acmqueue

All Your Database Are Belong to Us

In the big open world of the cloud, highly available distributed objects will rule.

Erik Meijer, Microsoft

In the database world, the raw physical data model is at the center of the universe, and queries freely assume intimate details of the data representation (indexes, statistics, metadata). This closed-world assumption and the resulting lack of abstraction have the pleasant effect of allowing the data to outlive the application. On the other hand, this makes it hard to evolve the underlying model independently from the queries over the model.

As the move to the cloud puts pressure on the closed-world assumption of the database, exposing naked data and relying on declarative magic becomes a liability rather than an asset. In the cloud, the roles are reversed, and objects should hide their private data representation, exposing it only via well-defined behavioral interfaces.

The programming-language world has always embraced such an open-world assumption by putting behavior at the center of the universe and imperatively building reliable applications from unreliable parts. Object databases are back with a vengeance. To help developers transition to the cloud, the existing class libraries and tool infrastructure need to evolve to run as highly available services exposed using regular object-oriented programming language interfaces that reflect the relevant operational details.

MODELERS: FIX THE REPRESENTATION, VARY THE BEHAVIOR

Database developers (let's call them *modelers*) and application developers (let's call them *programmers*) have dual views of the world. If you are a modeler, then everything revolves around data—and data about that data (metadata). The world of database models is *noun*-based, talking about `Customers`, `Orders`, `LineItems`, etc. Once modelers have designed the data model correctly, they consider their job done.

In the realm of modelers, there is no notion of data abstraction that separates abstract properties of the model from the concrete details of the fully normalized realization in terms of tables with PK/FK (primary-key/foreign-key) relationships.

```
DECLARE Customers TABLE
   (ID int PRIMARY KEY, …)
DECLARE Orders TABLE
   (ID int PRIMARY KEY, CID int REFERENCES Customers(ID), …)
DECLARE LineItems TABLE
   (ID int PRIMARY KEY, OID int REFERENCES Orders(ID), …)
```

The outside-in aspect of PK/FK relationships actually amplifies the need to expose all the details of the underlying rows and tables. For example, you cannot pick up one specific customer row from the `Customers` table and ask it what its list of orders is. Instead you must have access to and understand

the schemas of both the `Customers` and `Orders` tables to perform a join across *all* the order rows in the `Orders` table and all the rows in the `Customers` table to check if the foreign key in the order matches the primary key of the customer, and then filter by your target customer.

Modelers consider this lack of data hiding an advantage since it allows for ad-hoc queries on the data that may not have been thought of before, such as joining all customers with all products whose price matches their age. Imposing no data abstraction arguably has the advantage that the data can outlive the application.

(Databases do support a limited type of data abstraction in the form of views, stored procedures, and table-valued functions. These are more like .NET extension methods, however, in that they operate on the public data and do not enforce data hiding. Moreover, this complicates the conceptual model, because these additional concepts are not rooted in the underlying mathematical foundations of relational algebra.)

This noun-centric view of the world carries over to how modelers look at computation. For them, computation recursively is yet another model, a plan that can be inspected, transformed and optimized, and ultimately interpreted by a query engine. Often a query execution engine uses constant space—that is, it interprets a fixed static plan without needing a call stack or using term rewriting to execute. Even the context of a computation is considered to be data via the system catalog that maintains metadata about other data such as the tables in a database and their properties.

The ability to treat execution plans as data to inspect and optimize before execution is extremely powerful, but naturally leads to a first-order, non-recursive computational model with a limited set of operators. Modelers also love to draw pictures of their plans, which is a great debugging tool, but gets hard when dealing with higher-order constructs (such as first-class functions or nested structures), complex control-flow, or arbitrary recursion. Recursion is allowed, in the form of so-called common-table expressions that express linear recursion of the form `X=B∪F(X)`. This corresponds to a simple repetition and can still be drawn as a picture with a loop.

