Adaptive Join Processing in Pipelined Plans

Kwanchai Eurviriyanukul, Norman W. Paton, Alvaro A.A. Fernandes, Steven J. Lynden
School of Computer Science, University of Manchester
Oxford Road,Manchester M13 9PL, UK
(eurvirik,norm,alvaro,slynden)@cs.man.ac.uk

ABSTRACT

In adaptive query processing, the way in which a query is evaluated is changed in the light of feedback obtained from the environment during query evaluation. Such feedback may, for example, establish that misleading selectivity estimates were used when the query was compiled, leading to the optimizer choosing an inappropriate join order or unsuitable join algorithms. This paper describes how joins can be reordered, and the join algorithms used replaced, while they are being evaluated in pipelined plans. Where joins are reordered and/or replaced during their evaluation, the approach avoids duplicating work that has already been carried out, by resuming from where the previous plan left off. The approach has been evaluated empirically, and shown to be effective for improving query performance in the light of misleading selectivity estimates.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

1. INTRODUCTION

Adaptive query processing [2, 9] changes aspects of query evaluation while a query is executing, to take account of information about the query or its environment gathered from runtime monitoring. For example, predicate selectivity estimates may be revised to reflect correlations between columns that were not considered by the cost model of the optimizer.

Several proposals have been made that support join reordering during query evaluation. For example, [14, 17, 2] describe how an optimizer can be invoked at query runtime to identify a new plan in the light of information on progress to date. However, these approaches are coarse grained in the sense that they either reuse complete join results or restart join evaluation from scratch, discarding results produced using an earlier plan. As a result, they are not well suited for use in pipelined plans in which many operators are likely to be being evaluated simultaneously. Where pipelined query processing has been addressed, proposals have either required post-processing in the form of stitch-up plans [13] or have been limited to a single join algorithm [16]. As an alternative to plan reordering, Eddies [1] change the order in which tuples are routed through joins continuously during query evaluation. As such, they can be extremely fine grained, but obtain this flexibility by materializing (e.g. [18]) rather more intermediate data than may be required by other strategies.

This paper describes an approach to join reordering for use with pipelined plans in which previous work is reused, no post-processing is required, no state is maintained solely to support adaptation, and many different join operators are supported. The contributions of this paper are as follows: (i) algorithms that compute the work remaining to be done by a pipelined plan containing select, project and join operators; (ii) a characterization as to how frequently the circumstances arise in which the algorithms can be used safely; (iii) a description of the implementation of the algorithms and the approach used to monitor progress at runtime; and (iv) an experimental evaluation demonstrating a range of circumstances in which significant performance improvements have been obtained.

The remainder of the paper is structured as follows. Section 2 provides the technical context for the later results, in particular indicating how the work that remains to be done by a partially evaluated query can be described algebraically. Section 3 describes the adaptive technique and the states in which it can be applied. Section 4 considers how frequently query plans may be expected to be in a state that enables them to be adapted. Section 5 describes how the approach has been implemented in practice, and Section 6 presents the results of an experimental evaluation. Section 7 relates the results to other work, and Section 8 concludes.

2. DESCRIBING PARTIALLY EVALUATED QUERIES

2.1 Plans, Operators and Operator States

A query plan P can be represented by a tree consisting of a set of 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 of evaluating P.

In this paper, a distinction is drawn between logical and physical algebras. In essence, a *logical algebra* contains operators that perform an operation on tables (such as select, project or join), and a *physical algebra* may annotate these operators to indicate which specific implementation of an operator is to be used (thereby, for example, distinguishing between a hash join and a merge join).

Throughout the paper, an equality join condition and bag semantics are assumed, and R and S denote the left and right operands of a join, respectively. Four representative join algorithms are considered:

Hash Join ($\overset{H}{\bowtie}$): 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 to identify matching R-tuples.

Index Nested-Loop Join ($\overset{IN}{\bowtie}$): Each R-tuple is read in turn and its join attribute(s) are then used to search an index on S and to identify the tuples that match.

Symmetric Hash Join ($\overset{SH}{\bowtie}$): 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 tuples are returned.

Merge Join ($\overset{M}{\bowtie}$): Given inputs R and S sorted on join attribute(s) J, let v be the next smallest J-value occurring in both R and S. Tuples from R and S are read into corresponding caches until the last such tuples read have a J-value that is different from v. At this stage, leaving out the last tuples read in each cache, the tuples in the two caches are joined using a nested loop. Then, the caches are cleared of all but the last tuples read, and the process repeats. Where the last tuples read have different J-values, tuples are read, from either R or S as needed, until the corresponding J-values coincide.

In this paper, pipelined evaluation is described in terms of the iterator model [12], although the principal notions are independent of the specific implementation. The iterator model has three principal functions: OPEN, NEXT and CLOSE: OPEN prepares the operator for result production; NEXT produces one result at a time, and CLOSE performs cleaning up. 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.

In this paper, the primary emphasis is on adapting query plans in the states $\mathbb{N}'$, 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 $\mathbb{C}'$.

However, in addition to $\mathbb{N}'$ states, we also support adaptation during the $\mathbb{O}$ state of hash join. This is because hash join blocks in state $\mathbb{O}$ while the hash table is populated. As a result, adapting only in the $\mathbb{N}'$ state in plans that contain hash join significantly reduces opportunities for timely intervention.

2.2 Partially Evaluated Queries

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 its evaluation. 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, and $[\![I]\!]$ the bag that results from evaluating I.

- I^+ is the portion of $[\![I]\!]$ that has been returned by previous calls to the NEXT() function of I. Therefore, $I^+ \subseteq [\![I]\!]$.
- I^- is the portion of $[\![I]\!]$ that has yet to be returned by subsequent calls to the NEXT() function of I. As a result, $[\![I]\!] = (I^+ \cup I^-)$.
- $last(I)$ is the singleton set containing the last tuple added into I^+.

2.3 Quiescent States

Section 2.2 introduced 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 may be different at different points in its evaluation, i.e., at different occurrences of $\mathbb{N}'$. To see this, consider that 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^+ and S^+. If we assume that the left operand is used to populate the hash table, then $R^+ = [\![R]\!]$ (because the hash table must be fully populated before any results can be produced). The tuples produced so far by the algorithm may not simply be those denoted by $R^+ \bowtie S^+$, 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 $R^+ \bowtie S^+$ if $last(S^+)$ 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 can be precisely defined in terms of the inputs to the operator at that point in the execution.

The following are characterizations of the quiescent states for the example join operators in $\mathbb{N}'$ states:

Hash Join: The last S-tuple read has been joined with all matching R-tuples in the hash table.

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.

Merge Join: All tuples prior to the last R-tuple (resp. S-tuple) read have been joined with all matching S-tuples (resp. R-tuples) and the join attribute value(s) of the last R-tuple (resp. S-tuple) read is different from that of the last R-tuple (resp. S-tuple) joined.

In practice, the test for quiescence in $\mathbb{N}'$ states is straightforward to implement, normally involving a simple test on the state of the operator. Indeed, the other operators considered in this paper, namely select (σ), project (π), union ($\cup$), and *scan*, are always quiescent in $\mathbb{N}'$. We have extended the operator interface with an ISQUIESCENT function, which determines from the internal state of the operator whether or not it is quiescent in $\mathbb{N}'$.

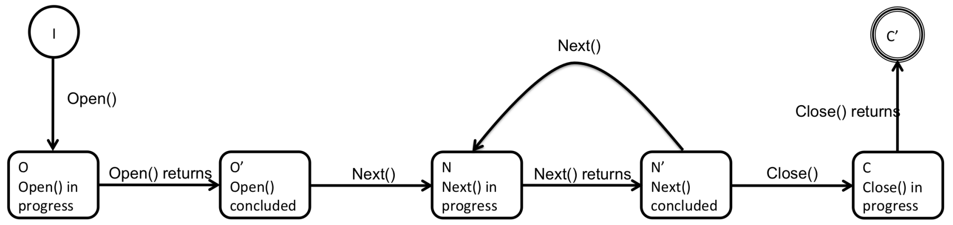


Figure 1: The state-transition diagram of an iterator.

