JAVA STATE OF THE UNION -- PART II

Editor Roberto V. Zicari, ODBMS.ORG- http://www.odbms.org

May 2, 2008

I have interviewed a group of leading persistence solution architects on their views on the current State of the Union of object persistence with respect to Java.

Part I of the panel is available at:
http://www.odbms.org/experts.html#article16

This is part II of the panel with the following experts:

Jose Blakeley - Microsoft
Dr. José Blakeley is a Partner Architect in the SQL Server Division at Microsoft where he works on server programmability, database engine extensibility, query processing, object-relational functionality, large scale query processing, and scientific database applications.
Before joining Microsoft, Dr. Blakeley was a member of the technical staff with Texas Instruments where he was a principal investigator developed of the DARPA funded Open-OODB object database management system.

Rick Cattell - Consultant
Dr. R. G. G. "Rick" Cattell was a Distinguished Engineer at Sun Microsystems. He has worked for over twenty years at Sun in management and senior technical roles, and for ten years in research at Xerox PARC and at Carnegie-Mellon University.
Dr. Cattell is best known for his contributions to database systems and middle ware --- particularly J2EE, object-oriented databases, object/relational mappings, and database interfaces. He is the author of several dozen papers and six books. He was a founder of SQL Access (a predecessor to ODBC), the founder and chair of the Object Data Management Group (ODMG), the co-creator of JDBC.

William Cook - University of Texas at Austin
William Cook is an Assistant Professor in the Department of Computer Sciences at the University of Texas at Austin. His research is focused on object-oriented programming, programming languages, modeling languages, and the interface between programming languages and databases.
Prior to joining UT, Dr. Cook was Chief Technology Officer and co-founder of Allegis Corporation. He was chief architect for several award-winning products, including the eBusiness Suite at Allegis, the Writer's Solution for Prentice Hall, and the AppleScript language at Apple Computer.
At HP Labs his research focused on the foundations of object-oriented languages, including formal models of mixins, inheritance, and typed models of object-oriented languages.

Robert Greene - Versant

Robert Greene is responsible for defining Versant's overall object database strategy and direction with over 15 years experience delivering OO solutions. Robert is an industry thought leader with extensive experience in object-oriented systems design writing and presenting regularly at Java conferences and seminars on the topic of object persistence and J2EE application architectures. Robert is an author on the topic of object relational mapping and is the project lead for the Eclipse EJB tooling initiative known as JSR220 ORM.

Alan Santos - Progress Software

Alan is currently a product manager for Progress Software helping to define the future role of the OODBMS in the enterprise. Prior to joining product management he was a principal engineer for the ObjectStore OODBMS focused on enhancing both the ObjectStore C++ and Java products. Prior to that he spent several years as a principal consultant for ODI; designing, implementing and supporting high performance OODBMS Solutions intended to solve a large, diverse number of problems for customers ranging from the fortune 500 to small privately funded startups.

Question 1: Do we still have an "impedance mismatch problem"?

[**Jose Blakeley**] The impedance mismatch problem has existed ever since the data models used to persistently store data whether file systems or database management systems and the data models used to write programs against the data (C++, Smalltalk, Visual Basic, Java, C#) have been different.

Today, I see two types of impedance mismatch problems: (1) the application's impedance mismatch problem, and (2) the impedance mismatch in data services. The first, is the type of impedance mismatch that Persistent Programming Languages and OODBMSs addressed in the mid 90's by embracing the type system of programming languages like C++, Smalltalk, Lisp as the data model for persistent data. Today, a lot of corporate data resides in relational DBMSs, and most significant applications whether they are packaged (e.g., CRM, ERP) or custom often implement a data access layer between the database and the application designed to bridge the impedance mismatch between the relational model and the object model used by the application. This level of indirection can represent about 40% - sometimes more - of the application code.
The second type of impedance mismatch appears in the way in which data services such as replication, reporting, business intelligence (OLAP) are written. These data services also persist or access relational data and need to bridge the semantic gap between the relational model and the higher-level conceptual models they use. We have found that such a conceptual model is closer to the Entity-Relationship Model introduced by Peter Chen in the mid 70's.

The software industry has created tools and services designed to help minimize the application impedance mismatch with object-relational mapping (ORM) technologies. In addition, programming languages are incorporating set-oriented, declarative programming expressions as native constructs inside these languages to further minimize, and in many cases completely eliminate, the impedance mismatch. The Language Integrated Query (LINQ) extensions and query expressions in C# 3.0 and Visual Basic 9.0 are examples of this.

