Rewriting Procedures for Batched Bindings

CSI 2009, Pune

S. Sudarshan and Ravindra Guravannavar
Indian Institute of Technology Bombay
Introduction

A key reason for SQL's success has been its optimizability.

- Almost every known RDBMS has a sophisticated query optimizer at its core.

- The query optimizer considers alternative execution plans for a given query, and chooses the best (or near best) plan to evaluate the query.
  - Ensures syntax-independence for most queries
    - E.g., Nested (correlated) queries and equivalent join queries are processed in the same way.
Introduction

Relational query processing has come a long way

- Extensive research over the past 3 decades since IBM's System R (1976).
  - Efficient query evaluation algorithms
  - Cost-based optimization
    - Join order selection
    - Selection of query processing algorithms
      - E.g., Hash-join Vs. Merge-join Vs. Index nested loops join
  - Query decorrelation (nested query to non-nested query)
Introduction

However, optimization of database applications as a whole has received little attention.

- Queries and updates are often intermixed with business logic expressed using procedural code.
  - Stored Procedures and User-Defined Functions (UDFs)
    - Written using procedural extensions to SQL (e.g., PL/SQL)
  - External applications
    - Queries/updates through JDBC/ODBC or language integrated querying features such as C#-LINQ.

- The database system optimizes individual query invocations separately.
Iterative Invocations of Queries

Database queries/updates are often invoked repeatedly with different parameter values in a single decision support task.

Reason 1.
Nested sub-queries (with correlated evaluation)

```sql
SELECT o_orderkey, o_custkey FROM orders
WHERE o_shipdate NOT IN (  
    SELECT l_shipdate FROM lineitem  
    WHERE l_orderkey = o_orderkey
);
```
Iterative Invocations of Queries

Database queries/updates are often invoked repeatedly with different parameter values in a single decision support task.

Reason 2.
Queries inside user-defined functions (UDFs) that are invoked from other queries

```
SELECT * FROM category
WHERE count_items(category-id) > 50;
```
Iterative Invocations of Queries

Database queries/updates are often invoked repeatedly with different parameter values in a single decision support task.

```java
int count_items(int categoryId) {
    ...
    ...
    SELECT count(item-id) INTO icount
    FROM item
    WHERE category-id = :categoryId;
    ...
}
```

Procedural logic with embedded queries
Iterative Invocations of Queries

**Reason 3**: Queries/updates inside *stored-procedures*, which are called repeatedly by external batch jobs.

**Reason 4**: Queries/updates invoked from *imperative loops* in application programs or complex procedures.

// A procedure to count items in a given category and sub-categories
int count_subcat_items(int categoryId) {
    ...
    while(...) {
        ...
        SELECT count(item-id) INTO icount FROM item
        WHERE category-id = :curcat;
        ...
    }
}
Problem with Iterative Execution

Naïve iterative execution of queries and updates is often inefficient

Causes:

- No sharing of work (e.g., disk IO)
- Random IO and poor buffer effects
- Network round-trip delays
Problem with Iterative Execution

Consider iterative execution of the query:

```sql
SELECT l_shipdate FROM lineitem WHERE l_orderkey = ?
```

- One random read for each outer tuple (assuming all internal pages of the index are cached), avg seek time 4ms.
Known Approach: Decorrelation

- Rewrite a nested query using set operations (such as join, semi-join and outer-join).
  [Kim82, Dayal87, Murali89, Seshadri96, Galindo01, ElHemali07]

- Increases the number of alternative plans available to the query optimizer.
  - Makes it possible choose set-oriented plans (e.g., plans that use hash or sort-merge join) which are often more efficient than naïve iterative plans.
Known Approach: Decorrelation

- An Example:

**A Correlated Query**

```sql
SELECT O.orderid, O.custid FROM orders O
WHERE 100 < (SELECT sum(lineprice)
             FROM lineitem L
             WHERE L.orderid=O.orderid);
```

**After Decorrelation** (Most systems do this automatically)

```sql
SELECT O.orderid, O.custid FROM orders O, lineitem L
WHERE O.orderid=L.orderid
GROUP BY O.orderid, O.custid
HAVING sum(L.lineprice) > 100;
```
Limitations of Decorrelation

Known decorrelation techniques are not applicable for:

- complex nested blocks such as user-defined functions and procedures
- repeated query executions from imperative program loops
Optimizing Iterative Calls to Stored Procedures and UDFs

Approach: Use Parameter Batching
Repeated invocation of an operation is replaced by a single invocation of its batched form.

