Event-Condition-Action Rule Execution for Transactional Queues

Using ECA rules for acting on messages in transactional queueing

This blog discusses an approach to execute ECA (Event-Condition-Action) rules on JSON objects within database transactions in PostgreSQL — specifically on transactional queues. This guarantees exactly once execution and consistency of rule execution in case of failures (independent of the underlying reason).

Background

ECA rules is an established concept in the active database domain (https://en.wikipedia.org/wiki/Event_condition_action). The three elements of ECA rules are

- **Event**. An event is one or more changes in the database, for example, an insert or update or deletion of a row in a table.
- **Condition**. A condition is a predicate that evaluates to `TRUE` or `FALSE` and that is evaluated on the changes that caused an event.
- **Action**. An action is a directive or code execution initiated when the condition evaluates to `TRUE`.

A familiar implementation are database triggers where the change that the trigger listens to is an event, the condition is the `WHEN` clause (in PostgreSQL syntax), and the action is a function being invoked when the condition applies.

In PostgreSQL triggers are executed in the same transaction that made the changes to the database that caused the trigger to execute. Several triggers can apply to a change and those are all executed in the same transaction.

ECA applied to transactional queues

An important application of the ECA rule concept is to queues and to messages within queues. The blog Implementing Queues in PostgreSQL (Part 1: Design and

Measurement) shows an implementation of a transactional message queue within a database.

In context of this blog a message represents an event that consists of three elements

- **Event identifier**. This is a unique identifier for an event.
- **Event name**. An event has a name. The event name is used to select all ECA rules that refer to the same name. The conditions of the selected ECA rules will be used for evaluation.
- **Event payload**. This is a JSON object of arbitrary schema.

Instead of dequeuing each event and acting on it (classical dequeue operation), some use cases require actions based on the event payload's values evaluated by a predicate. In addition, several actions for each event are possible if there are different predicates that need to be evaluated for one event.

Therefore, for a given event there might be zero, one or more actions. If there is no action for an event because no predicate applied it is often important that this is noted in the system for possible downstream analysis.

Overview

This blog demonstrates one approach of executing predicates on events with JSON payloads and deriving the corresponding actions. ECA rules are explicitly managed as data in a table, and not implemented directly as database triggers themselves.

The table `eca` stores all ECA rules in scope. An ECA rule has an identifier, an event name, a condition, and an action. Since ECA rules are represented as data in a table they are managed by insert, update or delete operations like any other data.

When an event is inserted into the `event` table (enqueue), an insert trigger initiates the ECA rule evaluation on the event of all rules that are stored in the `eca` table. For each matching ECA rule the corresponding action is stored with the event payload in an `action` table that can be accessed by downstream processors.

After ECA rule execution the event is moved into an `event_history` table. If no ECA rule matched, the event is moved into an `event_unmatched` table. Each event therefore is either in the event history or in the set of unmatched events after it has been processed.

Database tables

ECA table

The `eca` table is specified as follows:

```
CREATE TABLE eca
(
    eca_id          UUID    NOT NULL
        CONSTRAINT eca_id_pk
            PRIMARY KEY,
    eca_condition   VARCHAR NOT NULL,
    eca_action_name VARCHAR NOT NULL,
    eca_event_name  VARCHAR NOT NULL
);
```

The `eca_condition` column contains valid SQL predicates that are evaluated against the event payload. An example is introduced below to visualize values in the various columns.

Indexes are not shown, for example, on `eca_event_name`, the column that is used to match with an incoming event based on its `event_name`.

Event table

The `event` table is specified as follows:

```
CREATE TABLE event
(
    event_id      UUID    NOT NULL
        CONSTRAINT event_id_pk
            PRIMARY KEY,
    event_payload JSONB,
```

```
    event_name    VARCHAR NOT NULL
);
```

The `event_payload` is of type `JSONB` since an `eca_condition` might be a complex expression accessing different JSON properties. A binary representation at time of evaluation improves the execution duration.

Indexes are not shown, like for example a GIN index on `event_payload` in order to make access more efficient. The usefulness of an index depends on your particular use case and should be based on measurement.

Action table

If an event has at least one ECA rule for which the condition evaluates to `TRUE` the `event_payload` from the event, and the `action_name` from the ECA rule are inserted into an `action` table.

```
CREATE TABLE action
(
    action_id      UUID    NOT NULL
        CONSTRAINT action_id_pk
            PRIMARY KEY,
    action_name    VARCHAR NOT NULL,
    action_payload JSONB
);
```

This table is very simple right now in order to only show the ECA rule semantics. In a product environment this table is used to trigger executions of applications or microservices to execute the action.

