acmqueue

A co-Relational Model of Data for Large Shared Data Banks

Erik Meijer and Gavin Bierman, Microsoft Research

Contrary to popular belief, SQL and noSQL are really just two sides of the same coin.

Fueled by their promise to solve the problem of distilling valuable information and business insight from big data in a scalable and programmer-friendly way, noSQL databases have been one of the hottest topics in our field recently. With a plethora of open source and commercial offerings (Membase, CouchDB, Cassandra, MongoDB, Riak, Redis, Dynamo, BigTable, Hadoop, Hive, Pig, …) and a surrounding cacophony of technical terms (Paxos, CAP, Merkle trees, gossip, vector clocks, sloppy quorums, MapReduce, …), however, it is hard for businesses and practitioners to see the forest for the trees.

The current noSQL market satisfies the three characteristics of a monopolistically competitive market: the barriers to entry and exit are low; there are many small suppliers; and these suppliers produce technically heterogeneous, highly differentiated products. [12] Monopolistically competitive markets are inconsistent with the conditions for perfect competition. Hence in the long run monopolistically competitive firms will make zero economic profit.

In the early 1970s, the database world was in a similar sorry state.[14] An overabundance of database products exposed many low-level implementation details and, as database people like to say, forced programmers to work at the physical level instead of the logical level. The landscape changed radically when Ted Codd proposed a new data model and a structured query language (SQL) based on the mathematical concept of relations and foreign-/primary-key relationships.[4] In the relational model, data is stored in conceptually simple containers (tables of rows), and queries over this data are expressed declaratively without knowledge of the underlying physical storage organization.

Codd's relational model and SQL allowed implementations from different vendors to be (near) perfect substitutes, and hence provided the conditions for perfect competition. Standardizing on the relational model and SQL created a secondary network effect around complementary producers such as educators, tool vendors, consultants, etc., all targeting the same underlying mathematical principles. Differences between actual relational database implementations and SQL dialects became to a large extent irrelevant.[7]

Today, the relational database market is a classic example of an oligopoly. The market has a few large players (Oracle, IBM, Microsoft, MySQL), the barriers to entry are high, and all existing SQL-based relational database products are largely indistinguishable. Oligopolies can retain high profits in the long run; today the database industry is worth an estimated $32 billion and still growing in the double digits.

In this article we present a mathematical data model for the most common noSQL databases—namely, key/value relationships—and demonstrate that this data model is the mathematical dual of SQL's relational data model of foreign-/primary-key relationships. Following established mathematical nomenclature, we refer to the dual of SQL as *coSQL*. We also show how a single generalization of the relational algebra over sets—namely, monads and monad comprehensions—

forms the basis of a common query language for both SQL and noSQL. Despite common wisdom, SQL and coSQL are not diabolically opposed, but instead deeply connected via beautiful mathematical theory.

Just as Codd's discovery of relational algebra as a formal basis for SQL shifted the database industry from a monopolistically competitive market to an oligopoly and thus propelled a billion-dollar industry around SQL and foreign-/primary-key stores, we believe that our categorical data-model formalization model and monadic query language will allow the same economic growth to occur for coSQL key-value stores.

OBJECTS VERSUS TABLES

To set the scene let's look at a simple example of products with authors and recommendations as found in the Amazon SimpleDB samples, and implement it using both object graphs and relational tables.

While we don't often think of it this way, the RAM for storing object graphs is actually a key-value store where keys are addresses (l-values) and values are the data stored at some address in memory (r-values). Languages such as C# and Java make no distinction between r-values and l-values, unlike C or C++, where the distinction is explicit. In C, the pointer dereference operator `*p` retrieves the value stored at address `p` in the implicit global store. In the rest of this article we conveniently confuse the words object (graph) and key-value store.

In C# (or any other modern object-oriented language) we can model products using the following class declaration, which for each product has scalar properties for title, author, publication date, and number of pages, and which contains two nested collections—one for keywords and another for ratings:

```
class Product
{
    string Title;
    string Author;
    int Year;
    int Pages;
    IEnumerable<string> Keywords;
    IEnumerable<string> Ratings;
}
```

Given this class declaration, we can use object initializers to create a product and insert it into a collection using collection initializers:

```
var _1579124585 = new Product
{
   Title = "The Right Stuff",
   Author = "Tom Wolfe",
   Year = 1979,
   Pages = 320,
```

```
    Keywords = new[]{ "Book", "Hardcover", "American" },
    Ratings = new[]{ "*****", "4 stars" },
}
var Products = new[]{ _1579124585 };
```

The program produces in memory the object graph shown in figure 1. Note that the two nested collections for the Keywords and Ratings properties are both represented by actual objects with their own identity.

Using the LINQ (Language Integrated Query) comprehension syntax introduced in C# 3.0,[11] we can find the titles and keywords of all products that have four-star ratings using the following query:

```
var q = from product in Products
        where product.Ratings.Any(rating⇒rating == "*****")
        select new{ product.Title, product.Keywords };
```

The LINQ comprehension syntax is just syntactic sugar for a set of standard query operators that can be defined in any modern programming language with closures, lambda expressions (written here as rating⇒rating == "*****"), or inner-classes such as Objective-C, Ruby, Python, JavaScript, Java, or C++. The C# compiler translates the previous query to the following de-sugared target expression:

