Let's Build a Complex, Real-Time Data Management Application

...before the training session ends!

Jonathan S. Katz
PGCon 2018, May 30, 2018

Crunchy Enterprise PostgreSQL

About Crunchy Data

- Leading provider of trusted open source PostgreSQL and PostgreSQL related technologies, support and training to enterprises.
- We're Hiring!
- crunchydata.com

About Me

- Director of Communications & Customer Success, Crunchy Data
- Longtime PostgreSQL community contributor
  - Director, PgUS
  - Co-Organizer, NYCPUG
  - @postgresql + .org content
  - Conference organization + speaking galore!
- @jkatz05

How This is Going To Work

- Setting up Requirements
- Overview
- Code & Build & Test on Loop while exploring different features of PostgreSQL!
- [Discuss!]

Requirements

- PostgreSQL 10
- wal2json: https://github.com/eulerto/wal2json
- Python 3
  - psycopg2

Requirements

- Installing wal2json:

```
git clone https://github.com/eulerto/wal2json.git
cd wal2json
USE_PGXS=1 make
USE_PGXS=1 sudo make install

# [restart postgresql]
```

Requirements

- Set up Python 3 + psycopg2

```
mkdir app
cd app
python3 -m venv envs
. envs/bin/activate
pip install psycopg2
```

The Problem

- Imagine we are managing the rooms at the University of Ottawa
- We have a set of operating hours in which the rooms can be booked
- Only one booking can occur in the room at a given time

	DMS 1160	DMS 1110	DMS 1120	DMS 1150
09:00				**Let's Build a Complex, Real-Time Data Management Application** *...before the training session ends!* **Jonathan S. Katz** Track Tutorial
09:15				
09:30				
09:45				
10:00	**Session Pitches and Scheduling** *Coffee and snacks* **Stephen Frost** Track **Unconference**			
10:15				
10:30				
10:45				
11:00				
11:15	**Unconference** *Room #1* **Stephen Frost** Track **Unconference**	**Unconference** *Room #2* **Stephen Frost** Track **Unconference**		
11:30				
11:45				
12:00				

For example...

We Need to Know

- All the **rooms** that are available to book
- **When** the rooms are available to be booked (operating hours)
- **When** the rooms have been booked

And...

- The system needs to be able to CRUD fast
  - (**C**reate, **R**ead, **U**pdate, **D**elete. Fast).

First, let's talk about how we can find availability

PostgreSQL & Dates + Times

Name	Bytes	Range	Resolution
timestamp without	8	4713 BC to 294276 AD	1 microsecond / 14 digits
timestamp with timezone	8	4713 BC to 294276 AD	1 microsecond / 14 digits
date	4	4713 BC to 5874897 AD	1 day
time without timezone	8	00:00:00 to 24:00:00	1 microsecond / 14 digits
time with timezone	12	00:00:00+1459 to	1 microsecond / 14 digits
interval	12	-178000000 years to	1 microsecond / 14 digits

PostgreSQL & Dates + Times

```
SELECT CURRENT_DATE;
SELECT pg_typeof(CURRENT_DATE);

SELECT CURRENT_TIME;
SELECT pg_typeof(CURRENT_TIME);

SELECT CURRENT_DATE + CURRENT_TIME;
SELECT pg_typeof(CURRENT_DATE + CURRENT_TIME);

SELECT CURRENT_DATE + 3;
SELECT pg_typeof(CURRENT_DATE + 3);
```

PostgreSQL & Dates + Times

```
SELECT date_trunc('week', CURRENT_DATE);
SELECT date_trunc('month', CURRENT_DATE);
SELECT date_trunc('quarter', CURRENT_DATE);
SELECT date_trunc('year', CURRENT_DATE);

SELECT EXTRACT('year' FROM CURRENT_DATE);
SELECT EXTRACT('month' FROM CURRENT_DATE);
SELECT EXTRACT('day' FROM CURRENT_DATE);
SELECT EXTRACT('isodow' FROM CURRENT_DATE);
```

PostgreSQL & Dates + Times

```
SELECT '2018-05-30'::date BETWEEN
    date_trunc('week', '2018-05-30'::date) AND
    date_trunc('week', '2018-05-30'::date) + '1 week'::interval;

SELECT
    ('2018-05-30 09:00'::timestamptz, '2018-05-30 12:30'::timestamptz)
OVERLAPS
    ('2018-05-30 11:00'::timestamptz, '2018-05-30 13:00'::timestamptz);

SELECT
    ('2018-05-30 09:00'::timestamptz, '2018-05-30 12:30'::timestamptz)
OVERLAPS
    ('2018-05-30 12:30'::timestamptz, '2018-05-30 13:00'::timestamptz);
```

PostgreSQL Dates & Times

```
CREATE TABLE bookings (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    start_time timestamptz NOT NULL,
    end_time timestamptz NOT NULL
);

INSERT INTO bookings (start_time, end_time)
SELECT
    '2003-04-01 9:00'::timestamptz + (x || ' days')::interval,
    '2003-04-01 12:30'::timestamptz + (x || ' days')::interval
FROM generate_series(1, 400000) x;

INSERT INTO bookings (start_time, end_time)
SELECT
    '2003-04-01 14:00'::timestamptz + (x || ' days')::interval,
    '2003-04-01 16:00'::timestamptz + (x || ' days')::interval
FROM generate_series(1, 400000) x;

INSERT INTO bookings (start_time, end_time)
SELECT
    '2003-04-01 19:30'::timestamptz + (x || ' days')::interval,
    '2003-04-01 21:00'::timestamptz + (x || ' days')::interval
FROM generate_series(1, 400000) x;
```

PostgreSQL Dates & Times

```sql
CREATE INDEX bookings_start_time_idx ON bookings
    (start_time);

CREATE INDEX bookings_start_time_end_time_idx ON bookings
    (start_time, end_time);

SELECT
pg_size_pretty(pg_relation_size('bookings_start_time_end_time_idx'));

EXPLAIN ANALYZE SELECT *
FROM bookings
WHERE
    ('2018-05-30 09:00'::timestamptz, '2018-05-30 12:30'::timestamptz)
OVERLAPS
    (start_time, end_time);

EXPLAIN ANALYZE SELECT *
FROM bookings
WHERE
    start_time BETWEEN '2018-05-30'::date AND '2018-05-31'::date AND
    end_time BETWEEN '2018-05-30'::date AND '2018-05-31'::date;
```

PostgreSQL Ranges

- PostgreSQL 9.2 introduced "range types" that included the ability to store and efficiently search over ranges of data
- Built-in:
  - Date, Timestamps
  - Integer, Numeric

PostgreSQL Ranges

```
CREATE TABLE bookings2 (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    booking_time tstzrange NOT NULL
);


INSERT INTO bookings2 (booking_time)
SELECT
    tstzrange(
        y.start_time + (x || 'days')::interval,
        y.end_time + (x || 'days')::interval
    )
FROM generate_series(1, 400000) x,
    LATERAL (
        SELECT z.*
        FROM (
            VALUES
                ('2003-04-01 9:00'::timestamptz, '2003-04-01 12:30'::timestamptz),
                ('2003-04-01 14:00'::timestamptz, '2003-04-01 16:30'::timestamptz),
                ('2003-04-01 19:30'::timestamptz, '2003-04-01 21:00'::timestamptz)

        ) z(start_time, end_time)
    ) y;
```

GiST Indexes

- "Generalized Search Tree"
- Balanced, tree-structured
- Allows arbitrary indexing schemes
  - B-trees, R-trees
  - indexing on custom data types
- Supports lots more operators
  - <<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, ~=, &&, <->
- can implement your own indexing scheme

GiST Works With...

- Full-text search
- Arrays
- PostGIS data types (geometry, geography)
  - Geometrics types
- Trigrams (pg_trgm)
- ...and Ranges :-)

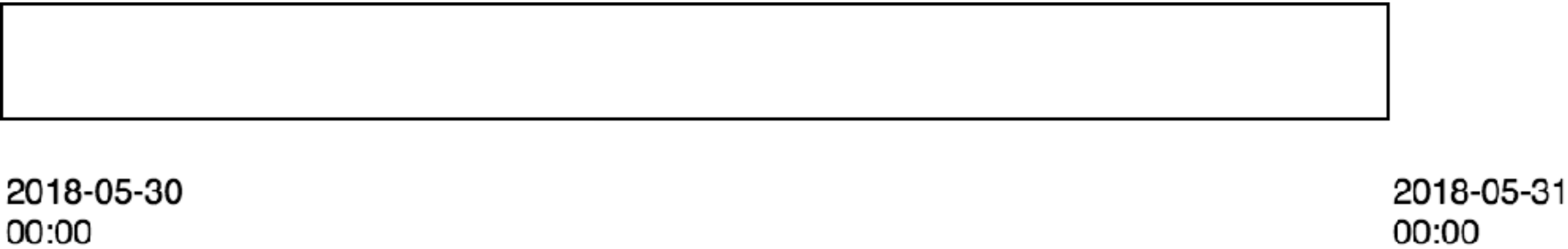
PostgreSQL Ranges

```
CREATE INDEX bookings_booking_time_gist_idx ON bookings2 USING gist(booking_time);

SELECT pg_size_pretty(pg_relation_size('bookings_booking_time_gist_idx'));

EXPLAIN ANALYZE SELECT *
FROM bookings2
WHERE
    tstzrange('2018-05-30 09:00'::timestamptz, '2018-05-30 12:30'::timestamptz) &&
    booking_time;

EXPLAIN ANALYZE SELECT *
FROM bookings2
WHERE booking_time <@ tstzrange('2018-05-30'::date, '2018-05-31'::date);
```

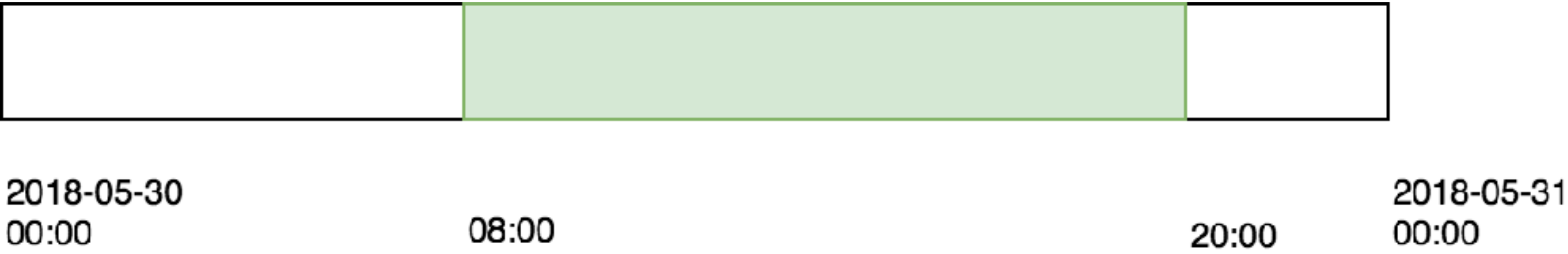
Managing Availability

- Now that we see range types can both simplify programming and improve performance, let's look into managing availability
- Availability can be thought about in three ways:
  - Closed
  - Available
  - Unavailable (or "booked")
- Our ultimate "calendar tuple" is (room, status, range)

