```
CREATE TABLE calltraces (
  calltrace_id serial NOT NULL,
  start_time timestamp without time zone,
  end_time timestamp without time zone,
  incomplete boolean NOT NULL DEFAULT false,
  categories integer[],
  location character varying(50),
  tracelogids character varying[],
  reset_tracelogids character varying[],
  sourceaddresses character varying[],
  nurses boolean[],
  cache_np timestamp,
  cache_accepted timestamp,
  cache_cat_changed timestamp,
  CONSTRAINT calltraces_pkey PRIMARY KEY (calltrace_id)
);

CREATE INDEX calltraces_start_time_idx ON calltraces USING btree
(start_time);


CREATE TABLE logentries (
  entryid serial NOT NULL,
  tracelogid character varying(50),
  activityid character varying(50),
  logtime timestamp without time zone,
  uniteaddress character varying(255),
  callid character varying(50),
  "time" timestamp without time zone,
  receivers jsonb,
  props jsonb,
  calltrace_id integer,
  type integer,
  CONSTRAINT test_pkey PRIMARY KEY (entryid),
  CONSTRAINT logentries_calltrace_id_fkey FOREIGN KEY (calltrace_id)
  REFERENCES calltraces (calltrace_id) MATCH SIMPLE
  ON UPDATE CASCADE ON DELETE SET NULL
);

CREATE INDEX test_props_idx ON logentries USING gin (props);

CREATE INDEX logentries_calltrace_id_idx ON logentries USING btree
(calltrace_id);

CREATE INDEX logentries_callid_idx ON public.logentries USING btree
(callid);
```

```
CREATE SEQUENCE next_entryid START 1;
/*
CREATE OR REPLACE VIEW entries_view AS
  SELECT e.entryid,
    e.tracelogid,
    e.activityid,
    e.logtime,
    o.uniteaddress,
    o.callid,
    o."time",
    (( SELECT json_agg(json_build_object('uniteaddress', r.uniteaddress,
'callid', r.callid, 'status', r.status)) AS json_agg
       FROM receivers r
       WHERE e.entryid = r.entryid))::jsonb AS receivers,
    (( SELECT json_object_agg(d.name, d.value) AS json_object_agg
       FROM details d
       WHERE e.entryid = d.entryid))::jsonb AS props
  FROM entries e
    JOIN origins o ON o.entryid = e.entryid;
*/

CREATE SCHEMA sal;


CREATE OR REPLACE FUNCTION sal.get_message_type(activity_id character
varying, props jsonb) RETURNS integer AS
$BODY$
DECLARE
  cancel boolean;

BEGIN
  CASE activity_id
    WHEN 'UNITE.appSpec+NI_LinkingReset' THEN
      RETURN 5; -- LINKING_RESET_MESSAGE
    WHEN 'UNITE.appSpec+NI_NurseLocationInfo' THEN
      RETURN 6; -- NURSE_LOCATION_INFO_MESSAGE
    WHEN 'UNITE.paging+Im' THEN
      RETURN 7; -- INTERACTIVE_MESSAGE
    WHEN 'UNITE.imResponse' THEN
      RETURN 8; -- INTERACTIVE_RESPONSE_MESSAGE
    WHEN 'UNITE.paging' THEN
      RETURN 9; -- PAGING
  ELSE
    IF activity_id = 'UNITE.paging+Im+Presentation' OR activity_id =
'UNITE.appSpec+NI_FullEvent' THEN
      -- Note: some time ago Presentation messages were replaced by
FullEvent messages
      IF activity_id = 'UNITE.appSpec+NI_FullEvent' THEN
        cancel := props @> '{"State":"Clear"}'::jsonb;
```

```
      END IF;
      IF props @> '{"Category":"6"}'::jsonb THEN
        IF cancel THEN
          RETURN 4; -- NP_CANCEL_MESSAGE
        ELSE
          RETURN 3; -- NP_MESSAGE
        END IF;
      ELSE
        IF (props->>'Category') in ('4', '9', '10') THEN
          IF cancel THEN
            RETURN 12; -- CALL_CANCEL_MESSAGE / MedicalCancelMessage
          ELSE
            RETURN 11; -- CALL_MESSAGE / MedicalCallMessage
          END IF;
        ELSE
          IF cancel THEN
            RETURN 2; -- CALL_CANCEL_MESSAGE
          ELSE
            RETURN 1; -- CALL_MESSAGE
          END IF;
        END IF;
      END IF;
    END IF;
  END CASE;

  RETURN 0;
END;
$BODY$
LANGUAGE plpgsql IMMUTABLE COST 100;


-- converting integer level for nurse_presence and cancel_nurse_presence
for correct organization
-- of nurses column in calltraces table (with present nurses for every
level)
CREATE OR REPLACE FUNCTION sal.get_level_np(event logentries)
  RETURNS integer AS
$BODY$DECLARE
  level integer;

BEGIN
  IF event.activityid = 'UNITE.appSpec+NI_FullEvent' THEN
    level := (event.props->>'EventType')::int;
    IF level <= 16 AND level >= 13 THEN
      level := level - 12;
    ELSE
      level := -1;
    END IF;
  ELSE
    level := substring(event.props->>'Alarm data' from '.')::int;
```

```
  END IF;

  RETURN level;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


-- gets input number from medical event (call or cancel call)
CREATE OR REPLACE FUNCTION sal.get_medical_input(event logentries) RETURNS
integer AS
$BODY$
DECLARE
  id integer;

BEGIN