FIGURE 1

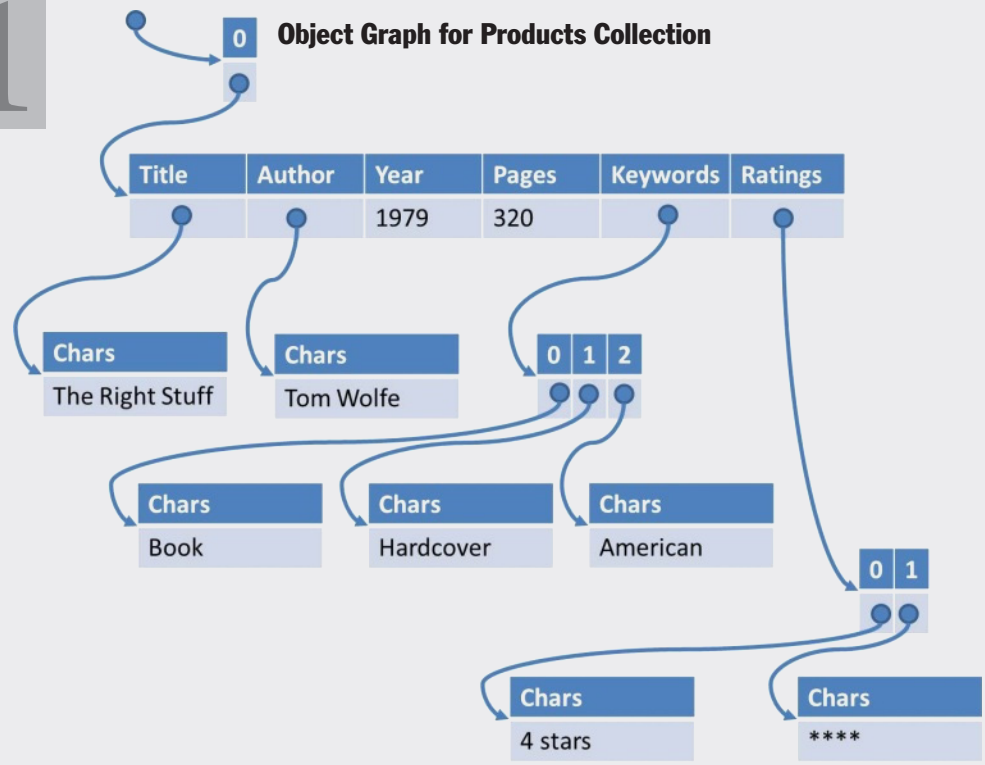
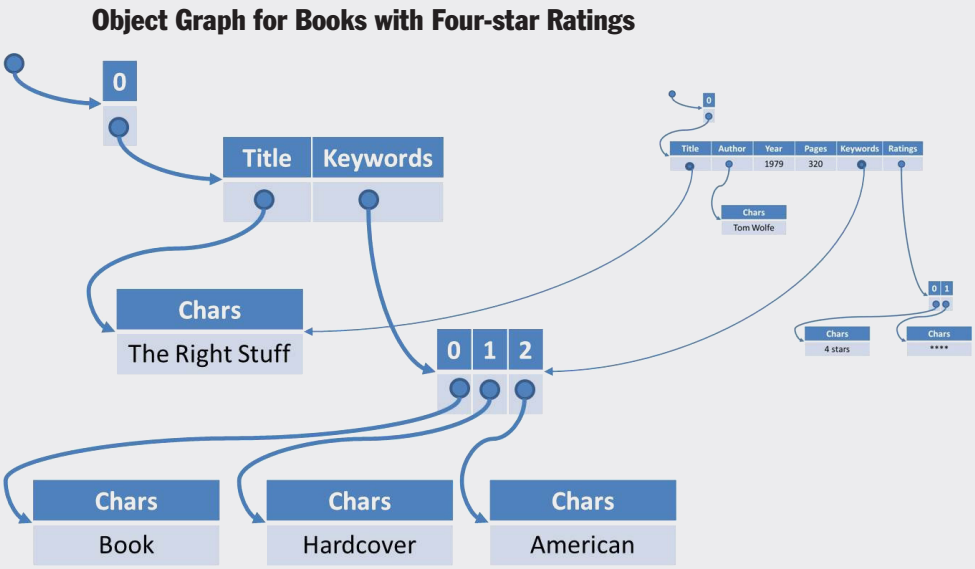


FIGURE 2




```
var q = Products.Where(product⇒
          product.Ratings.Any(rating⇒rating == "*****")).Select(product⇒
            new{ product.Title, product.Keywords });
```

The various values in the query result, in particular the Keywords collection, are fully shared with the original object graph, and the result is a perfectly valid object graph, as shown in figure 2.

Now let's redo this example using the relational model. First of all, we must normalize our nested Product class into three flat tables, for Products, Keywords, and Ratings respectively, as shown below. Each value in the relational world requires a new primary key property (here all called ID). Furthermore, the Keywords and Ratings tables require an additional foreign-key property ProductID that encodes the one-to-many association between Products and Keywords and Ratings. Later we will decorate these class declarations with additional metadata to reflect the underlying database tables.

```
class Products
{
  int ID;
  string Title;
  string Author;
  int Year;
  int Pages;
}
```

```
class Keywords
{
  int ID;
  string Keyword;
  int ProductID;
}
```

```
class Ratings
{
  int ID;
  string Rating;
  int ProductID;
}
```

In most commercial relational database systems, tables are defined by executing imperative CREATE TABLE DDL statements.

As usual in the relational world, we do not model the individual collections of keywords and ratings for each product as separate entities, but instead we directly associate multiple keywords

and ratings to a particular product. This shortcut works only for one-to-many relationships. The standard practice for many-to-many relationships (multivalued functions) requires intersection tables containing nothing but pairs of foreign keys to link the related rows.