For further details see the article "Next-Generation Data Access: Making the Conceptual Level Real":
http://msdn2.microsoft.com/en-us/library/aa730866(vs.80).aspx

[**Rick Cattell**] Given the popularity of object-oriented programming languages and relational databases, yes! The two solutions to the impedance mismatch problem are an object-oriented database system or an object/relational mapping. Most people have chosen the latter; the database choice is often already made for them.

[**William Cook**] Yes, we still have impedance mismatch. I believe that most of the impedance comes from integrating procedural languages and query languages, not from the data types (relational and object) involved. This goes back to the original paper “Representing database programs as objects” by David Maier: he said the issue was one-at-a-time operations versus bulk operations. Thus a key point is that you can have impedance mismatch even when using an object-oriented language and an object-oriented database. This happens when the object-oriented database has a query language, but the programming language cannot use this query language natively.

Linq is an example of a hybrid functional/procedural language, and it is a step in the right direction. However, the integration is still not seamless. There are issues with modularity in trying to optimize queries across procedure boundaries.

It is also awkward to load related objects, using LoadWith in Linq or fetch keywords in Hibernate. What related data to load depends upon the ways in which the query results are used. A given query can be used in different contexts, requiring different loading strategies. In addition, the use of a query’s results are often in a different tier of an application. For example, the query is often in the business logic tier, but the results get used in the user interface tier. As a concrete example, a business logic function might load employees, but the user interface determines what related information should be displayed about the employee on different pages.

[**Robert Greene**] Absolutely. The classic “impedance mismatch problem” can be defined loosely as the burden placed on developers in dealing with the disparity in programming –vs- storage paradigms. As tools have evolved, the impedance mismatch in that respect is better, but have you tried to map a 200+ class based application to a relational database? Anyone who’s done that

knows there is still an impedance mismatch, annotation or XML, it'll cause your hair to fall out……you're not publishing a picture with this are you ;-)

I would further assert that the mismatch is an issue that spans more than simply development burden. It is a "green" issue. The CPU work involved in handling the mapping back and forth through network calls is tremendous. Today it is well recognized that it cost more to power your computer than it does to buy it. Now if you can reduce the size of your production system by 50% by eliminating the impedance mismatch, then I say it's a "green" issue. The savings is tremendous for larger scale systems where that 50% means, less license cost, less maintenance costs, less power consumption and ultimately less hosting costs. Imagine if eBay, Amazon, Facebook, LiveJournal, etc could reduce their production systems by 50%. This is not an arbitrary figure, but something I've seen first hand with Versant customers.

[**Alan Santos**] Impedance mismatch will always exist as long as the language in which a program is written differs from the means in which data is made persistent.

Historically impedance mismatch has referred to the issues encountered when mapping data from a relational store into an object oriented data model. For some people, in some very practical ways, impedance mismatch is not an issue and has been solved with improvements in O/R mapping libraries and performance improvements in the runtime environments, as well as hardware itself.

In performance critical applications, with a complex data model, the problem has not been solved, and will never be solved given that performance expectations increase with improvements in hardware and software.

In other, non performance aspects, it still remains a significant issue. Three specific examples spring to mind: Complexity of development; maintenance and the inability of a RDBMS to efficiently model certain types of structures such as R-trees and Quad Trees. Time and effort spent dealing with the mapping problem is a delay in the time to a deliverable product, new features or bug fixes.

Finally, I believe that the problem is likely to increase in frequency, rather than decrease over time in that the range of participants in the mapping picture has increased. Mapping issues exist when integrating any object oriented language with a non ODBMS, from legacy systems to data services such as Amazons SimpleDB as well as the typical object to relational scenario. As non-relational persistence mechanisms continue to increase in acceptability within the software industry the exposure to to this particular problem will increase.

Question 2: In terms of what you're seeing used in the industry, how would you position the various options available for persistence for new projects?

[**Jose Blakeley**] Many applications and data services write their own proprietary data access layer designed to bridge the impedance mismatch problem. In the last 5 years, ORM technologies like Hibernate have seen increased popularity. At Microsoft, we have looked at the application and data services impedance mismatch problems and believe there is a need for a higher-level conceptual data model to raise the level of abstraction of the database above the relational model. This conceptual model is somewhat closer to an E-R model. This is a rich structural model that incorporates concepts such as entities, relationships, and inheritance, but not behaviors such as methods.

Multiple, programming-language-specific object layers can be built as a thin veneer around the entity based conceptual model. Such an object-layer would introduce methods, reference semantics and object caching in a programming-language specific way. The Microsoft instantiation of a conceptual model is called the Entity Data Model (EDM) and its supporting runtime is called the Entity Framework (EF). The EDM has a query language called EntitySQL which is a language for querying models defined using the EDM.

