Rewriting Procedures for Batched Bindings

Ravindra Guravannavar*
Indian Institute of Technology, Bombay
ravig@cse.iitb.ac.in

S Sudarshan
Indian Institute of Technology, Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

Queries, or calls to stored procedures/user-defined functions are often invoked multiple times, either from within a loop in an application program, or from the where/select clause of an outer query. When the invoked query/procedure/function involves database access, a naive implementation can result in very poor performance, due to random I/O. Query decorrelation addresses this problem in the special case of nested sub-queries, but is not applicable otherwise. This problem is traditionally addressed by manually rewriting the application to make it set-oriented, by creating a batch of parameters, and by rewriting the query/procedure to work on the batch instead of one parameter at a time. Such manual rewriting is time-consuming and error prone.

In this paper, we propose techniques that can be used to do the following. (a) Automatically rewrite programs to replace multiple calls to a query by a batched call to a correspondingly rewritten query. (b) Rewrite a stored procedure/function to accept a batch of bindings, instead of a single binding. Thereby, for example, a query which would have been invoked many times from different invocations of a stored procedure would be automatically replaced by one (or a few) invocations of a batched version of the query.

Our techniques can be applied to code written in any language, such as procedural versions of SQL, or Java. We have implemented the proposed rewriting techniques for a subset of Java, where database operations are performed using an API over JDBC. We demonstrate the benefits due to our rewrites with three cases from real-world applications, which faced significant performance problems due to repeated invocations of queries/procedures.

1. INTRODUCTION

Many database applications perform queries and updates from within procedural code that encodes business logic. Queries and updates inside the procedural code can use *host* language variables as parameters. Programs that use a mix of procedural constructs and SQL can run either inside a database system, as stored procedures or user-defined functions (UDF), or can run outside the database system. In such applications, iterative invocation of parameterized queries and updates is often the main cause of poor performance due to random I/O and network round-trip delays.

Iterative invocation of queries and updates can occur in several situations. For example, *(i)* programs and UDFs (including stored procedures) make explicit use of looping constructs and execute queries or updates inside a loop *(ii)* external batch jobs like end-of-day processing call database stored procedures repeatedly, by iterating over a set of parameters, causing repeated invocations of the queries and updates contained in the body of the stored procedure *(iii)* UDFs are invoked as part of SQL WHERE/SELECT or LATERAL clauses. The queries inside the UDF get repeatedly invoked for different parameters

Query 1.

```
SELECT orderid FROM sellorders
WHERE mkt='NSE' AND
      count_offers(itemcode, amount, curcode) > 0;

INT count_offers(INT itemcode, FLOAT amount, VARCHAR curcode)
DECLARE
    FLOAT amount_usd;
BEGIN
    IF (curcode == "USD")
        amount_usd := amount;
    ELSE
        amount_usd := amount * (SELECT exchrate FROM curexch
                        WHERE ccode := curcode;)    /* sql */
    END IF

    RETURN SELECT count(*) FROM buyoffers          /* sq2 */
           WHERE itemid = itemcode AND price >= amount_usd;
END;
```

Figure 1: Query Invoking a Simple UDF

Figure 1 shows a query that invokes a simple UDF in its WHERE clause. Standard decorrelation techniques [7, 2, 11] are applicable only in special cases when the body of the UDF is very simple, and cannot be applied in this case. In general, UDFs/procedures can be more complex with arbitrary control-flow and looping. Such an example is shown in Figure 2, where the UDF counts the number of items in a given category and all its sub-categories recursively.

Parameter batching is an important technique to speedup iterative execution of queries and updates. Parameter batching allows the choice of efficient set-oriented plans for queries and updates.

*Work supported by a Bell Laboratories Ph.D. fellowship.

VLDB '08, August 24-30, 2008, Auckland, New Zealand

In this paper we present our work on rewriting iterative programs to fulfill three related needs:

(a) Automatic rewrite of UDFs and stored procedures to accept batched bindings

(b) Automatic rewrite of programs that repeatedly execute a parameterized query or stored procedure to use batched invocation when possible

(c) Unnest queries having complex procedural nested blocks (e.g., queries that invoke UDFs)

The contributions of this paper are as follows.

1. We introduce the notion of *batched forms* and *batch-safe* operations. We identify the program transformation goal of pulling expensive operations out of loops as a key for addressing the needs mentioned above.
2. We present a set of program transformation rules that together achieve the transformation goal, when the program satisfies certain conditions. These rules rely on the *data dependence graph* of the given program and can work with complex procedures such as the ones shown in Figures 1 and 2.
3. We have implemented the proposed rewrite techniques and present an experimental study to demonstrate the gains due to the rewrites. Our experiments are based on real-life examples of performance problems and the results show significant performance improvements.

Magic set decorrelation [11] employs parameter batches for decorrelation of nested queries. Our techniques can be thought of as extending this approach for complex procedures. Related work, including prior work on loop optimization by program transformation [8], is discussed in Section 7. Our work is a step towards combining query optimization with program analysis and transformation techniques; we believe this combination will give significant benefits for database applications.

QUERY 2.
*SELECT * FROM category*
WHERE count_items(category_id) > f(level);

```
    INT count_items(INT catid)
    DECLARE
        INT totalcount; INT curcat; INT catitems; INT subcat;
        INT stack[100]; INT top; RECORD catrec;
    BEGIN
s1:     totalcount := 0;
s2:     top := 0;
s3:     stack[top] = catid;
s4:     top := top + 1;
s5:     WHILE top > 0 LOOP
s6:         top := top − 1;
s7:         curcat := stack[top];
s8:         catitems := SELECT count(item_id) FROM item
                        WHERE category_id = curcat;
s9:         totalcount := totalcount + catitems;
            // Now push all the subcategory ids onto the stack
s10:        FOR catrec IN SELECT category_id FROM category
            WHERE parent_category=curcat LOOP
s11:            stack[top] := catrec.category_id;
s12:            top := top + 1;
            END LOOP;
        END LOOP;
s13:    RETURN totalcount;
    END;
```

Figure 2: Query Invoking a Complex UDF

2. REWRITING FOR BATCHED BINDINGS

In order to exploit the benefits of batching, we must have an efficient *batched form* of the operation being invoked and a way of using the *batched form* in place of repeated invocations of the operation. In this section, we formally define *batched forms* of operations and introduce the problem of rewriting loops so as to make use of the *batched forms* of expensive operations invoked within them. We also consider the issues in batching invocations of operations that have side-effects. Finally, we introduce the problem of generating *batched forms* of complex procedures.

2.1 Batched Forms of Operations

Informally, the batched form of an operation takes a batch (or set) of parameters at once and processes them. Batched forms of operations are typically more efficient than iterative invocation of the corresponding non-batched forms. For example, a database bulk load operation can be thought of as a batched form of the *insert* operation. Similarly, a relational *join* can be thought of as a batched form of relational *selection* with a parameterized predicate. Note that we model a batch as a *set* and not a *sequence*. This choice is due to the fact that most batched forms do not guarantee the order in which the elements in the batch are processed and this is an important reason for their efficiency. We now define the batched forms formally.

Batched Forms of Pure (Side-Effect Free) Functions

Let $f : D \rightarrow R$ be a side-effect free function, where D is the domain and R is the range of f. A function $fb : BD \rightarrow BR$ is considered a batch form of f if the following are true.

1. The domain BD is the power set of D
2. The range BR is the power set of $D \times R$
3. $\forall b \in BD, fb(b) = \bigcup_{b_i \in b}\{(b_i, f(b_i))\}$

Example: Consider the square function defined as $sq(x) = x^2$. The corresponding batched form can be defined as $sqb(sx) = \{(x, x^2) : x \in sx\}$

Intuitively, the batched form of a function takes a set of parameters and returns a set comprising of all the results. To establish the correlation between a parameter and the corresponding result we require the batched form to return the parameter value along with the result.

Batched Forms of Parameterized Relational Queries

Relational queries are pure functions that return (multi)sets of tuples. Though we can use the above definition of batched forms for queries, it makes the return type of batched queries to violate the first normal form (1NF) as queries may have set-valued return type. We desire the first normal form on batched queries so as to be able to make our techniques easily implementable in existing relational query processing systems. Hence we use a slightly modified definition for the batched form of a query.

Let $q(p_1, p_2, \ldots p_n)$ be a query with n parameters. Let $v_1, v_2, \ldots v_m$ be the attributes in the result-set that q returns. The batched form qb of q takes a set p of *n-tuples* as its parameter (each *n-tuple* gives a binding for the parameters $p_1, p_2, \ldots p_n$). The result-set of qb contains the union of q's results for all the parameter tuples in p. Each tuple in *qb's* result-set contains $n+m$ attributes $p_1, p_2, \ldots p_n, v_1, v_2, \ldots v_m$.

sq1b(r): The batched form of query **sq1**

```
SELECT r.curcode, c.exchrate
FROM r JOIN curexch c ON r.curcode=c.ccode;
```

sq2b(r): The batched form of query **sq2**

```
SELECT r.itemcode, r.amount_usd,
       count(b.itemcode) AS count_offers
FROM r LEFT OUTER JOIN buyoffers b
ON b.itemid = r.itemcode AND b.price >= r.amount_usd
GROUP BY r.itemcode, r.amount_usd;
```

Figure 3: Batched Forms of Queries in Figure 1

Often only a subset of the parameters are sufficient to establish the correlation with the corresponding results. However, for simplicity, we assume the batched form returns all the parameter values along with the results. Formally,

$$qb(p) = \bigcup_{p_t \in p} \{\{p_t\} \times q(p_t)\}$$

When the result of q is an empty set for any parameter binding, the result-set of qb contains a tuple corresponding to the specific parameter binding but the attributes $v_1, v_2, \ldots v_n$ will be set to null. In the above formal definition of qb we omit this detail in the interest of readability.

Example:
Consider a parameterized query:

$$q(custid) = \Pi_{ordrid}(\sigma_{customer\text{-}id=custid}(\text{ORDERS}))$$

The corresponding batched form can be defined as:

$qb(\text{cs}) = \Pi_{(custid,ordrid)}(\text{cs} \ \exists\bowtie_{custid=customer\text{-}id} \text{ ORDERS})$,
where *cs* is the parameter relation attribute custid.

Batched forms of relational queries have been used in the context of query decorrelation [11, 7, 3, 2]. As shown in the above example, batched forms of simple SPJ queries use a join or an outer join. Batched forms of aggregate queries either use grouping followed by join or an outer-join followed by grouping. The details of deriving correct and efficient batched forms of SQL queries can be found in the literature on decorrelation. Figure 3 shows the batched forms of queries *sq1* and *sq2* used inside the UDF of Figure 1.

Most database systems also support batched bindings for basic data manipulation operations like insert, delete and update. The *insert into ... select from ...* construct can be used as the batched form of *insert* operation. For updates, we can use the *merge* construct of SQL:2003 (or the *update ... from ...* construct of SQLServer), as the batched form. Batched forms of these operations employ various techniques such as set-oriented index update and set-oriented integrity checks to offer increased efficiency over the corresponding non-batched operations.

