Coordinating Database and Programming Language Research*

Ali Ibrahim
University of Texas at Austin
aibrahim@cs.utexas.edu

Ben Wiedermann
University of Texas at Austin
ben@cs.utexas.edu

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

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

Abstract

In this essay we examine the gap between database and programming language research and practice. Relational databases and object-oriented programming have been great success stories over the last 40 years. While the database community works hard to improve and extend database capabilities, it does not seem to pay much attention to how databases are actually used. We still use query languages that were designed for ad-hoc human queries via unique logins, while many queries are now automatically generated by enterprise applications with sophisticated security models. Many in the programming language community view relational databases as a necessary evil that should be papered over, or completely eliminated if possible. Object-oriented databases, orthogonal persistence, or just the file system, all have proponents. Industry experiences a constant churn of APIs and tools, with little guidance on what architectures really work. What is needed is more coordination between database and programming language researchers, to evaluate complete systems with realistic metrics, not just for performance but also for maintainability.

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

1 Introduction

This essay discusses the relationship between database and programming language research and practice. Our goal is to present a call to action and stimulate discussion by presenting our subjective view of the problem, our interpretation of its history, and a vision for next steps.

The key issue we wish to discuss is the cultural and technical separation between database (DB) and programming language (PL) research. Much of the world's critical information infrastructure is built by combining general-purpose programming languages with relational databases. These *enterprise applications* manage the flow of untold transactions daily in support of our government, business, and personal lives. They support a mixture of online transaction processing (OLTP) and online analytic processing (OLAP). In this essay we focus on transaction processing applications, although similar points could be made about analytics. Building, managing, and maintaining these applications is a primary focus of a large portion of the software developers active today, spanning commercial software companies, global consulting firms, and corporate information technology departments. These systems are primarily built using object-oriented languages for general-purpose computation and relational databases to control concurrent access to data, efficiently search large amounts of data, and/or update data reliably.

Object-oriented programming and relational databases are two great success stories. They both began about the same time 40 years ago. Relational databases dominate both theory and practice in the database world. Object-oriented programming is ubiquitous in practice, but is still somewhat controversial within the programming language research community.

Yet for all its success, we argue that the work is still not complete. Industry struggles to interface programming languages and databases. Applications that access databases are awkward to design and develop. Careful optimizations are often needed to attain good performance, resulting in programs that are difficult to maintain and evolve. We discuss some of the different schools of thought on how best to architect enterprise systems, but the more fundamental problem is a lack of systematic efforts to evaluate these designs. New APIs and proposals are created every year; despite some significant recent progress, the problem is far from completely solved. At a theoretical level there may not be any problem at all. If so, then perhaps our theories are too abstract to capture the essence of a problem that is very real to practitioners.

The thesis of this essay is that programming language and database research and practice can benefit from greater coordination to solve joint problems. Right now there is very little coordination or mutual understanding between the research communities. The authors cannot claim to know the truth about how this came about, all we can do is try to infer causes from the effects we see today in our daily work.

2 Cultural Divide

Our experience is that the database community seems to have little interest in how databases are used to build larger systems. One explanation for this could be that the use of databases in larger systems is out of scope; that it belongs to software engineering, or industry should figure it out. But whenever we mention this to senior database researchers, they have agreed and said that it is a problem.

The programming language community has an analogous tendency: a sense that databases aren't really necessary, or can be subsumed by the programming language runtime. A cynical interpretation is that the kinds of programs programming language researchers usually write (e.g., compilers, type checkers) are not the kind that need industrial strength databases. But there can be good technical reasons for taking this view.

Orthogonal persistence [AM95] is a natural extension of the traditional concept of variable *lifetime* to allow objects or values to persist beyond a single program execution [AB87]. In the most pure form, persistent values exist as long as they are referenced (transitively) by a persistent root. 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 manipulate persistent data by *navigation*, traversing object references as they would for in-memory data structures.

