From Only-SQL to NoSQL to YeSQL

Solving the data scaling challenge without a complete rewrite

June 2011

Introduction

It has now been a good couple of years since the various anti-SQL proponents have gained enough momentum to come together under the wide umbrella of the term NoSQL. And it is clear that we can never go back: the typical relational database architecture is clearly insufficient for today's data-intensive applications, and the move to distributed architectures.

But is the problem in the architecture or the query language? The two are not interchangeable, though frequently confused.

Some answers can be found in the following articles, which represent a progression of ideas on this very relevant topic, based on various articles published in Nati Shalom's blog: http://natishalom.typepad.com

Should Web Apps "Just Say No" to SQL?

Pros and Cons of Non-SQL Patterns

This paper briefly reviews what is driving the trend of adopting alternatives to the traditional SQL DB, survey alternative approaches, and discuss not only their benefits but also the risks and caveats for real-life web applications.

DRIVERS FOR ADOPTING NON-SQL PATTERNS

A Computerworld article entitled "**No to SQL? Anti-database movement gains steam**" (Eric Lai, July 2009) points to three main drivers that lead companies like Google, Amazon and Facebook to choose an alternative approach:

- **Demand for extremely large scale:** If, until now, caching was the commonly used solution for read scalability, to resolve issues of centralized database bottlenecks with data-intensive applications; the social web is much more write intensive, requiring a solution for write scalability, a more difficult task than read scalability.
- **Complexity and cost of setting up database clusters:** "Sharding" a database, which involves cutting it up into multiple tables to run on large clusters or grids, is a complex process that requires hard work and special skills. Non-SQL solutions can run on regular PC clusters, which are inexpensive and easy to expand.
- **Better performance, even at the cost of reliability:** For example, HTTP Session data, which needs to be shared between several web servers. Because the data is transient (it is no longer needed when the user logs off) there is no need to store it in the database. However, for many applications this can cause inconsistency and make it very difficult to recover, in case of a widespread failure which interrupts customer transactions.

A growing number of application scenarios simply cannot be addressed with a traditional database approach, because of the continuous growth of data volumes, and the need for shorter processing time. The classic early adopters are those who hit the scalability wall when their applications simply cannot scale further without unbearable cost. It is very likely that as these alternative solutions mature, they will find their way into mainstream development as well.

This being said, many still agree that despite all the limitations of traditional database solutions, the SQL database is not going away. It will still be broadly used, but awareness of usage scenarios in which it has failed will grow, and needs to be replaced by the emerging alternatives.

WHAT ARE THE ALTERNATIVES?

The solutions most frequently mentioned in the context of non-SQL patterns are, Cassandra, Redis, HBase, Riak, MongoDB, Amazon's SimpleDB, Google's App Engine Megastore, CouchDB, and Microsoft's Azure Table. But the reality is that none of them can or should be positioned as a direct alternative to your existing database – a good case in point is SimpleDB. The main reason is that all of them offer weak or eventual consistency, which is unacceptable for many database-driven applications.

Another alternative solution is the in-memory data grid, such as memcached, GigaSpaces XAP, Oracle Coherence, and IBM eXtreme Scale. These solutions front-end the database with an in-memory cluster, which becomes the application's "system of record" and uses the SQL database as the background persistent store.

The main advantage of this model is that it is comparable to a traditional database – all the commercial data grid products (excluding memcached) guarantee more or less the same level of consistency and reliability as a traditional database. GigaSpaces XAP is unique in that it offers an even higher level of reliability than traditional databases. This is because it partitions and scales not only the data, but also the applications themselves and all the middleware they require to function, resulting in guaranteed uptime for the entire application.

STAYING FRIENDS WITH THE SQL WORLD

SQL databases are not going away, but there is definitely a place for a more specialized data management solutions alongside traditional SQL databases. The adoption of these new solutions should be determined by two main factors: **(1) How well they integrate with the SQL world**, and **(2) How easy it will be to develop** for these new alternatives, that is to say, how smooth the transition will be. For example, the in-memory data grid solutions are pre-integrated with SQL databases and enable various levels of synchronization with existing databases. This is extremely important because it minimizes the impact of non-SQL patterns on existing applications.

YeSQL (Part I)

An Overview of the Various Query Semantics in the Post Only-SQL World

The NoSQL (***Not Only*** SQL) movement faults the SQL query language for many of the scalability issues that we face today with the traditional database approach. I think that the main reason so many people have come to see SQL as the source of all evil is the fact that, traditionally, the query language was burned into the database implementation. So by saying NoSQL you basically say "No" to the traditional non-scalable RDBMS implementations.

