Get InfluxDB

# Why You Should Migrate from SQL to NoSQL for Time Series Data

## An overview of SQL vs. NoSQL databases

Most developers and engineers working with time series data today do so using noSQL databases. Still, there's a significant number using relational database management systems (RDBMS) for time series data. This web page explains the differences between SQL and noSQL databases in general as well as the specific differences for time series use cases.

One of the most important differences between SQL and noSQL is the schema, or lack thereof. The relational model means you can't create a SQL database without a predefined schema, where the data is normalized into t designated, and all the relationships between columns, ro

There are two disadvantages to this approach.

- You must define your schema ahead of time before you can begin creating tables or entering data.
- You cannot change your mind about your schema later.

Both can cause frustration, especially point 2: any kind of change in the structure of your database will be both difficult and disruptive.

NoSQL databases don't have this problem. The majority of noSQL databases have dynamic schemas, so the schema doesn't need to be defined up-front. As such, it's possible to create documents without a predefined structure, and to create a different structure for each document.

Another benefit of noSQL databases is the increased potential for scalability. SQL databases use table structures while noSQL databases typically use key-value stores. This means that SQL-based databases scale vertically (scaling them up requires a beefier server, more processing power, etc.) while noSQL databases tend to scale horizontally (scaling up involves some combination of sharding and adding more servers). Horizontally-scaled systems have the potential to become much more powerful than vertically-scaled ones overall: you'll hit an upper bound on how much RAM/CPU/GPU/etc. you can add to one server, but additional servers can tile out much farther.

All that being said, noSQL data stores have some downsides as a result of their distributive nature. NoSQL databases follow the CAP Theorem which states that it is impossible for a distributed database to simultaneously provide consistency, availability, and partition tolerance. However, InfluxDB is an exception. InfluxDB is a hybrid non-strict CP/AP.

Finally, the most significant downside of noSQL data stores comes from their lack of history compared with SQL ones.

Structured Query Language has been around for about 40 years now. As a result, it's the basis for a lot of databases, both relational (ex. MySQL, SQL Server) and non-relational (ex. Postgres). Because all these different databases — each with many thousands of users — use SQL, it's extremely stable and well-supported. By contrast, it can be difficult to find expert developers for many noSQL databases, and the selection of third-party consultants to help with complex implementations is more limited.

Because of this, it's critical for whatever noSQL database you wind up using to have excellent community support, because that's what you will end up relying on in the absence of many consultants.

# Using SQL vs. NoSQL for time series use cases

Gaining high performance for time series from a SQL database requires significant customization and configuration. Without that, unless you're working with a very small dataset, a SQL-based database will simply not work properly. As an example, un-configured PostgreSQL's ingest rate will dip significantly at 100M+ total dataset rows, and become near-unusable at 400M+ rows.

Configuring a SQL database — or, for that matter, any non-time-series-optimized database (i.e. MongoDB, Cassandra, etc.) — for time series data is not trivial. On the other hand, as a purpose-built time series database, InfluxDB comes equipped to work with time series data right out of the metaphorical box.

Time series data is unique. Time series data is almost always monitoring data and is usually collected to determine if a system, host, environment, patient, etc. is healthy. In order to optimize for time series use cases, several assumptions and tradeoffs were made during the design of InfluxDB. For example, data is added in time-ascending order. Additionally, when deletes occur, it’s typically against a large range of old data. These assumptions enable InfluxDB to have incredibly high query and write performance — and led to several design decisions that promote InfluxDB’s high performance. InfluxDB anticipates and supports high cardinality use cases by storing the index on disk. Furthermore, it is written in Go for fast and concurrent data fetching with an easy binary deploy. Finally, it is worth noting that InfluxDB is an exception in the NoSQL world because it is a hybrid non-strict CA/AP database — it is consistent, available, and partition-tolerant.

SQL databases are typically focused on CRUD — creating, reading, updating, and deleting data, all in equal measure. In addition, they're typically designed with the ACID principles (Atomicity, Consistency, Isolation, Durability) in mind. NoSQL, on the other hand, tends to be looser with these principles: for example, InfluxDB is focused primarily on creating and reading data, and much less on updates and deletes to specific rows.

Each database type comes with tradeoffs that are optimized for their use cases. Since time

series data comes in time order and is typically collected in real-time, time series databases are immutable and append-only to accomodate for extremely high volumes of data. The append-only property of time series databases distinguishes time series databases from relational databases, which are optimized for transactions but only accommodate lower ingest volumes. In general, depending on their particular use case, noSQL databases will trade off the ACID principles for BASE model (whose principles are Basic Availability, Soft State and Eventual Consistency). For example, one individual point in a time series is fairly useless in isolation, and the important thing is the trend in aggregate.

One of the barriers to choosing a noSQL database is the difficulty of learning a non-SQL query language and performing complex data transformations. Fortunately, InfluxDB has Flux lang — an easy-to-learn, standalone data scripting and query language that increases productivity and code reuse. Users can easily perform data transformation and apply complicated math with intuitive syntax.

For example, to query our data, “`my_data`”, from our bucket, “`my_bucket`”, while specifying a measurement, “`my_measurement`” for the last 7 days, we would use the following Flux query:

```
from(bucket: "my-bucket")
|> range(start: v.timeRangeStart, stop: v.timeRangeStop)
|> filter(fn: (r) => r["_measurement"] == "my_measurement")
|> filter(fn: (r) => r["_field"] == "my_data")
```

Flux enables users to interact with time series in a table format, giving InfluxDB a relational feel. With Flux, users can perform cross-measurement joins, pivots, maps, and math across measurements. InfluxDB Templates and user defined Flux packages make the barrier to Flux adoption extremely low. InfluxDB Templates prepackage configurations that contain everything from Flux queries, to dashboards and Telegraf configurations, to notifications and alerts — thereby enabling users to use Flux before they even know how to write it. User defined Flux packages allow users to write and share custom-functions that wrap up complicated Flux scripts. Flux provides a powerful way for working with data. Here’s where you can learn more about why InfluxData is building Flux.

Even though data structures vary between noSQL and SQL databases, almost all of the above query should feel intuitive to the SQL developer, except for maybe the bucket. A bucket is where time series data is stored. All buckets have retention policies. A retention policy describes how long InfluxDB keeps data. InfluxDB compares your local server’s timestamp to the timestamps on your data and deletes data that is older. Automatically

expiring and downsampling old data is critical for time series use cases because of the high-volume nature of time series problems. However, creating and enabling these recurring tasks are challenging in a SQL environment.

Specifically, SQL developers would need to write code to shard the data across the cluster, aggregate and downsampling functions, data eviction and lifecycle management, and summarization. They'd also have to create an API to write and query their new service. Furthermore, developers using Cassandra or HBase will need to write tools for data collection. They'll need to introduce a real-time processing system and write code for monitoring and alerting. Finally, they'll need to write a visualization engine to display the time series data to the user. All of these functionalities are already provided with InfluxDB.

548 Market St, PMB 77953
San Francisco, California 94104

Contact Us

**InfluxData**

About
Careers
Partners
Legal
Newsroom
Contact Sales

## Sign Up for the InfluxData Newsletter

Email*

I have read the privacy policy.

**Submit**