Operations having Side-Effects

An operation with side-effects, in addition to returning a value, modifies the system state. Further, the return value may be a function of not only the arguments (parameters) but also the system state. We can model such an operation

```
for each t in select orderid, itemcode, amount, curcode
              from sellorders where mkt='NSE' loop
    < body of the udf with parameters bound from t >
    if (return value > 0)
        output t.orderid;
end loop
```

Figure 4: An Iterative Program/Plan for Query 1

with a pair of functions, $fv : S \times D \rightarrow R$ and $fs : S \times D \rightarrow S$ where S is the set of all possible system states, D is domain of parameters and R is the domain of result values. Since we assume batched forms are free to process the arguments in any order, we can define the batched forms for a only a restricted class of side-effect causing operations. We call this restricted class of operations, for which the batched forms are defined, as *batch-safe* operations. We call an operation *batch-safe* if the following conditions hold:
When processing a set of arguments,

1. the operation's return value, for any parameter, is independent of the order in which the parameters are processed.
   $\forall s \in S, \forall x, y \in D, fv(S, x) = fv(fs(S, y), x)$
2. the final system state is independent of the order in which the arguments are processed.
   $\forall s \in S, \forall x, y \in D,\ fs(fs(s, x), y) = fs(fs(s, y), x)$

For example, an INSERT operation on a table that has no constraints defined, is *batch-safe*. However, in the presence of table constraints (*e.g.,* a unique column), the INSERT operation may or may not be *batch-safe* depending on the exact set of parameters. Operations that write to an external file/device or communicate with other systems may or may not be *batch-safe* depending on the specific application. If the programmer has enclosed an operation inside a loop that gives no guarantee of order, *e.g.,* iteration over the result set of a query without order-by clause, we may treat the operation as *batch-safe*. Such analysis for *batch-safety* of an operation can be extended further, but is beyond the scope of this paper. Further, in this paper we assume that operations with side-effects do not have a return value, they just cause a change in the system state.

2.2 Rewriting Loops to Use Batched Forms

Consider a statement that invokes operation q inside loop L of a program P. We say that the invocation of q is *batchable w.r.t* loop L if it is possible to rewrite P into an equivalent program P' by removing the invocation of q from the body of the loop and making a single invocation of the batched form of q (*i.e., qb*) outside the loop. As an example consider the invocation of the UDF in Figure 1 inside the loop of Query 1. A program corresponding to the iterative plan for this query is shown in Figure 4. In this example, both the queries inside the body of the UDF can be batched *w.r.t* the enclosing loop as shown in Figure 5. Note that the updates performed by the batched version of the function are only on the *temporary* table. In the batched version we still have an iterative loop but it contains only inexpensive operations. It is possible to pull even such operations out of the loop, we discuss about this Section 6.

Even if an operation is *batch-safe*, it may not always be possible to batch a given invocation of the operation *w.r.t*

```
Let r1 =  SELECT DISTINCT itemcode, amount, curcode
          FROM sellorders WHERE mkt='NSE';
```

Now, Query 1 can be written as:

```
SELECT  orderid FROM sellorder so, count_offers_batched(r1) br
WHERE   so.mkt='NSE' AND so.itemcode=br.itemcode AND
        so.amount=br.amount AND so.curcode=br.curcode AND
        br.count_offers > 0;
```

where, *count_offers_batched* is the batched form of the UDF *count_offers* defined as follows. For brevity we omit the schema details when it is obvious.

```
TABLE count_offers_batched(TABLE r1)
DECLARE
    TABLE (itemcode, amount, curcode, cond1,
           amount_usd, count_offers) r2; // A temporary table
BEGIN
    FOR EACH t1 IN r1 LOOP
        FLOAT amount_usd; BOOLEAN cond1; INT count_offers;
        cond1 := (t1.curcode == "USD");
        cond1 == true? amount_usd := t1.amount;
        // variables below take default values if unassigned
        r2.addRecord((t1.itemcode, t1.amount, t1.curcode,
                      cond1, amount_usd, count_offers));
    END LOOP

    MERGE INTO r2 USING sq1b(e1) AS sq1b
    ON (r2.curcode=sq1b.curcode) WHEN MATCHED THEN
    UPDATE SET amount_usd = amount * sq1b.exchrate;

//  where e1 denotes SELECT DISTINCT curcode
//                   FROM r2 WHERE cond1=false;
//  and sq1b, the batched form of query sq2, is shown in Figure 3.

    MERGE INTO r2 USING sq2b(e2) AS sq2b
    ON (r2.itemcode=sq2b.itemcode AND
        r2.amount_usd=sq2b.amount_usd)
    WHEN MATCHED THEN
    UPDATE SET count_offers = sq2b.count_offers;

//  where e2 denotes
//  SELECT DISTINCT itemcode, amount_usd FROM r2;
//  and sq2b, the batched form of query sq2, is shown in Figure 3.

    RETURN (SELECT itemcode, amount, curcode, count_offers
            FROM r2;)
END
```

Note: MERGE is a SQL:2003 construct.

Figure 5: Batched Form of Query 1

a loop, because of side effects within the loop. In Section 4 we provide a set of program transformation rules that allow batching invocations of an operation when the program satisfies certain conditions.

2.3 Generating Batched Forms of Procedures

To speed up applications or queries that make repeated calls to stored procedures or UDFs we need efficient batched forms of these procedures. However, batched forms of complex operations like stored procedures and UDFs are (as far as we know) not available unless implemented by the programmers manually. We therefore consider the problem of automatically generating batched forms of stored procedures and UDFs. Our goal is to generate efficient batched forms by batching the expensive operations within the body of the procedure/UDF.

Given any side-effect free function or *batch-safe* operation (which could be a complex procedure) *f*, we can generate its trivial batched form as shown in Figure 6.

fb-trivial(pt) $\Longleftrightarrow$ *Apply(pt, f)*
where the function *Apply* is defined as :

Apply(pt, f):
$r = \{\}$;
for each t in pt
  $<$ *body of f with parameters bound from attributes of t* $>$
  rf = *return value of f*;
  r.*addRecords*({t} $\times$ rf);
return r;

Figure 6: Trivial Batched Form of a Procedure

Batched version of any procedure can thus be generated by enclosing it in a loop that iterates over the parameter set and invokes the procedure repeatedly. However, such a rewriting is not of significant benefit as *cost(fb-trivial)* for a batch size of k is nearly same as $k \times$ *cost(f)*; Such a rewriting can still be useful in reducing round-trip delays in client server environments. More interesting batched rewrites are the ones that use specialized and efficient strategies for batch processing, *e.g.,* batched *selection* within the procedure can be processed as a *join*, while a query in the procedure which performs a selection followed by an aggregate would have a batched form that employs grouping followed by join.

To generate an efficient batched form of a procedure, we can start with the trivial batched form of the procedure and try to batch each expensive sub-operation in the body of the procedure *w.r.t.* to the enclosing loop. To do so, the sub-operation must be taken out of the enclosing loop and substituted by its batched equivalent. If the sub-operation is a query, its batched form may be known. If the sub-operation is a procedure call, we recursively invoke the method to generate the batched form of the called procedure.

As we can see, generating batched form of a complex procedure reduces to the task of batching selected operations *w.r.t.* a loop (see Section 2.2). We address this problem in Section 4 after introducing some preliminaries in Section 3.

3. BACKGROUND

In this section, we formally outline the language constructs we support and provide background material on *data dependency* terminology used in this paper.

3.1 Language Constructs

For our illustration, we use a simple procedural language. The language offers expressions, assignment, conditional - branching and looping. The supported language constructs are briefly described below.

- *cursor loops* are of the form *for each* `record` *in* `query/table` *loop ... end loop;* An *order by* clause can be present if the iteration is over a table. When present, the *order by* is assumed to be *ascending* by default.
- *while loops* are of the form *while*`(predicate)` *loop ... end loop;*. Unlike the more general and powerful *while loops*, the cursor loops iterate over the result of a query and hence their iteration space is known once the query is evaluated.
- Branching is possible through *if-then-else* having the syntax *if* `(predicate)` {...} *else* {...}.

- Scalar variables. We consider only scalar variables in our discussion. However, our techniques can be easily extended to handle arrays, records and collection types. Each looping block can have variables local to the block. Statements can access variables local to their block or variables defined in any of the ancestor blocks.
- Result of scalar queries (queries that return exactly one tuple) can be assigned to variables.
  e.g., $v_1, v_2, \ldots v_n$ = *select* $c_1, c_2, \ldots c_n$ *from* ...;
  Note that a single assignment can be used to simultaneously assign values to multiple variables. Set-valued queries can be used only in the context of *cursor loops*.

We also use a few additional constructs in the transformed code. We assume these constructs are not available for the end-user and hence cannot be present in the input program.

- The TABLE type is used to construct the parameter batches. The TABLE type can be implemented so as to make use of both main memory and disk. We discuss this in more detail in Section 6.
- Updatable cursor loops are of the form *for each* `record` *by ref in* `table` *loop ... end loop;* Any updates to the record modify the underlying table variable.
- The transformed program may use relational operators such as selection, projection and join.

Assumptions

We make the following assumptions about the program.

1. Unconditional control transfer statements like GOTO, EXIT and CONTINUE are not used
2. Statements have no hidden side-effects. Information about all reads and writes performed by a statement (either on memory locations or on external resources like files and databases) are captured in the *data dependence graph* (explained in the next section).

3.2 Data Dependence Graph

The *Data Dependence Graph* (DDG) (sometimes referred to as Program Dependence Graph [9]) of a program is a directed multi-graph in which program statements are nodes (vertices) and the edges represent data dependencies between the statements. The different types of data dependency edges are explained below.

- A *flow-dependency* edge ($\xrightarrow{FD}$) exists from statement (node) s_a to statement s_b if s_a writes a location that s_b may read and s_b follows s_a in the forward control-flow.
- An *anti-dependency* edge ($\xrightarrow{AD}$) exists from statement s_a to statement s_b if s_a reads a location that s_b may write and s_b follows s_a in the forward control flow.
- An *output-dependency* edge ($\xrightarrow{OD}$) exists from statement s_a to s_b if both s_a and s_b may write to the same location and s_b follows s_a in the forward control flow.
- A *loop-carried flow-dependency* edge ($\xrightarrow{LFD_L}$) exists from s_a to s_b if s_a writes a value in the i^{th} iteration of a loop L and s_b may read the value in a later iteration (j^{th} iteration where $j > i$).

