InfluxDB: A NoSQL Database

[Anonymous]

ABSTRACT

NoSQL databases are non relational databases that provides storage and retrieval of data without using tabular relations. They are built for specific data models and have flexible schema.[1] NoSQL databases are optimized for applications that handle large amounts of data and require low latency and flexible data models. There are five major types of NoSQL databases which are used depending on the type of application.[2] These are:

- *Key-value*:
  Key-value databases can be partitioned and allows horizontal scaling of data. The advantage of this type of database is that other databases do not provide the same amount of horizontal scaling.
- *Document based*:
  These types of databases make it easy to store and query data by using the same document model which was used in the application code. By being flexible, semi-structured and hierarchical, this type of database evolves with the needs of the application.
- *Graph based*:
  A graph database makes it easy to create and execute applications that have many connected data-sets. These databases are used for fraud detection, recommendation engines like Google, etc.
- *In-memory*:
  This database is used for real time analysis that require micro second response times. It provides low latency and high throughput.
- *Search based*:
  This database is used by logging applications that need to provide real time visualizations and analyse machine generated data.

In this paper we analyse a NoSQL database, InfluxDB. InfluxDB is Time Series based database that is a type of Key-vaue NoSQL. Written in Go, InfluxDB is optimized for fast retrieval of real time series data.[3] InfluxDB has no external dependencies. Queries are similar to SQL and has built in time centric functions for querying data like measurements, series and points. A point is made up of many key-value pairs field-set and timestamp (i.e [field-set, timestamp]). A group of these key-value pairs form a tag-set. A group of tag-sets define a series. These series are then grouped together by a string identifier which is used for measurement. InfluxDB listens on port 8086 and accepts data from HTTP, TCP and UDP. Input to the database is provided using a line protocol. The syntax of a line protocol is:
*measurement(,tag_key=tag_val)**
*field_key=field_val(,field_key_n=field_value_n)**
(nanoseconds-timestamp)?

Keywords

InfluxDB, Flux, TSI, TSM, Telegraf

1. INTRODUCTION

Time series is a collection of data points that are ordered by the time in which they are measured.[4] The data is measured in real time over equally spaced points in time. The measurements are added (appended) to the database without replacing the previous stored values. This means that unlike RDBMS, the data in a time series NoSQL database performs only INSERT operations and not UPDATE operations.

For example suppose we want to view the weather data at a particular weather station in real time. The data (i.e, station name, weather condition, minimum temperature, maximum temperature, etc) is obtained and recorded as a new entry with an index of the time at which it was recorded. After a specific amount of time, a new value is obtained and appended to the database with a new index of the time it was recorded. This implies that each newly obtained data has a different time stamp. This feature of only INSERT new data with different time stamps instead of updating the database is powerful as we can perform various analysis such as measuring the changes of weather in the past and predicting the future weather at that particular weather station. By comparing this data with other obtained data from other weather stations, we can compare the weather at different weather stations measured at the same time as well as predict the occurrence of various weather patterns over the next few days based on the movement of weather between each weather station.

The advantage of using time series databases are that these databases can handle large data-sets. By having an index only on the timestamp, various efficient algorithms result in faster query processing, good data compression's, efficiently insert data into the database and improves the performance. Time series databases can also perform various time based data analysis such as data retention policies, flexible time aggregation, continuous queries, etc.[5]

RDMS on the other hand struggles to handle large databases, which results in a huge drop in efficiency. They are also not able to perform time based data analysis.

However since the data is only INSERT in a time series database, the amount of available data tends to quickly pile up. Hence a major disadvantage is that a large amount of storage is required while using a time based database.

2. WHAT IS A TIME SERIES DATABASE?

Time Series Databases mainly improves the efficiency on two major requirements in modern day databases, high throughput and faster query processing. Throughput is the amount of data generated. Since real time data quickly piles up, RDBMs find it difficult to efficiently handle all the data. However large amounts of data can be easily handled by Time Series Database. The Time Series Database also concentrates on recently inserted data to improve query processing. This is done because in most cases only the recently inserted data is required.[6]

3. WHY DO WE NEED A TIME SERIES DATABASE?

In the past, most companies were dependent on static big data. This was because most databases were used either to obtain static data like current amount in a bank, checking whether a username is valid, etc. However in the modern world, most applications rely on processing huge amounts of real time data.[7] For example, Google Maps require real time data in order to measure current traffic trends in a particular area and to predict the amount of time required by the user to get to their destination. Some other examples of where TSDBs are useful are:

- Monitoring software systems like various resources used by applications, servers, etc.
- Financial operations like the stock market, cryptocurrency trends, etc
- Tracking applications like finding the location of physical machines

The most important application is IOT. With the rapid growth in IOT the need for Time Series Databases which handle Time Series specific functions have increased drastically evidently from the graphs given below.

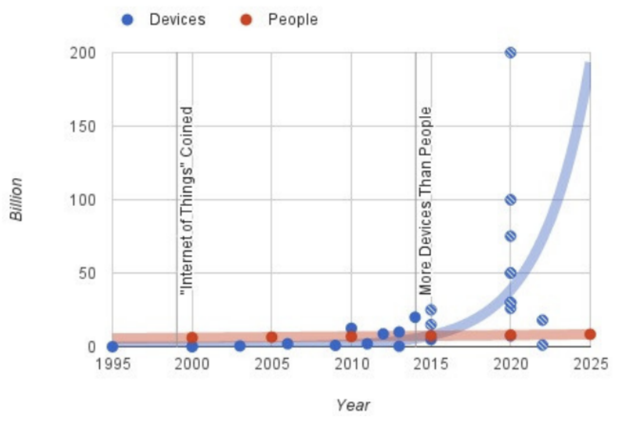


Figure 1: IOT Trends [12]

According to the ranking by *DB-Engines Ranking* in 2016, the popularity of Time Series is rising the most.

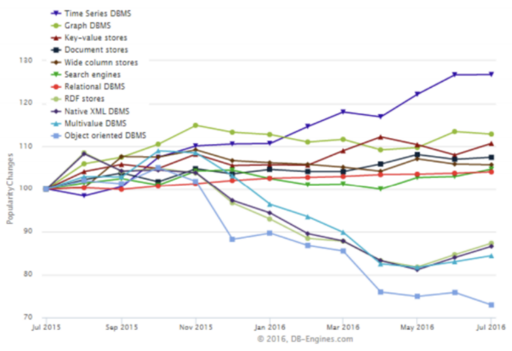


Figure 2: Popularity changes [12]

4. INFLUXDB

As of November 2019, Influx DB is ranked the highest by *https://db-engines.com* on the basis of Number of mentions of the system on websites, General interest in the system,Frequency of technical discussions about the system,Number of job offers, Number of profiles in professional networks, Relevance in social networks.

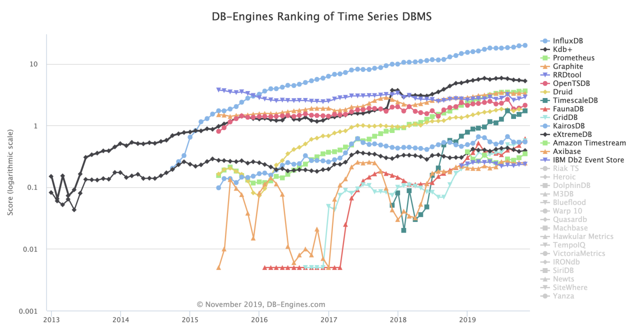


Figure 3: DB Engine popularity [13]

InfluxDB is an open-source schemaless time series database with optional closed-sourced components developed by InfluxData.[12] The open source the TICK Stack , provides a full time series database platform with various services including the InfluxDB core and can run on cloud and on premises on a single node.[12] Though we are demonstrating the use and features on InfluxDB in this report. The TICK stack together serves as a powerful tool for developers dealing with extensive Time Series data. It is made specifically for time series data and hence gives better write and read throughput. The developers of InfluxDB realized that the community working with Time series data had to perform a set of common tasks to work with Time series data and hence InfluxDB came into existence to serve as a platform to handle these common tasks and to give better performance and efficiency.

4.1 Telegraf

Telegraf is a plugin-driven server agent for collecting and sending metrics and events from databases, systems, and IoT sensors. [14] This is a powerful tool with which we can convert data of one multiple format into a line protocol. Line protocol is the format in which data is fed to influx db.

While writing the data, a timestamp must be associated with it. If a timestamp is not present, the InfluxDB automatically assigns a timestamp using the current local time. Input to InfluxDB is in the form of Line Protocol. A line protocol is of the format :

Measurement	Tagset	Field_set	Timestamp

- Measurement: Measurement name given by the user
- Tagset: All the key value pair for the data point. Both the key and the value should be of type string.
- Field_set: This is also a key value pair. It however differs from the Tagset as the value here is a measurement which can change at any time.
- Timestamp: The timestamp related to the data point. The timestamp uses the UNIX nanosecond timestamp format or the ts-epoch format. This field indicates the time at which the data point was collected.

We can collect multiple data of the same timestamps, but they will always belong to different Measurements. For example 2 data entries can be collected from both cities *atlanta* as well as *rochester*

4.2 InfluxDB

It is the database at the core of TICK stack. It can be queried with the use of an SQL like Query language called the InfluxQl which is powerful in dealing with time series data. Each data series needs to have a timestamp associated with it. Further details can be found in the rest of the report.

4.3 Chronograf

This module serves as a GUI for interpreting and visualizing the data. As influx can work with real time data, Chronograf plots and visualizations make sense of the data in real time.

4.4 Kapacitor

Kapacitor helps to create alerts, run ETL jobs and detect anomalies on our realtime database. [15]

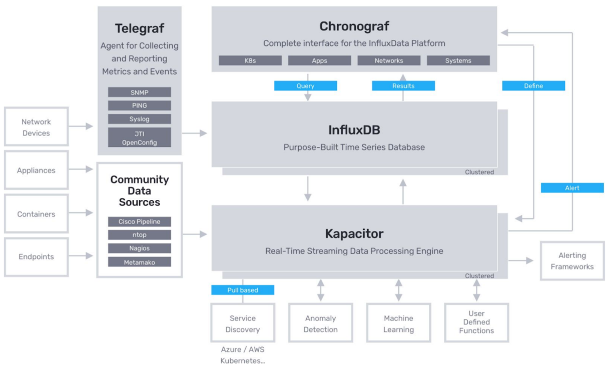


Figure 4: TICK stack [13]

5. DATA STORAGE COMPARISON

The way data is handled and stored in a NoSQL database is different from the way data is handled and stored in an RDBMS. To explain this we describe in detail how data is stored in an SQL followed by how data is stored in a InfluxDB database.