  IF event.activityid = 'UNITE.appSpec+NI_FullEvent' THEN
    id := (event.props->>'EventType')::int;
    IF id >= 6 AND id <= 9 THEN
      RETURN (id - 5);
    ELSE
      RETURN -1;
    END IF;
  ELSE
    id := strpos(event.uniteaddress, '=');
    IF id = 0 THEN
      RETURN 0; -- call from SMA
    ELSE
      RETURN substring(event.uniteaddress from id for 1)::int; -- call
from MMA
    END IF;
  END IF;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


CREATE OR REPLACE FUNCTION sal.get_call_id(call_id character varying,
unite_address character varying) RETURNS character varying AS
$BODY$
DECLARE

BEGIN
  IF call_id IS NOT NULL AND call_id <> '' THEN
    RETURN call_id;
  END IF;

  RETURN coalesce(substring(unite_address from '#"%?#"@%' for '#'), '');
END;
$BODY$
```

```
LANGUAGE plpgsql IMMUTABLE COST 100;


CREATE OR REPLACE FUNCTION sal.get_response_type(response_data character
varying) RETURNS character varying AS
$BODY$
DECLARE

BEGIN
  CASE trim(response_data)
    WHEN 'A', '1' THEN
      RETURN 'ACCEPT';
    WHEN 'B', '2' THEN
      RETURN 'REJECT';
    WHEN '3' THEN
      RETURN 'SPEECH';
    WHEN '4' THEN
      RETURN 'PARK';
    WHEN '5' THEN
      RETURN 'CANCEL';
    WHEN '6' THEN
      RETURN 'CONNECTED';
    ELSE
      RETURN 'UNKNOWN';
  END CASE;
END;
$BODY$
LANGUAGE plpgsql IMMUTABLE COST 100;


CREATE OR REPLACE FUNCTION sal.trigger_tag_event() RETURNS trigger AS
$BODY$
BEGIN
  NEW.type := sal.get_message_type(NEW.activityid, NEW.props);
  CASE NEW.type
    WHEN 3, 4 THEN
      NEW.props := NEW.props || ('{"NP level":' || sal.get_level_np(NEW)
|| '}')::jsonb;
    WHEN 11, 12 THEN
      NEW.props := NEW.props || ('{"Med input":' ||
sal.get_medical_input(NEW) || '}')::jsonb;
    WHEN 7, 9 THEN
      NEW.callid := sal.get_call_id(NEW.receivers#>>'{0,callid}',
NEW.receivers#>>'{0,uniteaddress}');
    WHEN 8 THEN
      IF NEW.callid = '' OR NEW.callid IS NULL THEN
        NEW.callid := sal.get_call_id(NEW.callid, NEW.uniteaddress);
      END IF;
      NEW.props := NEW.props || ('{"Response type":"' ||
sal.get_response_type(NEW.props->>'Response data') || '"}')::jsonb;
```

```
    ELSE

  END CASE;

  RETURN NEW;
END;
$BODY$
LANGUAGE plpgsql IMMUTABLE COST 100;

CREATE TRIGGER trigger_add_event_type BEFORE INSERT ON logentries FOR EACH
ROW EXECUTE PROCEDURE sal.trigger_tag_event();


CREATE OR REPLACE FUNCTION sal.create_calltrace(event logentries, np_level
integer) RETURNS calltraces AS
$BODY$
DECLARE
  categories integer[] := array[]::integer[];
  result calltraces%ROWTYPE;
  nurses_arr boolean[] := array[]::boolean[];
  loc varchar;
  np_time timestamp := NULL;

BEGIN
  IF event.props ? 'Category' THEN
    categories := array[(event.props->>'Category')::integer];
  END IF;

  IF np_level IS NOT NULL THEN
    nurses_arr[np_level] := true;
    np_time := event.time;
  END IF;

  IF event.props ? 'LocationText' THEN
    loc := event.props->>'LocationText';
  ELSE
    loc := event.props->>'Location';
  END IF;

  INSERT INTO calltraces (start_time, categories, location, tracelogids,
reset_tracelogids, sourceaddresses, nurses, cache_np)
  VALUES (event.time, categories, loc, array[event.tracelogid],
array[]::varchar[], array[event.uniteaddress], nurses_arr, np_time)
  RETURNING * INTO result;

  RETURN result;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;
```

```
CREATE OR REPLACE FUNCTION sal.create_calltrace(event logentries) RETURNS
calltraces AS
$BODY$
SELECT sal.create_calltrace(event, null);
$BODY$
LANGUAGE sql VOLATILE COST 100;


-- TODO run this function where java Log Viewer attempts to calculate
length of previous calltrace (and it's existence)
CREATE OR REPLACE FUNCTION sal.finish_calltrace(calltrace calltraces)
RETURNS boolean AS
$BODY$
DECLARE
  finish timestamp;

BEGIN
  IF calltrace.nurses && ARRAY[true] THEN
    -- Here if nurse present in the calltrace ==> calltrace not finished
    RETURN false;
  END IF;

  IF EXISTS(
      SELECT 1 FROM logentries
      WHERE calltrace_id = calltrace.calltrace_id AND type IN (1, 11) AND
NOT (props ? 'Reset entry')
  ) THEN
    -- Here if there are unfinished events in the calltrace ==> calltrace
not finished
    RETURN false;
  END IF;

  SELECT INTO finish max(time) FROM logentries WHERE calltrace_id =
calltrace.calltrace_id AND type IN (2, 12, 4);
  IF finish IS NOT NULL THEN
    -- Finish the calltrace and return true (saying that it is finished)
    UPDATE calltraces SET end_time = finish WHERE calltrace_id =
calltrace.calltrace_id;
    RETURN true;
  END IF;