Examples of orthogonal persistence systems include PJama [ADJ+96], Thor [LAC+96], and OPJ [MBMZ01]. We believe orthogonality is not absolute, but describes the degree of uniformity in the treatment of persistent and non-persistent data. A fully orthogonal persistent version of a conventional language cannot have a transaction model [BZ99], although other options may be possible if both persistent and non-persistent data are transactional.

The idea of orthogonal persistence is appealing to the programming languages community. Programmers are already familiar with using navigation for data access. Studies of orthogonal persistence can also show much better performance than systems based on relational databases [Jor04]. In Section 4 we discuss these studies.

Object-oriented databases (OODBs) are a form of orthogonal persistence, as described in the Object-Oriented Database System Manifesto [ABD+89]. But OODB implementations were not necessarily able to meet all its goals. For example, early OODBs did not support automatic indexing, query optimization, or transactions. The Object Query Language (OQL) [Bie03] which was eventually proposed for object databases was not adopted by many databases. We disagree with some requirements in the manifesto, for example, the requirement that behavior (methods) be stored in the database [ADJ+96]. Finally, some important requirements were optional or omitted entirely, including evolution and support for multiple client languages.

We sense that object-oriented databases are a sore point for many researchers and practitioners. The common belief on the fate of OODBs was expressed in a review of an early version of this essay: "the problem [integration of PL and

DB] raised in this paper is solved by OODB. The only reason why we are still talking about it is that the big database vendors managed to kill the competition instead of adopting the new technology. ... The issue is not technical—we had the technical solution. It is a business issue—convincing the big guys to modify dramatically their engines and move away from dirty solutions such as object-relational."

We don't believe the fundamental assumption of this argument: that OODBs fully covered *all* the requirements for building enterprise applications. This is just a conjecture, but it is one of the key points in this essay: we believe that object databases have never been fully evaluated with respect to relational databases for effectiveness in building, deploying and maintaining large, scalable, reliable enterprise applications. Small changes in the relational style, to create object-relational databases [CD96], are not a significant step toward OODBs. This evaluation may find points lacking in OODBs that can lead to new research opportunities.

2.1 Data & Access Models

It is often assumed that the mismatch between object and relational data models is the main source of problems in connecting programming languages and databases. We believe that the more fundamental problem is the mismatch between *access* models; programming languages rely on navigation while databases rely on queries.

The Entity Relationship (ER) model [Che76] is a point of unification between relations and objects. It is common practice to use ER models for logical database design, then translate these to relational tables. ER diagrams are also the basis for UML Class diagrams [Bur97]. The fundamental model is that of a graph of nodes and edges representing an information model. Objects and relations are two different representations of abstract information models. Trees and XML have a natural place in this unification as well: trees have a natural embedding in graphs. Alternatively, relations and objects can be encoded as trees [MS03].

There are many other differences, in the details, between relational, object-oriented and XML tree implementations of information models: handling of null values, encoding of inheritance, relationships, etc. However, none of these differences adds up to fundamental incompatibility in the structure of data.

The real pain in integrating programming languages and databases comes not so much from the mismatch in representation as from the differences in access style: navigation versus queries. Programmers prefer to navigate predefined relationships, rather than specify joins when connecting objects. These relationships may be specified with ER or UML diagrams. Some database researchers feel that graph-based navigation is tainted by the failure of the network data model (CODASYL) [TF76] However, CODASYL was a flawed experiment, and so its failure is not an indictment of graph-based navigation. Another argument against the primacy of ER models is that they do not allow for ad-hoc joins. However, transactional enterprise applications generally do not perform ad-hoc

joins, although they are frequently used in analytic processing.

Access patterns are the key to Maier's original definition of *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" [Mai87].

Historically the dominant way to "pick out database operations" is to embed SQL queries into object-oriented programs. But manipulating SQL as strings in a program is fraught with well-known problems, so a series of increasingly sophisticated and complex libraries has been created to generate queries automatically.

