Remote Batch Invocation for SQL Databases

William R. Cook
Department of Computer Science
University of Texas at Austin
wcook@cs.utexas.edu

Ben Wiedermann
Department of Computer Science
Harvey Mudd College
benw@cs.hmc.edu

ABSTRACT

Batch services are a new approach to distributed computation in which clients send *batches* of operations for execution on a server and receive hierarchical results sets in response. In this paper we show how batch services provide a simple and powerful interface to relational databases, with support for arbitrary nested queries and bulk updates. One important property of the system is that a single batch statement always generates a constant number of SQL queries, no matter how many nested loops are used.

1. INTRODUCTION

Batches are a new programming model for efficient access to distributed services [15, 16]. The key architectural difference from previous approaches is that batches require clients to send *collections of operations* to servers rather than individual messages/method invocations. The collection of operations sent by a client are represented as *scripts* written in a domain-specific language designed specifically to represent batches of related operations. With batches, conventional wisdom is inverted: fine-grained interfaces are encouraged, proxies are avoided, serialization is not needed. The key innovation that makes batches work is a new client-side invocation model, the **batch** statement.

A **batch** statement specifies a *remote root* and a block of statements that can intermingle operations on the remote root and ordinary local objects. The examples in this paper are written in *Jaba*, a version of Java extended with a **batch** statement. The following code fragment prints information about files located on a remote server:

```
1 batch (File root : new TCPClient<File>("74.1.9.14", 1025))
2  for (File f : root.listFiles())
3   if (f.length() > 500)
4    System.out.println(f.getName() + ": " + f.lastModified());
```

The batch statement defines `root` as a `File` handle on a remote server with a given IP address and port number. The body of the batch statement in lines 2–4 iterates over each remote file and prints its name and last-modified date if the file's length exceeds a given length. The block intermingles remote operations (e.g., getting a remote file's name) and local operations (e.g., printing the file's name). Although this code follows normal syntax and typing conventions, its execution model is radically different from ordinary sequential execution:

- Remote and local operations are reordered relative to one another, duplicating loops and conditionals where necessary, so that all remote operations are performed in one round trip to the server.
- The order of local operations is preserved, but the semantics of remote operations is defined by the server.

Step 1 only succeeds if there are no back and forth interdependencies between local and remote operations, otherwise a compiler error is issued. The compiler then translates the batch statement to send the remote operations to the server as a *batch script*. The batch script for the example given above produces a table with two columns, labeled `A` and `B`:

```
for (f : *.listFiles()) /* generated batch script code */
 if (f.length() > 500)
   A: f.getName()
   B: f.lastModified()
```

Details on batch scripts, updated to support database access, are presented in Section 3. The resulting tree is a hierarchy of records with tagged values, similar to JSON, XML, or hierarchical tables. The remaining client code produced by the compiler, once the remote operations are removed, is the following:

```
for (r in resultSet) /* generated code */
  System.out.println(r.get("A") + ": " + r.get("B"));
```

Note that it does not contain any remote operations. The batch execution model does not involve the use of proxies or object serialization. The batch script creates a dynamic Service Façade on the server, and the result tree is a dynamically created Data Transfer Object [10]. The end result is a natural fine-grained programming model that provides easy access to remote services.

Batches generalize the communication model used for access to database servers, where a small imperative scripting language replaces SQL, and results are hierarchical record sets, not flat tables. Thus it is not surprising that batches can be used to encode access to relational databases as a special case.

2. BATCHES FOR DATABASE ACCESS

Batches combine the two most important requirements for database access: a natural programming model and optimized execution. Batches are a natural programming model because they allow clients

[1]This material is based upon work supported by the National Science Foundation under Grants #0448128 and #0724979.
[2]The Jaba compiler allows the **for** keyword to be used in place of **batch**, so that Jaba programs are compatible with Java syntax.

```
1  @Entity(name="Products")
2  class Product {
3    @Id int ProductID;
4    String Name;
5    double UnitPrice;
6    long UnitsInStock;
7    @Column(name="CategoryID") Category Category;
8    @Inverse("Product")   Many<Order_Details> Orders;
9    ... }
10 @Entity(name="Categories")
11 class Category {
12   @Id int CategoryID;
13   String Name;
14   String Description;
15   @Inverse("Category") Many<Product> Products;
16   ... }
17 abstract class Northwind {
18   Many<Product> Products;
19   Many<Category> Categories;
20   ... }
```