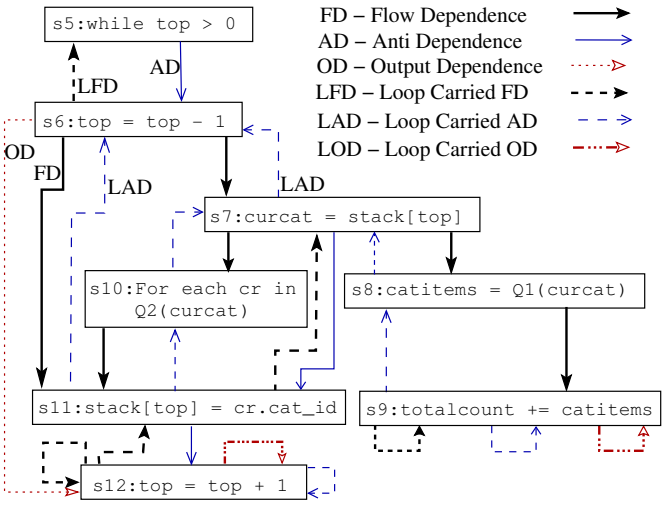

Figure 7: A subgraph of the Data Dependence Graph for the UDF in Figure 2

- Similarly, there are loop carried counter parts of *anti* and *output* dependencies and are denoted by ($\xrightarrow{LAD_L}$) and ($\xrightarrow{LOD_L}$) respectively.

The data dependence graph for the sample UDF of Figure 2 is shown in Figure 7.

External Dependencies

Statements may have dependencies not only through program variables but also through the database and other external resources like files. For example, we have $s_1 \xrightarrow{FD} s_2$ if s_1 writes a value to the database, which s_2 may read subsequently. Though standard dataflow analysis performed by compilers considers only dependencies through program variables, it is not hard to extend the techniques to consider external dependencies, at least in a conservative manner. For instance, we could model the entire database (or file system) as a single program variable and thereby assume every query/read operation on a database/file to be conflicting with an update/write of the database/file. In practice, it is possible to perform a more accurate analysis on the external writes and reads. When referring to external dependencies explicitly, we use E as a superscript to the corresponding type of dependence edge *e.g.,* $s_1 \xrightarrow{FD^E} s_2$.

4. PROGRAM TRANSFORMATION

Recall from Section 2.2 that an invocation of an operation q inside a loop L is said to be *batchable w.r.t* loop L if it is possible to rewrite the program into an equivalent program where the invocation of q is removed from the body of the loop and a single invocation of the batched form qb is made outside the loop. For such a rewrite, it is necessary that the operation should be *batch-safe*. However, the data and control dependencies between program statements may make it impossible to batch a statement that invokes an operation even if the operation is *batch-safe*. In this section, we present a set of program transformation rules, which enable us to batch a statement *w.r.t.* a loop when the program satisfies certain conditions.

The program transformation rules we present, like the equivalence rules of relational algebra, allow us to repeatedly refine a given program. Applying a rule to a program involves substituting a program fragment that matches the antecedent (LHS) of the rule with the program fragment instantiated by the consequent (RHS) of the rule. Some rules facilitate the application of other rules and together achieve the goal of batching a desired statement *w.r.t.* a loop. Applying any rule results in an equivalent program and hence the rule application process can be stopped at any point.

Notation Used in the Transformation Rules

- $R(s)$: The read-set of s is the set of variables read by statement or statement sequence s.
- $W(s)$: The write-set of s is the set of variables written by statement or statement sequence s.
- $U(s)$: $R(s) \cup W(s)$. Called the use-set of s.
- *pred*? s : Conditional statement. Equivalent to *if (pred) then s*.
- *LV(s)* : Set of variables local to the block statement s.
- *NLV(s)* : Set of variables accessible but not local to the block statement s. These are variables defined in an ancestor block.
- $|ss|$: Length of the statement sequence ss
- $ss[i]$: Stmt at the i^{th} position in sequence ss, $1 \leq i \leq |ss|$.
- $s_1 + s_2$: Concatenation (of statement sequences or strings).
- $SUBS(s, v, v')$: Statement obtained by substituting all occurrences of variable (or expression) v in statement s by variable v'.
- $SUBS(s, vs, map)$: Statement obtained as follows. For each variable $v \in U(s) \bigcap vs$, where vs is a set of variables, substitute all occurrences of v in statement s by $map(v)$.
- $s \bigcup^* r$: Disjoint union (UNION ALL) of relations s and r.
- $\Pi^d_{a_1,a_2,\ldots a_n}(r)$: Projection without duplicate elimination
- $(a_1, a_2, \ldots a_n)$: Tuple constructor
- *type-of(e)* : Data type of expression e.

Predicates on the DDG

- $s_1 \xrightarrow{FD} s_2$: True only if the DDG contains a flow-dependence edge (either internal or external) from s_1 to s_2.
- $s_1 \xrightarrow{FD+} s_2$: True only if the DDG contains a path from s_1 to s_2 having only FD edges.
- $s_1 \xrightarrow{(FD|LFD)+} s_2$: True only if the DDG contains a path from s_1 to s_2 having only FD or LFD edges.
- $indep(s_1, s_2)$: True only if there are no dependencies between statements s_1 and s_2.
- Similarly we have predicates for the existence of other types of dependencies.

Conventions

1. Loops of the form "*for each t by ref in r*" are updatable cursor loops. The underlying set *(r)* can be modified by assignments to the tuple's attributes.

2. Suppose a table-valued expression e has arity n. The rename operator $\rho_x(a_1, a_2, \ldots a_n)(e)$ returns the result of expression e under the name x, and with the attributes renamed to $a_1, a_2, \ldots a_n$.

3. The merge operator $\mathcal{M}_{a_1=b_1,\ldots a_n=b_n}(r, s)$ (based on the SQL:2003 merge construct) updates relation r by merging in the records of s. For each record in s that matches a record in r on the attributes common to r and s, the record in r is updated by assigning the values of attributes $b_1, \ldots b_n$ from the s tuple to the attributes $a_1, \ldots a_n$ of the r tuple correspondingly.

4. Projection that removes the specified attributes: $\Pi_{\bar{a_1},\bar{a_2},\ldots\bar{a_n}}(r)$ is equivalent to $\Pi_{S-\{a_1,a_2,\ldots a_n\}}(r)$, where S is the set of all attributes in *schema(r)*

5. Multi-assignment from scalar queries: Let q be a query returning exactly one tuple of arity n. The assignment $v_1, v_2, \ldots v_m = q$ (where $m \leq n$) assigns the values of the first m attributes of the returned tuple to the m variables on the LHS in that order.

In all the rules, unless specified, we assume **q** to be a **batch-safe** operation with **qb** as its batched form.

4.1 Rewriting Set Iteration Loops (Rule 1)

In the simplest case, the loop contains a single statement that invokes the operation we want to batch. In this rule, we consider cursor update loops - the loop iterates over a set of tuples and the values returned by the operation are assigned back to the attributes of the tuple associated with current iteration. Rules 1A through 1C shown in Figure 8 are the basic rules that allow replacing a loop by a batched invocation. Rule 1D and all the other rules presented in this section help us transform the program so as to enable the application Rules 1A, 1B or 1C.

In Rule-1A, q can be any batch-safe operation (with or without side-effects). Note the use of projection without duplicate elimination (Π^d) for constructing the parameter multiset. However, in rules 1B and 1C, we require q to be a *pure* function returning exactly one tuple (*e.g., a scalar query*). In rules 1B and 1C we construct a duplicate-free parameter set using the standard relational projection. This avoids the duplicate record problem while merging back the results of the batched invocation.

For brevity, in rules 1B and 1C we omit the form with loop invariant parameters. In rules 1B and 1C we deal only with assignments to cursor attributes and not variables. The reason for this will be clear when we describe rules 2 and 3. For now, it suffices to know that the later transformations bring the program into a form in which we can apply one of the rules described in this section.

4.2 Splitting a Loop (Rule 2)

In general, the statement to be batched may appear along with other statements inside the loop. Consider the examples in Figures 9 and 10. The statements to be batched are shown in bold. As shown in the figures, we try to split the loop into multiple set-iteration loops. The aim is to have the statement to be batched appear in a loop by itself, a form in which we can apply Rule 1. For example, in the rewritten code of Figure 9, the loop containing a single INSERT statement can be replaced by a batched invocation, by first removing the *order by* using Rule 1D and then applying Rule 1A(ii).

If the sequence of statements ss in a loop is made up of two consecutive sub-sequences ss_1, ss_2 (*i.e.,* $ss = ss_1 + ss_2$) and if there are no loop-carried flow/output dependencies from any statement in ss_2 to any statement in ss_1 or in ss_2 or to the loop predicate, then the loop can be split such that ss_1 and ss_2 appear in separate loops.

Unlike cursor loops, the iteration space for general WHILE loops cannot be known upfront [10] and is constructed dynamically. In general, this transformation, for the case of *while* loops, can be expressed as Rule 2A shown in in Figure 11. Note that after the split, the newly formed loops

Rule 1A: Unconditional invocation & no return value
1A(i) Basic form

for each t in r loop
  $q(t.c_1, t.c_2, \ldots t.c_m)$; $\iff$ $qb(\Pi^d_{c_1,c_2,\ldots c_m}(r))$;
end loop;

1A(ii) Form with loop invariant parameters

for each t in r loop
  $q(t.c_1, t.c_2, \ldots t.c_m, v_1, v_2, \ldots v_n)$;
end loop;
$\Updownarrow$
$qb(\Pi^d_{c_1,c_2,\ldots c_m}(r) \times \{(v_1, v_2, \ldots v_n)\})$;

Rule 1B: Unconditional invocation with return value

for each t by ref in r loop
  $t.c_{w1}, t.c_{w2}, \ldots t.c_{wn} = q(t.c_{r1}, t.c_{r2}, \ldots t.c_{rm})$;
end loop;
where q is a *pure* function.
$\Updownarrow$
$\mathcal{M}_{c_{w1}=c_{w1'}\ldots c_{wn}=cwn'}(r, e)$
where $e = \rho_x(c_{r1}, \ldots c_{rm}, c_{w1'}, \ldots c_{wn'})qb(\Pi_{c_{r1},\ldots c_{rm}}(r))$;

Rule 1C: Conditional Invocation

for each t by ref in r loop
  $(t.cv == true)?\ t.c_{w1}, \ldots t.c_{wn} = q(t.c_{r1}, \ldots t.c_{rm})$;
end loop;
where q is a *pure* function.
$\Updownarrow$
$\mathcal{M}_{c_{w1}=c_{w1'}\ldots c_{wn}=cwn'}(r, e)$, where
$e = \rho_x(c_{r1}, \ldots c_{rm}, c_{w1'}, \ldots c_{wn'})qb(\Pi_{c_{r1},\ldots c_{rm}}(\sigma_{cv=true}\ r))$;

Rule 1D: Removal of Order-By

for each t [by ref] in r **order by *cols* loop**
  batch-safe-operation(t)
end loop;
$\Updownarrow$
for each t [by ref] in r loop
  batch-safe-operation(t)
end loop;

Figure 8: Rule 1

that iterate on the *loop local table* must have an ORDER BY clause. The ORDER BY clause can then be eliminated if the statement inside is *batch-safe* (Rule 1D).

In splitting the loop we introduce a table valued variable and make the local variables of the loop as attributes of this table. We call these tables as *loop-local tables*. Each loop in the original program, if required to be split, introduces exactly one table. The table essentially serves to break the loop carried anti-dependencies (*i.e.,* to avoid overwriting the same location before the value is read). We call the table associated with loop L in the original program as the L-table. *Loop local tables*, when small enough, can be held in memory but this may not always be possible. We discuss the implementation issues of *loop-local* tables in Section 6.