Functional programming is a clean way to bridge the gap between navigation and queries. The `map` function from Lisp applies an operation to all items in a collection. List comprehensions are an alternative notation for mapping and filtering without using an explicit higher-order function [Won00]. C# has recently been extended with a reflective mechanism to access the abstract syntax of a higher-order function, enabling user-defined translation of C# functions into alternative execution environments. This technique was used to create a data access library called LINQ [BH07] which translates map into a procedure over data. DLINQ and XLINQ perform the specialized translations needed for data access from databases and XML respectively. These advances in programming languages promise finally to overcome the impedance mismatch. More work is needed, however before this solution provides full functionality for updates, prefetch, security, and modularity.

3 The DB/PL Interface

We believe that there are good reasons for maintaining separation between application programs and databases. Business and historical reasons often dictate that different applications possibly written in different languages share a single database. Separating the application and database also allows developers to distribute and replicate each component to provide better performance and scalability. If such a separation is desirable then it is important to study this interface and—from time to time—re-evaluate its design in light of the continuous progress made in both domains.

One conclusion from the previous section is that SQL queries in enterprise applications are automatically generated by translation libraries. In fact, many of the applications we have worked on have no hand-written SQL. Since SQL is the target of translation from model level query languages or program navigation patterns, it is reasonable to view SQL as an assembly language of data. As in hardware assembly language, the words of memory (tables) are interpreted by the operations (joins) performed upon them, but have little inherent semantics of their own.

One of the design ideas behind reduced instruction set architectures (RISC) was that the interface between the compiler and the architecture can be analyzed and optimized, by moving functionality between the architecture and the

compiler. SQL could benefit from a similar analysis and would be better able to accommodate queries generated by high-level translators. For example, a common logical operation is to retrieve a set of entities and some of their relationships. But entity graphs or trees cannot be compactly represented in SQL query results, so generators have to balance a tradeoff between the size of the results and the number of queries.

Our point is not to promote the inclusion of specific features in SQL or any other query language. Instead we are suggesting that the interface between programming languages and databases is a fertile ground for research, which looks at overall system behavior and considers the interfaces as flexible. How might databases be modified to better interface with programming languages? Other topics that might be reconsidered in this light are active databases, which include some application functionality in the database, and security. In the interest of space, we discuss security and concurrency briefly.

Security is a large area with many aspects. Two that we have experience with are authentication and authorization. Databases typically have an authentication model based on login IDs. Programming languages do not have predefined authentication models, although the current user is typically represented as an application object. In an enterprise system, the application server typically authenticates to the database with a single system login ID; the end-users of the application do not have database login IDs. As a result, the authorization models supported by databases cannot be used, since they depend on users having unique login IDs.

Even if they could be used, database authorization models are not typically sophisticated enough to implement the attribute-based authorization that is more and more common in enterprise applications [GGW02, OGM08]. Query rewriting is often used to implement security, leading to even more complex automatic query generation [RMSR04]. More study is needed to determine how to partition authorization tests between the application and the database.

Concurrency is also an area in which applications and databases use different techniques. Traditionally, programming languages use locks to manage concurrency, while relational databases use transactions. Concurrency which spans application logic and data access is messy. An example of such concurrency is in-memory cache management. In practice, applications demarcate database transactions with explicit library calls or meta programming, e.g. Java annotations, and interpret database transaction states using error codes and exceptions. There is no way for a database to signal an application that a transaction has failed until the next application command. With software transactional memory, there is an opportunity for a more uniform interface. Currently, the few languages such as Fortress [Sun07] that do support transactional memory do not provide any method to integrate external transaction partners such as databases.

4 Evaluation

A thorough evaluation of an enterprise application is a monumental undertaking. Even if researchers could agree on the standards against which such applications should be evaluated, the heterogeneous nature of enterprise applications frustrates researchers' attempts to evaluate them. This obstacle, however, presents an opportunity to researchers: If we are able to provide more complete evaluation of our efforts we can be of real service to application developers and facilitate the adoption of our efforts in industrial applications. To meet this challenge, the database and programming language research communities should work together to widen our scope as much as possible. The effort will require significant investment in research infrastructure, including platforms, tools, methodologies, and benchmarks.

Every researcher is aware of the gulf that exists between most research platforms and their industrial counterparts. The PL / DB research community maintains a symbiotic, though at times estranged, relationship with industry. Our research ideas are not fully validated until they are deployed, yet we often lack industry-quality platforms on which to develop and test our research. This means that we cannot provide industry-quality evaluation of our results.