Figure 1: Entity-Relationship Model (**public** omitted)

```
interface Many<T> extends Iterable<T> {
  int   count();
  int   count(Fun<T, Boolean> f);
  double sum(Fun<T, Double> f);
  double average(Fun<T, Double> f);
  double min(Fun<T, Double> f);
  double max(Fun<T, Double> f);
  boolean exists();
  boolean exists(Fun<T, Boolean> f);
  boolean all(Fun<T, Boolean> f);
  <P extends Comparable<P>>
     Many<T> orderBy(Fun<T, P> f, boolean asc);
     Many<T> distinct();
     Many<T> first(int n);
  <I> Many<I> project(Fun<T,I> f);
  <I> Many<Group<I, T>> groupBy(Fun<T, I> f);
     T id(Object id);
  void insert(@NamedArguments T item);
}
interface Fun<S, T> { T apply(S x); }
interface Group<S, T> { S Key; Many<T> Items; }
```

Figure 2: Interface for tables and many-valued relationships

to operate on database objects as if they were in-memory objects, without any need for explicit queries. Database access with batches is optimized by lifting multiple database operations out of the program as a batch script, which can be optimized for executed on a relational database engine. Achieving these goals requires an appropriate implementation of a database-specific batch service connection. We call this implementation Batch2SQL and illustrate how a developer uses Batch2SQL to model relational data and to describe database queries and modifications. Using batches to access SQL is a general programming concept that can be added to any programming language. The examples given in this paper have been written using Jaba, a full implementation of batches for Java with support for SQL, available at
`www.cs.utexas.edu/~wcook/projects/batches`.

2.1 Data Model

Batch2SQL uses the conventional approach to object-relational (OR) mapping, where a relational database is described by a package of Java classes with annotations to specify tables, attributes, and relationships. Batch2SQL mapping is a simplified version of the mappings used in the Java Persistence Architecture (JPA) [18]. Figure 1 gives the classes that describe a subset of the Northwind database [24]. Although it is conventional for tables to be named using singular nouns, the Northwind database uses plural names. The class model corrects this problem by using singular class names, with the table names specified in `Entity` annotations. The `Entity`, `Id`, and `Column` attributes are standard as defined in JPA. Column attributes are omitted if the column name is the same as the Java field name. This presentation uses fields rather than *getter/setter* methods, although Batch2SQL supports both conventions.

As is typical of OR mapping, object references represent foreign key relationships. A one-to-many relationship has two sides, which are declared as fields in the two classes involved in the relationship. The single-valued side is an object reference, with a column representing a foreign key. For example, the `Category` of a `Product` is represented by the `CategoryID` foreign key column in the database (line 7). The `Products` field in the `Category` class is defined as the *inverse* of the `Category` field (line 15). The `Products` field has type `Many<Product>` because it is a many-valued field. The `Many` interface is defined in Figure 2. The database is represented by a class with one many-valued field for each entity, as illustrated by the `Northwind` class in Figure 1.

2.2 Database Queries

A client queries the database by traversing the interface classes defined in the previous section. Behind the scenes, Batch2SQL converts the database traversals into queries. Some of the examples in this section are based on a collection of LINQ [5] queries that exhibit—among other capabilities—selection, filtering, projection, joining, and aggregation [23].

Each batch block binds its remote root to a Batch2SQL service connection, which plugs into the underlying batch architecture to connect batches to a SQL database. The connection is created as follows:

```
String connectionStr = "jdbc:mysql://localhost/Northwind";
SQLBatch<Northwind> connection =
  new SQLBatch<Northwind>(Northwind.class, connectionStr);
```

The class `SQLBatch<Northwind>` is the connection object that interprets batch scripts as SQL, specialized to operate over the Northwind database. The generic parameter is used for type checking, while the **`class`** provides reflective access to the class attributes.