Database queries are declarative. For example, a typical SQL query such as the following, which computes the total price of all products by category for a customer, semantically defines a triple nested loop that iterates over each customer, order, and line item. Modelers, however, trust that the query optimizer will find a more efficient execution plan[2] that exploits indexes and other advanced techniques to make it run fast.

```
SELECT Customer, LineItem.Category, Sum(LineItem.Price)
FROM Customers, Orders, LineItems
WHERE Customers.ID = Orders.CID AND Orders.ID = LineItems.OID
GROUP BY LineItem.Category
```

This requires a perfect closed world where the optimizer can reason across all tables used by the query, and where modelers can be shielded from explicitly handling latency, exceptions, or other low-level operational concerns. These assumptions break down when tables get so big that they no longer fit on a single machine, or when you try to join two tables that live in different "worlds" or administrative domains, or do a distributed transaction across a slow and unreliable network. Queries suddenly need to become partition-aware, defeating many of the advantages of a declarative query language that hides the "how" from the "what."

The inability to naturally perform arbitrary computations, the difficulties of reasoning about the operational behavior of code, and the lack of data abstraction arguably disqualify conventional relational database technology as an operational model for cloud programming. The stunningly beautiful mathematical abstractions of Ted Codd served us well at the micro level where the database can control everything, but they fall apart when we move to the macro level of the cloud. Just as in physics where you need to trade in classical mechanics for quantum mechanics when you move to the subparticle level, in the computer science world you need to trade in nouns for verbs when you move beyond the single-machine level.

The problem with SQL databases is not the data model or the superb query processing functionality, it's the assumption that all the data lives in the same place and meets a bunch of consistency constraints, which is hard to maintain in an open distributed world. The details that are relevant to build a working system have shifted, which means the level of abstraction has to evolve likewise. In the local context of a closed world, however, the expressiveness, efficiency, and reliability of an RDMS (relational database management system) remain hard to beat.

PROGRAMMERS: FIX THE BEHAVIOR, VARY THE REPRESENTATION

Concepts such as "behavior," "imperative," "side effects," "arbitrary nesting," "higher-order functions," or "unbounded recursion" are the bread and butter for programmers, whose world revolves around computation, as opposed to data. For programmers it is all about *verbs* such as `DeleteFiles`, `OnDragDrop`, etc., and enforcement of strong data abstraction by strictly separating the implementation and interface of their objects.

Taking a page from Wikipedia (http://en.wikipedia.org/wiki/Object-oriented_programming), "Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified."

Take for example the `using` statement in C#, which obtains a disposable resource, executes a side-effecting statement using that resource, and then disposes of the resource via the following syntax:

```
using (var r = e) s
```

A resource is *any* object that implements the `IDisposable` interface, no further questions asked. The `IDisposable` interface has a single parameterless `Dispose()` method that releases the resource:

```
interface IDisposable{ void Dispose(); }
```

Because the compiler assumes only that the resource implements the `IDisposable` interface but does not care about the concrete type of the resource, it can blindly desugar the `using` statement into the following underlying primitive sequence of an assignment and a `try-catch` block:

```
var r = e; try s finally (r as IDisposable).Dispose();
```

The actual implementation and type of the resource are kept strictly private and are not exposed to the `using` statement. This allows the same syntax to be used over a range of objects such as `TextReader` and `IEnumerable` that follow the same general pattern of create/use/delete.

The method implementations of an interface or class can use arbitrary imperative code. In fact, because each method takes an implicit `this` parameter that contains the method itself, methods are inherently recursive,[1] and code is executed using a runtime call stack that dynamically keeps track of recursive calls as the computation unfolds into a potentially infinite call tree.

Compilers try to optimize the static-code representation or trace the runtime execution paths before optimizing the resulting straight-line code. Developers expect only constant time improvements, however, and generally do not like the structure of their computation to be rearranged dramatically and unpredictably by the optimizer, because the correctness of a program may critically rely on the order of evaluation due to side effects. Moreover, programmers demand that the debugger be able to map the optimized code back to the actual source they wrote such that they can inspect the intermediate values in the computation to resolve bugs or problems.