Perhaps surprisingly for a "declarative" language, SQL does not have expressions that denote tables or rows directly. Instead we must fill the three tables in an imperative style using loosely typed DML statements, which we express in C# as follows:

```
Products.Insert
  ( 1579124585
  , "The Right Stuff"
  , "Tom Wolfe"
  , 1979
  , 320
  );
```

```
Keywords.Insert
  ( 4711, "Book"
  , 1579124585
  );
Keywords.Insert
  ( 1843, "Hardcover"
  , 1579124585
  );
Keywords.Insert
  ( 2012, "American"
  , 1579124585
  );
```

```
Ratings.Insert
  ( 787, "****"
  , 1579124585
  );
Ratings.Insert
  ( 747, "4 stars"
  , 1579124585
  );
```

These DML statements create three tables filled with the rows shown in figure 3.

FIGURE 3

Relational Tables for Product Database

Ratings

ID	Rating	ProductID
787	****	1579124585
747	4 stars	1579124585

Products

ID	Title	Author	Year	Pages
1579124585	The Right Stuff	Tom Wolfe	1979	304

Keywords

ID	Keyword	ProductID
4711	Book	1579124585
1843	Hardcover	1579124585
2012	American	1579124585

An important consequence of normalizing a single type into separate tables is that the database must maintain *referential integrity* to ensure that: the foreign-/primary-key relationships across related tables remain synchronized across mutations to the tables and rows; the primary key of every row remains unique within its table; and foreign keys always point to a valid primary key. For example, we cannot delete a row in the Products table without also deleting the related rows in the Keywords and Ratings tables.

Referential integrity implies a *closed-world assumption* where transactions on the database are serialized by (conceptually) suspending the world synchronously, applying the required changes, and resuming the world again when referential integrity has been restored successfully, rolling back any changes otherwise. Assuming a closed world is, we claim, both a strength and a weakness of the relational model. On the one hand, it simplifies the life of developers via ACID (atomicity, consistency, isolation, durability) transactions (although in practice, for efficiency, one must often deal with much weaker isolation levels) and allows for impressive (statistics-based) query optimization. The closed-world assumption, however, is in direct contradiction with distribution and scale-out. The bigger the world, the harder it is to keep closed.

Returning to our example, we present the naïve query to find the titles and keywords of all products that have four stars, expressed directly in terms of the relational model. It creates the cross-product of all possible combinations of Products, Keywords, and Ratings, and selects only the title and keyword where the keyword and rating are related to the product and the rating has four stars:

```
var q =  from product in Products
         from rating in Ratings
         from keyword in Keywords
         where product.ID == rating.ProductId
             && product.ID == keyword.ProductID
             && rating == "*****"
         select new{ product.Title, keyword.Keyword };
```

The result of this query is the row set shown in figure 4. Disappointingly, this row set is not itself normalized.

In fact, to return the normalized representation of our object-graph query, we need to perform two queries (within a single transaction) against the database: one to return the title and its primary key, and a second query that selects the related keywords.

FIGURE 4

Tabular Result for Books with Four-star Ratings

Title	Keyword
The Right Stuff	Book
The Right Stuff	Hardcover
The Right Stuff	American

What we observe here is SQL's lack of *compositionality*—the ability arbitrarily to combine complex values from simpler values without falling outside the system. By looking at the grammar definition for table rows, we can immediately see that SQL lacks compositionality; since there is no recursion, rows can contain only scalar values:

```
row ::= new { ..., name = scalar, ... }
```

Compare this with the definition for anonymous types, where a row can contain arbitrary values, including other rows (or nested collections):

```
value ::= new { ..., name = value, ... } | scalar
```

SQL is rife with noncompositional features. For example, the semantics of NULL is a big mess: why does adding the number 13 to a NULL value, `13+NULL`, return NULL, but summing the same two values, `SUM(13, NULL)`, returns 13?

Also, even though query optimizers in modern SQL implementations are remarkably powerful, the original query will probably run in cubic time when implemented via three nested loops that iterate over every value in the three tables. A seasoned SQL programmer would instead use explicit join syntax to ensure that the query is as efficient as our object-based query:

```
var q = from product in Products
        join rating in Ratings on product.ID equals rating.ProductId
        where rating == "*****"
        select product into FourStarProducts
        from fourstarproduct in FourStarProducts
        join keyword in Keywords on product.ID equals keyword.ProductID
        select new{ product.Title, keyword.Keyword };
```

Depending on the encoding of the nesting of the result of a join using flat result sets, the SQL programmer must choose among various flavors of `INNER`, `OUTER`, `LEFT`, and `RIGHT` joins.

IMPEDANCE MISMATCH

In 1984 George Copeland and David Maier recognized the impedance mismatch between the relational and the object-graph model just described,[5] and in the quarter century since, we have seen an explosion of O/R (object-relational) mappers that attempt to bridge the gap between the two worlds.