  RETURN calltrace.incomplete;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


CREATE OR REPLACE FUNCTION sal.add_calltrace_source(calltrace calltraces,
address character varying) RETURNS void AS
```

```
$BODY$
BEGIN
  IF NOT (address = ANY(calltrace.sourceaddresses)) THEN
    UPDATE calltraces SET sourceaddresses = sourceaddresses || address
WHERE calltrace_id = calltrace.calltrace_id;
  END IF;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


-- Every message processing function name must begin with 'message_'
prefix (including PAGING message).
-- Processing function takes row from 'logentries' table as input and
returns calltrace_id or NULL
-- if row should be discarded from parsing process.

-- Processing of 'Nurse presence on' FullEvent
CREATE OR REPLACE FUNCTION sal.message_np(event logentries) RETURNS
integer AS
$BODY$
DECLARE
  level integer;
  calltrace calltraces%ROWTYPE;
  finished boolean;

BEGIN
  level := (event.props->>'NP level')::integer + 2;

  SELECT INTO calltrace * FROM calltraces
  WHERE event.tracelogid = ANY(tracelogids) OR event.tracelogid =
ANY(reset_tracelogids)
  ORDER BY calltrace_id DESC LIMIT 1;

  IF FOUND THEN
    IF event.activityid <> 'UNITE.paging+Im+Presentation' THEN
      -- Note: some time ago Presentation messages were replaced by
FullEvent messages
      PERFORM sal.add_calltrace_source(calltrace, event.uniteaddress);
    END IF;

    finished := sal.finish_calltrace(calltrace);
  ELSE
    -- try to find unfinished calltrace from the location that began from
NURSE_LOCATION_INFO
    SELECT INTO calltrace * FROM calltraces
    WHERE location = event.props->>'LocationText' AND NOT incomplete AND
end_time IS NULL
    ORDER BY calltrace_id DESC LIMIT 1;
```

```
    IF FOUND THEN
      UPDATE calltraces SET tracelogids = tracelogids ||
array[event.tracelogid] WHERE calltrace_id = calltrace.calltrace_id;
      finished := false;
    END IF;
  END IF;

  IF NOT FOUND OR finished OR (calltrace.nurses[level] IS NOT NULL AND
calltrace.nurses[level]) THEN
    IF FOUND THEN
      UPDATE calltraces SET incomplete = true WHERE calltrace_id =
calltrace.calltrace_id;
    END IF;

    calltrace := sal.create_calltrace(event, level);
  ELSE
    calltrace.nurses[level] := true;
    IF NOT ((event.props->>'Category')::integer =
ANY(calltrace.categories)) THEN
      calltrace.categories = calltrace.categories || (event.props-
>>'Category')::integer;
    END IF;
    UPDATE calltraces SET
      cache_np = COALESCE(calltrace.cache_np, event.time),
      nurses = calltrace.nurses,
      categories = calltrace.categories
    WHERE calltrace_id = calltrace.calltrace_id;
  END IF;

  RETURN calltrace.calltrace_id;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


-- Processing of 'Nurse presence off' FullEvent
CREATE OR REPLACE FUNCTION sal.message_np_cancel(event logentries)
  RETURNS integer AS
$BODY$DECLARE
  level integer;
  calltrace calltraces%ROWTYPE;
  np logentries%ROWTYPE;

BEGIN
  SELECT INTO calltrace * FROM calltraces WHERE event.tracelogid =
ANY(tracelogids) ORDER BY calltrace_id DESC LIMIT 1;

  IF FOUND THEN
    IF event.activityid <> 'UNITE.paging+Im+Presentation' THEN
      -- Note: some time ago Presentation messages were replaced by
```

```
FullEvent messages
      PERFORM sal.add_calltrace_source(calltrace, event.uniteaddress);
    END IF;

    IF sal.finish_calltrace(calltrace) THEN
      RETURN null;
    END IF;
  END IF ;

  level := (event.props->>'NP level')::integer + 2;

  IF calltrace.calltrace_id IS NOT NULL AND calltrace.nurses[level] IS NOT
DISTINCT FROM true THEN
    -- Find corresponding 'Nurse presence on' FullEvent
    SELECT INTO np * FROM logentries
    WHERE type = 3
      AND props->>'NP level' = (level - 2)::text
      AND props->>'EventReferenceId' = event.props->>'EventReferenceId'
      AND calltrace_id = calltrace.calltrace_id
    LIMIT 1;

    IF NOT FOUND THEN
      -- Here if strange situation (nurse presence off without
corresponding nurse presence on event)
      UPDATE calltraces SET incomplete = true WHERE calltrace_id =
calltrace.calltrace_id;
    ELSE
      calltrace.nurses[level] := false;
      UPDATE calltraces SET nurses = calltrace.nurses WHERE calltrace_id =
calltrace.calltrace_id;
    END IF;
  END IF;

  RETURN calltrace.calltrace_id;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_call(event logentries)
  RETURNS integer AS
$BODY$
DECLARE
  inp       text;
  calltrace calltraces%rowtype;
  old_call  logentries%rowtype;