Batched form: Processes a set of parameters
- Enables set-oriented execution of queries(updates inside the procedure
  - Possible to use alternative plans
  - Efficient integrity checks and index updates
  - Reduced network round-trip delays
Batched (Set-Orientated) Forms of Basic Operations

■ INSERT:
  - `insert into ... select ... from ...`
  - Also possible to use bulk load if semantics are acceptable

■ DELETE
  - Nested sub-query – most databases `decorrelate`.

■ UPDATE
  - `update ... from ...` (or the SQL:2003 `merge`)
### Set-Oriented Update using SQL Merge

#### GRANTMASTER

<table>
<thead>
<tr>
<th>empid</th>
<th>name</th>
<th>grants</th>
</tr>
</thead>
<tbody>
<tr>
<td>S101</td>
<td>Ramesh</td>
<td>8000</td>
</tr>
<tr>
<td>S204</td>
<td>Gopal</td>
<td>4000</td>
</tr>
<tr>
<td>S305</td>
<td>Veena</td>
<td>3500</td>
</tr>
<tr>
<td>S602</td>
<td>Mayur</td>
<td>2000</td>
</tr>
</tbody>
</table>

#### GRANTLOAD

<table>
<thead>
<tr>
<th>empid</th>
<th>grants</th>
</tr>
</thead>
<tbody>
<tr>
<td>S204</td>
<td>5000</td>
</tr>
<tr>
<td>S602</td>
<td>2600</td>
</tr>
</tbody>
</table>

merge into GRANTMASTER GM using GRANTLOAD GL on GM.empid=GL.empid when matched then update set GM.grants=GL.grants;
Effect of Batch Size on Inserts

JDBC Batched Insert on SYS-2 (Remote Client)

Time (in min)

Batch Size

Loading TPC-H lineitem table
6M records
Commit for every 10,000 records

Bulk Load: 1.3 min
Effect of Batch Size on Updates

JDBC Batched Update

Update of TPC-H PARTSUPP table (800K records)
Commit for every 10,000 records
C.Index on update WHERE clause columns (partkey, suppkey)
Iterative and Set-Oriented Updates on the Server Side

- TPC-H PARTSUPP (800,000 records), Clustering index on (partkey, suppkey)
- Iterative update of all the records using T-SQL script (each update has an index lookup plan)
- Single commit at the end of all updates

Takes 1 minute

Same update processed as a `merge (update ... from ...)`

Takes 15 seconds
Why does Batching Perform Better?

- Reduced network round-trip delays
- Possibility of employing efficient algorithms
  - Repeated *selection* can be processed as a *join*
    - Can possibly use Hash/Merge join or INLJ with sorted RIDs
- Batched integrity checks and index updates
- Efficient disk access (sorted RID list, parallel I/O)
Batched Forms of Queries/Updates

- Batched forms of queries/updates are known

**Example:**

```
SELECT item-id FROM item WHERE category-id=?
```

**Batched form:**

```
SELECT pb.category-id, item-id
FROM param-batch pb LEFT OUTER JOIN item
    ON pb.category-id = item.category-id;
```
Ways to Exploit Set Orientation

- Programmer can explicitly use set-oriented form of the insert or update.

- Statement is automatically rewritten

```java
stmt = con.prepareStatement("insert into ... values (?, ?, ...)”);
while(...) {
    stmt.setInt(1, p1);
    stmt.setInt(2, p2);
    ...
    stmt.addBatch();
}
stmt.executeBatch();
```
Batched Forms of Procedures

- Batched forms of complex procedures are hard to derive automatically

- Manual rewriting: ad-hoc, time-consuming and error-prone

- Automate the generation of batched forms of complex procedures.
Challenges in Generating Batched Forms of Procedures

- Must deal with control-flow, looping and variable assignments
- Inter-statement dependencies may not permit batching of desired operations
- Presence of “non batch-safe” (order sensitive) operations along with queries to batch

Approach:

- Equivalence rules for program transformation
- Make use of the data dependence graph
Generating Batched Form of a Procedure

Step 1: Create trivial batched form.
Transform: \[
\text{procedure } p(r) \{ \text{body of } p \} \]
To
\[
\text{procedure } p\_\text{batched}(pb) \{ \\
\text{for each record } r \text{ in } pb \{ \\
\text{body of } p > \\
\text{collect the return value } > \\
\} \\
\text{return the collected results paired with corresp. params;}
\}
\]