5.1 Storing Data in SQL

Data in SQL is stored in:

- Records
- Pages

5.1.1 Records

A record is the smallest storage structure in SQL. Each row of a table, the indexes, metadata, etc are all stored as an individual records on the disk. However, the most important record is the data record.[8]

Data records have a *fixedvar* format. A *fixedvar* format consists of both fixed length data as well as variable length data.

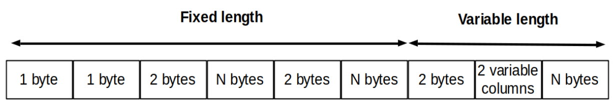


Figure 5: Data Record

The first 2 bytes of a data record define the record type, whether a NULL bitmap exists and whether any variable length data is present. The next 4 bytes define the total length of the fixed length data. The next 4 bytes define which columns have NULL values and which columns do not have NULL values. The next two bytes store the variable length columns, variable length offset values, and variable length data. The offset values help the SQL server to find the end data for each column. This helps the server in calculating the length of each column as well as querying the data.

5.1.2 *Pages*

Storing data records sequentially in a large data file would reduce efficiency. To overcome this, an SQL server stores the records in smaller units known as pages. These pages are handled by the buffer manager and are cached in memory. Different types of pages store different types of data. Some pages store only data records while certain other pages store only index records. However the structure of each page is consistent. [9]

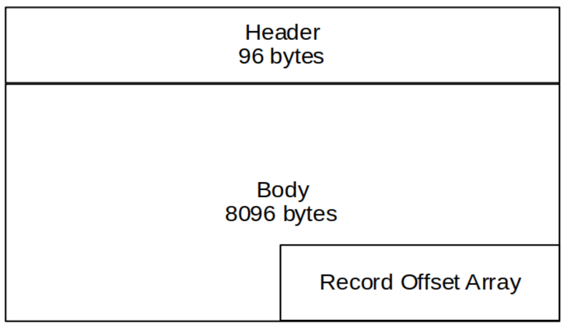


Figure 6: Page Structure

All pages have a fixed size of 8KB. Each pages is divided into two parts: The header and the body. The header has a fixed size of 96 bytes while the body has a fixed size of 8096 bytes. The header consists of information on the amount of free space left, the number of records stored in the page, the object to which the page belongs and, an index to find the location of other pages.The body consists of data as well as a record offset array. The header of this array defines the number of values the server can read.

5.2 Storing Data in InfluxDB

Data in InfluxDB is mainly stored in one of the following ways:

- Write Ahead Log (WAL)
- Time Structured Merge Trees (TSM)
- Cache

Each database in InfluxDB has a set of its own WAL and data files.

5.2.1 *Write Ahead Log (WAL)*

WAL stands for "Write Ahead Log". The Write Ahead Log provides Atomicity and Durability (two of the four ACID properties in Databases). In the WAL, changes are first written to a log with the data being immutable. Any change to the data, including deletions, is also recorded on the log file as a mutation. Once the WAL reaches a certain size or meets a certain condition, the data is written to the database.

Data in the WAL is first compressed using Snappy compression. This is an entropy encoder technique developed by Google. The data in the WAL are written as and when WRITE and DELETE operations arrive. These files are WRITE optimized which ensures quick execution of WRITE and DELETE operations. However, since the files are WRITE optimized, they are not optimized for READ operations.

WAL has an extension of ".wal". Logs are initially labelled as "_0001.wal". Each time a new WAL is created, the number is incremented. When the size of a WAL reaches 10MB, it is closed and a new WAL is used to record incoming data. These logs also record modifications and DELETE in the form of compressed blocks. Whenever the database sends in a WRITE request, the request is applied to serialized data points. These serialized points are in turn compressed using the Snappy compression technique and are written to the WAL.

In order for a WRITE operation to be successful a function called *fsync*. This leads to data being added to an in memory index before a success message is returned. However it is inefficient to execute this process for each WRITE request. Hence batch processing is used to ensure efficiency. In InfluxDB, a typical batch size has 5,000 to 10,000 data points per WRITE process.

Records in Write Ahead Logs (WAL)

Records in a WAL follows a *type-length-value (TLV)* format. Each record consists of three blocks:

- *Type*: This block is of fixed length. The length varies from 1 to 4 bytes and indicates the type of data present in the record.
- *Length*: This block is also of fixed length. The length varies from 1 to 4 bytes and indicates the size of the value field.
- *Value*: This block is of variable length. It contains the data stored in the record.

Records which follow the TLV format is an advantage as it helps in parsing data quickly. The format of a TLV record is as follows:

- *Type*: This field has a fixed length of 1 byte. It represents two data types.
- *Write*: This field indicates whether the record is to be written.
- *Delete*: This field indicates whether the record is to be deleted.

- *Length*: This field has a fixed length of 4 bytes. It contains a 32-bit integer that indicates the length of the record.
- *Value*: This field contains the compressed data block.

5.2.2 Time Structured Merge Trees (TSM)

TSM files help InfluxDB to quickly read data. It also ensures that the data does not occupy too much of space.

A TSM file is a read only file that has the following format:

Header	Blocks	Index	Footer
5 bytes	N bytes	N bytes	4 bytes

Figure 7: TSM file format

Header

The header has a total size of 5 bytes. Out of the total size, 4 bytes is used to record the file type. The 4 bytes is also informally called a "Magic Number". The remaining 1 byte is used to record the version number of the file.

Header	
Type 4 bytes	Version 1 byte

Figure 8: Header format

Blocks

A main block consists of many sub-blocks. Each sub-block has a data cell, where data is stored, and a CRC value.

Blocks					
Block 1		Block 2		Block N	
CRC 4 bytes	Data N bytes	CRC 4 bytes	Data N bytes	CRC 4 bytes	Data N bytes

Figure 9: Blocks format

Index

The index is similar to the WAL. However, unlike the WAL, all data is indexed and written directly to the disk. The index consists of:

- *TagBlocks*: Maintains an index on a single tag key.
- *MeasurementBlock*: Maintains an index on the measurements and their respective tag keys.
- *Trailer*: Stores the offset information.

Indexes are stored in the lexicographical order of key and then time. A key consists of a measurement name, a tag and several fields as shown in *Figure 4*.

Index								
Key Len 2 bytes	Key N bytes	Type 1 bytes	Count 2 bytes	Min Time 8 bytes	Max Time 8 bytes	Offset 8 bytes	Size 4 bytes	...

Figure 10: Index format

In *Figure 6*, we see that Key Len and Key are TagBlocks. Count, Min Time, and Max Time are MeasurementBlocks. Offset and Size are the Trailer.

Type stores the data type of the block. There are 4 block types:

- *Float*: Float is an implementation based on Facebook Gorilla Paper.
- *Int*: Any value less than -1 are encoded in simple8b encoding. Any value greater than -1 are left uncompressed.
- *Bool*: 1 bit is used as a Boolean value
- *String*: Strings are compressed using the Snappy compression method.
- *Min Time*: This value indicate the smallest timestamp present in the block.
- *Max Time*: This value indicate the highest timestamp present in the block.

5.2.3 Cache

Cache stores all the points that are currently in the Write Ahead Log. Points are indexed by a key. This key is a combination of measurement, tagset and a unique fieldset. The cache is never compressed and once the system is turned off, the data in the cache is lost.

The cache is extremely important for the querying process. A query causes data from the cache to be merged with data in TSM files. This helps in executing a WRITE process as soon as a query is parsed. While querying the cache, only TSM files are modified and not the WAL. Due to this, querying can be executed in parallel to the writing process.

The removal of a log file depends on the cache. The cache has two parameters:

- Cache-snapshot-memory-size:
  When the cache size exceeds the Cache-snapshot-memory-size, a snapshot of the cache is taken. The corresponding log files are then deleted from the cache and combined with the data in TSM files.
- Cache-max-memory-size:
  When cache size exceeds Cache-max-memory-size, all WRITE requests are aborted. This allows the cache to take a snapshot.

6. INDEXING COMPARISONS

As with storing data, the way NoSQL and SQL databases perform indexing also varies. In the following sections we describe how SQL performs indexing followed by how InfluxDB performs indexing and why it is better.

6.1 Indexing data in SQL

Data in SQL can be indexed by using one of the following methods:

- Heaps
- Indexes

6.1.1 Heaps

Heaps are a group of pages owned by the same object. An *Index Allocation Map (IAM)* tracks which pages belong to which object. Each IAM page contains one large bitmap. To increase efficiency, the IAM tracks a group of pages instead of each individual page. These groups of pages are known as *extents*. In order for the server to find a requested page, it finds the first IAM page and then follows each extent untill it finds the page.

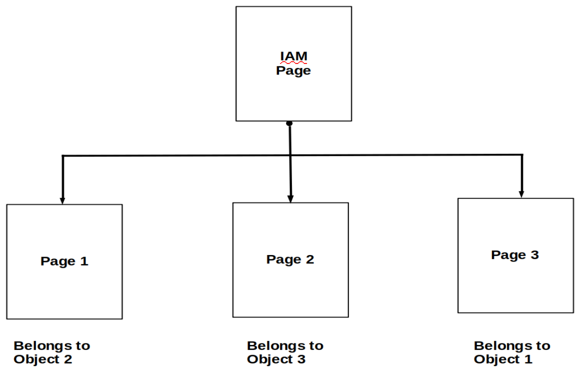


Figure 11: Heap

6.1.2 Indexes

Index pages are stored in a b-tree structure. Indexes can be divided into clustered and non-clustered indexes. The main difference between these two indexes is that while unclustered indexes store key values in the nodes and data values in the leaf nodes, clustered indexes store both the key as well as the data in the leaf nodes. Due to this, each table can have only 1 clustered index. For unclustered indexes, the pointer can either point to the physical location of a record or create a copy of a clustered index.

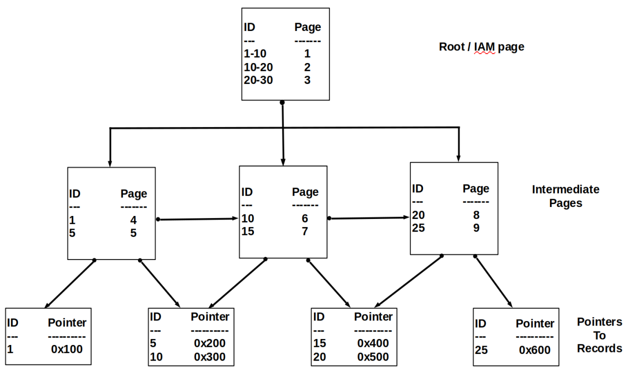


Figure 12: SQL Index

6.2 Indexing data in InfluxDB

InfluxDB performs data indexing using one of the following types of indexing:

- Time Series Indexing (TSI)
- Time Structured Merge Tree (TSM)

6.2.1 Time Series Indexing (TSI)

Time Series Indexing shifts index data to the disk memory and maintains a map for all the data that is pulled up from the operating systems page cache. However since the number of index entries are usually large, storing indexes in the main memory causes the main memory to run out of space.

