines NESTED-LOOP JOIN(R,S) and formalizes the quiescent-state test for this algorithm, viz., that a state is quiescent if there are no more S-tuples to read (in that pass).

Related notions include *moments of symmetry*, from the work on eddies, which determine when the order of the inputs to a join can be changed [1]; this is a narrower notion than that of a quiescent state, as it defines conditions for a specific adaptation. In [12], operators are classified, with respect to their ability to participate in adaptations, on the basis of properties shared by groups of algorithms (e.g., that they have fixed memory consumption); here, quiescent states are used not so much to identify different forms of algorithm as to support the algebraic-level description of operator states, as described in Section 3.3.

3.3 Describing Partial Results

When an operator is in a *quiescent* state, it is possible to define its result precisely in terms of the data it has consumed. As a consequence, it is also possible to define precisely the portion of the result that remains to be produced.

Table 1 describes both the intermediate results produced by the different operators at quiescent states and the corresponding portion of the result that has yet to be returned. As an example, for hash join with operands R and S, at a quiescent state, every tuple that has been read into the hash table (i.e., $R^{+}_{[N:\mathbb{N}']} = R$) has been joined with every tuple that has been read from the other operand (i.e., $S^{+}_{[N:\mathbb{N}']}$). To complete the evaluation, every tuple in $R^{+}_{[N:\mathbb{N}']} = R$ needs to be joined with the tuples that have yet to be read from S (i.e., $S^{-}_{[N:\mathbb{N}']}$). A similar justification lies behind the other entries in Table 1.

When one join operator is to be replaced with another operator at a quiescent state, the rightmost column in Table 1 describes the work that remains to be done by the new operator. Section 3.4 describes how the validity of the entries in Table 1 can be proved.

Table 1. The intermediate result and the remainder of $R \bowtie S$ at a quiescent $\mathbb{N}'$

Physical Join Operator ($\overset{X}{\bowtie}$)	Intermediate Result	Remainder
Hash Join ($\overset{H}{\bowtie}$)	$\llbracket R^{+}_{[\overset{H}{\bowtie}:\mathbb{N}']} \bowtie S^{+}_{[\overset{H}{\bowtie}:\mathbb{N}']} \rrbracket$	$\llbracket R^{+}_{[\overset{H}{\bowtie}:\mathbb{N}']} \bowtie S^{-}_{[\overset{H}{\bowtie}:\mathbb{N}']} \rrbracket$
Nested-Loop Join ($\overset{NL}{\bowtie}$)	$\llbracket R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket$	$\llbracket R^{-}_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket$
Merge Join ($\overset{M}{\bowtie}$)	$\llbracket (R^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']} - last(R^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']}))$ $\bowtie (S^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']} - last(S^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']})) \rrbracket$	$\llbracket (last(R^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']}) \cup R^{-}_{[\overset{M}{\bowtie}:\mathbb{N}']})$ $\bowtie (last(S^{+}_{[\overset{M}{\bowtie}:\mathbb{N}']}) \cup S^{-}_{[\overset{M}{\bowtie}:\mathbb{N}']}) \rrbracket$
Index Nested-Loop Join ($\overset{IN}{\bowtie}$)	$\llbracket R^{+}_{[\overset{IN}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket$	$\llbracket R^{-}_{[\overset{IN}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket$
Symmetric Hash Join ($\overset{SH}{\bowtie}$)	$\llbracket R^{+}_{[\overset{SH}{\bowtie}:\mathbb{N}']} \bowtie S^{+}_{[\overset{SH}{\bowtie}:\mathbb{N}']} \rrbracket$	$\llbracket (R^{+}_{[\overset{SH}{\bowtie}:\mathbb{N}']} \bowtie S^{-}_{[\overset{SH}{\bowtie}:\mathbb{N}']})$ $\cup (R^{-}_{[\overset{SH}{\bowtie}:\mathbb{N}']} \bowtie S^{+}_{[\overset{SH}{\bowtie}:\mathbb{N}']})$ $\cup (R^{-}_{[\overset{SH}{\bowtie}:\mathbb{N}']} \bowtie S^{-}_{[\overset{SH}{\bowtie}:\mathbb{N}']}) \rrbracket$