The EF implements all the entity-to-relational mappings which take care of transforming data from tables in the data base to entities and vice versa during updates. The EF automatically translates EntitySQL queries over entity sets to the underlying SQL queries over tables. The object services layer, built on top of the EDM, provides language-specific bindings and exposes LINQ queries over collections of objects. Over time, we see the conceptual EDM, EntitySQL and EF runtime being absorbed natively by relational systems.
For further details see:
1.Architecture of the ADO.NET Entity Framework:
http://research.microsoft.com/~melnik/pub/adonetindustrial_SIGMOD07.pdf

2. Compiling Mappings:
http://research.microsoft.com/~adya/pubs/compiling-mappings_SIGMOD07.pdf

3. Formal query extensions to the C# language:
http://delivery.acm.org/10.1145/1300000/1297063/p479-bierman.pdf?key1=1297063&key2=5515818911&coll=GUIDE&dl=GUIDE&CFID=47476501&CFTOKEN=59525605

4. A description of Entity SQL:
http://blogs.msdn.com/adonet/archive/2007/05/30/entitysql.aspx

[**Rick Cattell**] The top three options for Java are JDBC, O/R mapping, and an ODBMS. Over the past 10+ years of Java I've worked on all three of these approaches, and sub-approaches within them, e.g. JDO vs JPA APIs for O/R mapping. Each of them has its strengths.

[**William Cook**] I have been out in industry, but am in academia now. From a programming language side, Linq has made a great leap forward in integrating bulk data handling with procedural languages. I've been doing research on the problem in the context of Java. Practical work on Java, including my own, often assumes that the language cannot be changed, which has a significant influence of the range of designs that can be considered. It also leads to an increasing reliance on reflection and dynamic code generation, which can complicate the design. Here are some of the projects I've been working on together with industry.

AutoFetch is used to automatically infer what data to load when performing a query, based on past program behavior. You can think of a query as having two parts: the selection conditions, and the prefetch of related objects. AutoFetch handles the prefetch part, and it works very well. It's a general idea that could be applied to any DB/PL interface, including Linq. We have implemented and released a version for Hibernate, see
http://*www.cs.utexas.edu/~aibrahim/autofetch*

Rob Bygrave recently implemented AutoFetch for the Ebean ORM. AutoFetch cleanly handles the parts of query construction that are relate only to optimization of data loading, which are messy in Linq and Hibernate.

I worked on Native Queries with db4Objects. It uses Java methods to define query conditions, much as Linq uses lambda-functions. Native Queries don't require syntax changes to Java, but support query optimization and dynamic queries. They could also be combined with AutoFetch, but this has not been done as far as I know.

Finally, I have been working on a more long term project to automatically extract full queries from Java programs. This work uses static analysis to extract prefetch and conditions from code. It does not work as well for aggregation queries. Again, it could be combined with AutoFetch and explicit queries, as in Linq, to make a more powerful system. This full integration is still several years down the road.

[**Robert Greene**] This is a very complicated question because there are so many factors that influence the plethora of persistence options. In general, I say use the right tool for the job. The "job" varies though based on factors including and not limited to: money, available skill set, time to market, expected scalability requirements, high availability, longevity of solution, deployment platform, so many more factors. I am of the school of thought that of all the factors, expected scalability requirements should be the primary driver.

You see today's newer Web2.0 type companies are at a disadvantage compared to the old boys. When the eBay's of the early days hacked out their web site for free in the garage and launched their sites, it took a long time before 1M users hit it. The internet just wasn't that pervasive. So, they had time, in fact years,

to throw away first designs, rebuild, make mistakes, add caching, and eventually come to their functionally segregated, asynchronously orchestrated, horizontally data partitioned solutions.

So, if you are doing something relatively simple and small scale, then use what works the easiest, perhaps something like Ajax over a web server and a Java Content Repository. If you're in the middle of the road, figure out where the middle really is and make sure to shore it up with something tougher, say Ajax over an application server or spring or some framework you like while using an ORM solution to your favorite RDB. If you're on the high scale side of the equation, then you probably need to consider a proper SOA based solution built on an ESB or some other asynchronous workflow framework on top of a partitionable data tier. It's on that far scalability end that you'll be really happy if can have a production system that's gotten rid of the impedance mismatch problem and you're not funding Chevron's next big tanker.

Of course, to be complete, there is the other end of the spectrum. What if you're just trying to store some stuff on a mobile phone or smart appliance? What's wrong with Java serialization? Perhaps nothing, or maybe you want to query, so maybe you actually need an embeddable database? The factors and the possible persistence solutions are vast for good reason.

Oh gasp, complete I say, and I did not discuss XML storage. Enough said.

[**Alan Santos**] I think I can say that, in general, the software community considers there to be two options: Relational and Non-Relational.

