Integrating Programming Languages & Databases: What's the Problem?*

William R. Cook, Ali H. Ibrahim

Department of Computer Sciences, University of Texas at Austin
{wcook, aibrahim}@cs.utexas.edu

Abstract. The problem of integrating databases and programming languages has been open for nearly 45 years. During this time much progress has been made, in exploring specialized database programming languages, orthogonal persistence, object-oriented databases, transaction models, data access libraries, embedded queries, and object-relational mapping. While new solutions are proposed every year, none has yet proven fully satisfactory. One explanation for this situation is that the problem itself is not sufficiently well defined, so that partial solutions continue to be proposed and evaluated based upon incomplete metrics, making directed progress difficult. This paper is an attempt to clarify the *problem*, rather than propose a new solution. We review issues that arise on the boundary between programming languages and databases, including typing, optimization, and reuse. We develop specific criteria for evaluating solutions and apply these to the solution approaches mentioned above. The analysis shows that progress has been made, yet the key problem of meeting all the criteria simultaneously remains open.

Updated 10/12/2005.

So the solution's easy enough; each of us stays put in his or her corner and takes no notice of the others. You here, you here, and I there. Like soldiers at our posts. Also, we mustn't speak. Not one word. That won't be difficult; each of us has plenty of material for self-communings.
– Huis Clos (No Exit) by Jean Paul Sartre

1 Introduction

Programs that use databases are a critical part of our information infrastructure. These systems generally use programming languages for general-purpose computation and databases to control concurrent access to data, search large amounts of data, and/or update data reliably and securely. Such systems are increasingly being developed using procedural object-oriented languages and relational databases. For scalability and reliability, multiple application servers typically communicate with a shared, replicated database server.

* This material is based upon work supported by the National Science Foundation under Grant No. 0448128.

Procedural languages and database query languages are based on different semantic foundations and optimization strategies. These differences are known informally as "impedance mismatch" [32]: imperative programs versus declarative queries, compiler optimization versus query optimization, algorithms and data structures versus relations and indexes, threads versus transactions, null pointers versus nulls for missing data, and different approaches to modularity and information hiding. Because databases and programming languages can perform many of the same tasks, developers must make difficult architectural decisions about how to organize and partition system functionality. Distributed execution also requires efficient structuring and management of specialized communication patterns. As a result, applications that access databases are awkward to design and develop. Programming languages do not facilitate effective use of databases, and attaining good performance usually requires careful optimization based on expert knowledge, which can make programs difficult to maintain and evolve.

The primary contribution of this paper is a better understanding of impedance mismatch, or the *problem* of integrating databases and programming languages. We examine issues that affect the boundary between programming languages and databases to create a list of criteria for evaluating solutions. The criteria fall into three main categories: *typing*, *optimization* and *reuse*. In selecting the criteria, we rely on our experience in developing commercial data-oriented applications and applying the theory of programming languages and databases. The selection process is inherently subjective, but we measure the criteria by their ability to make useful distinctions between different solution approaches.

We apply our criteria to a range of specific solutions to impedance mismatch, including object-oriented databases, object-relational mappers, data access APIs, orthogonally persistent programming languages, and embedded query languages. We consider approaches that involve modifications to either the programming language or database side of the interface. Our criteria, however, measures both programming language and database issues, so a solution that provides a clean programming model but no database-style optimizations will not be counted as a successful solution to impedance mismatch.

In summarizing our findings, we identify areas where significant progress has been made, but also point out specific areas where more work is required. The proposed criteria provide a useful basis for understanding the decisions made by architects in selecting solutions for integrating programming languages and databases, and a guide for future research. We believe that the key to the complexity of impedance mismatch is the difficulty of meeting all the criteria simultaneously.

2 Related Work

This section reviews papers that have focused on clarifying the problems involved in integrating languages and databases. Specific integration solutions are discussed in the body of the paper.

In 1987, Atkinson & Buneman [6] reviewed early work on integrating programming languages and databases; their focus on creating a clean, uniform programming model for persistent data provided a framework for later research. David Maier [32] stated a key requirement for solving impedance mismatch: "Whatever the database programming model, it must allow complex, data-intensive operations to be picked out of programs for execution by the storage manager, rather than forcing a record-at-a-time interface." Bloom and Zdonik [12] identified cultural and technical differences, including the handling of consistency, triggers, optimization, and data scaling. The object-oriented database system manifesto [4] did not include any requirements on how OODBs would interface with programming languages, or list performance as a top-level requirement. Ten years later, Carey & DeWitt predicted the demise of persistent programming languages and object-oriented databases, and the ultimate success of object-relational databases [14]. They also identified the integration of databases and programming languages, which they called *client integration*, as one of five key research challenges. Atkinson reviewed the difficulty of experimental validation of a new approach to persistence [5].