To overcome this, routines are constantly executed, which compacts the index files and ensure that query processing does not slow down. InfluxDB also uses a RobinHood hashing method in order to speed up index lookups.

Time Series Indexing is turned off by default and the user needs to activate it in order to use it.

6.2.2 Time Structured Merge Tree (TSM)

Time Structured Merge Tree uses an in-memory data structure. This index consists of a composite key made up of measurement name, tag, and field metadata. Every unique combination has a look-up table in the memory to map the metadata to the time series. This causes a delay in start-up time as the data has to be loaded onto a heap.

Compaction

Compaction is the process of moving data from a write-only format to read-only format. This process has several stages:

- *Snapshot*: When cache values exceed the Cache-snapshot-memory-size, they are combined with appropriate logs and merged with TSM files.
- *Level Compaction*: When a snapshot is compacted with a TSM file, it is called a Level 1 compaction. When there are several Level 1 TSM files, they can be further compacted to form Level 2 TSM files. Compacted TSM files can go up to Level 4 using this hierarchical compaction. As the level of compaction increases, the compression ratio of data also increases.
- *Index Optimization*: When there are a large number of Level 4 TSM files, it is difficult to access them via indexes. Hence an Index Optimization process is created, where the indexes and the series associated with them is added to separate TSM files.
- *Full compaction*: A shard is a container for time series data. When a shard has not been written for a long time, Full Compaction produces an optimal set of TSM files that queries the data efficiently. Full Compaction TSM files are not further compacted unless new WRITE or DELETE requests is made on data present in the TSM files.

7. CRUD VS CR

SQL supports CRUD (create read update delete). However, InfluxDB is mainly C and R (create and read). Though

there are ways to delete data, there are no specific methodology to update data in InfluxDB. One can overwrite the entire table if there needs to be an update. However, a developer cannot go to a particular timestamp and edit data. People might describe this as weakness. However, depending on the kind of purpose of the application, this can be an advantage.
In today's world with advent of sensor driven data and need for statistical analysis on bulk data. There is very little need to update or delete data that is recorded from sensors. For example, data collected from devices like fitbits help organizations understand the various health parameters of a person. This sort of data (heart monitor, calories burnt etc.) is read from sensors and there is no need to update them. Thus, InfluxDB is suited for applications where bulk data has to be recorded and used for further analysis.

8. QUERY PROCESSING

InfluxDB versions 1.x use a query language called the Influx Functional Query Language(IFQL). IFQL was developed specifically for the ease of handling Time Series Database(TSDB) in the InfluxDB.

8.1 What is Influx Query Language(IFQL)?

Influx Query Language (IFQL) is used to store and analyse the data for InfluxQL.[11] It's syntax is very similar to SQL and hence a new user does not take a lot of time to get well versed with it. The syntax is specified using Extended Backus-Naur Form (EBNF). The notations of IFQL in the increasing order of precedence is:

- | : alternation
- () : grouping
- [ ] : option (0 or 1 times)
- {} : repetition (0 to n times)

8.2 Queries

A query consists of one or more statements delimited by a semicolon. Some example queries, unique to InfluxDB are:

- CREATE RETENTION POLICY:
  Retention policy describes how long the data persists in the InfluxDB.

```
policy_stmt = "CREATE RETENTION POLICY" policy_name
                policy_duration
                policy_replication
                [ policy_shard_group_duration ]
                [ "DEFAULT" ]
```

- CREATE SUBSCRIPTION:
  Subscriptions are local or remote endpoints which contains all data written to the database.

```
sub = "CREATE SUBSCRIPTION" sub_name "ON"
       db_name "."
       retention_policy "DESTINATIONS" ("ANY"|"ALL")
       host { "," host}
```

- SHOW DIAGNOSTICS:
  Displays node information. For example, uptime, hostname, server configuration, memory usage, etc

```
diag = "SHOW DIAGNOSTICS"
```

- SHOW MEASUREMENTS:
  Displays the data stored in the associated fields

```
measure_stmt = "SHOW MEASUREMENTS" [on_clause]
               [ measurement_clause ]
               [ where_clause ]
               [ limit_clause ] [ offset_clause ]
```

- SHOW SERIES: Displays data defined by shared measurement, tag set, and field key.

```
series_stmt = "SHOW SERIES" [on_clause]
              [ from_clause ] [ where_clause ]
              [ limit_clause ] [ offset_clause ]
```

- SHOW SHARDS:
  Displays the actual encoded and compressed data

```
shards_stmt = "SHOW SHARDS"
```

- SHOW STATS:
  Displays detailed statistics on available components

```
stats_stmt = "SHOW STATS
             [ FOR '<component>' | 'indexes' ]"
```

- DROP SHARD:
  Deletes a shard from the database

```
shard_stmt = "DROP SHARD" ( shard_id )
```

- DROP SUBSCRIPTION: Deletes a subscription from the database

```
sub_stmt = "DROP SUBSCRIPTION" subscription_name
           "ON" db_name "." retention_policy
```

8.3 Life Cycle of a Query

The life cycle of a typical query in IFQL is:

- The query is tokenised and parsed into an Abstract Syntax Tree (AST).
- The QueryExecutor obtains the AST and sends it to its appropriate handlers. For example, SELECT queries are handled by the shards themselves.
- The Query Engine determines which shard match the query and forms iterators for each field in the query
- The iterators are sent to the Emitter which first groups them into points. It then converts the points into complex objects and returns them to the client

8.4 Rules to follow when writing Queries in Influx Querying Language (IFQL)

Like every other language, IFQL has its own set of rules on how to write quries. These rules are:

- When using the command line interface, all queries require the target database to be specified. This can be specified using the tag

```
        -use <database name>
```

- Key words must not be used as identifiers
- All values in IFQL are case sensitive. For example, *exampleDB* $\neq$ *exampledb*

8.5 Why use IFQL and not SQL?

Since IFQL syntax is similar to SQL, some queries are shared between the two. However there are some major differences.
Since SQL queries are geared towards static data, they are not well equipped to handle continuous data input. One of the major disadvantages in SQL is that the performance drops when handling large amounts of data. Real time continuous data also quickly piles up data, swiftly increasing the amount of data stored on the disk in just a matter of seconds. Without fine tuning SQL to suit each databases need, the efficiency in handling queries is low.
On the other hand, IFQL can easily handle large amounts of data without any drop in performance or efficiency. IFQL was also designed to handle time centric functions, making it very easy to handle real time continuous data. A special identifier *fill()*, unique to IFQL, fills in valid points when points do not exist while grouping by intervals.

8.6 Query Execution Time

According to the paper published by Syeda Noor Zehra Naqvi et al. from Universite Libre de Bruxelles,[12] a series of tests were performed to check query execution time. Queries were either aggregation based or didn't use aggregation and each category had three subtypes:

- Range query
- Equality query
- Scan the entire database

They found the following execution times when the above queries were run on a database containing 30 million rows.

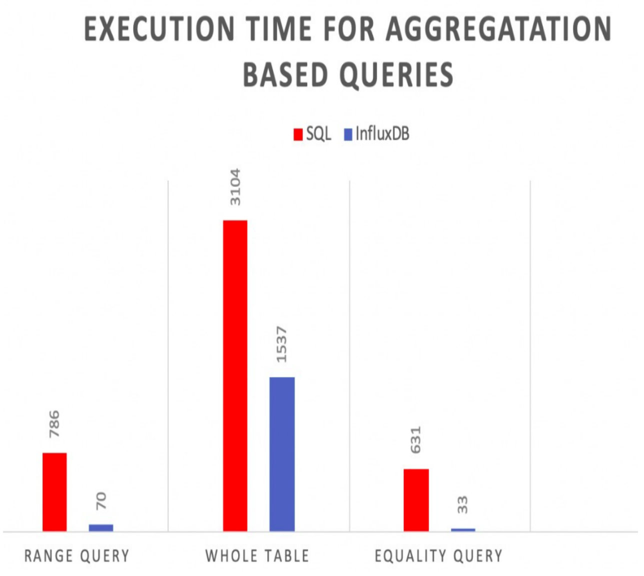


Figure 13: Aggregate Queries

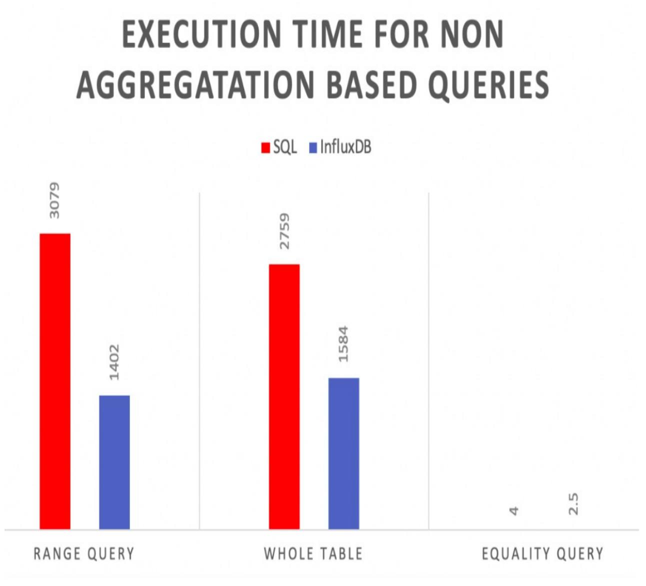


Figure 14: Non Aggregate Queries

9. COMPARISON BETWEEN INFLUXDB AND RDBMS

Although the syntax of IFQL and SQL look similar, the queries and inner working of the databases themselves vary greatly. These differences are found in the following areas:

- Schema definition
- Querying in relative time
- Time to live
- Index change
- Creating tables and importing data
- GROUP BY time duration
- Stream processing and IOT

9.1 Schema definition

RDBMS try to establish relationships between various data points. Hence a user is required to plan ahead and create a Schema. The Schema contains information on how to reduce redundancy of data, the purpose of a database, the primary key of a table, relationships between tables, etc. While designing the Schema, users have to keep in mind the requirements of the database and define relationships like one-to-many, many-to-many, one-to-one, etc; and also create tables with respect to normalization rules.
However all this complex planning is not required in InfluxDB. The database automatically handles the Schema without the user worrying about creating a Schema. However, there are certain guidelines the developer must follow while creating tables and fields:

- Use tags for metadata
- Maintain one data per tag

9.2 Querying in relative time

Querying with respect to time is an important feature when dealing with real time continuous data. Since InfluxDB easily handles time centric functions, this feature is easy to execute. However it is not very easy to do the same on RDBMS.