It is with the slightest sense of hyperbole that I note no one has ever been fired for choosing a relational database. It is generally the de-facto, if not de-jure, choice for most organizations. In the typical cases of integrating with existing data sources; or where technological requirements are trumped by other choices, such as the need to easily obtain resources familiar with a technology; or where it's just acceptable to head down the path of least resistance, persisting Java objects indirectly into a relational store is certainly an acceptable option. In this context I include JDBC as well as O/R mapping techniques.

Outside of the enterprise there seems to be an awakening that relational storage is not always the optimal choice. In some ways this is the pragmatic realization that one should use the right tool for the job. I believe that large relational vendors have realized this, unfortunately when you only have a hammer everything looks like a nail and we ended up with data stores which aren't sure if they're relational or hierarchical; if they're a database or an application server.

I would say that with regard to non-relational persistence options there's a bloom of new growth with non-relational persistence: Examples include CouchDB and FeatherDB targeted towards particular content, data services

such as Amazon SimpleDB and the Google App Engine, as well as a renewed (or continued) interest in ODBMS, including new participants in the industry such as db4o.

I am reluctant to 'position' any of these choices other than to stress that they each have an appropriate scenario that highlights their strengths and provides the best use of the technology.

Question 3: What are in your opinion the pros and cons of these existing solutions?

[**Jose Blakeley**] For the last 30+ years, application developers have used conceptual models (e.g., ER, UML) in the early phases of the application development life cycle.
However, as development moves from design to code, the initial conceptual model and the implementation object model begin to diverge. This is because the industry has lacked a concrete runtime implementation of a conceptual model.
There has never been a runtime implementation of the E-R model. ORM technologies like Hibernate or the ADO.NET Entity Framework attempt to provide mapping runtimes that abstract the object-to-relational mapping problem away from applications. The ADO.NET EF takes this a step further by clearly providing a rich conceptual entity abstraction based on the EDM and an object abstraction for .NET objects. EntitySQL is a query language for EDM. LINQ is a query language for objects.

The pros of these solutions are increased programmers' productivity by enabling programmers to stay at the conceptual or object layers and not having to roll their own mapping layers. There is also a lower TCO due to more readable and maintainable code, since it doesn't contain hard-coded, non-compiler checked, implicit assumptions about the semantics of data within the model.

Consider an application code fragment to obtain the name, the id and the hire date for all salespersons who were hired prior to some date. There are four main shortcomings in this code fragment that have little to do with the business question that needs to be answered. First, even though the query can be stated in English very succinctly, the SQL statement is quite verbose and requires the developer to be aware of the normalized relational schema to formulate the multi-table join required to collect the appropriate columns from the SContacts, SEmployees, and SSalesPerson tables. Additionally, any change to the underlying database schemas will require corresponding changes in the code fragment below.
Second, the user has to define an explicit connection to the data source. Third, since the results returned are not strongly typed, any reference to non-existing columns names will be caught only after the query has executed. Since, the SQL statement is a string property to the Command API and any errors in its

formulation will only be caught at execution time.

While this code is written using ADO.NET 2.0, the code pattern and its shortcomings applies to any other relational data access API such as ODBC, JDBC, or OLE-DB.

```
void EmpsByDate(DateTime date) {
using( SqlConnection con =
  new SqlConnection (CONN_STRING) ) {
    con.Open();
    SqlCommand cmd = con.CreateCommand();
    cmd.CommandText = @"SELECT SalesPersonID, FirstName, HireDate
              FROM SSalesPersons sp INNER JOIN SEmployees e ON
sp.SalesPersonID = e.EmployeeID
              INNER JOIN SContacts c ON e.EmployeeID = c.ContactID
              WHERE e.HireDate < @date";
    cmd.Parameters.AddWithValue("@date",date);

    DbDataReader r = cmd.ExecuteReader();
    while(r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r.GetDateTime(0), r.GetString(1));
    }
  }
}
```

The equivalent program at the conceptual layer is written as follows:

```
void EmpsByDate (DateTime date) {
using( EntityConnection con =
  new EntityConnection (CONN_STRING) ) {
    con.Open();
    EntityCommand cmd = con.CreateCommand();
    cmd.CommandText = @"SELECT VALUE sp FROM ESalesPersons sp WHERE
sp.HireDate
< @date";
    cmd.Parameters.AddWithValue("date", date);
    DbDataReader r = cmd.ExecuteReader();
    while (r.Read()) {
      Console.WriteLine("{0:d}:\t{1}",
        r.GetDateTime(0), r.GetString(1));
    }
  }
}
```

The SQL statement has been considerably simplified—the user no longer has to know the precise database layout. Furthermore, the application logic can be isolated from changes to the underlying database schema. However, this fragment is still string-based, still does not get the benefits of programming language type-checking, and returns weakly typed results. By adding a thin object wrapper around entities and using the Language Integrated Query