Jordan [29] compares persistence frameworks for the Java platform. Jordan defines an implementation of the OO7 benchmark as a java program manipulating in-memory java objects [15]. This benchmark is then used as the standard for qualitative and quantitative comparison. Unfortunately, the OO7 benchmark only models a single-user, so the critical issue of concurrency control is not addressed. Jordan also assumes that all data can fit in memory. Finally, the OO7 benchmark is not representative of the most common operations in typical transactional/enterprise applications because 007 is focused on extensive traversals of hierarchical structures. OO7 was created to test the kind of specialized applications for which object-oriented databases were designed. Jordan provides performance numbers but does not summarize his qualitative analysis. In this survey we do not provide performance numbers. Instead we assume that programming languages should enable access to database optimizations, and provide a qualitative analysis of how effective they are at providing this access.

3 Typing

Difficulty in aligning types between programming languages and databases is traditionally viewed as a key cause of impedance mismatch.

Both programming languages and databases have support for primitive types and data structures. While the details of mapping between different representations of data can cause annoying problems, at a conceptual level the model of data in a database and in a programming language are compatible. This is not surprising, given the universality of techniques for structuring data. While the data and types are compatible, there are still significant issues in the static typing of queries and composite programs.

```
class Employee {                class Department {
  String     name;                 String        name;
  float      salary;               Set<Employee> employees;
  Department department;           Employee      manager;
}                                  }
```

Fig. 1. Example database schema defined via classes

3.1 Data Mapping (T1)

Primitive types in a programming language typically do not correspond to the types in a database, and usually primitive types differ between databases. For example, SQL-92 does not define the absolute precision of many numeric types. Operations may also be inconsistent; a common example is international string comparisons.

Techniques for mapping classes to relational databases has been subject to extensive research and development [1]. In summary, the most common approach is to define mappings between an entity/relationship (ER) model and an object oriented class model. An ER model provides a logical view of the structure of a relational database. In a ER model, attributes represent primitive data values like strings and integers. These are mapped to object instance members in the object-oriented model. Relationships in the ER model are mapped to references between objects. A multi-valued relationship is a collection of references. Subtyping in an object-oriented model can also be represented within an ER model. In some cases there are several ways in which the mapping can be performed, and the resulting design decisions are typically based on performance or other issues.

For example, a simple model of employees and departments is defined in Figure 1 as a pair of Java classes. In database terms, the `department` and `employees` fields represent a one-to-many relationship between `Department`s and `Employee`s.

Persistence for Methods When considering a mapping between objects and databases, some researchers have proposed that the *methods* of an object should be stored persistently in addition to the object state [7]. Research has even focused on allowing threads, user interface controls, or network connections to be persistent [30]. Given that the integration of state and behavior is one of the key concepts in object-oriented programming, it can be argued that a persistence mapping that does not store behavior/methods violates the basic principles of object-oriented programming. On the other hand, separation of data and behavior has proven quite useful in the design and evolution of data-intensive applications. This question is unresolved, and in this survey we do not propose any criteria for evaluating the utility of persistence for methods.

3.2 Interpretation of Null Values (T2)

Nulls in SQL behave differently from nulls in most procedural object oriented languages. In SQL, null represents "unknown", thus primitive operations such as addition or conjunction will return null if either operand is null. For example `x == null` always returns null even if `x` is null. On the other hand, aggregate SQL functions such as `sum` ignore nulls. Object-oriented programming languages typically allow object references to be null, but primitive types like integer cannot be null. Relational joins also treat null as "unknown" – but dereferencing a null pointer in an object-oriented language typically throws an exception.

Some languages, like C++ and C#, allow definitions of user-defined data types that match database semantics but can be used in place of the built-in programming language types. Not all programming languages are able to seamlessly integrate foreign types in this way.