9.2.1 Relative Time queries in SQL

The following functions return time related data in SQL

- GETDATE
- GETUTCDATE
- CURRENT_TIMESTAMP

The above methods return tuples with respect to time. However we need to define additional variables as well.

- Query in SQL using Date range

```
SELECT MEAN(humidity) FROM atlanta
WHERE time >= CAST(DATEADD
(DAY, -1, GETDATE()) as DATE)
AND time < CAST(GETDATE() as DATE)
```

- Query in SQL using Relative time

```
set @now = getdate()
set @lastweek = dateadd(day,-7,@now)
SELECT MEAN(humidity)
FROM atlanta
WHERE time BETWEEN @lastweek AND @now
```

The above queries can only select a range between two dates and not a specific time. It cannot be used to query for hours and minutes relative to present time.

9.2.2 Relative Time queries in InfluxDB

InfluxDB supports relative time with respect to "now()". The default value of now() is set to the current time. However its value can be changed to any timestamp.
As an example, consider

```
SELECT MEAN("humidity") FROM atlanta
WHERE time < now()
```

This query displays the mean humidity from the measurement "Atlanta" and selects all data records before the current Unix timestamp.
We can also tweak the value of now() to any minute or hour. For example,

```
SELECT MEAN("humidity") FROM atlanta
WHERE time < now() - 21d
```

displays results from 21 days before the current time.

```
SELECT MEAN("humidity") FROM atlanta
WHERE time < now() - 6h
```

displays results from 6 hours before the current time.

```
SELECT MEAN("humidity") FROM atlanta
WHERE time < now() - 5m
```

displays results from 5 minutes before the current time.
The same tweaks can be used in aggregate functions like MEAN()

```
SELECT MEAN("humidity") FROM atlanta
WHERE time <= '2019-07-16T14:34:00.000Z'
```

displays mean humidity of all records from days before the mentioned timestamp.

```
SELECT MEAN("humidity") FROM atlanta
WHERE time <= '2019-07-16T14:34:00.000Z' - 2h
```

displays mean humidity of all records from 2 hours before the mentioned timestamp.

```
SELECT MEAN("humidity") FROM atlanta
WHERE time <= '2019-07-16T14:34:00.000Z' + 6m
```

displays mean humidity of all records from 6 minutes before the mentioned timestamp.

9.3 Time to live

SQL does not automatically delete old data. The developer has to explicitly create a query which performs this task.

```
declare @my_duration datetime
set @my_duration = DATEADD(DAY, -14, GETDATE())
DELETE FROM my_table
WHERE time < @my_duration
```

InfluxDB however has a flexible automated script that decides when to delete old data. The developer needs to specify the retention policy only once by running a simple query.

```
CREATE RETENTION POLICY <retention_policy_name>
ON <database_name> DURATION <duration>
```

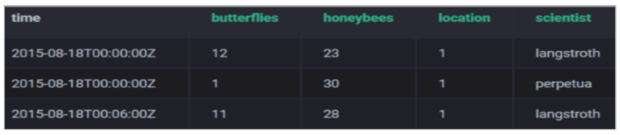
The duration can be set to any time, from as low as 1 hour to a maximum of infinity (inf)

9.4 Index change

Indexes in RDBMS cannot be easily altered. They have to be dropped and recreated again. The "ALTER INDEX" SQL query can only change a definition of the index and not the index itself.
However in InfluxDB, only the tag key is indexed and not the field key. Field keys and tag keys and easily be exchanged using a simple statement

```
SELECT * INTO B FROM A GROUP BY FIELD_NAMES
```

In the above statement, A is the field measurement and B is the new tag. The advantage of thi process is that the old table is not deleted. Consider the following table



time	butterflies	honeybees	location	scientist
2015-08-18T00:00:00Z	12	23	1	langstroth
2015-08-18T00:00:00Z	1	30	1	perpetua
2015-08-18T00:06:00Z	11	28	1	langstroth

We can convert the fields to tags using the following query

```
SELECT * INTO census_new FROM census
GROUP BY butterflies, honeybees
```

The two fields from the measurement "census" are converted into tags in a new measurement called "census_new".

9.5 Creating tables and importing data

Importing a CSV file into an RDBMS is complicated as the developer has to create a table and define the rows and columns. The data in the CSV file must strictly adhere to the data types defined for the particular column. Any error in the CSV file will cause the entire operation to abort.

```
BULK INSERT my_table
FROM 'filepath'
WITH
(
    FIELDTERMINATOR = ',',
    ROWTERMINATOR = '\n'
)
```

However in InfluxDB, tables are automatically created after importing data from a CSV file. The following command create the table automatically

```
influx -use <database name>
-import -path=<yourfilepath> -precision=ns
```

9.6 GROUP BY time duration

In SQL, GROUP BY is based on unique values. Performing GROUP BY on continuous data is not supported. This is because SQL aggregates every unique element in the column. If we have 10 different timestamps, the group by will aggregate using those 10 unique values. There is no possibility of mentioning a range and finding the mean in that range.
However, InfluxDB can display results grouped by any time duration. For example,

```
SELECT MEAN("humidity") FROM atlanta
WHERE time >= '2019-07-16T14:34:00.000Z'
AND time <= '2019-07-17T14:34:00.000Z'
GROUP BY time(5h)
```

The query returns the mean humidity for every 5 hour duration from 14th July 2019 to 17th July 2019. The time duration for group by can be in terms of hours, minutes, days, seconds and microseconds.

9.7 Stream Processing and IOT

Modern day data management systems have to process data on the fly. Application performance monitoring and infrastructure monitoring have high amounts of data coming in that have to be processed quickly. Processing data from sensors can be done easily as InfluxDB is built to store data with respect to time where timestamps are the primary key. While SQL was not built for stream processing, it requires SQLStream to be added to SQL to achieve streaming capabilities.
InfluxDB can directly store incoming stream data from IOT sensors and when combined with Flux it can be used to provide real time analysis.
The ability to add data to InfluxDB from sensors is done through three steps:

- Sensor data is stored as a continuous log using an agent.
- inputs.logsparser file takes this continuous log file as an input file and searches for data points as defined in the grok pattern
- Input.logparser.grok file enables the input file data to be converted to a format that adheres to line protocol.
- The file outputs.influxdb takes the output from inputs.logparser and feeds in into the port defined for InfluxDB (8086)
- InfluxDB adds the incoming data to the database.

The agent is used to define a configuration of the incoming data. One can define variables like jitter and incoming data interval.
The inputs.logparser file requires us to mention the input file to parse, the fieldnames of the measurement (to which we plan to add the data) and set the pattern to grok pattern.
The outputs.influxdb file gives us the URL to which we push the data to (http://127.0.0.1:8086 or http://localhost:8086, this can be an external url where the data is housed in an external server) and other user information like username and password.
The data present in the continuous log file looks like this:

```
1570130669.852521 Temp=24.8 C and Humidity=64.1%
and CPU_Temp=50.5 and ot=11.6 and oh=93.0%

1570130674.022393 Temp=24.9 C and Humidity=63.5%
and CPU_Temp=50.5 and ot=11.6 and oh=93.0%

1570130678.148942 Temp=24.8 C and Humidity=64.2%
and CPU_Temp=49.9 and ot=11.5 and oh=93.0%

1570130682.303456 Temp=25.0 C and Humidity=64.1%
and CPU_Temp=50.5 and ot=11.6 and oh=93.0%
```

The time stamp values mentioned above are fed to telegraph which converts to line protocol and pushes to InfluxDB. Thus, making it simple to push live sensor data continuously to InfluxDB.
In SQL there is currently no stack that is built specifically to push live continuous sensor data directly to SQL. This makes InfluxDB better suited to handle data generated from IOT platforms.

10. TRANSACTION MANAGEMENT

Transaction support is almost non existent in InfluxDB. Since InfluxDB is written entirely in Go, a relatively new programming language, it has had to develop the transaction tools from scratch instead of relying on already tested tools that other databases use. Hence it does not provide the same amount of transaction support as other databases. It does however attempt to follow the ACID properties to guarantee data integrity.

- *Atomicity*: Atomicity is maintained by using the Write Ahead Log. Since all changes are written into the log before being executed, all operations in a transaction is guaranteed to be executed.
- *Consistency*: InfluxDB uses its own language, Flux, to ensure database consistency.
- *Isolation*: Isolation is provided by the Compaction Planner. This planner ensures that each transaction does not hinder another transaction.

- *Durability*: InfluxDB provides durability by using snapshots as a backup in case of a power failure. However since data is always changing, in rare cases, a backup of a large volume of data could cause a mix up of timestamps causing a different order of data when compared to the original order of data.

11. SECURITY SUPPORT

InfluxDB provides various mechanisms in order to protect the data stored in its servers. These mechanisms are:

- *Authentication*:
  The database is password protected so that unauthorized users cannot access the data. Authentication requires that at least one admin account is created. The database authenticates a user with either Basic Authentication or URL query parameters. Basic authentication is described in RFC 2617 and is the preferred method for authentication. The other method requires query parameters to be provided in the URL/request body.
- *Authorization*:
  Users access to data are restricted based on their user type and privileges. There are two types of users, Admin and Non-admin. Admin users can READ, WRITE and access all databases. They can manage both the database as well as create users and set their privileges. Non-admin users can READ and WRITE only in the database that they have been assigned to. They do not have access to any other database unless an Admin user explicitly grants them the privilege to access another database.
- *Using HTTPS*:
  InfluxDB uses HTTPS to secure the communication between the server and the client. HTTPS in InfluxDB supports the following TLS certificates:
  Single domain certificate signed by Certificate Authority: With this certificate, every database requires a unique certificate.
  Wildcard domain certificate signed by Certificate Authority: With this certificate, multiple databases can be used on multiple servers.
  Self signed certificate: These certificates are similar to Single Domain certificates. The only difference is that the client cannot verify the identity of the database
  HTTPS can also be used for authentication purposes. Users with incorrect credentials receive a 401 error code while unauthorized users receive a 403 error code.

12. INFLUXDB APPLICATION

12.1 Installation and getting started(MAC OS):

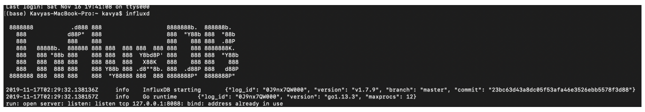
InfluxDB can be installed using the HomeBrew Package Manager.