(LINQ) extensions in C#, one can rewrite the equivalent function with no impedance mismatch as follows:

```
void EmpsByDate(DateTime date) {
  using (AdventureWorksDB aw = new AdventureWorksDB()) {
    var people = from p in aw.SalesPersons where p.HireDate < date select p;
    foreach (SalesPerson p in people) {
      Console.WriteLine("{0:d}\t{1}",
                  p.HireDate, p.FirstName );
    }
  }
}
```

The query is simple; the application is (largely) isolated from changes to the underlying database schema; and the query is fully type-checked by the C# compiler. In addition to queries, one can interact with objects and perform regular Create, Read, Update and Delete (CRUD) operations on the objects.

The cons are around the possible performance penalties introduced by the intermediate mapping layers. However, we expect over time performance penalties will be minimized and the improvements in productivity will offset the performance overhead.

[**Rick Cattell**] I think the panelists in *Part I* have already provided good arguments for the pros and cons of their favorites: http://www.odbms.org/experts.html#article16

[**William Cook**] My current viewpoint is academic, which means that I'm looking several years ahead rather than at current technologies. My impression (hope?) is that the Java community is finally rallying around a few solutions (for enterprise and embedded development) rather than just engaging of endless in-fighting over slightly different but incompatible approaches.

Real applications are quite complex. They tend to have dynamic security policies, which need to be compiled into SQL to execute efficiently. They use lots of dynamic SQL. They have feature modularity which requires entire parts of a query to be present in some configurations but absent in others. The tools we are using now work pretty well for medium-sized applications, but break down when applications grow larger and more complex.

[**Robert Greene**] Well, I've not really nailed them all down, but I will answer in general terms. In general, the easier something is to use, the less control you have to customize. At some point, you sacrifice ease of use, out of necessity, for more control as your requirements become more stringent.

The only time this does not hold true is when an aggregated set of abstractions have been sufficiently created to obviate an underlying complexity. Case in point, C++ -vs- Java in the early days. People would use C++ for more control, sacrificing Java's ease of use until Java rose to a sufficient level of aggregated

abstraction that more control in C++ no longer provided significant benefit.

So, the pros of something like Java serialization or JCR is that they are easy to use. The con's are that you have little control and find it difficult to tune for larger scale. As you move up the stack to ORM tools you have greater difficulty to use them, but much added control for things like performance tuning.

I see alternatives like object databases as an aggregated set of abstractions that obviate underlying complexity. They are so easy to use and yet have very powerful control and flexible capabilities. People always ask me, what about expertise though asserting, "I can find an ORM guy pretty easy, I just have to be willing to pay him well". Well I say, "show me a good OO Java developer and I'll show you a good object database guy".

[**Alan Santos**] As I noted above each of these solutions has a sweet spot where it is the right tool for the job. For example, nothing beats relational when it comes to data mining, ad-hoc reporting and the market for 3rd-party utilities. Nothing beats the right ODBMS when it comes to performance.

Problems arise when you try to fit the square block into the round hole.

Question 4: Do you believe that Object Relational Mappers are a suitable solution to the "object persistence" problem? If yes why? If not, why?

[**Jose Blakeley**] Yes, I do. See answer to Question #3.

[**Rick Cattell**] They are the best solution available if you've already chosen a relational database... with anything else you have to deal with the impedance mismatch problem yourself.

[**William Cook**] Yes, I think that there will always be a mapping later. These should be called "Object-Persistence Mappers" because they map objects to whatever kind of database is being used. One reason for the mapping later is that I don't believe in storing full objects in a database – methods and code should not be persisted with the data. As a result, it will always be necessary to wrap data in its behavior when loading it into a program.

[**Robert Greene**] I would have to defer to the above questions and say, I believe the right tool for the job is the answer. There isn't any black and white yes or no, the world is just not that simple. ORM's are very suitable and are working better than jdbc with hand coded persistence layers. You've got an existing database with relational data and you want to expose some new capability, it's easier to reverse engineer the relational schema and presto you're done. You've got an enterprise license with Oracle and 15 guys that know Hibernate and you don't need something that will scale to millions of users (not that it won't, but again there is that green effect).

However, there will come a point for every developer where they have to ask themselves, are my models sufficiently complex that it's better to just get rid of the Hibernate/JPA/JDO mapping file? Will I save enough time to market or will I save enough money managing my production systems if I just get rid of the ORM layer and use an OODB? That's a grey scale because the point at which one person will ask that question is always different from another. Are they the kind of person who likes to stick with what they know, have a fear of the unknown, only use a new technology once everyone else is (late adopter) or are they the kind of person who embraces change, is interested in best of breed solutions, likes to be the early bird and get the worm (early adopter).