4 Static Typing (T*)

Static typing is a common tool used to increase reliability and performance in both programming languages and databases. Programming languages use static typing to check programs before they are run – to ensure that only valid operations are applied to data at runtime. Static typing can improve performance because these checks can be omitted at runtime. It also aids in modular development, since clients and servers can be written and checked against well-defined interfaces. In a database, a query is typically checked for type errors before the query is compiled.

Static typing is a different kind of criteria from mapping of data and interpretation of null values. This is because static typing is not a property of data, but is instead a property of the system that manages data and the way it interprets programs or queries. Thus static typing is a meta-issue that applies to other criteria. For example, data mapping may be performed at runtime or it may be statically checked. In our evaluation, static typing is an additional *dimension* of evaluation for other criteria, rather than being a single criteria itself.

5 Interface Styles

The solution space for integrating programming languages and databases can be characterized by two extremes: *orthogonal persistence* and *explicit query execution*. The specific solutions examined in Section 9 all use some combination of these two approaches. Orthogonal persistence is a pure approach to persistence in which the mechanisms of persistence, or even the existence of an underlying database, is largely hidden from programmers. Explicit query execution is a pragmatic approach that allows existing languages to explicitly invoke database operations.

```
void printInfo(String prefix) {
  for (Employee e in db.allEmployees() )
    if ( e.name.startsWith(prefix) && e.salary > e.manager.salary ) {
      print( e.name );
      print( e.salary );
      print( e.department.name );
} }
```

Fig. 2. Printing employee information

5.1 Orthogonal Persistence (S1)

Orthogonal persistence is a natural extension of the traditional concept of variable *lifetime* to allow objects or values to persist beyond a single program execution [6]. In the most pure form, persistent values exist as long as they are referenced (transitively) by a persistent root, although explicit operations such as deletion have been explored. Persistence is *orthogonal* because the persistence behavior of a value is independent of any other programming considerations, including the type of the value or where it was created.

Programs that manipulate persistent data look the same as ordinary programs. Assuming that `db` is a persistent root that contains a collection of employees, Figure 2 finds employees whose last name begins with a prefix and whose salary is greater than their manager's salary. It then prints the employee's name, salary, and department name.

Examples of orthogonal persistence systems include PJama [7], Thor [31], and OPJ [33]. Pure orthogonal persistence systems often implement their own storage manager, rather than relying upon existing database technology.

Rather than view orthogonality as a binary property, it is more useful to view it as a spectrum. In this view, it is a measure of the degree of uniformity in the treatment of persistent and non-persistent data. This view is also reasonable given that some operations, like those related to transactions, are only meaningful for persistent data, so some degree of non-orthogonality is essential [11].

Most Object-oriented databases (OODB) implement a degree of orthogonal persistence, although the values that can be persistent are often restricted to be objects. [16]. They are rarely purely orthogonal, since special operations for querying are provided on persistent data. Object-relational mapping (O/R) tools also provide a degree of orthogonal persistence. Examples include TopLink, JDO, EJB, and Hibernate [20, 26, 37, 34].

5.2 Explicit Query Execution (S2)

The primary alternative, and historical predecessor, to orthogonal persistence is the execution of queries written in a specialized query language. The main advantage of explicit query execution is that it allows the programmer to directly interact with the database engine.

```
string empQuery = "SELECT e.name, e.salary, d.name as deptName"
  + " FROM (Employee e INNER JOIN Department d ON d.ID = e.department)"
  + "         INNER JOIN Employee m ON m.ID = e.manager"
  + " WHERE e.name LIKE ? AND e.salary > m.salary"

Connection conn = DriverManager.getConnection(...);
PreparedStatement stmt = con.prepareStatement(empQuery);
stmt.setString(1, prefix + "%");
ResultSet rs  = stmt.executeQuery(empQuery);
while ( rs.next() ) {
  print( rs.getString("name") );
  print( rs.getDecimal("salary") );
  print( rs.getString("deptName") );
}
```

Fig. 3. Explicit query execution with JDBC

Embedded Queries Explicit queries may be embedded within the programming language or handled by a preprocessor [27]. Examples include SQLJ [8]. Embedded SQL provides a statically-typed approach to explicit query execution. One significant drawback of embedding is that it does not support dynamic queries, as discussed in Section 7. Another problem is that the change to the syntax of a programming language typically break other tools, including refactoring tools, IDEs, and CASE tools.