The need for conditions (**c1**) and (**c2**) in Rule 2A is straight forward. The need for (**c3**) arises due to the way in which we construct the L-tables. Only the variables local to loop L are made attributes of the L-table. Any variable defined in an ancestor block L_p of L becomes an attribute of the

```
for each r in  SELECT grantid, empid, gnum FROM grantload loop
    int internalid = foo(r.grantid, r.empid);
    INSERT INTO grants VALUES
                (interalid, r.empid, r.gnum);
    total + = r.gnum;
end loop;
```

⇓

```
TABLE(key, empid, gnum, internalid) t;
int loopkey = 0;
for each r in  SELECT grantid, empid, gnum FROM grantload loop
    RECORD(key, empid, gnum, internalid) s;
    int internalid = foo(r.grantid, r.empid);
    s.key = loopkey++;
    s.empid = r.empid;   s.gnum = r.gnum;
    s.internalid = internalid;
    t.addRecord(s);
end loop;

for each s in t loop order by key
    INSERT INTO grants VALUES
                (s.interalid, s.empid, s.gnum);
end loop;

for each s in t loop order by key
    total += s.gnum;
end loop;
```

Figure 9: Splitting a `cursor` loop

```
while (top > 0) loop
    top = top − 1;
    curcat = stack[top];
    catitems = SELECT count(itemid) FROM item
               WHERE category=curcat;
    totalcount + = catitems;
end loop;
```

⇓

```
TABLE(key, curcat, catitems) t;
int loopkey = 0;
while(top > 0) loop
    RECORD(key, curcat, catitems) r;
    top = top − 1;
    curcat = stack[top];
    r.key = loopkey++;
    r.curcat = curcat;
    t.addRecord(r);
end loop;

for each r by ref in t order by key loop
    t.catitems = SELECT count(itemid) FROM item
                 WHERE category=t.curcat;
end loop;

for each r in t order by key loop
    totalcount + = t.catitems;
end loop;
```

Figure 10: Splitting a `while` loop

L_p-table. Constructing the L-tables in this way avoids redundant space consumption and more expensive updates of redundant information by giving normalized representation of the L-tables. By introducing an extra loop local variable, we can make a loop satisfy condition (**c3**) even if it did not satisfy it originally. The extra loop local variable preserves the value of the non-local variable for each iteration. For brevity, we omit the transformation rule for this step.

Rule 2 generalizes rule T4 of Lieuwen and DeWitt[8]. We compare our work with [8] in Section 7.

4.3 Separating Batch-Safe Operations (Rule 3)

A program statement may contain the expression we want to batch in combination with other non batch-safe opera-

Rule 2A: Splitting a WHILE Loop

while p **loop**
  ss_1 /* Stmt sequence (first part) */
  ss_2 /* Stmt sequence (second part) */
end loop;
where
(**c1**) No backward (loop carried) flow/output dependencies from ss_2 to the loop predicate p or to any stmt in ss_1 or ss_2
Formally, $\nexists s_2 \in ss_2$ *such that:* (see Note 3)
$((s_2 \xrightarrow{LFD|LOD} p) \vee (\exists s_1 \in ss_1 + ss_2 \; AND \; s_2 \xrightarrow{LFD|LOD} s_1))$

(**c2**) No external flow dependencies between stmts in ss_1 and ss_2
$\nexists s_1 \in ss_1, s_2 \in ss2 \; s.t. \; s_1 \xrightarrow{FD^E} s_2$

(**c3**) No non-local variable written in the first partition is read in the second. Formally,
$NLV(\mathcal{S}) \bigcap W(ss_1) \bigcap R(ss_2) = \phi$ (see Note 1)

TABLE(**T**) t;
int loopkey = 0;
while p **loop**
  RECORD(**T**) r;
  ss_1
  r.key=loopkey++;
  ss_r;
  t.addRecord(r);
end loop;

for each r **by ref in** t **order by** t.key **loop**
  ss_2'
end loop;
delete t;
where
Let SL (split locals) be the set of local variables written in the first partition and read in the second.
$SL = LV(\mathcal{S}) \bigcap W(ss_1) \bigcap R(ss_2)$ (see Note 1)

(A) The schema **T** of the table (t) and the record (r) contains an attribute corresponding to each variable in SL and a key.
(B) ss_r contains statements assigning values of split locals to the corresponding attributes of r.
(C) The stmt sequence ss_2' is same as ss_2, except that each reference v to a variable in set SL is replaced by $r.v$. Formally,
$ss_2'[i] = SUBS(ss_2[i], SL, map : v \rightarrow r.v), 1 \leq i \leq |ss_2|$

Note 1: Here we conservatively use the write and read sets $(W(ss_1), R(ss_2))$ for simplicity. Our implementation takes into consideration only those variables involved in the flow-dependencies that cross the partitions.
Note 2: If the all operations in the loop are *batch-safe* we can omit the *loopkey* and the *key* attribute.
Note 3: Condition **(c1)** can be relaxed to allow back edges (LFDs and LODs) within *ss2* if the edge is for a non-local variable. We have shown a stronger condition for simplicity.

Rule 2B: Splitting a Cursor Loop The rule for splitting cursor loops is a minor variant of Rule 2A, and we omit details for brevity.

Figure 11: Rule 2

tions. In such a case, we isolate the batch-safe operation by introducing an extra variable. Figure 12 shows an example and Rule 3 expresses the transformation formally.

Rule 3: Isolating Batch-Safe Expressions

Let e be a batch-safe expression in statement *stmt*. Then,

$stmt; \quad \Leftrightarrow \quad T \; v = e; \quad stmt';$
where $stmt' = SUBS(stmt, e, v)$ and $T =$*type-of*(e);

Variable *assignment* is not batch-safe in general, *e.g.,* assignment to a global variable. However, assignments to different locations (e.g., different rows of a cursor loop) can be

for each *t* in *r* order by *r.key* loop
  print(**q(t.c)**); *// print() is not batch-safe*
end loop;

for each *t* in *r* order by *r.key* loop
  T v = q(t.c); *// where T=type-of(q(...))*
  print(v);
end loop;


After loop split

for each *t* by ref in *r* loop *// order-by removed with Rule 1D*
  t.v = q(t.c);
end loop;
for each *t* in *r* order by *r.key* loop *// order-by is needed*
  print(t.v);
end loop;

Figure 12: Separating batch-safe operation

for each t by ref in sales loop
  if (t.brcode == 58)
    t.brcode = 1;
    q(t.item, t.qty, t.brcode);
  end if
end loop;

for each t by ref in sales loop
  // Using a control variable remember the branching decision
  boolean cv = (t.brcode == 58);
  (cv==true)? t.brcode = 1;
  (cv==true)? q(t.item, t.qty, t.brcode);
end loop;

Now, we can apply Rule-2 and split the loop. The conditional invocation of **q** can then be batched using Rule 1C.

Figure 13: Splitting a loop after transforming control dependencies to flow-dependencies

performed in any order and hence batch-safe in the context of *loop local* variables that are converted to attributes of *loop local table* by Rule 2. If the return value of a query is assigned to a non-local variable, applying Rule 3 introduces a new loop local variable and thus enables batching the query.

4.4 Control to Flow Dependencies (Rule 4)

Conditional branching (*if-then-else*) and while loops lead to control dependencies. If the predicate evaluated at a conditional branching statement $s1$ determines whether or not control reaches statement $s2$, then $s2$ is said to be control dependent on $s1$. During loop split, it may be necessary to convert the control dependencies into flow dependencies [6]. Figure 13 shown an example. Rule 4 specifies the transformation formally.

Rule 4: Converting control-dependencies to flow-dependencies

if (p) { ss_1 } else { ss_2 }

boolean $cv = p$;
ss

where $ss[i] = (cv == true)?ss_1[i], 1 \leq i \leq |ss_1|$ and
$ss[k+j] = (cv == false)?ss_2[j], 1 \leq j \leq |ss_2|, k = |ss_1|$

4.5 Reordering Statements (Rule 5)

Consider the example in Figure 14. Assume we want to batch the query invocation **q(category)** in statement s1. We cannot directly split the loop so as to batch s1 because there is a loop-carried flow-dependency from s3 to s1 (and

Original Program

```
s0:  while (category != null) loop
s1:       int icount = q(category); // Query to batch
s2:       sum = sum + icount;
s3:       category = getParentCategory(category);
     end loop;
```

⇕

After Order Reversal

```
s0:  while (category != null) loop
s1':      int category_stub = category;
s3:       category = getParentCategory(category);
s1:       int icount = q(category_stub);
s2:       sum = sum + icount;
     end loop;
```

⇕

After Loop Split

```
TABLE(...) r;
int loopkey = 0;
while (category != null) loop
     RECORD(...) t;
     int category_stub = category;
     category = getParentCategory(category);
     t.key = loopkey++;
     t.category_stub = category_stub;
     r.addRecord(t);
end loop;

for each t by ref in r loop
     t.icount = q(t.category_stub);
end loop;

for each t in r order by key loop
     sum = sum + t.icount;
end loop;
```

Figure 14: Reordering statements to satisfy conditions c1 of Rule-2A

Rule 5A: Reordering Independent Statements
Two statements can be reordered if there exists no dependency between them.

$s_1; s_2;$ where $indep(s_1, s_2)$ $\iff$ $s_2; s_1;$

Rule 5B: Reordering with Anti-Dependency
In the presence of an anti-dependency, statements can be reordered by using an extra variable.

$s_1; s_2;$
where $s_1 \xrightarrow{AD} s_2 \wedge \neg(s_1 \xrightarrow{FD|OD} s_2)$
⇕
$ss_{stub}; s_2; s'_1;$

where ss_{stub} is the sequence of assignment statements to preserve the values read by s_1 in stub variables. s'_1 is same as s_1 except that it uses the stub variables. Formally,
(i) ss_{stub} is *s.t.* $\forall v \in R(s_1) \bigcap W(s_2)$, the statement $v' = v$; is in ss_{stub}
(ii) $s'_1 = SUBS(s_1, R(s_1) \bigcap W(s_2), map : v \rightarrow v')$

Figure 15: Rule 5: Reordering Statements

also to the loop predicate), which violates condition **(c1)** of Rule 2A. Statement s3, which appears after s1, writes a value and statement s1 reads it in a subsequent iteration. We therefore reverse the order of statements s1 and s3 before splitting the loop (Figure 14). Intuitively, we first collect all the categories in the hierarchy and then perform a batched invocation of the query that computes the item counts for the categories. The basic rules that allow us to reorder statements are specified in Rule 5.

Rule 6: Batching Across Nested Loops

Let s be a table valued attribute of table r and S be the schema of $r.s$, *i.e.,* $S = schema(r.s)$.

