SQL vs NoSQL: How to Choose

Adapted from: http://www.sitepoint.com/sql-vs-nosql-choose/

SQL databases:

- store related data in tables
- require a schema which defines tables prior to use
- encourage normalization to reduce data redundancy
- support table JOINs to retrieve related data from multiple tables in a single command
- implement data integrity rules
- provide transactions to guarantee two or more updates succeed or fail as an atomic unit
- can be scaled (with some effort)
- use a powerful declarative language for querying
- offer plenty of support, expertise and tools.

NoSQL databases:

- store related data in JSON-like, name-value documents
- can store data without specifying a schema
- must usually be denormalized so information about an item is contained in a single document
- should not require JOINs (presuming denormalized documents are used)
- permit any data to be saved anywhere at anytime without verification
- guarantee updates to a single document — but not multiple documents
- provide excellent performance and scalability
- use JSON data objects for querying
- are a newer, exciting technology.

SQL databases are ideal for projects where requirements can be determined and robust data integrity is essential. NoSQL databases are ideal for unrelated, indeterminate or evolving data requirements where speed and scalability are more important. In simpler terms:

- **SQL is digital.** It works best for clearly defined, discrete items with exact specifications. Typical use cases are online stores and banking systems.
- **NoSQL is analog.** It works best for organic data with fluid requirements. Typical use cases are social networks, customer management and web analytics systems.
Few projects will be an exact fit. Either option could be viable if you have shallower or naturally denormalized data. But please be aware these simplified example scenarios with sweeping generalizations! You know more about your project than I do, and I wouldn’t recommend switching from SQL to NoSQL or vice versa unless it offers considerable benefits. It’s your choice. Consider the pros and cons at the start of your project and you can’t go wrong.

Scenario One: a Contact List

Let’s re-invent the wheel and implement an SQL-based address book system. Our initial naive contact table is defined with the following fields:

- id
- title
- firstname
- lastname
- gender
- telephone
- email
- address1
- address2
- address3
- city
- region
- zipcode
- country

*Problem one:* few people have a single telephone number. We probably need at least three for land-line, mobile and workplace, but it doesn’t matter how many we allocate — someone, somewhere will want more. Let’s create a separate telephone table so contacts can have as many as they like. This also normalizes our data — we don’t need a NULL for contacts without a number:

- contact_id
- name *(text such as land-line, work mobile, etc.)*
- number
Problem two: we have the same issue with email addresses, so let’s create a similar **email** table:

- contact_id
- name *(text such as home email, work email, etc.)*
- address

Problem three: we may not wish to enter a (geographic) address, or we may want to enter multiple addresses for work, home, holiday homes, etc. We therefore need a new **address** table:

- contact_id
- name *(text such as home, office, etc.)*
- address1
- address2
- address3
- city
- region
- zipcode
- country

Our original **contact** table has been reduced to:

- id
- title
- firstname
- lastname
- gender

Great — we have a normalized database which can store any number of telephone numbers, email addresses and addresses for any contact. Unfortunately …

**The schema is rigid**

We’ve not considered the contact’s middle name(s), date of birth, company or job role. It doesn’t matter how many fields we add, we’ll soon receive update requests for notes, anniversaries, relationship statuses, social media accounts, inside leg measurements,
favorite type of cheese etc. It's impossible to foresee every option, so we'd possibly create an otherdata table with name-value pairs to cope.

**The data is fragmented**

It's not easy to for developers or system administrators to examine the database. The program logic will also become slower and more complex, because it's not practical to retrieve a contact's data in a single SELECT statement with multiple JOIN clauses. *(You could, but the result would contain every combination of telephone, email and address: if someone had three telephone numbers, five emails and two addresses, the SQL query would generate thirty results.)*

Finally, full-text search is difficult. If someone enters the string “SitePoint”, we must check all four tables to see if it's part of a contact name, telephone, email or address and rank the result accordingly. If you've ever used WordPress's search, you'll understand how frustrating that can be.

