By Craig Buckler (https://www.sitepoint.com/author/craig-buckler/)

SQL vs NoSQL: The Differences

Trending posts on SitePoint today:

7 Ways to Make WordPress Simpler for Users (https://www.sitepoint.com/make-wordpress-simpler-users/)
I Need a Website. What Do I Need to Know About Hosting? (https://www.sitepoint.com/i-need-a-website-what-do-i-need-to-know-about-hosting/)
WordPress Version Control with Git (https://www.sitepoint.com/wordpress-version-control-git/)
10 Easy WordPress Security Tips (https://www.sitepoint.com/10-easy-wordpress-security-tips/)
What Is the Right Image Format for Your Website? (https://www.sitepoint.com/what-is-the-right-image-format-for-your-website/)

SQL (Structured Query Language) databases have been a primary data storage mechanism for more than four decades. Usage exploded in the late 1990s with the rise of web applications and open-source options such as MySQL, PostgreSQL and SQLite.

NoSQL databases have existed since the 1960s, but have been recently gaining traction with popular options such as MongoDB, CouchDB, Redis and Apache Cassandra.

You'll find many tutorials explaining how to use a particular flavor of SQL or NoSQL, but few discuss why you should choose one in preference to the other. I hope to fill that gap. In this article, we'll cover the fundamental differences. In a later follow-up article, we'll look at typical scenarios and determine the optimal choice.

Most examples apply to the popular MySQL (https://www.mysql.com/) SQL and MongoDB (https://www.mongodb.org/) NoSQL database systems. Other SQL/NoSQL databases are similar, but there will be minor differences in features and syntax.

The SQL vs NoSQL Holy War

Before we go further, let's dispel a number of myths …

MYTH: NoSQL supersedes SQL
That would be like saying boats were superseded by cars because they're a newer technology. SQL and NoSQL do the same thing: store data. They take different approaches, which may help or hinder your project. Despite feeling newer and grabbing recent headlines, NoSQL is not a replacement for SQL — *it's an alternative*.

MYTH: NoSQL is better / worse than SQL
Some projects are better suited to using an SQL database. Some are better suited to NoSQL. Some could use either interchangeably. This article could never be a SitePoint Smackdown, because you cannot apply the same blanket assumptions everywhere.

MYTH: SQL vs NoSQL is a clear distinction
This is not necessarily true. Some SQL databases are adopting NoSQL features and vice versa. The choices are likely to become increasingly blurred, and NewSQL hybrid databases could provide some interesting options in the future.

MYTH: the language/framework determines the database
We've grown accustom to technology stacks, such as —

LAMP: Linux, Apache, MySQL (SQL), PHP
MEAN: MongoDB (NoSQL), Express, Angular, Node.js
.NET, IIS and SQL Server
Java, Apache and Oracle.

There are practical, historical and commercial reasons why these stacks evolved — but don't presume they are rules. You can use a MongoDB NoSQL database in your PHP (http://php.net/manual/en/class.mongodb.php) or .NET (http://docs.mongodb.org/ecosystem/drivers/csharp/) project. You can connect to MySQL (https://www.npmjs.com/package/mysql) or SQL Server (https://www.npmjs.com/package/mssql) in Node.js. You may not find as many tutorials and resources, but your requirements should determine the database type — *not the language*.

(That said, don't make life purposely difficult for yourself! Choosing an unusual technology combination or a mix of SQL and NoSQL is possible, but you'll find it tougher to find support and employ experienced developers.)

With that in mind, let's look at the primary differences …

SQL Tables vs NoSQL Documents

SQL databases provide a store of related data tables. For example, if you run an online book store, book information can be added to a table named `book`:

ISBN	title	author	format	price
9780992461225	JavaScript: Novice to Ninja	Darren Jones	ebook	29.00
9780994182654	Jump Start Git	Shaumik Daityari	ebook	29.00

Every row is a different book record. The design is rigid; you cannot use the same table to store different information or insert a string where a number is expected.