In addition to $\mathbb{N}'$ states, we also support adaptation during hash table population for $\overset{H}{\bowtie}$, and as a result must also identify quiescent states when operators are in $\mathbb{I}$, $\mathbb{O}$ and $\mathbb{O}'$ states. Quiescence is normally trivial to characterize in these states; as no result has yet been produced, the remaining work to be done is all the work there was to do in the first place. However, as $\overset{H}{\bowtie}$ populates its hash table in Open, to avoid repeating evaluation of part of its left input, we need a more precise characterization that takes account of its internal state, which is discussed more fully in Section 2.4.

2.4 Describing Partial Results

When an operator is in a *quiescent* state, it is possible to define its result algebraically in terms of the data it has consumed. As a consequence, it is also possible to define algebraically the portion of the result that remains to be produced, as described for specific contexts below.

2.4.1 Partial Results in $\mathbb{N}'$ States

This subsection defines intermediate and remaining operator results in $\mathbb{N}'$ states. The precise description of such results enables the work that remains to be done by a pipelined plan to be defined precisely when all the operators in the plan are in quiescent $\mathbb{N}'$ states, as described in Section 3.

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 $\mathbb{N}'$ state, every tuple that has been read into the hash table (i.e., $R^+ = [\![R]\!]$) has been joined with every tuple that has been read from the other operand (i.e., S^+). To complete the evaluation, every tuple in $R^+ = [\![R]\!]$ needs to be joined with the tuples that have yet to be read from S (i.e., S^-). A similar justification lies behind the other entries in Table 1. The validity of the entries in the table has been proved [10].

When join operators in quiescent $\mathbb{N}'$ states are to be reordered, the rightmost column in Table 1 describes the portion of a join that remains to be evaluated by the new plan.

2.4.2 Partial Results in $\mathbb{I}$, $\mathbb{O}$ and $\mathbb{O}'$ States

This subsection defines intermediate and remaining results for operators in $\mathbb{I}$, $\mathbb{O}$ and $\mathbb{O}'$ states. The precise description of such results enables the work that remains to be done by a pipelined plan to be defined precisely when one of the operators is a hash join that is populating its hash table in Open, as described in Section 3.

Table 2 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. Most of the entries are trivial, as results are not returned by Open. However, in $R\overset{H}{\bowtie}S$ the definitions for $\mathbb{O}$ and $\mathbb{O}'$ reflect the fact that R is evaluated and the hash table populated by Open. Hash join is only considered quiescent in $\mathbb{O}$ if $last((^+R))$ has been inserted into the hash table.

Table 1: The intermediate result and the remainder of operators in a quiescent $\mathbb{N}'$

Operator	Intermediate Result	Remainder
σR	σR^+	σR^-
πR	πR^+	πR^-
$scan(R)$	$scan(R^+)$	$scan(R^-)$
$R \cup S$	$R^+ \cup S^+$	$R^- \cup S^-$
$R\overset{H}{\bowtie}S$	$R^+ \bowtie S^+$	$R^+ \bowtie S^-$
$R\overset{IN}{\bowtie}S$	$R^+ \bowtie S$	$R^- \bowtie S$
$R\overset{SH}{\bowtie}S$	$R^+ \bowtie S^+$	$(R^+ \bowtie S^-) \cup (R^- \bowtie S^+) \cup (R^- \bowtie S^-)$
$R\overset{M}{\bowtie}S$	$(R^+ - last(R^+)) \bowtie (S^+ - last(S^+))$	$(last(R^+) \cup R^-) \bowtie (last(S^+ \cup S^-))$

Table 2: The intermediate result and the remainder of operators in $\mathbb{I}$, $\mathbb{O}$ and $\mathbb{O}'$.

Operator	State	Intermediate Result	Remainder
σR	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	σR
πR	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	πR
$scan(R)$	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	$scan(R)$
$R \cup S$	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	$R \cup S$
$R\overset{H}{\bowtie}S$	$\mathbb{I}$	$\emptyset$	$R \bowtie S$
$R\overset{H}{\bowtie}S$	$\mathbb{O}$	$\emptyset$	$(R^+ \cup R^-) \bowtie S$
$R\overset{H}{\bowtie}S$	$\mathbb{O}'$	$\emptyset$	$R^+ \bowtie S$
$R\overset{IN}{\bowtie}S$	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	$R \bowtie S$
$R\overset{SH}{\bowtie}S$	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	$R \bowtie S$
$R\overset{M}{\bowtie}S$	$\mathbb{I}$, $\mathbb{O}$, $\mathbb{O}'$	$\emptyset$	$R \bowtie S$

3. COMPUTING THE REMAINING WORK

This section describes the adaptive strategy that supports join reordering and operator replacement for a plan P. The overall approach to adapting a query plan is as follows: (1)

```
proc GetRemainingℕ′(PhysicalPlan $P$) → LogicalPlan
case $P$ of
  $scan(R)$: return $scan(R^-)$
  $\sigma_{Pred}(R)$: return $\sigma_{Pred}$(GetRemainingℕ′($R$));
  $\pi_{Cols}(R)$: return $\pi_{Cols}$(GetRemainingℕ′($R$));
  $R \cup S$:
    return GetRemainingℕ′($R$)) $\cup$ GetRemainingℕ′($S$);
  $R\overset{H}{\bowtie}S$:
    return $scan(P.$GetHashTable()) $\bowtie$ GetRemainingℕ′($S$);
  $R\overset{IN}{\bowtie}S$:
    return GetRemainingℕ′($R$) $\bowtie$ PhysicalToLogical($S$);
  $R\overset{SH}{\bowtie}S$:
    j1 = $scan(P.$GetLeftHashTable()) $\bowtie$ GetRemainingℕ′($S$));
    j2 = GetRemainingℕ′($R$) $\bowtie$ $scan(P.$GetRightHashTable());
    j3 = GetRemainingℕ′($R$) $\bowtie$ GetRemainingℕ′($S$));
    return $j1 \cup j2 \cup j3$;
  $R\overset{M}{\bowtie}S$:
    lhs = $scan(R.$GetLast()) $\cup$ GetRemainingℕ′($R$);
    rhs = $scan(S.$GetLast()) $\cup$ GetRemainingℕ′($S$);
    return $lhs \bowtie rhs$
end case
```

Figure 2: Compute the work that remains to be done by a plan P

```
proc PhysicalToLogical(PhysicalPlan $P$) → LogicalPlan
case $P$ of
  $scan(R)$: return $scan(R)$
  $\sigma_{Pred}(R)$: return $\sigma_{Pred}$(PhysicalToLogical($R$));
  $\pi_{Cols}(R)$: return $\pi_{Cols}$(PhysicalToLogical($R$));
  $R \cup S$:
    return PhysicalToLogical($R$) $\cup$ PhysicalToLogical($S$);
  $R\overset{H}{\bowtie}S$ or $R\overset{IN}{\bowtie}S$ or $R\overset{M}{\bowtie}S$ or $R\overset{SH}{\bowtie}S$:
    return PhysicalToLogical($R$) $\bowtie$ PhysicalToLogical($S$);
end case
```

Figure 3: Convert a physical algebraic expression into logical algebra

monitor the progress of the currently executing physical plan P to revise selectivity estimates that were used to choose P; (2) compute the work that remains to be done by P, in the form of a logical plan P' expressed using the notation introduced in Section 2; (3) submit P' to a query optimizer to select an effective physical plan (call it P' too) given the statistics collected in Step (1); and (4) reuse state from P in P' as required; for example, this could be cached operator state or the current position in an evaluation, and then evaluate P'. This section focuses on Step (2) above. Steps (1), (3) and (4) are considered further in Section 5.

3.1 Remaining Work in Quiescent $\mathbb{N}'$ States

This subsection describes an algorithm that computes the work remaining to be done by a plan P, building on the characterization of the work that remains to be carried out by a plan in Table 1. The assumption is made that every join node in P is in a quiescent state. Where every node is quiescent, the result of every operator in P can be defined algebraically in terms of its inputs.

Throughout this paper, the *physical algebra* contains the operators σ, π, $\cup$, $\overset{IN}{\bowtie}$, $\overset{H}{\bowtie}$, $\overset{SH}{\bowtie}$, $\overset{M}{\bowtie}$ and *scan*, and the *logical algebra* contains the operators σ, π, $\cup$, $\bowtie$ and *scan*. In addition to accessing database tables, *scan* operators are able to provide access to collections that represent state cached by operators. The emphasis is on non-blocking operators because of our focus on pipelined plans, although we show how the approach can be applied to blocking operators by supporting adaptation during the construction of the hash table in hash join.