Call Level Interfaces The dominant mechanism for explicit query execution is the *call level interface* (CLI) [28], which allows a programming language to access a database engine through a standardized API [28, 24]. The key characteristic of a CLI is the ability to execute database queries and commands, which are represented as *strings* or other runtime data structures [26]. Figure 3 illustrates how the SQL query in Figure 2 can be performed using JDBC [24].

Most orthogonal persistence systems do not support explicit queries. Some, but not all, object-oriented databases support explicit queries. Most object-relational mapping (O/R) tools support explicit queries in addition to orthogonal persistence. Sometimes explicit queries are added to address performance problems; for example, EJB 1.0 did not include queries, but EJB 2.0 does. Other systems, including TopLink, JDO, and Hibernate have sophisticated query languages. Instead of strings, queries may also be represented by runtime data structures. Hibernate allows queries to be represented as criteria objects. In these systems the field names are still represented as strings.

Call level interfaces have a number of significant problems. The syntax and types of database programs are not checked statically, so any errors are not detected until runtime. Constructing and reusing queries at runtime requires complex and error-prone string manipulation. Query results are represented as dynamically typed objects that are accessed by string names. It is possible to

type-check embedded queries in some situations. Gould, Su and Devanbu [23] apply static analysis to check programs that use call level interfaces. Their analysis does not currently cover query parameters or result types and can produce incorrect results when components are compiled separately. Its primary advantage is that it can apply to existing programs.

Despite these problems, many commercial software development projects use call level interfaces to leverage database query optimizers and reduce communication latency in order to improve overall system performance.

6 Optimization

Most data-intensive applications handle large amounts of data. Therefore, it is important for the application to optimize data access. An appropriate query strategy may often be orders of magnitude faster than a naive query strategy [41].

The examples in this section present common database optimizations, but from a programming language viewpoint. In order to provide a fine-grained analysis of existing solutions, we separate the concepts of *search* from *navigation*. These correspond roughly to the WHERE and SELECT clauses of SQL: search is concerned with selecting a subset of objects of interest, while navigation is used to process the output of the search. This distinction is important because many solutions support one but not the other.

6.1 Optimizing Search

The first problem is optimization of search. The straightforward program given in Figure 2 takes time proportional to the number of employees, yet only a few of them may match the prefix. Traditional database optimization techniques can be applied to improve this algorithm by using an index. This can effectively reduce the running time to be proportional to the number of matching employees.

Explicit Indexes (P1) One common technique is to invert the order of suboperations like iteration and testing. Thus the query optimizer may use an index to compute the set of identifiers for records that match a condition, then find the corresponding data by looking up these identifiers in a second index. This aspect of the plan is illustrated in Figure 4, which includes this optimization in the original Java code.

The prefix test is implemented by searching an index: the `match` method returns an iterator over the index matches. An efficient index from record IDs to record values is then used to find the employee data. This code will be more efficient than the linear search as long as most names do not start with the prefix. Explicit programming against indexes is supported by Exodus [13] and Ontos [40].

Programmer productivity is significantly reduced if such optimizations must be coded by hand, because even a slight change to the original unoptimized

```
void printInfo(String prefix) {
  for (IndexItem l in employeeNameIndex.match("S")) {
    Employee e = employeeID_Index.lookup(l.ID);
    if (e.salary > managerID_Index.lookup(e.managerID).salary) {
      print( e.name );
      print( e.salary );
      Department d = departmentID_Index.lookup(e.deptID);
      print( d.name );
} } }
```

Fig. 4. Optimized printing of employee information

code, e.g. adding another condition to the if statement, may require a significant rewrite of the optimized form.

Criteria Shipping (P2) Most databases manage indexes automatically and perform query optimization based on detailed knowledge of the structure, content, and location of the data [21]. A database query optimizer will build a plan based on the complete operation being performed. Query optimization takes into account the details of the operation being performed and the current context in which it will be executed. Context includes statistical properties of the actual data being stored, the amount of memory available, the load on the processor, the frequency of different kinds of queries, etc. These kinds of optimizations require specialized knowledge of the data and its relationships – but this information is typically not available to compilers for general purpose languages.

Explicit query execution is a pragmatic approach to search optimization: it reduces the number of round-trips to the database and also gives the query optimizer more scope for optimization. Of course, it requires programmers to manually create appropriate queries.