Availability



Availability



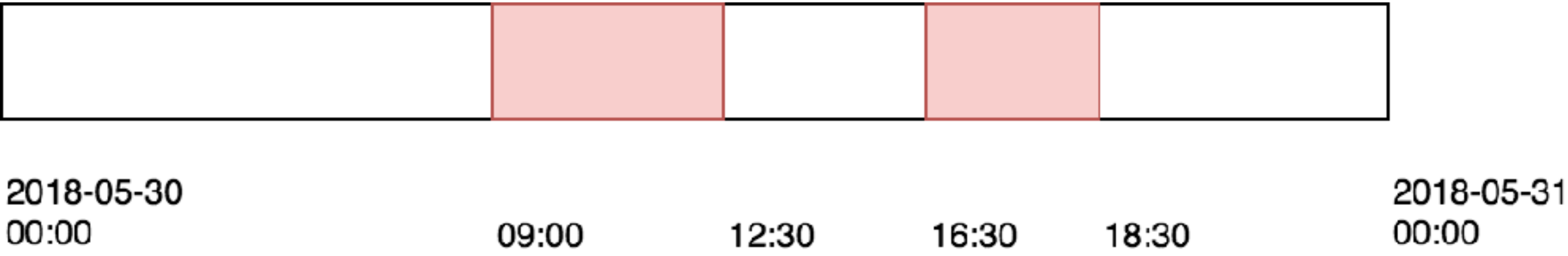
2018-05-30
00:00

08:00

20:00

2018-05-31
00:00

Availability



2018-05-30
00:00

09:00

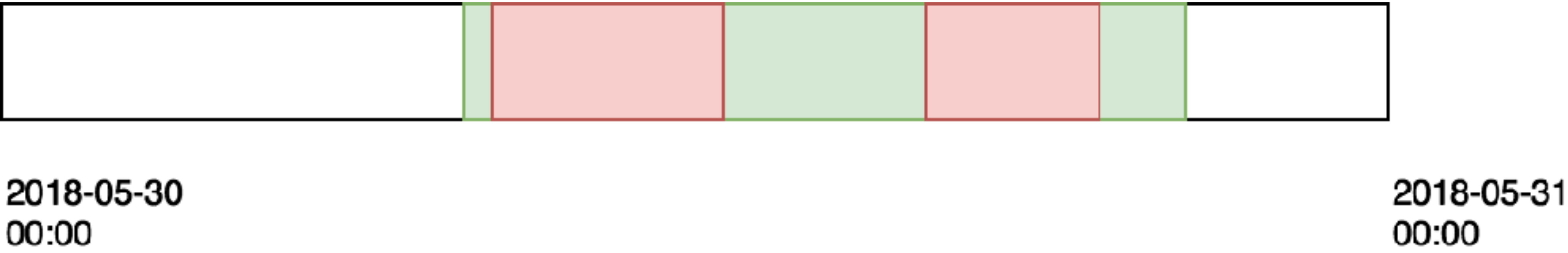
12:30

16:30

18:30

2018-05-31
00:00

Availability



Availability

```
SELECT *
FROM (
    VALUES
        ('closed', tstzrange('2018-05-30', '2018-05-31')),
        ('available', tstzrange('2018-05-30 08:00', '2018-05-30 20:00')),
        ('unavailable', tstzrange('2018-05-30 09:00', '2018-05-30 12:30')),
        ('unavailable', tstzrange('2018-05-30 16:30', '2018-05-30 18:30'))
) x(status, calendar_range);
```

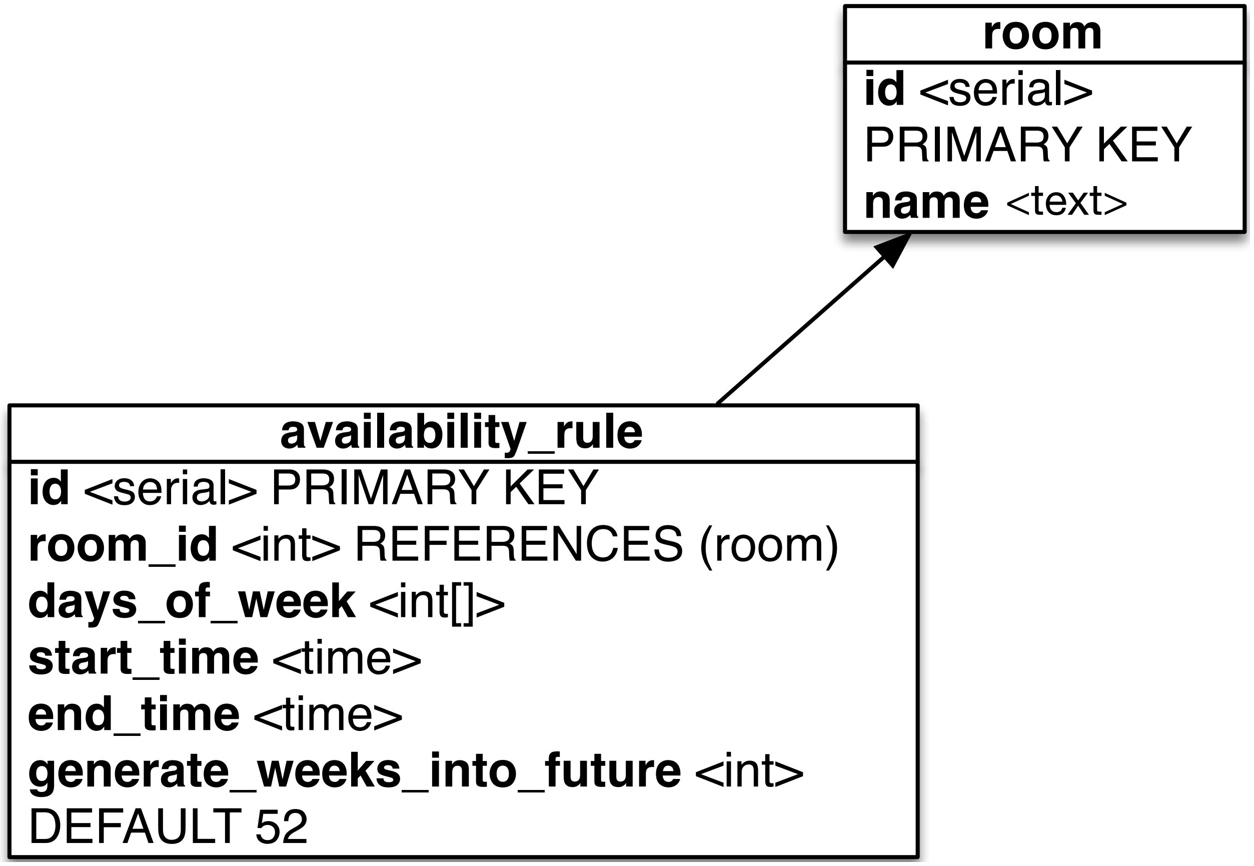
Easy, Right?

But...

- Insert new ranges and dividing them up
    - PostgreSQL does not work well with discontiguous ranges (...yet)
- Availability
    - Just for one day - what about other days?
    - What happens with data in the past?
    - What happens with data in the future?
- Unavailability
    - Ensure no double-bookings
    - Overlapping Events?
- Just one space

Managing Availability

- Can create rules that can generate availability



room
id <serial> PRIMARY KEY
name <text>

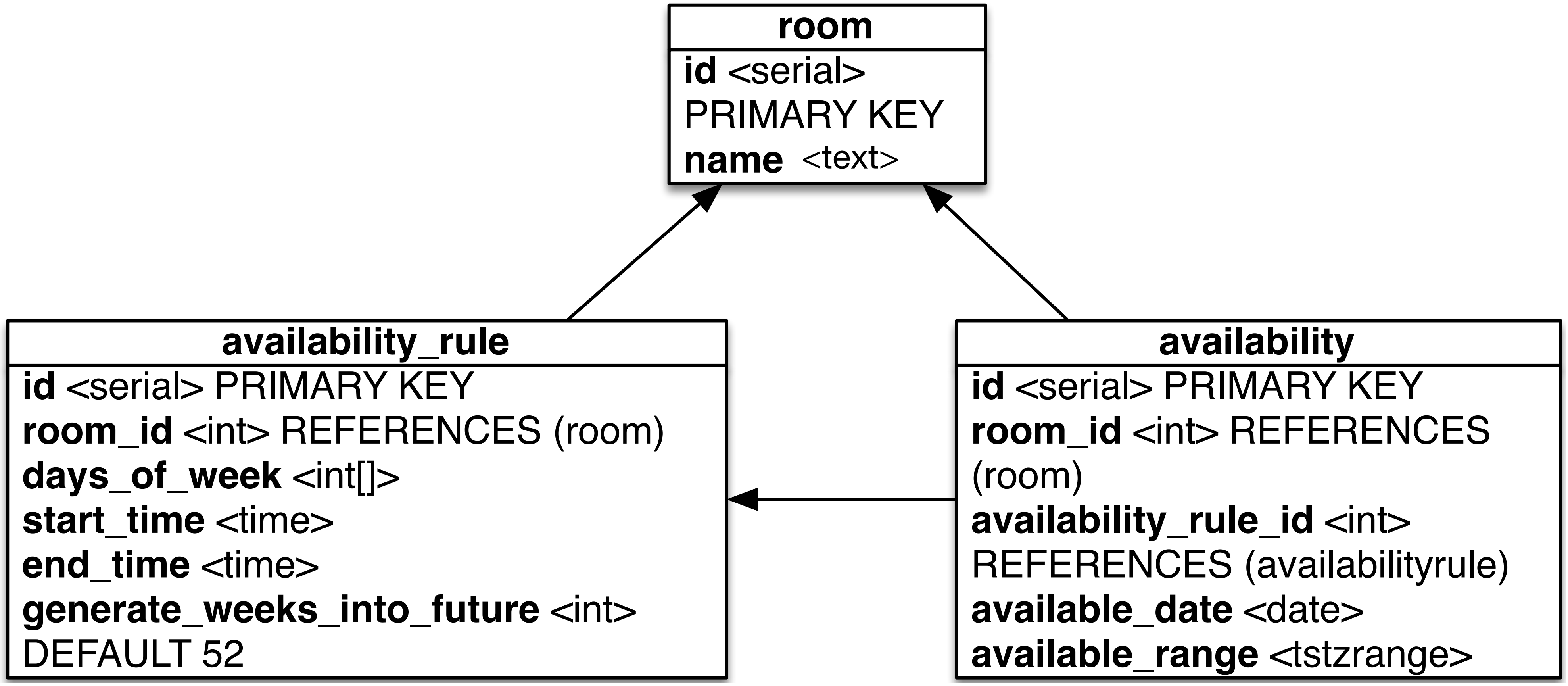
availability_rule
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
days_of_week <int[]>
start_time <time>
end_time <time>
generate_weeks_into_future <int> DEFAULT 52

Managing Availability

```
CREATE TABLE room (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    name text NOT NULL
);

CREATE TABLE availability_rule (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    room_id int NOT NULL REFERENCES room (id) ON DELETE CASCADE,
    days_of_week int[] NOT NULL,
    start_time time NOT NULL,
    end_time time NOT NULL,
    generate_weeks_into_future int NOT NULL DEFAULT 52
);
```

Managing Availability

- The rules can then determines what the availability is for a given date



room
id <serial> PRIMARY KEY
name <text>

availability_rule
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
days_of_week <int[]>
start_time <time>
end_time <time>
generate_weeks_into_future <int> DEFAULT 52

availability
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
availability_rule_id <int> REFERENCES (availabilityrule)
available_date <date>
available_range <tstzrange>