Figure 3 compares the LINQ and Batch2SQL versions of a simple query that selects data, traverses relationships, filters results, and aggregates a collection by counting. The LINQ version in Figure 3a accesses a (virtual) collection of all products in the database (line 1). A query on this collection (line 2) filters the items (line 4) and then creates an anonymous record of results, containing the product name, category name, count of orders, and unit price (lines 5–8). The second half of the example iterates over the results of the query to print out results (line 10–13). The programmer is required to create the anonymous record structure to convey the results from the query to the code that uses the results. The artificial names `ProductName` and `CategoryName` were created to avoid the name clash that would arise if two fields named `Name` are included in the result. Such name clashes reduce the modularity of the code, because operations on product and categories cannot be applied to the anonymous record.

By contrast, the Batch2SQL version in Figure 3b contains a batch block that intermingles data retrieval (e.g., of the products in line 2) with data use (e.g., printing the results in lines 4–7). The batch block's body appears to iterate over the products and print a result for each iteration. However, the Batch Java compiler enforces an alternate semantics: It disentangles the query from the local code

```
List<Product> products = GetProductList();
var infos =
  from p in products
  where p.UnitsInStock == 0
  select new { ProductName = p.Name,
               CategoryName = p.Category.Name,
               OrderCount = p.Orders.Count(),
               p.UnitPrice };
foreach (var info in infos)
  print("{0} in category {1} sold {2} times {3}/unit",
      info.ProductName, info.CategoryName,
      info.OrderCount,
      info.UnitPrice);
```

(a) LINQ

```
batch (Northwind db : connection)
 for (Product product : db.Products)
  if (product.UnitsInStock == 0)
   print("{0} in category {1} sold {2} times {3}/unit",
      product.Name, product.Category.Name,
      product.Orders.count(),
      product.UnitPrice);
```

```
SELECT T1.Name AS g1, T2.Name AS g2,
  (SELECT COUNT(*) FROM Order_Detail T3
   WHERE (T3.ProdutID=T1.ProdutID)) AS g4,
  T1.UnitPrice AS g3
FROM Products T1 INNER JOIN Categories T2
  ON T2.CategoryID=T1.CategoryID
WHERE T1.UnitsInStock=0
```

(b) Batch2SQL, with generated SQL

Figure 3: Simple selection, joining, filtering, and aggregation.

and arranges for the program to consume the query results all at once, at runtime, by transforming the program to execute a SQL query. The SQL generated by Batch2SQL is given below the batch code in Figure 3b. For convenience, both the LINQ and Java versions are written using a generic `print` function modeled on the standard C# `Console.WriteLine` function.

Figure 4 illustrates the common *master/detail* pattern. In the LINQ version (Figure 4a), the main query selects customers in Washington state, then constructs an anonymous record containing the company name and a nested collection, called `OrderInfo`, with information on the company's orders. The results of this query are processed in lines 10–14. Two nested iterations, which are analogous to the nested collections in the query, print out the results. In LINQ, many small changes to a program require coordinated changes to two different places in the code. For example, to print additional data, it must be included both in the query and in the code that consumes the query results.

In contrast, the Batch2SQL version in Figure 4b iterates directly over the customers and orders, printing out results as needed. Any field that is printed or otherwise used in the body of the loop is *automatically* added to the generated query. The necessary intermediate data structure to transmit the results is also created automatically, while it was defined manually in the LINQ version. Batch2SQL uses two SQL select statements to load the data for the two loops. The first loads all customers in Washington. The second loads all orders in the date range for customers in Washington. The Batch2SQL runtime automatically reorganizes the results of these two queries into a hierarchical record set that groups orders under each customer. This example illustrates a critical property of Batch2SQL: a batch always generates exactly one select

```
1  List<Customer> customers = GetCustomerList();
2  DateTime cutoffDate = new DateTime(1997, 1, 1);
3  var custInfo =
4    from c in customers
5    where c.Region == "WA"
6    select new { c.CompanyName,
7                 OrderInfo = from o in c.Orders
8                        where o.OrderDate >= cutoffDate
9                        select { o.OrderDate }}
10 foreach (var c in custInfo) {
11   print("Customer {0}:", c.CompanyName)
12   foreach (var o in c.OrderInfo)
13     print(" {0}", o.OrderDate);
14 }
```