A more skeptical view of O/R mappers is that they are undoing the damage caused by normalizing the original object model into multiple tables. For our running example this means that we have to add back information to the various tables to recover the relationships that existed in the original model. In this particular case we use the LINQ-to-SQL custom metadata annotations; other O/R mappers use similar annotations, which often can be inferred from naming conventions of types and properties.

```
[Table(name="Products")]
class Product
{
    [Column(PrimaryKey=true)]int ID;
    [Column]string Title;
    [Column]string Author;
    [Column]int Year;
    [Column]int Pages;
    private EntitySet<Rating> _Ratings;
    [Association( Storage="_Ratings", ThisKey="ID",OtherKey="ProductID"
    , DeleteRule="ONDELETECASCADE")]
    ICollection<Rating> Ratings{ … }

    private EntitySet<Keyword> _Keywords;
    [Association( Storage="_Keywords", ThisKey="ID"
    , OtherKey="ProductID", DeleteRule="ONDELETECASCADE")]
    ICollection<Keyword> Keywords{ … }
}

[Table(name="Keywords")]
class Keyword
{
    [Column(PrimaryKey=true)]int ID;
    [Column]string Keyword;
    [Column(IsForeignKey=true)]int ProductID;
}

[Table(name="Ratings")]
class Rating
{
    [Column(PrimaryKey=true)]int ID;
    [Column]string Rating;
    [Column(IsForeignKey=true)]int ProductID;
}
```

Note that the resulting object model is necessarily more complex than we started with since we are forced to introduce explicit representations for Rating and Keyword collections that did not exist in our original object model. The existence of the various foreign- and primary-key properties is further evidence that the O/R mapping abstraction is leaky.

Aside from those small differences, the net result of all this work is that we can now write the query to find all products nearly as concisely as we did before normalization:

```
var q = from product in Products
        where product.Ratings.Any(rating⇒rating.Rating == "*****")
        select new{ product.Title, product.Keywords };
```

Since the results must be rendered as object graphs, the O/R mapper will make sure that the proper nested result structures are created. Unfortunately, not every O/R mapper does this efficiently.[9]

It is not only the programmer who needs to recover the original structure of the code. The database implementer must also jump through hoops to make queries execute efficiently by building indexes that avoid the potential cubic effect that we observed earlier. For one-to-many relationships, indexes are nothing more than nested collections resulting from precomputing joins between tables to quickly find all the rows whose foreign keys point to a row with a particular primary key. Since the relational model is not closed under composition, however, the notion of index has to be defined *outside* the model.

Two natural indexes correspond to the relationships between Products and Ratings and Products and Keywords, respectively. For each product in the Product table, the *ratings* index contains the collection of all related ratings:

```
from rating in Ratings where rating.ProductID == product.ID
select rating;
```

Similarly, for each product in the Product table, the *keywords* index contains the collection of all keywords related to that product:

```
from keyword in Keywords where keyword.ProductID == product.ID
select keyword;
```

If we visualize the indexes as additional columns on the Products table, the reversal of the original relationships between the tables becomes apparent. Each row in the Products table now has a collection of foreign keys pointing to the Keywords and Ratings tables much like the original object graph, as shown in figure 5.

One of the advantages touted for normalization over hierarchical data is the ability to perform ad-hoc queries—that is, to join tables on conditions not envisioned in the original data model. For example, we could try to find all pairs of products where the length of the title of one product is the same as the length of the author's name in the other using the following query:

```
from p1 in Products
from p2 in Products
where p1.Title.Length == p2.Author.Length
select new{ p1, p2 };
```

Without an index, however, this query requires a full table scan and hence takes quadratic time in the length of the Products table.

The ability to create indexes makes a closed-world assumption. For example, if we modify the

FIGURE 5

Keyword and Ratings Index on Products Table

ID	Title	Author	Year	Pages
579124585	The Right Stuff	Tom Wolfe	1979	304

Keywords		
4711	1843	2012

Ratings	
787	747

ID	Keyword	ProductID
4711	Book	1579124585
1843	Hardcover	1579124585
2012	American	1579124585

ID	Rating	ProductID
787	****	1579124585
747	4 stars	1579124585

previous ad-hoc query to find all pairs of Web pages where one page has a URL referencing the other, it should be obvious that building an index for this join is quite a hard task when you do not have the whole Web available inside a single database:

```
from p1 in WWW
from p2 in WWW
where p2.Contains(p1.URL)
select new{ p1, p2 };
```

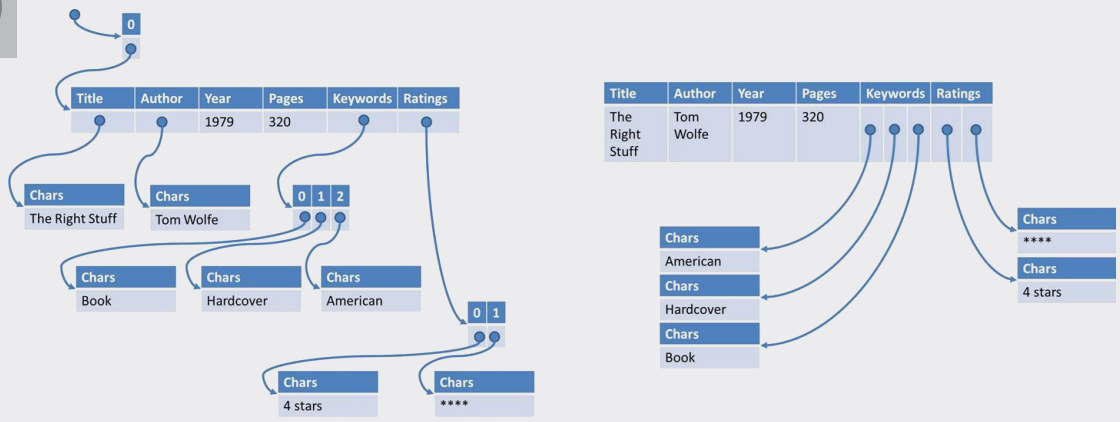
Summarizing what we have learned so far, we see that in order to use a relational database, starting with a natural hierarchical object model, the designer needs to normalize the data model into multiple types that no longer reflect the original intent; the application developer must reencode the original hierarchical structure by decorating the normalized data with extra metadata; and, finally, the database implementer has to speed up queries over the normalized data by building indexes that essentially re-create the original nested structure of the data as well.