It is also possible to define programming constructs that allow search criteria to be defined using standard boolean expression syntax but executed as a query against the database. The Linq extension to C# uses this technique to collect queries to be sent to the database [18, 10]. A new iteration construct, similar to a SQL select statement, is added to indicate which criteria that can be remotely executed.

The standard iteration constructs in AppleScript allow search criteria to be specified relative to the object model of a remote application [2]. The resulting search criteria are passed to the remote application for efficient execution.

Existing syntax can also be used to express queries. Safe Query Objects use an ordinary boolean method to define a query [17]. Rather than executing as standard byte-code, the queries are converted to database queries and appropriate wrapper code to call a database CLI. Because the queries are type-checked before conversion, the CLI calls will not cause type errors.

6.2 Optimizing Navigation

In addition to optimizing the search for objects, it is also important to optimize navigation to related objects. The problem is that there is typically significant *latency* in loading objects from the persistent store.

Prefetching Related Objects (P3) The original code in Figure 2 traverses the `department` relationship to print the name of each employee's department. In a persistent object system this traversal will load the appropriate department object, if it has not been already loaded. When persistence is connected to a relational database, each department may be loaded with a separate query, significantly reducing performance.

Most existing object persistence runtimes do not optimize navigation, although techniques for improving performance using prefetching have been explored [9]. This issue, and its interaction with modularity, will be discussed again in Section 7.

Call-level interfaces require the programmer to specify the data produced by the query, therefore, the programmer is responsible for navigation optimization. Object-relational mapping tools support limited navigation optimization. EJB and JDO can specify automatic loading of related objects, but this is currently a global property, not specific to a query. Toplink and Hibernate 3 have more flexible support for navigation optimization, but adding appropriate loading hints is cumbersome, and the mechanisms are not fully general. For example, TopLink only supports loading one level of multi-valued sub-attributes. Optimization of navigation should be a goal for any solution to impedance mismatch.

Multilevel Iteration (P4) A particularly difficult case of navigation is multiple levels of interaction through multi-valued relationships. This pattern is awkward to express in current SQL. One example is multi-level iteration, in which several levels of multi-valued relationships are included in the results of a query. Figure 5 illustrates this pattern. Even if collections of related items are loaded in one query, a query is needed to load the employees of each department, and the projects of each employee. If there are n departments in Austin and on average m employees per department in Austin, $1 + n + nm$ queries would be executed.

It is possible to load the required data in three queries: one to load departments in Austin, one to load employees whose department is in Austin, and one to load projects of employees who work in departments in Austin. The condition at the top of the loop must be replicated in each of the queries. The sorting orders must also be carefully nested if all the results are to be returned in the right order. Finally, the client must associate items in one table with corresponding subsets in nested tables. Note that a single query is also possible, although department and employee names must be replicated.

This common idiom cannot be expressed in SQL, although it is possible in OQL. In revising SQL, more attention should be placed on the kinds of queries

```
for each Department d in DB.getDepartments() sorted by size
   if d.city = 'Austin' then
      print( d.name );
      for each Employee e in d.employees sorted by e.name
         print( e.name );
         for each Project p in e.projects sorted by p.date
            print( p.name );
```

Fig. 5. Pseudo-code for multi-level iteration

that are needed to support object-relational mapping [42]. Taking the development of RISC processors as an analogy, SQL can be viewed as a form of assembly language: rather than design a clean, human-readable interface, it is more effective to measure the common patterns generated by client programs to design an optimized interface. There has been some research into detecting multi-level iteration and improving its performance using prefetching[25], however, no commercial system has implemented this type of prefetching.

6.3 Bulk Data Manipulation (P5)

While searches inherently involve many objects, updates are frequently performed only a few at a time, leaving little room for the kinds of optimizations described in the previous sections. However, there are typically a few cases in any application where *bulk* data manipulation is required. Data manipulation operations include inserts, deletes, and updates to data. A simplified example is the following:

```
for (Employee e in db.allEmployees() )
  if ( e.department.name.equals("Sales") )
    e.salary = e.salary * 1.2;
```

The optimizations described in the previous section also apply in this case. It is easy to use explicit query execution to run a custom SQL statement to perform a bulk operation.

```
UPDATE Employee set salary = salary * 1.2 from Employee
INNER JOIN Department d ON d.ID = e.Department
WHERE d.name = 'Sales'
```

But neither object-relational mapping tools or Java-based persistent programming languages allow bulk update operations to be efficiently executed in relational databases. AppleScript allowed update operations to be executed remotely. The DBPL language [38] and it successor Tycoon [35] explored optimization of search and bulk operations within the framework of orthogonal persistence. Tycoon proposed integrating compiler optimization and database query optimization, but no final results were published [22]. Queries that cross

modular boundaries were optimized at runtime by dynamic compilation [39]. No performance evaluations have been published for DBPL or Tycoon; the only published metrics cover lines of code in the implementation.