Step 2: Move the query exec stmts out of the loop and replace them with a call to their batched form.
Rule 1A: Rewriting a Simple Set Iteration Loop

for each $t$ in $r$ loop

$q(t.c_1, t.c_2, \ldots t.c_m)$; $\iff q_b(\prod_{c_1,c_2,\ldots,c_m}^d (r))$; end loop;

where $q$ is any batch-safe operation with $q_b$ as its batched form

for each $t$ in $r$ loop

insert into orders values (t.order-key, t.order-date, \ldots);
end loop;

insert into orders select \ldots from $r$;
Rule 1B: Assignment of Scalar Result

for each r in t {
    r.count = select count(itemid) from item
              where category=r.catid;
}

merge into t using bq on t.catid=bq.catid
when matched then update set t.count=bq.item-count;

bg: select t.catid, count(itemid) as item-count
    from t left outer join item on item.category=t.catid;

* SQL Merge: Lets merging the result of a query into an existing relation

Formal specification of Rule-1B in thesis.
Rule 2: Splitting a Loop

while (p) {
    ss1;
    s_q;
    ss2;
}

Table(T) t;
while (p) {
    ss1 modified to save local variables as a tuple in t
} for each r in t {
    s_q modified to use attributes of r;
} for each r in t {
    ss2 modified to use attributes of r;
}

* Conditions Apply
Loop Splitting: An Example

while(top > 0) {
   catid = stack[--top];
    count = select count(itemid) from item where category=:catid;
    total += count;
}

Table(key, catid, count) t; int loopkey = 0;
while(top > 0) {
    Record r;
    catid = stack[--top];
    r.catid = catid; r.key = loopkey++; t.addRecord(r)
}
for each r in t order by key {  // order-by is removed if operation is batch-safe
    t.count = select count(itemid) from item where category=t.catid;
}
for each r in t order by key {
    total += t.count;
}
Rule 2: Pre-conditions

- Conditions are on the **data dependence graph** (DDG)
  - Nodes: program statements
  - Edges: data dependencies between statements

- Types of dependence edges
  - Flow (Write → Read), Anti (Read → Write) and Output (Write → Write)
  - Loop-carried flow/anti/output: Dependencies across loop iterations

- Pre-conditions for Rule-2 (Loop splitting)
  - No loop-carried flow dependencies cross the points at which the loop is split
  - No loop-carried dependencies through external data (e.g., DB)

Formal specification of Rule-2 in thesis.
Rule 4: Control Dependencies

```java
while (...) {
    item = ...; qty = ...; brcode = ...;
    if (brcode == 58)
        brcode = 1;
        insert into ... values (item, qty, brcode);
}
```

`Store the branching decision in a boolean variable`
Cascading of Rules

After applying Rule 2 (loop splitting)

Table(…) t;

while (…) {
    r.item = ...; r.qty = ...; r.brcode = ...;
    r.cv = (r.brcode == 58);
    r.cv = true? r.brcode = 1;
    t.addRecord(r);
}

for each r in t {
    r.cv = true? insert into ... values (r.item, r.qty, r.brcode);
}

insert into .. ( select item, qty, brcode from t where cv=true );

Rule 1C deals with batching conditional statements. Formal spec in the Thesis.
Batching Across Nested Loops

- Statement to batch can be present inside nested loops
- Goal: Pull the statement out of multiple enclosing loops
  - Batch the stmt w.r.t inner-most loop and then with the next higher level loop.
- Batches as nested tables: use nest and unnest
- Formal rules in thesis
Need for Reordering Statements

\[(s1)\] while (category != null) {
  \[(s2)\] item-count = \texttt{q1}(\texttt{category});
  \[(s3)\] sum = sum + item-count;
  \[(s4)\] category = getParent(category);
}\]
Reordering Statements to Enable Rule 2

```java
while (category != null) {
    int item-count = q1(category)  // Query to batch
    sum = sum + item-count;
    category = getParent(category);
}

Splitting made possible after reordering
while (category != null) {
    int temp = category;
    category = getParent(category);
    int item-count = q1(temp);
    sum = sum + item-count;
}
```
Statement Reordering

Algorithm **reorder**: Reorders statements in a loop to enable loop splitting. (Details in thesis.)

**Theorem:**

If a query execution statement does not lie on a true-dependence cycle in the DDG, then algorithm **reorder** always reorders the statements such that the query execution can be batched.

