Intro' to NoSQL

WHAT? Why NoSQL? Why not SQL? Is SQL going away??

What does 'No' stand for?

NoSQL DB means/meant:

- Non relational, non SQL
- NO SQL
- NotOnly SQL
- ..

SQL is dead! Long live SQL!

Relational DBs held sway for almost 3 DECADES: 80s, 90s, 00s.

What started to change was this - the Internet, coupled with storage and processing revolution has enabled:

- **BigUsers** - LOTS of people using databases
- **BigData** - LOTS of data being generated
- **BigVariety** - there is a huge VARIETY in the types of data being stored and searched
- **BigFlux** - there is rapid CHANGE in a lot of data being handled

RDBMSs are simply unsuitable for the above! A different kind of NON-RELATIONAL database scheme was needed - enter 'NoSQL' DBs.

Need for NoSQL

Again - what is happening is this:

- lots of new data, new types of data, are being rapidly generated
- developers are finding it hard to 'shoehorn' all this data into a relational model
- also hard to scale up to fit more data, more users
- and, hard to keep up performance too

Need a flexible, efficient, available, scalable solution/DB design! THAT is what NoSQL provides - high performance, high availability at a large scale.

NoSQL - history

The term NoSQL was used as early as.. 1998!

The term started to become reused in 2009, by Last.fm's developer, and subsequently by a Rackspace employee blogger who popularized it.

Today, NoSQL refers to an umbrella of technologies that are all non-relational-DB-oriented.

NoSQL DBs

A NoSQL DB is:

- schema-less: no tables, no relations!
- flexible: easy to add new types of data
- (data) scalable: specifically, ability to 'scale out', ie. do 'horizontal scaling' - both terms means that we can simply add more nodes (eg. servers) to an existing cluster, to accommodate more users, or to add more data to existing users.
- fast: easy to process large (massive) volumes of data

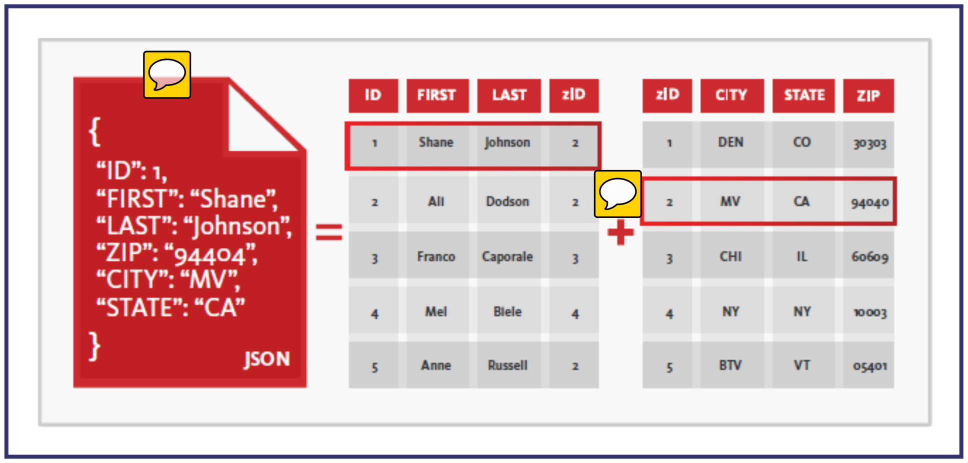
Drivers of the NoSQL movement

What led to NoSQL starting to get adopted?

- Avoidance of complexity [eg. no need to worry about immediate consistency]
- High throughput
- Easy, quick scalability
- Ability to run on commodity hardware
- Avoidance of need for O-R mapping
- Avoidance of complexity associated with cluster setup
- Realization that 'one-size-fits-all' was wrong [Stonebraker]
- Ability to use DB APIs for various programming languages, that mirror the languages' own structures

JSON (or XML) for storing an ENTIRE db!!

db == JSON []array of {}objects, where each object ('row') is a set of a key-value pairs.



```
{
 "ID": 1,
 "FIRST": "Shane",
 "LAST": "Johnson",
 "ZIP": "94404",
 "CITY": "MV",
 "STATE": "CA"
}
```

JSON

=

ID	FIRST	LAST	zID
1	Shane	Johnson	2
2	Ali	Dodson	2
3	Franco	Caporale	3
4	Mel	Biele	4
5	Anne	Russell	2