Caching Caching strategies are independent of the design of the *interface* between the language and the database, at least as far as the programmer using the interface is concerned. The implementor of data access infrastructure and databases must clearly pay attention. However, our focus here is on design issues that affect the *user* of data access infrastructure, not implementors.

7 Reuse

The previous section discussed issues of local optimization. In addition, there are issues relating to composition, or decomposition, of operations.

Parameterized Queries (R1) Parameterized and dynamic queries arise when queries are extracted from a program for remote execution in a database. For example, the prefix becomes a query parameter when moving from the orthogonally persistent code in Figure 2 to the explicit execution of a SQL query in Figure 3. With explicit query execution, query parameters are awkward to specify, and the types of parameters are not checked until runtime.

Dynamic Queries (R2) Dynamic queries are query strings that are constructed at runtime. Although dynamic queries would seem to be a terrible idea, they are quite common and must not be dismissed out of hand. Dynamic queries can be handled by partial evaluation of a query relative to values that affect the query but do not depend upon the database [17]. For example, if a search form in a user interface allows a set of optional search criteria to be specified, the resulting query can be partially evaluated relative to the choice of which criteria to include. A detailed explanation is given in [17]. Dynamic queries also arise in implementing fine-grained authorization rules that apply individually to each user [36]. We conjecture that lack of support for dynamic queries is the primary reason for abandonment of most forms of embedded SQL [27].

Dynamic queries are also needed to create *ad-hoc joins* in reporting applications supporting online analytical processing (OLAP). These applications are outside the focus of the work presented here.

Modular Queries (R3) Since most general-purpose programming languages have support for modular decomposition – using functional and data abstraction – the corresponding persistent languages have the same capabilities. It is well-known that modularity can interfere with optimization, but the problem may be worse in relation to database access. Both search and navigation optimization depend upon knowing all the conditions and data involved in access to persistent

data. Query optimization works best when large units of work are sent to the database, rather than individual operations. This both reduces the number of round-trips, and also gives the query optimizer more scope for optimization.

It is worth noting that while relational algebra supports modular composition, because every query is a relation, in practice it can be quite difficult to combine the effect of two SQL queries at a syntactic level. A solution to integration of languages and databases should support modularity, composition, and reuse of data-intensive program structures.

8 Concurrency

Concurrency in databases has a different focus from the kind of concurrency supported in typical programming languages [3]. Programming languages typically use threads and synchronization to define concurrent processes that *cooperate* to achieve an overall goal while periodically communicating, or requesting shared resources.

In a database, concurrency is viewed as *competitive*: multiple transactions compete for access to shared resources. Individual operations may be interleaved, as long as the overall transactions are serializable. At any point the transaction manager can abort and roll back a transaction. Databases provide guarantees on the overall behavior of the system, programming languages typically do not.

Concurrency causes significant problems for the pure form of orthogonal persistence [11]. Rather than insist on complete orthogonality, research should focus on providing semantically meaningful transaction/recovery behavior [19] within programming languages.

Search optimization also benefits transactions: if an operation touches many objects but updates only one, the touched objects create a large "footprint" that can interfere with other transactions. Search optimization reduces the number of objects touched in a transaction. Long-running transactions are more likely to block other transactions or be aborted.

9 Evaluation

Figure 6 summarizes the results of qualitative evaluation of different existing solutions to impedance mismatch. The evaluations in the optimization and maintenance sections each have two grades: The first grade is support for the feature. The second grade specifies whether static typing applies to the feature. For systems that were not designed to work with relational databases, a mapping to a relational database is certainly possible.