GetRemainingℕ′ in Figure 2 takes as input the physical algebra of plan P representing the query that is being evaluated, and returns a logical relational algebra expression that describes the result that remains to be computed by P.

For $\overset{M}{\bowtie}$, GetRemainingℕ′ uses GetLast to produce, for each of the join's operands, a *scan* operator that provides access to the last tuple read from the operand. For $\overset{H}{\bowtie}$ or $\overset{SH}{\bowtie}$, GetRemainingℕ′ uses the GetHashTable, GetLeftHashTable and GetRightHashTable operations to produce a *scan* operator for each hash table constructed by the join. The *scan* operators produced by $\overset{H}{\bowtie}$ and $\overset{SH}{\bowtie}$ provide indexed access to the tuples cached in a hash table, where the index attributes are those used to build the hash table in order to evaluate the join predicate. The interface provided here is the same as the *scan* operator for accessing indexed persistent collections, which is used as the right operand of $\overset{IN}{\bowtie}$.

GetRemainingℕ′ traverses the query plan rooted at P, and for each operator visited identifies the work that remains to be done, from Table 1. Notice that where the expression in the *Remainder* column contains an operand of the form I^+ for some I, this data is cached by the operator, so there is no need to recompute its value.

Where a subplan has to be evaluated in its entirety, as is the case for the right hand operand of $\overset{IN}{\bowtie}$, the procedure PhysicalToLogical from Figure 3 is called to convert the physical algebra expression into a corresponding logical algebraic expression.

For example, for the query:
$(((A\overset{IN}{\bowtie}B)\overset{IN}{\bowtie}C)\overset{IN}{\bowtie}D)$,

GetRemainingℕ′ returns:
$(((A^- \bowtie B) \bowtie C) \bowtie D)$.

The index nested loop joins don't maintain intermediate state, and the value that remains to be computed is that obtained by joining the data that has yet to be read from A, namely A^-, with the data in B, C and D (see the entry for $\overset{IN}{\bowtie}$ in Table 1).

There are circumstances in which, although GetRemainingℕ′ correctly characterizes the work that remains to be done, computing those results may be problematic. For example, if $R\overset{IN}{\bowtie}S$ is the original plan, the work remaining to be done is $R^-\bowtie S$. By the commutativity of join, this is equivalent to $S\bowtie R^-$. However, in practice, if the R^- was to be used as the right operand of an $\overset{IN}{\bowtie}$, index lookups would access the complete table, and not only the portion R^-. As a result, tuples from R^+ could contribute to the result of the join more than once, thereby leading to duplicates. Thus when processing expressions generated by GetRemainingℕ′, the optimizer

```
proc GetRemaining$\mathbb{O}$(PhysicalPlan $P$, Operator $h$)
  $\rightarrow$ LogicalPlan
if $P$.GetRoot() $\in$ $Predecessors(h)$
  case $P$ of
    $scan(R)$: return $scan(R)$
    $\sigma_{Pred}(R)$: return $\sigma_{Pred}$(GetRemaining$\mathbb{O}(R, h)$);
    $\pi_{Cols}(R)$: return $\pi_{Cols}$(GetRemaining$\mathbb{O}(R, h)$);
    $R \cup S$:
      return GetRemaining$\mathbb{O}(R, h)$ $\cup$ GetRemaining$\mathbb{O}(S, h)$;
    $R \overset{H}{\bowtie} S$:
    // Hash join is in $\mathbb{O}'$ and thus has a populated hash table
    return
      $scan(P$.GetHashTable()$)$ $\bowtie$ GetRemaining$\mathbb{O}(S, h)$;
    $R \overset{IN}{\bowtie} S$ or $R \overset{M}{\bowtie} S$ or $R \overset{SH}{\bowtie} S$:
      return GetRemaining$\mathbb{O}(R, h)$ $\bowtie$ GetRemaining$\mathbb{O}(S, h)$;
  end case
else if $P$.GetRoot() $= h$ then
  // Hash join is in $\mathbb{O}$ and thus has partly populated hash table
  $R = h$.GetLeft();
  $S = h$.GetRight();
  return ($P$.GetHashTable() $\cup$ GetRemaining$\mathbb{N}'(R)$) $\bowtie$
    GetRemaining$\mathbb{O}(S, h)$
else // $P$.GetRoot() $\in$ $Successors(h)$
  case $P$ of
    $scan(R)$: return $scan(R)$
    $\sigma_{Pred}(R)$: return $\sigma_{Pred}$(GetRemaining$\mathbb{O}(R, h)$);
    $\pi_{Cols}(R)$: return $\pi_{Cols}$(GetRemaining$\mathbb{O}(R, h)$);
    $R \cup S$:
      return GetRemaining$\mathbb{O}(R, h)$ $\cup$ GetRemaining$\mathbb{O}(S, h)$;
    $R \overset{H}{\bowtie} S$ or $R \overset{IN}{\bowtie} S$ or $R \overset{M}{\bowtie} S$ or $R \overset{SH}{\bowtie} S$:
      return GetRemaining$\mathbb{O}(R, h)$ $\bowtie$ GetRemaining$\mathbb{O}(S, h)$;
  end case
end if
```

Figure 4: Compute the work that remains to be done by a plan P containing the hash join operator h in state $\mathbb{O}$.

must not construct plans in which the right operand of $\overset{IN}{\bowtie}$ is of the form T^- for any table T. Note that there are alternative approaches (e.g. [16]) that allow the computation of duplicates that are subsequently removed, or that assume the presence of ordered row identifiers that are easily filtered, but such approaches are not considered further here.

3.2 Remaining Work when Hash Join is Evaluating Open

This subsection provides an algorithm that computes the work remaining to be done by a plan P, in which the Open operation of a hash join operator is being evaluated, and thus in the process of populating its hash table. As we assume that it is not possible to ask an operator which state of execution in Figure 1 it is in, we need to be able to infer this from the location of the operator in the plan.

Without loss of generality, we assume that all Open operations call Open on their children before carrying out any operator specific behaviors, and that binary operators call Open on their left child before calling Open on their right child. In the case of $\overset{H}{\bowtie}$, for example, this means that both children are opened (recursively down to the leaves) before the hash table starts to be populated. Under this assumption, it follows that whenever there is algorithm-specific behavior in the implementation of Open, this behavior occurs at a place in a sequence that is determined by the occurrence of that operator in a postorder traversal of the operator tree representing the query plan.

Consider P^N, any subplan of a plan P rooted on an operator N. Given N, let the following two sets of operators in P be defined:

1. $Predecessors(N)$ contains the operators that are visited before N in the postorder traversal of P that unfolds from a call to the Open operation of the root node of P.

2. $Successors(N)$ contains the operators that are visited after N in the postorder traversal of P that unfolds from a call to the Open operation of the root node of P.

The predecessors and successors of N can be used to infer the states of other operators in the plan as follows. Given the state of an operator, the work that remains to be done by the operator can be read from Table 2.

Let N be in $\mathbb{O}$ and $p \in Predecessors(N)$. Then, if p is not a left-hand descendent of a $\overset{H}{\bowtie}$ in $Predecessors(N)$ then p is in $\mathbb{O}'$. If p is a left-hand descendent of a $\overset{H}{\bowtie}$ in $Predecessors(N)$, then p will have been fully evaluated by the Open of the $\overset{H}{\bowtie}$, and the result of the subplan of which it is part will have been cached in the hash table of the $\overset{H}{\bowtie}$. Such a result is represented by the R^+ in the Remainder column of $\overset{H}{\bowtie}$ in state $\mathbb{O}'$ in Table 2. Among the operators considered in this paper, only $\overset{H}{\bowtie}$ can cause a child to move to a state subsequent to $\mathbb{O}'$ in Figure 1 as a result of executing algorithm-specific behavior in its Open operation.