NOSQL IS COSQL

At this point it feels like the conceptual dissonance between the key-value and foreign-/primary-key data models is insurmountable. That would be a pity since clearly each has its strengths and weaknesses. Wouldn't it be great if we could give a more mathematical explanation of where the relational model shines and where the object-graph model works best? As it turns out, we can find the answer to this question by taking a closer look at the (in-memory) structures created for our

FIGURE 6

Object Graph for Products Collection with Keywords and Ratings Inlined



running example in both models.

Let's start by slightly simplifying the object-graph example. We do so by removing the object identity of the Ratings and Authors collections to reflect more directly how they are modeled in the relational world. We inline the Keywords and Ratings items directly into the parent Product, as if we had value-based collections. Pictorially, we move from the diagram on the left to the one on the right in figure 6:

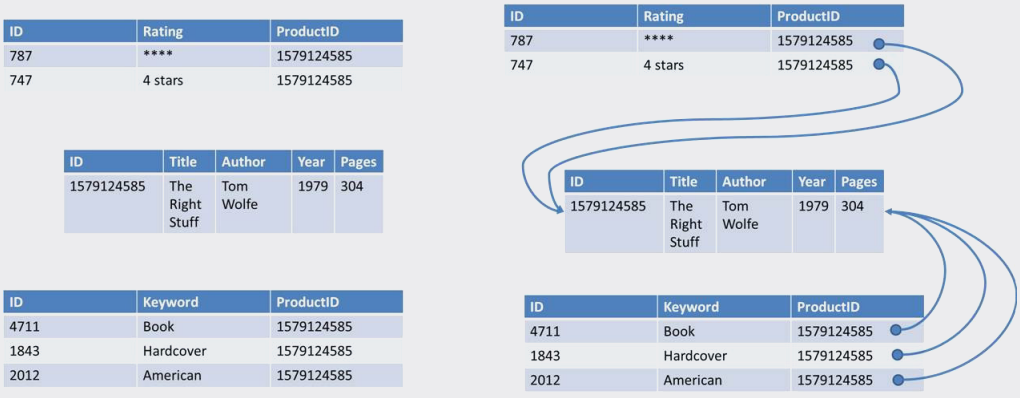
For the tabular representation, we show explicit arrows for the relationship from foreign keys to primary keys. Again, pictorially, we move from the diagram on the left to the one on the right in figure 7:

When we do a side-by-side comparison of the two rightmost diagrams, we notice two interesting facts:

• In the object-graph model, the identity of objects is *intensional*—that is, object identity is not part of the values themselves but determined by their keys in the store. In the relational model, object identity is *extensional*—that is, object identity is part of the value itself, in the form of a primary key.

FIGURE 7

Relational Tables for Product Database with Explicit Relationships



• Modulo the notion of object identity, the two representations are extremely similar; the only difference is that the arrows are reversed!

At this point, it appears that there is a strong correspondence between these two representations: they both consist of a collection of elements (objects or rows) and a collection of arrows between the elements, the only difference being the direction of the arrows. Is there some precise way of describing such a situation? Fortunately, there is an entire area of mathematics designed exactly for this: category theory.[1]

Obviously, the precise formalization of SQL and noSQL as categories is outside the scope of this article, but it is illustrative to learn a little bit of category theory nonetheless. Category theory arose from studies of mathematical structures and an attempt to relate classes of structures by considering the relations between them. Thus, category theory expresses mathematical concepts in terms of *objects*, *arrows* between objects, and the *composition* of arrows, along with some axioms that the composition of arrows should satisfy. A computational view of category theory is that it is a highly stylized, compact functional language. Small as it is, it's enough to represent all of mathematics. For computer scientists, category theory has proved to be a rich source of techniques that are readily applicable to many real-world problems. For example, Haskell's approach to modeling imperative programming is lifted straight from a categorical model of the problem.

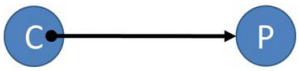
The first powerful concept from category theory that we will use is *duality*. Examples of duality abound in computer science. Every programmer is familiar with the two dual forms of De Morgan's law:

```
!(a && b) == (!a) | | (!b)
 ! (a| |b) == (!a)&&(!b)
```

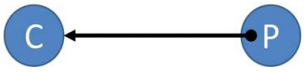
Other examples of dualities in computer science are between reference counting and tracing garbage collection, between call-by-value and call-by-name, between push- and pull-based collections, and between transactional memory and garbage collection, among many others.

Formally, given a category C of objects and arrows, we obtain the dual category co(C) by reversing all the arrows in C. If a statement T is true in C, then its dual statement co(T) is true in the dual category co(C). In the context of this article, we can read "opposite statement" for "dual statement".

In the SQL category, child nodes point to parent nodes when the foreign key of a child node equals the primary key of the parent node:



In the noSQL category, the arrows are reversed. Parent nodes point to child nodes when the child pointer in the parent equals the address of the child node in the store:



In other words, the noSQL category is the dual of the SQL category—noSQL is really coSQL. The implication of this duality is that coSQL and SQL are not in conflict, like good and evil. Instead they

are two opposites that coexist in harmony and can transmute into each other like yin and yang. Interestingly, in Chinese philosophy yin symbolizes open and hence corresponds to the open world of coSQL, and yang symbolizes closed and hence corresponds to the closed world of SQL.