PJama and OPJ are pure orthogonally persistent variants of Java [30]. These systems don't support true concurrent database transactions, but do have a notion of "checkpointing" a globally consistent state of a store [11]. Although the goal of pure orthogonal persistence prevents the use of explicit queries, it is an open question whether a Java compiler could use criteria shipping or navigation

	PJama OPJ	Exodus Ontos EJB 1.0	ObjStore O_2 JDO 1.0 EJB 2.0	Hibernate TopLink	ODBC JDBC	SQLJ	S/NQ	Linq
Types								
T1. Mapping	√	√	√	√	×	×	√	√
T2. Nulls	√	√	√	√	√	√	√	√
Interface								
S1. Orthog. persistence	√	*	*	*	×	×	*	×
S2. Explicit query exec.	×	×	√	√	√	√	√	√
Optimization								
P1. Explicit indexes	× –	√ ×	–	–	–	–	–	–
P2. Criteria shipping	× –	× –	√ ×	√ ×	√ ×	√ √	√ √	√ √
P3. Navigation prefetch	× –	× –	* √	√ *	√ ×	√ √	× –	* *
P4. Multilevel iteration	× –	× –	× –	* ×	× –	× –	× –	* √
P5. Bulk data manip.	× –	× –	× –	× –	√ ×	√ √	× –	× –
Reuse for explicit queries								
R1. Query parameters	–	–	√ ×	√ ×	√ ×	√ √	√ √	√ √
R2. Dynamic queries	–	–	√ ×	√ ×	√ ×	× –	√ √	√ √
R3. Modular queries	–	–	√ ×	√ ×	√ ×	× –	√ √	√ √
Concurrency								
C1. Transactions	×	√	√	√	√	√	√	√

√√ = Feature supported and statically typed
√× = Feature supported but not statically typed
× – = Feature not supported
√ = Feature supported
* = Partially supported
× = Not supported
– = Not applicable

Fig. 6. Summary of qualitative evaluation of solutions to impedance mismatch

prefetch as part of its compilation strategy. It should also be possible to include a true model of concurrent ACID transactions within these systems, although this would reduce the degree of orthogonality for persistent data.

Exodus [13] and Ontos [40] are object-oriented databases with their own storage manager. Programmers can use explicit indexes to optimize search. In Exodus the index key is statically typed, but the data objects loaded from the index are not. EJB 1.0 is an object-relational mapper that resembles an object-database interface. EJB 1.0 did not support explicit queries, but allowed *finder methods*, which are a form of explicit index.

ObjectStore and O_2 are object-oriented databases with their own storage manager [40]. They support criteria shipping through the use of criteria objects. But these are not statically checked, and fields are named by strings.

JDO 1.0 and EJB 2.0 are object-relational mapping tools for Java. They provide a high degree of orthogonal persistence [29]. Criteria shipping is supported.

Navigation optimization is partially supported: it can be specified as a global property of a relationship, but not on individual queries.

Hibernate and TopLink are object-relational mapping for Java a degree of orthogonal persistence. Each has a specialized language for criteria shipping. These query languages support criteria shipping and navigation prefetch, although the prefetch specifications are not fully general. They also provide partial solutions to multilevel iteration.

ODBC and JDBC are standard call level interfaces. They provide full functionality but no static typing. Optimization of multilevel iteration is difficult using these interfaces. SQLJ [8] is a form of embedded SQL for Java. It provides a high degree of static typing, but does not support dynamic or modular queries.

Safe/Native Queries (S/NQ) [17] and Linq [18] are two recent proposals. Safe Queries build on top of an object-relational mapping tool, like JDO or Hibernate, but provide static typing of queries, which are represented as standard Java classes. Linq is an extension to C# to define statically typed queries over relational or other data sources. Unlike embedded SQL, these approaches support dynamic queries. Linq supports a form of multi-level iteration; since it uses a single join to return all the data, the values from enclosing iterations are repeated for each item returned for a nested iteration. It also allows a form of prefetch, although the objects returned are records, not instances of mapped classes. Note that the evaluation of Linq is still preliminary.

10 Conclusions

A complete solution to the problem of impedance mismatch must provide both a clean programming model and high performance. While issues of mapping data between databases and programming languages have largely been resolved, significant issues remain. The interface should leverage the best capabilities of both databases and programming languages to for optimization, static typing, and modular development. Each of these aspects has a solution by itself. The problem of impedance mismatch is meeting all the goals simultaneously. In this paper we have proposed qualitative criteria for evaluating proposed solutions, and evaluated a range of existing solutions against these criteria.