-
- Install HomeBrew Package manager if it isn't already installed in the system. Go to the terminal prompt and run the following command :
  /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)
- Once HomeBrew is installed, install InfluxDB by running the following commands :

```
brew update
brew install influxdb
```

- Start the influx process by :

```
influxd
```



```
Last login: Sat Nov 16 19:41:08 on ttys000
(base) Kavyas-MacBook-Pro:~ kavya$ influxd

 8888888           .d888 888                   8888888b.  888888b.
   888            d88P"  888                   888  "Y88b 888  "88b
   888            888    888                   888    888 888  .88P
   888   88888b.  888888 888 888  888 888  888 888    888 8888888K.
   888   888 "88b 888    888 888  888  Y8bd8P' 888    888 888  "Y88b
   888   888  888 888    888 888  888   X88K   888    888 888    888
   888   888  888 888    888 Y88b 888 .d8""8b. 888  .d88P 888   d88P
 8888888 888  888 888    888  "Y88888 888  888 8888888P"  8888888P"

2019-11-17T02:29:32.138136Z     info    InfluxDB starting       {"log_id": "0J9nx7QW000", "version": "v1.7.9", "branch": "master", "commit": "23bc63d43a8dc05f53afa46e3526ebb5578f3d88"}
2019-11-17T02:29:32.138157Z     info    Go runtime      {"log_id": "0J9nx7QW000", "version": "go1.13.3", "maxprocs": 12}
run: open server: listen: listen tcp 127.0.0.1:8088: bind: address already in use
```

Figure 15: Start the influx process

12.2 Obtain data using line protocol format

While writing the data, a timestamp must be associated with it. If a timestamp is not present, the InfluxDB automatically assigns a timestamp using the current local time. Input to InfluxDB is in the Form of Line Protocol.

12.3 Preprocess data

We obtain our data from the OpenWeatherMap API. This data is in CSV format.

12.3.1 cleaner.py

This program takes in the weather data from the CSV files as a *Pandas dataframe*. It then strips away all the unnecessary columns and keeps the columns of interest like 'dt','city_id','temp','temp_min','temp_max','pressure', 'humidity','wind_speed','clouds_all','weather_id','weather_main', 'weather_description', 'weather_icon'. These columns of interest are saved in a new csv file.

12.3.2 timestamp.py

Converts the timestamps from minutes to nano seconds.

12.3.3 lineprotocol.py

Converts the modified csv file to a "line protocol" txt file. It is of the form :

```
atlanta,type=weather, weather\_main=Clouds,
weather\_description=overcastclouds
city\_id=4180439,temp=292.5,temp\_min=291.15,
temp\_max=293.75,pressure=1017,
humidity=81,wind\_speed=4,clouds\_all=90,
weather\_id=804 1546300800000000000
```

where :

- Measurement : atlanta
- tag_set : type, weather_main, weather_description
- field_set : city_id=4180439,temp=292.5,temp_min=291.15, temp_max=293.75,pressure=1017,humidity=81, wind_speed=4, clouds_all=90,weather_id=804
- Timestamp : 1546300800000000000

12.4 Add the data to InfluxDB

Download the data files and copy them unto the *usr/local/etc/* directory. This is where the InfluxDB is located. You could find the lineprotocol files needed "atlanta.txt" and "rochester.txt", in the data_files folder of the project. These are line protocol files obtained from the pre-processing step. Once the text files are in the directory we can move on to creating a database in influx.

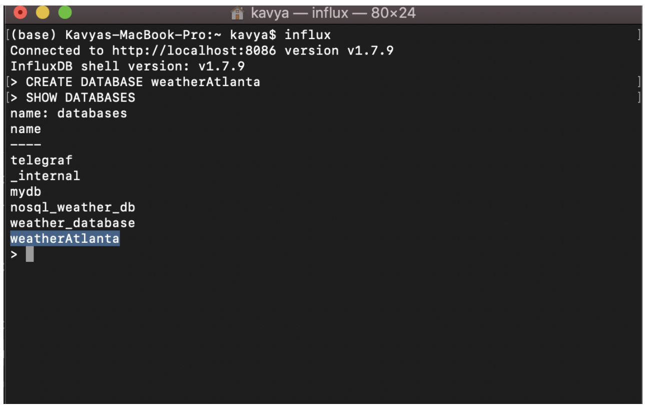
12.5 Create Database in influx

Go to terminal and start the *influx* with the command :

```
influx
```

This pulls up the *influx* CLI. Create a database *"weatherAtlanta"* in influx.

```
CREATE DATABASE weatherAtlanta
```



```
(base) Kavyas-MacBook-Pro:~ kavya$ influx
Connected to http://localhost:8086 version v1.7.9
InfluxDB shell version: v1.7.9
> CREATE DATABASE weatherAtlanta
> SHOW DATABASES
name: databases
name
----
telegraf
_internal
mydb
nosql_weather_db
weather_database
weatherAtlanta
>
```

Figure 16: Create table on influx CLI

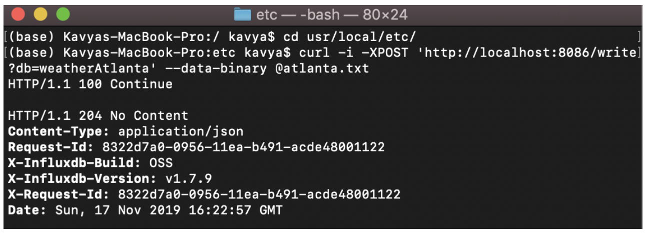
Once the database is created. exit the influx CLI. Go to the *usr/local/etc/* directory and run the command :

- For data in Atlanta:

```
curl -i -XPOST
'http://localhost:8086/write?db=weatherAtlanta'
--data-binary @atlanta.txt
```

- For data in Rochester:

```
curl -i -XPOST
'http://localhost:8086/write?db=weatherAtlanta'
--data-binary @rochester.txt
```



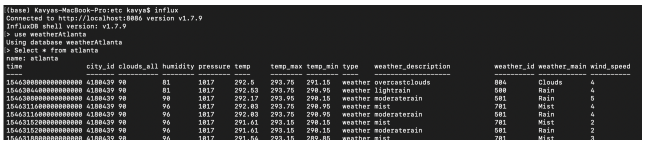
```
(base) Kavyas-MacBook-Pro:/ kavya$ cd usr/local/etc/
(base) Kavyas-MacBook-Pro:etc kavya$ curl -i -XPOST 'http://localhost:8086/write
?db=weatherAtlanta' --data-binary @atlanta.txt
HTTP/1.1 100 Continue

HTTP/1.1 204 No Content
Content-Type: application/json
Request-Id: 8322d7a0-0956-11ea-b491-acde48001122
X-Influxdb-Build: OSS
X-Influxdb-Version: v1.7.9
X-Request-Id: 8322d7a0-0956-11ea-b491-acde48001122
Date: Sun, 17 Nov 2019 16:22:57 GMT
```

Figure 17: Add data to database using curl

This helps to load all the data from the line protocol text file *"atlanta.txt"* into the *weatherAtlanta* database. All the data can now be accessed as a table of SERIES.
To ensure that the data has been loaded correctly we could go back into the *influx* CLI and display values from MEASUREMENT *atlanta*.

```
use weatherAtlanta
Select * from atlanta
```



```
(base) Kavyas-MacBook-Pro:etc kavya$ influx
Connected to http://localhost:8086 version v1.7.9
InfluxDB shell version: v1.7.9
> use weatherAtlanta
Using database weatherAtlanta
> Select * from atlanta
name: atlanta
time                city_id clouds_all humidity pressure temp     temp_max temp_min type    weather_description            weather_id weather_main wind_speed
----                ------- ---------- -------- -------- ----     -------- -------- ----    -------------------            ---------- ------------ ----------
1546300800000000000 4180439 90         81       1017     292.5    293.75   291.15   weather overcastclouds                 804        Clouds       4
1546304400000000000 4180439 90         81       1017     292.53   293.75   290.95   weather lightrain                      500        Rain         4
1546308000000000000 4180439 90         90       1017     292.17   293.95   290.15   weather moderaterain                   501        Rain         5
1546311600000000000 4180439 90         96       1017     292.03   293.75   290.95   weather mist                           701        Mist         4
1546311600000000000 4180439 90         96       1017     292.03   293.75   290.95   weather moderaterain                   501        Rain         4
1546315200000000000 4180439 90         96       1017     291.61   293.15   290.15   weather mist                           701        Mist         2
1546315200000000000 4180439 90         96       1017     291.61   293.15   290.15   weather moderaterain                   501        Rain         2
1546318800000000000 4180439 90         96       1017     291.54   293.15   289.85   weather mist                           701        Mist         3
```

Figure 18: Display all values from atlanta

12.6 Backend

We have created a backend using Node.js and Express. You can install node.js by :

```
brew update
brew install node
```

NPM(Node) has an *influx* package.

```
npm install --save influx
```

Using these packages we establish a connection between the influx database that runs on port 8086 and our node server which listens on the port 5000. Thus now we have access to the databases.

```
const Influx = require('influx');
const influx= new Influx.InfluxDB({
    host: 'localhost',
    database: 'weatherAtlanta',
    port:8086
});
```

Querying is handled as follows:
The query given below gets the desired tags and fields from the zipcode(atlanta or rochester) between the two given timestamps provided.

```
influx.query(`select ${tag}, ${field} from ${zipcode}
              where time >= ${startdate}
              AND time <= ${enddate}
    `       ).then(result => {
                res.json(result).then(data => {
                res.send({ data });
      })
    }).catch(err => {
      res.status(500).send(err.stack)
    })
```

The query given below gets aggregate functions on a particular field from the zipcode(atlanta or rochester) over the time interval provided which can be seconds, days, weeks etc

```
    influx.query(`
      select ${aggFunc}("${aggAttr}") from ${city}
    where time >= '${startDateTime}'
    AND time <= '${endDateTime}'
    GROUP BY time(${timeInterval})
  ').then(result => {
    res.json(result)
    .then(data => {
      res.send({ data });
    })
  }).catch(err => {
    res.status(500).send(err.stack)
```

12.7 How to install the project

To install:

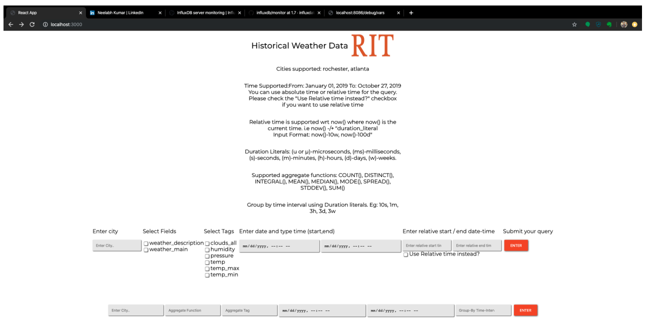
- Clone the repository to your local machine
- Once the repository is cloned locally, go ahead and access the /weather_app_tutorial directory from the terminal and run npm install to install the backend dependencies from the package.json file.
- While still inside /weather_app_tutorial navigate into the /client directory and then do the same thing by running npm install to install all of the dependencies that are required for React.
- In the root directory navigate to /server/ and run node server.js to start up the Express server..
- If you head over to http://localhost:5000/ in your web browser, you should see 'PORT 5000'. This confirms the server is up and running.
- From the terminal, navigate into the /client directory and then run npm start. This will start up the React server on port 3000. (Check out: http://localhost:3000/ if the React scripts did not launch the application automatically.)

