Towards a mathematical model for noSQL
NoSQL Took Away The Relational Model And Gave Nothing Back

Benjamin Black
10/26/2010 Palo Alto NoSQL meetup

What he meant:

NoSQL systems are lacking a standard model for describing and querying. Developing one should be a high priority task.

noSQL is dual to SQL
Objects vs Tables

I do consider assignment statements and pointer variables to be among computer science's most valuable treasures.

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

var _1579124585 = new Product
{
    Title = "The Right Stuff",
    Author = "Tom Wolfe",
    Year = 1979,
    Pages = 304,
    Keywords = new[] { "Book", "Hardcover", "American" },
    Ratings = new[] { "****", "4 stars" },
}

var Products = new[] { _1579124585 };
var q = from product in Products
    where product.Ratings.Any(rating => rating == "****")
    select new { product.Title, product.Keywords };

Tables

The relational model is a particularly suitable structure for the truly casual user (i.e., a non-technical person who merely wishes to interrogate the database, for example a housewife who wants to make enquiries about this week’s best buys at the supermarket). In the not too distant future the majority of computer users will probably be at this level.

C.J. Date & E.F. Codd
http://troels.arvin.dk/db/rdbms/links/#hierarchical
```
**table** Products
{}
  int ID;
  string Title;
  string Author;
  int Year;
  int Pages;
}

**table** Keywords
{}
  int ID;
  string Keyword;
  int ProductID;
}

**table** Ratings
{}
  int ID;
  string Rating;
  int ProductID;
}

Products.Insert
  (1579124585, "Tom Wolfe", 1979, 304);

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);

In SQL rows
are not expressible

<table>
<thead>
<tr>
<th>ID</th>
<th>Rating</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>787</td>
<td>****</td>
<td>1579124585</td>
</tr>
<tr>
<td>747</td>
<td>4 stars</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>Year</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1579124585</td>
<td>The Right Stuff</td>
<td>Tom Wolfe</td>
<td>1979</td>
<td>304</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Keyword</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Book</td>
<td>1579124585</td>
</tr>
<tr>
<td>1843</td>
<td>Hardcover</td>
<td>1579124585</td>
</tr>
<tr>
<td>2012</td>
<td>American</td>
<td>1579124585</td>
</tr>
</tbody>
</table>
```
Referential Integrity
Maintained by the environment

<table>
<thead>
<tr>
<th>ID</th>
<th>Rating</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>787</td>
<td>****</td>
<td>1579124585</td>
</tr>
<tr>
<td>747</td>
<td>4 stars</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

Foreign key must have corresponding primary key

Primary key must be unique

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

```
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 };
```
In mathematics, semantics, and philosophy of language, the **Principle of Compositionality** is the principle that the meaning of a complex expression is determined by the meanings of its constituent expressions and the rules used to combine them.

*Gottlob Frege 1848-1925*

---

**Objects**

*Fully compositional*

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

**Tables**

*Non compositional*

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

*Non compositional*

Query results denormalized
Query can only return single table
No recursion (but have CTEs)

NULL semantics a mess

\[
\text{Sum}(1, \text{NULL}) = 1
\]
\[
1 + \text{NULL} = \text{NULL}
\]

Impedance Mismatch

The problem with having two languages is “impedance mismatch.” One mismatch is conceptual - the data language and the programming languages might support widely different programming paradigms. [...] The other mismatch is structural - the languages don’t support the same data types, [...]

*George Copeland & David Maier 1984*
The "relational" data model, enunciated by Ted Codd in a landmark 1970 article, was a major advance over DBTG. The relational model unified data and metadata so that there was only one form of data representation. It defined a non-procedural data access language based on algebra or logic. It was easier for end-users to visualize and understand than the pointers-and-records-based DBTG model. Programs could be written in terms of the "abstract model" of the data, rather than the actual database design; thus, programs were insensitive to changes in the database design.