+

zID	CITY	STATE	ZIP
1	DEN	CO	30303
2	MV	CA	94040
3	CHI	IL	60609
4	NY	NY	10003
5	BTV	VT	05401

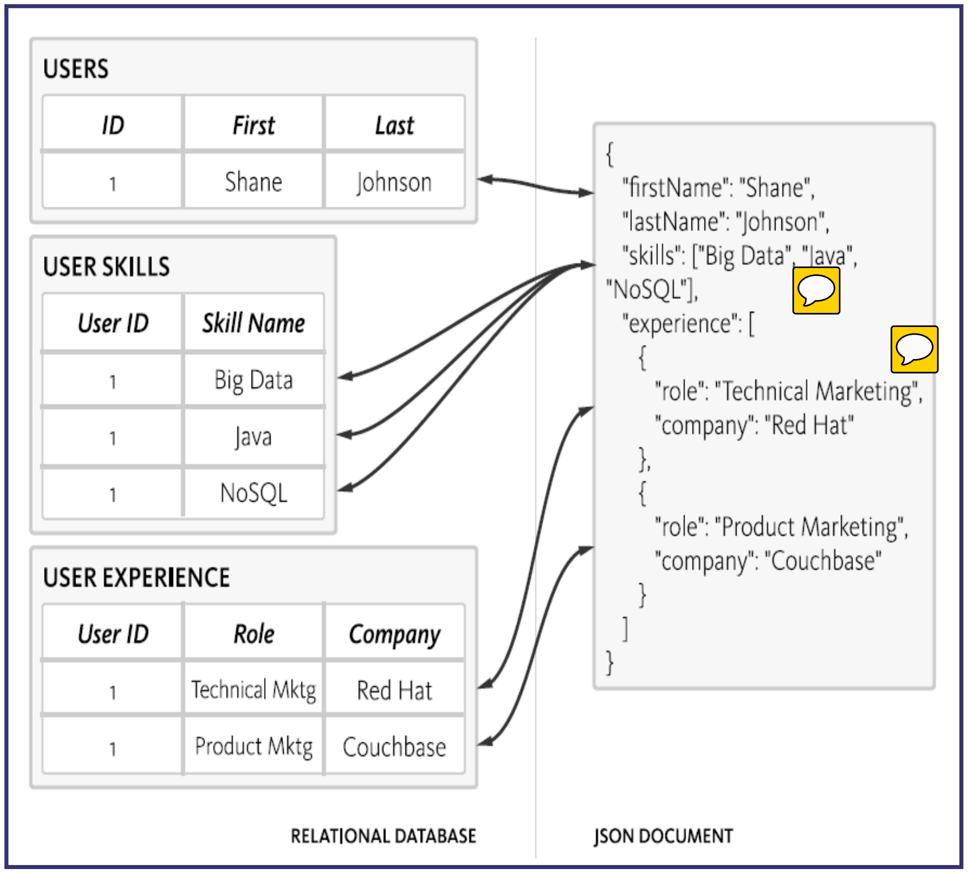
http://www.data.gov has almost 200,000+ datasets, 50,000+ of which are in JSON.

Example: https://health.data.ny.gov/api/views/es3k-2aus/rows.json

The simplicity of the JSON scheme also makes it possible/easy to add extra data (eg metadata) to the files.

RDB -> JSON

It is very straightforward to represent table data as JSON:



USERS

ID	First	Last
1	Shane	Johnson

USER SKILLS

User ID	Skill Name
1	Big Data
1	Java
1	NoSQL

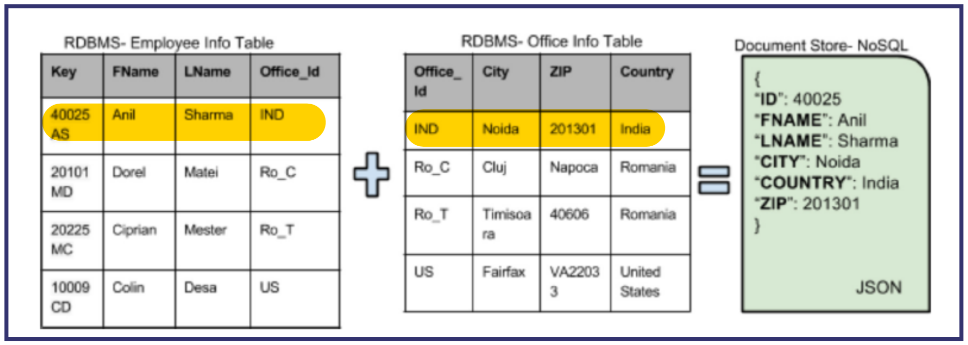
USER EXPERIENCE

User ID	Role	Company
1	Technical Mktg	Red Hat
1	Product Mktg	Couchbase

```
{
  "firstName": "Shane",
  "lastName": "Johnson",
  "skills": ["Big Data", "Java",
"NoSQL"],
  "experience": [
    {
      "role": "Technical Marketing",
      "company": "Red Hat"
    },
    {
      "role": "Product Marketing",
      "company": "Couchbase"
    }
  ]
}
```