NoSQL databases store JSON-like field-value pair documents, e.g.

```
{
  ISBN: 9780992461225,
  title: "JavaScript: Novice to Ninja",
  author: "Darren Jones",
  format: "ebook",
  price: 29.00
}
```

Similar documents can be stored in a **collection**, which is analogous to an SQL table. However, you can store any data you like in any document; the NoSQL database won't complain. For example:

```
{
  ISBN: 9780992461225,
  title: "JavaScript: Novice to Ninja",
  author: "Darren Jones",
  year: 2014,
  format: "ebook",
  price: 29.00,
  description: "Learn JavaScript from scratch!",
  rating: "5/5",
  review: [
    { name: "A Reader", text: "The best JavaScript book I've ever read." },
    { name: "JS Expert", text: "Recommended to novice and expert developers alike." }
  ]
}
```

SQL tables create a strict data template, so it's difficult to make mistakes. NoSQL is more flexible and forgiving, but being able to store any data anywhere can lead to consistency issues.

SQL Schema vs NoSQL Schemaless

In an SQL database, it's impossible to add data until you define tables and field types in what's referred to as a *schema*. The schema optionally contains other information, such as —

primary keys — unique identifiers such as the ISBN which apply to a single record
indexes — commonly queried fields indexed to aid quick searching
relationships — logical links between data fields
functionality such as **triggers** and **stored procedures**.

Your data schema must be designed and implemented before any business logic can be developed to manipulate data. It's possible to make updates later, but large changes can be complicated.

In a NoSQL database, data can be added anywhere, at any time. There's no need to specify a document design or even a collection up-front. For example, in MongoDB the following statement will create a new document in a new `book` collection if it's not been previously created:

```
db.book.insert(
  ISBN: 9780994182654,
  title: "Jump Start Git",
  author: "Shaumik Daityari",
  format: "ebook",
  price: 29.00
);
```

(MongoDB will automatically add a unique `_id` value to each document in a collection. You may still want to define indexes, but that can be done later if necessary.)

A NoSQL database may be more suited to projects where the initial data requirements are difficult to ascertain. That said, don't mistake difficulty for laziness: neglecting to design a good data store at project commencement will lead to problems later.

SQL Normalization vs NoSQL Denormalization

Presume we want to add publisher information to our book store database. A single publisher could offer more than one title so, in an SQL database, we create a new `publisher` table:

id	name	country	email
SP001	SitePoint	Australia	feedback@sitepoint.com

We can then add a `publisher_id` field to our `book` table, which references records by `publisher.id`:

ISBN	title	author	format	price	publisher_id
9780992461225	JavaScript: Novice to Ninja	Darren Jones	ebook	29.00	SP001
9780994182654	Jump Start Git	Shaumik Daityari	ebook	29.00	SP001

This minimizes data redundancy; we're not repeating the publisher information for every book — only the reference to it. This technique is known as normalization, and has practical benefits. We can update a single publisher without changing `book` data.

We can use normalization techniques in NoSQL. Documents in the `book` collection —

```
{
  ISBN: 9780992461225,
  title: "JavaScript: Novice to Ninja",
  author: "Darren Jones",
  format: "ebook",
  price: 29.00,
  publisher_id: "SP001"
}
```

— reference a document in a `publisher` collection:

```
{
  id: "SP001"
  name: "SitePoint",
  country: "Australia",
  email: "feedback@sitepoint.com"
}
```

However, this is not always practical, for reasons that will become evident below. We may opt to denormalize our document and repeat publisher information for every book:

```
{
  ISBN: 9780992461225,
  title: "JavaScript: Novice to Ninja",
  author: "Darren Jones",
  format: "ebook",
  price: 29.00,
  publisher: {
    name: "SitePoint",
    country: "Australia",
    email: "feedback@sitepoint.com"
  }
}
```

This leads to faster queries, but updating the publisher information in multiple records will be significantly slower.