12.8 Project Structure

- data_Files :- Contains the data to be inserted (Line Protocols)
- Client:- Contains code for the client part.
- Views: Defines different front end views.
  - CurrentWeather.js: Handles displaying the /search-location-weather route results.
  - groupWeather.js: Handles displaying the /group-location-weather route results.
  - Home.js: Defines the homepage.
  - Assets: Contains Images and Logos
- css: Contains all the css styling for all the pages.
- Server:- Contains the server part(routs and API).
  - Searchlocation.js: Defines all the routes
  - index.js: Looks for each route one by one.
  - server.js: Starts the express server.
  - Screenshots:- Contains all the screenshots
- README.md:- The readme file.
- package.json:- Package dependencies.

12.9 Frontend

We use React, HTML, CSS for the front end. The client runs on port 3000 and makes connection with the backend at port 5000 to handle queries and give the desired output. Our Application gives output both in tabular form and a plot to better visualise the trends.



Historical Weather Data

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city | Select Fields | Select Tags | Enter date and type time (start,end) | Enter relative start / end date-time | Submit your query

☐ weather_description
☐ weather_main

☐ clouds_all
☐ humidity
☐ pressure
☐ temp
☐ temp_max
☐ temp_min

mm/dd/yyyy, --:-- --
mm/dd/yyyy, --:-- --

☐ Use Relative time instead?

Figure 19: Home page

12.9.1 *Steps to navigate Through the UI*

We offer 3 Functionalities :

1 Query on the Measurement using Absolute time

2 Query on the Measurement using Relative time

3 Aggregate function display using Group by on time interval

Query on the Measurement using Absolute time

1 Please enter a city . (Cities supported: rochester, atlanta)

2 Select the desired fields and tags

3 Enter Date and time. Date supported : January 01, 2019 To: October 27, 2019

4 You can use the black down arrow to access the calendar and select date.

5 You will have to manually select the time fields and enter the time in the format "08:00:AM" or use the white up down arrow to change it.

6 Valid absolute time format : "06/05/2019, 08:00 AM"

7 Hit Enter

Query on the Measurement using Relative time

1 Please enter a city . (Cities supported: rochester, atlanta)

2 Select the desired fields and tags

3 Select the "Use Relative time instead?" checkbox

4 Relative time is supported wrt now() where now() is the current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

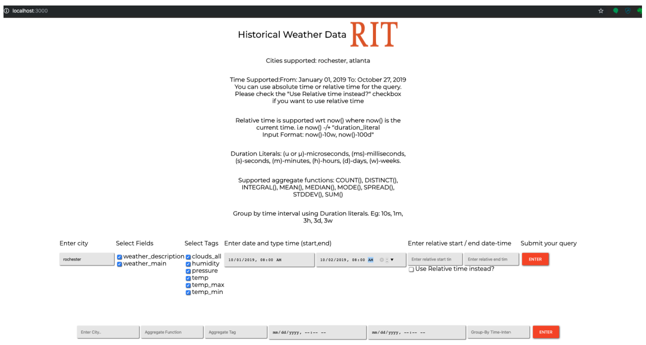
Aggregate function display using Group by on time interval

1. Please enter a city . (Cities supported: rochester, atlanta)
2. Supported aggregate functions: COUNT(), DISTINCT(), INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(), STDDEV(), SUM() (Case sensitive : Capitalization is needed)
3. Enter a tag : clouds_all,humidity,pressure,temp,temp_max, temp_min
4. Enter Date and time. Date supported : January 01, 2019 To: October 27, 2019
5. You can use the black down arrow to access the calendar and select date.
6. You will have to manually select the time fields and enter the time in the format “08:00:AM” or use the white up down arrow to change it.
7. Valid absolute time format : “06/05/2019, 08:00 AM”
8. Group by time interval using Duration literals. Input samples: 10s, 1m, 3h, 3d, 3w
9. Hit Enter

12.10 Application Details

12.10.1 *Absolute time queries*

The first query demonstrates how InfluxDB uses absolute time to return the desired tags and fields.


localhost:3000

Historical Weather Data RIT

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city: rochester

Select Fields: [✓ weather_description] [✓ weather_main]

Select Tags: [✓ clouds_all] [✓ humidity] [✓ pressure] [✓ temp] [✓ temp_max] [✓ temp_min]

Enter date and type time (start,end): 10/01/2019, 08:00 AM — 10/02/2019, 08:00 AM

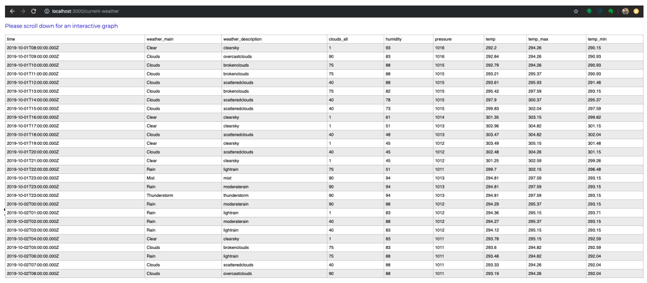
Enter relative start / end date-time: Enter relative start tin — Enter relative end tim

☐ Use Relative time instead?

Submit your query: ENTER

Enter City.. | Aggregate Function | Aggregate Tag | mm/dd/yyyy, --:-- -- | mm/dd/yyyy, --:-- -- | Group-By Time-Interv | ENTER

Figure 20: Aggregate query


localhost:3000/current-weather

Please scroll down for an interactive graph

time	weather_main	weather_description	clouds_all	humidity	pressure	temp	temp_max	temp_min
2019-10-01T08:00:00.000Z	Clear	clearsky	1	93	1016	292.2	294.26	290.15
2019-10-01T09:00:00.000Z	Clouds	overcastclouds	90	83	1016	292.84	294.26	290.93
2019-10-01T10:00:00.000Z	Clouds	brokenclouds	75	88	1015	292.79	294.26	290.93
2019-10-01T11:00:00.000Z	Clouds	brokenclouds	75	88	1015	293.21	295.37	290.93
2019-10-01T12:00:00.000Z	Clouds	scatteredclouds	40	88	1015	293.61	295.93	291.48
2019-10-01T13:00:00.000Z	Clouds	brokenclouds	75	82	1015	295.42	297.59	293.15
2019-10-01T14:00:00.000Z	Clouds	scatteredclouds	40	78	1015	297.9	300.37	295.37
2019-10-01T15:00:00.000Z	Clouds	scatteredclouds	40	73	1015	299.83	302.04	297.59
2019-10-01T16:00:00.000Z	Clear	clearsky	1	61	1014	301.35	303.15	299.82
2019-10-01T17:00:00.000Z	Clear	clearsky	1	51	1013	302.96	304.82	301.15
2019-10-01T18:00:00.000Z	Clouds	scatteredclouds	40	48	1013	303.47	304.82	302.04
2019-10-01T19:00:00.000Z	Clear	clearsky	1	45	1012	303.49	305.15	301.48
2019-10-01T20:00:00.000Z	Clouds	scatteredclouds	40	45	1012	302.48	304.26	301.15
2019-10-01T21:00:00.000Z	Clear	clearsky	1	45	1012	301.25	302.59	299.26
2019-10-01T22:00:00.000Z	Rain	lightrain	75	51	1011	299.7	302.15	296.48
2019-10-01T23:00:00.000Z	Mist	mist	90	94	1013	294.81	297.59	293.15
2019-10-01T23:00:00.000Z	Rain	moderaterain	90	94	1013	294.81	297.59	293.15
2019-10-01T23:00:00.000Z	Thunderstorm	thunderstorm	90	94	1013	294.81	297.59	293.15
2019-10-02T00:00:00.000Z	Rain	moderaterain	90	88	1012	294.29	295.37	293.15
2019-10-02T01:00:00.000Z	Rain	lightrain	1	83	1012	294.36	295.15	293.71
2019-10-02T02:00:00.000Z	Rain	moderaterain	40	88	1012	294.27	295.37	293.15
2019-10-02T03:00:00.000Z	Rain	lightrain	40	83	1012	294.12	295.15	293.15
2019-10-02T04:00:00.000Z	Clear	clearsky	1	83	1011	293.78	295.15	292.59
2019-10-02T05:00:00.000Z	Clouds	brokenclouds	75	83	1011	293.6	294.82	292.59
2019-10-02T06:00:00.000Z	Rain	lightrain	75	88	1011	293.48	294.82	292.04
2019-10-02T07:00:00.000Z	Clouds	scatteredclouds	40	88	1011	293.33	294.26	292.04
2019-10-02T08:00:00.000Z	Clouds	overcastclouds	90	88	1011	293.19	294.26	292.04

Figure 21: Result for Absolute time query : Table output

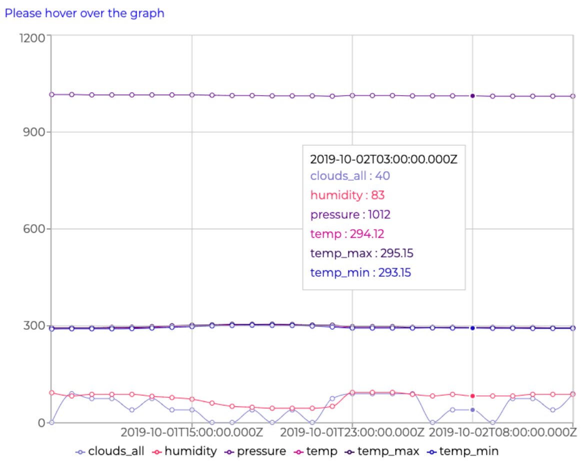
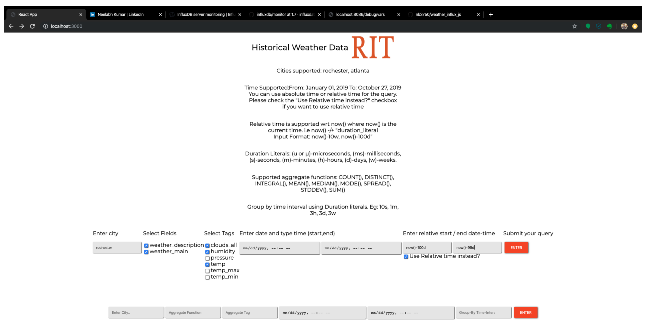


Figure 22: Result for Absolute time query : Plot output

12.10.2 *Relative time queries*

This query demonstrates how InfluxDB uses relative time to return the desired tags and fields. Relative time is the time relative to now. You need not have to type out a date and time, you can simply phrase the time with respect to *now()* . Example : now() - 10w; This commands the database to fetch all values between now and 10 weeks before this point in time.