The ability to specify a behavioral contract between specification and implementation, along with the fact that method implementations can invoke arbitrary code, allows applications to outlive data. This is especially important when leveraging disruptive technological advances (such as the cloud) to improve the implementation of a given interface, with minimal disruption for the programmer and application.

EVOLVING COLLECTIONS

As a concrete example of data abstraction, consider the .NET standard collections library, `System.Collections.Generic`, or similarly, the Java collections library and C++ STL containers. These libraries were implemented when machines were single-core and most programs were single-threaded. Hence, the logical tradeoff was not to make collections thread-safe by default. For example, in the .NET generic collections library, the class `Dictionary<K,V>` implements three (standard) collection interfaces:

```
class Dictionary<K,V> : IDictionary<K,V>,
   ICollection<KeyValuePair<K,V>>, IEnumerable<KeyValuePair<K,V>>
```

Programmers do not really care how the dictionary class is implemented, as long as it satisfies the contract defined by its base interfaces. As a bonus, because `Dictionary<K,V>` implements `IEnumerable<KeyValuePair<K,V>>`, you can automatically use LINQ to Objects to query over dictionaries viewed as collections of key-value pairs.

(Since the LINQ to Objects implementation is defined in terms of the `IEnumerable` interface, it cannot use specific features of the dictionary implementation to improve efficiency, which is what modelers would do. In practice, however, the implementation of LINQ to Objects "cheats" by checking for known concrete collection types and dynamically dispatches to a type-specific implementation.)

With the advent of many-core processors, concurrency has become commonplace, and, hence,

it makes sense to make collections thread-safe by default. Also, we would like to implement LINQ to Objects to maximize the parallelism inherent in the LINQ "queries" via PLINQ (Parallel LINQ). To leverage many-core machines, the .NET Framework 4.0 introduced a new implementation of collections in the `System.Collections.Concurrent` namespace (http://msdn.microsoft.com/en-us/library/system.collections.concurrent.aspx). And guess which interfaces the new thread-safe class `ConcurrentDictionary<K,V>` implements? Precisely the same as those of the old `Dictionary<K,V>` class:

```
class ConcurrentDictionary<K,V> : IDictionary<K,V>,
   ICollection<KeyValuePair<K,V>>, IEnumerable<KeyValuePair<K,V>>
```

Java has seen exactly the same evolution, where `HashMap<K,V>` and `ConcurrentHashMap<K,V>` both implement the unchanged base interface `Map<K,V>`.

This is data abstraction in its purest form—the same interface with a new and improved implementation. In this particular situation, the new implementation for collections could partition the dictionary into independent shards that can be accessed and updated concurrently, a relevant implementation detail that therefore should be exposed explicitly to the programmer in the .NET framework via the `Partitioner<T>` and `OrderablePartitioner<T>` abstract base classes.

(Note that just as the regular LINQ to Objects implementation cheats by downcasting to the concrete implementations of `IEnumerable`, the PLINQ implementation applies the same cruel and unnatural act with the new concurrent collections.)

DATA STRUCTURES AS A SERVICE

As we move from the many-core world to the massively parallel and distributed world of the cloud, the next logical step[3] in the evolution of the `System.Collections` namespace is to add support for collections as a service by introducing new types such as `DistributedDictionary<K,V>` that use the familiar collection interfaces, but are implemented as highly available scalable services running on some underlying cloud fabric such as Windows Azure, Amazon EC2 (Elastic Compute Cloud), or Google AppEngine:

```
class DistributedDictionary<K,V> : IDictionary<K,V>,
   IObservable<KeyValuePair<K,V>>
```

Since the constraints of a distributed system such as the cloud with regard to latency and error conditions are different from the constraints of running objects locally on a single- or many-core machine, support for asynchronous operations must be added by modifying the interfaces to use the new `Task`-based asynchronous pattern supported by `await` in the latest versions of C# and Visual Basic. For developers this is a small and nondisruptive evolution of familiar interfaces to reflect essential operational changes in the underlying runtime.