SP-GiST

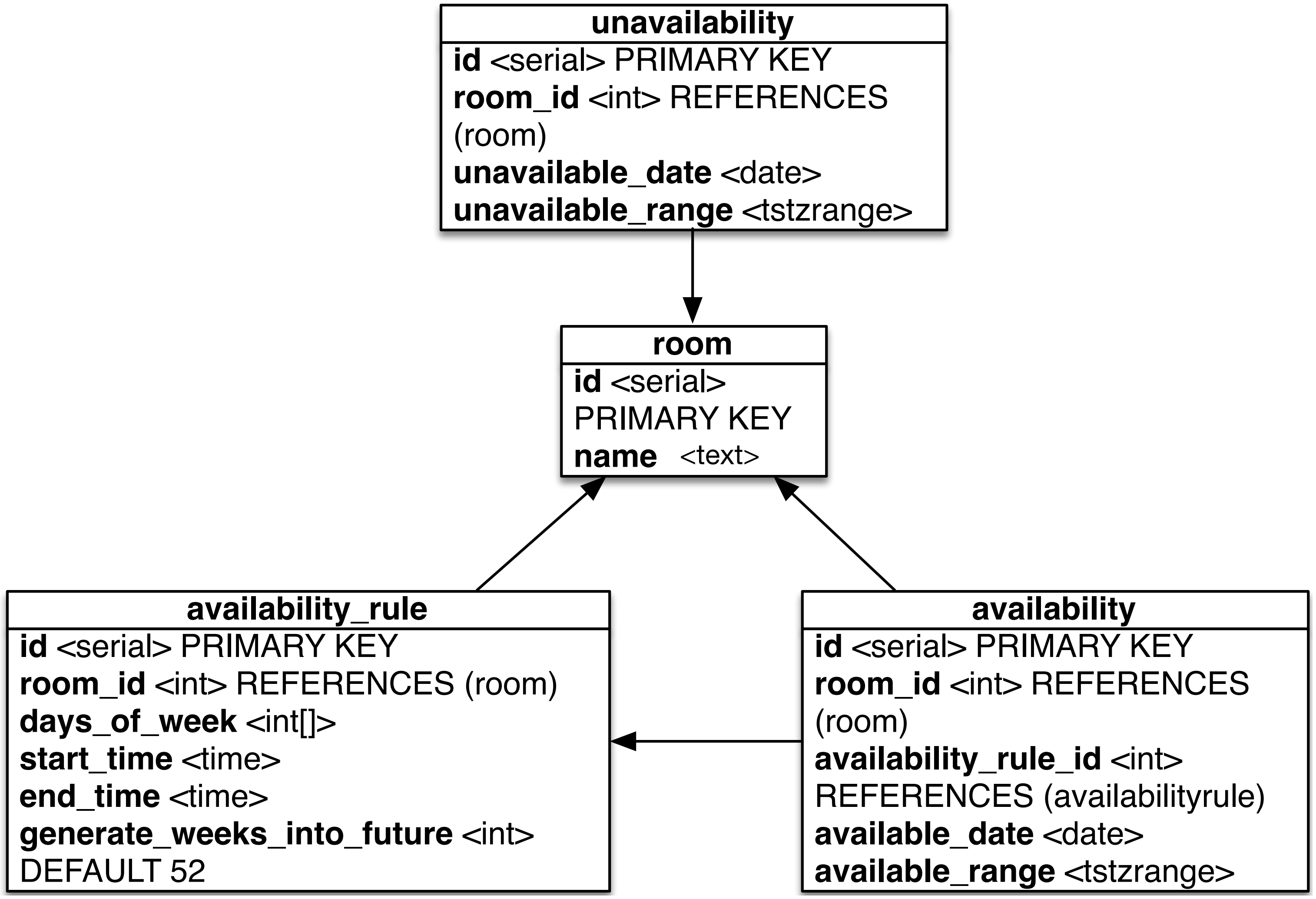
- "Space-partitioned Generalized Search Tree"
- Designed for handling unbalanced data structures
    - quadtrees
    - k-d trees
    - radix trees
- Searches are fast if match partitioning rules

Managing Availability

```
CREATE TABLE availability (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    room_id int NOT NULL REFERENCES room (id) ON DELETE CASCADE,
    availability_rule_id int NOT NULL REFERENCES availability_rule
(id) ON DELETE CASCADE,
    available_date date NOT NULL,
    available_range tstzrange NOT NULL
);
CREATE INDEX availability_available_range_gist_idx
    ON availability
    USING gist(available_range);
```

Managing Availability

- We need to know when a room is being used



unavailability

id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
unavailable_date <date>
unavailable_range <tstzrange>

room

id <serial> PRIMARY KEY
name <text>

availability_rule

id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
days_of_week <int[]>
start_time <time>
end_time <time>
generate_weeks_into_future <int> DEFAULT 52

availability

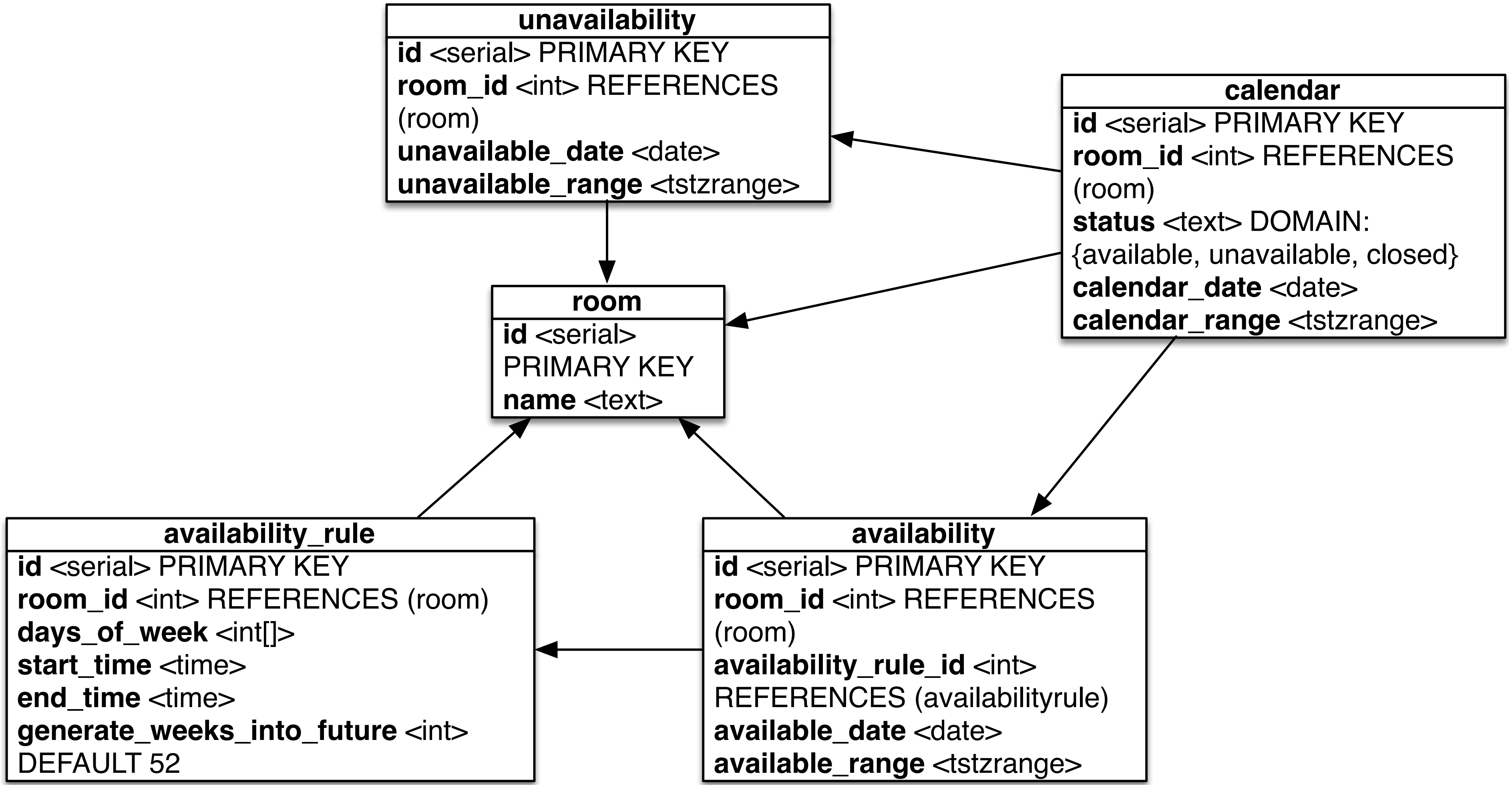
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
availability_rule_id <int> REFERENCES (availabilityrule)
available_date <date>
available_range <tstzrange>

Managing Availability

```
CREATE TABLE unavailability (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    room_id int NOT NULL REFERENCES room (id) ON DELETE CASCADE,
    unavailable_date date NOT NULL,
    unavailable_range tstzrange NOT NULL
);
CREATE INDEX unavailability_unavailable_range_gist_idx
    ON unavailability
    USING gist(unavailable_range);
```

Managing Availability

- And we can have a calendar set up for quick lookups



unavailability
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
unavailable_date <date>
unavailable_range <tstzrange>

calendar
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
status <text> DOMAIN: {available, unavailable, closed}
calendar_date <date>
calendar_range <tstzrange>

room
id <serial> PRIMARY KEY
name <text>

availability_rule
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
days_of_week <int[]>
start_time <time>
end_time <time>
generate_weeks_into_future <int> DEFAULT 52

availability
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
availability_rule_id <int> REFERENCES (availabilityrule)
available_date <date>
available_range <tstzrange>

Managing Availability

```
CREATE TABLE calendar (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    room_id int NOT NULL REFERENCES room (id) ON DELETE CASCADE,
    status text NOT NULL,
    calendar_date date NOT NULL,
    calendar_range tstzrange NOT NULL
);
CREATE INDEX calendar_room_id_calendar_date_idx
    ON calendar (room_id, calendar_date);
```

Managing Availability

- We can now store data, but what about:
    - Generating initial calendar?
    - Generating availability based on rules?
    - Generating unavailability?
- Sounds like we need to build an application

Managing Availability

- To build our application, there are a few topics we will need to explore first:
    - generate_series
    - Recursive queries
    - SQL Functions
        - Set returning functions
    - PL/pgsql
    - Triggers

Shall we take a break?

generate_series:
More than just generating test data

- Generate series is a "set returning" function, i.e. a function that can return multiple rows of data
- Generate series can return:
    - A set of numbers (int, bigint, numeric) either incremented by 1 or some other integer interval
    - A set of timestamps incremented by a time interval(!!)

generate_series:
More than just generating test data

```
SELECT x::date
FROM generate_series('2018-01-01', '2018-12-31', '1 day'::interval) x;
```

Recursion in my SQL?