React App | Neelabh Kumar \| LinkedIn | InfluxDB server monitoring \| Influ | influxdb/monitor at 1.7 · influxda | localhost:8086/debug/vars | nk3750/weather_influx_js

localhost:3000

Historical Weather Data RIT

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city: rochester

Select Fields: [✓ weather_description] [✓ weather_main]

Select Tags: [✓ clouds_all] [✓ humidity] ☐ pressure [✓ temp] ☐ temp_max ☐ temp_min

Enter date and type time (start,end): mm/dd/yyyy, --:-- -- — mm/dd/yyyy, --:-- --

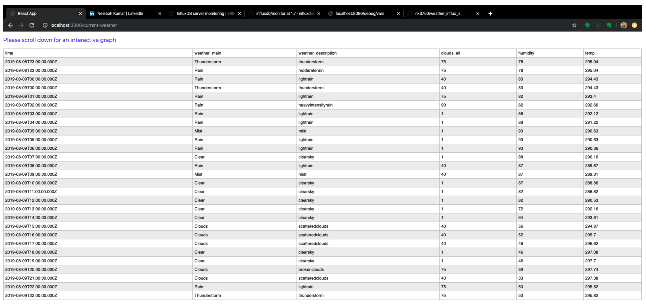
Enter relative start / end date-time: now()-100d — now()-99d

[✓ Use Relative time instead?]

Submit your query: ENTER

Enter City.. | Aggregate Function | Aggregate Tag | mm/dd/yyyy, --:-- -- | mm/dd/yyyy, --:-- -- | Group-By Time-Interv | ENTER

Figure 23: Relative time query

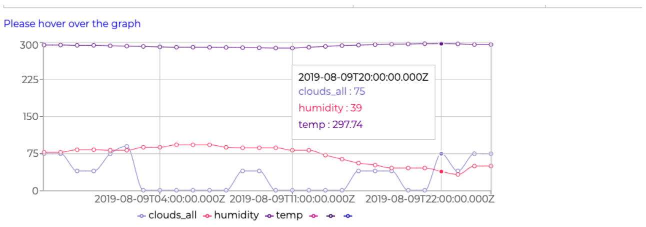

React App | Neelabh Kumar | LinkedIn | InfluxDB server monitoring | Influ | influxdb/monitor at 1.7 · influxdat | localhost:8086/debug/vars | nk3750/weather_influx_js

localhost:3000/current-weather

Please scroll down for an interactive graph

time	weather_main	weather_description	clouds_all	humidity	temp
2019-08-08T23:00:00.000Z	Thunderstorm	thunderstorm	75	78	295.04
2019-08-08T23:00:00.000Z	Rain	moderaterain	75	78	295.04
2019-08-09T00:00:00.000Z	Rain	lightrain	40	83	294.43
2019-08-09T00:00:00.000Z	Thunderstorm	thunderstorm	40	83	294.43
2019-08-09T01:00:00.000Z	Rain	lightrain	75	82	293.4
2019-08-09T02:00:00.000Z	Rain	heavyintensityrain	90	82	292.66
2019-08-09T03:00:00.000Z	Rain	lightrain	1	88	292.12
2019-08-09T04:00:00.000Z	Rain	lightrain	1	88	291.22
2019-08-09T05:00:00.000Z	Mist	mist	1	93	290.63
2019-08-09T05:00:00.000Z	Rain	lightrain	1	93	290.63
2019-08-09T06:00:00.000Z	Rain	lightrain	1	93	290.36
2019-08-09T07:00:00.000Z	Clear	clearsky	1	88	290.16
2019-08-09T08:00:00.000Z	Rain	lightrain	40	87	289.67
2019-08-09T09:00:00.000Z	Mist	mist	40	87	289.31
2019-08-09T10:00:00.000Z	Clear	clearsky	1	87	288.86
2019-08-09T11:00:00.000Z	Clear	clearsky	1	82	288.82
2019-08-09T12:00:00.000Z	Clear	clearsky	1	82	290.53
2019-08-09T13:00:00.000Z	Clear	clearsky	1	72	292.16
2019-08-09T14:00:00.000Z	Clear	clearsky	1	64	293.81
2019-08-09T15:00:00.000Z	Clouds	scatteredclouds	40	56	294.87
2019-08-09T16:00:00.000Z	Clouds	scatteredclouds	40	52	295.7
2019-08-09T17:00:00.000Z	Clouds	scatteredclouds	40	46	296.62
2019-08-09T18:00:00.000Z	Clear	clearsky	1	46	297.08
2019-08-09T19:00:00.000Z	Clear	clearsky	1	46	297.7
2019-08-09T20:00:00.000Z	Clouds	brokenclouds	75	39	297.74
2019-08-09T21:00:00.000Z	Clouds	scatteredclouds	40	33	297.38
2019-08-09T22:00:00.000Z	Rain	lightrain	75	50	295.82
2019-08-09T22:00:00.000Z	Thunderstorm	thunderstorm	75	50	295.82

Figure 24: Result for Relative time query : Table output

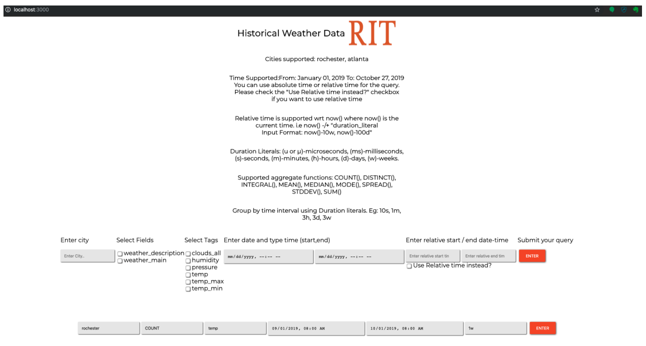

Please hover over the graph

2019-08-09T20:00:00.000Z
clouds_all : 75
humidity : 39
temp : 297.74

Figure 25: Result for Relative time query : Plot output

12.10.3 *Aggregate queries*

This query outputs aggregate summarizations of our data for the desired in the desired timeframe.
This query returns a count of temperature values between September 1st and October 1st 2019, in 1 week intervals, in rochester.


localhost:3000

Historical Weather Data RIT

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

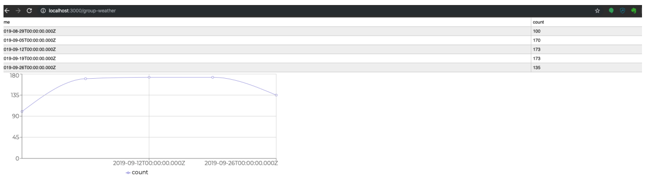
Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city | Select Fields | Select Tags | Enter date and type time (start,end) | Enter relative start / end date-time | Submit your query

Enter City..
☐weather_description ☐weather_main
☐clouds_all ☐humidity ☐pressure ☐temp ☐temp_max ☐temp_min
mm/dd/yyyy, --:-- -- mm/dd/yyyy, --:-- --
Enter relative start tim | Enter relative end tim
☐Use Relative time instead?
ENTER

rochester | COUNT | temp | 09/01/2019, 08:00 AM | 10/01/2019, 08:00 AM | 1w | ENTER

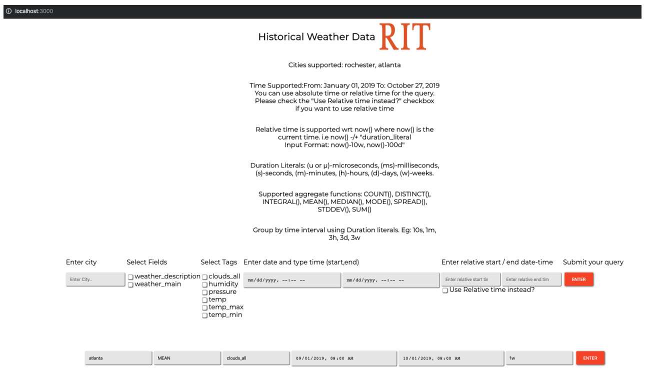
Figure 26: Aggregate query: COUNT


localhost:3000/group-weather

me	count
019-08-29T00:00:00.000Z	100
019-09-05T00:00:00.000Z	170
019-09-12T00:00:00.000Z	173
019-09-19T00:00:00.000Z	173
019-09-26T00:00:00.000Z	135

Figure 27: Result for Aggregate query: COUNT

This query returns the mean of *clouds_all* values between September 1st and October 1st 2019, in 1 week intervals.


localhost:3000

Historical Weather Data RIT

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

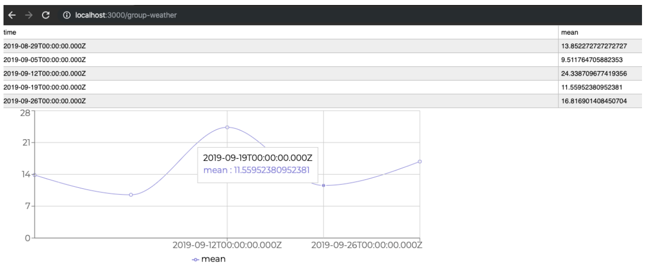
Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city | Select Fields | Select Tags | Enter date and type time (start,end) | Enter relative start / end date-time | Submit your query

Enter City..
☐weather_description ☐weather_main
☐clouds_all ☐humidity ☐pressure ☐temp ☐temp_max ☐temp_min
mm/dd/yyyy, --:-- -- mm/dd/yyyy, --:-- --
Enter relative start tim | Enter relative end tim
☐Use Relative time instead?
ENTER

atlanta | MEAN | clouds_all | 09/01/2019, 08:00 AM | 10/01/2019, 08:00 AM | 1w | ENTER

Figure 28: Aggregate query: MEAN

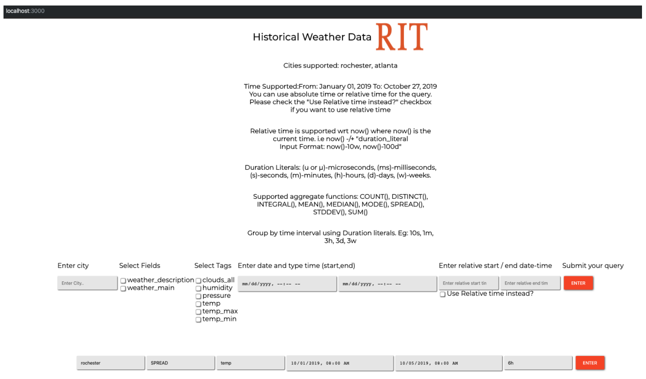

localhost:3000/group-weather

time	mean
2019-08-29T00:00:00.000Z	13.852272727272727
2019-09-05T00:00:00.000Z	9.511764705882353
2019-09-12T00:00:00.000Z	24.338709677419356
2019-09-19T00:00:00.000Z	11.55952380952381
2019-09-26T00:00:00.000Z	16.816901408450704

2019-09-19T00:00:00.000Z
mean : 11.55952380952381

Figure 29: Result for Aggregate query: MEAN

This query returns the spread(diff between min and max) of *temperature* values between October 1st and October 5th 2019, in 6 hr intervals.


localhost:3000

Historical Weather Data RIT

Cities supported: rochester, atlanta

Time Supported:From: January 01, 2019 To: October 27, 2019
You can use absolute time or relative time for the query.
Please check the "Use Relative time instead?" checkbox
if you want to use relative time

Relative time is supported wrt now() where now() is the
current time. i.e now() -/+ "duration_literal
Input Format: now()-10w, now()-100d"

Duration Literals: (u or µ)-microseconds, (ms)-milliseconds,
(s)-seconds, (m)-minutes, (h)-hours, (d)-days, (w)-weeks.