So, suitable, what is suitable? I can build anything with today's technology. I say the smart people will realize that it's not about what's suitable, it's about what's best. With "suitable" it takes 6 months and runs on 100 servers while with "best" it takes 3 months and runs on 10 servers. It's the green effect at work.

[**Alan Santos**] For some, yes. For many others relational mappers are not. As I've noted above there may be non-technical issues, there may be other technical issues that trump performance, it may be that the standards of the IT organization dictate nothing else; in these cases then relational mapping technologies are appropriate.

In other cases object relational mappers are not a suitable solution: performance critical applications, lightweight stand alone or embedded applications that require persistence, applications whose access patterns may disrupt an enterprise data source and can be used with a distributed cache. For those cases an ODBMS is a better solution.

Ultimately the answer to this question is yes, they are a suitable solution when business dictates, but many people don't realize that there a slew of situations where mapping technologies are not the correct answer.

Question 5: Do you believe that Relational Database systems are a suitable solution to the "object persistence" problem? If yes why? If not, why?

[**Jose Blakeley**] Relational systems are the de facto standard for data persistence. Over time, the data model supported by commercial relational systems will be enhanced to support richer conceptual models like EDM. Also, the query language of relational systems will evolve to something closer to EntitySQL.

[**Rick Cattell**] No, they are not a good solution, because of the impedance mismatch problem. But as I mentioned, many programmers don't get to

choose the database system, and there are other constraints involved in the choice.

[**William Cook**] From the database side, I think of SQL as assembly language. It is like the pre-RISC assembly languages, which were designed to be human-readable. But the trend is that most SQL will be computer-generated. The age-old wisdom that you have to hand-optimize SQL and write stored procedures is no longer true, except for very rare cases. What needs to happen now is to analyze the common communication patterns between a program and a database, and evolve SQL to be a better target language for automatically generated queries. This is what RISC did for the compiler/architecture boundary. One example is returning more complex structures than a simple relation. The fact that queries can only return tables has been a problem for years, and it's holding back progress. The high-level query generators have to do back-flips to load multiple related datasets efficiently. This has to change.

[**Robert Greene**] I think the answer to this question is the same as above. We live in a world of grey. There are situations where I, an object database guru, would recommend a relational database. Yes they are suitable, but not always the right tool for the job. And in a reality check, as much as it would thrill me to see more innovators and early adopters, there are lots of people who fear the unknown. So, the line where one chooses between suitable in lieu of best of breed will undoubtedly hold closer to suitable than many of us would view reasonable.

[**Alan Santos**] I believe this question has been answered above.

Question 6: Do you believe that Object Database systems are a suitable solution to the "object persistence" problem? If yes why? If not, why?

[**Jose Blakeley**] If an Object Database System not only supports a richer object model, but it also supports set-oriented, scalable, cost-based-optimized query processing, and high-throughput transactions, then I think they could be a suitable solution to the object persistence problem. OODBMSs never delivered on all these dimensions. Relational systems are rapidly becoming object database systems.

[**Rick Cattell**] Yes; object databases were designed specifically to solve the object persistence problem.

[**William Cook**] I am not a relational purist or object-oriented purist. When I look at OQL, I don't seem much that makes it "object-oriented". It assumes a data model in which records are explicitly linked together, while in a relational model the links are implicit in the foreign keys. But almost all relational databases are designed using Entity-Relationship modeling, which was later adopted by UML and renamed "class diagrams". OQL allows construction of

arbitrary nested record collections as a query output, while the relational model historically requires a single set of records. Yet relational databases can return multiple record sets, or support other data shaping output. The XML extensions that have been made to many RDBMS give them functionality that is much closer to OQL. Perhaps that makes them "object-oriented"?

I'm not a database implementer, but I suspect that OODBMS and RDBMS begin to look quite similar under the covers, once you implement the key database features of query optimization, caching, and ACID transactions. There isn't that much difference between OQL and SQL either. Perhaps the old distinctions are not the ones that matter?

[**Robert Greene**] Again, no black and white answers. Yes, they are suitable, but are they _always_ the right tool for the job, no. As stated above, where they make more sense than a relational database will vary on many factors and person to person, late to early adopter. I will say that in general, as things become more complex and involve more data, object databases become a more viable option.

One of the main issues here, is that a lot of the worlds systems are built on relational technology and those systems need to be extended and integrated. That job is always difficult and using an object database complicates that task if not by any other way than complicating things at the conceptual level. Reality is that solutions now exist for Java where both systems can be used together. Versant has solutions which will use POJO's in both object and relational databases. So, those can be easily and seamlessly integrated in newer systems. As the world moves more toward architectures like SOA, where resources are relatively implementation independent, it will be easier to make choices for reasons like "the green effect".