Again, let N be in $\mathbb{O}$, but now let $s \in Successors(N)$. If s is not an ancestor of N then s is in $\mathbb{I}$. Alternatively, if s is an ancestor of N then s is in $\mathbb{O}$, and, because of our assumption that any operator-specific behavior takes place after calls to Open on children nodes, no operator-specific behavior will have been carried out by s.

Given that N is known to be in state $\mathbb{O}$, we have now identified how to determine the state of every operator in P, and thus can identify the work remaining from Table 2.

GetRemaining$\mathbb{O}$ in Figure 4 takes as input a plan P representing the query that is being evaluated and a hash join $h = R\overset{H}{\bowtie}S$ in state $\mathbb{O}$, where h is quiescent in $\mathbb{N}'$ – this is important because the plan that computes the remaining work depends on R^-. Note that, if so, $last(R)$ has been inserted into the hash table of h. P is expressed in a physical relational algebra expression, and GetRemaining$\mathbb{N}'$ returns a logical relational algebra expression that describes the result that remains to be computed by P.

As an example, for the physical plan:
$(((A\overset{H}{\bowtie}B)\overset{IN}{\bowtie}C)\overset{IN}{\bowtie}D)$,

where h is $A\overset{H}{\bowtie}B$, GetRemaining$\mathbb{O}'$ returns:

$((((A^+ \cup A^-) \bowtie B) \bowtie C) \bowtie D)$,

where A^+ is cached in the hash table. The hash join has yet to return any data, so the work that remains to be done includes the evaluation of the remainder of A and all the joins.

Operator (o)	Max $\|Quiescent\mathbb{N}'\|_o$	Probability Estimate
$R\overset{H}{\bowtie}S$	$\|S\|$	$\frac{1}{\|R\|*selectivity(R\bowtie S)}$
$R\overset{IN}{\bowtie}S$	$\|R\|$	$\frac{1}{\|S\|*selectivity(R\bowtie S)}$
$R\overset{SH}{\bowtie}S$	$max(\|R\|,\|S\|)$	$\frac{max(\|R\|,\|S\|)}{\|R\|*\|S\|*selectivity(R\bowtie S)}$
$R\overset{M}{\bowtie}S$	$max(\|R\|,\|S\|)$	$\frac{max(\|R\|,\|S\|)}{\|R\|*\|S\|*selectivity(R\bowtie S)}$

Table 3: The probability of that a join operator is quiescent when it returns a result tuple.

4. OPPORTUNITIES FOR ADAPTATION

A precondition on the application of GetRemaining$\mathbb{N}'$ from Section 3 is that all the operators in a subplan to which it is applied are quiescent. Thus, in order to establish the practicality of the approach, one must establish that join operators, and in particular compositions of join operators in a subquery, are quiescent simultaneously often enough in practical circumstances. The probability that an operator o is quiescent in state $\mathbb{N}'$ (i.e. when a tuple has just been returned) can be calculated as follows:

$$ProbOperatorQuiescence(o) = \frac{|Quiescent\mathbb{N}'|_o}{|\mathbb{N}'|_o} \quad (1)$$

where $|Quiescent\mathbb{N}'|_o$ is the number of quiescent $\mathbb{N}'$ states of o, and $|\mathbb{N}'|_o$ is the number of $\mathbb{N}'$ states if o, where $Quiescent\mathbb{N}'_o \subseteq \mathbb{N}'_o$.

Let R and S be the inputs of a join and $|R|$, $|S|$ their respective cardinalities. The maximum number of quiescent states that each join algorithm can be in during its evaluation is stated in Table 3. To take an example, $R\overset{H}{\bowtie}S$ is in a quiescent state whenever the last S-tuple read has been matched with every R-tuple. Thus the maximum number of quiescent states is the number of S-tuples. The number of quiescent states is less than this whenever there are tuples in S that don't match any tuples in R.

The numbers of tuples in a join result can be computed using the following equation, where the selectivity of a join is the fraction of the tuples in the cross product that appears in the result. The value of the join selectivity can be estimated in a range of different ways (e.g. see [11]).

$$|\mathbb{N}'| = |R| * |S| * selectivity(R\bowtie S) \quad (2)$$

Substitution of the maximum number of quiescent states in Table 3 and the right-hand side of Equation (2) into the right-hand side of Equation (1) gives an estimate of the probability that each join algorithm is quiescent when it returns a tuple, as shown in the rightmost column of Table 3.

For a query plan P, the probability of the plan being quiescent is the probability that all the operators in the plan are quiescent at the same time when a tuple is produced, which is the product of their individual probabilities:

$$ProbPlanQuiescence(P) = \prod_{o\in P} ProbOperatorQuiescence(o)$$

Relation	Cardinality
OWNER (O)	500,001
CAR (C)	715,142
DEMOGRAPHICS (D)	500,001
ACCIDENTS (A)	2,145,438
TIME (T)	25,523
LOCATION (L)	269

Table 4: Tables from the DMV data set.

As noted in Section 3, all of σ, π, $\cup$ and *scan* are quiescent whenever a tuple is returned

The probability model has been applied to queries over a Department of Motor Vehicles (DMV) database and data generator obtained from IBM Almaden. The cardinalities of the tables used are stated in Table 4. Table 5 indicates how frequently quiescent states occur in several example queries involving hash joins for the database in Table 4. The *Number of Quiescent States* column indicates the numbers of times when all operators are quiescent at the same time. The *Actual Prob(ability)* column states the probability that all the operators in the query are quiescent when the query produces a result, on the basis of experimental runs of the queries. The *Estimated Prob(ability)* column is calculated based on the formula in Table 3, where the join *selectivity* values used were obtained experimentally (and as a result, are essentially correct).

The results in Table 5 show that quiescent states are quite common for most of the queries, and thus that there is unlikely to be lengthy delay between opportunities for adaptation. The probability estimates, which are known to be upper bounds, were also shown to be reasonably accurate for representative examples (as explored further in Section 6). These estimates could be improved, for example by taking foreign key information into account; we do not consider this further here, as the purpose of this section has not been to provide highly accurate predictions, but rather: (i) to identify the factors that determine how frequently all the operators in a subquery can be expected to be in quiescent states simultaneously; and (ii) to give an indication as to how frequently co-occurrence of quiescent states is in practice. The overall lesson is that the frequency with which a complete subquery is quiescent can be estimated with reasonable reliability from widely available statistics; in practice, this frequency is usually sufficiently large to allow the adaptive strategy to be widely used.

5. IMPLEMENTATION

5.1 Architecture

The principal components and their relationships are illustrated in Figure 5. In the figure, standard static query processing is depicted to the left, where the query is first compiled and optimized, with the optimizer selecting plans on the basis of statistics from the catalog. The architecture has been implemented in Java, using the XXL [7] libraries to access secondary storage, and a greedy algorithm to order joins based on predicted intermediate result sizes [11]. The resulting logical plan is then traversed by a heuristic physical join selection algorithm that uses a cost model to determine which of the physical operators should be used to evaluate each of the joins. When a query is initially optimized, all

Query	Number of Results	Number of Quiescent States	Actual Prob.	Estimated Prob.
$O \overset{H}{\bowtie} C$	715,142	715,142	1.000	1.000
$O \overset{IN}{\bowtie} C$	715,142	426,992	0.597	0.699
$O \overset{M}{\bowtie} C$	715,142	426,991	0.597	1.000
$O \overset{SH}{\bowtie} C$	715,142	715,115	1.000	1.000
$(O \overset{H}{\bowtie} C) \overset{H}{\bowtie} A$	2,145,438	2,145,438	1.000	1.000
$(O \overset{IN}{\bowtie} C) \overset{IN}{\bowtie} A$	2,145,438	426,992	0.199	0.233
$(O \overset{M}{\bowtie} C) \overset{IN}{\bowtie} A$	2,145,438	426,991	0.199	0.333
$(O \overset{SH}{\bowtie} C) \overset{SH}{\bowtie} A$	2,145,438	2,145,411	1.000	1.000

Table 5: Frequency of quiescent states.