BEGIN
  -- Find existing calltrace for the event
  SELECT INTO calltrace * FROM calltraces
```

```
  WHERE event.tracelogid = ANY(reset_tracelogids) AND NOT incomplete AND
end_time IS NULL
  ORDER BY calltrace_id DESC LIMIT 1;

  IF NOT FOUND THEN
    -- Find the last calltrace in the location
    SELECT INTO calltrace * FROM calltraces
    WHERE location = event.props->>'LocationText' ORDER BY calltrace_id
DESC LIMIT 1;
  END IF;

  IF NOT FOUND OR sal.finish_calltrace(calltrace) THEN
    -- Here if calltrace not found, or found and it is finished
    calltrace := sal.create_calltrace(event);
  ELSE
    -- Here if unfinished calltrace was found
    -- Find its init event
    IF event.type = 1 THEN
      SELECT INTO old_call * FROM logentries
      WHERE calltrace_id = calltrace.calltrace_id AND uniteaddress =
event.uniteaddress AND type = 1
      ORDER BY entryid DESC LIMIT 1;
    ELSE
      inp := event.props->>'Med input';
      SELECT INTO old_call l.* FROM logentries l
      WHERE l.calltrace_id = calltrace.calltrace_id AND l.uniteaddress =
event.uniteaddress AND
            l.type = 11 AND l.props->>'Med input' = inp LIMIT 1;
    END IF;

    IF old_call.calltrace_id IS NOT NULL AND NOT (old_call.props ? 'Reset
entry') THEN
      IF old_call.tracelogid <> event.tracelogid THEN
        -- Missed the reset of the old call. --> start a new call trace
        UPDATE calltraces SET incomplete = true WHERE calltrace_id =
calltrace.calltrace_id;
        calltrace := sal.create_calltrace(event);

        RETURN calltrace.calltrace_id;
      ELSE
        IF old_call.props->>'Category' = event.props->>'Category' AND
--           old_call.props->>'CurrentLocationText' IS NOT DISTINCT FROM
event.props->>'CurrentLocationText' THEN
          -- emulate Log Viewer bug (this condition matches only for empty
'CurrentLocationText' in LV source)
          trim(old_call.props->>'CurrentLocationText') = '' AND
trim(event.props->>'CurrentLocationText') = '' THEN
          RETURN NULL; -- call repetition, do not log
        END IF;
        -- need to somehow close previous call (in Java new reset with
```

```
LOCAL_RESET type is created)
        -- or just pass event as a normal (and close old call on the later
stages of processing)
        UPDATE logentries SET props = props || ('{"Reset entry": ' ||
event.entryid || '}') :: jsonb WHERE entryid = old_call.entryid;
        calltrace.cache_cat_changed :=
          CASE WHEN calltrace.cache_cat_changed IS NOT NULL OR
(event.props->>'Category')::integer = ANY(calltrace.categories)
            THEN calltrace.cache_cat_changed
            ELSE event.time
          END;
        UPDATE calltraces SET
          cache_cat_changed = calltrace.cache_cat_changed,
          sourceaddresses = array_append(sourceaddresses,
event.uniteaddress),
          tracelogids = array_append(tracelogids, event.tracelogid),
          -- appends only unique category id to categories
          categories = CASE WHEN (event.props->>'Category')::integer =
ANY(categories)
            THEN categories ELSE array_append(categories, (event.props-
>>'Category')::integer) END
        WHERE calltrace_id = calltrace.calltrace_id;

        RETURN calltrace.calltrace_id;
      END IF;
    END IF;

    calltrace.cache_cat_changed :=
      CASE WHEN calltrace.cache_cat_changed IS NOT NULL OR (event.props-
>>'Category')::integer = ANY(calltrace.categories)
        THEN calltrace.cache_cat_changed
        ELSE event.time
      END;
    calltrace.tracelogids := CASE WHEN event.tracelogid =
ANY(calltrace.tracelogids) THEN calltrace.tracelogids
                              ELSE calltrace.tracelogids ||
event.tracelogid END;
    calltrace.sourceaddresses := CASE WHEN event.uniteaddress =
ANY(calltrace.sourceaddresses) THEN calltrace.sourceaddresses
                                  ELSE calltrace.sourceaddresses ||
event.uniteaddress END;
    calltrace.categories := CASE WHEN (event.props->>'Category')::integer
= ANY(calltrace.categories) THEN calltrace.categories
                              ELSE calltrace.categories || ((event.props-
>>'Category')::integer) END;
    UPDATE calltraces SET
      cache_cat_changed = calltrace.cache_cat_changed,
      sourceaddresses = calltrace.sourceaddresses,
      tracelogids = calltrace.tracelogids,
      categories = calltrace.categories
```

```
    WHERE calltrace_id = calltrace.calltrace_id;
  END IF;

  RETURN calltrace.calltrace_id;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