Also, it is important to point out that unlike the general case for relational databases, object databases vary greatly from vendor to vendor. That has impact on other ancillary issues of which there are too many to enumerate here, just like there are so many factors that go into choosing an object persistence solution. Some OODB's are great at high volume updates and lots of concurrent transactions, some are great at high volumes of mostly read operations, some support less than 100M in data reliably, others 100's of gigabytes, while others 10's of terabytes and yet others petabytes. Bottom line, you need to use the right tool for the job, even within the category of object databases.

[**Alan Santos**] Of course. I've highlighted several scenarios where object databases are not simply suitable, they are the correct choice as well as situations where they are **not** suitable.

One less obvious scenario where an ODBMS offers value worth considering in a little more detail is that of a persistent cache and the benefits offered as such. Besides the obvious performance improvements, there is the potential for a

great degree of flexibility, data augmentation, data aggregation from multiple heterogeneous sources, semantic integration and life cycle management when used in conjunction with change data capture and semantic integration.

More than one respondent in the previous discussion noted that object databases are less suitable when data needs to be shared with an application that is unaware of the ODBMS or require ad-hoc query capabilities. With all due respect, those are **product** limitations as opposed to technology limitations.

An ODBMS should be able to fully participate in the enterprise data ecosystem as well as any other DBMS for both new development as well as enhancing existing applications.

Question 7: What would you wish as a new research/development in the area of Object Persistence in the next 12 months?

[**Jose Blakeley**] Perhaps 12 months is too short a time frame. Over the next 3-5 years, I would like to see technologies like the EDM, EntitySQL, and EF be absorbed natively by relational database systems. I would also like to see innovations like LINQ become part of the next-generations stored procedure languages supported natively by database systems.

[**Rick Cattell**] I'm disappointed with the slow progress on object persistence APIs for Java over the years. Politics impeded things: in addition to personal biases, particular vendors had strong interests in particular technologies as part of the solution. To pick two particular examples, the OMG Persistence Service and EJB 2.0 Persistence were strongly influenced by IBM and Oracle, and both were a disaster for the programmer (IMHO).
JDO was great, but had opposition from the big vendors. JPA has been the best political compromise that is also reasonable for the programmer, but it still lacks some crucial features, e.g. better supporting non-relational object persistence. So, I would say that JPA or its successor is where I'd most like to see new R&D.

[**William Cook**] I would like to see the major database vendors implement OQL (or some variant, like HQL) as a native database interface to their relational databases. Sure, I hear the screams of anger as they tell me this violates the relational dogma handed down by the founding DBAs. But I don't care. It would be much easier to store and load objects if the output format of the database was more flexible. The result of a query should be a complete mini-database, not a particular component of a database.

I would also like to see Microsoft experiment with other ways to deal with "LoadWith", like AutoFetch or Query Extraction. (Disclaimer: this is plug for two of my students' research projects; I shouldn't have to apologize for that.) Maybe they can come up with something better. But they need to try, because

the current solution is not going to scale to larger programs.

Finally, I think that the Java community needs to figure out how they are going to respond to Linq. They could copy Microsoft, just like they copied MTS when creating EJB. Or they could organize an effort to come up with something better. Linq is not the full story, so there is an opportunity here.

[**Robert Greene**] I think much of the needed research is already being done. It's absolutely fabulous that people are finding innovative ways to get beyond the jdbc abstraction for Java object persistence. A research issue, close to object persistence, but equally applicable in other layers of application architecture, is model to model mapping. Using object databases, this area is impacted by schema evolution and versioning. This is another kind of impedance mismatch that exists and should be dealt with in research and in practice.

The mismatch is manifested via the way we want to represent the presentation tier, different from the business logic, different in varied work flow contexts, and ultimately different in various data resources. This will become even more of and issue if things like domain specific languages begin to take hold on our programming practices. You have XSLT for XML, but having to move your language objects to XML to get a decent transformation technology will not suffice.

Research aside, I would like to see real world case studies of OODBs. As an object database provider that is a public company, it is very clear that we are making lots of money, which means people are buy licenses and using our technology. However, when is the last time we produced a customer case study or press release? Adopters of object database technology continually claim it as a competitive edge, strategic to their success, and avoid the idea of telling the world how it's working for them. This undermines the adoption of the technology and puts it at risk for the very people who embrace its value. Anyone who wants to build a simple, small scale application can easily use an ORM tool and relational database. The only reason we exist, is because we can, and do, run some of the most demanding systems in the world. I would like to see our users step up and profess their success in using our technology.