6A. No Return Value
for each t in r loop
  $qb((t.c_1, \ldots t.c_n) \times t.s)$;
end loop;
⇕
$qb(\Pi^d_A(\mu_s(r)))$ where $A = \{c_1, \ldots c_n\} \bigcup S$

Rule 6B: With Return Value
for each t in r loop
  $\mathcal{M}_{c_1=c'_1 \ldots c_n=cn'}(t.s, qb(t.s))$
end loop;
where qb is a *pure* function.
⇕
$rs = \mu_s(r)$;
$\mathcal{M}_{c_1=c'_1 \ldots c_n=cn'}(rs, qb(\Pi_S(rs)))$;
$r = \nu_{S \rightarrow s}(rs)$;

Figure 16: Rule 6: Batching Across Nested Loops

4.6 Batching Across Nested Loops (Rule 6)

Loops in a program may be nested within other loops and form a hierarchy. The query or update operation we are interested in batching may lie anywhere in the loop hierarchy. It is often desirable to batch the query or update operation *w.r.t.* as many ancestor loops as possible. The aim here is to make fewest possible calls to the expensive operation, in other words, to make the size of the batch in each invocation as large as possible.

Consider a loop L_c nested under loop L_p and a query q inside L_c. When the child loop (L_c) is split using Rule 2, a TABLE valued local variable (*loop local table*) is introduced in the parent loop (L_p). With the application of Rule 1, q is pulled out of L_c and is replaced by qb that lies directly inside L_p. In turn, when the parent loop is split, the *loop local table* of the child loop becomes a TABLE valued attribute (nested table) in the parent's *loop local table*. We now perform a second level batching of qb *w.r.t.* L_p by unnesting the *loop local table* of L_p. Rule 6 enables this transformation. Intuitively, we first try to pull the statement out of the inner most loop enclosing it and then out of the next (higher) level loops. In Rule 6 we make use of the *nest*(ν) and *unnest*(μ) operators of nested relational algebra [1]. Below we give a brief description of these operators and refer to [1] for the formal definitions.

Nest: The *nest* operator ($\nu_{S \rightarrow s}(r)$) groups the tuples of r on attributes *schema(r)* $- S$, then for each group forms a single tuple with a relation valued attribute s containing the S values of the tuples grouped together.

Unnest: The *unnest* operator ($\mu_s(r)$), where s is a relation valued attribute of r, performs the inverse operation of *nest*.

$\mu_s(r) = \bigcup_{t \in r}(\Pi_R\{t\} \times t.s)$, where R is the set of all attributes in *schema(r)* excluding s.

Though we use a nested relational model for the loop local tables, our techniques are easy to implement on any RDBMS by storing the nested tables separately.

4.7 Correctness

Let P_L be a program fragment that matches the LHS of a rule and P_R be the program fragment instantiated by the corresponding RHS. Let p be the program position at which P_L begins. Let (G, S) be the pair of *any* valid program and system states at p. The program state G comprises of values for all variables accessible at the program position p and the system state S comprises of the state of all external resources like database and file system. To prove the correctness of a transformation rule, we must show the following. If the execution of P_L on (G, S) results in the state (G', S') then the execution of P_R on (G, S) will also result in the state (G', S'). Note that we assume intermediate program and system states are not observable. This is a valid assumption in many practical applications. The correctness of many rules directly follows from the definition of *batch-safe* operation and that of *batched forms*. In some cases, we need to show the multiset equivalence of the *loop local table*, which is being updated, at the end of the execution of P_L and P_R. The correctness proof of *Rule 2* uses an argument on the values seen by each statement in the i^{th} iteration $(1 \leq i \leq n)$, where n is the number of loop iterations. (Formal proofs of correctness of all the rules can be found in Appendix A.)

5. EXPERIMENTAL RESULTS

Our rewrite rules can conceptually be used with any language. However, to implement the rules we need to perform dataflow analysis of a program and build the *data dependence graph*. For our implementation, we chose Java, since tools for its dataflow analysis are available in public domain. Our implementation uses the *SOOT* optimization framework [12]. *SOOT* uses an intermediate code representation called *Jimple* and provides dependency information on *Jimple* statements. Our implementation transforms the *Jimple* code using the dependence information. Finally, the *Jimple* code is translated back into a Java program.

Our current implementation requires that queries and updates be performed using our API layer built on top of JDBC. During rewrite we recognize these calls and transform them for batched bindings when possible. We have not yet implemented query rewriting to get batched forms and this step is done manually. The techniques for deriving batched forms of queries are well known and we plan to implement them in future.

Tables (batches) used in the rewritten procedures are constructed in-memory and transferred to the database before evaluating the batched queries. Nest/unnest and merge operations are performed on these in-memory tables.

There are no benchmarks for procedural SQL that we could use for our experiments. However, we had earlier seen three real-world applications which were facing serious performance problems due to non-set-oriented execution, which were affecting their usability. We use these scenarios for our experiments. We do not have access to the actual data used in these applications and hence we synthesized data with similar characteristics. In one case we used TPC-H data as it matched the scenario. As we cannot report timings on the actual application code, we used independent programs having only the code required for the specific scenarios. The experiments were performed on a widely used commercial database system (we call it SYS1) running on an Intel P4 (HT) PC with 1GB of RAM.

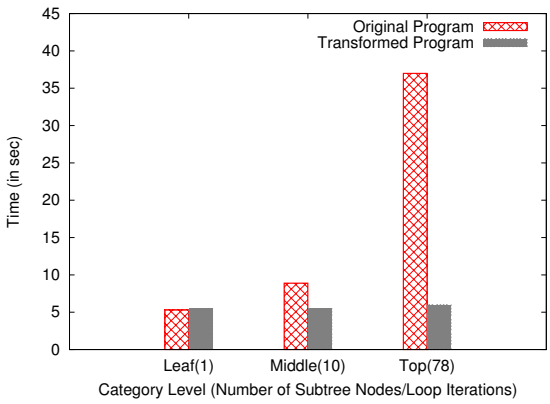

Figure 17: Experiment 1

Experiment 1: Traversal of Category Hierarchy

For this experiment, we used a program, which is a slight variant of the UDF in Figure 2. The program finds the item (part) with maximum size under a given category (including all its sub-categories) by performing a DFS of the category hierarchy. For each node (category) visited, the program queries the *item* table. The TPC-H *part* table, augmented with a new column *category-id* and populated with 2 million rows, was used as the *item* table. The *category* table had 1000 rows - 900 leaf level, 90 middle level and 10 top level categories (approximately). A clustering index was present on the *category-id* column of the *category* table and a secondary index was present on the *category-id* column of the *item* table. All relevant statistics were built. Figure 17 shows the performance of the program before and after rewrite.

For the non-batched query on the *item* table, SYS1's default choice was to use the secondary index. This plan results in a lot of random I/O, and we found an alternative plan, which performs a sequential scan takes less time since the entire relation is brought into memory on the first invocation, and there is no I/O on subsequent invocations. Since this plan was found to be cheaper, we enforced it using optimizer hints. Figure 17 compares the time taken by the best plan for the original program with the batched version. The batched version, in this case, performed a *group-by* followed by a join, whereas the original program repeatedly executed a query that performed selection followed by group by; as a result the batched version showed much better performance.

In transforming the program, Rule-5 (reordering), Rule-2 (loop splitting) and Rule-1 (batching) were applied in that order (see Appendix B). There was a 12.5% increase in the program size (lines of code) due to the transformation.

Experiment 2: ESOP Management Application

For this experiment, an application meant for managing stock option grants of multiple organizations was considered. During each *upload* operation, a large number of records fed in the form of a delimited file were processed. The application performed a mix of queries, inserts and updates. A brief outline of the program logic is given below to indicate the complexity of control-flow involved.

For each record read from the input file, the program performs validation and pre-processing of the fields and then queries the *options* table to check if a record for the person

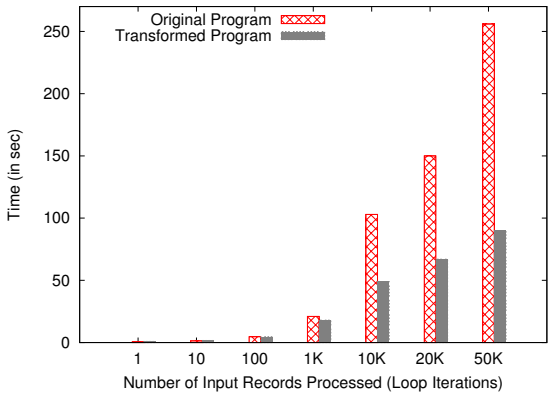

Figure 18: Experiment 2

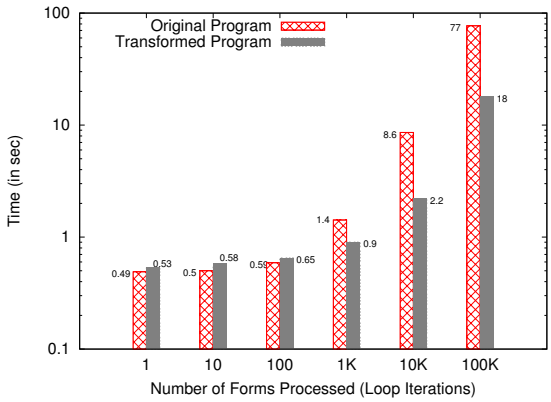

Figure 19: Experiment 3

already exists. The query predicate is parameterized on the values read from the input record. If a record is present, the old values of the various fields and the *internal-emp-id* are obtained as part of the same query. Further, the *contactinfo* table is queried using the *internal-emp-id* to obtain contact info fields. If the input record being processed has empty values for any of the fields, the old values (when present) are copied to those fields. Finally, new records are inserted or existing records updated in both *options* and *contactinfo* tables. Figure 30 in Appendix C shows the procedure.

Our rewrite techniques turn the iterative selections into an outer join and perform batched update and inserts. Figure 18 compares the performance of the rewritten program with the original program for varying number of input records.

In transforming the program, Rule-4 (control-dependencies to flow-dependencies), Rule-5 (reordering), Rule-2 (loop splitting) and Rule-1 (batching) were applied. After transformation there was a 17% increase in the program size.

Experiment 3: Value Range Expansion

In this application, data about *forms* issued to various agents would arrive in the format *(agent-id, start-form-number, end-form-number)*. The program (shown in Figure 31 of Appendix C) would iterate over all the *form issue* records, expand the *issue* range and populate the *forms-master* table with entries corresponding to each individual form. The purpose was to be able to update and track the status of each individual form subsequent to its issue. The original program had an outer loop iterating over the *form issue* records and an inner loop iterating over the range *(start-form-number, end-form-number)*. An INSERT operation was performed inside the inner loop. The transformed program could pull the *insert* operation out of both the loops and perform a batched insert. The running times of the original and transformed program are shown in Figure 19. The batched version performs much better for large batch sizes. For small batch sizes (less than 1K) the computational overheads due to batch creation and nest/unnest operations cause the batched version to perform marginally slower than the original program.

In transforming the program, Rule-5 (reordering), Rule-2 (loop splitting), Rule-1 and Rule-6 (batching) were applied. The increase in program size was 16.5%.