3.4 Replacing Operators

To compute the remainder of the result, an adaptive system may choose to replace an operator in a plan with one that it is predicted will perform better in a specific setting. For example, a nested-loop join may have been selected by the optimizer based on inaccurate predictions for the cardinalities of the inputs to a join; if in practice the cardinalities used by the optimizer are revealed to be underestimates, it may be appropriate to migrate to a hash join instead. Alternatively, a hash join may have been assigned on the assumption that the selectivity of the right hand operand was quite high; if it turns out to be low, it may be more effective to complete the evaluation using an index nested-loop join. There could also be resource restrictions that, e.g., lead to a hash table within a join exceeding the available memory, which in turn leads to options being explored such as changing to a join algorithm that uses less memory, e.g., nested-loop join.

In essence, with reference to Table 1, an operator in a quiescent state can be replaced by any other operators if the remainder of the result can be computed. The complete result is then the union of that produced by the original operator with that produced by the replacement operators; this union may not be carried out explicitly, as replacement operators may simply be planted within a suspended plan, which then resumes evaluation. The suspension and resumption of plans has been discussed in the literature (e.g., [14]).

Using the notation from Section 3.3, the following rule could be used to indicate that a nested loop join can be replaced during its evaluation at a quiescent state by a hash join, where the value to be computed by the hash join is that described in the *Remainder* column in Table 1. For any quiescent $[\overset{NL}{\bowtie} : \mathbb{N}']$:

$$[\![R \bowtie S]\!] = [\![R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S]\!] \cup [\![R^{-}_{[\overset{NL}{\bowtie}:\mathbb{N}']} \overset{H}{\bowtie} S]\!] \tag{1}$$

The decision as to whether or not a specific operator replacement is appropriate in a context could be made with reference to a cost model that compares the cost of completing the existing plan with the cost of changing from one plan to another plus the cost for the evaluation of the replacement plan.

The next step is to prove that an operator replacement is result-neutral. To do so, there are two proof obligations:

1. To show that the value of intermediate the join result from Table 1 in the quiescent state $\mathbb{N}'$ is correct.
2. To show that the union of the value produced by the original plan and the value computed by the transformed plan provides a correct result for the query.

Such proofs must be provided on a case-by-case basis, reflecting the fact that different algorithms have different quiescent states, which leave different amounts of work to be carried out by other algorithms. Due to limited space, proofs are only provided for the replacement of a nested-Loop join with a hash join operator. Similar proofs for other physical join operators have been conducted in an analogous manner but space constraints preclude their presentation here.

Proof by induction shows that, for a nested-loop join operator in a quiescent state $\mathbb{N}'$, the value of the intermediate join result is as stated in Table 1. Given a sub-plan $P^{\overset{NL}{\bowtie}}$, for all integers $i \geq 1$, let $\mathbb{N}'(i)$ represent the i-th occurrence of the quiescent state in the execution trace of this operator's evaluation.

Theorem 1 (Value of the Intermediate Result for Nested-Loop Join).

$$[\![P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(i)]}]\!] = [\![R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(i)]} \bowtie S]\!] \tag{2}$$

Proof.

Basis step: *If $i = 1$, then (2) becomes:*

$$[\![P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(1)]}]\!] = [\![R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(1)]} \bowtie S]\!] \tag{3}$$

Let the first matching tuple read from R be t_{r_1}. Then, $R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(1)]} = \{t_{r_1}\}$. Substituting $R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(1)]}$ in (3) yields:

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(1)]}\rrbracket = \llbracket \{t_{r_1}\} \bowtie S \rrbracket \tag{4}$$

which holds, which holds, because given that the quiescent-state test for nested-loop join in Section 3.2 is satisfied in the first quiescent $\mathbb{N}'$, it follows that the first matching tuple from R has been joined with all matching tuples in S.