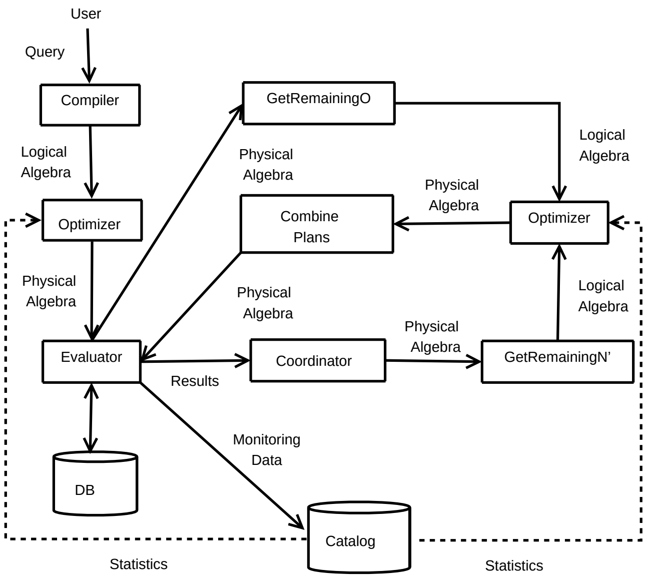


Figure 5: Architecture of adaptive query processor.

available selectivity estimates are for atomic predicates, and the cost model assumes that predicates are independent (i.e. there are no correlations between values of attributes in the database).

The pseudocode for the top level of the query processor is provided in Figure 6. Given a *Query* and a *Threshold* that indicates the minimum number of results between adaptations, RunQuery returns the bag of tuples that consitute the result of the query. In essence, the query is first compiled and optimized, subsequent to which the main program loops, fetching tuples from the result. Every *Threshold* tuples adaptation is considered, and an alternative plan is generated for comparison with the existing plan if the *reoptimization heuristic* is satisfied and the plan is quiescent. The *Threshold* is set to 100 in most of the experiments; this value needs to be low enough to allow timely adaptation, but high enough to prevent the overhead for considering reoptimization at a reasonable level, as explored further in Section 6.

As evaluation takes place, monitoring data on selectivities

```
proc RunQuery(String Query, Int Threshold) → Bag
Int Count = 0;
Bool Adapting = false;
Bag Result = {};

PhysicalPlan P = Optimize(Compile(Query));
P.Open();
while (P.HasNext())
  if (++Count ≥ Threshold)
    if CheckReoptHeuristic()
      if P.IsQuiescent()
        Count = 0;
        PhysicalPlan P' = Optimize(GetRemainingN'(P));
        if (P'.GetCost() < P.GetCost())
          P = CombinePlans(P',P);
        end if
      end if
    end if
  end if
  Result.Add(P.Next());
end while
P.Close();
return Result;
```

Figure 6: Top level program, including invocation of adaptations.

is collected; both selectivity estimates for atomic and complex conditions may be revised at query runtime, thereby allowing selectivity estimation to take into account correlations between predicates used in the query. These selectivity estimates are used both for reoptimization and in the reoptimization heuristic, which considers reoptimization only if the cost or the cardinality estimated for the currently executing plan using dynamically obtained statistics has changed by at least 20% compared with the cost or cardinality for the same plan when it was selected, an approach similar to that adopted in [14]. A sensitivity analysis was carried out that showed that overall performance was affected little by significant changes to the threshold. We also evaluated a heuristic based on changes to the estimated cost of the plan, but this approach had higher overheads and generally gave rise to the same decisions as its less expensive counterpart.

The same optimization strategy is used for initial optimization and reoptimization, although the original optimizer has been changed slightly to support queries referring to partial results. The best plan proposed by the optimizer is compared with the predicted cost of the existing plan using the updated statistics, and evaluation continues using the plan with the lowest predicted cost.

When the new plan is chosen for evaluation, all occurrences of table names of the form R^+ or R^- need to be made to refer to the specific objects that represent them in the original plan. Hence, CombinePlans traverses the existing (P) and new (P') plans, updating the new plan to include references to the objects in the existing plan from which remaining results or cached intermediate state can be obtained. In essence, whenever P' contains an operator of the form $scan(R^-)$, the Java object representing the *scan* in P' will be the same instance as was used for scanning R in existing plan P. Furthermore, whenever P' contains an operator of the form $scan(R^+)$, R^+ will be implemented

Operator (Op)	Selectivity
σR	$\frac{\lvert res_p(op)\rvert}{\lvert res_p(R)\rvert}$
$R \overset{H}{\bowtie} S$	$\frac{\lvert res_p(op)\rvert}{\lvert R\rvert * \lvert res_p(S)\rvert}$
$R \overset{IN}{\bowtie} S$	$\frac{\lvert res_p(op)\rvert}{\lvert res_p(R)\rvert * \lvert S\rvert}$
$R \overset{SH}{\bowtie} S$	$\frac{\lvert res_p(op)\rvert}{(\lvert res_p(R)\rvert * \lvert S\rvert + \lvert res_p(S)\rvert * \lvert R\rvert)/2}$
$R \overset{M}{\bowtie} S$	$\frac{\lvert res_p(op)\rvert}{(\lvert res_p(R)\rvert * \lvert S\rvert + \lvert res_p(S)\rvert * \lvert R\rvert)/2}$

Table 6: Estimating operator selectivity

over the data structure that materialises R^+ in P. By this means, the new plan P' obtains both progress information (encapsulated within the objects implementing the *scan* operators) and intermediate result state from existing plan P.

The algorithm in Figure 6 indicates how adaptation is considered at the outer level of query evaluation, thereby supporting adaptation in $\mathbb{N}'$ states following the approach described in Section 3.1. Similar code is implemented within the OPEN operation of hash join, to support adaptation in its $\mathbb{O}$ state, following the approach described in Section 3.2.

5.2 Dynamic Selectivity Estimation

During query evaluation, the selectivities of the predicates on σ and $\bowtie$ operators are computed directly based on counts of the numbers of tuples that satisfy them. The selectivities are computed as described in Table 6, in which $|C|$ represents the number of items in the collection C, and $res_p(Q)$ represents the collection of values produced by the query or subquery Q during the period p. We assume access to cardinalities of base tables, and thus by propagation, cardinality estimates for inputs to other operators. In practice, we are principally interested in changes to selectivities relative to those that informed the last optimization of the query, and thus p is the period since the last (re)optimization. The selectivity estimates will be most representative of the overall selectivity of the predicate where the order in which tuples are processed by operators essentially selects random tuples from the underlying tables; the reliability of such estimates is an active research area in its own right, and is not discussed further here [4].

The selectivity of a predicate is the proportion of the tuples that satisfy the predicate, and as such can be computed by dividing the number of tuples in a collection that satisfy the predicate by the total number of tuples in the collection. Thus the selectivity of a predicate over a period p is obtained by dividing the number of tuples produced during p by a value representing the portion of the total input processed during p. For the operator $op = \sigma R$, this is straightforward, as both the number of tuples produced during p (i.e. $|res_p(op)|$) and the number of tuples from the input considered during P (i.e. $|res_p(R)|$) can both be measured directly using simple counts.

For a join operator $op = R \bowtie S$, however, although the number of tuples produced during p (i.e. $|res_p(op)|$) can be counted directly, the portion of the input considered needs to be estimated, and is specific to the operator used. In essence, the selectivity of a predicate on a join operator op is the ratio of the size of the result of the join to the size of the Cartesian product, i.e. $\frac{|res(op)|}{|R|*|S|}$.

As such, we need a way of estimating the portion of $|R| * |S|$ considered during a period p based on values that can be measured directly (e.g. $|res_p(R)|$ and $|res_p(S)|$). To take hash join as an example, during any period p in which the join produces results, $|res_p(R)| = 0$, so we estimate the portion of the Cartesian product considered during p as $|R| * |res_p(S)|$, i.e., every tuple read from S is joined with the whole of R, and thus the portion of the Cartesian product that has been explored during period p is that involving all of R and $res_p(S)$.

For symmetric hash join and merge join, during any period p, both operands are simultaneously consumed by the operators. Every tuple read from R (resp. S) is joined with every matching tuple from the other operand read so far. The portion of the Cartesian product considered during p can be estimated as $(|res_p(R)| * |S| + |res_p(S)| * |R|)/2$. It is divided by 2 to reflect the fact that the output of these join operators produced so far during the period p is estimated to have been contributed by the Cartesian products of the portion of two inputs processed so far. A similar argument can be applied to derive the $R \overset{IN}{\bowtie} S$ entry in Table 6. The reliability of selectivity estimates has been an active research area since [5], and is not discussed further here.