(a) LINQ

```
Date cutoffDate = Date.valueOf("1997-01-01");
batch (Northwind db : connection)
 for (Customer c : db.Customers)
  if (c.Region == "WA") {
   print("Customer {0}:", c.CompanyName);
   for (Order o : c.Orders)
    if (o.OrderDate.after(cutoffDate))
     print(" {0}", o.OrderDate);
  }
```

```
SELECT T1.CompanyName AS g43, T1.CustomerID AS id
FROM Customers T1
WHERE T1.Region="WA"
ORDER BY id ASC

SELECT T3.CustomerID AS parent, T3.OrderDate AS g45
FROM Orders T3 INNER JOIN Customers T4
  ON (T4.CustomerID=T3.CustomerID)
WHERE T3.OrderDate>? AND T4.Region="WA"
ORDER BY parent ASC
```

(b) Batch2SQL, with generated SQL

Figure 4: Master-detail queries

query for each static (syntactic) **for** loop in the batch, no matter how many nested iterations are executed at runtime. This property is not shared by LINQ, although it is guaranteed by Ferry [11].

Figure 5 illustrates the use of *dynamic* queries in LINQ and Batch2SQL. A query is dynamic if the structure of the query (for example, the conditions in the where clause) vary at runtime, rather than being static. The example corresponds to the common case of generic search criteria on a web page. If the price/name is specified, then a filter is defined, otherwise the entire test is omitted. In LINQ, dynamic queries are created by incremental construction of a query object, as shown in Figure 5a. In Batch2SQL, the system identifies that the tests on `Price` and `Name` do not depend upon the database, so these conditions are evaluated at query construction time, and short circuit evaluation of the `||` operator either omits or includes the condition on the right side of the `||` expression.

2.3 Database Modification

Programmers can also use Batch2SQL to express database modifications. Figure 6 shows batch blocks for insertion, bulk update, and deletion, as well as their generated SQL statements. We omit the corresponding LINQ versions, since those versions are similar to Batch2SQL.

Line 3 in Figure 6a creates a new `Product`. The programmer defines the record's attributes by assigning values to fields of the

```
1 batch (Northwind db : connection) {
2  Product p = new Product();
3  p.Name = "New Widget";
4  p.UnitPrice = 23.23;
5  print(db.Products.insert(p));
6 }
```

```
INSERT INTO Products(UnitPrice, Name)
SELECT 23.23 AS UnitPrice,
      "New Widget" AS Name
```

(a) insert

```
batch (Northwind db : connection)
 for (Product p : db.Products)
  if (p.Category.Name == "Produce")
    p.UnitPrice *= .9;
```

```
UPDATE Products T1
INNER JOIN Categories T2
 ON (T2.CategoryID=T1.CategoryID)
SET T1.UnitPrice=T1.UnitPrice * 0.9
WHERE (T2.Name="Produce")
```

(b) bulk update

```
void deleteProduct(String name) {
 batch (Northwind db : connection)
  for (Product p : db.Products)
   if (p.ProductName == name)
     p.delete();
}
```

```
DELETE T1
FROM Products T1
WHERE (T1.Name=?)
```

(c) delete

Figure 6: Database modifications

```
void FindProducts(float price, String name) {
 List<Product> products = GetProductList();
 if (Price != 0)
   products = products.Where(p => p.UnitPrice > Price);
 if (Name.Length > 0)
   products = products.Where(p => p.Name.contains(Name));
 products = products.Select(p => p.Name);
 foreach (String name : products.ToList())
   print(name);
}
```

(a) LINQ

```
void findProducts(float price, String name) {
 batch (Northwind db : connection)
  for (Product product : db.Products)
   if ((price == 0 || product.UnitPrice > price)
     && (name.length() == 0 || product.Name.contains(name)))
     print(product.Name);
}
```