Collections as a service are immensely useful. Imagine you want to build a Web-based game that maintains a high score among millions of players of that game (every aspiring game developer dreams of being as popular as Mafia Wars or Farmville). Assume that each player maintains a list

instance of `System.Collections.Distributed.DistributedSortedList<int,Player>`, say in the variable `highScores`. You can then update and compare scores programmatically by mutating and invoking the list in code as follows (using C# 5.0 asynchronous methods):

```
var highScores = … get object instance connected to service …
await highScores.Add(100000, me);
var top10 = await highScores.TakeAsync(10);
```

Since the service is highly available, players can disconnect and reconnect to it at any moment. Since it is scalable, it can support millions of concurrent players. When developers dream of platform as a service, they envision a class library of services like this.

Note that we choose to expose the services in `System.Collections.Distributed` using regular programming language APIs instead of via a REST or another network-level interface. Just as we like to hide the state of an object behind an interface, the exact protocol that the client proxy of an object uses to communicate with the service that implements its behavior is an internal detail that should be kept private for the same reasons. In this context it is interesting to note that certain Web browsers apply the same principles to accessing Web pages by switching from human-readable HTTP to the more efficient SPDY protocol. Most modern services, such as Twilio and the distributed event notification service of Google's Thialfi, prefer language-specific APIs over protocol-level access. As explained by the Google Thialfi team, "Initially, we had no client library whatsoever, opting instead to expose our protocol directly. Engineers, however, strongly prefer to develop against native-language APIs. And, a high-level API has allowed us to evolve our client-server protocol without modifying application code."

The idea of `System.Collections.Distributed` is not science fiction. A concrete example of a distributed "drop-in" reimplementation of the Java standard collection classes is KeptCollections (https://github.com/anthonyu/KeptCollections), which uses Apache ZooKeeper as the backing store to provide a distributed and scalable implementation of the `Map<K,V>` interface. The Redis project announces itself as a "data structure server" and is another realization of the idea of providing distributed collections such as tables, lists, and sets as services. Services such as MemcacheDB and Windows Azure Caching and Service Bus are other examples of problem-focused services that implement standard collection classes.

Switching back to modelers for a second, Dynamo[9]-based noSQL databases such as Riak appeal to the CAP theorem to trade *consistency* (which modelers are used to) for *availability* (which certain applications such as shopping carts require) as they try to create databases that can cope with the open world of the cloud. The consequence of dropping consistency is that it is now up to the application to resolve conflicts, which deeply worries many modelers. Programmers, however, are already familiar with nonconsistent and nonavailable systems. Every time you call a method on an object, it may throw an exception and leave the hidden state of the object in an undefined state. The very fact that you are able to read this article on a Web site is testament to the fact that programmers know pretty well how to build highly scalable, practical, distributed systems out of unreliable components, using neither transactions nor consistency nor availability. They design their systems to be robust with regard to inconsistency to start with.

An underexposed consequence of choosing availability over consistency is that queries have to

deal with version conflicts. For example, since a key may contain multiple versions of a value, the map function in a MapReduce query must resolve the conflict (without being able to write back the choice) as it is applied to each key. Most programmers probably prefer to drop availability rather than having to solve versioning conflicts.

The research community has been studying so-called CRDTs (convergent or commutative replicated data types[8])—distributed data types that satisfy algebraic properties guaranteeing eventual consistency under asynchronous replication. These data structures strike a nice balance between programmer convenience and scalability of implementation. Also interesting and relevant in this area is the work on concurrent revisions[7] and older work such as reversible computations in the Time Warp operating system.[4]

CONCURRENCY AS A SERVICE

The same analogy can be drawn for threading and concurrency as for collections. On single-core machines, programmers dealt with (simulated) concurrency using the `System.Threading.Thread` class. For many-core machines the model was extended to `System.Threading.Tasks`. In this case threads and tasks do not implement a common interface (although see Java's Executor framework and Rx schedulers), but conceptually they are very similar. Both threads and tasks are constructed by passing an action delegate to the respective constructors

```
var thread = new Thread(delegate{ … });
var task = new Task(delegate{ … });
```

and then are started by calling their `Start()` method. The task API is much richer than the original threading API and, for example, allows comonadic composition of existing tasks into new tasks using the `ContinueWith` method. Yet both threads and tasks embody the data-hiding maxim of providing a coherent set of behaviors that represent an abstract notion of concurrency without revealing the underlying implementation details, such as the thread pool using a work-stealing scheduling algorithm under the covers.