Developers who write enterprise applications attempt to maximize the performance of a conflicting set of application behaviors, but our current research methodologies do not adequately address this reality. Enterprise applications should be evaluated on a range of metrics including performance, scalability, resilience to failures, development costs and maintainability. Some of these metrics—like performance and maintainability—are often in direct conflict.

It is a tall order to provide quantitative metrics for heterogeneous systems. It is much easier to evaluate a homogeneous system with a well-defined interface against a single objective measurement. But we must consider our audience and attempt to address all their concerns.

Most current evaluation methodologies focus on one or only a few metrics such as program execution time, number of queries executed, or query execution time. Future methodologies should also report other performance metrics including overall system throughput and latency, memory footprint, and communication time and bandwidth. Methodologies should also measure the system's software engineering properties by reporting soft metrics including maintainability, extensibility and scalability.

If we are to provide a more realistic evaluation of our research, we need more realistic benchmarks. Current popular benchmarks, including OO7 [CDKN94] and TORPEDO [Mar05], do not accurately represent the reality of enterprise application architecture nor are they equipped to measure the full range of metrics we have advocated. If we are not measuring our work against a realistic standard, how can we be sure our efforts are not misguided?

Although we advocate ambitious goals for future methodologies, we believe our communities can achieve these goals by making our research techniques and results more available to one another.

5 Cultural Exchange

Programming Languages and Databases are both relatively small specialties within computer science whose basic concepts are not very well-known outside the specialty. At job talks for candidates seeking tenure-track positions in databases or programming languages, we have noticed that many computer science academics do not have solid understanding of either relational algebra and query optimization, nor of lambda calculus and abstract interpretation.

Yet databases and programming languages have a lot in common. A database management system can be viewed as an interpreter for SQL, a hybrid functional/imperative domain specific language. The query engine is a very sophisticated algorithm compiler. New database optimization techniques can achieve orders of magnitude speedups that researchers working on compilers for general purpose languages can only dream of. Issues of data representation, modularity, security, and abstraction are also relevant to both databases and programming languages.

Our separate theoretical foundations, formalisms and vocabularies naturally impede cultural exchange. The programming language and database research communities rarely communicate their successes directly to one another. Nor does either community communicate its needs to the other. There is a biannual conference on Database Programming Languages [AS07]. However, a quick review of the papers reveals that very few of them address the goal of the conference directly, which is to explore the intersection of programming languages and databases.

The economic culture of each community also impedes cultural exchange, because it affects the availability of quality research platforms. Programming tools are commercially viable (e.g., compilers, IDEs) but have never been the foundation for large companies. Programming tools typically support operating systems, and operating system vendors invest in tools in order to drive adoption and innovation on their platforms (e.g., Microsoft, Sun, Apple). Many widely-used programming language tools (e.g., gcc, Eclipse) are open source. Databases on the other hand have been the foundation for several large companies. More recently there has been increasing success of open-source databases.

Programming language researchers therefore have access to quality research-oriented implementations of language runtimes (e.g., Jikes RVM) and language and compiler tools (e.g., Polyglot, Soot and JastAdd). Unfortunately, databases are not everyday tools for most computer science academics and certainly not for most programming language researchers. We feel these effects in our research efforts: other than Berkeley DB and Apache DB, we are unaware of research database platforms.

6 Conclusions

The thesis of this essay is that there are research opportunities related to the assembly of complete systems that incorporate databases and programming lan-

guages. We believe that improving enterprise systems is not just a matter of technology transfer of existing results. To address these opportunities, researchers must redefine the scope of the problems they are willing to address. We believe:

- Programmers prefer logical Entity-Relational/Class models with navigational (OQL) query languages rather than direct use of the relational model and joins. As programming languages blend object-oriented and functional features, it is easier for them to express queries concisely and check them for type safety.
- Most queries are generated automatically, and often use query rewriting to implement security resulting in very dynamic queries. Many question the prevailing wisdom that queries must be hand-crafted as stored procedures. Databases should embrace this trend and provide an interface that is designed for automatic query generators, not humans. This is similar to the shift from CISC to RISC in hardware architecture.
- Databases and programming languages should be evaluated in the context of complete systems, with metrics for scalability, redundancy, and maintainability in addition to performance. There is a lot of uncertainty about how to partition and architect effective solutions. This could be a fertile ground for new research, and it would help to lay to rest some long-standing open questions about persistence models.
- Research communities should deliver research results in a way that is usable by other research communities, not just in papers but in demonstration systems.