6. EXPERIMENTAL EVALUATION

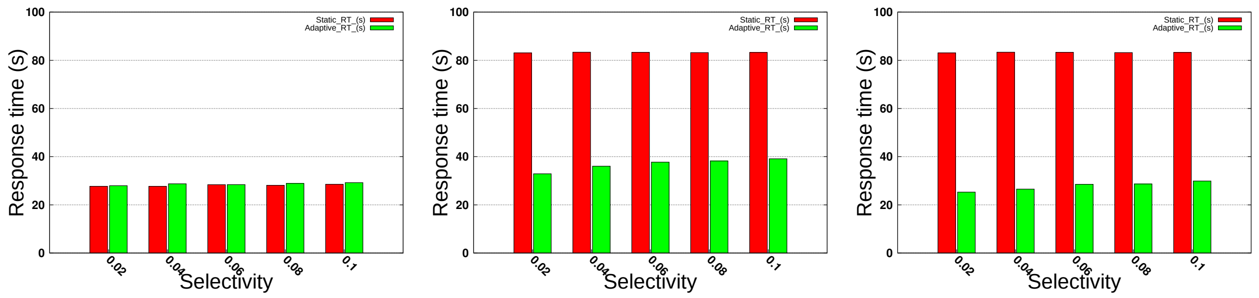
The experiments compare the adaptive approach with static optimization, where queries are initially optimized using accurate cardinalities for all tables, but estimated selectivities for all atomic predicates in which the selectivity of every equality is 0.1 and of every inequality is 0.3. We used the DMV database, which includes both data skew and correlated attribute values, with the relation sizes and names as listed in Table 4. Every query was run four times on an unloaded Dell Optiplex GX620 with a 3.00 GHz Intel Pentium D, 2GB of memory, running Fedora Linux Core 5. Response times reported are the average of the last three runs, i.e. the queries are run warm. B+Tree indexes are constructed on all key and foreign key columns. The inputs to merge joins are index scan operators that traverse these indexes. The following subsections evaluate the performance of the adaptations from Section 3 in $\mathbb{N}'$ and $\mathbb{O}$ states, respectively.

6.1 Adaptation in $\mathbb{N}'$ States

Experiment 1: Adapting to selectivity errors. The aim of this experiment is to establish how effectively adaptation handles errors in selectivity estimates. The performance of the following query is compared for values of v that are varied so that the actual selectivity ranges from 0.02 to 0.1.

```
Query-1:
select O.o_name, A.a_driver, D.d_age, C.c_id
from owner O, car C, demographics D, accidents A
where O.o_id = C.c_ownerid and O.o_id = D.d_ownerid
    and C.c_id = A.a_carid and A.a_id < v;
```

In the results, we consider two groups of joins, *All Joins* consisting of $\{\overset{H}{\bowtie}, \overset{SH}{\bowtie}, \overset{M}{\bowtie}, \overset{IN}{\bowtie}\}$ and *Pipelined Joins* consisting of $\{\overset{SH}{\bowtie}, \overset{M}{\bowtie}, \overset{IN}{\bowtie}\}$. Figure 7 shows the response times for varying selectivities, where different collections of join algorithms are made available to the adaptive infrastructure. The captions for (a) to (c), are of the form *initial group – reoptimize*



(a) All-All (b) Pipeline-Pipeline (c) Pipeline-All

Figure 7: $\mathbb{N}'$ States: Exp. 1: Response times for Query-1 for different join algorithms

group, and indicate which groups of join algorithms are made available for initial query optimization and for reoptimization during query evaluation.

The following observations can be made: (i) In Figure 7(a), all join algorithms are made available to the optimizer throughout. The initial plan is:

$$((demographics \overset{M}{\bowtie} owner) \overset{H}{\bowtie} car) \overset{H}{\bowtie}\ accidents.$$

As hash joins are blocking, this query only produces results when most of the query has been evaluated, and thus evidence to support optimizaton in $\mathbb{N}'$ states only surfaces from runtime monitoring as to the actual cardinality of the predicate on *accidents* when it is too late for reoptimization to make any difference, and thus no adaptation takes place. The change in selectivity of the predicate has little effect on response times, as the predicate is applied late in the evaluation of the query.

(ii) In Figure 7(b), only *Pipelined Joins* are available both for initial query planning and for reoptimization. The initial plan is $((demographics \overset{M}{\bowtie} owner) \overset{IN}{\bowtie} car) \overset{IN}{\bowtie} accidents$. As the operators are not blocking, runtime monitoring identifies the actual selectivity of the predicate on *accidents* early during query evaluation, giving rise to a proposed plan for computing the remainder of the result of the form:

$$(((accidents \overset{IN}{\bowtie} car) \overset{SH}{\bowtie} (last(owner^{+}) \cup owner^{-})) \overset{SH}{\bowtie}$$
$$(last(demographics^{+}) \cup demographics^{-})).$$

This plan is predicted to out-perform the original for the remaining work, so the original plan is replaced. The consequence is that the selective predicate on *accident* is evaluated much earlier, significantly reducing response times. (iii) The *Pipeline-Pipeline* configuration can be seen as enabling timely response to changes in selectivities throughout query evaluation, but restricting the range of algorithms that are available. Hence in Figure 7(c), only *Pipelined Joins* are available for initial query planning to enable timely reoptimization, but all joins are available for reoptimization. With the wider range of algorithms available, reoptimization proposes the same plan as in Figure 7(b), except that $\overset{H}{\bowtie}$ replaces $\overset{SH}{\bowtie}$ and $\overset{IN}{\bowtie}$, giving rise to improved response times.

Overall, the experiment indicates that significant performance gains can be obtained for pipelined plans, and that a strategy in which evaluation starts using pipelined evaluation to refine statistical estimates before reoptimizing in the light of those statistics to use pipelined or blocking operators can be effective.

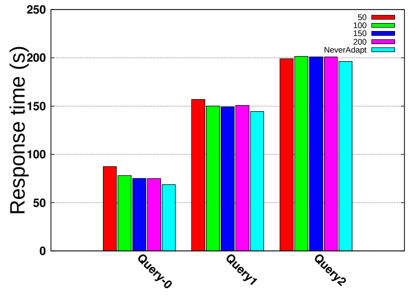


Figure 8: $\mathbb{N}'$ States: Exp. 2: Optimization overheads

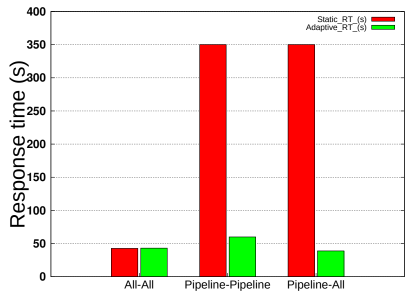


Figure 9: $\mathbb{N}'$ States: Exp. 3: Correlated predicates

Experiment 2: Overheads. The aim of this experiment is to determine the overhead resulting from adaptation being considered when no adaptation takes place.

Figure 8 shows response times for queries with two, three and four pipelined joins, in which all non-join predicates have a selectivity of 1, and adaptation is switched off (i.e. the adaptive infrastructure is in place, but a new plan never replaces the existing plan). As described in Section 5.1, a *Threshold* indicates how many result tuples are produced before the system considers reoptimization. This experiment measures response times for queries with different values for this *Threshold*. *Query-1* is as in Experiment 1; *Query-0* joins *owner*, *car* and *demographics*; *Query-2* joins the tables in *Query-0* to *accidents* and *time*. Figure 8 indicates that the overheads associated with adaptivity are quite low, even where reoptimization is considered frequently (e.g. the overhead for *Query-1* is *3.9%* when reoptimization is considered every 100 tuples). As the queries have predicates with selectivity of 1, the cost of producing a single result tuple is low, so the graph shows higher overheads than are generally encountered in practice. Overheads are addressed further in the discussion of *Experiment 4*.