(b) Batch2SQL

Figure 5: Dynamic queries

e	$=$	c	literal constant
	$\mid$	$\star(\overline{e})$	primitive operation
	$\mid$	x	variable, where * is the service root
	$\mid$	$x := e$	variable assignment
	$\mid$	$e.f$	field access
	$\mid$	$e.f := e$	field assignment
	$\mid$	$e.m(\overline{e})$	method call
	$\mid$	$e.m(\overline{x = e})$	named argument method call
	$\mid$	**if** e **then** e **else** e	conditional
	$\mid$	**let** $x = e$ **in** e	local variable definition
	$\mid$	**for** $\circ\ (x : e)\ e$	for/comprehension aggregate by $\circ$
	$\mid$	$\lambda\overline{x}.e$	first-class function
	$\mid$	$l : e$	output result named l
$\circ$	$=$	$; \mid + \mid \times \mid \wedge \mid \vee \mid$ average $\mid$ min $\mid$ max	
$\star$	$=$	$\circ \mid - \mid \div \mid \neg \mid = \mid \neq \mid < \mid \leq \mid > \mid \geq$	
		$\mid$ substring $\mid$ contains $\mid$ startswith $\mid$ endswith	

x, f, m are variable, field, and method names, respectively

Figure 7: Batch script syntax.

newly created record (lines 3–4). A call to the remote database's `Products.insert` method in line 5 completes the record creation. The primary key of the newly inserted row is returned. This works equally well if the insert is performed inside a loop.

Batch2SQL programs can perform bulk database updates. Figure 6b applies a bulk discount to all the products in a particular category, for which Batch2SQL generates a single SQL statement.

Figure 6c illustrates deleting a record from a table. The `delete` method call is translated to a SQL **DELETE** statement. This example also demonstrate's Jaba's ability to pass query parameters. The name of the product to delete is a SQL query parameter, represented by `?`. The actual value is passed as an argument to the query. Systematic use of query parameters completely prevents the possibility of SQL query injection attacks. Note that a single batch may contain multiple operations, including inserts, deletes, queries, and updates.

3. BATCH SCRIPT TO SQL TRANSLATION

The abstract syntax of the batch script language is given in Figure 7. The language supports primitive data (strings, integers, floating point numbers, booleans, dates, durations) and basic operations on these types, field access and update, method calls, mutable local variables, iteration/comprehension, functions, and result output. The sequence operator (;) is used to create sequences (blocks) of expressions. The **for** expression includes a binary operator that is used to aggregate results, and the sequence operator (;) creates a list of the results. Binary operators (other than min, max, and average) return an appropriate *unit* for iteration over an empty collection.

We do not give a semantics for the language, because the semantics is defined by the server that implements a batch service. What this means is that the semantics of batch scripts must be specified by a service as part of its interface contract. Different services can choose to interpret batch scripts in different ways. While this design decision may seem unconventional, including deliberate ambiguity is a common practice in language design (e.g. order of argument evaluation is deliberately undefined in C). However, batch scripts do take this practice to an extreme. Most services will define the semantics of batch scripts according to the standard operational semantics of imperative languages.

Not every batch script is legal. In particular, a standard type system (omitted) identifies type-safe batch scripts, relative to a particular service interface. For clients written in statically typed languages, the type system of the client language ensures that all batch scripts created by a batch block are well typed. Servers may place additional restrictions in batch scripts. For example, they may not support imperative update of local batch variables. Most clients will prohibit recursive calls in local batch functions (λ-expressions). Given these two restrictions, all local batch variables (**let** expressions) can be eliminated from batch scripts by variable substitution.

For database services, the semantics of batch scripts is given by translation to SQL. Defining the full translation is beyond the scope of this paper, but we will provide an overview of the key steps. Details can be found in the implementation.

1. Remove **let** expressions, as mentioned above.

2. Convert aggregate methods with λ arguments to corresponding **for** expressions. For example,

   ```
   db.Products.sum(fun(p) p.inventory)
   ```

   becomes

   ```
   for + (p : db.Products) p.inventory
   ```

3. Partition the script into query, delete, update, insert operations. Field assignments are interpreted as SQL UPDATE expressions.

4. Convert field traversals to joins. This requires keeping track of a tree of joins. Consider an expression of the form $e.r$ where e is an expression representing rows of table t and r is a single-valued relationship to a table t'. The expression $e.r$ is converted to t' and the join from t to t' on r is added to the join tree.