[**Alan Santos**] Intelligent data distribution; structural and semantic integration; self optimizing applications, both in terms of the physical clustering of persistent data as well as simple application level tuning; scaling efficiencies and performance across machines and up into very large sizes.

Question 8: If you were all powerful and could have influenced technology adoption in the last 10 years, what would today's typical project use as a persistence mechanism and why?

[**Jose Blakeley**] It took 3-4 years to develop the EDM, EntitySQL, EF runtime, and LINQ. Now that they have shipped as part of Visual Studio 2008, I would like to see these technologies be adopted as the persistence abstractions for new projects. I would like to collect feedback and experiences from real applications and improve all these new technologies and surrounding developer tools to improve programmers' productivity even more.

[**Rick Cattell**] JDO!

[**William Cook**] I have been following the steady stream of database APIs for over 10 years. ODBC, DAO, RDO, OLE DB, MTS, JDBC, ADO, ADO.NET, JDBC, EJB 1, JDO, EJB 2, Hibernate, Linq, SDO, JPA. Not to mention the dozens (hundreds) or other open source and homebew solutions. I don't expect this to stop anytime soon. This is a really hard problem, and most of the solutions are partial or incomplete in some significant way.

However, I think that just about every program written should be using a clean persistence model for stored data and a database to manage it. The fact that people find it easier to just throw stuff into a file is a sad commentary on the level of progress we have made.

[**Robert Greene**] If I were all powerful, the typical project would use exactly what I want it to use, call me the Necromancer ;-)

Stated a different way, if I had substantial influence over technology adoption over the last 10 years, I think it would not look too different than it does today, though the ratios would change. I mean, more than 10 years ago relational databases were pervasive and yet still a lot of data was managed in the older network databases, ISAM files, mainframes, etc. As the old saying goes, where data falls, data stays. It is still true today that many systems use the mainframe as the system of record. So, all this stuff would still be here. However, if I had substantial influence, rather than a 95/5 relational/object split, at this point in the adoption curve I would like to imagine it as something closer to 50/50 ;-) There are still a lot of simple systems out there.

[**Alan Santos**] I would have attempted to resolve the issue that ODBMS products have historically been dependent upon new software development or the failures of RDBMS solutions before they're pursued as a solution for persistence. Looking to the future, I believe that this is just now starting to be addressed.

Question 9: Any parting words about this topic?

[**Jose Blakeley**] The incorporation of set-oriented, declarative expressions natively in programming languages will have a tremendous impact in developer's productivity and in exploiting new hardware trends such as increasing RAM and multi-cores by bringing database technologies such as query optimization to mainstream programming. I would like to thank you for inviting me to be part of this forum.

[**Rick Cattell**] Thanks for organizing these panels on ODBMS.ORG. You're doing a great service, providing a forum for communication on the alternatives and products.

[**William Cook**] I think that common themes are emerging from all this effort. It's amazing how long they are taking to emerge. But I think that object-oriented and relational databases should be merged to take the best ideas from both, rather than being viewed as competitive. A similar process is happening in the research work on object-oriented programming and functional programming. We need to get back to the original idea of aspects: user interface, security, data models, workflow, and evolution; these are aspects of a system we need to focus on integrating cleanly.

I also think that we are getting to the point where we need a new paradigm. The old rivalries don't necessarily make sense any more. I am working on model-driven development and trying to help it become a full programming paradigm. It has a strong foundation in both data and process, so it may eventual integrate "methods" and "data" in a new way. If this works out, then by the time we get the object-oriented database impedance mismatched resolved, it will be irrelevant. We live in interesting times. I hope that you have found these comments useful, and perhaps provocative. That is, after all, the purpose of this kind of panel.

[**Robert Greene**] We would all be a lot better off if we just use the right tool for the job. You wouldn't use a sledge hammer to drive a penny nail. Conversely, you wouldn't use a pocket knife to cut down a tree. Understand what you are trying to build and choose the simplest solution that will not tie your hands nor steal your profits. Don't be afraid to try the newer tools, you may find they save you lots of work.

[**Alan Santos**] It's great to see knowledgeable discussions like this which further the awareness of data persistence, bring new ideas into bloom and present alternative viewpoints to light.

Finally, it's even more exciting to see the growing acceptance of non-relational persistence. There are many possible reasons for this: the realization that an RDBMS is not always the right choice given size, scope and performance growth on the high and low end; the proliferation and distribution of ever growing quantities of data; the growth of "alternative" languages; changing

requirements within industry; or more likely all of those in combination with many more that I haven't considered. Whatever the reason, the end result should be better products that solve more problems in a more efficient and complete manner than the industry provides today.

Thanks for the invitation to participate.

##