CREATE OR REPLACE FUNCTION sal.merge_calltrace(event logentries)
  RETURNS integer AS
$BODY$ DECLARE
  first_calltrace calltraces%ROWTYPE;
  second_init_event logentries%ROWTYPE;

  update_calltrace boolean := false;
  entry_id integer; -- the first init event of the first call trace
  le logentries%ROWTYPE;
  nurses_touched boolean := false;
  move_calltraces integer[];
  new_start_time timestamp;
  moved_start_time timestamp;
  new_categories integer[];
  moved_acc_time timestamp := null;
  moved_np_time timestamp := null;
  moved_change_time timestamp := null;

BEGIN

  -- Finds init event of the first call trace
  SELECT INTO entry_id entryid FROM logentries l
  WHERE l.type = 1 AND l.tracelogid = event.tracelogid
  ORDER BY time ASC LIMIT 1;

  IF FOUND THEN
    UPDATE logentries SET props = props || jsonb_build_object ('LR entry',
event.entryid) WHERE entryid = entry_id;
  END IF;

  IF event.props ? 'InitiatorTraceLogId' AND event.props-
>>'InitiatorTraceLogId' != '' THEN
    -- Find first_calltrace (event belongs to first_calltrace)
    SELECT * INTO first_calltrace FROM calltraces c WHERE c.calltrace_id =
event.calltrace_id LIMIT 1;

    -- Find second_init_event, that initiated our event
    SELECT * INTO second_init_event FROM logentries l WHERE l.tracelogid =
event.props->>'InitiatorTraceLogId' ORDER BY entryid LIMIT 1;
    IF NOT FOUND THEN
      RETURN NULL;
```

```
    END IF;


    -- Add 3 fields from second_init_event (not from its calltrace yet) to
first_calltrace
    IF NOT(second_init_event.uniteaddress =
ANY(first_calltrace.sourceaddresses)) THEN
      first_calltrace.sourceaddresses = first_calltrace.sourceaddresses ||
second_init_event.uniteaddress;
      update_calltrace := true;
    END IF;
    IF NOT(second_init_event.tracelogid =
ANY(first_calltrace.reset_tracelogids)) THEN
      first_calltrace.reset_tracelogids =
first_calltrace.reset_tracelogids || second_init_event.tracelogid;
      update_calltrace := true;
    END IF;
    IF NOT(second_init_event.tracelogid =
ANY(first_calltrace.tracelogids)) THEN
      first_calltrace.tracelogids = first_calltrace.tracelogids ||
second_init_event.tracelogid;
      update_calltrace := true;
    END IF;
    IF update_calltrace THEN
      UPDATE calltraces
      SET sourceaddresses = first_calltrace.sourceaddresses,
reset_tracelogids = first_calltrace.reset_tracelogids, tracelogids =
first_calltrace.tracelogids
      WHERE calltrace_id = first_calltrace.calltrace_id;
    END IF;


    -- Find calltraces, corresponding to second_init_event, that will be
removed (merged into first_calltrace)
    IF second_init_event.calltrace_id IS NOT NULL AND event.calltrace_id
<> second_init_event.calltrace_id THEN
      move_calltraces := ARRAY[second_init_event.calltrace_id];
    END IF;

    IF array_length(move_calltraces, 1) > 0 THEN
      -- Removing move_calltraces

      UPDATE logentries SET calltrace_id = first_calltrace.calltrace_id
WHERE calltrace_id = ANY(move_calltraces);

      SELECT INTO moved_start_time, moved_acc_time, moved_np_time,
moved_change_time
        min(start_time), min(cache_accepted), min(cache_np),
min(cache_cat_changed)
      FROM calltraces WHERE calltrace_id = ANY(move_calltraces);
```

```
      SELECT INTO first_calltrace.sourceaddresses array_agg(DISTINCT s)
FROM calltraces, unnest(sourceaddresses) s
      WHERE calltrace_id = ANY(move_calltraces ||
first_calltrace.calltrace_id);

      SELECT INTO first_calltrace.reset_tracelogids array_agg(DISTINCT r)
FROM calltraces, unnest(reset_tracelogids) r
      WHERE calltrace_id = ANY(move_calltraces ||
first_calltrace.calltrace_id);

      SELECT INTO first_calltrace.tracelogids array_agg(DISTINCT r) FROM
calltraces, unnest(tracelogids) r
      WHERE calltrace_id = ANY(move_calltraces ||
first_calltrace.calltrace_id);

      new_start_time := least(moved_start_time,
first_calltrace.start_time);

      SELECT INTO new_categories array_agg(t.a)
      FROM (SELECT DISTINCT unnest(categories) AS a FROM calltraces WHERE
calltrace_id = ANY(move_calltraces)) t
      -- Sets only unique category id into new_categories
      WHERE NOT (t.a = ANY(first_calltrace.categories));

      IF moved_start_time >= first_calltrace.start_time THEN
        first_calltrace.categories := first_calltrace.categories ||
new_categories;

        first_calltrace.cache_cat_changed = CASE WHEN
first_calltrace.cache_cat_changed IS NULL OR
first_calltrace.cache_cat_changed > moved_start_time
          THEN moved_start_time ELSE first_calltrace.cache_cat_changed