- PostgreSQL 8.4 introduced the "WITH" syntax and with it also introduced the ability to perform recursive queries
  - WITH RECURSIVE ... AS ()
- Base case vs. recursive case
- UNION vs. UNION ALL
- ***CAN HIT INFINITE LOOPS***

Recursion in my SQL?

```
WITH RECURSIVE fac AS (
    SELECT
        1::numeric AS n,
        1::numeric AS i
    UNION
    SELECT
        fac.n * (fac.i + 1),
        fac.i + 1 AS i
    FROM fac
    WHERE i + 1 <= 100
)
SELECT fac.n, fac.i
FROM fac;
```

Recursion in my SQL?

```
WITH RECURSIVE fac AS (
    SELECT
        1::numeric AS n,
        1::numeric AS i
    UNION
    SELECT
        fac.n * (fac.i + 1),
        fac.i + 1 AS i
    FROM fac
)
SELECT fac.n, fac.i
FROM fac;
```

Recursion in my SQL?

```
WITH RECURSIVE fac AS (
    SELECT
        1::numeric AS n,
        1::numeric AS i
    UNION
    SELECT
        fac.n * (fac.i + 1),
        fac.i + 1 AS i
    FROM fac
    WHERE i + 1 <= 100
)
SELECT max(fac.n)
FROM fac;
```

Functions

- PostgreSQL provides the ability to write functions to help encapsulate repeated behavior
  - PostgreSQL 11 introduces stored procedures which enables you to embed transactions within functions!
- SQL functions have many properties, including:
  - Input / output
  - Volatility (IMMUTABLE, STABLE, VOLATILE) (default VOLATILE)
  - Parallel safety (default PARALLEL UNSAFE)
  - LEAKPROOF; SECURITY DEFINER
  - Execution Cost
  - Language type (more on this later)

Functions

```
CREATE OR REPLACE FUNCTION pgcon_fac(n int)
RETURNS numeric
AS $$
    WITH RECURSIVE fac AS (
        SELECT
            1::numeric AS n,
            1::numeric AS i
        UNION
        SELECT
            fac.n * (fac.i + 1),
            fac.i + 1 AS i
        FROM fac
        WHERE i + 1 <= $1
    )
    SELECT max(fac.n)
    FROM fac;
$$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;

SELECT pgcon_fac(100);
SELECT pgcon_fac(1000);
SELECT pgcon_fac(10000);

EXPLAIN ANALYZE SELECT pgcon_fac(100);
EXPLAIN ANALYZE SELECT pgcon_fac(1000);
EXPLAIN ANALYZE SELECT pgcon_fac(10000);
```

Functions

```
CREATE OR REPLACE FUNCTION pgcon_fac_set(n int)
RETURNS SETOF numeric
AS $$
    WITH RECURSIVE fac AS (
        SELECT
            1::numeric AS n,
            1::numeric AS i
        UNION
        SELECT
            fac.n * (fac.i + 1),
            fac.i + 1 AS i
        FROM fac
        WHERE i + 1 <= $1
    )
    SELECT fac.n
    FROM fac
    ORDER BY fac.n;
$$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;

SELECT pgcon_fac_set(100);
SELECT pgcon_fac_set(1000);

EXPLAIN ANALYZE SELECT pgcon_fac_set(100);
EXPLAIN ANALYZE SELECT pgcon_fac_set(1000);
EXPLAIN ANALYZE SELECT pgcon_fac_set(10000);
```

Functions

```
CREATE OR REPLACE FUNCTION pgcon_fac_table(n int)
RETURNS TABLE(n numeric)
AS $$
    WITH RECURSIVE fac AS (
        SELECT
            1::numeric AS n,
            1::numeric AS i
        UNION
        SELECT
            fac.n * (fac.i + 1),
            fac.i + 1 AS i
        FROM fac
        WHERE i + 1 <= $1
    )
    SELECT fac.n
    FROM fac
    ORDER BY fac.n;
$$ LANGUAGE SQL IMMUTABLE PARALLEL SAFE;

SELECT pgcon_fac_table(100);
SELECT pgcon_fac_table(1000);
SELECT pgcon_fac_table(10000);

EXPLAIN ANALYZE SELECT pgcon_fac_table(100);
EXPLAIN ANALYZE SELECT pgcon_fac_table(1000);
EXPLAIN ANALYZE SELECT pgcon_fac_table(10000);
```

Procedural Languages

- PostgreSQL has the ability to load in procedural languages and execute code in them beyond SQL
  - "PL"
- Built-in: pgSQL, Python, Perl, Tcl
- Others: Javascript, R, Java, C, JVM, Container, LOLCODE, Ruby, PHP, Lua, pgPSM, Scheme

PL/pgSQL

```
CREATE EXTENSION IF NOT EXISTS plpgsql;

CREATE OR REPLACE FUNCTION pgcon_fac_plpgsql(n int)
RETURNS numeric
AS $$
    DECLARE
        fac numeric;
        i int;
    BEGIN
        fac := 1;
        FOR i IN 1..n LOOP
            fac := fac * i;
        END LOOP;
        RETURN fac;
    END;
$$ LANGUAGE plpgsql IMMUTABLE PARALLEL SAFE;

SELECT pgcon_fac_plpgsql(100);
SELECT pgcon_fac_plpgsql(1000);
SELECT pgcon_fac_plpgsql(10000);

EXPLAIN ANALYZE SELECT pgcon_fac_plpgsql(100);
EXPLAIN ANALYZE SELECT pgcon_fac_plpgsql(1000);
EXPLAIN ANALYZE SELECT pgcon_fac_plpgsql(10000);
```

PL/pgSQL

```
SELECT pgcon_fac(100);
SELECT pgcon_fac(1000);
SELECT pgcon_fac(10000);

SELECT pgcon_fac_plpgsql(100);
SELECT pgcon_fac_plpgsql(1000);
SELECT pgcon_fac_plpgsql(10000);
```

Triggers

- Triggers are functions that can be called before/after/instead of an operation or event
  - Data changes (INSERT/UPDATE/DELETE)
  - Events (DDL, DCL, etc. changes)
- Atomic
- Must return "trigger" or "event_trigger"
  - (Return "NULL" in a trigger if you want to skip operation)
    - (Gotcha: RETURN OLD [INSERT] / RETURN NEW [DELETE])
- Execute once per modified row or once per SQL statement
  - Multiple triggers on same event will execute in **alphabetical** order
- Writeable in any PL language that defined trigger interface

Triggers

NEW	Contains the NEW data to be saved (INSERT / UPDATE)
OLD	Contains the previous data available (UPDATE / DELETE)
TG_OP	INSERT, UPDATE, DELETE
TG_NAME	Name of the trigger that fired
TG_TABLE_NAME	Name of the table that fired the trigger
TG_TABLE_SCHEMA	Name of the schema the table is in that fired the trigger
TG_WHEN	BEFORE, AFTER, INSTEAD OF
TG_LEVEL	ROW, STATEMENT

Triggers

```
CREATE EXTENSION IF NOT EXISTS pgcrypto;

CREATE TABLE a (
    id int GENERATED BY DEFAULT AS IDENTITY (START WITH 1 INCREMENT BY 7)
PRIMARY KEY,
    last_name text NOT NULL,
    secret text NOT NULL,
    updated_at timestamptz NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE b (
    id int GENERATED BY DEFAULT AS IDENTITY PRIMARY KEY,
    a_id int NOT NULL REFERENCES a (id) ON DELETE CASCADE INITIALLY
DEFERRED,
    iv int NOT NULL,
    secret text NOT NULL,
    updated_at timestamptz NOT NULL DEFAULT CURRENT_TIMESTAMP
);
```

Triggers

```
CREATE OR REPLACE FUNCTION a_sync_trigger()
RETURNS trigger
AS $$
    DECLARE
        i int;
        secret text;
    BEGIN
        NEW.secret := encode(digest(NEW.id::text || NEW.last_name ||
CURRENT_TIMESTAMP, 'sha512'), 'hex');
        IF TG_OP = 'INSERT' THEN
            FOR i IN 1..10 LOOP
                secret := encode(digest(i::text || NEW.secret, 'sha512'), 'hex');
                INSERT INTO b (a_id, iv, secret) VALUES (NEW.id, i, secret);
            END LOOP;
        ELSIF TG_OP = 'UPDATE' THEN
            NEW.updated_at = CURRENT_TIMESTAMP;
            UPDATE b
            SET
                secret = encode(digest(iv::text || NEW.secret, 'sha512'), 'hex'),
                updated_at = CURRENT_TIMESTAMP
            WHERE b.a_id = NEW.id;
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
```

Triggers

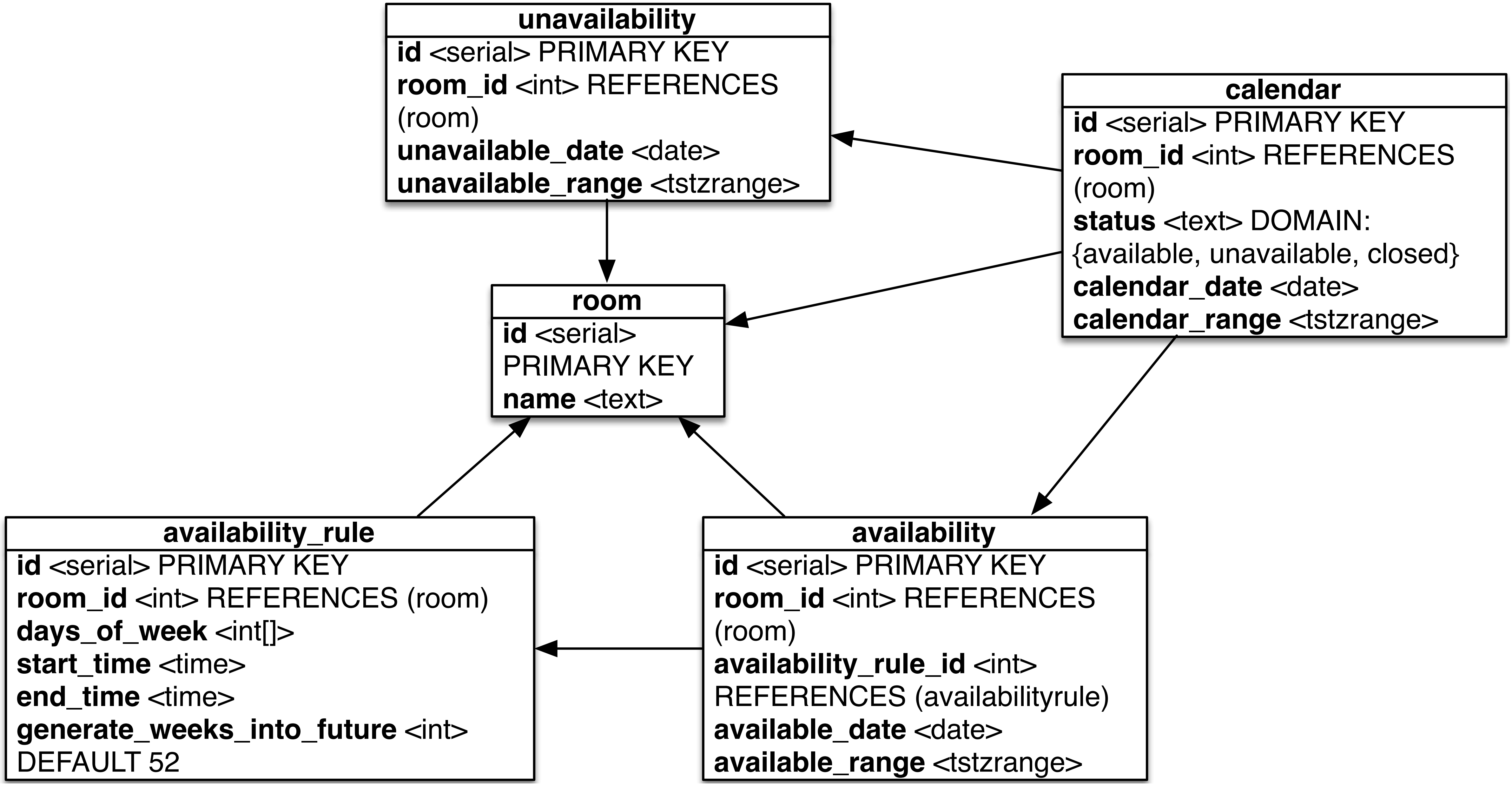
```
CREATE TRIGGER a_sync_trigger_insert
BEFORE INSERT ON a
FOR EACH ROW
EXECUTE PROCEDURE a_sync_trigger();

CREATE TRIGGER a_sync_trigger_update
BEFORE UPDATE ON a
FOR EACH ROW
WHEN (
    OLD.last_name IS DISTINCT FROM NEW.last_name
)
EXECUTE PROCEDURE a_sync_trigger();

INSERT INTO a (last_name) VALUES ('KATZ');
TABLE a;
TABLE b;
```

Triggers

```
UPDATE a SET last_name = 'KATZ';
TABLE a;
TABLE b;

UPDATE a SET updated_at = CURRENT_TIMESTAMP;
TABLE a;
TABLE b;

UPDATE a SET last_name = 'K';
TABLE a;
TABLE b;
```