Because SQL and coSQL are mathematically dual, we can reason precisely about the tradeoffs between the two instead of relying on rhetoric and anecdotal evidence. Table 1 gives a number of statements and their duals as they hold for SQL and coSQL, respectively.

Table 1. Consequences of the Duality between SQL and coSQL

SQL	coSQL
Children point to parents	Parents point to children
Closed world	Open world
Entities have identity (extensional)	Environment determines identity (intensional)
Necessarily strongly typed	Potentially dynamically typed
Synchronous (ACID) updates across multiple rows	Asynchronous (BASE) updates within single values
Environment coordinates changes (transactions)	Entities responsible to react to changes (eventually consistent)
Value-based, strong reference (referentially consistent)	Computation-based, weak reference (expect 404)
Not compositional	Compositional
Query optimizer	Developer/pattern

If we really take the duality to heart, we may also choose to (but don't have to) fine-tune our model for key-value stores to reflect the duality between values and computations, and that between synchronous ACID and asynchronous BASE (basically available, soft state, eventually consistent).[13]

Looking up a value given its address or key in a key-value store can involve a *computation*, which in a truly open world has potential latency and may fail. For example, in the C# language getters and setters, known as properties, can invoke arbitrary code. Perhaps an even better example of a computation-driven key-value store with long latency and high probability of failure (always able to handle a 404) is the Web, with URIs as keys, "resources" as values, and the HTTP verbs as a primitive query and data-manipulation language. On the other hand, in a C-like key-value memory model, we usually make the simplifying assumption that a key lookup in memory takes constant time and does not fail.

Traversing a relationship in the closed world of the relational model involves comparing two *values* for equality, which is guaranteed to succeed because of referential integrity; and vice versa, referential consistency dictates that relationships are value-based. Otherwise, we could never be sure that referential consistency actually holds.

Note that comparing keys by value requires that objects in the SQL category are strongly typed, at least enough to identify primary and foreign keys; and dually, since we do not need to know anything about the value of a coSQL object to find it using its key, objects in the coSQL world can be loosely typed.

FIGURE 8

Amazon SimpleDB Representation of Products Collection

Title	Author	Year	Pages	Keywords	Ratings
The Right Stuff	Tom Wolfe	1979	320	Hardcover American Book	**** 4 stars

RELATIONSHIP TO THE REAL WORLD

Our abstract model of the SQL category did not impose any restrictions on the structure of rows; we assumed only that we could determine a primary or foreign key to relate two rows.

In the typical relational model we would further impose the constraint that rows consist of flat sequences of scalar values (the so-called First Normal Form, or 1-NF). If we dualize relations in 1-NF, then we get a key-value model where values consist of either scalars or keys or collections of scalars or keys. Surprisingly, this is precisely the Amazon SimpleDB data model (figure 8).

If we assume that rows in the foreign-/primary-key model are simply blobs and keys are strings, then the dual key-value model is exactly the HTML5 key-value storage model:

```
interface Storage {
  readonly attribute unsigned long length;
  getter DOMString key(in unsigned long index);
  getter any getItem(in DOMString key);
  setter creator void setItem(in DOMString key, in any data);
  deleter void removeItem(in DOMString key);
  void clear();
}
```

A LITTLE MORE CATEGORY THEORY

So far we have discussed the basic data models for SQL and coSQL, but we have not yet touched upon queries. By applying a little more category theory we can show how a single abstraction, monads and monad comprehensions, can be used as a unified query language for both SQL and coSQL.

To talk about queries, we need to be more precise about what we mean by *collections* of values. Pure relational algebra is based on sets of rows, but actual relational databases use multisets (bags) or ordered multisets (permutations). To model collections abstractly, we look at sets/bags/permutations of rows and apply the category theory dictum: "What is the interface that these various collections of rows implement?" and "How do we generalize queries based on such an interface?"

First, let us stick with simple set collections. When we write a SQL query such as

```
SELECT F(a,b)
FROM as AS a, bs AS b
WHERE P(a,b)
```

the SQL compiler translates that pretty syntax into a relational-algebra expression in terms of selection, projection, joins, and Cartesian product. As is the custom in the relational world, the various operations are denoted using Greek symbols:

$\pi_F(\sigma_p(as \times bs))$

There is no reason to restrict the relational algebra operators to work over just sets (or bags, or permutations) of rows. Instead, we can define similar operators on *any* collection M<T> of values of *arbitrary* type T.

The interface for such abstract collections M<T> is a straightforward generalization of that of sets. It allows us to create an empty collection using the constant ∅; create a singleton collection of type M<T> given some value of type T using the function {_} ∈ T→ M<T> (the notation T→M<T> denotes a function/closure/lambda expression that maps an argument value of type T to a result collection of type M<T>); and combine two collections into a larger collection using the binary operator ∪ (depending on the commutativity and idempotence of ∪, we obtain the various sorts of collections such as lists, permutations, bags, and sets):

$\emptyset \in M\langle T\rangle$
$\{\_\} \in T \to M\langle T\rangle$
$\cup \in M\langle T\rangle \times M\langle T\rangle \to M\langle T\rangle$

Using these constructors, we can generalize the traditional relational algebra operators (selection σ_p, projection π_F, and Cartesian product ×Π to operate over generalized collections using the following signatures:

$\sigma_p \in M\langle T\rangle \times (T \to bool) \to M\langle T\rangle$
$\pi_F \in M\langle T\rangle \times (T \to S) \to M\langle S\rangle$
$\times \in M\langle T\rangle \times M\langle S\rangle \to M\langle T \times S\rangle$