END;

        first_calltrace.cache_np := CASE WHEN first_calltrace.cache_np IS
NULL OR first_calltrace.cache_np > moved_np_time
          THEN moved_np_time ELSE first_calltrace.cache_np END;
      ELSE
        first_calltrace.categories := new_categories ||
first_calltrace.categories;
        -- set accepted time only when linked module event happened before
main event and ACCEPT for the first event
        -- must be the only one taken into account
        first_calltrace.cache_accepted := moved_acc_time;

        first_calltrace.cache_cat_changed := CASE WHEN moved_change_time
IS NULL OR first_calltrace.start_time < moved_change_time
          THEN first_calltrace.start_time ELSE moved_change_time END;
```

```
        first_calltrace.cache_np := CASE WHEN moved_np_time IS NULL OR
(first_calltrace.cache_np IS NOT NULL AND first_calltrace.cache_np <
moved_np_time)
          THEN first_calltrace.cache_np ELSE moved_np_time END;
      END IF;

      DELETE FROM calltraces WHERE calltrace_id = ANY(move_calltraces);

      FOR le IN SELECT l.* FROM logentries l WHERE l.calltrace_id =
event.calltrace_id AND l.entryid < event.entryid
                                                    AND l.entryid >
entry_id AND type IN (3,4) LOOP
        IF le.type = 3 THEN
          first_calltrace.nurses[(le.props->>'NP level')::integer + 2] :=
true;
        ELSE
          first_calltrace.nurses[(le.props->>'NP level')::integer + 2] :=
false;
        END IF;
        nurses_touched := true;
      END LOOP;

      IF nurses_touched AND true = ANY(first_calltrace.nurses) THEN
        first_calltrace.end_time = NULL;
      END IF;

      UPDATE calltraces SET
        cache_cat_changed = first_calltrace.cache_cat_changed,
        cache_np = first_calltrace.cache_np,
        cache_accepted = first_calltrace.cache_accepted,
        sourceaddresses = first_calltrace.sourceaddresses,
        tracelogids = first_calltrace.tracelogids,
        reset_tracelogids = first_calltrace.reset_tracelogids,
        nurses = first_calltrace.nurses,
        start_time = new_start_time,
        end_time = first_calltrace.end_time,
        categories = first_calltrace.categories
      WHERE calltrace_id = first_calltrace.calltrace_id;
    END IF;
  END IF;

  RETURN first_calltrace.calltrace_id;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_call_cancel(event logentries)
  RETURNS integer AS
  $BODY$
```

```
DECLARE
  calltrace calltraces%ROWTYPE;
  call logentries%ROWTYPE;
  inp text;

BEGIN
  SELECT INTO calltrace * FROM calltraces
  WHERE event.tracelogid = ANY(tracelogids || reset_tracelogids) AND NOT
incomplete AND end_time IS NULL ORDER BY calltrace_id DESC LIMIT 1;

  IF NOT FOUND THEN
    -- TODO temporarily disabled to match LogViewer variant
    -- SELECT INTO calltrace * FROM calltraces WHERE location =
event.props->>'LocationText' AND NOT incomplete AND end_time IS NULL LIMIT
1;
    SELECT INTO calltrace * FROM calltraces WHERE location = event.props-
>>'LocationText' ORDER BY calltrace_id DESC LIMIT 1;
  END IF;

  IF FOUND THEN
    PERFORM sal.add_calltrace_source(calltrace, event.uniteaddress);
    -- get previous call (or opening event) that is cancelled by this
event
    IF event.type = 12 THEN
      inp := event.props->>'Med input';
      SELECT INTO call * FROM logentries l
      WHERE l.calltrace_id = calltrace.calltrace_id AND l.type = 11 AND
l.props->>'LocationText' = calltrace.location
            AND props->>'Med input' = inp ORDER BY time DESC LIMIT 1;
    ELSE
      SELECT INTO call * FROM logentries l
      WHERE l.calltrace_id = calltrace.calltrace_id AND l.type = 1
            AND (l.props->>'LocationText' = calltrace.location OR
l.tracelogid = ANY(calltrace.tracelogids || calltrace.reset_tracelogids))
            AND event.uniteaddress = l.uniteaddress
            AND ((event.props->>'Category' = '11' AND l.props->>'Category'
= '11')
              OR (event.props->>'EventType' = l.props->>'EventType') )
      ORDER BY time DESC LIMIT 1;
    END IF;

    IF NOT FOUND THEN
      RETURN null;
    END IF;

    IF call.tracelogid <> event.tracelogid THEN
      UPDATE calltraces SET incomplete = true WHERE calltrace_id =
calltrace.calltrace_id;
      RETURN null;
    END IF;
```

```
    -- In opening event, set props 'Reset entry' pointing to closing event
    -- (Existence of 'Reset entry' props in opening event props means that
    -- the opening event was cancelled by corresponding closing event.)