Okay,
Let's Build Something Real™

Recall Our Data Structure



unavailability
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
unavailable_date <date>
unavailable_range <tstzrange>

calendar
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
status <text> DOMAIN: {available, unavailable, closed}
calendar_date <date>
calendar_range <tstzrange>

room
id <serial> PRIMARY KEY
name <text>

availability_rule
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
days_of_week <int[]>
start_time <time>
end_time <time>
generate_weeks_into_future <int> DEFAULT 52

availability
id <serial> PRIMARY KEY
room_id <int> REFERENCES (room)
availability_rule_id <int> REFERENCES (availabilityrule)
available_date <date>
available_range <tstzrange>

Building a Synchronized System

```
/**
 * ROOM: Need to create an initial calendar for all the data in the room
indicating all
 *   the times that everything is closed
 */
CREATE OR REPLACE FUNCTION room_insert()
RETURNS trigger
AS $$
    BEGIN
        IF TG_OP = 'INSERT' THEN
             INSERT INTO calendar (
                room_id,
                status,
                calendar_date,
                calendar_range
            )
            SELECT
                NEW.id, 'closed', calendar_date, tstzrange(calendar_date,
calendar_date + '1 day'::interval)
            FROM generate_series(
                date_trunc('week', CURRENT_DATE),
                date_trunc('week', CURRENT_DATE + '52 weeks'::interval),
                '1 day'::interval
            ) calendar_date;
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;
```

```
/**
 * Only need to fire trigger inserting a room as UPDATEs do not affect the timings and DELETEs
 * are cascaded
 */
CREATE TRIGGER room_insert
AFTER INSERT ON room
FOR EACH ROW
EXECUTE PROCEDURE room_insert();
```

```
/**
 * AVAILABILITY RULE: Ensure that updates to general availability
 */
/** Helper: Bulk create availability rules; day_of_week ~ isodow (Mon: 1 - Sat: 7) */
CREATE OR REPLACE FUNCTION availability_rule_bulk_insert(availability_rule
availability_rule, day_of_week int)
RETURNS void
AS $$
    INSERT INTO availability (
        room_id,
        availability_rule_id,
        available_date,
        available_range
    )
    SELECT
        $1.room_id,
        $1.id,
        available_date::date + $2 - 1,
        tstzrange(
            /** start of range */
            (available_date::date + $2 - 1) + $1.start_time,
            /** end of range */
            /** check if there is a time wraparound, if so, increment by a day */
            CASE $1.end_time <= $1.start_time
                WHEN TRUE THEN (available_date::date + $2) + $1.end_time
                ELSE (available_date::date + $2 - 1) + $1.end_time
            END
        )
    FROM
        generate_series(
            date_trunc('week', CURRENT_DATE),
            date_trunc('week', CURRENT_DATE) + ($1.generate_weeks_into_future::text || '
weeks')::interval,
            '1 week'::interval
        ) available_date;
$$ LANGUAGE SQL;
```

```
/**
 * availability_rule trigger function
 */
CREATE OR REPLACE FUNCTION availability_rule_manage()
RETURNS trigger
AS $trigger$
    DECLARE
        day_of_week int;
    BEGIN
        IF TG_OP = 'INSERT' THEN
            /** Loop over the days of the week */
            FOREACH day_of_week IN ARRAY NEW.days_of_week
            LOOP
                PERFORM availability_rule_bulk_insert(NEW, day_of_week);
            END LOOP;
```

```
        ELSIF TG_OP = 'UPDATE' THEN
            /** Update is tricky if the days_of_week has changed */
            IF OLD.days_of_week IS DISTINCT FROM NEW.days_of_week THEN
                /** NAIVE: We will delete everything and re-insert */
                DELETE FROM availability
                WHERE availability_rule_id = NEW.id;
                /** insertion */
                FOREACH day_of_week IN ARRAY NEW.days_of_week
                LOOP
                    PERFORM availability_rule_bulk_insert(NEW, day_of_week);
                END LOOP;
            ELSE
                /** Otherwise, just modify the start/end time ranges */
                UPDATE availability
                SET
                    available_range = tstzrange(
                        /** start of range */
                        available_date + NEW.start_time,
                        /** end of range */
                        /** check if there is a time wraparound, if so, increment by a day */
                        CASE NEW.end_time <= NEW.start_time
                            WHEN TRUE THEN (available_date + 1) + NEW.end_time
                            ELSE available_date + NEW.end_time
                        END
                    )
                WHERE availability_rule_id = NEW.id;
            END IF;
        END IF;
        RETURN NEW;
    END;
$trigger$
LANGUAGE plpgsql;
```

```
/** availability_rule trigger only fires on insert or update as DELETE is
cascaded */
CREATE TRIGGER availability_rule_insert_or_update
AFTER INSERT OR UPDATE ON availability_rule
FOR EACH ROW
EXECUTE PROCEDURE availability_rule_manage();
```

```
/** AVAILABILITY, UNAVAILABILITY, and CALENDAR */
/** We need some lengthy functions to help generate the calendar */
```

```sql
/** Helper function: generate the available chunks of time within a block of time for a day within a calendar */
CREATE OR REPLACE FUNCTION calendar_generate_available(room_id int, calendar_range tstzrange)
RETURNS TABLE(status text, calendar_range tstzrange)
AS $$
WITH RECURSIVE availables AS (
    SELECT
        'closed' AS left_status,
        CASE
            WHEN availability.id IS NULL THEN tstzrange(calendar_date, calendar_date + '1 day'::interval)
            ELSE
                tstzrange(
                    calendar_date,
                    lower(availability.available_range * tstzrange(calendar_date, calendar_date + '1 day'::interval))
                )
        END AS left_range,
        CASE isempty(availability.available_range * tstzrange(calendar_date, calendar_date + '1 day'::interval))
            WHEN TRUE THEN 'closed'
            ELSE 'available'
        END AS center_status,
        availability.available_range * tstzrange(calendar_date, calendar_date + '1 day'::interval) AS center_range,
        'closed' AS right_status,
        CASE
            WHEN availability.id IS NULL THEN tstzrange(calendar_date, calendar_date + '1 day'::interval)
            ELSE
                tstzrange(
                    upper(availability.available_range * tstzrange(calendar_date, calendar_date + '1 day'::interval)),
                    calendar_date + '1 day'::interval
                )
        END AS right_range
    FROM generate_series(lower($2), upper($2), '1 day'::interval) AS calendar_date
    LEFT OUTER JOIN availability ON
        availability.room_id = $1 AND
        availability.available_range && $2
    UNION
    SELECT
        'closed' AS left_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                tstzrange(
                    lower(availables.left_range),
                    lower(availables.left_range * availability.available_range)
                )
            ELSE
                tstzrange(
                    lower(availables.right_range),
                    lower(availables.right_range * availability.available_range)
                )
        END AS left_range,
        CASE
            WHEN
                availability.available_range && availables.left_range OR
                availability.available_range && availables.right_range
            THEN 'available'
            ELSE 'closed'
        END AS center_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                availability.available_range * availables.left_range
            ELSE
                availability.available_range * availables.right_range
        END AS center_range,
        'closed' AS right_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                tstzrange(
                    upper(availables.left_range * availability.available_range),
                    upper(availables.left_range)
                )
            ELSE
                tstzrange(
                    upper(availables.right_range * availability.available_range),
                    upper(availables.right_range)
                )
        END AS right_range
    FROM availables
    JOIN availability ON
        availability.room_id = $1 AND
        availability.available_range && $2 AND
        availability.available_range <> availables.center_range AND (
            availability.available_range && availables.left_range OR
            availability.available_range && availables.right_range
        )
)
SELECT *
FROM (
    SELECT
        x.left_status AS status,
        x.left_range AS calendar_range
    FROM availables x
    LEFT OUTER JOIN availables y ON
        x.left_range <> y.left_range AND
        x.left_range @> y.left_range
    GROUP BY 1, 2
    HAVING NOT bool_or(COALESCE(x.left_range @> y.left_range, FALSE))
    UNION
    SELECT DISTINCT
        x.center_status AS status,
        x.center_range AS calendar_range
    FROM availables x
    UNION
    SELECT
        x.right_status AS status,
        x.right_range AS calendar_range
    FROM availables x
    LEFT OUTER JOIN availables y ON
        x.right_range <> y.right_range AND
        x.right_range @> y.right_range
    GROUP BY 1, 2
    HAVING NOT bool_or(COALESCE(x.right_range @> y.right_range, FALSE))
) x
WHERE
    NOT isempty(x.calendar_range) AND
    NOT lower_inf(x.calendar_range) AND
    NOT upper_inf(x.calendar_range) AND
    x.calendar_range <@ $2
$$ LANGUAGE SQL STABLE;
```

This is the first of two helpers functions...

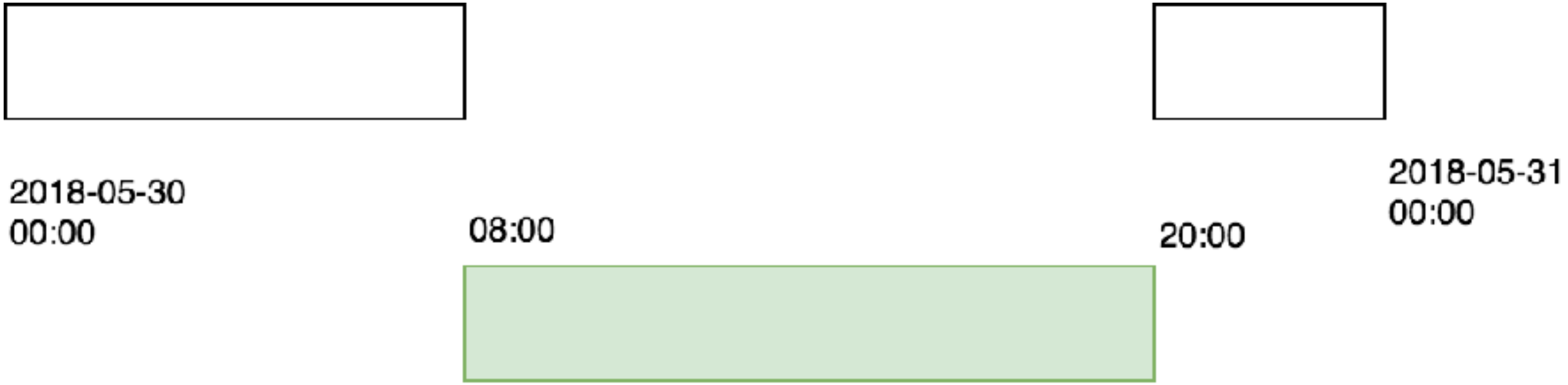
For this experiment

- We will have two availability rules:
  - Open every day 8am - 8pm
  - Open every day 9pm - 10:30pm

```
INSERT INTO room (name) VALUES ('Test Room');

INSERT INTO availability_rule
    (room_id, days_of_week, start_time, end_time)
VALUES
    (1, ARRAY[1,2,3,4,5,6,7], '08:00', '20:00'),
    (1, ARRAY[1,2,3,4,5,6,7], '21:00', '22:30');
```

```
/** Helper function: generate the available chunks of time within a
block of time for a day within a calendar */
CREATE OR REPLACE FUNCTION calendar_generate_available(room_id int,
calendar_range tstzrange)
RETURNS TABLE(status text, calendar_range tstzrange)
AS $$
```

```
WITH RECURSIVE availables AS (
    SELECT
        'closed' AS left_status,
        CASE
            WHEN availability.id IS NULL THEN tstzrange(calendar_date, calendar_date + '1
day'::interval)
            ELSE
                tstzrange(
                    calendar_date,
                    lower(availability.available_range * tstzrange(calendar_date,
calendar_date + '1 day'::interval))
                )
        END AS left_range,
        CASE isempty(availability.available_range * tstzrange(calendar_date, calendar_date +
'1 day'::interval))
            WHEN TRUE THEN 'closed'
            ELSE 'available'
        END AS center_status,
        availability.available_range * tstzrange(calendar_date, calendar_date + '1
day'::interval) AS center_range,
        'closed' AS right_status,
        CASE
            WHEN availability.id IS NULL THEN tstzrange(calendar_date, calendar_date + '1
day'::interval)
            ELSE
                tstzrange(
                    upper(availability.available_range * tstzrange(calendar_date,
calendar_date + '1 day'::interval)),
                    calendar_date + '1 day'::interval
                )
        END AS right_range
    FROM generate_series(lower($2), upper($2), '1 day'::interval) AS calendar_date
    LEFT OUTER JOIN availability ON
        availability.room_id = $1 AND
        availability.available_range && $2
```

