Introduction

In the realm of cutting-edge database solutions, JaguarDB stands tall as a robust and scalable distributed database, uniquely tailored to meet the demands of Internet of Things (IoT) data platforms and applications. With its unrivaled capabilities, JaguarDB opens up a world of possibilities for handling vast amounts of IoT-generated data efficiently and effectively.

JaguarDB's core strength lies in its exceptional power and performance. Built from the ground up to handle the complexities of IoT data, it boasts lightning-fast data processing and storage, enabling real-time insights and analytics. The database's architecture is optimized to deliver seamless scalability, effortlessly adapting to the growing data needs of IoT-driven environments.

JaguarDB's scalability is unparalleled, making it a formidable contender in the world of distributed databases. Regardless of the volume of data generated by IoT devices, JaguarDB rises to the occasion, gracefully accommodating the ever-expanding data horizons while maintaining optimal performance. Its elastic scaling capabilities ensure that your IoT infrastructure can grow effortlessly, meeting the needs of even the most dynamic environments.

Application Scenario

In this document, we showcase the incredible capabilities of JaguarDB in managing and tracking various types of IoT devices, with a particular focus on location tracking and event monitoring. Whether dealing with stationary sensors, moving vehicles, or drones orchestrating complex

interactions, JaguarDB excels in seamlessly handling diverse IoT data streams and providing rapid access to statistical insights through fast queries.

With JaguarDB's prowess in event data handling, monitoring emitted events from IoT devices becomes a streamlined process. The database efficiently stores and organizes event data, allowing users to analyze device behavior, trigger responses, and gain valuable insights into IoT device interactions. This capability is crucial for applications ranging from smart home automation to large-scale industrial processes.

Creating Table and Indexes

He we use JaguarDB's SQL-like statements to illustrate the operations in managing the data. The statements can be embedded in any Java, Python, or any supported API programming languages.

The following statement creates a table that will store various type of objects such as sensors, cars, and drones.

```
create table if not exists iot (
     key:
         id uuid,
     value:
         name char(16),
         type char(8),
         model char(8),
         year date,
         lonlat point(srid:wgs84)
);
```

In this data schema, we define the essential fields that play a pivotal role in uniquely identifying objects within the system using JaguarDB. Each object is characterized by specific attributes that

facilitate effective tracking and management. Let's delve into the key components of this schema:

"id" Field:

The "id" column serves as a fundamental and crucial element, acting as a unique identifier for each object in the system. Employing the "uuid" data type, it ensures that every identifier is a globally unique string of precisely 32 characters. This guarantees that no two objects share the same "id," facilitating seamless and unambiguous object identification.

"name" Field:

The "name" field represents the name of the object. This attribute can be either unique or non-unique, depending on the context of the system. While uniqueness may be required for some objects, non-unique names allow for flexibility in naming conventions. This facilitates easy recognition and categorization of objects based on their names.

"type" Field:

The "type" field is dedicated to defining the classification of each object in the system. Objects can belong to different categories, such as "sensor," "car," or "drone." Categorization by type enables efficient grouping and retrieval of objects based on their functionalities, enhancing overall system organization.

"Year" Field:

The "Year" attribute provides a time reference for when an object was put into use within the system. This temporal information enables historical tracking and helps identify the duration of an object's active usage. It is particularly useful for conducting performance evaluations and maintenance scheduling.

"lonlat" Field:

The "lonlat" field captures the geo-location of each object in the system. This attribute is crucial for tracking the real-time position and movements of objects. For stationary sensors or base stations, which remain fixed in location, the "lonlat" remains constant. The geographic coordinates conform to the WGS84 standard, ensuring universal compatibility and seamless integration with various mapping applications.

With the successful implementation of the “iot” table, which efficiently manages the inventory of all IoT objects, we now move forward to augment our system's capabilities by introducing a new table dedicated to tracking the dynamic state of mobile objects. This addition will enable real-time monitoring and seamless handling of objects that exhibit mobility characteristics.

```
create table iotstatus (
     key:
          id char(32),
          ts timestamp,
     value:
          type char(8),
          tmp smallint,
          speed smallint,
          lonlat point(srid:wgs84)
);
```

The above statement represents the SQL-like syntax for creating a table named "iotstatus" in a database. The components of this table and their respective data types are described as follows:

Table Name: iotstatus

This line indicates that we are creating a table named "iotstatus" to store the dynamic states of all IoT objects in our system.

Key:

The "key" section of the table represents the primary key or unique identifier for each record in the table.

In this case, the primary key is represented by the "id" field, which is of data type "char(32)". The "char(32)" means that the "id" field can hold a character string of 32 characters. It corresponds to the uuid string of an object in the “iot” table.