This view has brought a flood of alternative query languages, each aiming to solve a defficiency in the traditional SQL query approach, such as a document model, or providing a simpler approach, such as the Key/Value query.

Most of the people I speak with seem fairly confused on this subject, and tend to use query semantics and architecture interchangeably. So I thought that a good start would be to provide a quick overview of what each query term stands for in the context of the NoSQL world. Then, I've tried to clear some common misconceptions -- which led me to come up with the YeSQL term.

COMMON QUERY SEMANTICS IN THE POST ONLY-SQL WORLD

The following are some of the common query semantics in the NoSQL world:

- **Key/Value query:** Probably the most basic query form. Each data item is associated with a unique identifier (key). In memcached, a common implementation, complex queries are deferred to an underlying database that is used as a search engine. The result is a key or set of keys that is then used to perform subsequent fetching of values through the memcache data store. They have gained new momentum in the post-SQL world because they lend themselves fairly natively to partitioning and distribution, which are key to making a data store scalable. In other words, people were willing to trade the rich query functionality provided by most traditional RDBMS for scalability with only basic query support if that was their only choice.
- **Document-based query:** This model's roots are in the search engine world, where it's common to store different document types, even if each represents an entirely different object. "Document" is not a Word or PDF found in search engines, but rather JSON or XML objects, or binary objects associated with a set of key/values, as in the case of Cassandra. In SQL terms, a document can be seen as a blob associated with a set of keys, each indexed independently and maintaining a reference to this blob. Each blob can be a different type (tables), and each blob can have different set of associated indexes (keys). Matching is done through the associated indexes. The result-set often includes multiple types, each containing a different set of data. Because the indexes and the blob don't need to conform to a strict structure of rows and tables, it is referred to as "schemaless", i.e., it can have different versions of the same type, and add fields to new types without having to modify any table or update older version of the data. Examples that support the document model are CouchDB and MongoDB.
- **Template query:** Template queries were common in JavaSpaces and even in later versions of Hibernate. With template-based matching, you can fetch an object based on class or inheritance hierarchy, as well as attribute values of the object. In more object-oriented versions of template matching you can also match based on specific items within a graph attribute. GigaSpaces is one of the better-known implementations that support the JavaSpaces template query model.

- **Map/Reduce:** Used to perform aggregated queries on a distributed data store. Simple scenarios would be Max or Sum. The query request must be executed independently in each partition (Map) and then aggregated back to the client (Reduce). An implicit Map/Reduce takes a query request and spreads its execution implicitly. The client gets the aggregated query as if it was a single query. In the explicit model you execute code in free-form, and can control the mapping model -- which call goes to which data, the code to run in each node (aka tasks), and the results. In a typical Hadoop implementation, Map/Reduce is often uses the explicit model. Frameworks like Hive and Pig provide an abstraction model that can handle this process implicitly.
- **SQL query:** If you think about it, SQL is yet another dynamic language specifically designed for complex data management, with data often ordered in tables and rows. Some SQL query semantics, such as Joins, distributed transactions, and others are known to be anti-pattern for scalability. This is the main reason SQL is associated with scalability limitations. Examples of NoSQL implementations that support SQL are Google Bigtable using JPA, Hive/Hadoop, MongoDB and GigaSpaces. I will discuss in further details below what this actually means.

YESQL: THERE'S NOTHING WRONG WITH SQL!

As we cover these query formats, it becomes apparent that there is nothing really wrong with SQL. Like many languages, SQL gives you a fairly long rope with which to hang yourself, but that is true of almost any language. If you design your data model to fit into a distributed model, SQL can be a fairly useful format to manage your data. A good example is Hive/Pig/Hbase and Google JPA/Bigtable. In all these cases the underlying data store is based on a scalable Key/Value store, but the front-end query language happens to be SQL-based. MongoDB aims toward a similar goal with the main difference that it provides SQL-like support and doesn't fully comply with any of the existing standards.

IT'S ABOUT THE ARCHITECTURE, STUPID!

NoSQL implementations such as Hive/HBase as well as JPA/Bigtable can be a good example of how next-generation databases can support both linear scaling and an SQL API.

The key is decoupling the query semantics from the underlying data-store as illustrated here.

CONVERGENCE IS UNDERWAY

There are already many new frameworks today that provide different levels of abstraction to the way Hadoop manages data in both query and processing, such as Hive, Cascading, and Pig. Many of them provide tools that the original creator of Hadoop never imagined.