References

1. Scott Ambler. Mapping objects to relational databases. 1998.
2. Apple Computer Inc. *AppleScript Language Guide*. Addison-Wesley, 1993.
3. M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.
4. Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. The object-oriented database system manifesto. pages 946–954, 1994.
5. Malcolm P. Atkinson. Persistence and java - a balancing act. In *Proceedings of the International Symposium on Objects and Databases*, pages 1–31, London, UK, 2001. Springer-Verlag.

6. Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.
7. Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.
8. Julie Basu. An overview of SQLJ: Embedded SQL in Java. Oracle Open World, 1998.
9. Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch for implementing objects on relations. In *The VLDB Journal*, pages 327–338, 1999.
10. Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cω. In *European Conference on Object-Oriented Programming*, 2005.
11. Stephen Blackburn and John N. Zigman. Concurrency — the fly in the ointment? In *POS/PJW*, pages 250–258, 1998.
12. Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 441–451. ACM Press, 1987.
13. M. J. Carey, D. J. DeWitt, G. Graefe, D. M. Haight, J. E. Richardson, D. T. Schuh, E. J. Shekita, and S. Vandenberg. The EXODUS extensible DBMS project: An overview. In D. Maier and S. Zdonik, editor, *Readings on Object-Oriented Database Sys.* Morgan Kaufmann, San Mateo, CA, 1990.
14. Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 3–14. Morgan Kaufmann Publishers Inc., 1996.
15. Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 414–426. ACM Press, 1994.
16. R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, January 2000.
17. William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106, New York, NY, USA, 2005. ACM Press.
18. Microsoft Corporation. The LINQ project. `msdn.microsoft.com/netframework/future/linq`.
19. Laurent Daynes. Extensible transaction management in PJava. In *Proceedings of the First International Workshop on Persistence and Java (PJW2)*, 1996.
20. Jacques-Antoine Dub, Rick Sapir, and Peter Purich. Oracle Application Server TopLink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.
21. Michael J. Franklin, Björn Thór Jónsson, and Donald Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 149–160. ACM Press, 1996.
22. A. Gawecki and F. Matthes. Integrating query and program optimization using persistent CPS representations. In Malcom P. Atkinson and Ray Welland, editors, *Fully Integrated Data Environments*, ESPRIT Basic Research Series, pages 496–501. Springer Verlag, 2000.
23. C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings, 26th International Conference on Software Engineering (ICSE)*. IEEE Press, 2004.

24. Graham Hamilton and Rick Cattell. JDBC^{TM}: A Java SQL API. Sun Microsystems, 1997.
25. Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational dbmss. *Information Sciences*, 152(1):47–61, 2003.
26. Hibernate reference documentation. `http://www.hibernate.org/hib_docs/v3/reference/en/html`, May 2005.
27. INCITS/ISO/IEC. Information technology - database languages - SQL - part 5: Host language bindings (SQL/Bindings). Technical Report 9075-5-1999, INCITS/ISO/IEC, 1999.
28. ISO/IEC. Information technology - database languages - SQL - part 3: Call-level interface (SQL/CLI). Technical Report 9075-3:2003, ISO/IEC, 2003.
29. Mick Jordan. Comparative study of persistence mechanisms for the java platform. Technical Report TR-2004-136, Sun Microsystems, September 2004.
30. Mick Jordan and Malcolm P. Atkinson. Orthogonal persistence for the java platform - specification and rationale. Technical Report TR-2000-94, Sun Microsystems, September 2000.
31. 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 *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329. ACM Press, 1996.
32. David Maier. Representing database programs as objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1*, pages 377–386. ACM Press / Addison-Wesley, 1987.
33. Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent Java. In *Proceedings of the Workshop on Persistent Object Systems (POS)*, 2000.
34. V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.
35. F. Matthes, G. Schroder, and J.W. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M.P. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.
36. Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 551–562. ACM Press, 2004.
37. C. Russell. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 2003.
38. Joachim W. Schmidt and Florian Matthes. The DBPL project: advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.
39. J.W. Schmidt, F. Matthes, and P. Valduriez. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*. Springer Verlag, October 1993.
40. V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and O_2. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 21(1):93–105, 1992.
41. Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

42. W.P. Zhang and Norbert Ritter. The real benefits of object-relational db-technology for object-oriented software development. In B. Read, editor, *Proc. 18th British National Conference on Databases (BNCOD 2001), Advances in Databases*, pages 89–104. Springer-Verlag, 7 2001.