RDB -> JSON, second example

Joined data again:



RDBMS- Employee Info Table

Key	FName	LName	Office_Id
40025 AS	Anil	Sharma	IND
20101 MD	Dorel	Matei	Ro_C
20225 MC	Ciprian	Mester	Ro_T
10009 CD	Colin	Desa	US

RDBMS- Office Info Table

Office_ Id	City	ZIP	Country
IND	Noida	201301	India
Ro_C	Cluj	Napoca	Romania
Ro_T	Timisoa ra	40606	Romania
US	Fairfax	VA2203 3	United States

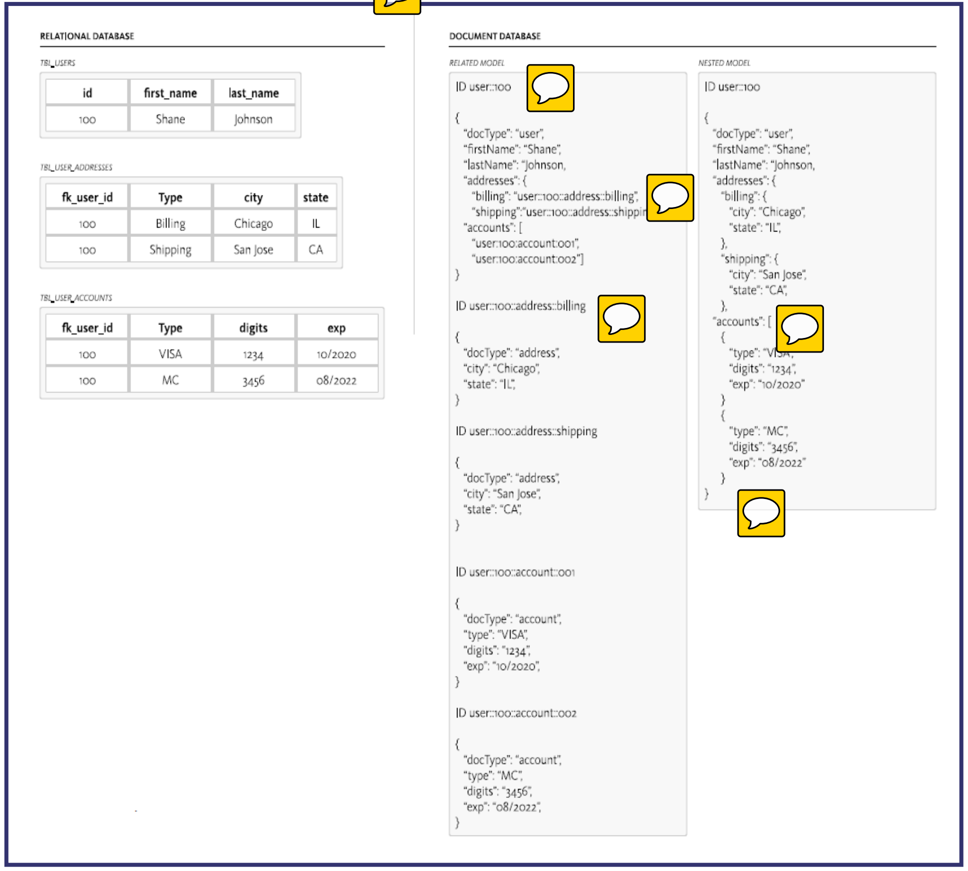
Document Store- NoSQL

```
{
"ID": 40025
"FNAME": Anil
"LNAME": Sharma
"CITY": Noida
"COUNTRY": India
"ZIP": 201301
}
```

JSON

RDB to JSON, third example

Note - we also have the choice of using a 'related model' (where a JSON value is contained in a related (pointed-to) structure), instead of a 'nested model' (representing joined data):