Which brings me to the point that we can apply the same decoupling pattern mentioned above to support a document model in connection with SQL. Going forward, I believe that most of the leading databases will support all of the semantics listed above, and we won't have to choose a database implementation just because it supports a certain query language.

We've already seen a similar trend with dynamic languages. In the past, a language had to come with a full stack of tools, compiler, libraries, and development tools behind it, making the selection of a particular language quite strategic. Today, a JVM in Java or a CLR in .Net provides a common substrate that can support a large variety of dynamic languages on top of the same JVM runtime. Good examples are Groovy and Java or Jruby. The data management world is going to be built in similar way, from a variety of tools and data management languages, each serving a particular purpose but which can still share the same data store substrate. In this world we should be able to access any data using any of several query languages, regardless of how the data was stored. For example, storing a JSON object using a document model and then, at any time, querying that JSon object using SQL query semantics or a simple Key/Value API.

YeSQL (Part II):

PUTTING NoSQL, SQL, AND THE DOCUMENT MODEL TOGETHER

Above, I introduced common query semantics in the post Only-SQL world and made an argument that the right approach is to decouple the query semantics from the underlying NoSQL implementation. This would allow us to combine SQL semantics with the NoSQL backend to achieve the best of both worlds -- standard queries and scalability.

Here, I will illustrate this idea through some code examples using GigaSpaces as the underlying implementation of the concept. The example is based on the the same code examples from another earlier post, WTF is Elastic Data Grid? (For Example) where I covered some of the simple models for writing and reading objects to a distributed data grid. Toward the end, I will reference other examples such as Datanucleus and Hbase, as well as cover patterns for supporting NoSQL semantics such as document models with existing RDBMS such as MySQL.

SQL/NoSQL CODE EXAMPLE (USING GIGASPACES)

THE DATA MODEL

```
1          @SpaceClass
2          public class Data implements Serializable {
3
4           private Long id;
5           private String data;
6           private Map<string, object=""> info;
7
8           public Data(long id, String data , Map<string, object=""> info )
9          {
10
11            this.id = id;
12            this.data = data;
13            this.info = info;
14
15          }
16
17          @SpaceRouting
18          @SpaceId
19          public Long getId() {
20
21           return id;
22
23          }
24
25          // getter/setter for id, data, info attributes omitted
26
27         }
```

We use the @SpaceRouting annotation to set the routing index. This index determines which target partition this instance should belong to. @SpaceId determines the unique identifier (Key) for this instance.

ADDING DOCUMENT MODEL SEMANTICS

Most of the common examples for document-based APIs use a JSON data model as the document. A document in the JSON world would look something like this:

```
{
        "name":"Product",
        "properties":
        {
                "id":
                {
                        "type":"number",
                        "description":"Product identifier",
                        "required":true
                },
                "name":
                {
                        "description":"Name of the product",
                        "type":"string",
                        "required":true
                },
                "price":
                {
                        "type":"number",
                        "minimum":0,
                        "required":true
                },
                "tags":
                {
                        "type":"array",
                        "items":
                        {
                                "type":"string"
                        }
                }
        }
}
```

(Source: Wikipedia)

From a NoSQL implementation perspective, a document often translates to a Map of Maps where each attribute is mapped to a key/value representation and a nested value to a key whose value is a Map and so on.

DOCUMENT *MODEL IN GIGASPACES*

In the current version of GigaSpaces a document is basically a Map attribute whose values are indexed. That gives the flexibility to add/remove attributes on an existing object without changing the object schema as with any schemaless API. *A nested object would be mapped to a key whose value is a complex object that is stored just like any regular POJO*.

In this example I added an **info** attribute that is basically a Map of Key/Values.

```
1       private Map<string, object=""> info;
```

One of the main benefits of this approach is the combination of a strongly typed POJO/Table model with the flexibility of the document model. In other words, I could match objects by type, and each object can have a “fixed” structure that must conform to a certain type, and a variable

part that can vary on a per instance basis. In our specific example, when I look for a Data object I'm guaranteed to get the Data.id and Data.data, but Data.info could include a variable list of attributes that can vary on a per instance basis. I can still “pin” some attributes by forcing an indexing on the relevant keys as illustrated below:

```
1        // this defines several indexes on the same info property
2         @SpaceIndexes( { @SpaceIndex(path = "info.address", type =
3        SpaceIndexType.BASIC),
4         @SpaceIndex(path = "info.socialSecurity", type = SpaceIndexType.BASIC)
5         })
6         public Map<string, object=""> getInfo() {
7         return info;
8         }
```

Note: As of GigaSpaces 8.0 release the document model semantics has extended to support for the full hierarchy of Maps of Maps and dynamic indexes.