    UPDATE logentries SET props = props || ('{"Reset entry": ' ||
event.entryid || '}')::jsonb WHERE entryid = call.entryid;

    RETURN calltrace.calltrace_id;
  ELSE
    RETURN null;
  END IF;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_nurse_location_info(event
logentries)
  RETURNS integer AS
$BODY$
DECLARE
  calltrace calltraces%ROWTYPE;
  presence logentries%ROWTYPE;
  card_count integer;

BEGIN

  -- Here if current event (the function argument) has NurseLocationInfo
type.
  --
  -- 1) The nurse entered the location using RFID card (when
event.props.NpState present). 2 situations are possible here:
  --   1.a) The event was preceded by FullEvent ("nurse presence on" or
"normal call" etc), that already started new calltrace in the location,
  --     and it is not finished yet, and this calltrace will be found
here, and current event will be just inserted into it.
  --   1.b) The event is the first event of calltrace, so new calltrace
should be created here.
  --
  -- 2) The nurse went out from the location using RFID card (when
event.props.NpState absent). 2 situations are possible here:
  --   2.a) The event came before FullEvent "nurse presence off" that will
finish calltrace. Here we just find current unfinished calltrace
  --     and insert current event into it.
  --   2.b) The event was preceded by FullEvent "nurse presence off" that
already finished calltrace. We will need to find this calltrace
  --     among finished calltraces to insert current event to it.

  -- Find unfinished calltrace, into which current event should be
inserted
```

```
  SELECT INTO calltrace * FROM calltraces
  WHERE event.uniteaddress = ANY(sourceaddresses)
    AND event.tracelogid = ANY(tracelogids || reset_tracelogids)
    AND end_time IS NULL
    AND NOT incomplete
  ORDER BY calltrace_id DESC LIMIT 1;

  IF FOUND THEN
    -- Here if (1.a) or (2.a)
      RETURN calltrace.calltrace_id;
  END IF;

  -- Here if NOT FOUND

  IF event.props ? 'NpState' THEN
    -- Here if (1.b)
    calltrace := sal.create_calltrace(event);
  END IF;

  -- Here if (2.b)

  -- Find already finished calltrace, to insert current event into it
  SELECT INTO calltrace * FROM calltraces
  WHERE event.uniteaddress = ANY(sourceaddresses)
    AND event.tracelogid = ANY(tracelogids || reset_tracelogids)
  ORDER BY calltrace_id DESC LIMIT 1;

  RETURN calltrace.calltrace_id;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_interactive(event logentries)
  RETURNS integer AS
$BODY$
DECLARE
  calltrace calltraces%ROWTYPE;
  receiver_prev varchar;

BEGIN
  SELECT INTO calltrace * FROM calltraces WHERE event.tracelogid =
ANY(tracelogids) ORDER BY calltrace_id DESC LIMIT 1;

  IF FOUND AND sal.finish_calltrace(calltrace) THEN
    IF calltrace.incomplete OR event.time NOT BETWEEN calltrace.start_time
AND calltrace.end_time THEN
      RETURN null;
    END IF;
  END IF;
```

```
  IF calltrace.calltrace_id IS NOT NULL THEN
    SELECT INTO receiver_prev receivers#>>'{0,uniteaddress}' FROM
logentries
    WHERE calltrace_id = calltrace.calltrace_id AND (type = 7 OR type = 9)
AND callid = event.callid ORDER BY entryid DESC LIMIT 1;

    IF FOUND AND receiver_prev <> event.receivers#>>'{0,uniteaddress}'
THEN
      RETURN null;
    END IF;

    IF EXISTS(SELECT 1 FROM logentries WHERE calltrace_id =
calltrace.calltrace_id AND type = 8
                                            AND props->>'Response type' =
'ACCEPT') THEN
      RETURN null;
    END IF;
  END IF;

  RETURN calltrace.calltrace_id;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_interactive_response(event
logentries)
  RETURNS integer AS
$BODY$
DECLARE
  calltrace calltraces%ROWTYPE;
  check_event character varying;

BEGIN

  SELECT INTO calltrace * FROM calltraces WHERE event.tracelogid =
ANY(tracelogids) ORDER BY calltrace_id DESC LIMIT 1;

  IF (event.props->>'Response type') = 'PARK' THEN
    SELECT INTO check_event props->>'Response type' FROM logentries