In fact, if we assume a single operator for correlated subqueries, which we call SelectMany, but is often called CROSS APPLY in SQL

$SelectMany \in M\langle T\rangle \times (T \to M\langle S\rangle) \to M\langle S\rangle$

then we can define the other operators in terms of this single one, as follows:

$\sigma_p(as) = SelectMany(as, a \Rightarrow P(a) ? \{a\} : \emptyset)$
$\pi_f(as) = SelectMany(as, a \Rightarrow \{F(a)\})$
$as \times bs = SelectMany(as, a \Rightarrow \pi(b \Rightarrow (a,b), bs))$

Rather incredibly, an interface of this shape is well known in category theory. It is called a *monad*, where the type constructor M<_> is a *functor* of the monad; the constructor {_} is the unit of the monad; SelectMany is the bind of the monad; and ∅ and ∪ are the neutral element and addition,

respectively. For the rest of us, they are just the signatures for methods defined on an abstract interface for collections.

This is no theoretical curiosity. We can play the same syntactic tricks that SQL does with relational algebra, but using monads instead. Such *monad comprehensions* have been recognized as a versatile query language by both functional programmers and database researchers.[8]

LINQ queries are just a more familiar SQL-like syntax for monad comprehensions. Instead of Greek symbols, LINQ uses human-readable identifiers such as xs.Where(P) for $\sigma_p(xs)$ and xs.Select(F) for $\pi_F(xs)$. To accommodate a wide range of queries, the actual LINQ standard query operators contain additional operators for aggregation and grouping such as

Aggregate ∈ M<T>x(TxT→T) → T
GroupBy ∈ M<T>x(T→K) → M<KxM<T>>

Any data source that implements the standard LINQ query operators can be queried using comprehension syntax, including both SQL and coSQL data sources, as we show in the next section.

The .NET framework defines a pair of standard interfaces IEnumerable<T> and IQueryable<T> that are often implemented by data sources to support querying, but it is by no means necessary to use these particular interfaces. Other standard interfaces that support LINQ query operators include the IObservable<T> and IQbservable<T> interfaces that make it possible to use LINQ for complex event processing.[10]

SCALABILITY AND DISTRIBUTION

In contrast to most treatments of noSQL, we did not mention scalability as a defining characteristic of coSQL. The openness of the coSQL model eases scalable implementations across large numbers of physically distributed machines.

When using LINQ to query SQL databases, typically similar rows are stored in tables of some concrete type that implements the IQueryable<T> interface. Relationships between rows are lifted to bulk operations over collections that perform joins across these tables; hence, queries take any number of related tables and from those produce a new one (shown pictorially in figure 9):

IQueryable<S>xIQueryable<T>xIQueryable<U>→IQueryable<R>

FIGURE 9

Foreign Key Relationships between Three Relational Tables

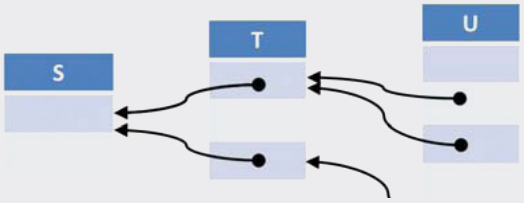
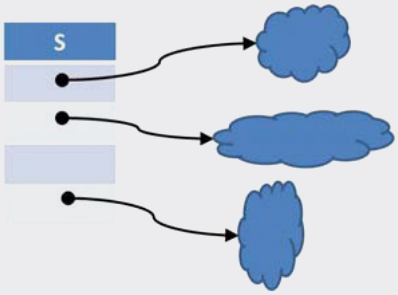


FIGURE 10

Collection of coSQL Documents



Because relationships in SQL cross different tables, it is nontrivial to partition the single closed world into independent worlds of "strongly connected" components that can be treated independently. In a closed world, however, query optimizers can leverage all the information and statistics that are available about the various tables. This allows users to write declarative queries focusing on the "what" and letting the system take care of the "how".

In the coSQL case, a typical scenario is to have a single collection of type IQueryable<S> of (pointers to) self-contained denormalized "documents" of type S. In that case, queries have the following type (shown pictorially in figure 10):

IQueryable<S> → R

When there are no cross-table relationships, collections $\{x_0,x_1,...,x_{n-1}\} \in$ M<S> that are the source of coSQL queries can be naturally horizontally partitioned, or sharded, into individual subcollections $\{x_0\} \cup \{x_1\} \cup ... \cup \{x_{n-1}\}$, and each such subcollection $\{x_i\}$ can be distributed across various machines on a cluster.

For a large subset of coSQL queries, the shape of the query closely follows the shape of the data. Such *homomorphic* queries map a collection $xs=\{x_0\} \cup \{x_1\} \cup ... \cup \{x_{n-1}\}$ to the value $f(x_0) \oplus f(x_1) \oplus ... \oplus f(x_{n-1})$—that is, they are of the form xs.Select(f).Aggregate(⊕) for some function $f \in S \rightarrow R$ and binary operator $\oplus \in RxR \rightarrow R$. In fact, Richard Bird's first homomorphism lemma[3] says that any function $h \in M<S> \rightarrow R$ is a homomorphism with respect to ∪ if and only if it can be factored into a map followed by a reduce: h(xs) = xs.Select(f).Aggregate(⊕). Mathematics dictates that coSQL queries are performed using MapReduce.[6]

Practical implementations of MapReduce usually slightly generalize Bird's lemma to use SelectMany instead of Select so that the map phase can return a collection instead of a single value, and insert an intermediate GroupBy as a way to "write" equivalence classes of values from the map phase into the key-value store for subsequent processing in the reduce phase, and then aggregate over each subcollection:

xs.SelectMany(f).GroupBy(s).Select((k,g) →g.Aggregate($\oplus_k$))