Experiment 3: Adapting to selectivity errors resulting from correlations. The aim of this experiment is to establish how effectively adaptation is able to address errors in selectivity estimates that result from correlations between attribute values. The following example query is used:

Query-3:
select O.o_id, C.c_id, D.d_age, A.a_timeid, T.t_id
from owner O, car C, accidents A, demographics D, time T
where O.o_id = C.c_ownerid **and** O.o_id = D.d_ownerid
  and C.c_id = A.a_carid **and** A.a_timeid = T.t_id
  and A.a_seatbelton = 'y' **and** A.a_driver = 'unharmed';

Figure 9 shows response times for *Query-3* run using the same groups of join operators as in Experiment 1. The following observations can be made: (i) As in Experiment 1, no reoptimization takes place in the *All-All* strategy. (ii) Both *Pipeline-Pipeline* and *Pipeline-All* show substantial gains from reoptimization, which arise from reoptimization computing a higher selectivity (0.38) for the predicate *A.a_seatbelton = 'y' and A.a_driver = 'unharmed'* than was predicted by the optimizer (0.01) considering the equalities to be independent, when in fact they are correlated. This change leads to *accident* being placed later in query evaluation. In both *Pipeline-Pipeline* and *Pipeline-All*, the initial plan is $(((accidents \overset{IN}{\bowtie} time) \overset{IN}{\bowtie} car) \overset{IN}{\bowtie} owner) \overset{IN}{\bowtie} demographics$. In the *Pipeline-All* case, the plan produced to evaluate the remaining work is: $((((demographics \overset{M}{\bowtie} owner) \overset{H}{\bowtie} car) \overset{H}{\bowtie} accidents^{-})$ $\overset{H}{\bowtie} time)$ (iii) The best overall response time is for the adaptive version of *Pipeline-All*, which at 38.88s is about 9.7% faster than the 42.68s of the static *All-All*.

Experiment 4: Performance with multiple queries. This experiment compares the performance of multiple queries generated by a program that produces *4*-way joins[1] between randomly selected tables, with varying numbers of non-join predicates. Predicates on numerical attributes are inequalities with randomly selected literals from the range of the attribute, and predicates on strings are equalities with arbitrary literals from the extent of the attribute.

The scatter plots in Figure 10 show the relative performance of the static and adaptive cases for 30 randomly-selected queries for *Pipeline-Pipeline*, *Pipeline-All* and *All-All*. The following observations can be made: (i) No adaptation took place with the *All-All* configuration but the average overhead is small, at 1.9%. (ii) Adaptation shows significant benefits overall; in *Pipeline-Pipeline* the average performance improvement with adaptation switched on is *39.3%* and in *Pipeline-All* is *66.1%*. (iii) The benefit in some cases is great – in *Pipeline-Pipeline*, *2* of the *30* queries (Q5 and Q13) take less than *20%* of the time of their static counterparts, and in *Pipeline-All* it is *10* out of *30*. (iv) In *Pipeline-Pipeline* and *Pipeline-All*, adaptation took place for *27* and *28* of the *30* queries, respectively. (v) The average slowdown where adaptation did not take place (i.e. the overhead) in *Pipeline-Pipeline* was *0.3%* and in *Pipeline-All* was *0.9%*; as expected, these are much the same. (vi) For the *30* queries in the experiment, in the *Pipeline-All* case, the maximum, minimum and average number of reoptimization calls were, resp., 13, 1 and 7.86. (vii) For the *30* queries in the experiment, in the *Pipeline-All* case, the maximum, minimum and average number of adaptations were, resp., 1, 0 and 0.93. (viii) For the *30* queries in the experiment, the maximum, minimum and average number of result tuples per quiescent state were, resp., 6.8, 1.0 and 2.3 (where these values ignore Q16 and Q18, which proved to extreme outliers with ratios in the order of a thousand tuples per quiescent state).

[1]Results, not shown here, for different numbers of joins support the conclusions presented here using *4*-way joins.

6.2 Adaptation in $\mathbb{O}$ States

Experiment 5: Adapting to selectivity errors resulting from correlations. The aim of this experiment is to establish how effectively adaptation in $\mathbb{O}$ is able to address errors in selectivity estimates that result from correlations between attribute values. The experiment uses *Query-3*.

Figure 11 shows response times that compare the static plan with adaptive query evaluation, where adaptation in $\mathbb{N}'$ states is always enabled, and adaptation in $\mathbb{O}$ is either enabled or not. The following observations can be made: (i) The *All-All* configuration benefits (by 13.2%) from inclusion of adaptation in $\mathbb{O}$, as the early detection of the correlation between the predicates on *accident* results in the initial plan:

$$(((accidents \overset{H}{\bowtie} time) \overset{H}{\bowtie} car) \overset{H}{\bowtie} owner) \overset{H}{\bowtie} demographics$$

being replaced by:

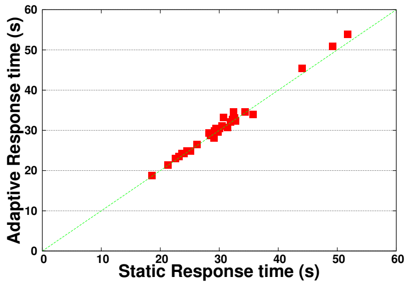
$$((((demographics \overset{H}{\bowtie} owner) \overset{H}{\bowtie} car) \overset{H}{\bowtie}$$
$$(accidents^{+} \cup accidents^{-})) \overset{H}{\bowtie} time)$$

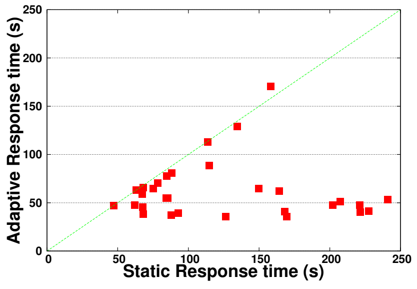
where $accidents^{+}$ tuples are obtained by scanning the partially constructed hash table produced by the initial evaluation of $(accidents \overset{H}{\bowtie} time)$. (ii) In the *Pipeline-All* configuration, as the initial plan uses only pipelined operators, adaptation takes place in an $\mathbb{N}'$ state, giving rise to an appropriate plan, and thus there is no adaptation in the $\mathbb{O}$ states of the resulting hash joins. The *Pipeline-Pipeline* configuration is included only for completeness.

Experiment 6: Performance with multiple queries. This experiment compares the performance of multiple queries generated by a program that produces *4*-way joins between randomly selected tables, with varying numbers of non-join predicates.

The scatter plot in Figure 12 compares the performance of static and adaptive queries for the *All-All* configuration, where adaptation is enabled in both $\mathbb{N}'$ and $\mathbb{O}$ states. The following observations can be made: (i) Adaptation in $\mathbb{O}$ states shows some benefits overall; *11* of the *30* queries adapted; they all adapted only once, in each case in $\mathbb{O}$. The average benefit obtained for the *11* adapted queries is 12.77%. (ii) The average overhead for the 19 queries that did not adapt is 1.9%, giving an average benefit for all 30 queries (including adapted and non-adapted queries) of 3.9%.

Overall, the benefits from adaptation in $\mathbb{O}$ are less marked than for $\mathbb{N}'$ for several reasons: (i) In pipelined plans, monitoring information becomes available from the evaluation of operators from throughout the plan before the first adaptation, thus enabling early reoptimization to benefit from wide ranging updates to statistics, which in turn means that the


(a) All-All


(b) Pipeline-Pipeline

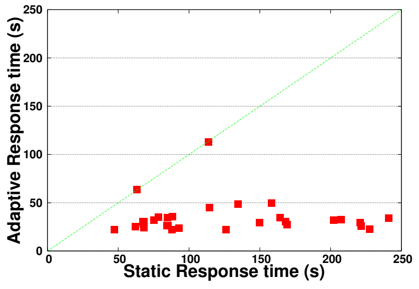

(c) Pipeline-All

Figure 10: $\mathbb{N}'$ States: Exp. 4: Randomly generated 4-way join queries.