Induction step: *In a quiescent state $\mathbb{N}'$, for any integer $k \geq 1$, the induction hypothesis states that:*

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k)]}\rrbracket = \llbracket R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \bowtie S \rrbracket \tag{5}$$

Assuming (5), we prove that

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket = \llbracket R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]} \bowtie S \rrbracket \tag{6}$$

Firstly, following from the quiescent-state test for nested-loop join in Section 3.2, the value of a sub-plan in the subsequent quiescent state to a quiescent state k (i.e., $\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket$) equals the intermediate result at k (i.e., $\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k)]}\rrbracket$) unioned with the result of joining the most recently read tuple of R (i.e., $last(R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]})$), with the tuples in S:

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket = \llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k)]}\rrbracket \cup \llbracket last(R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]}) \bowtie S \rrbracket \tag{7}$$

Using the right-hand side of (5) to substitute $\llbracket P^{\overset{H}{\bowtie}}{}_{[\mathbb{N}'(k)]}\rrbracket$ in (7) gives:

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket = \llbracket R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \bowtie S \rrbracket \cup \llbracket last(R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]}) \bowtie S \rrbracket \tag{8}$$

Since the value of *symbol ($\llbracket\ \rrbracket$) denotes the result set of evaluating a plan fragment, two or more results can be unioned, so (8) becomes:*

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket = \llbracket (R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \bowtie S) \cup (last(R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]}) \bowtie S) \rrbracket \tag{9}$$

From (9), by the distributivity of join with respect to union, it follows that:

$$\llbracket P^{\overset{NL}{\bowtie}}{}_{[\mathbb{N}'(k+1)]}\rrbracket = \llbracket (R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \cup last(R^{+}_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]})) \bowtie S \rrbracket \tag{10}$$

By the definition of the quiescence condition in Section 3.2, it follows that:

$$last(R^+_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]}) \not\subseteq R^+_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \tag{11}$$

Therefore, the left operand of the join in the right-hand side of (10) becomes:

$$(R^+_{[\overset{NL}{\bowtie}:\mathbb{N}'(k)]} \cup last(R^+_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]})) = R^+_{[\overset{NL}{\bowtie}:\mathbb{N}'(k+1)]} \tag{12}$$

Substituting the right-hand side of (12) in (10) gives (6), as desired. □

We note that this proof is independent of the join operator that is to replace the nested-loop, and thus that the proof need not be repeated for different target operators. Next, we have to prove that changing from $\overset{NL}{\bowtie}$ to $\overset{H}{\bowtie}$ is result-neutral.

Given a sub-plan P^J, where J is quiescent in state $\mathbb{N}'$, we show that replacing J with J' is result-neutral when J is a nested-loop join and J' is a hash join.

Theorem 2 (Validity of Replacing Nested-Loop Join with Hash Join).

$$\llbracket P^J \rrbracket = \llbracket P^{J_{[\mathbb{N}']}} \rrbracket \cup \llbracket P^{J'} \rrbracket \tag{13}$$

Proof.

According to Table 1, we get $\llbracket P^J \rrbracket = \llbracket R\overset{NL}{\bowtie}S \rrbracket, \llbracket P^{J_{[\mathbb{N}']}} \rrbracket = \llbracket R^+_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket$ *and* $\llbracket P^{J'} \rrbracket = \llbracket R^-_{[\overset{NL}{\bowtie}:\mathbb{N}']} \overset{H}{\bowtie} S \rrbracket$. *Substituting into (13) gives:*

$$\llbracket R\overset{NL}{\bowtie}S \rrbracket = \llbracket R^+_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S \rrbracket \cup \llbracket R^-_{[\overset{NL}{\bowtie}:\mathbb{N}']} \overset{H}{\bowtie} S \rrbracket \tag{14}$$

Since the value of *symbol* ($\llbracket\ \rrbracket$) *represents the result of a plan fragment, two or more results can be unioned, so (14) becomes:*

$$\llbracket R\overset{NL}{\bowtie}S \rrbracket = \llbracket (R^+_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S) \cup (R^-_{[\overset{NL}{\bowtie}:\mathbb{N}']} \overset{H}{\bowtie} S) \rrbracket \tag{15}$$

Assuming the correctness of Nested-Loop Join *(i.e.,* $\overset{NL}{\bowtie}$*) and* Hash Join *(i.e.,* $\overset{H}{\bowtie}$*) in implementing the semantics of logical join operation (i.e.,* $\bowtie$*), (15) becomes:*

$$\llbracket R \bowtie S \rrbracket = \llbracket (R^+_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S) \cup (R^-_{[\overset{NL}{\bowtie}:\mathbb{N}']} \bowtie S) \rrbracket \tag{16}$$