Jim Gray

Codd's relational theory dressed up these concepts with the trappings of mathematics (wow, we lowly Cobol programmers are now mathematicians!) by calling files relations, records rows, fields domains, and merges joins. Computing history will consider the past 20 years as a kind of Dark Ages of commercial data processing in which the religious zealots of the Church of Relationalism managed to hold back progress until a Renaissance rediscovered the Greece and Rome of pointer-based databases. Database research has produced a number of good results, but the relational database is not one of them.

Henry G. Baker
LINQ to SQL provides a runtime infrastructure for managing relational data as objects without losing the ability to query. Your application is free to manipulate the objects while LINQ to SQL stays in the background tracking your changes automatically.

When one takes a look at the amount of code that the average application developer must write to address the impedance mismatch across various data representations (for example objects and relational stores) it is clear that there is an opportunity for improvement.
[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;
}

And we did not even talk about inheritance yet.
var q = from product in Products
    from rating in Ratings
    where product.ID == rating.ProductId
        && rating == "****"
    from keyword in Keywords
    where product.ID == keyword.ProductID
    select new { product.Title, keyword.Keyword };

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

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>1579124585</td>
<td>The Right Stuff</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Keyword</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Book</td>
<td>1579124585</td>
</tr>
<tr>
<td>1843</td>
<td>Hardcover</td>
<td>1579124585</td>
</tr>
<tr>
<td>2012</td>
<td>American</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

Indexes
Recover
Nesting
<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>Year</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1579124585</td>
<td>The Right Stuff</td>
<td>Tom Wolfe</td>
<td>1979</td>
<td>304</td>
</tr>
</tbody>
</table>

```sql
from rating in Ratings
where ID = rating.ID
select rating.ID
```

```sql
from keyword in Keywords
where ID = keyword.ID
select keyword.ID
```

<table>
<thead>
<tr>
<th>ID</th>
<th>Keyword</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Book</td>
<td>1579124585</td>
</tr>
<tr>
<td>1843</td>
<td>Hardcover</td>
<td>1579124585</td>
</tr>
<tr>
<td>2012</td>
<td>American</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Rating</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>787</td>
<td>****</td>
<td>1579124585</td>
</tr>
<tr>
<td>747</td>
<td>4 stars</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Keywords</th>
<th>4711</th>
<th>1843</th>
<th>2012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ratings</td>
<td>787</td>
<td>747</td>
<td></td>
</tr>
</tbody>
</table>
Normalization is for Sissies

Ad-hoc queries

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

Does not really work:
O(n^2)
No referential integrity

Ad-hoc queries don’t scale

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

Sorting the whole Web
Might be a bit of a challenge
**Designer**
*Remove* original hierarchical structure into normalized data

**App Developer**
*Recover* original hierarchical structure from normalized data

**Database Implementer**
*Recover* original hierarchical structure from normalized data

**PEACE**
*not* WAR
http://en.wikipedia.org/wiki/Math_Rescue
The Right Stuff
Tom Wolfe
1979
320
Chars
4 stars
Book
Char
American
Chars
Hardcover
<table>
<thead>
<tr>
<th>ID</th>
<th>Rating</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>787</td>
<td>****</td>
<td>1579124585</td>
</tr>
<tr>
<td>747</td>
<td>4 stars</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Title</th>
<th>Author</th>
<th>Year</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>1579124585</td>
<td>The Right Stuff</td>
<td>Tom Wolfe</td>
<td>1979</td>
<td>304</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ID</th>
<th>Keyword</th>
<th>ProductID</th>
</tr>
</thead>
<tbody>
<tr>
<td>4711</td>
<td>Book</td>
<td>1579124585</td>
</tr>
<tr>
<td>1843</td>
<td>Hardcover</td>
<td>1579124585</td>
</tr>
<tr>
<td>2012</td>
<td>American</td>
<td>1579124585</td>
</tr>
</tbody>
</table>