Value:

The "value" section includes attributes that hold the actual data or status information for the IoT objects.

ts Field:

The "ts" field represents a timestamp, denoted by the data type "timestamp". Timestamps store date and time information.

The "ts" field is used to record the time at which a specific status update occurred for an IoT object.

type Field:

The "type" field is of data type "char(8)". It is used to store the type of the IoT object, which could be represented as a character string with a maximum length of 8 characters. For example, it might store values like "sensor," "car," or "drone."

tmp Field:

The "tmp" field represents temperature data and is of data type "smallint". Smallint is used to store small integer values, typically within a range from -9999 to 9999. In this case, it might store temperature readings as whole numbers.

speed Field:

The "speed" field is also of data type "smallint". It stores the speed of the IoT object, which could be measured in units like meters per second or kilometers per hour.

lonlat Field:

The "lonlat" field represents the geographic location (longitude and latitude) of the IoT object.

It is of data type "point(srid:wgs84)". "point" is a spatial data type that stores a single point in two-dimensional space, and "srid:wgs84" indicates that the spatial reference system used is WGS84, which is a widely used standard for Earth's surface coordinates.

In summary, the "iotstatus" table is designed to store status updates for IoT objects. The primary key "id" identifies each object, while the "ts" field records the timestamp of each status update. The combined key “id” and “ts” ensures the uniqueness of reach record. The "type" "tmp" "speed" and "lonlat" fields hold the type, temperature, speed, and location data, respectively, for the IoT objects being tracked.

With the table “iotstatus” it is easy to find the dynamic states of an object since it is the first part of the composite key in the table. However, to find states of all objects or any object during a time interval, we need a time-axis to store the state data. The following index can be created for this purpose:

```
create index statidx
     on iotstatus(key: ts, id, value: type, tmp, speed);
```

Here the timestamp “ts” field and the uuid “id” field are combined to work as the primary key in the index “statidx”. The values of fields “type”, “tmp”, and “speed” are here duplicated to the index for easy lookup without joining the “iotstatus” table. Normally NoSQL databases do not support table joins.

With the successful establishment of tables for inventory management and tracking states of moving objects, the need for seamless inter-object communication arises. To facilitate this communication and efficiently track messages exchanged among various objects, we propose the creation of a dedicated "iotmessage" table. This table will serve as a central repository for recording messages transmitted between objects, enabling effective communication scenarios such as car-to-car, car-to-base station, drone-to-car, drone-to-drone, or drone-to-base station interactions.

```
create table if not exists iotmessage (
     key:
          id uuid,
     value:
          from char(32),
          to char(32),
          data char(64)
```

```
);

create index msgidx on iotmessage(key: from, to, id);
```

Each message created among the objects has a unique uuid. Three essential elements in a message include: 1) the “from” field to represent the id of the sender; 2) the “to” field to represent the receiver; 3) data or message transmitted between the sender and the receiver. The index “msgidx” is created for easily finding the messages sent by each sender.

Storing Data

First let us fill the “iot” inventory with the following statements:

```
insert into iot values ( 'MON1_SEAT', 'SENS', 'SZTM0001', '2000-01-23',
point(-122.335167 47.608013) );

insert into iot values ( 'MON2_PORT', 'SENS', 'SZTM0001', '2000-01-24',
point(-122.6784 45.5152) );

insert into iot values ( 'MON3_SANF', 'SENS', 'SZTM0002', '2000-01-25',
point(-122.4194 37.7749) );

insert into iot values ( 'MON4_CHCG', 'SENS', 'SZTM0003', '2000-02-03',
point(-87.6198 41.8781) );


insert into iot values ( 'CAR1', 'CAR', 'TESLA3', '2010-02-03' );

insert into iot values ( 'CAR2', 'CAR', 'TESLAS', '2010-03-03' );

insert into iot values ( 'CAR3', 'CAR', 'TESLAX', '2010-04-03' );

insert into iot values ( 'CAR4', 'CAR', 'TESLAY', '2010-05-03' );


insert into iot values ( 'DRONE1', 'DRONE', 'XYZ1', '2015-02-03' );

insert into iot values ( 'DRONE2', 'DRONE', 'XYZ2', '2015-03-03' );

insert into iot values ( 'DRONE3', 'DRONE', 'XYZ3', '2016-04-03' );

insert into iot values ( 'DRONE4', 'DRONE', 'XYZ4', '2018-05-03' );
```