To maintain event arrival order (if that is required) a column is needed that captures the event arrival time. If the execution status of the action is to be managed, another column can contain the action execution status.

Event history and unmatched event table

After events are processed they are either moved to an `event_history` table or an `event_unmatched` table containing events that were not matched by an ECA rule:

```
CREATE TABLE event_history
(
    event_id      UUID    NOT NULL
        CONSTRAINT event_history_id_pk
            PRIMARY KEY,
    event_payload JSONB,
    event_name    VARCHAR NOT NULL
);
```

```
CREATE TABLE event_unmatched
(
    event_id      UUID    NOT NULL
        CONSTRATIN event_unmatched_id_pk
            PRIMARY KEY,
    event_payload JSONB,
    event_name    VARCHAR NOT NULL
);
```

After execution of one or more ECA rules for an event, the event is removed from the `event` table and resides either in the `event_history` table or the `event_unmatched` table.

The latter table is not necessary for ECA rule execution. It is in place for debugging purposes to ensure that unmatched events are a valid outcome in the given use case. If that is not the case, this table can be used for observing unmatched events and subsequently improve the ECA rules so that these exceptions can be addressed.

evaluate() function

Executing an `eca_condition` on an `event_payload` is done by using the `evaluate()` function introduced here: evaluate() PostgreSQL Function for Evaluating Stored Expressions (Part 1).

The function `evaluate()` has the following signature:

```
evaluate(object jsonb, expression varchar) returns boolean;
```

In this blog's use case of ECA rule execution the actual parameter value for `object` is the `event_payload`, and the actual parameter value for `expression` is the `eca_condition`.

If `evaluate()` returns `TRUE` the `eca_condition` matches, otherwise it does not.

ECA rule execution

ECA rules are executed as follows for an inserted event into the `event` table:

1. Select all `eca` rules for the `event_name`
2. For each `eca` rule found, execute `evaluate()` using `event_payload` and `eca_condition`
3. For each matching ECA rule, insert a row in `action` with `eca_action` and `event_payload`
4. If there was at least one ECA rule found (matching or not), move the `event` to `event_history`
5. If there was no ECA rule found at all, move the event to `event_unmatched`

This algorithm is implemented as a function triggered by an `AFTER INSERT` trigger on `event`.

The full algorithm is shown at the end in the Appendix.

Example

The following example is taken from evaluate() PostgreSQL Function for Evaluating Stored Expressions (Part 1) and refactored as an example for queueing:

- New car model notifications arrive as events
- ECA rules specify notifications as actions, and their priority depends on a car's specification

The following shows the ECA rules:

```
INSERT INTO eca (eca_id, eca_condition,
                 eca_action_name, eca_event_name)
VALUES (gen_random_uuid(),
        '(object -> ''horsepower'')::int > 1000',
        'NOTIFY_HIGH_PRIORITY',
        'NEW_CAR');

INSERT INTO eca (eca_id, eca_condition,
                 eca_action_name, eca_event_name)
VALUES (gen_random_uuid(),
        '(object -> ''price'')::int < 100000 and
          object ->> ''color'' = ''silver'' ',
        'NOTIFY_NORMAL_PRIORITY',
        'NEW_CAR');
```

Note that an explicit cast to `int` is necessary for the correct execution. Note also that the predicate refers to the values using `object` as the parameter of the `evaluate()` function is named `object`.

Here are two sample events:

```
INSERT INTO event (event_id, event_payload, event_name)
VALUES (gen_random_uuid(),
        '{"make": "Koenigsegg",
          "model": "CC850",
          "color": "silver",
          "horsepower": 1385,
          "price": 3650000}',
         'NEW_CAR');

INSERT INTO event (event_id, event_payload, event_name)
VALUES (gen_random_uuid(),
        '{"make": "Honda",
          "model": "Jazz",
          "color": "silver",
          "horsepower": 0,
          "price": 21394}',
         'NEW_CAR');
```