SQL: Relational JOIN vs NoSQL

SQL queries offer a powerful JOIN clause. We can obtain related data in multiple tables using a single SQL statement. For example:

```
SELECT book.title, book.author, publisher.name
FROM book
LEFT JOIN book.publisher_id ON publisher.id;
```

This returns all book titles, authors and associated publisher names (presuming one has been set).

NoSQL has no equivalent of JOIN, and this can shock those with SQL experience. If we used normalized collections as described above, we would need to fetch all `book` documents, retrieve all associated `publisher` documents, and manually link the two in our program logic. This is one reason denormalization is often essential.

SQL vs NoSQL Data Integrity

Most SQL databases allow you to enforce data integrity rules using foreign key constraints (unless you're still using the older, defunct MyISAM storage engine in MySQL). Our book store could —

> ensure all books have a valid `publisher_id` code that matches one entry in the `publisher` table, and
> not permit publishers to be removed if one or more books are assigned to them.

The schema enforces these rules for the database to follow. It's impossible for developers or users to add, edit or remove records, which could result in invalid data or orphan records.

The same data integrity options are not available in NoSQL databases; you can store what you want regardless of any other documents. Ideally, a single document will be the sole source of all information about an item.

SQL vs NoSQL Transactions

In SQL databases, two or more updates can be executed in a transaction — an all-or-nothing wrapper that guarantees success or failure. For example, presume our book store contained `order` and `stock` tables. When a book is ordered, we add a record to the `order` table and decrement the stock count in the `stock` table. If we execute those two updates individually, one could succeed and the other fail — thus leaving our figures out of sync. Placing the same updates within a transaction ensures either both succeed or both fail.

In a NoSQL database, modification of a single document is atomic. In other words, if you're updating three values within a document, either all three are updated successfully or it remains unchanged. However, there's no transaction equivalent for updates to multiple documents. There are transaction-like options (http://docs.mongodb.org/manual/core/write-operations-atomicity/), but, at the time of writing, these must be manually processed in your code.

SQL vs NoSQL CRUD Syntax

Creating, reading updating and deleting data is the basis of all database systems. In essence —

> SQL is a lightweight declarative language. It's deceptively powerful, and has become an international standard, although most systems implement subtly different syntaxes.
> NoSQL databases use JavaScripty-looking queries with JSON-like arguments! Basic operations are simple, but nested JSON can become increasingly convoluted for more complex queries.

A quick comparison:

<table>
<tr><th>SQL</th><th>NoSQL</th></tr>
<tr><td colspan="2">insert a new book record</td></tr>
<tr><td>

```
INSERT INTO book (
`ISBN`, `title`, `author`
)
VALUES (
'9780992461256',
'Full Stack JavaScript',
'Colin Ihrig & Adam Bretz'
);
```

</td><td>

```
db.book.insert({
ISBN: "9780992461256",
title: "Full Stack JavaScript",
author: "Colin Ihrig & Adam Bretz"
});
```

</td></tr>
<tr><td colspan="2">update a book record</td></tr>
</table>

SQL	NoSQL

`UPDATE book`
`SET price = 19.99`
`WHERE ISBN = '9780992461256'`
	`db.book.update(`
	`{ ISBN: '9780992461256' },`
	`{ $set: { price: 19.99 } }`
	`);`

return all book titles over $10	

`SELECT title FROM book`
`WHERE price > 10;`
	`db.book.find(`
	`{ price: { &gt;: 10 } },`
	`{ _id: 0, title: 1 }`
	`);`
	The second JSON object is known as a **projection**: it sets which fields are returned (`_id` is returned by default so it needs to be unset).

count the number of SitePoint books	

`SELECT COUNT(1) FROM book`
`WHERE publisher_id = 'SP001';`
	`db.book.count({`
	`"publisher.name": "SitePoint"`
	`});`
	This presumes denormalized documents are used.