During the operation of our system, the mobile objects can enter their live status information to the system such as:

```
insert into iotstatus (id, type, tmp, speed, lonlat ) values ( ‘car1uuid’, 'CAR', 20, 60, point(-122.335167 47.608013) );

insert into iotstatus (id, type, tmp, speed, lonlat ) values ( ‘car1uuid’, 'CAR', 24, 50, point(-122.336168 47.608114) );

insert into iotstatus (id, type, tmp, speed, lonlat ) values ( ‘car1uuid’, 'CAR', 26, 55, point(-122.337169 47.608215) );

insert into iotstatus (id, type, tmp, speed, lonlat ) values ( ‘car1uuid’, 'CAR', 27, 65, point(-122.337369 47.608516) );
```

Any object (mobile or stationary) can send messages to any other object as shown in the following:

```
insert into iotmessage values (‘car1uuid’, ‘car2uuid’, 'Hello, are you there?');

insert into iotmessage values (‘car2uuid’, ‘car1uuid’, 'I am here, where are you?');

insert into iotmessage values (‘car1uuid’, ‘car2uuid’, 'I am near Sunnyvale');

insert into iotmessage values (‘car2uuid’, ‘car1uuid’, 'Let us meet at Sunnyvale');
```

When data is inserted into a table, its related indexes are also updated, automatically.

Query and Analysis

We can make queries about the objects in the inventory:

```
### details of objects within 100 meters from the location

select * from iot where distance(lonlat, point(-122.335167 47.608013)) < 100;
```

```
### number of objects within 100 meters from the location


select count(1) cnt from iot where distance(lonlat, point(-122.335167
47.608013)) < 100;


### details of objects within 10 kilometers from the location

select * from iot where distance(lonlat, point(-122.335167 47.618013)) <
10000;


### details of all sensors or base stations within 10 kilometers

select * from iot where type='SENS' and distance(lonlat, point(-122.335167
47.618013)) < 10000;


### details of all sensors in the inventory

select * from iot where type='SENS';
```

We can analyze the dynamic states of all objects:

```
select avg(tmp) avgtmp from iotstatus where id=‘car1uuid’;

select min(tmp) mintmp from iotstatus where id=‘car1uuid’;

select max(tmp) maxtmp from iotstatus where id=‘car1uuid’;

select stddev(tmp) stdevtmp from iotstatus where id=‘car1uuid’;


select avg(speed) avgspeed from iotstatus where id=‘car1uuid’;

select max(speed) maxspeed from iotstatus where id=‘car1uuid’;

select min(speed) minpeed from iotstatus where id=‘car1uuid’;


### get all data of all objects within a time interval

select * from test.iotstatus.statidx where  ts >= '2022-01-01 00:00:01' and
ts <= '2023-07-08 11:01:01';


### get average temperature of all objects during a time interval
```

```
select avg(tmp) avgtmp from test.iotstatus.statidx where ts >= '2022-01-01
00:00:01' and ts <= '2023-07-08 11:01:01';


### get maximum temperature of all objects during a time interval

select max(tmp) maxtmp from test.iotstatus.statidx where ts >= '2022-01-01
00:00:01' and ts <= '2023-07-08 11:01:01';


### get minimum temperature of all objects during a time interval

select min(tmp) mintmp from test.iotstatus.statidx where ts >= '2022-01-01
00:00:01' and ts <= '2023-07-08 11:01:01';


### get standard deviation of temperature fluctuation of all objects

select stddev(tmp) as stddevtmp from test.iotstatus.statidx where ts >=
'2022-01-01 00:00:01' and ts <= '2023-07-08 11:01:01';


### get maximum temperature of all cars during a time interval

select max(tmp) maxtmp from test.iotstatus.statidx where type = 'CAR' and
ts >= '2022-01-01 00:00:01' and ts <= '2023-07-08 11:01:01';
```

We can also find messages sent during an interval or between objects.

```
### get all messages sent during a time interval

select uuidtime(id), from, to, data from iotmessage where uuidtime(id) >=
'2022-07-19 04:12:00.422028';


### get all messages sent from one object to another

select * from test.iotmessage.msgidx where from=’car1uuid’ and to=’car2uuid’;
```

JaguarDB employs efficient and optimized data structures and algorithms to conduct all the above queries so that the results are returned to the user quickly.

Summary

In summary, we demonstrated using tables and indexes to manage IoT data systems and perform fast queries and analyses on IoT data. JaguarDB emerges as a powerful database solution that efficiently handles IoT data and supports the complexities of IoT systems. Leveraging its robust time series data support and geospatial capabilities, JaguarDB empowers businesses to gain valuable insights, monitor objects in real-time, and make data-driven decisions. With high reliability, and effortless scalability, JaguarDB is a formidable choice for building sophisticated, high-performance IoT infrastructures.