    WHERE calltrace_id = calltrace.calltrace_id AND type = 8 AND props-
>>'Response type' IN ('SPEECH', 'PARK')
    ORDER BY entryid DESC LIMIT 1;

    IF check_event = 'PARK' THEN
      RETURN null; -- duplicated PARK message
    END IF;
  END IF;
```

```
  IF (event.props->>'Response type') = 'ACCEPT' AND
calltrace.cache_accepted IS NULL THEN
    UPDATE calltraces SET cache_accepted = event.time WHERE calltrace_id =
calltrace.calltrace_id;
  END IF;

  RETURN calltrace.calltrace_id;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.message_paging(event logentries)
  RETURNS integer AS
$BODY$
BEGIN
  RETURN sal.message_interactive(event);
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.process_entry(entry_id integer)
  RETURNS integer AS
$BODY$
DECLARE
  event logentries%ROWTYPE;
  calltrace calltraces%ROWTYPE;
  cid integer := null;

BEGIN
  SELECT INTO event * FROM logentries WHERE entryid = entry_id;

  IF FOUND THEN

    -- Abort calls in parser if they takes more than 3 days
    SELECT INTO calltrace * FROM calltraces
      WHERE event.tracelogid = ANY(tracelogids) OR event.tracelogid =
ANY(reset_tracelogids)
      ORDER BY calltrace_id DESC LIMIT 1;
    IF FOUND AND event.time - calltrace.start_time > interval '3 days'
THEN
      UPDATE calltraces SET incomplete = true WHERE calltrace_id =
calltrace.calltrace_id;
      RETURN event.entryid;
    END IF;

    CASE event.type
      WHEN 1, 11 THEN
        cid := sal.message_call(event);
```

```
      WHEN 2, 12 THEN
        cid := sal.message_call_cancel(event);
      WHEN 3 THEN
        cid := sal.message_np(event);
      WHEN 4 THEN
        cid := sal.message_np_cancel(event);
      WHEN 6 THEN
        cid := sal.message_nurse_location_info(event);
      WHEN 7 THEN
        cid := sal.message_interactive(event);
      WHEN 8 THEN
        cid := sal.message_interactive_response(event);
      WHEN 9 THEN
        cid := sal.message_paging(event);
      ELSE

    END CASE;
    IF cid IS NOT NULL THEN
      IF event.props ? 'InitiatorType' AND event.props->>'InitiatorType'
IN ('LinkedReset', 'ImReset') THEN
        event.calltrace_id := cid;
        PERFORM sal.merge_calltrace(event);
      END IF;
      UPDATE logentries SET calltrace_id = cid WHERE entryid = entry_id;
      IF event.type IN (2, 12, 4) THEN
        PERFORM sal.finish_calltrace(c) FROM calltraces c WHERE
c.calltrace_id = cid;
      END IF;
    END IF;
  END IF;

  RETURN event.entryid;
END;$BODY$
LANGUAGE plpgsql VOLATILE
COST 100;


CREATE OR REPLACE FUNCTION sal.process_all_entries_left(start_entryid
integer) RETURNS integer AS $BODY$
DECLARE
  end_entryid integer;

BEGIN
  SELECT INTO end_entryid max(r.id)
  FROM (
        SELECT sal.process_entry(entryid) AS id
        FROM logentries WHERE entryid >= start_entryid
        ORDER BY entryid
      ) r;
```

```
  RETURN end_entryid + 1;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;

-- To run: SELECT setval('next_entryid',
sal.process_all_entries_left((SELECT last_value::integer FROM
next_entryid)));
-- But better run: SELECT sal.parser_run_local();

-- Run log parser.
CREATE OR REPLACE FUNCTION sal.parser_run() RETURNS integer AS $BODY$
DECLARE
  new_entry_id integer;
BEGIN
  LOCK calltraces IN EXCLUSIVE MODE; -- Prevent several simultaneous log
parser runs.
  SELECT INTO new_entry_id setval('next_entryid',
sal.process_all_entries_left((SELECT last_value::integer FROM
next_entryid)));
  RETURN new_entry_id;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;


-- Run log parser locally.
-- The function helps to avoid erroneous double processing of logentries
table rows
-- in case when several falcons use the same SAL DB and simultaneously run
log parser.
CREATE OR REPLACE FUNCTION sal.parser_run_local() RETURNS integer AS
$BODY$
DECLARE
  new_entry_id integer;
BEGIN
  IF inet_client_addr() = inet_server_addr() THEN
    SELECT INTO new_entry_id FROM sal.parser_run();
  END IF;
  RETURN new_entry_id;
END;
$BODY$
LANGUAGE plpgsql VOLATILE COST 100;
```