RELATIONAL DATABASE

TBL_USERS

id	first_name	last_name
100	Shane	Johnson

TBL_USER_ADDRESSES

fk_user_id	Type	city	state
100	Billing	Chicago	IL
100	Shipping	San Jose	CA

TBL_USER_ACCOUNTS

fk_user_id	Type	digits	exp
100	VISA	1234	10/2020
100	MC	3456	08/2022

DOCUMENT DATABASE

RELATED MODEL

```
ID user::100

{
  "docType": "user",
  "firstName": "Shane",
  "lastName": "Johnson,
  "addresses": {
    "billing": "user::100::address::billing",
    "shipping":"user::100::address::shippi
  "accounts": [
    "user:100:account:001",
    "user:100:account:002"]
}

ID user::100::address::billing

{
  "docType": "address",
  "city": "Chicago",
  "state": "IL",
}

ID user::100::address::shipping

{
  "docType": "address",
  "city": "San Jose",
  "state": "CA",
}


ID user::100::account::001

{
  "docType": "account",
  "type": "VISA",
  "digits": "1234",
  "exp": "10/2020",
}

ID user::100::account::002

{
  "docType": "account",
  "type": "MC",
  "digits": "3456",
  "exp": "08/2022",
}
```

NESTED MODEL

```
ID user::100

{
  "docType": "user",
  "firstName": "Shane",
  "lastName": "Johnson,
  "addresses": {
    "billing": {
      "city": "Chicago",
      "state": "IL",
    },
    "shipping": {
      "city": "San Jose",
      "state": "CA",
    },
  "accounts": [
    {
      "type": "V[illegible]
      "digits": "1234",
      "exp": "10/2020"
    }
    {
      "type": "MC",
      "digits": "3456",
      "exp": "08/2022"
    }
}
```

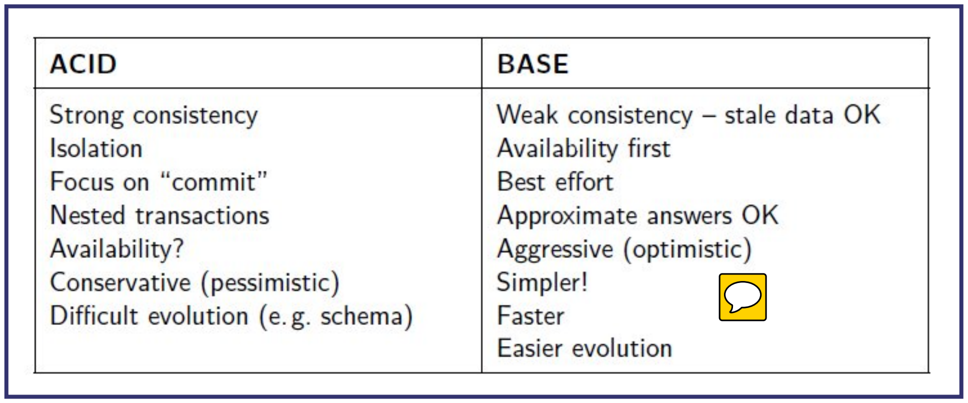
An XML db example

Compared to JSON, XML is more verbose (on account of opening and closing tags), but is a popular alternative format for creating DBs. Here is an example file (in addition to viewing the data here in the browser, you can use a viewer such as this one to view the XML file, or you can save locally and open with a text editor).

BASE, not ACID

The NoSQL DBs are characterized by their 'BASE' property, in contrast with RDBMS' 'ACID' property.

BASE stands for BAsic availability (db is up most of the time), Soft state (of consistency - consistency is not guaranteed while data is written to a node, or between replicas), Eventual consistency (at a later point in time, by push or pull, data will become consistent across nodes and replicas).



ACID	BASE
Strong consistency	Weak consistency – stale data OK
Isolation	Availability first
Focus on “commit”	Best effort
Nested transactions	Approximate answers OK
Availability?	Aggressive (optimistic)
Conservative (pessimistic)	Simpler!
Difficult evolution (e.g. schema)	Faster
	Easier evolution

'Schema-less'

A NoSQL database does not have an explicit schema that describes the relationships between its data items.

Calling this situation 'schema-less' is not quite appropriate - rather, the schema is **implicit** - the schema resides in the application code that reads and writes data.

To put it differently, the DB itself "doesn't care" about what it is storing, it is the application code that imparts meaning to the data. As a result, changing the data model (eg. adding or deleting an attribute) is trivial - just write and run (application) code to make the change in the DB!

In a schema-less environment, developers use intuitive data structures (well supported by underlying host languages) to do data manipulation (including querying and updating).

Eg. JSON is very easy/intuitive to grasp, comprising of just six underlying datatypes (number, string, boolean, array, object, null). It is also quite easy to parse.