**The NoSQL Alternative**

Our contact data concerns people. They are unpredictable and have differing requirements at different times. The contact list would benefit from using a NoSQL database, which stores all data about an individual in a single document in the contacts collection:

```json
{
    name: ["Billy", "Bob", "Jones"],
    company: "Fake Goods Corp",
    jobtitle: "Vice President of Data Management",
    telephone: {
        home: "0123456789",
        mobile: "9876543210",
        work: "2244668800"
    },
```
email: {
    personal: "bob@myhomeemail.net",
    work: "bob@myworkemail.com"
},
address: {
    home: {
        line1: "10 Non-Existent Street",
        city: "Nowhere",
        country: "Australia"
    }
},
birthdate: ISODate("1980-01-01T00:00:00.000Z"),
twitter: '@bobsfakeaccount',
note: "Don't trust this guy",
weight: "200lb",
photo: "52e86ad749e0b817d25c8892.jpg"
}

In this example, we haven’t stored the contact’s title or gender, and we’ve added data which need not apply to anyone else. It doesn’t matter — our NoSQL database won’t mind, and we can add or remove fields at will.

Because the contact’s data is contained in a single document, we can retrieve some or all information using a single query. A full-text search is also simpler; in MongoDB we can define an index on all contact text fields using:

```
db.contact.createIndex({ "$**": "text" });
```

then perform a full-text search using:

```
db.contact.find({
    $text: { $search: "something" }
});
```
Scenario Two: a Social Network

A social network may use similar contact data stores, but it expands on the feature set with options such as relationship links, status updates, messaging and "likes". These facilities may be implemented and be dropped in response to user demand — it's impossible to predict how they will evolve.

In addition:

- Most data updates have a single point of origin: the user. It's unlikely we'll need to update two or more records at any one time, so transaction-like functionality is not required.
- Despite what some users may think, a failed status update is unlikely to cause a global meltdown or financial loss. The application’s interface and performance take a higher priority than robust data integrity.

NoSQL appears to be a good fit. The database allows us to quickly implement features storing different types of data. For example, all the user's dated status updates could be placed in a single document in the status collection:

```json
{
  user_id: ObjectId("65f82bda42e7b8c76f5c1969"),
  update: [ 
    {
      date: ISODate("2015-09-18T10:02:47.620Z"),
      text: "feeling more positive today"
    },
    {
      date: ISODate("2015-09-17T13:14:20.789Z"),
      text: "spending far too much time here"
    },
    {
      date: ISODate("2015-09-17T12:33:02.132Z"),
      text: "considering my life choices"
    }
  ]
}
```
While this document could become long, we can fetch a subset of the array, such as the most recent update. The whole status history for every user can also be searched quickly.

Now presume we wanted to introduce an emoticon choice when posting an update. This would be a matter of adding a graphic reference to new entries in the `update` array. Unlike an SQL store, there’s no need to set previous message emoticons to NULL — our program logic can show a default or no image if an emoticon isn’t set.

Scenario Three: a Warehouse Management System

Consider a system which monitors warehoused goods. We need to record:

- products arriving at the warehouse and being allocated to a specific location/bay
- movements of goods within the warehouse, e.g. rearranging stock so the same products are in adjacent locations
- orders and the subsequent removal of products from the warehouse for delivery.

Our data requirements:

1. Generic product information such as box quantities, dimensions and color can be stored, but it’s discrete data we can identify and apply to anything. We’re unlikely to be concerned with specifics, such as laptop processor speed or estimated smartphone battery life.
2. It’s imperative to minimize mistakes. We can’t have products disappearing or being moved to a location where different products are already being stored.
3. In its simplest form, we’re recording the transfer of items from one physical area to another — or removing from location A and placing in location B. That’s two updates for the same action.

We need a robust store with enforced data integrity and transaction support. Only an SQL database will (currently) satisfy those requirements. The best advice: expose yourself to as many technologies as possible. That knowledge will allow you to make a reasoned and emotionally impartial judgment regarding SQL or NoSQL.