return the number of book format types	

`SELECT format, COUNT(1) AS `total``
`FROM book`
`GROUP BY format;`
	`db.book.aggregate([`
	`{ $group:`
	`{`
	`_id: "$format",`
	`total: { $sum: 1 }`
	`}`
	`}`
	`]);`
	This is known as **aggregation**: a new set of documents is computed from an original set.

delete all SitePoint books	

`DELETE FROM book`
`WHERE publisher_id = 'SP001';`
Alternatively, it's possible to delete the `publisher` record and have this cascade to associated `book` records if foreign keys are specified appropriately.
	`db.book.remove({`
	`"publisher.name": "SitePoint"`
	`});`

SQL vs NoSQL Performance

Perhaps the most controversial comparison, NoSQL is regularly quoted as being faster than SQL. This isn't surprising; NoSQL's simpler denormalized store allows you to retrieve all information about a specific item in a single request. There's no need for related JOINs or complex SQL queries.

That said, your project design and data requirements will have most impact. A well-designed SQL database will almost certainly perform better than a badly designed NoSQL equivalent and vice versa.

SQL vs NoSQL Scaling

As your data grows, you may find it necessary to distribute the load among multiple servers. This can be tricky for SQL-based systems. How do you allocate related data? Clustering is possibly the simplest option; multiple servers access the same central store — but even this has challenges.

NoSQL's simpler data models can make the process easier, and many have been built with scaling functionality from the start. That is a generalization, so seek expert advice if you encounter this situation.

SQL vs NoSQL Practicalities

Finally, let's consider security and system problems. The most popular NoSQL databases have been around a few years; they are more likely to exhibit issues than more mature SQL products. Many (https://www.trustwave.com/Resources/SpiderLabs-Blog/Mongodb---Security-Weaknesses-in-a-typical-NoSQL-database/) problems (http://developer.olery.com/blog/goodbye-mongodb-hello-postgresql/) have (http://www.information-age.com/technology/security/123459001/major-security-alert-40000-mongodb-databases-left-unsecured-internet) been (http://www.theregister.co.uk/2015/07/21/drongo_mongodbs_spew_600_terabytes_of_unauthenticated_data/) reported (https://aphyr.com/posts/322-call-me-maybe-mongodb-stale-reads), but most boil down to a single issue: *knowledge*.

Developers and sysadmins have less experience with newer database systems, so mistakes are made. Opting for NoSQL because it feels fresher, or because you want to avoid schema design inevitably, leads to problems later.

SQL vs NoSQL Summary

SQL and NoSQL databases do the same thing in different ways. It's possible choose one option and switch to another later, but a little planning can save time and money.

Projects where SQL is ideal:

- logical related discrete data requirements which can be identified up-front
- data integrity is essential
- standards-based proven technology with good developer experience and support.

Projects where NoSQL is ideal:

- unrelated, indeterminate or evolving data requirements
- simpler or looser project objectives, able to start coding immediately
- speed and scalability is imperative.

In the case of our book store, an SQL database appears the most practical option — especially when we introduce ecommerce facilities requiring robust transaction support. In the next article, we'll discuss further project scenarios, and determine whether an SQL or NoSQL database would be the best solution.


(htt

Meet the author

Craig Buckler (https://www.sitepoint.com/author/craig-buckler/) (https://twitter.com/craigbuckler) (http://plus.google.com/+CraigBuckler) (http://www.facebook.com/craigbuckler) (http://www.linkedin.com/in/craigbuckler) (https://github.com/craigbuckler)

Craig is a freelance UK web consultant who built his first page for IE2.0 in 1995. Since that time he's been advocating standards, accessibility, and best-practice HTML5 techniques. He's written more than 1,000 articles for SitePoint and you can find him @craigbuckler (http://twitter.com/craigbuckler)