Draw relationships as arrows
Spot the differences
• Arrows are reversed
• Identity extensional/intensional
Categories, objects, and morphisms

Main articles: Category (mathematics) and Morphism

A category \( 	ext{C} \) consists of the following three mathematical entities:

- A class \( \text{ob} (\text{C}) \), whose elements are called objects;
- A class \( \text{hom} (\text{C}) \), whose elements are called morphisms or maps or arrows. Each morphism has a unique source object \( a \) and target object \( b \). We write \( f : a \rightarrow b \) and we say "\( f \) is a morphism from \( a \) to \( b \)." We write \( \text{hom}(a, b) \) (or \( \text{Hom}(a, b) \), or \( \text{hom}(a, b) \), or \( \text{Mor}(a, b) \), or \( C(a, b) \)) to denote the hom-class of all morphisms from \( a \) to \( b \);
- A binary operation \( \circ \), called composition of morphisms, such that for any three objects \( a, b, \) and \( c \) we have \( \text{hom}(a, b) \times \text{hom}(b, c) \rightarrow \text{hom}(a, c) \). The composition of \( f : a \rightarrow b \) and \( g : b \rightarrow c \) is written as \( g \circ f \) or \( g f \), governed by two axioms:
  - Associativity: If \( f : a \rightarrow b, g : b \rightarrow c \) and \( h : c \rightarrow d \) then \( h \circ (g \circ f) = (h \circ g) \circ f \).
  - Identity: For every object \( x \), there exists a morphism \( 1_x : x \rightarrow x \) called the identity morphism for \( x \), such that for every morphism \( f : a \rightarrow b \), we have \( 1_b \circ f = f = f \circ 1_a \).

From these axioms, it can be proved that there is exactly one identity morphism for every object. Some authors deviate from the definition just given by identifying each object with its identity morphism.

Relations among morphisms (such as \( fg = h \)) are often depicted using commutative diagrams, with "points" (corners) representing objects and "arrows" representing morphisms.

The definitions of categories and functors provide only the very basics of categorical algebra; additional important topics are listed below. Although there are strong interrelations between all of these topics, the given order can be considered as a guideline for further reading.

- The functor category \( \text{D}^{\text{C}} \) has as objects the functors from \( \text{C} \) to \( \text{D} \) and as morphisms the natural transformations of such functors. The Yoneda lemma is one of the most famous basic results of category theory; it describes representable functors in functor categories.
- Duality: Every statement, theorem, or definition in category theory has a dual which is essentially obtained by "reversing all the arrows." If one statement is true in a category \( \text{C} \) then its dual will be true in the dual category \( \text{C}^{\text{op}} \). This duality, which is transparent at the level of category theory, is often obscured in applications and can lead to surprising relationships.
- Adjoint functors: A functor can be left (or right) adjoint to another functor that maps in the opposite direction. Such a pair of adjoint functors typically arises from a construction defined by a universal property; this can be seen as a more abstract and powerful view on universal properties.
ForeignKey(f,s) = PrimaryKey(t)

Address(s) = Property(f,t)
In logic and mathematics, an **intensional** definition gives the meaning of a term by specifying all the properties required to come to that definition, that is, the necessary and sufficient conditions for belonging to the set being defined.

An **extensional** definition of a concept or term formulates its meaning by specifying its extension, that is, every object that falls under the definition of the concept or term in question.

**Objects**
A memory location contains an **object**
A pointer is the memory location of some object
*Memory location is not part of the object*

**Rows**
A **row** has a primary key
A foreign key is the value of a primary key
*Primary key is part of a row*
F-algebra

From Wikipedia, the free encyclopedia

In mathematics, specifically in category theory, an F-algebra for an endofunctor
\[ F : C \longrightarrow C \]
is an object \( A \) of \( C \) together with a \( C \)-morphism
\[ \alpha : FA \longrightarrow A. \]
In this sense F-algebras are dual to F-coalgebras.