**True-dependence cycle:**

A directed cycle made up of only FD and LCFD edges.
Implementation and Evaluation

- Conceptually the techniques can be used with any language (PL/SQL, Java, C#-LINQ)
- We implemented for Java using the SOOT framework for program analysis

Evaluation

- No benchmarks for procedural SQL
- Scenarios from three real-world applications, which faced performance problems
- Data Sets: TPC-H and synthetic
Application 1: ESOP Management App

Process records from a file in custom format. Repeatedly called a stored procedure which:
- Validates inputs
- Looks up existing record
- Updates or Inserts

Rewritten program generated by our implementation used outer-join and merge.
Application 2: Category Traversal

Find the maximum size of any part in a given category and its sub-categories.

**Clustered Index**
CATEGORY (category-id)

**Secondary Index**
PART (category-id)

**Original Program**
Repeatedly executed a query that performed selection followed by grouping.

**Rewritten Program**
Group-By followed by Join
Application 3: Value Range Expansion

Expand records of the form:
(start-num, end-num, issued-to, …)
Performed repeated inserts.

Rewritten program
Pulled the insert stmt out of the loop and replaced it with batched insert.

~75% improvement

~10% overhead
Related Work

- Query unnesting - Kim [TODS82], Dayal [VLDB87], …
  - We extend the benefits of unnesting to procedural nested blocks

- Optimizing set iteration loops in database programming languages - Lieuwen and DeWitt [SIGMOD 92]
  - Perform rule-based program rewriting, but
  - Do not address batching of queries/procedure calls within the loop
  - Limited language constructs – e.g., no WHILE loops

- Rewriting CODASYL programs – Katz and Wong [TODS82], Demo and Kundu [SIGMOD 85]
  - Employ data-flow analysis, for identifying dependencies between FIND statements.

- Parallelizing compilers Kennedy[90], Padua[95]
  - We borrow and extend their techniques
  - Transformation for batching can afford extra CPU operations to save IO.
Summary and Conclusions

- Parameter batching is the key for set-oriented evaluation of repeatedly called procedures

- We presented a program transformation based approach, which can be used for:
  a. Generating batched forms of procedures and
  b. Replacing imperative loops with calls to batched forms

- The work combines query optimization with program analysis and transformation.

- Experiments on real-world applications show the applicability and benefits of the proposed approach.
Future Work

- Cost-based decision of calls to batch
- Pipelined execution of queries within a UDF
- Inferring batch-safety of an operation
- Recognizing/handling JDBC calls (for Java)
- Implementation for PL/SQL
- Exploring the use of program analysis beyond batching
  - E.g., pushing externally applied predicates to the database.
Thanks
Rewriting Simple Set Iteration Loops

**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.

\[
\mathcal{M}_{c_{w1}=c_{w1}', \ldots c_{wn}=c_{wn}'}(r, e)
\]

where \( e = \rho_x(c_{r1}, \ldots c_{rm}, c_{w1}', \ldots c_{wn}')q_b(\Pi_{c_{r1}, \ldots c_{rm}}(r))\);
Rule 1C: Batching Conditional Statements

for each $t$ by ref in $r$ loop

$(t.cv == true) ? t.c_{w1}, \ldots, t.c_{wn} = q(t.c_{r1}, \ldots, t.c_{r_m})$;

end loop;

where $q$ is a pure function.

Let $b = \Pi_{cr1,\ldots,crm}(σ_{cv=true} \ r)$ // Parameter batch

Let $e = ρ_{x(c_{r1},\ldots,c_{rm},c_{w1'},\ldots,c_{wn'})} q_{b(b)}$

$M_{c_{w1}=c_{w1'},\ldots,c_{wn}=c_{wn'}}(r, e)$ // Merge the results*

* SQL Merge: Lets merging the results of a query into an existing relation
Batching Across Nested Loops

- Batch the stmt \( w.r.t. \) the inner loop and then with the outer loop (increases the batch size)
- Batches as nested tables: use \textit{nest} and \textit{unnest}

\textit{Details in the Thesis.}

\textbf{Rule 6A}

\begin{align*}
\text{for each } t \text{ in } r \text{ loop} \\
qb((t.c_1, \ldots t.c_n) \times t.s); \\
\text{end loop;}
\end{align*}

\[
\uparrow \downarrow \\
qb(\Pi^d_A(\mu_s(r))) \quad \text{where } A = \{c_1, \ldots c_n\} \cup S
\]