COMBINING SQL *QUERY WITH (GIGASPACES) NOSQL DATA STORE*

Now that we've gone through the API example let's see how we could insert this data into a NoSQL data store (GigaSpaces in this specific case) and query it through SQL.

INSERTING THE DATA OBJECT

```
1        for (long i=0;i<1000;i++)
2        {
3         gigaSpace.write(new Data(i,"message" + i, createInfo(i)));
4        }
```

The createInfo() generates a new Map and fill it with values as can be seen below:

```
1        public static Map<string, object=""> createInfo(long i)
2        {
3         Map< String, Object> info = new HashMap<string, object="">();
4         info.put("address", i + " Broadway");
5         info.put("socialSecurity", 1232287642L + i);
6         info.put("salary", 10000 + i);
7
8         return info;
9        }
```

QUERYING THE NoSQL DOCUMENT DATA USING SQL

There are basically two models for querying data using SQL in GigaSpaces. (1) Adding SQL-like queries using the GigaSpaces SQLQuery API (2) using a fully standard SQL JDBC driver.

QUERYING THE DATA USING A SQL-LIKE MODEL

```
1        Data[] d = gigaSpace.readMultiple(
2        new SQLQuery<data>(Data.class, "info.salary < 11000 and info.salary >=
3        10000"),
4        Integer.MAX_VALUE);
```

The gigaspaces.readMultiple(..) operation is equivalent to a “select” statement. It takes the class (the equivalent of the “from <table name>” clause in SQL), and query clause “info.salary < 11000 and info.salary >= 10000” (the equivalent of the where clause in SQL).

As we can see the syntax of the SQL query borrows the syntax of an object-oriented model where I can reference the associated attributes within the document attribute just like any nested attribute. In our example, info.salary would point to the info Map and pull the attribute who's key=”salary”.

This API is useful if you're already working in POJO as your domain model, as it works with objects natively and therefore can bypass the need for any O/R Mapping.

QUERYING THE DATA USING STANDARD JDBC

In this example we will illustrate how we can plug different query engines to the same data instance. In this case we will use a standard JDBC connection to connect to the data in the following way:

```
Connection conn;
Class.forName("com.j_spaces.jdbc.driver.GDriver").newInstance();
String url = "jdbc:gigaspaces:url:jini://*/*/myElasticDataGrid";
conn = DriverManager.getConnection(url);
Statement st = conn.createStatement();
String query = "SELECT * FROM com.gigaspaces.examples.Data";
ResultSet rs = st.executeQuery(query);
// Iterate through the result set
int i = 0;
while (rs.next()) {
System.out.println("Data [" + (i++) +"] "+ rs.getString("data") );
}
```

The first line sets up the GigaSpaces JDBC driver. The GigaSpaces JDBC driver is responsible for mapping the SQL query language into a the underlying GigaSpaces methods. That means that from a GigaSpaces data store perspective JDBC calls look pretty much the same as any other call. The URL uses the following pattern **<gigaspaces jdbc prefix>:<gigaspaces space url>,** where the gigaspaces-jdbc-prefix is always set to jdbc:gigaspaces. The space URL points to the relevant data grid cluster.

Note, that one of the interesting concepts that comes with this is that you don't need to point to a specific host as you would in most of today's databases, but rather we use "*" as the host name, which initiates a network discovery using multicast to find the relevant instances of the cluster.

The rest of the code looks just like any other SQL call. We use fairly basic mapping where every class is mapped to a table and and an object attribute is mapped to a column. Note, that unlike complex O/R mapping we leverage the fact that a space can store objects in their native format. That means that we don't need to break nested objects into different tables but instead we store them as single embedded Java object where the relationships are kept consistent with their original Java representation.

WHAT ABOUT PERFORMANCE?

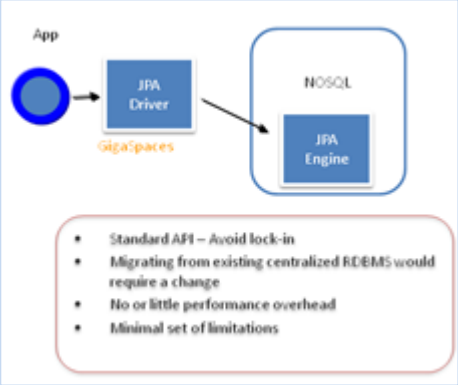
Alex Popescu, co-founder and CTO of InfoQ.com, made a comment in his post **NoSQL Databases Should Support SQL Queries** (http://nosql.mypopescu.com/post/835599437/nosql-databases-should-support-sql-queries) questioning the performance overhead associated with adding another layer of indirection, as I pointed out in my original post. He rightfully quoted Jeff Kesselman:

The two software problems that can never be solved by adding another layer of indirection are that of providing adequate performance or minimal resource usage.