F-coalgebra

From Wikipedia, the free encyclopedia

In mathematics, specifically in category theory, an F-coalgebra for an endofunctor
\[ F : C \longrightarrow C \]
is an object \( A \) of \( C \) together with a \( C \)-morphism
\[ \alpha : A \longrightarrow FA. \]
In this sense F-coalgebras are dual to F-algebras.

Relational Algebra

Algebraic: Table \( \bowtie \) Table \( \Downarrow \) Table

Join constructs new row by combining other rows

A Relational Model of Data for Large Shared Data Banks

E. F. Codd

IBM Research Laboratories, San Jose, California
Object CoAlgebra

c\text{o-}\text{Algebraic}: \text{Object} \bullet \text{Member} \triangleright \text{Object}^*

Member access \textit{destructs} existing object into constituent objects
noSQL is coSQL

noSQL and SQL are not in conflict, like good and evil.

They are two opposites that co-exist in harmony and can transmute into each other.

Like yin (open Ė noSQL) and yang (closed Ė SQL).
Consequences of Duality

If a statement \( T \) is true in \( C \)
Then its dual \( \text{co}(T) \) is true in \( \text{co}(C) \)

<table>
<thead>
<tr>
<th>SQL</th>
<th>coSQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children point to parents</td>
<td>Parents point to children</td>
</tr>
<tr>
<td>Closed world</td>
<td>Open world</td>
</tr>
<tr>
<td>Entities have identity (extensional)</td>
<td>Environment determines identity (intensional)</td>
</tr>
<tr>
<td>Synchronous (ACID)</td>
<td>Asynchronous (BASE)</td>
</tr>
<tr>
<td>Environment coordinates changes (transactions)</td>
<td>Entities responsible to react to changes (eventually consistent)</td>
</tr>
<tr>
<td>Not compositional</td>
<td>Compositional</td>
</tr>
<tr>
<td>Query optimizer</td>
<td>Developer/pattern</td>
</tr>
</tbody>
</table>

Open world
Cannot join, build indexes
Cannot coordinate transactions
Cannot maintain referential integrity
Pre-computed indexes
Eventually consistent
Weak pointers (expect 404)

**Life beyond Distributed Transactions: an Apostate’s Opinion**

Entities are collections of named (keyed) data which may be atomically updated within the entity but never atomically updated across entities.

Pat Helland
**SimpleDB Datamodel**

Domain ::= {Item; Row}*
Row ::= { ...; Attribute = Value+; ... }
Value ::= string | key

<table>
<thead>
<tr>
<th>Title</th>
<th>Author</th>
<th>Year</th>
<th>Pages</th>
<th>Keywords</th>
<th>Ratings</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Right Stuff</td>
<td>Tom Wolfe</td>
<td>1979</td>
<td>320</td>
<td>Hardcover</td>
<td>****</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>American Book</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4 stars</td>
<td></td>
</tr>
</tbody>
</table>

Actual mathematical dual of flat relational tables with scalars in columns

**SimpleDB Downside**

No way to retrieve multi-valued attributes using select query. Needs two round trips (can batch writes).

```
sdb.GetAttributes(new GetAttributesRequest
    {
        AttributeName = {"Keyword", "Rating"},
        DomainName="Books",
        ItemName = "...itemName() from query ...",
    });
```
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();
}

Actual mathematical dual of relational tables with blobs

What About SQL (the query language)
Monads as Kleisli triples

Rather than focusing on a specific \( T \), we want to find the general properties common to all notions of computation, therefore we impose as only requirement that \textit{programs} should form a category. The aim of this section is to convince the reader, with a sequence of informal argumentations, that such a requirement amounts to say that \( T \) is part of a Kleisli triple \((T, \eta, \mu)\) and that the category of programs is the Kleisli category for such a triple.