For example, DryadLINQ[15] uses the type PartitionedTable<S>:IQueryable<S> to represent the partitioned input collection for LINQ queries and then implements MapReduce over the partitioned collection using the following function:

```
MapReduce ∈ IQueryable<S>                    // source
            x(S→IEnumerable<M>)              // mapper
            x(M→K)                           // key selector
            x(KxIEnumerable<M>→R)            // reducer
            →IQueryable<R>
```

In an open world where collections are distributed across the network, it is much harder for a query optimizer to perform a global optimization taking into account latency, errors, etc. Hence, most coSQL databases rely on explicit programmatic queries of a certain pattern such as MapReduce that can be executed reliably on the target physical machine configuration or cluster.

CONCLUSION

The nascent noSQL market is extremely fragmented, with many competing vendors and technologies. Programming, deploying, and managing noSQL solutions requires specialized and low-level knowledge that does not easily carry over from one vendor's product to another.

A necessary condition for the network effect to take off in the noSQL database market is the availability of a common abstract mathematical data model and an associated query language for noSQL that removes product differentiation at the *logical level* and instead shifts competition to the *physical* and *operational* level. The availability of such a common mathematical underpinning of all major noSQL databases can provide enough critical mass to convince businesses, developers, educational institutions, etc. to invest in noSQL.

In this article we developed a mathematical data model for the most common form of noSQL—namely, key-value stores as the mathematical dual of SQL's foreign-/primary-key stores. Because of this deep and beautiful connection, we propose changing the name of noSQL to coSQL. Moreover, we show that monads and monad comprehensions (i.e., LINQ) provide a common query mechanism for both SQL and coSQL and that many of the strengths and weaknesses of SQL and coSQL naturally follow from the mathematics.

In contrast to common belief, the question of big versus small data is orthogonal to the question of SQL versus coSQL. While the coSQL model naturally supports extreme sharding, the fact that it does not require strong typing and normalization makes it attractive for "small" data as well. On the other hand, it is possible to scale SQL databases by careful partitioning.[2]

What this all means is that coSQL and SQL are not in conflict, like good and evil. Instead they are two opposites that coexist in harmony and can transmute into each other like yin and yang. Because of the common query language based on monads, both can be implemented using the same principles. Q

ACKNOWLEDGMENTS

Many thanks to Brian Beckman, Jimmy "the aggregator" Nilsson, Bedarra-Dave Thomas, Ralf Lämmel, Torsten Grust, Maarten Fokkinga, Rene Bouw, Alexander Stojanovic, and the anonymous referee for their comments that drastically improved the presentation of this paper, and of course to Dave Campbell for supporting work on all cool things LINQ.

REFERENCES

1. Awodey, S. 2010. *Category Theory* (second edition). Oxford University Press.
2. Baker, J., Bond, C., Corbett, J., Furman, J., Khorlin, A., Larson, J., et al. 2011. Megastore: providing scalable, highly available storage for interactive services. *Conference on Innovative Data Systems Research (CIDR).*
3. Bird, R. 1987. An introduction to the theory of lists. In *Logic Programming and Calculi of Discrete Design,* ed. M. Broy, 3-42. Springer-Verlag.
4. Codd, T. 1970. A relational model of data for large shared data banks. *Communications of the ACM, 13*(6).
5. Copeland, G., Maier, D. 1984. Making Smalltalk a database system. *Proceedings of the ACM SIGMOD International Conference on Management of Data.*
6. Fokkinga, M. 2008. MapReduce—a two-page explanation for laymen; http://www.cs.utwente.nl/~fokkinga/mmf2008j.pdf.
7. Ghosh, R. A. 2005. An economic basis for open standards; flosspols.org.
8. Grust, T. 2003. Monad comprehensions: a versatile representation for queries. In *The Functional Approach to Data Management,* eds. P. Gray, L. Kerschberg, P. King, and A. Poulovassilis, 288-311. Springer Verlag.
9. Grust, T., Rittinger, J., Schreiber, T. 2010. Avalanche-safe LINQ compilation. *Proceedings of the VLDB Endowment 3*(1-2).
10. Meijer, E. 2010. Subject/Observer is dual to iterator. Presented at FIT: Fun Ideas and Thoughts at the Conference on Programming Language Design and Implementation; http://www.cs.stanford.edu/pldi10/fit.html.
11. Meijer, E., Beckman, B., Bierman, G. 2006. LINQ: reconciling objects, relations, and XML in the .NET framework. *Proceedings of the ACM SIGMOD International Conference on Management of Data.*
12. Pirayoff, R. 2004. *Economics Micro & Macro.* Cliffs Notes.
13. Pritchett, D. 2008. BASE: an Acid alternative. *ACM Queue* (July).
14. Stonebraker, M., Hellerstein, J. M. 2005. What goes around comes around. In *Readings in Database Systems* (Fourth Edition), eds. M. Stonebraker, and J. M. Hellerstein, 2-41. MIT Press.
15. Yuan Yu, M. I. 2008. DryadLINQ: a system for general-purpose distributed data-parallel computing using a high-level language. *OSDI (Operating Systems Design and Implementation).*

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

ERIK MEIJER (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 the Reactive Framework (Rx).

GAVIN BIERMAN (gmb@microsoft.com) is a senior researcher at Microsoft Research Cambridge focusing on database query languages, type systems, semantics, programming language design and implementation, data model integration, separation logic, and dynamic software updating.