Time Taken for Program Transformation

Although the time taken for program transformation is usually not a concern (as it is a one-time activity), we note that, in our experiments the program transformation took very little time (less than a second).

6. DISCUSSION

Implementation of Parameter Batches: The *loop local table*, discussed in Section 4, serves to hold the parameter batch with which the *batched form* of an operation is invoked. Though small batches can be held in memory, in general we may need to materialize the batches and the cost of materialization must be taken into account while deciding to batch an operation. However, for procedures that run entirely inside the database engine (*e.g.,* UDFs) it may be possible to avoid materialization of batches by constructing a single dataflow containing both relational and *procedural nodes*. Our loop splitting transformation is designed to facilitate such an approach. Appendix A contains few additional rules that can be used for (i) avoiding creation of intermediate batches by passing a relational expression, instead of a table, to the batched forms and (ii) mapping program statements that perform simple and inexpensive operations (*e.g., expression evaluation and variable assignment*) to operators that work on sets. The later of these helps in eliminating loops such as the one left over in Figure 5. When a single dataflow is thus built, code that cannot be mapped to relational operators is executed by *procedural nodes* in the dataflow. Variable bindings inside such nodes are obtained from input tuples. Depending on the specific code, each *procedural node* may be blocking or non-blocking. The *loop local tables* become streams in the pipeline.

Cost-Based Choice of Queries to Batch: The operations to batch must be chosen taking into account the potential benefits and costs associated with the transformed code. Our current implementation requires the user to specify which operations to consider for batching. The important parameters on which this decision depends include (i) cost model for the operation as a function of batch size, (ii) expected number of iterations of the program loop and (iii) branch probabilities for the branching statements *(if-then-else)* in the program. Automating the choice of operations to batch is a future work.

Applicability of Transformation Rules: The proposed set of transformation rules succeed in rewriting fairly complex programs for batched bindings. However, it may not be always be possible to rewrite a program to batch the invo-

```
s0: while(eid ! = NULL) loop
s1:     mgr =SELECT manager
              FROM emp WHERE empid=eid;
s2:     idx = SELECT perfindex FROM rating
              WHERE reviewer=mgr and reviewed=eid;
s3:     sumidx += idx;
s4:     eid = mgr;
    end loop;
```

Figure 20: Cyclic Flow-Dependencies

cation of a specific operation. As an example, consider the program shown in Figure 20. Our transformation rules can batch the query in statement s2 but not the one in statement s1. The query in statement s1 lies on the flow-dependency cycle s1 $\xrightarrow{FD}$ s4 $\xrightarrow{LFD}$ s1 and hence cannot be batched. If an operation has no side-effects, in some cases, it may be possible to derive a superset of the parameters and batch the operation even if it is on a flow-dependency cycle. But such an approach is beyond the scope of our transformation rules. Similarly, in the DDG of Figure 7, the query in statement s8 is batchable, whereas the query in statement s10 is not batchable due to the presence of a flow-dependency cycle.

7. RELATED WORK

Queries such as those shown in Figures 1 and 2 can be thought of as nested queries with complex inner blocks. However, the inner block in such cases may contain multiple subqueries embedded in procedural code. Hence, standard decorrelation techniques such as [7, 4, 11, 3, 2] cannot be directly applied. The techniques we propose in this paper help in rewriting of procedural code so as to enable set-oriented evaluation of the embedded subqueries through batched bindings. This is an essential step in decorrelation [11]. Further optimizations such as pipelining the output of the expression that produces the parameter batch into the expression that consumes it are possible and we are investigating these optimizations. Graefe [5] highlights the benefits of batched bindings for speeding up index nested loops joins. Batched bindings not only help in performing I/O efficiently but can also make it possible to employ a set-oriented strategy at the operator level.

Lieuwen and DeWitt[8] consider the problem of optimizing set iteration loops in database programming languages. Their techniques can convert nested set iteration loops into joins. However, their work does not address the issue of batching procedure calls. Their transformation rules work on a restricted syntax. The program transformation rules in this paper can work with complex control-flow including *if-then-else* and *while* loops and can even deal with loop carried dependencies (Rule-5). Rule-2 in this paper is a more general version of Rule T4 in [8].

Some of the program transformation techniques we employ are derived from those proposed in the area of parallelizing compilers [6, 10]. However, the problem of batching differs from the problem of parallelizing in the following ways: *(i)* Presence of flow-dependencies (described in Section 3) does not allow parallelization. However, batching is possible even if the order of two operations cannot be changed due to flow-dependencies *(ii)* As the aim of batching is to improve the performance of expensive I/O bound queries and other database operations, it may be acceptable for the transformations to introduce additional CPU operations or consume extra memory to make the batching possible. However, such approaches do not generally yield significant benefits in the context of parallelizing the instructions and are not considered to the best of our knowledge.

8. CONCLUSION

Procedural extensions to SQL and SQL extensions to programming languages offer new challenges and opportunities for query optimization. To deal with these challenges query optimization must be augmented with program analysis and transformation techniques developed for procedural languages. The work presented in this paper enables automatic rewriting of database applications for set-orientation execution of database operations. Our implementation and performance study show the practicality and usefulness of the the proposed techniques. Our work is a step towards combining query optimization with program analysis and transformation techniques; we believe this combination will give significant benefits for database applications.

Acknowledgments

We would like to thank Yogesh Murarka for the many discussions on techniques used in parallelizing compilers.

9. REFERENCES

[1] L. S. Colby. A Recursive Algebra and Query Optimization for Nested Relations. *SIGMOD Rec.*, 18(2), 1989.

[2] U. Dayal. Of Nests and Trees: A Unified approach to Processing Queries That Contain Nested Subqueries, Aggregates, and Quantifiers. In *VLDB*, 1987.

[3] C. A. Galindo-Legaria and M. M. Joshi. Orthogonal Optimization of Subqueries and Aggregation. In *ACM SIGMOD*, 2001.

[4] R. A. Ganski and H. K. T. Wong. Optimization of Nested SQL Queries Revisited. In *SIGMOD*, 1987.

[5] G. Graefe. Executing Nested Queries. In *10th Conference on Database Systems for Business, Technology and the Web*, 2003.

[6] K. Kennedy and K. S. McKinley. Loop Distribution with Arbitrary Control Flow. In *Proceedings of Supercomputing*, 1990.

[7] W. Kim. On Optimizing an SQL-like Nested Query. In *ACM Trans. on Database Systems, Vol 7, No.3*, 1982.

[8] D. F. Lieuwen and D. J. DeWitt. A transformation based approach to optimizing loops in database programming languages. In *ACM SIGMOD*, 1992.

[9] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann.

[10] L. Rauchwerger and D. Padua. Parallelizing While Loops for Multiprocessor Systems. In *Proc. of the 9th International Parallel Processing Symposium*, 1995.

[11] P. Seshadri, H. Pirahesh, and T. C. Leung. Complex Query Decorrelation. In *ICDE*, 1996.

[12] Soot: A Java Optimization Framework *www.sable.mcgill.ca/soot*.

APPENDIX

A. EXTRA RULES AND CORRECTNESS

In this section we present a few additional program transformation rules, and proofs of correctness for all the transformation rules.

A.1 Extra Rules

Rule 7 in Figure 21 and Rule 8 in Figure 22 are useful in replacing loops containing simple expressions and assignment with relational operators and avoiding materialization of intermediate results. As mentioned in Section 6 these rules can be used to build a single dataflow for queries, which invoke UDFs that run inside the database. The *trivial batched form* (see Figure 6) constructs the *return table* iteratively. Rule 9 in Figure 23 is useful to convert this into a set valued expression. The example in Appendix B illustrates the use of this rule.

Rule 7

FOR EACH t BY REF IN r [ORDER BY key] LOOP
    t.b = *arith-expr*$(t.a_1, t.a_2, \ldots t.a_n)$;
END LOOP;

$\Updownarrow$

$r = \Pi_{A, \textit{arith-expr}(t.a_1, t.a_2, \ldots t.a_n) \text{ as } b}(r)$,
where A=*schema(r)*$-\{b\}$

Figure 21: Loops with Arithmetic Expressions and Assignment

Rule 8

Let expr1 be a side-effect free expression (*e.g.,* a query).

TABLE r1 = expr1();
TABLE r2 = expr2(r1); $\Leftrightarrow$ TABLE r2 = expr2(expr1);
dead(r1) // r1 unused hereafter

Figure 22: Rule for Avoiding Materialization

Rule 9

TABLE result;
FOR EACH t [BY REF] IN r [ORDER BY key] LOOP
    result.addRecord($(c_1, c_2, \ldots c_n)$);
    // where each c_i is a function of attributes of t.
END LOOP;
RETURN result;

$\Updownarrow$

RETURN SELECT $c_1, c_2, \ldots c_n$ FROM r;

Figure 23: Rule for RETURN Statement

A.2 Proof of Correctness

The *program state* G comprises of values for all variables accessible at a program position p and the *system state* S comprises of the state of all external resources like database and file system. Let P_L be a program fragment that matches the LHS of a rule and P_R be the program fragment instantiated by the corresponding RHS. Let p be the position in the program at which P_L begins. Let (G, S) be the pair of *any* valid program and system states at p. To prove the correctness of a transformation rule, we must show the following. If the execution of P_L on (G, S) results in the state (G', S') then the execution of P_R on (G, S) will also result in the state (G', S').

- **Rule 1A(i):** In Rule 1A(i), both P_L and P_R do not modify the program state (as we use a call by value semantics and there are no global variables).

  Consider the multiset $S = \Pi^d_{c_1, c_2, \ldots c_m}$ with which qb is invoked. Let S' be the multiset of tuples constructed from parameters passed to each invocation of q inside the loop. We can see that S is multiset equivalent to S'. Now the equivalence of the two program fragments follows directly from the definition of *batch-safe* operation (when an operation is batch-safe the final system state depends only on the set of parameters and not the order of invocations).
- **Rule 1A(ii):** Proof is similar to that of Rule 1A(i).
- **Rule 1B:** The only program state P_L and P_R modify is the table r. Let the initial state (state at the point where P_L/P_R begins) of the table be r_{init}. Let r' be the state the table reaches if P_L is executed and r'' be the state the table reaches if P_R is executed. We show r' and r'' to be multiset equivalent.

  Since q is a scalar query, from the definition of *batched forms* it follows that, during merge, each tuple in r_{init} matches with exactly one tuple in the result of the batched invocation, $qb(\Pi_{c_{r1}, \ldots c_{rm}}(r))$. As a result, *(i)* cardinalities of r', r'' and r_{init} are equal, i.e., $|r'| = |r''| = |r_{init}|$ and *(ii)* For each tuple $t \in r_{init}$, there exists a distinct tuple $t' \in r'$ and a distinct tuple $t'' \in r''$ such that t, t' and t'' have the same values for all attributes except (possibly) the updated attributes *viz.,* $c_{w1}, c_{w2}, \ldots c_{wn}$.