**Definition 1.2 ((Man76))** A Kleisli triple \textit{over a category} \( \mathcal{C} \) is a triple \((T, \eta, \mu)\), where \( T : \text{Obj}(\mathcal{C}) \rightarrow \text{Obj}(\mathcal{C}), \eta_A : A \rightarrow TA \) for \( A \in \text{Obj}(\mathcal{C}), \mu^* : TA \rightarrow TB \) for \( f : A \rightarrow TB \) and the following equations hold:

- \( \eta^*_A = \text{id}_{\eta_A} \)
- \( \eta_A ; f^* = f \) for \( f : A \rightarrow TB \)
- \( f^* ; g^* = (f ; g^*)^* \) for \( f : A \rightarrow TB \) and \( g : B \rightarrow TC \).

A Kleisli triple satisfies the \textit{mono} requirement provided \( \eta_A \) is \textit{mono} for \( A \in \mathcal{C} \).

Intuitively \( \eta_A \) is the \textit{inclusion} of values into computations (in several cases \( \eta_A \) is indeed a \textit{mono}) and \( f^* \) is the \textit{extension} of a function \( f \) from values to computations to a function from computations to computations, which first evaluates a computation and then applies \( f \) to the resulting value. In
What is the interface that the relational algebra implements?

We want to query both SQL and noSQL using the same query language

And every other data source as well.
Sets \notin "Collections"
Tuples \notin "Generics"

\emptyset :: M<T>
\cup :: M<T> \times M<T> \nsubseteq M<T>
\{\} :: T \nsubseteq M<T>

\sigma_p :: M<T> \times (T \nsubseteq \text{bool}) \nsubseteq M<T>

\pi_F :: M<T> \times (T \nsubseteq S) \nsubseteq M<S>

\chi :: M<T> \times M<S> \nsubseteq M<T \times S>

Correlated Subqueries

SelectMany ::
M<T> \times (T \nsubseteq M<S>) \nsubseteq M<S>

\sigma_p(as) =
as.SelectMany(\lambda a \nsubseteq
P(a)?\{a\}: \emptyset)
Correlated Subqueries

\[ \pi_F(as) = as.\text{SelectMany}(\lambda a \not\in \{F(a)\}) \]

\[ as \times bs = as.\text{SelectMany}(\lambda a \not\in \sigma_{\lambda b \not\in (a,b)}(bs)) \]

One important twist

\[ \text{SelectMany} :: M<T> \times (\text{Expr}<T\not\in M<S>>) \not\in M<S> \]

Intensional representation of code
Recognize the Monads?

M<_> † Functor
SelectMany † bind
{_} † return/η

μ :: M<M<T>> † M<T>
μ tss = tss.SelectMany(λts‡ ts)

LINQ == Monads

Syntactic sugar for monad comprehensions

Data source “implements” monadic interface (pattern)

One query syntax over multiple data models
coSQL naturally allows extreme horizontal partitioning

\[ 0...99 \cup 100...199 \cup 200...299 \]

**Bird’s First Homomorphism Lemma**

1987

A function
\[ h :: M<A> \rightarrow B \]
is a homomorphism wrt to \( \cup \) iff
\[ h = (\oplus/) \bullet (f^*) \]

for some
\[ f :: A \rightarrow B \]
and
\[ \oplus :: B \times B \rightarrow B \]

For the rest of us

*Every LINQ query can be executed as a MapReduce computation*
Google's MapReduce Programming Model -- Revisited

class MapReduce<k1, k2, v1, v2, v3>
{
    IEnumerable<KeyValuePair<k2, v2>> Map(k1 Key, v1 Value);
    v3 Reduce(k2 Key, IEnumerable<v2> Values);

    IEnumerable<KeyValuePair<k2, v3>> MapReduce (IEnumerable<KeyValuePair<k1, v1>> Input)
    {
        ...
    }
}
We Are Hiring

Databases
LINQ
Category
Theory
Functional Programming
Hacker
Distributed Systems