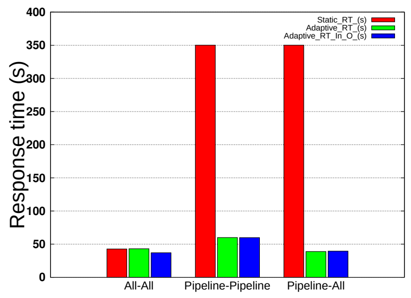


Figure 11: $\mathbb{O}$ States: Exp. 5: Correlated predicates

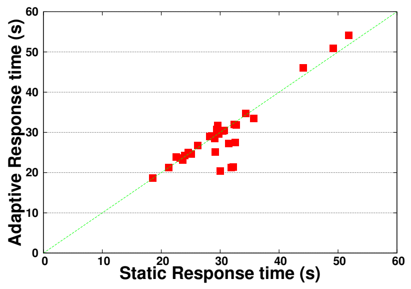


Figure 12: $\mathbb{O}$ States: Exp.6: Randomly generated All-All queries

reoptimized plan is more likely to be significantly different from the original plan than where few of the statistics have been updated, as in non-pipelined plans. (ii) In *All-All* configurations, the main memory hash join is almost always the algorithm of choice, and thus adaptations tend to reorder but not replace join operators, which means, for example, that IO costs are not changed by adaptations, thus restricting the potential overall benefits.

7. RELATED WORK

Approaches to adaptive query processing can be classified as: *plan preserving*, in which adaptation takes place continuously throughout evaluation over an essentially stable representation of a query; and *plan changing*, in which evaluation is halted, and some form of planning activity gives rise to a revised plan, with which evaluation is resumed. For adaptations that relate to joins, the *plan preserving* approach characterizes proposals such as Eddies [1]. In Eddies, the order in which data is routed through operators is determined dynamically on the basis of the costs and selectivities of the operations, and as a result join orders are adapted at query runtime. However, such flexible routing of data through joins generally involves materializing partially processed tuples in hash tables (e.g. [8]), whereas the proposal in this paper allows the query optimizer to choose from a wide range of join strategies, with minimal state management in support of adaptation.

For adaptations that relate to joins, the *plan changing* approach characterizes proposals such as Tukwila [13], POP [17] and Rio [2]. Plans may be changed either by switching between statically determined alternatives (e.g. [6]) or by reinvoking the optimizer at runtime to create a new plan. In Dynamic Re-Optimization [14] and POP [17], like the proposal in this paper, the optimizer constructs revised plans that reflect runtime selectivities. The replacement plans can reuse materialized intermediate results from operators that have run to completion, but the results of any partially computed subqueries are discarded. As such, the unit of reuse is considerably coarser than in the proposal from this paper, and unsuitable for use with pipelined evaluation. In addition, unlike in this paper, each reoptimization step could lead to a plan that repeats work carried out by any of the previous plans, to the extent that in POP a threshold is set to ensure that reoptimization terminates. Rio [2] proposes several enhancements to POP, including a *switch* operator that enables certain decisions on the plan used to be deferred until query runtime without discarding intermediate results. However, switchable plans must have similar structures, so the range of runtime changes supported without discarding results is narrower than in this paper. The approach in this paper also imposes some restrictions on the plans produced after reoptimization. For example, the new plan completes partially evaluated scans and makes use of intermediate data structures from the original plan. These restrictions reduce the options available to the optimizer, but enable fine grained reoptimization without redoing work.

Tukwila [13] shares several aspects with the work in this paper; in particular, it works with pipelined plans, and supports reuse of previously computed results. In Tukwila, adaptation suspends a partially evaluated plan p_0, and creates a new plan p_1 that carries out work required to answer the original query. However, derivation of p_1 builds on the logical algebraic representation of p_0, and thus cannot characterize the work that remains to be done as precisely as Table 1, which builds on the physical algebra. A consequence is that there is a need for a *stitch-up* phase that constructs the result of the query from values computed by p_0 and p_1. The *stitch-up* phase in turn requires access to intermediate results computed within p_0 and p_1, and thus Tukwila must cache intermediate results, which it does by reusing the hash tables of symmetric hash joins. Thus Tukwila depends on state associated with symmetric hash join algorithms, and thereby supports fewer join algorithms than our approach.

The proposal in Tukwila has several aspects in common with plan migration in continuous stream queries [19, 15], in that both migrate plans in a way that exploits intermediate state in symmetric hash joins, and both have *stitch-up* activities to ensure that all data that should be joined is or that no duplicates are produceds. In the stream setting, however, functionality is required to account for timestamps, for example, by draining the existing plan of tuples while redirecting later tuples as the new plan. A significant difference in approach between [19] and the proposal in this paper is that we describe plan progress at the level of a physical algebra, which, combined with the notion of quiescent states, provides a straightforward account that can be applied to multiple join algorithms. By contrast, [19] is cast at the level of the internal state used by the operator, in the context of a specific join operator.

The closest piece of related work, however, is [16], which describes an adaptive reordering of pipelined indexed nested loop joins. In common with this paper, states are identified in which adaptation can safely take place, selectivities are revised incrementally during evaluation, and joins are reordered when a new order is predicted to have a lower cost in the light of the updated selectivities. Thus [16] essentially provides a subset of the capability described in this paper. The key extension in this paper is the provision of support for multiple join algorithms, both in source and target plans, and the side-effect that adaptations may not only change the join order but also the physical join operators used.

We note that precise characterization of the work done and to be done by a query is also important for starting and stopping long running queries (e.g. [3]), and techniques described in Section 2 may also be relevant to checkpoint creation and query resumption.

8. CONCLUSIONS

We have presented an approach to adaptive join reoptimization for pipelined plans that: (i) accommodates multiple well established join algorithms; (ii) supports fine grained reuse of intermediate results; (iii) characterizes the progress of a plan and the work that remains to be done algebraically; and (iv) requires no auxiliary data structures to support adaptation. In combining these features, the approach generalises previous results. The key insight that underpins the work is that the progress of a plan, and the work that remains to be done, can be described precisely using a physical algebra. This abstraction both supports the writing of straightforward algorithms to support plan migration and interfaces naturally with existing optimizers. The experimental results have demonstrated significant benefits for the approach in pipelined query processing.

9. REFERENCES

[1] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *ACM SIGMOD*, pages 261–272, 2000.

[2] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-Optimization. In *Proc. ACM SIGMOD*, pages 107–118, 2005.

[3] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang. On suspending and resuming dataflows. In *ICDE*, pages 1289–1291, 2007.

[4] S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When can we trust progress estimators for sql queries? In *SIGMOD Conference*, pages 575–586, 2005.

[5] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *SIGMOD Conference*, pages 150–160, 1994.

[6] R. L. Cole and G. Graefe. Optimization of dynamic query evaluation plans. In *Proc. SIGMOD*, pages 150–160, 1994.

[7] J. V. den Bercken, B. Blohsfeld, J. Dittrich, J.ämer, T. Schäfer, M. Schneider, and B. Seeger. Xxl - a library approach to supporting efficient implementations of advanced database queries. In *Proc. VLDB*, pages 39–48, 2001.

[8] A. Deshpande and J. M. Hellerstein. Lifting the burden of history from adaptive query processing. In *Proc. VLDB*, pages 948–959, 2004.

[9] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[10] K. Eurviriyanukul, A. A. Fernandes, and N. W. Paton. A Foundation for the Replacement of Pipelined Physical Join Operators in Adaptive Query Processing. In *Proc. EDBT Workshops*, pages 589–600. Springer-Verlag, 2006.

[11] H. Garcia-Molina, J. Widom, and J. Ullman. *Database System Implementation.* Prentice-Hall, Inc., 1999.

[12] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[13] Z. Ives, A. Halevy, and D. Weld. Adapting to Source Properties in Data Integration Queries. In *Proc. SIGMOD*, pages 395–406, 2004.

[14] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. SIGMOD*, pages 106–117, 1998.

[15] J. Krämer, Y. Yang, M. Cammert, B. Seeger, and D. Papadias. Dynamic plan migration for snapshot-equivalent continuous queries in data stream systems. In *EDBT Workshops*, pages 497–516. Springer, 2006.

[16] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman. Adaptively Reordering Joins during Query Execution. In *Proc. ICDE*, pages 26–35, 2007.

[17] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. ACM SIGMOD*, pages 659–670, 2004.

[18] V. Raman, A. Deshpande, and J. M. Hellerstein. Using State Modules for Adaptive Query Processing. In *Proc. ICDE*, pages 353–364, 2003.

[19] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Conference*, pages 431–442, 2004.