  Let t_r be the tuple in the result of the batched invocation (qb) that matches (during merge) with tuple t of r_{init}. Let $(v_1, v_2, \ldots v_m)$ be the values of attributes $c_{r1}, c_{r2}, \ldots c_{rm}$ of t. Therefore, attributes $c_{r1}, c_{r2}, \ldots c_{rm}$ of t_r must also have the values $(v_1, v_2, \ldots v_m)$. Let $(w_1, w_2, \ldots w_n)$ be the values of the remaining attributes (named $c_{w1'}, c_{w2'}, \ldots c_{wn'}$) of t_r. From the definition of *batched forms*, we have $(w_1, w_2, \ldots w_n) = q(v_1, v_2, \ldots v_m)$. *i.e.,* the tuple resulting from the merge (t'' in r'') has values assigned from $q(v_1, v_2, \ldots v_m)$ for its attributes $c_{w1}, c_{w2}, \ldots c_{wn}$. From the LHS of the rule, it is clear that the corresponding tuple $t' \in r'$ also has the values of $q(v_1, v_2, \ldots v_m)$ assigned for its attributes $c_{w1}, c_{w2}, \ldots c_{wn}$. This makes $t' = t''$ and hence $r' = r''$.

  Since q is a *pure* function the system state remains unaffected by both P_L and P_R.
- **Rule 1C:** Proof is similar to the proof for 1B.
- **Rule 1D:** The equivalence directly follows from the definition of batch-safe operation.
- **Rule 2:** Let P_L be the program fragment matching the LHS of the rule and P_R be the program fragment instantiated by the RHS. Let us call the *while* loop in P_L as L, the first loop of P_R as L_1 and the second loop of P_R as L_2.

  First, we note that there exists a one-to-one correspondence between statements in ss_2 of L and statements in ss'_2 of L_2. This follows from the construction of ss'_2. For every statement s in ss_2, the corresponding

statement s' in ss_2' performs exactly the same set of operations, the only change being in the names of the local variables (uses $r.a$ instead of a).

Let v be the value of a variable a at statement $s \in ss_1$ in the i^{th} iteration of L. Because there are no back edges from ss_2 to ss_1, we can see that v will be the value of a at s' (the statement corresponding to s) in the i^{th} iteration of L_1.

Similarly, if v is the value of a variable a at statement $s \in ss_2$ in the i^{th} iteration of L, we can see that v will be the value of $r.a$ at s' (the statement corresponding to s) in the i^{th} iteration of L_2.

Since there are no inter-statement dependencies involving external system state, the output and change in system state produced by s and the corresponding statement s' will be the same. Further, for every non local variable gv, if the last assignment in P_L was made by a statement s in the i^{th} iteration of L, then in P_R the last assignment will be made by the corresponding statement s' in the i^{th} iteration of L_1 or L_2. The only additional change introduced by P_R to the program state is the new variable *loopkey*, which is not used after the point where P_R ends and hence does not affect the program. Hence, the execution of both the P_L and P_R result in equivalent program and system states.

- **Rules 3, 4 and 5:** The equivalence of these rules is straight forward to infer.
- **Rule 6A:** As in the proof for *Rule 1A*, we can observe that the multiset of parameters passed to qb in P_R is equivalent to the multiset of parameters passed over all the iterations of P_L. Hence, from the definition of the batch-safe operation, the equivalence holds.
- **Rule 6B:** The proof is similar to that of *Rule 1B*.
- **Rules 7, 8 and 9:** The equivalence of these rules is straight forward to infer.

B. CONTROL ALGORITHM FOR RULE APPLICATION

Rewriting a program for set-orientation involves the following steps. (i) Identify iteratively invoked query execution statements (ii) Decide whether it is beneficial to batch the query execution and the ancestor loop (in the hierarchy of loops enclosing the statement) with respect to which the statement must be batched and (iii) Rewrite the program by *systematically* applying the transformation rules presented in this paper.

Identifying the query execution statements in a loop is usually straight forward. However, the decision on whether a statement should be batched and the level in the loop hierarchy up to which the statement must be pulled-out, requires a cost-based analysis. Some of the parameters needed for cost-based analysis are discussed in Section 6. We plan to address cost-based analysis in the future. In this paper, we assume these two inputs (the query invocation to be batched and the ancestor loop up to which the query must be pulled-out) are available from the user. Given a query execution statement and a loop with respect to which the statement must be batched, the transformation rules presented in this paper can be used to rewrite the program. However, it is important to apply the rules in a systematic way so as to achieve the goal of batching the given statement.

Inputs:
- **s**: The query execution statement to be batched
- **l**: A program loop *w.r.t.* which the query must be batched (*i.e.*, **s** must be pulled out of **l**). **s** may be present directly inside **l** or within a descendant loop of **l**.

Note: As discussed in Section 6, deciding the above inputs in a cost-based manner is a future work.

Goal:
Rewrite the program to batch the query execution in statement **s** *w.r.t.* loop **l**.

```
procedure batch(Stmt s, Loop l)
begin
    Let lp be the loop which directly encloses s.
    // Pull s out of lp. Let the batched statement be sb.
    sb = do-batching(s, lp);
    if (lp != l)
        Let lpp be the parent loop of lp
        batch(sb, lpp);
end;

procedure do-batching(Stmt s, Loop l)
begin
    Rewrite s using Rule-3 so that s is a simple assignment
    with only the query invocation expression on its RHS.

    If s is control dependent on any statement inside loop l (other
    than the loop predicate), convert the control-dependency to flow
    dependency (using Rule-4).

    Reorder the statements in l to satisfy pre-conditions for Rule-2.
    (making use of Rule 5)

    Applying Rule-2 split the loop l at the program points before
    and after s.

    Batch the query execution using Rule 1 or Rule 6.

    Return the reference to statement sb, the batched form of s.
end;
```

Figure 24: Control Algorithm for Rule Application

In Figure 24 we give an algorithm for applying the rules so as to batch a given query execution statement *w.r.t.* a given ancestor loop enclosing it. The procedure *batch* recursively pulls out the given statement, starting from the inner most loop enclosing it. Procedure *do-batching* performs the actual task of rewriting by applying the rules. First, we apply Rule-3 on the statement and ensure the RHS contains only the query execution expression. We then convert all the control-dependencies to flow-dependencies by applying Rule-4. This allows us to treat the entire body of the loop as a *basic block* (a straight-line sequence of statements with no branches into or out of the sequence). We perform a reordering of the statements (if needed) to satisfy the pre-conditions for Rule-2 and then split the loop before and after the query execution statement. This leaves the query execution statement in a loop by itself - a form in which we can apply Rule-1 or Rule-6 and make use of the batched form. Rule-1 gets applied for the inner most loop enclosing the statement and Rule-6 gets applied for the higher level loops.

We now illustrate the transformation of the UDFs in Figure 1 and Figure 2 as the rules get applied following the batching procedure in Figure 24. We call these two UDFs as UDF-1 and UDF-2 respectively. Here, we assume that

```
TABLE count-offers-batched(TABLE r1)
DECLARE
    TABLE result;
BEGIN
    FOR EACH t IN r1 LOOP
        FLOAT amount-usd;
        INT count-offers; // The return value named after the function
        IF (t.curcode == "USD")
            amount-usd := t.amount;
        ELSE
            amount-usd := t.amount * (SELECT exchrate FROM curexch
                                      WHERE ccode = t.curcode);
        END IF
        count-offers := SELECT count(*) FROM buyoffers
                        WHERE itemid = t.itemcode AND
                                price >= amount-usd;
        result.addRecord((t.itemcode, t.amount, t.curcode, count-offers));
    END LOOP;
    RETURN result;
END;
```

Figure 25: UDF-1: Trivial Batched Form

```
TABLE count-offers-batched(TABLE r1)
DECLARE
    TABLE result;
BEGIN
    FOR EACH t IN r1 LOOP
        FLOAT amount-usd; INT count-offers;
        BOOLEAN cond1; FLOAT exchrate;
        cond1 := (t.curcode == "USD");
        cond1 == true? amount-usd := t.amount;
        cond1 == false? exchrate := SELECT exchrate FROM curexch
                                    WHERE ccode = t.curcode;
        cond1 == false? amount-usd := t.amount * exchrate;
        count-offers := SELECT count(*) FROM buyoffers
                        WHERE itemid = t.itemcode AND
                                price >= amount-usd;
        result.addRecord((t.itemcode, t.amount, t.curcode, count-offers));
    END LOOP;
    RETURN result;
END;
```

Figure 26: UDF-1: After Applying Rules 3 and 4

every query needs to be batched (when possible) and with respect to all the loops enclosing it.

B.1 Rewriting UDF-1

- **Generate the Trivial Batched Form:** First, we generate the *trivial batched form* of the procedure as explained in Section 2.3. Figure 25 shows the resulting procedure.
- **Isolate the Query Execution:** We isolate the query (expression) to be batched using Rule-3 and convert the control-dependencies to flow-dependencies using Rule-4. The resulting procedure after applying these two rules is shown in Figure 26.
- **Split the Loop:** Split the loop (by applying Rule-2) before and after the query execution statements. In this example, the pre-conditions for Rule-2 are directly satisfied and we do not need to reorder any statements. However, in some cases we may need to reorder the statements using Rule-5 to satisfy the pre-conditions for Rule-2. Figure 27 shows the resulting program.
- **Replace Loops with Batched Calls:** Apply Rule-1D to remove the order-by clauses around batch-safe operations and then replace the iterations with batched calls using Rules 1B and 1C. We further apply Rule 9 (Figure 23) for the RETURN statement. The resulting program is given in Figure 28. Earlier, in Figure 5 we had shown this batched form with minor simplifications for readability.

```
TABLE count-offers-batched(TABLE r1)
DECLARE
    TABLE result; INT loopkey = 0;
    TABLE (key, itemcode, amount, curcode, cond1, exchrate,
           amount-usd, count-offers) r2;
BEGIN
    FOR EACH t IN r1 LOOP
        FLOAT amount-usd; BOOLEAN cond1;
        RECORD rec;

        cond1 := (t.curcode == "USD");
        cond1 == true? amount-usd := t.amount;
        rec.key = loopkey++;
        rec.itemcode = t.itemcode;
        rec.amount = t.amount;
        rec.curcode = t.curcode;
        rec.cond1 = cond1;
        rec.amount-usd = amount-usd;
        r2.addRecord(rec);
    END LOOP;

    FOR EACH t BY REF IN r2 ORDER BY key LOOP
        t.cond1 == false? t.exchrate :=
                    SELECT exchrate FROM curexch
                    WHERE ccode = t.curcode;
    END LOOP;

    FOR EACH t BY REF IN r2 ORDER BY key LOOP
        t.cond1 == false? t.amount-usd := t.amount * t.exchrate;
    END LOOP;

    FOR EACH t BY REF IN r2 ORDER BY key LOOP
        t.count-offers := SELECT count(*) FROM buyoffers
                          WHERE itemid = t.itemcode AND
                                  price >= t.amount-usd;
    END LOOP;

    FOR EACH t BY REF IN r2 ORDER BY key LOOP
        result.addRecord((t.itemcode, t.amount, t.curcode,
                          t.count-offers));
    END LOOP;

    RETURN result;
END;
```

Figure 27: UDF-1: After Loop Split

B.2 Rewriting UDF-2

UDF-2 (Figure 2) contains two queries, one in statement s8 and the other in statement s10. As mentioned in Section 6, we cannot batch the query in statement s10 due to cyclic flow dependence. However, we can batch the query in statement s8 with respect to the WHILE loop (of s5) as well as the outermost cursor loop, which iterates over all the parameters (in the *trivial batched form*).