5. Nested **for** loops that produce result sets (rather than aggregations) are lifted to be independent of the containing **for** loop. If the outer loop is over data items $a_1, ..., a_n$ and each iteration a_i has a corresponding list of items $b_{i1}, ..., b_{im_i}$ in the nested loop, then after lifting the inner loop generates all the nested iterations at once, in the form $b_{11}, ..., b_{1m_1}, b_{21}, ..., b_{2m_2}, ..., b_{n1}, ..., b_{nm_n}$. A `parent` field representing a_i is created in the nested/lifted loop to connect each row of the nested loop with a specific iteration in the outer loop. Proper association of outer and nested loops requires that the filters and sorting of the outer loop are added to the nested loop during lifting.

   Consider a nested iteration of the form **for** `(y :` $\overline{x.r_y}.\overline{m_y}$`) e` where y is an enclosing **for** variable, $\overline{r}$ is a sequence of field traversals resulting in an object of type `T`, and $\overline{m(e)}$ are modifiers (defined in step 6 below). This query is rewritten as an iteration over `T`, where the parent object x is defined as the *inverse* of $\overline{r}$, and the conditions and modifiers of x are copied to the new query. This copying explains the inclusion of the test `c.Region == "WA"` in the subquery in Figure 4b.

6. Convert **for** expressions to SQL SELECT expressions. Output expressions in the body of the **for** become output columns in the resulting SELECT. If the body is an **if** expression with only one branch, it is converted to a WHERE clause. Other **if** expressions are ignored. If the collection contains `orderBy`, `distinct`, `first`, or `groupBy` operations, these are added to the SELECT query.

The number of queries executed is equivalent to the number of **for** expressions in the batch script. After the queries are executed, the SQL result sets are merged into a tree of results, using the `parent` fields to associate nested rows to their containing iteration.

For batches that contains multiple insert, delete, update operations, possibly with a query as well, the operations are executed in the following order: select, delete, update, insert. Care must be taken in this case to ensure that the operations do not interfere with each other in unintended ways. If a more strict semantics is desired, an error can be signaled in situations where there may be conflicts.

4. RELATED WORK

Researchers have contributed many approaches that integrate databases and general-purpose programming languages. These techniques differ in how they (a) partition code into database and client-side computation, (b) translate from a general-purpose query to a database query, and (c) define the semantics of the interaction between database and client-side computations.

The use of first-class queries in a programming language is one common approach to database integration. Some languages use comprehension syntax [29, 6] (e.g., Staple [22], Haskell/DB [19], Kleisli [31], Links [9], and Hop [27]) while others use SQL-like syntax (e.g., SQLJ [1] and LINQ [3, 5]). As in Batch2SQL, some expressions in the host programming language can be used as parts of high-level queries, which are translated to a database query language like SQL. However, the key difference is that Batch2SQL allows local and remote operations to be intermixed, while automatically partitioning them to create queries and manage the structure of data that result from the query. Links [9] uses a type-and-effect system to determine whether queries can be performed in the database and rejects programs that incorrectly mix query and non-query operations, while Batch2SQL uses a simple dataflow analysis to lift database operations from the program and rejects programs that require more than one round trip to the database server. In addition, Links supports full higher-order functional programming in the client, while Batch2SQL supports first-order procedures that can be inlined into a batch. The question of whether it is better to require programmers to write queries explicitly, or allow implicit query operations mixed with other imperative operations as in BatchDB, is still open.

Translating high-level queries to SQL is an important topic of research in its own right. Cooper [8] proves that high-level queries without nested many-valued fields can be encoded into a single SQL statement. Batch2SQL improves this result by encoding arbitrary nested queries in a finite number of SQL statements, but we do not provide a formal proof of this property. Ferry [11] also compiles high-level queries, with nested multi-valued sub-queries, into a constant number of SQL queries. The main difference between Ferry and Batch2SQL translation schemes is that Ferry uses a powerful algebraic query transformation approach, while Batch2SQL uses local syntactic query transformation. In practice Ferry can produce better queries, but Batch2SQL generates SQL that is more closely related to the source query. The Ferry translation engine could be plugged in as another handler for batch scripts created by the Jaba **batch** statement.