From (16), by the distributivity of join with respect to union, it follows that:

$$\llbracket R \bowtie S \rrbracket = \llbracket (R^+_{[\overset{NL}{\bowtie}:\mathbb{N}']} \cup R^-_{[\overset{NL}{\bowtie}:\mathbb{N}']}) \bowtie S) \rrbracket \tag{17}$$

By the definitions of I^+ and I^- in Section 3.1, it follows that $(R^+ \cup R^-) = R$. Substituting into (17) gives:

$$[\![R \bowtie S]\!] = [\![R \bowtie S]\!] \tag{18}$$

Thereby establishing (13), i.e., that replacing J with J' is result-neutral when J is a nested-loop join and J' is a hash join. □

We observe that no re-computation for the initial segment of the join result is needed.

4 Conclusion

Adaptive query processing shows promise for improving the performance of query evaluation, especially in settings in which available statistical information may be unreliable or out-of-date.

This paper adds the following to existing results on adaptive query processing:

1. A notation for describing the properties of partially evaluated query plans; this notation enables systematic and precise description of transformations to query execution plans at runtime.
2. A demonstration of the use of the notation for describing changes to physical join operators within pipelined plans, including an example of how the validity of such transformations can be proved.

The notation and proof strategies contributed in this paper can be generalized to other pipelined physical operators. The paper is thus best thought of as contributing a framework for such formal analysis task.

The above results seek to contribute to ongoing work on adaptive databases by:

1. Encouraging the formal description of adaptations, thereby providing assurances as to the correctness of changes made to execution plans at runtime. Although a few adaptive strategies have received a formal treatment (e.g., [13] includes several proofs of properties of eddies and their extensions), such a practice does not seem to be widespread.
2. Providing a formal underpinning for the replacement of physical operators during their evaluation, thereby allowing finer-grained adaptations than have been supported by most previous work that changes plans at runtime (e.g., [15, 9, 8, 10]).

Acknowledgement K. Eurviriyanukul thanks Rajamangala University of Technology Lanna, Chiang Mai, Thailand, for their financial support.

References

1. R. Avnur and J.M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, pages 261–272, 2000.
2. S. Babu and P. Bizarro. Adaptive Query Processing in the Looking Glass. In *CIDR*, pages 238–249, 2005.
3. S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *Proc. ACM SIGMOD*, pages 107–118, 2005.
4. H. Garcia-Molina, J. Widom, and J.D. Ullman. *Database System Implementation*. Prentice-Hall, Inc., 1999.
5. A. Gounaris, J. Smith, N. W. Paton, R. Sakellariou, and A. A. A. Fernandes. Adapting to Changing Resources in Grid Query Processing. In *Proc. 1st International Workshop on Data Management in Grids*. Springer-Verlag, 2005.
6. G. Graefe. Encapsulation of Parallelism in the Volcano Query Processing System. In *Proc. SIGMOD*, pages 102–111, 1990.
7. G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.
8. Z.G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An Adaptive Query Execution System for Data Integration. In *SIGMOD Conference*, pages 299–310, 1999.
9. N. Kabra and D. J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In *SIGMOD Conference*, pages 106–117, 1998.
10. V. Markl, V. Raman, D. E. Simmen, G. M. Lohman, and H. Pirahesh. Robust Query Processing through Progressive Optimization. In *SIGMOD Conference*, pages 659–670, 2004.
11. P. Mishra and M. H. Eich. Join Processing in Relational Databases. *ACM Comput. Surv.*, 24(1):63–113, 1992.
12. Kenneth W. Ng, Zhenghao Wang, and Richard R. Muntz. Dynamic Reconfiguration of Sub-Optimal Parallel Query Execution Plans. Technical Report CSD-980033, UCLA, 1998.
13. V. Raman, A. Deshpande, and J. M. Hellerstein. Using State Modules for Adaptive Query Processing. Technical Report UCB/CSD-03-1231, University of California Berkeley, 2003.
14. M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, pages 25–36, 2003.
15. Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost Based Query Scrambling for Initial Delays. In *SIGMOD Conference*, pages 130–141, 1998.
16. A. N. Wilschut and P. M. G. Apers. Dataflow Query Execution in a Parallel Main-memory Environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.