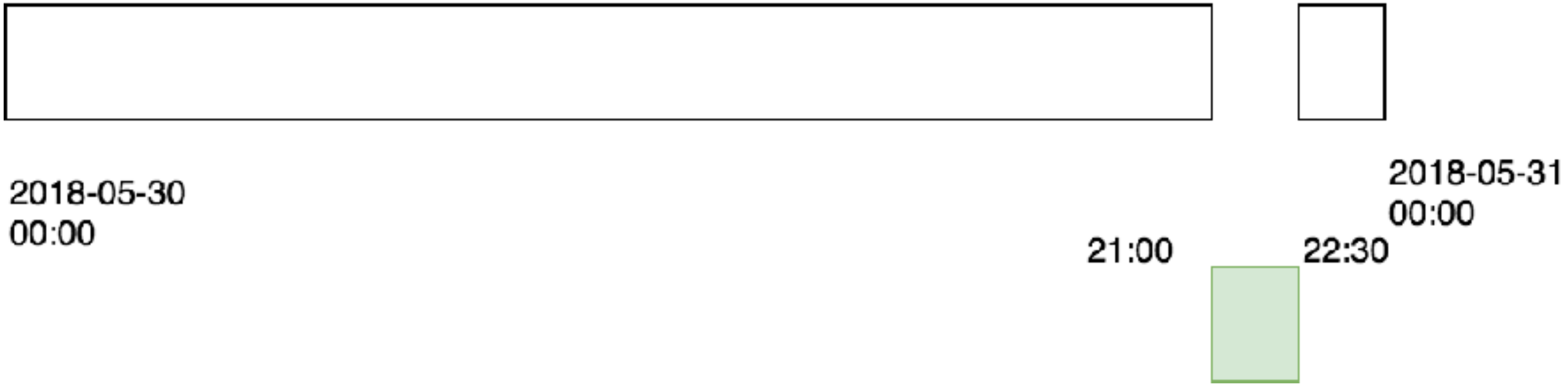


2018-05-30
00:00

08:00

20:00

2018-05-31
00:00



2018-05-30
00:00

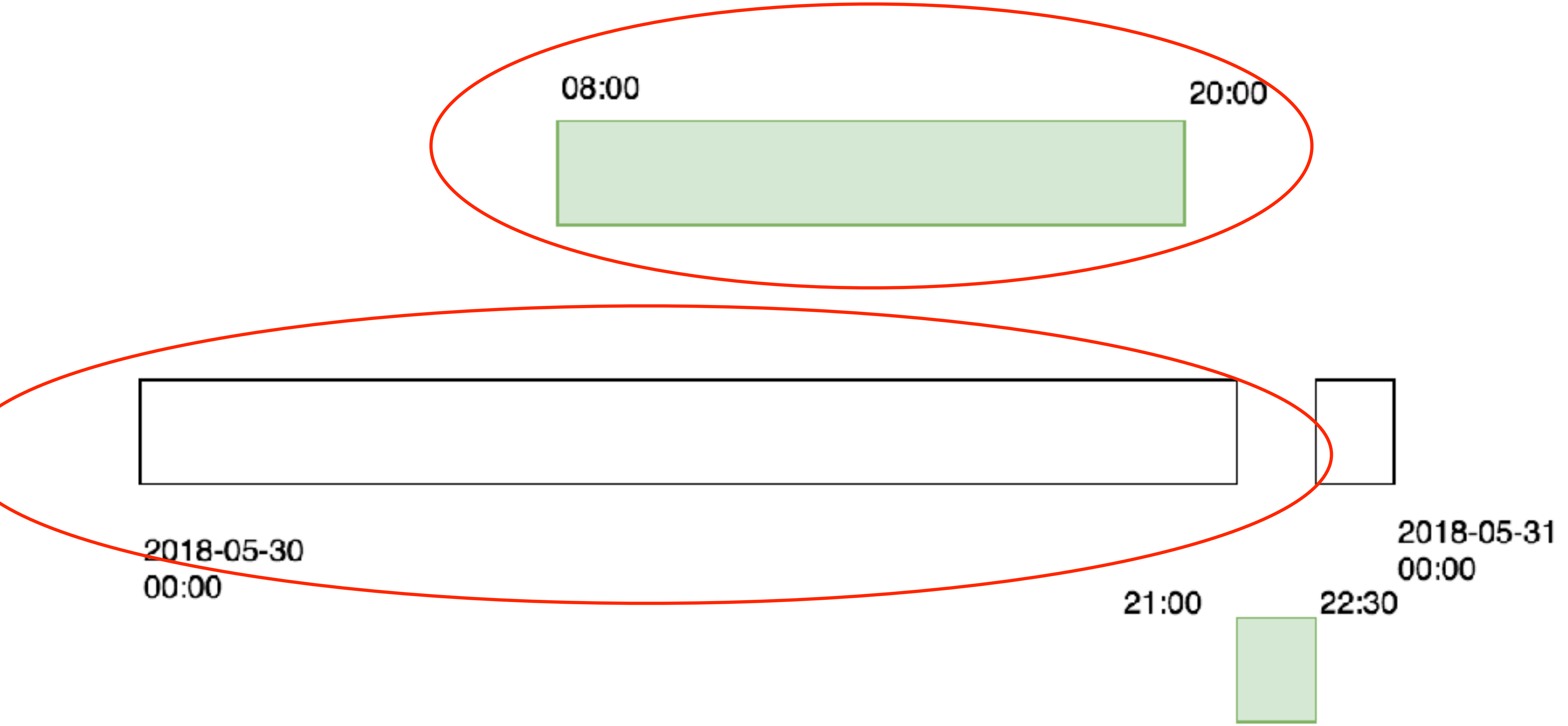
21:00

22:30

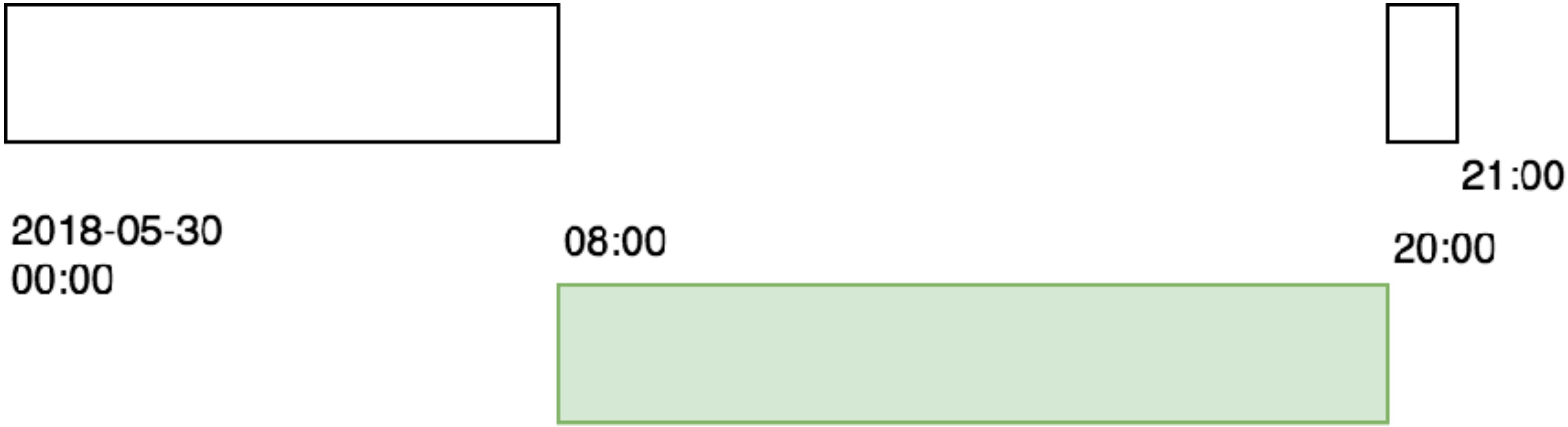
2018-05-31
00:00

```
    UNION
    SELECT
        'closed' AS left_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                tstzrange(
                    lower(availables.left_range),
                    lower(availables.left_range * availability.available_range)
                )
            ELSE
                tstzrange(
                    lower(availables.right_range),
                    lower(availables.right_range * availability.available_range)
                )
        END AS left_range,
        CASE
            WHEN
                availability.available_range && availables.left_range OR
                availability.available_range && availables.right_range
            THEN 'available'
            ELSE 'closed'
        END AS center_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                availability.available_range * availables.left_range
            ELSE
                availability.available_range * availables.right_range
        END AS center_range,
        'closed' AS right_status,
        CASE
            WHEN availability.available_range && availables.left_range THEN
                tstzrange(
                    upper(availables.left_range * availability.available_range),
                    upper(availables.left_range)
                )
            ELSE
                tstzrange(
                    upper(availables.right_range * availability.available_range),
                    upper(availables.right_range)
                )
        END AS right_range
    FROM availables
    JOIN availability ON
        availability.room_id = $1 AND
        availability.available_range && $2 AND
        availability.available_range <> availables.center_range AND (
            availability.available_range && availables.left_range OR
            availability.available_range && availables.right_range
        )
)
```

```
UNION
SELECT
    ...
FROM availables
JOIN availability ON
    availability.room_id = $1 AND
    availability.available_range && $2 AND
    availability.available_range <> availables.center_range AND (
        availability.available_range && availables.left_range OR
        availability.available_range && availables.right_range
    )
```


08:00
20:00
2018-05-30
00:00
2018-05-31
00:00
21:00
22:30

```
'closed' AS left_status,
CASE
    WHEN availability.available_range && availables.left_range THEN
        tstzrange(
            lower(availables.left_range),
            lower(availables.left_range * availability.available_range)
        )
    ELSE
        tstzrange(
            lower(availables.right_range),
            lower(availables.right_range * availability.available_range)
        )
END AS left_range,
CASE
    WHEN
        availability.available_range && availables.left_range OR
        availability.available_range && availables.right_range
    THEN 'available'
    ELSE 'closed'
END AS center_status,
CASE
    WHEN availability.available_range && availables.left_range THEN
        availability.available_range * availables.left_range
    ELSE
        availability.available_range * availables.right_range
END AS center_range,
'closed' AS right_status,
CASE
    WHEN availability.available_range && availables.left_range THEN
        tstzrange(
            upper(availables.left_range * availability.available_range),
            upper(availables.left_range)
        )
    ELSE
        tstzrange(
            upper(availables.right_range * availability.available_range),
            upper(availables.right_range)
        )
END AS right_range
```