Supported aggregate functions: COUNT(), DISTINCT(),
INTEGRAL(), MEAN(), MEDIAN(), MODE(), SPREAD(),
STDDEV(), SUM()

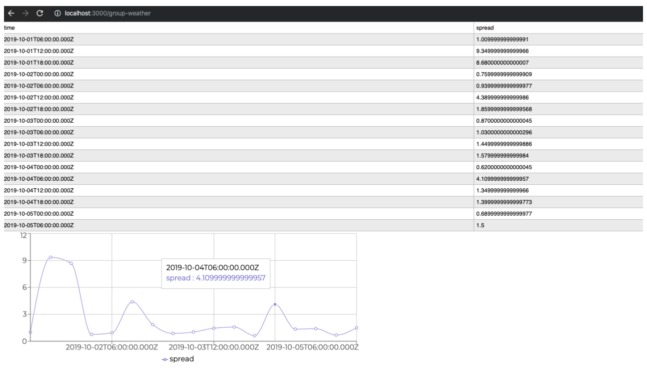
Group by time interval using Duration literals. Eg: 10s, 1m,
3h, 3d, 3w

Enter city | Select Fields | Select Tags | Enter date and type time (start,end) | Enter relative start / end date-time | Submit your query

Enter City..
☐weather_description ☐weather_main
☐clouds_all ☐humidity ☐pressure ☐temp ☐temp_max ☐temp_min
mm/dd/yyyy, --:-- -- mm/dd/yyyy, --:-- --
Enter relative start tim | Enter relative end tim
☐Use Relative time instead?
ENTER

rochester | SPREAD | temp | 10/01/2019, 08:00 AM | 10/05/2019, 08:00 AM | 6h | ENTER

Figure 30: Aggregate query: SPREAD

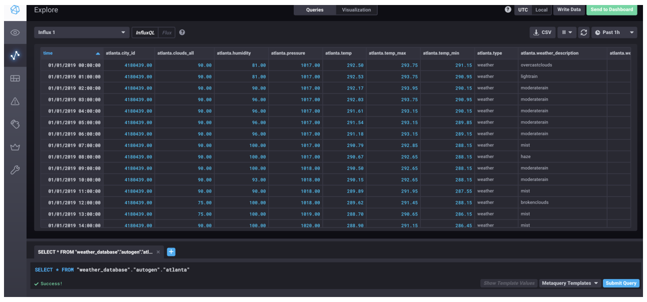

localhost:3000/group-weather

time	spread
2019-10-01T06:00:00.000Z	1.009999999999991
2019-10-01T12:00:00.000Z	9.349999999999966
2019-10-01T18:00:00.000Z	8.680000000000007
2019-10-02T00:00:00.000Z	0.7599999999999909
2019-10-02T06:00:00.000Z	0.9399999999999977
2019-10-02T12:00:00.000Z	4.389999999999986
2019-10-02T18:00:00.000Z	1.8599999999999568
2019-10-03T00:00:00.000Z	0.8700000000000045
2019-10-03T06:00:00.000Z	1.0300000000000296
2019-10-03T12:00:00.000Z	1.4499999999999886
2019-10-03T18:00:00.000Z	1.579999999999984
2019-10-04T00:00:00.000Z	0.6200000000000045
2019-10-04T06:00:00.000Z	4.109999999999957
2019-10-04T12:00:00.000Z	1.349999999999966
2019-10-04T18:00:00.000Z	1.3999999999999773
2019-10-05T00:00:00.000Z	0.6899999999999977
2019-10-05T06:00:00.000Z	1.5

Figure 31: Result for Aggregate query: SPREAD

12.11 Chronograf Display

Chronograf stands for the 'C' in the *TICK* stack and serves as a GUI for influxDB. For our database we can query the table to retrieve values and even plot graphs for visualisation.



time	atlanta.city_id	atlanta.clouds_all	atlanta.humidity	atlanta.pressure	atlanta.temp	atlanta.temp_max	atlanta.temp_min	atlanta.type	atlanta.weather_description	atlanta.we
01/01/2019 00:00:00	4180439.00	90.00	81.00	1017.00	292.50	293.75	291.15	weather	overcastclouds	
01/01/2019 01:00:00	4180439.00	90.00	81.00	1017.00	292.53	293.75	290.95	weather	lightrain	
01/01/2019 02:00:00	4180439.00	90.00	90.00	1017.00	292.17	293.95	290.15	weather	moderaterain	
01/01/2019 03:00:00	4180439.00	90.00	96.00	1017.00	292.03	293.75	290.95	weather	moderaterain	
01/01/2019 04:00:00	4180439.00	90.00	96.00	1017.00	291.61	293.15	290.15	weather	moderaterain	
01/01/2019 05:00:00	4180439.00	90.00	96.00	1017.00	291.54	293.15	289.85	weather	moderaterain	
01/01/2019 06:00:00	4180439.00	90.00	96.00	1017.00	291.18	293.15	289.15	weather	moderaterain	
01/01/2019 07:00:00	4180439.00	90.00	100.00	1017.00	290.79	292.85	288.15	weather	mist	
01/01/2019 08:00:00	4180439.00	90.00	100.00	1017.00	290.67	292.65	288.15	weather	haze	
01/01/2019 09:00:00	4180439.00	90.00	100.00	1018.00	290.50	292.65	288.15	weather	moderaterain	
01/01/2019 10:00:00	4180439.00	90.00	93.00	1018.00	290.15	292.65	288.15	weather	moderaterain	
01/01/2019 11:00:00	4180439.00	90.00	90.00	1018.00	289.89	291.95	287.55	weather	mist	
01/01/2019 12:00:00	4180439.00	75.00	100.00	1018.00	289.62	291.45	288.15	weather	brokenclouds	
01/01/2019 13:00:00	4180439.00	75.00	100.00	1019.00	288.70	290.65	286.15	weather	mist	
01/01/2019 14:00:00	4180439.00	90.00	100.00	1020.00	288.90	291.15	286.45	weather	mist	

```
SELECT * FROM "weather_database"."autogen"."atl...
SELECT * FROM "weather_database"."autogen"."atlanta"
✓ Success!
```

Figure 32: Chronograf : Table output

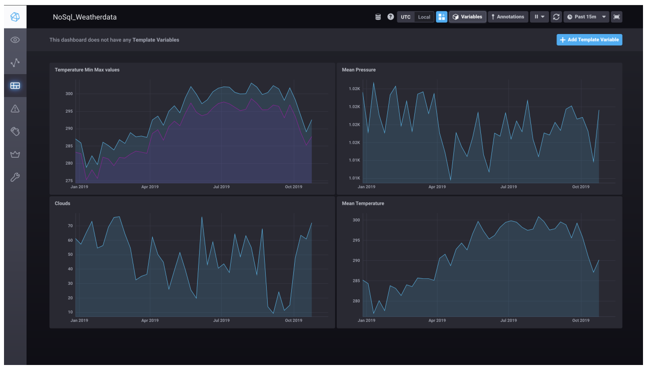


Figure 33: Chronograf : Plot visualisation

13. NOTE

We are trying to plot a graph in our project to mimic Chronograf. Rendering of the graph makes the processing slow, and not because of fetching values from the database.

14. FINAL REMARKS

RDBMS uses the SQL language. SQL is preferred by most people as it has been around for a long time. However it has many disadvantages. In the mordern era, handling real time continuous data is required. SQL databases do not natively support this kind of data. Many special queries must be written specifically for each database.

On the other hand, InfluxDB was developed keeping in mind that time centric functions must be executed efficiently. It handles real time continuous data easily. It has a far superior compression rate compared to any other database. Its query proccessing speed is also faster than RDBMS even though Influx deals with larger datasets. It also has many inbuilt features which make it easier for a developer to set up a new database.

Hence we can see that when it comes to handling continuous data, using InfluxDB makes it easier store and query the database than using a RDBMS.

APPENDIX

A. REFERENCES

[1] Definition of NoSQL, `https://en.wikipedia.org/wiki/Time_series/`
[2] Types of NoSQl, `https://aws.amazon.com/nosql/`
[3] Introduction to InfluxDB, `https://en.wikipedia.org/wiki/InfluxDB/`
[4] Definition of Time Series, `https://en.wikipedia.org/wiki/Time_series/`
[5] Advantage of using Time Series, `https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/`
[6] What is a Time Series Database?, `https://www.influxdata.com/blog/introduction-to-influxdatas-influxdb-and-tick-stack/`
[7] Why do we need Time Series Database?, `https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/`
[8] Records in SQL, `https://www.red-gate.com/simple-talk/sql/database-administration/sql-server-storage-internals-101/`
[9] Pages in SQL, `https://www.red-gate.com/simple-talk/sql/database-administration/sql-server-storage-internals-101/`
[10] SQL vs. Flux: Choosing the right query language for time-series data, `https://blog.timescale.com/blog/sql-vs-flux-influxdb-quer y-language-time-series-database-290977a01a8a/`
[11] Query InfluxDB with Flux `https://docs.influxdata.com/influxdb/v1.7/query_language/spec/`
[12] Syeda Noor Zehra Naqvi, Sofia Yfantidou, 2017, *Time Series Databases and InfluxDB*, Advanced Databases Winter Semester 2017-2018
[13] https://db-engines.com/en/ranking_trend/time+series+dbms
[14] https://www.influxdata.com/time-series-platform/telegraf/
[15] https://docs.influxdata.com/kapacitor/v1.5/