First-class queries place the burden of program partitioning on the programmer. In contrast, some systems provide *transparent* access to persistent data [25, 2]. In transparent systems, queries are implicit, data queries are intermingled with data use, and the programmer need not partition code into queries and clients. Many such systems, including modern object-relational mappers, achieve transparency via *object faulting*—a runtime mechanism that loads values from the database on demand [13]. This *record-at-a-time* retrieval behavior limits performance because it inhibits query optimization [21]. Although transparency promotes good software engineering by seamlessly blending the semantics of in-memory and persistent operations, it thwarts efficient database access. For this reason, modern object-relational mappers [12, 18] offer either object faulting or an explicit query language, which allows the programmer to trade transparency for efficiency.

Some systems, including Batch2SQL, seek to provide both transparency and efficiency. In our previous work, we developed a program analysis that inferred and generated SQL queries from Java programs [30]. This technique—called *query extraction*—suffered from a few drawbacks. Specifically, the compiler was required to infer the scope of a query, and it could not handle data aggregation or modification. Based on insights from this work, we developed the general *batch* statement, which explicitly delimits the scope of intermingled local and remote computation [15, 16]. Batch2SQL comes full circle on this line of research to demonstrate that batches can provide a more comprehensive integration of programming languages and databases.

MIDAS transforms a Cobol program that uses the network model to access database values into one that uses declarative, relational queries [7]. Like Batch2SQL, MIDAS abstracts a program's implicit data traversals then transforms the program to execute explicit queries. MIDAS detects implicit aggregations; whereas Batch2SQL requires aggregations to be explicit. MIDAS is a completely static approach; whereas Batch2SQL combines static and dynamic techniques to leverage the genericity of batches.

Queryll uses bytecode rewriting to translate Java code to SQL [17]. Queryll's design goals differ from ours in that we aim to provide maximal transparency. Queryll, on the other hand, trades some transparency for ease of query translation.

Thor is an object-oriented database system whose implementation includes batching optimizations to reduce the system's latency overhead [20]. Thor's *batched future* is a runtime mechanism that delays a program's retrieval of remote objects until the program requires their values [4]. *Batched control structures* extend this descriptive capability to include loops and conditions, but not method invocations [33]. *Basic value promises* are analogous to Batch2SQL's primitive types, and they permit Thor to delay their retrieval until the local program needs their values [33]. Thor's runtime optimizations do not partition intermingled code. Yeung and Kelly describe a runtime mechanism similar to Thor's, but which permits more intermingling of client and server computations via Remote Method Invocation (RMI) [32]. Their mechanism preserves the global order of operations, so there is less scope for combining multiple remote calls than in Batch2SQL, which uses information about dependences to reorder local and remote computation while preserving the overall program semantics. Other systems have also been developed that distribute non-database, client-server programs while preserving global semantics [28, 14, 26].

5. CONCLUSION

Batch2SQL is a new approach to database integration in which databases are viewed as instances of a general-purpose batch service model. The database structure is reified as a service interface that specifies the properties of the data and appropriate operations (insert, delete, update). Clients access the service using a new control structure, a **`batch`** block, that intermixes local remote computations. The batch block is partitioned at compile time into a batch script describing the remote operations, and residual client code that uses the results from the server. The batch script is translated into SQL, with the guarantee that a constant number of SQL statements is executed no matter how many nested iterations used in the block. The result is an effective approach to database integration.

Acknowledgments We thank our collaborators, Eli Tilevich and Ali Ibrahim, for their help in making Batch2SQL possible.

6. REFERENCES

[1] ANSI/INCITS. Database Languages - SQLJ - Part 1: SQL Routines using the Java Programming Language. Technical Report 331.1-1999, ANSI/INCITS, 1999.

[2] M. P. Atkinson and R. Morrison. Orthogonally Persistent Object Systems. *The VLDB Journal*, 4(3), 1995.

[3] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The Essence of Data Access in Cω. In *The European Conference on Object-Oriented Programming*, 2005.