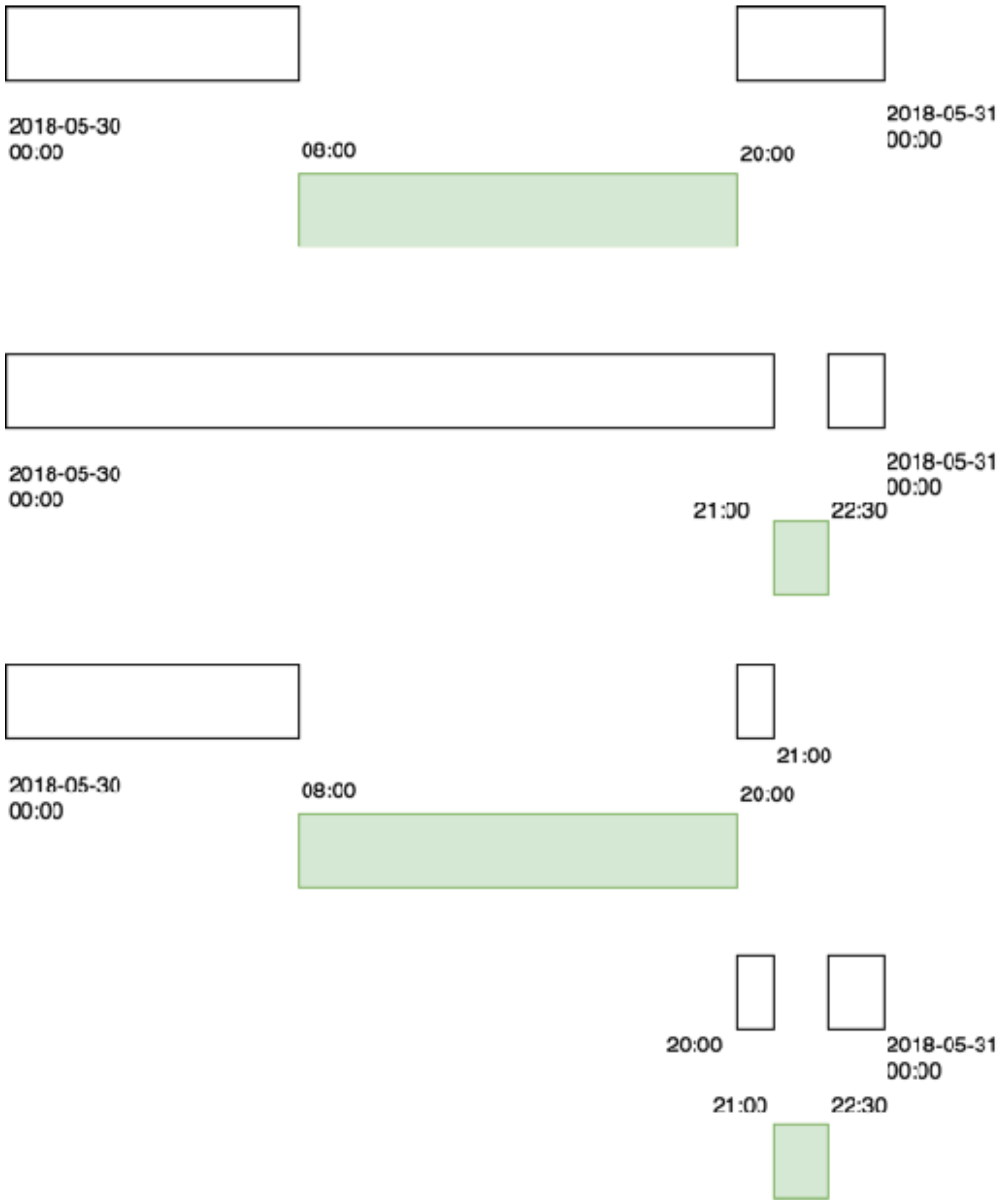


21:00

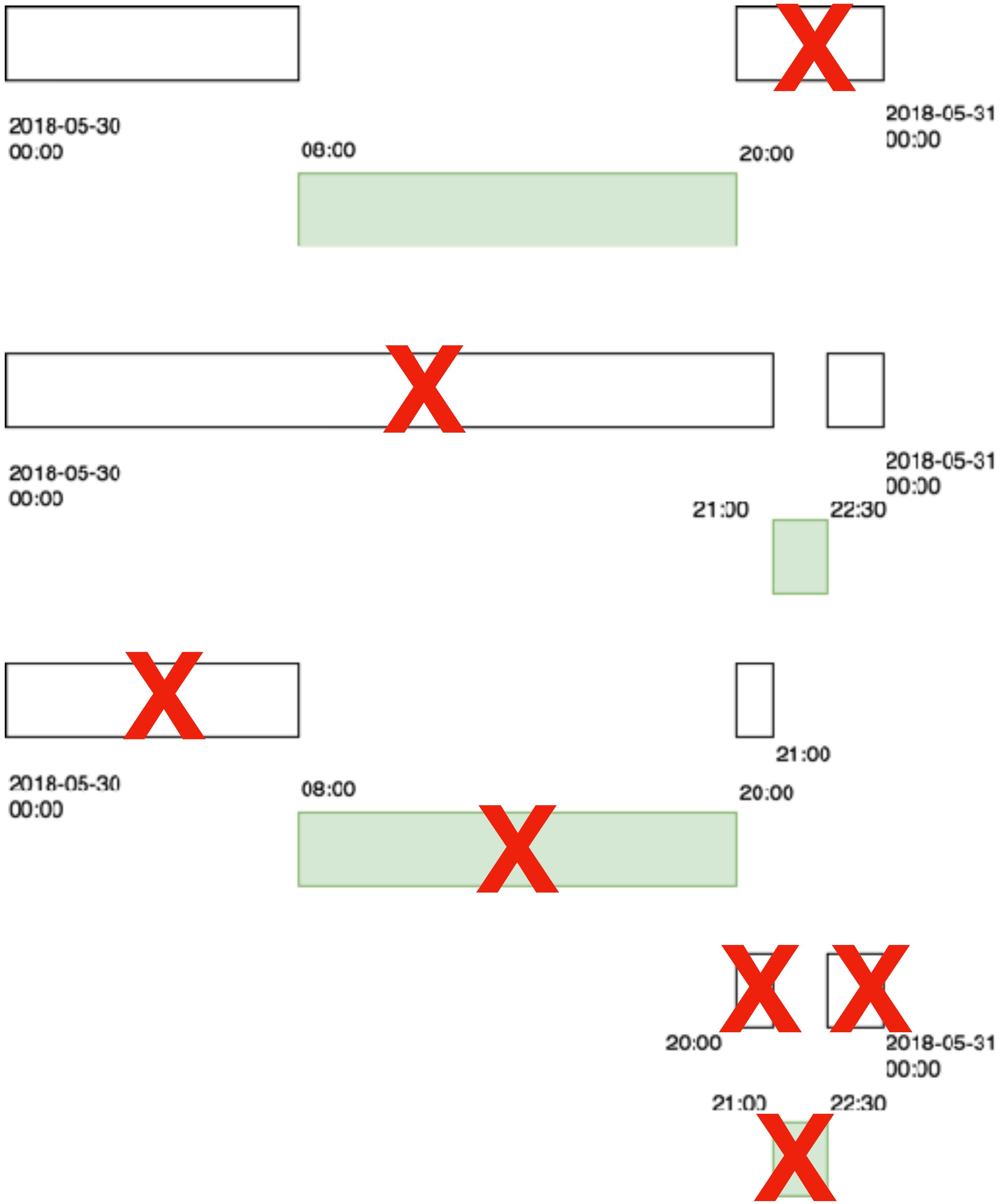
2018-05-30
00:00

08:00

20:00


2018-05-30
00:00
08:00
20:00
2018-05-31
00:00
2018-05-30
00:00
2018-05-31
00:00
21:00
22:30
21:00
2018-05-30
00:00
08:00
20:00
20:00
2018-05-31
00:00
21:00
22:30

```
SELECT *
FROM (
    SELECT
        x.left_status AS status,
        x.left_range AS calendar_range
    FROM availables x
    LEFT OUTER JOIN availables y ON
        x.left_range <> y.left_range AND
        x.left_range @> y.left_range
    GROUP BY 1, 2
    HAVING NOT bool_or(COALESCE(x.left_range @> y.left_range, FALSE))
    UNION
    SELECT DISTINCT
        x.center_status AS status,
        x.center_range AS calendar_range
    FROM availables x
    UNION
    SELECT
        x.right_status AS status,
        x.right_range AS calendar_range
    FROM availables x
    LEFT OUTER JOIN availables y ON
        x.right_range <> y.right_range AND
        x.right_range @> y.right_range
    GROUP BY 1, 2
    HAVING NOT bool_or(COALESCE(x.right_range @> y.right_range, FALSE))
) x
WHERE
    NOT isempty(x.calendar_range) AND
    NOT lower_inf(x.calendar_range) AND
    NOT upper_inf(x.calendar_range) AND
    x.calendar_range <@ $2
$$ LANGUAGE SQL STABLE;
```


2018-05-30
00:00
08:00
20:00
2018-05-31
00:00
2018-05-30
00:00
21:00
22:30
2018-05-31
00:00
2018-05-30
00:00
08:00
20:00
21:00
20:00
2018-05-31
00:00
21:00
22:30


22:30
2018-05-30
00:00
08:00
20:00
21:00
2018-05-31
00:00

Great!
What about unavailability?

```
/**
 * Helper function: combine the closed and available chunks of time with the unavailable chunks
 * of time to output the final calendar for the given `calendar_range`
 */
CREATE OR REPLACE FUNCTION calendar_generate_calendar(room_id int, calendar_range tstzrange)
RETURNS TABLE (status text, calendar_range tstzrange)
AS $$
    WITH RECURSIVE calendars AS (
        SELECT
            calendar.status AS left_status,
            tstzrange(
                lower(calendar.calendar_range),
                lower(unavailability.unavailable_range * calendar.calendar_range)
            ) AS left_range,
            CASE
                unavailability.unavailable_range IS NULL OR
                isempty(calendar.calendar_range * unavailability.unavailable_range)
            WHEN TRUE THEN calendar.status
            ELSE 'unavailable'
            END AS center_status,
            CASE
                unavailability.unavailable_range IS NULL OR
                isempty(calendar.calendar_range * unavailability.unavailable_range)
                WHEN TRUE THEN calendar.calendar_range
                ELSE unavailability.unavailable_range * calendar.calendar_range
            END AS center_range,
            calendar.status AS right_status,
            tstzrange(
                upper(unavailability.unavailable_range * calendar.calendar_range),
                upper(calendar.calendar_range)
            ) AS right_range
        FROM calendar_generate_available($1, $2) calendar
        LEFT OUTER JOIN unavailability ON
            unavailability.room_id = $1 AND
            unavailability.unavailable_range && $2
        UNION ALL
        SELECT DISTINCT
            calendars.left_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    tstzrange(
                        lower(calendars.left_range),
                        lower(calendars.left_range * unavailability.unavailable_range)
                    )
                ELSE
                    tstzrange(
                        lower(calendars.right_range),
                        lower(calendars.right_range * unavailability.unavailable_range)
                    )
            END AS left_range,
            CASE
                WHEN
                    unavailability.unavailable_range && calendars.left_range OR
                    unavailability.unavailable_range && calendars.right_range
                THEN 'unavailable'
                ELSE calendars.center_status
            END AS center_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    unavailability.unavailable_range * calendars.left_range
                ELSE
                    unavailability.unavailable_range * calendars.right_range
            END AS center_range,
            calendars.right_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    tstzrange(
                        upper(calendars.left_range * unavailability.unavailable_range),
                        upper(calendars.left_range)
                    )
                ELSE
                    tstzrange(
                        upper(calendars.right_range * unavailability.unavailable_range),
                        upper(calendars.right_range)
                    )
            END AS right_range
        FROM calendars
        JOIN unavailability ON
            unavailability.room_id = $1 AND
            unavailability.unavailable_range && $2 AND
            unavailability.unavailable_range <> calendars.center_range AND (
                unavailability.unavailable_range && calendars.left_range OR
                unavailability.unavailable_range && calendars.right_range
            )
    )
    SELECT *
    FROM (
        SELECT
            x.left_status AS status,
            x.left_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.left_range <> y.left_range AND
            x.left_range @> y.left_range AND
            y.left_status <> 'unavailable'
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.left_range @> y.left_range, FALSE))
        UNION
        SELECT
            x.center_status AS status,
            x.center_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.center_range <> y.center_range AND
            x.center_range @> y.center_range
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.center_range @> y.center_range, FALSE))
        UNION
        SELECT
            x.right_status AS status,
            x.right_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.right_range <> y.right_range AND
            x.right_range @> y.right_range AND
            y.right_status <> 'unavailable'
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.right_range @> y.right_range, FALSE))
    ) x
    WHERE
        NOT isempty(x.calendar_range) AND
        NOT lower_inf(x.calendar_range) AND
        NOT upper_inf(x.calendar_range)
$$ LANGUAGE SQL STABLE;
```