The cloud does not just have a pool of threads, but a gigantic ocean of machines. We would like to expose all that abundant potential concurrency as a service that programmers can tap into. Besides simple actions, we also want to enable long-running and highly available computations in the cloud-pool that would survive fault-domain failures, as well as cluster restarts. Carl Hewitt's Actors are the prevalent model for concurrency at cloud-scale, as evidenced by languages and libraries such as Rx, Dart, Erlang, Akka, F#, Axum, and Ciel. In contrast to more static models for distributed computation such as MapReduce or Pregel that are favored by modelers, Actors allow programmers to express fully recursive and dynamic distributed computations. Observant readers will have correctly realized that distributed Actors are a natural way to implement the data structures as a service, as well as objects with a fixed behavior.

Operationally, programmers need to deal with four basic forms of dual computational effects:

1. The built-in effect of blocking computations (in a pure language such as Haskell this would surface as type `IO<T>`) that synchronously return a single value.
2. Blocking or pull-based collections of type `IEnumerable<T>` that synchronously return multiple values.

3. Nonblocking computations of type `Task<T>` or `Future<T>` that asynchronously return a single value.
4. Streaming or push-based collections of type `IObservable<T>` that asynchronously return multiple values.

The various concurrency primitives such as Threads or Actors and regular method calls produce their results using some particular combination of these four basic effects (see table 1).

SERVICES AS A SERVICE

When developing code, programmers rely not only on programming languages and libraries, but also on various tools such as debuggers and profilers. When talking about moving existing programming idioms to the cloud, we tend to sweep a lot of details under the rug that are necessary for building enterprise-grade services, such as monitoring, tracing, distributed locking, fault detection, replication, and fail-over. Interestingly, many of these can be exposed as objects implemented by a service. Implementing the building blocks for debugging, understanding, and monitoring distributed systems is no panacea, but it is a necessary condition to make programmers successful in the new world of distributed applications.

Other aspects that become relevant when composing larger services from smaller services are colocation and efficient service resolution. Even on a single Windows machine, it is necessary to reason about thread affinity—for example, in order to update the UI, you must run on the UI thread. A practical substrate for building distributed applications must provide abstractions that make location and placement boundaries explicit—and perhaps provide local atomicity or reliability guarantees. Within the confines of a single machine, programmers often gloss over such details (just as modelers do), but in a distributed world, exposing them becomes relevant and important to acknowledge for both programmers and modelers.

ALL DATA IS A MONAD

Even though we have argued that models are not the right *operational* basis for cloud computing, let there be absolutely no mistake that we do acknowledge the enormous economic and intellectual value of (relational) database technology, and programmers are eager to use the data that modelers have curated, cleansed, normalized, and refined.

Fortunately, models can be embedded into programs by appealing to a generalization of Codd's theory[6]—namely, Category Theory. As it turns out, most of the notions of "collections" or "tables" in various database implementations are actually instances of a mathematical concept, really just a kind of interface, called *monads*. Queries can be translated into the underlying monadic operators implemented as code.

Programmers call this LINQ[5] instead of monads, and there are LINQ bindings for countless

TABLE 1 **Computational effects**

Effects	Single result	Multiple results
Pull, synchronous, blocking	T	IEnumerable<T>
Push, asynchronous, nonblocking	Task<T>/Future<T>	IObservable<T>

databases such as SQL, Azure Tables, MongoDB, CouchDB, Hadoop, and HBase that expose models to developers, by embedding them as push or pull collections using a common interface.