In Figure 2, observe that splitting the WHILE loop (of s5) before and after the query execution statement (of s8) is not directly possible due to the loop-carried dependencies from s11 and s12 to s5, s6 and s7, which violate pre-condition c1 of Rule-2. We therefore, reorder of statements by moving statements s8 and s9 past s12 (using Rule-5). We then split the WHILE loop and batch the query execution. The batched query execution is further pulled out of the outermost cursor loop in the *trivial batched form* using Rule-6.

Figure 29 shows the final batched form of UDF-2. The functions NEST and UNNEST implement the *nest* and *unnest* operations discussed in Section 4.6 and take the corresponding arguments. NEST takes as its arguments the table, columns to be nested and the name for the resulting table-valued column. Similarly, the UNNEST method takes the

```
TABLE count-offers-batched(TABLE r1)
DECLARE
    TABLE (key, itemcode, amount, curcode, cond1, exchrate,
           amount-usd, count-offers) r2;
    INT loopkey = 0;
BEGIN
    FOR EACH t IN r1 LOOP
        FLOAT amount-usd; BOOLEAN cond1;
        RECORD rec;
        cond1 := (t.curcode == "USD");
        cond1 == true? amount-usd := t.amount;
        rec.key = loopkey++;
        rec.itemcode = t.itemcode;
        rec.amount = t.amount;
        rec.curcode = t.curcode;
        rec.cond1 = cond1;
        rec.amount-usd = amount-usd;
        r2.addRecord(rec);
    END LOOP;

    MERGE INTO r2 USING sq1b(b1) AS sq1b
    ON (r2.curcode = sq1b.curcode) WHEN MATCHED THEN
    UPDATE SET exchrate = sq1b.exchrate;

//  where the parameter batch b1 is constructed as:
//      SELECT distinct curcode FROM r2 WHERE cond1=false;
//  and the batched form sq1b(b1) is defined as:
//      SELECT b1.curcode, c.exchrate
//      FROM b1 JOIN curexch c ON b1.curcode=c.ccode;

    FOR EACH t BY REF IN r2 ORDER BY key LOOP
        t.cond1 == false? t.amount-usd := t.amount * t.exchrate;
    END LOOP;

    MERGE INTO r2 USING sq2b(b2) AS sq2b
    ON (r2.itemcode=sq2b.itemcode AND r2.amount-usd=sq2b.amount-usd)
    WHEN MATCHED THEN
    UPDATE SET count-offers = sq2b.count-offers;

    where b2 = SELECT distinct itemcode, amount-usd FROM r2;
    and sq2b(b2) = SELECT b2.itemcode, b2.amount-usd,
                          count(o.itemcode) AS count-offers
                   FROM b2 LEFT OUTER JOIN buyoffers o
                   ON o.itemid = b2.itemcode AND
                       o.price >= b2.amount-usd
                   GROUP BY b2.itemcode, b2.amount-usd;

    RETURN SELECT itemcode, curcode, amount, count-offers FROM r2;
END;
```

Figure 28: UDF-1: The Final Batched Form

table and the name of the table-valued column that needs to be unnested.

C. PROCEDURES USED IN EXPERIMENTS

This section gives pseudocode for the additional procedures used for performance evaluation (Section 5). Figure 30 shows the procedure for experiment-2 and the procedure for experiment-3 is given in Figure 31. The logic implemented by these procedures was explained earlier, in Section 5.

D. JAVA API AND CODE PATTERNS

As mentioned in Section 5, we implemented the transformation rules for Java because tools for Java program analysis are available in public domain. We make use of the *SOOT* optimization framework for obtaining data dependence information. To simplify the task of recognizing query execution statements and code patterns that match a rule, our current implementation requires that queries and updates be performed using our API layer built on top of JDBC. During rewrite, we recognize these calls and transform them for batched bindings when possible. In this section, we give the

```
TABLE count-items-batched(TABLE pb)
DECLARE
    TABLE (key, catid, loop-table2, totalcount) loop-table1;
    INT loopkey1 = 0;
BEGIN
    FOR EACH t IN pb LOOP
        INT totalcount := 0; INT top := 0; INT stack[100];
        RECORD rec1;

        stack[top] := t.catid;
        top := top + 1;

        TABLE (key, curcat, catitems) loop-table2;
        int loopkey2 = 0;
        WHILE top > 0 LOOP
            RECORD rec2;
            top := top - 1;
            curcat := stack[top];

            // Now push all the subcategory ids onto the stack
            FOR catrec IN SELECT category-id FROM category
                          WHERE parent-category=curcat LOOP
                stack[top] := catrec.category-id;
                top := top + 1;
            END LOOP;

            rec2.key = loopkey2++;
            rec2.curcat = curcat;
            loop-table2.addRecord(rec2);
        END LOOP;

        rec1.key = loopkey1++;
        rec1.catid = t.catid;
        rec1.loop-table2 = loop-table2;
        loop-table1.addRecord(rec1);
    END LOOP;

    temp = UNNEST(loop-table1, "loop-table2");
    MERGE INTO temp USING qb(b) AS qbr(curcat, res)
    ON temp.curcat = qbr.curcat
    WHEN MATCHED THEN UPDATE SET catitems = res;

//  where the parameter batch b is constructed as:
//      SELECT distinct curcat FROM temp;
//  and the batched form qb(b) is defined as:
//      SELECT b.curcat, count(itemid) AS catitems
//      FROM b LEFT OUTER JOIN item ON category-id=curcat
//      GROUP BY curcat;
    loop-table1 = NEST(temp, schema(loop-table1.loop-table2),
                       "loop-table2");

    FOR EACH rec1 BY REF IN loop-table1 ORDER BY key LOOP
        FOR EACH rec2 BY REF IN rec1.loop-table2
                             ORDER BY key LOOP
            rec1.totalcount := rec1.totalcount + rec2.catitems;
        END LOOP;
    END LOOP;

    RETURN SELECT catid, totalcount FROM loop-table1;
END;
```

Figure 29: UDF-2: The Final Batched Form

details of our API layer and the Java code patterns, which map to constructs described earlier, in this paper. *SOOT* uses an intermediate code representation called *Jimple*. Our implementation works on *Jimple* and transforms it back to Java. Recognizing *Jimple* code patterns corresponding to each of our API calls and Java code patters is relatively straight forward and we omit the details.

- **Query/Update Execution Statements**: The class *DBI* in our implementation provides various methods for executing queries and updates.
  - *Record executeScalarQuery(int queryId, Record params)*
  - *Table executeQuery(int queryId, Record params)*
  - *int executeUpdate(int queryId, Record params)*

```
PROCEDURE emp-upload(VARCHAR filename)
DECLARE
    // Data types omitted for brevity.
    empid, clientid, iempid, optcode, optinfo, termcode, taxinfo,
    city, state, zip, operation, curtaxinfo, curcity, curstate, curzip
BEGIN
    fd := open(filename);
    linestr := readline(fd);
    WHILE (linestr ! = null) LOOP
        tokenize linestr and extract empid, clientid, optcode, . . . zip
        // some validation and pre-processing code
        if(optcode == 0)
            optinfo = ... ;
        . . .

        SELECT internl-empid into iempid, tax-info into curtaxinfo,
        FROM options
        WHERE client-id=clientid AND emp-id=empid;

        // If options has no record for the employee
        if(iempid == null) {
            operation := 1; // we must insert
            iempid := gen-new-id();
        }
        else {
            operation := 2; // we must update

            SELECT city into curcity, state into curstate, zip into curzip
            FROM contactinfo
            WHERE internal-empid=iempid;

            // Retain the current values if new ones are blank
            if(taxinfo == "")
                taxinfo := curtaxinfo;
            if(city == "")
                city := curcity;
            . . .
            . . .
        }
        if(operation == 1) {
            INSERT INTO options VALUES (iempid, clientid, ...
                                                optinfo, ... taxinfo);
            INSERT INTO contactinfo VALUES(iempid, city, state, zip);
        }
        else {
            UPDATE options set option-info=optinfo, . . . tax-info=taxinfo
            WHERE internal-empid=iempid;
            UPDATE contactinfo SET city=city, state=state, zip=zip
            WHERE internal-empid=iempid;
        }
        linestr := readline(fd);
    END LOOP;
END;
```

Figure 30: Procedure for Experiment 2

The *queryId* specifies a parameterized SQL query or update statement in a query registry. Unlike JDBC, where the query string is directly specified in the program, we make use of a registry. The query registry, in addition to the query string, also contains the manually written batched form for the query. Unlike position-based parameters in JDBC, we use named parameters. The *Record* class is used to pass parameters as name-value pairs and also to obtain the result of a scalar query. The class *Table* implements a tuple set and is used to retrieve query results and for constructing parameter batches.

- **Looping Statements**: The *WHILE* loops have a direct mapping to Java. Unlike some of the procedural languages offered by database systems (e.g., PL/SQL), Java does not have cursor loops. Each cursor loop maps to a sequence of statements in Java. Figure 32 shows the code pattern.

```
PROCEDURE expand-issued-forms(DATE issuedate)
DECLARE
    INT num;
BEGIN
    FOR EACH irec IN SELECT agent_id, start_no, end_no, issue_date
                         FROM issued_forms
                         WHERE issue_date = issuedate LOOP
        num := irec.start_no;
        WHILE (num <= irec.end_no) LOOP
            INSERT INTO forms-master VALUES (num, irec.agent_id,
                                             irec.issue_date, 'NEW');
            num := num + 1;
        END LOOP;
    END LOOP;
END;
```

Figure 31: Procedure for Experiment 3

```
Table res = DBI.executeQuery(. . .);
Iterator resIter = res.iterator();
while(resIter.hasNext()) {
    Record r = (Record) resIter.next();
    . . .
}
```

$\Longrightarrow$

```
for each r in query
    . . .
end loop;
```

Figure 32: Pattern for Cursor Loops

- **Batched Execution**: The class *Table* in our implementation can be used for constructing parameter batches row by row using the method *addRecord(Record r)*. The DBI class has methods for executing batched forms of queries by passing a batch of parameters and also for merging back the results. These methods are used by the transformed program (generated code).
  - *Table executeBatchedQuery(int queryId, Table paramBatch, FilterPred pred)*
  - *void executeBatchedUpdate(int queryId, Table paramBatch, FilterPred pred)*
- **Other Constructs**: Control flow, assignment and - other constructs in the simple procedural language of this paper have a direct mapping to Java. Conditional statements, generated when control-dependencies are converted to flow-dependencies, are Java *if* statements, where the *if-block* contains a single statement and the predicate is a boolean variable.