[4] Phillip Bogle and Barbara Liskov. Reducing cross domain call overhead using batched futures. *ACM SIGPLAN Notices*, 29(10), 1994.

[5] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. `msdn.microsoft.com/en-us/library/bb308959`, 2007.

[6] Peter Buneman, Leonid Libkin, Dan Suciu, Val Tannen, and Limsoon Wong. Comprehension syntax. *SIGMOD Record*, 23(1), 1994.

[7] Yossi Cohen and Yishai A. Feldman. Automatic high-quality reengineering of database programs by abstraction, transformation and reimplementation. *ACM Transactions on Software Engineering and Methodology*, 12(3), 2003.

[8] Ezra Cooper. The script-writer's dream: How to write great SQL in your own language, and be sure it will succeed. In *The ACM SIGMOD Workshop on Database Programming Languages*, pages 36–51, 2009.

[9] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: Web Programming Without Tiers. In *The International Symposium on Formal Methods for Components and Objects*, 2006.

[10] Martin Fowler. *Patterns of Enterprise Application Architecture*. Adison Wesley, 2003.

[11] Torsten Grust, Manuel Mayr, Jan Rittinger, and Tom Schreiber. Ferry: database-supported program execution. In *International Conference on Management of Data*, pages 1063–1066, 2009.

[12] Hibernate reference documentation. `www.hibernate.org`.

[13] Antony L. Hosking and J. Eliot B. Moss. Towards Compile-Time Optimisations for Persistence. In *The International Workshop on Persistent Object Systems*, September 1990.

[14] Galen C. Hunt and Michael L. Scott. The Coign automatic distributed partitioning system. In *The USENIX Symposium on Operating Systems Design and Implementation*, 1999.

[15] Ali Ibrahim, Yang Jiao, William R. Cook, and Eli Tilevich. Remote batch invocation for compositional object services. In *The European Conference on Object-Oriented Programming*, 2009.

[16] Ali Ibrahim, Yang Jiao, Marc Fisher II, William R. Cook, and Eli Tilevich. Remote batch invocation for web services: Document-oriented web services with object-oriented interfaces. In *The European Conference on Web Services*, 2009.

[17] Ming-Yee Iu and Willy Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In *The ACM/IFIP/USENIX Middleware Conference*, 2006.

[18] Eric Jendrock, Jennifer Ball, Debbie Carson, Ian Evans, Scott Fordin, and Kim Haase. The Java EE 5 Tutorial. Sun Microsystems, 2007.

[19] Daan Leijen and Erik Meijer. Domain specific embedded compilers. In *The USENIX Conference on Domain Specific Languages*, 1999.

[20] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *The ACM SIGMOD International Conference on Management of Data*, 1996.

[21] David Maier. Representing database programs as objects. In *The ACM SIGMOD Workshop on Database Programming Languages*, 1987.

[22] David J. McNally. *Models for Persistence in Lazy Functional Programming Systems*. PhD thesis, University of St. Andrews, October 1986.

[23] Microsoft. 101 LINQ samples. `msdn.microsoft.com/en-us/vcsharp/aa336746`.

[24] Microsoft. Northwind sample database. `msdn.microsoft.com/en-us/library/aa276825`.

[25] J. Eliot B. Moss and T. Hosking. Approaches to Adding Persistence to Java. In *The International Workshop on Persistence and Java*, 1996.

[26] Matthias Neubauer and Peter Thiemann. Placement inference for a client-server calculus. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 75–86, 2008.

[27] Manuel Serrano, Erick Gallesio, and Florian Loitsch. Hop: a language for programming the web 2.0. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2006.

[28] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 19(1), 2009.

[29] Phil Trinder. Comprehensions, a query notation for DBPLs. In *The ACM SIGMOD Workshop on Database Programming Languages*, 1992.

[30] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, 2008.

[31] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1), 2000.

[32] Kwok Cheung Yeung and Paul H. J. Kelly. Optimising Java RMI Programs by Communication Restructuring. In *The ACM/IFIP/USENIX Middleware Conference*, 2003.

[33] Q. Y. Zondervan. Increasing Cross-Domain Call Batching Using Promises and Batched Control Structures. Technical Report LCS-TR-658, MIT, 1995.