THE REVENGE OF OBJECT DATABASES

Relational databases are like the ancient Chinese south-pointing chariots, amazingly elaborate and intricate mechanisms and marvels of engineering. They can be sold for lots of money and provide many opportunities for additional features and upselling. The chariot works great when driving on smoothly paved roads, but quickly sinks when used to navigate the ocean. Objects are like the compass: cheap, simple, and versatile, and they do survive the transition toward the cloud. The cloud era is one of monadic and comonadic computation and verbs, as opposed to data and nouns.

The object-oriented databases from the 1980s tried too much to be like databases and too little like objects, and they did not play to their strengths. In the small closed world of tables, declarative queries, and sophisticated optimizers,[2] it is hard to beat RDMSs at their own game. In the big open world of the cloud, however, highly available distributed objects/actors will rule. To ease the transition to the cloud for programmers, we have identified the following requisites:

- Expose every data source created by modelers to programmers using monads/LINQ.
- Create a class library of distributed collections implemented as highly available and scalable services but exposed using standard programming language bindings and interfaces.
- Give programmers access to the ocean of concurrency in the cloud via comonads/actors.
- Expose tracing, monitoring, debugging, and diagnostic infrastructure as another service in the cloud.

Many of the required materials are available today. What is missing is a tasteful assembly of all these pieces into a set of elegant and functional packages that target the challenges developers face when migrating into the cloud.

ACKNOWLEDGMENTS

I would like to thank Brian Beckman, Terry Coatta, Gavin Bierman, Joe Hoag, Brian Grunkemeyer, and Rafael Fernandez Moctezuma for improving both the style and substance of this article.

REFERENCES

1. Abadi, M., Cardelli, L. 1997. Theory of objects. ECOOP (European Conference on Object-oriented Programming) Tutorial; http://lucacardelli.name/Talks/1997-06%20A%20Theory%20of%20Object%20(ECOOP%20Tutorial).pdf.
2. DeWitt, D. J. 2010. SQL query optimization: why is it so hard to get right? Keynote address at PASS Summit; http://www.slideshare.net/GraySystemsLab/pass-summit-2010-keynote-david-dewitt.
3. Gribble, S. D., Brewer, E. A., Hellerstein, J. M., Culler, D. 2000. Scalable, distributed data structures for Internet service construction. Proceedings of 4th Usenix Symposium on Operating Systems Design and Implementation; http://static.usenix.org/event/osdi00/full_papers/gribble/gribble_html/dds.html.
4. Jefferson, D., et al. 1988. The status of the Time Warp operating system. *Proceedings of the third conference on Hypercube concurrent computers and applications: Architecture, software, computer systems, and general issues* 1(738-744); http://users.ecs.soton.ac.uk/rjw1/misc/VirtualTime/p738-jefferson.pdf.

5. Meijer, E. 2011. The world according to LINQ. *ACM Queue* 9(8); http://queue.acm.org/detail.cfm?id=2024658.
6. Meijer, E., Bierman, G. 2011. A co-relational model of data for large shared data banks. *ACM Queue* 9(3); http://queue.acm.org/detail.cfm?id=1961297.
7. Microsoft Research. Concurrent revisions; http://research.microsoft.com/en-us/projects/revisions/default.aspx
8. Shapiro, M., Preguiça, N., Baquero, C., Zawirski, M. 2011. A comprehensive study of convergent and commutative replicated data types. Inria No. RR-7506; http://hal.inria.fr/inria-00555588/en/.
9. Vogels, W. 2007. Amazon's Dynamo. All Things Distributed; http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html
10. Helland, P. 2005. Data on the Outside Versus Data on the Inside (144-153); http://www.cidrdb.org/cidr2005/papers/P12.pdf

LOVE IT, HATE IT? LET US KNOW
feedback@queue.acm.org

ERIK MEIJER (hjmmeijer@tudelft.com emeijer@microsoft.com) has been working on "democratizing the cloud" for the past 15 years. He is perhaps best known for his work on the Haskell language and his contributions to LINQ and Rx (Reactive Framework). He is a part-time professor of cloud programming at TUDelft and runs the cloud programmability team in Microsoft's Server and Tools Business.