After the two events are inserted, this is the status of the `action` table and the `event_history` table:

```
SELECT * FROM action;

| action_id | action_name            | action_payload         |
+-----------+------------------------+------------------------+
| 5f640...  | NOTIFY_HIGH_PRIORITY   | {"make": "Koenigsegg", |
|           |                        |  "color": "silver",    |
|           |                        |  "model": "CC850",     |
|           |                        |  "price": 3650000,     |
|           |                        |  "horsepower": 1385}   |
+-----------+------------------------+------------------------+
| de179...  | NOTIFY_NORMAL_PRIORITY | {"make": "Honda",      |
|           |                        |  "color": "silver",    |
|           |                        |  "model": "Jazz",      |
|           |                        |  "price": 21394,       |
|           |                        |  "horsepower": 0}      |
+-----------+------------------------+------------------------+
```

```
SELECT * FROM event_history;

| event_id | event_payload          | event_name |
+----------+------------------------+------------+
| 9d1d...  | {"make": "Koenigsegg", | NEW_CAR    |
|          |  "color": "silver",    |            |
|          |  "model": "CC850",     |            |
|          |  "price": 3650000,     |            |
|          |  "horsepower": 1385}   |            |
+----------+------------------------+------------+
| 6f9a...  | {"make": "Honda",      | NEW_CAR    |
|          |  "color": "silver",    |            |
|          |  "model": "Jazz",      |            |
|          |  "price": 21394,       |            |
|          |  "horsepower": 0}      |            |
+----------+------------------------+------------+
```

Design considerations

Event names as well as action names should be database supervised and not just implemented as datatype `VARCHAR`. In most cases they are fixed enumerations, but in some they are dynamically changed. Implementations are therefore use case specific, for example, as checked constraints, or values in table containing the currently specified values.

The SQL predicates in the column `eca_condition` should be checked for syntactic correctness as shown here evaluate() PostgreSQL Function for Evaluating Stored Expressions (Part 2). This ensures that there are no runtime errors due to incorrect predicate syntax.

The event payload is moved between tables and also replicated (for example, in the action and history tables in case of a matching event). A possible design change is to store the event payload in a separate event payload table and use foreign key references to refer to the payload instead of replicating it by value. This avoids duplication and moving of potentially many large values.

Table maintenance is required since the `action`, `event_history` and `event_unmatched` tables grow continuously. Regular and periodic maintenance is important to restrict the tables' growth and keep those below a set limit.

Summary

This blog demonstrates an implementation of an Event-Condition-Action rule (ECA) system in the context of transactional queueing in PostgreSQL. ECA rules are managed as data and transactionally executed in order to derive actions to be taken on event arrival based on the specified ECA rules. The blog discusses detailed table and function specifications and provides an example.

Appendix — ECA rule evaluation function

The following listing shows the insert trigger and the invoked functions in order to execute ECA rules as specified in the `eca` table.

```
CREATE OR REPLACE FUNCTION evaluate_eca_rules()
    RETURNS TRIGGER
    LANGUAGE plpgsql
AS
```

```
$$
DECLARE
    v_action    RECORD;
    v_eca_found BOOLEAN;
BEGIN
    v_eca_found = FALSE;
    FOR v_action IN (SELECT eca_action_name,
                            event_payload
                     FROM determine_actions(
                             NEW.event_name,
                             NEW.event_payload))
        LOOP
            IF v_eca_found IS FALSE
            THEN
                v_eca_found = TRUE;
            END IF;

            INSERT INTO action (action_id,
                                        action_name,
                                        action_payload)
            VALUES (gen_random_uuid(),
                    v_action.eca_action_name,
                    v_action.event_payload);
        END LOOP;

    IF v_eca_found IS FALSE
    THEN
        -- Move event to unmatched table
        INSERT INTO event_unmatched (event_id,
                                                event_payload,
                                                event_name)
        VALUES (NEW.event_id,
                NEW.event_payload,
                NEW.event_name);
    ELSE
        -- Move event into history table
        INSERT INTO event_history (event_id,
                                               event_payload,
                                               event_name)
        VALUES (NEW.event_id,
                NEW.event_payload,
                NEW.event_name);
    END IF;

    DELETE FROM event WHERE event_id = NEW.event_id;

    RETURN NEW;
END;
$$;
```

```
CREATE OR REPLACE FUNCTION determine_actions(
    p_new_event_name VARCHAR,
    p_new_event_payload JSONB)
    RETURNS TABLE
            (
                eca_action_name VARCHAR,
                event_payload   JSONB
            )
    LANGUAGE plpgsql
AS
$$
BEGIN
    RETURN QUERY SELECT eca.eca_action_name,
                        p_new_event_payload
                 FROM eca
                 WHERE eca.eca_event_name = p_new_event_name
                   AND evaluate_schema.evaluate(
                               p_new_event_payload,
                               eca.eca_condition) = TRUE;
END
$$;

DROP TRIGGER IF EXISTS evaluate_eca_after_insert_event ON event;

CREATE TRIGGER evaluate_eca_after_insert_event
    AFTER INSERT
    ON event
    FOR EACH ROW
EXECUTE FUNCTION evaluate_eca_rules();
```