References

[AB87] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–170, 1987.

[ABD+89] Malcolm Atkinson, François Bancilhon, David DeWitt, Klaus Dittrich, David Maier, and Stanley Zdonik. The object-oriented database system manifesto. In *Deductive and Object-Oriented Databases*, pages 223–240, 1989.

[ADJ+96] Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

[AM95] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–401, 1995.

[AS07] Marcelo Arenas and Michael I. Schwartzbach, editors. *Database Programming Languages, 11th International Symposium, DBPL 2007, Vienna, Austria, September 23-24, 2007, Revised Selected Papers*, volume 4797 of *Lecture Notes in Computer Science*. Springer, 2007.

[BH07] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. `http://msdn.microsoft.com/en-us/library/bb308959.aspx`, 2007.

[Bie03] G. M. Bierman. Formal semantics and analysis of object queries. In *The ACM SIGMOD International Conference on Management of Data*, pages 407–418, 2003.

[Bur97] Rainer Burkhardt. *UML: Unified Modeling Language*. Addison-Wesley, 1997.

[BZ99] Stephen Blackburn and John N. Zigman. Concurrency — the fly in the ointment? In *The International Workshop on Persistent Object Systems*, pages 250–258, 1999.

[CD96] Michael J. Carey and David J. DeWitt. Of Objects and Databases: A Decade of Turmoil. In *The International Conference on Very Large Data Bases*, pages 3–14, 1996.

[CDKN94] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *The ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 414–426, 1994.

[Che76] Peter P. Chen. The Entity-Relationship Model — Toward a Unified View of Data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976.

[GGW02] Richard Goodwin, SweeFen Goh, and Frederick Y. Wu. Instance-level access control for business-to-business electronic commerce. *IBM Systems Journal*, 41(2):303–321, 2002.

[Jor04] Mick Jordan. Comparative study of persistence mechanisms for the Java platform. Technical Report TR-2004-136, Sun Microsystems, September 2004.

[LAC+96] 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*, pages 318–329, 1996.

[Mai87] David Maier. Representing database programs as objects. In *The ACM SIGMOD Workshop on Database Programming Languages*, pages 377–386, 1987.

[Mar05] Bruce E. Martin. Uncovering database access optimizations in the middle tier with TORPEDO. In *The IEEE International Conference on Data Engineering*, pages 916–926, 2005.

[MBMZ01] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing Orthogonally Persistent Java. In *The International Workshop on Persistent Object Systems*, pages 247–261, 2001.

[MS03] Erik Meijer and Wolfram Schulte. Programming with rectangles, triangles, and circles. In *Conference on XML*, 2003.

[OGM08] Lars E. Olson, Carl A. Gunter, and P. Madhusudan. A formal framework for reflective database access control policies. In *The ACM Conference on Computer and Communications Security*, October 2008.

[RMSR04] Shariq Rizvi, Alberto Mendelzon, S. Sudarshan, and Prasan Roy. Extending query rewriting techniques for fine-grained access control. In *The ACM SIGMOD International Conference on Management of Data*, pages 551–562, 2004.

[Sun07] Sun Microsystems Corporation. The Fortress language specification. `http://research.sun.com/projects/plrg/Publications/fortress1.0beta.pdf`, 2007.

[TF76] Robert W. Taylor and Randall L. Frank. CODASYL database management systems. *ACM Computing Surveys*, 8:67–103, 1976.

[Won00] Limsoon Wong. Kleisli, a functional query system. *Journal of Functional Programming*, 10(1):19–56, 2000.