Indeed, this is one of the main challenges in this entire discussion. It is relatively simple to add another level of indirection but it's almost impossible to make it perform well.

The key to addressing the performance challenge relies on the implementation of the underlying data store and how well it is suited to support the functionality required by the higher level abstraction. In our specific NoSQL discussion the performance of the SQL abstraction would be greatly influenced by the ability of the underlying NoSQL data store to support complex queries at the core level. In the case where the rich query engine is built into the core NoSQL data-store, adding a different set of query semantics would be a matter of simple syntax mapping which should yield negligible overhead and in some cases could turn out to be more efficient, as we could support algorithms that are not simple to implement at a lower level. A good example is found in Kevin Weil's presentation **Hadoop, Pig, and Twitter** (http://www.slideshare.net/kevinweil/hadoop-pig-and-twitter-nosql-east-2009). Kevin provides an example (slides 17-18) of how a simple Pig query could map to a fairly complex hadoop task in Java. So even if a lower level API could be more efficient at a micro level, it might turn out to be less efficient at the macro level as a result of the associated complexity.

In our example, the abstraction provided by putting Datanucleus on top of Google BigTable is probably going to yield fairly high overhead when it comes to complex queries, because most of the query semantics are implemented at the mapping layer. With GigaSpaces, we chose to support all the query semantics at the core layer and make the SQL abstraction a thin semantics mapping layer. A recent benchmark test showed only 2% difference between JDBC queries and native queries.



GigaSpaces Native Query engine makes the overhead of the SQL abstraction negligible

FINAL WORDS

In this paper I've tried to illustrate, mainly through the GigaSpaces example, how the SQL and NoSQL models can be brought together to achieve the best of both worlds, i.e., the scalability of the NoSQL model and the rich and standard query semantics of SQL. At the same time, I believe that additional semantics that are not currently supported by SQL such as the document model and object relationship can also be brought together as an extension to current SQL semantics as illustrated above.

However, there will be a time where an object or document-centric language is more intuitive or simpler to use then SQL. The decoupling of the datastore and query language should enable us to write objects in an Object/Document-centric model and still query it using a SQL engine and vice versa. This will give us flexibility to choose the language that best fits the context in which it is used, and mix and match languages just as we now do in any web application where we mix together different languages such as HTML, CSS, JavaScript and others.

Finally, I believe that large part of the discussion in the NoSQL community took a wrong turn by putting too much emphasis on the query language rather then the scalability patterns, which, in my opinion, should remain the only motivation for switching between one datastore to the other.

ABOUT GIGASPACES

GigaSpaces Technologies provides a new generation of application virtualization platforms. Our flagship product, eXtreme Application Platform (XAP), delivers end-to-end scalability across the entire stack, from the data all the way to the application. XAP is the only product that provides a complete in-memory solution on a single platform, enabling high-speed processing of extreme transactional loads. XAP was designed from the ground up to support any cloud environment – private, public, or hybrid – and offers a pain-free, evolutionary path from today's data center to the technologies of tomorrow.

Hundreds of organizations worldwide are leveraging XAP to enhance IT efficiency and performance. Among our customers are Fortune Global 500 companies, including top financial services enterprises, telecom carriers, online gaming providers, and e-commerce companies.

U.S. Headquarters
GigaSpaces Technologies Inc.
317 Madison Ave, Suite 823
New York, NY 10017
Tel: 646-421-2830
Fax: 646-421-2859

U.S. West Coast Office
GigaSpaces Technologies Inc.
101 Metro Drive, Suite 350
San Jose, CA 95110
Tel: 408-878-6982
Fax: 408-878-6149

International Office
GigaSpaces Technologies Ltd.
4 Maskit St., P.O. Box 4063
Herzliya 46140, Israel
Tel: +972-9-952-6751
Fax: +972-9-956-4410

Europe Office
GigaSpaces Technologies Ltd.
2 Sheraton St.
London, W1F 8BH, United Kingdom
Tel: +44-207-117-0213
Fax: +44-870-383-5135