Remember when I said that it was the first of two helper functions?

```
/**
 * Helper function: combine the closed and available chunks of time with the unavailable chunks
 * of time to output the final calendar for the given `calendar_range`
 */
CREATE OR REPLACE FUNCTION calendar_generate_calendar(room_id int, calendar_range tstzrange)
RETURNS TABLE (status text, calendar_range tstzrange)
AS $$
    WITH RECURSIVE calendars AS (
        SELECT
            calendar.status AS left_status,
            tstzrange(
                lower(calendar.calendar_range),
                lower(unavailability.unavailable_range * calendar.calendar_range)
            ) AS left_range,
            CASE
                unavailability.unavailable_range IS NULL OR
                isempty(calendar.calendar_range * unavailability.unavailable_range)
            WHEN TRUE THEN calendar.status
            ELSE 'unavailable'
            END AS center_status,
            CASE
                unavailability.unavailable_range IS NULL OR
                isempty(calendar.calendar_range * unavailability.unavailable_range)
                WHEN TRUE THEN calendar.calendar_range
                ELSE unavailability.unavailable_range * calendar.calendar_range
            END AS center_range,
            calendar.status AS right_status,
            tstzrange(
                upper(unavailability.unavailable_range * calendar.calendar_range),
                upper(calendar.calendar_range)
            ) AS right_range
        FROM calendar_generate_available($1, $2) calendar
        LEFT OUTER JOIN unavailability ON
            unavailability.room_id = $1 AND
            unavailability.unavailable_range && $2
        UNION ALL
        SELECT DISTINCT
            calendars.left_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    tstzrange(
                        lower(calendars.left_range),
                        lower(calendars.left_range * unavailability.unavailable_range)
                    )
                ELSE
                    tstzrange(
                        lower(calendars.right_range),
                        lower(calendars.right_range * unavailability.unavailable_range)
                    )
            END AS left_range,
            CASE
                WHEN
                    unavailability.unavailable_range && calendars.left_range OR
                    unavailability.unavailable_range && calendars.right_range
                THEN 'unavailable'
                ELSE calendars.center_status
            END AS center_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    unavailability.unavailable_range * calendars.left_range
                ELSE
                    unavailability.unavailable_range * calendars.right_range
            END AS center_range,
            calendars.right_status,
            CASE
                WHEN unavailability.unavailable_range && calendars.left_range THEN
                    tstzrange(
                        upper(calendars.left_range * unavailability.unavailable_range),
                        upper(calendars.left_range)
                    )
                ELSE
                    tstzrange(
                        upper(calendars.right_range * unavailability.unavailable_range),
                        upper(calendars.right_range)
                    )
            END AS right_range
        FROM calendars
        JOIN unavailability ON
            unavailability.room_id = $1 AND
            unavailability.unavailable_range && $2 AND
            unavailability.unavailable_range <> calendars.center_range AND (
                unavailability.unavailable_range && calendars.left_range OR
                unavailability.unavailable_range && calendars.right_range
            )
    )
    SELECT *
    FROM (
        SELECT
            x.left_status AS status,
            x.left_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.left_range <> y.left_range AND
            x.left_range @> y.left_range AND
            y.left_status <> 'unavailable'
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.left_range @> y.left_range, FALSE))
        UNION
        SELECT
            x.center_status AS status,
            x.center_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.center_range <> y.center_range AND
            x.center_range @> y.center_range
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.center_range @> y.center_range, FALSE))
        UNION
        SELECT
            x.right_status AS status,
            x.right_range AS calendar_range
        FROM calendars x
        LEFT OUTER JOIN calendars y ON
            x.right_range <> y.right_range AND
            x.right_range @> y.right_range AND
            y.right_status <> 'unavailable'
        GROUP BY 1, 2
        HAVING NOT bool_or(COALESCE(x.right_range @> y.right_range, FALSE))
    ) x
    WHERE
        NOT isempty(x.calendar_range) AND
        NOT lower_inf(x.calendar_range) AND
        NOT upper_inf(x.calendar_range)
$$ LANGUAGE SQL STABLE;
```

Good news: principle is the same, so we are going to move on!

```
/**
 * Helper function: substitute the data within the `calendar`; this can be used
 *  for all updates that occur on `availability` and `unavailability`
 */
CREATE OR REPLACE FUNCTION calendar_manage(room_id int, calendar_date date)
RETURNS void
AS $$
    WITH delete_calendar AS (
        DELETE FROM calendar
        WHERE
            room_id = $1 AND
            calendar_date = $2
    )
    INSERT INTO calendar (room_id, status, calendar_date, calendar_range)
    SELECT $1, c.status, $2, c.calendar_range
    FROM calendar_generate_calendar($1, tstzrange($2, $2 + 1)) c
$$ LANGUAGE SQL;
```

```
/** Now, the trigger functions for availability and unavailability  */
CREATE OR REPLACE FUNCTION availability_manage()
RETURNS trigger
AS $trigger$
    BEGIN
        IF TG_OP = 'DELETE' THEN
            PERFORM calendar_manage(OLD.room_id, OLD.available_date);
            RETURN OLD;
        END IF;
        PERFORM calendar_manage(NEW.room_id, NEW.available_date);
        RETURN NEW;
    END;
$trigger$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION unavailability_manage()
RETURNS trigger
AS $trigger$
    BEGIN
        IF TG_OP = 'DELETE' THEN
            PERFORM calendar_manage(OLD.room_id, OLD.unavailable_date);
            RETURN OLD;
        END IF;
        PERFORM calendar_manage(NEW.room_id, NEW.unavailable_date);
        RETURN NEW;
    END;
$trigger$
LANGUAGE plpgsql;
```

```
/** And the triggers, applied to everything */
CREATE TRIGGER availability_manage
AFTER INSERT OR UPDATE OR DELETE ON availability
FOR EACH ROW
EXECUTE PROCEDURE availability_manage();

CREATE TRIGGER unavailability_manage
AFTER INSERT OR UPDATE OR DELETE ON unavailability
FOR EACH ROW
EXECUTE PROCEDURE unavailability_manage();
```

And we're done!

But we should probably test the whole system

Find out after the break!

The Test

Lessons or The Test

- [Test your live demos before running them, and you will have much success!]
- availability_rule inserts took some time, > 500ms
    - availability: INSERT 52
    - calendar: INSERT 52 from nontrivial function
- Updates on individual availability / unavailability are not too painful
- Lookups are faaaaaaaast

How about at (web) scale?

Web Scale :(

- Even with only 100 more rooms with a few set of rules, <u>rule generation time increased significantly</u>
- Lookups are still lightning fast!

Logical Decoding

- Added in PostgreSQL 9.4
- Replays all logical changes made to the database
  - Create a logical replication slot in your database
  - Only one receiver can consume changes from one slot at a time
  - Slot keeps track of last change that was read by a receiver
    - If receiver disconnects, slot will ensure database holds changes until receiver reconnects
  - ***Only changes from tables with primary keys are relayed***
- Basis for Logical Replication

Logical Decoding Out of the Box

- A logical replication slot has a name and a decoder
  - PostgreSQL comes with the "test" decoder
  - Have to write a custom parser to read changes from test decoder

Decoder Examples

- wal2json: https://github.com/eulerto/wal2json
- jsoncdc: https://github.com/posix4e/jsoncdc
- Debezium: http://debezium.io/

Driver Support

- C: libpq
  - pg_recvlogical
- PostgreSQL functions
- Python: psycopg2 - version 2.7
- JDBC: version 42

Using Logical Decoding

postgresql.conf

```
wal_level = logical
max_wal_senders = 2
max_replication_slots = 2
```

pg_hba.conf

```
local    replication      jkatz      trust      # DEVELOPMENT ONLY
```

In the database:

```
SELECT * FROM pg_create_logical_replication_slot('schedule', 'wal2json');
```

Testing Logical Decoding

```
import json
import sys

import psycopg2
import psycopg2.extras

class StreamReader(object):
    def __init__(self):
        self.connection = psycopg2.connect("dbname=realtime",
            connection_factory=psycopg2.extras.LogicalReplicationConnection,
        )

    def __call__(self, msg):
        data = json.loads(msg.payload, strict=False)
        print(str(data))
        msg.cursor.send_feedback(flush_lsn=msg.data_start)

reader = StreamReader()
cursor = reader.connection.cursor()
cursor.start_replication(slot_name='schedule', decode=True)
try:
    cursor.consume_stream(reader)
except KeyboardInterrupt:
    print("Stopping reader...")
finally:
    cursor.close()
    reader.connection.close()
    print("Exiting reader")
```

Testing Logical Decoding

```
/** Create the room */
INSERT INTO room (name) VALUES ('DMS 1150');

/** Look at initial calendar on May 30, 2018 */
SELECT *
FROM calendar
WHERE calendar_date = '2018-05-30'
ORDER BY lower(calendar_range);

/** DMS 1150 only allows bookings from 8am - 1pm, and 4pm to 10pm on Mon - Fri */
INSERT INTO availability_rule
    (room_id, days_of_week, start_time, end_time, generate_weeks_into_future)
SELECT
    room.id, ARRAY[1,2,3,4,5]::int[], times.start_time::time, times.end_time::time, 52
FROM room,
    LATERAL (
        VALUES ('8:00', '13:00'), ('16:00', '22:00')
    ) times(start_time, end_time)
WHERE room.name = 'DMS 1150';

INSERT INTO unavailability (room_id, unavailable_date, unavailable_range)
SELECT
    room.id, '2018-05-30', tstzrange('2018-05-02 9:00', '2018-05-02 11:00')
FROM room
WHERE room.name = 'DMS 1150';
```

Testing Logical Decoding

- Every change in the database is streamed
- Need to be aware of the logical decoding forma

Thoughts

- We know it takes time to regenerate calendar
- Want to ensure changes always propagate but want to ensure all users (managers, calendar searchers) have good experience

Replacing the Triggers

- Will use the same data model as before as well as the same helper functions, but **without** the triggers
    - (That's a lie, we will have one set of DELETE triggers as "DELETE" in the decoder does not provide enough information)

Replacing the Triggers

```
/**
 * Helper function: substitute the data within the `calendar`; this can be used
 *  for all updates that occur on `availability` and `unavailability`
 */
CREATE OR REPLACE FUNCTION calendar_manage(room_id int, calendar_date date)
RETURNS void
AS $$
    WITH delete_calendar AS (
        DELETE FROM calendar
        WHERE
            room_id = $1 AND
            calendar_date = $2
    )
    INSERT INTO calendar (room_id, status, calendar_date, calendar_range)
    SELECT $1, c.status, $2, c.calendar_range
    FROM calendar_generate_calendar($1, tstzrange($2, $2 + 1)) c
$$ LANGUAGE SQL;
```

```
/** Now, the trigger functions for availability and unavailability; needs this for DELETE  *
CREATE OR REPLACE FUNCTION availability_manage()
RETURNS trigger
AS $trigger$
    BEGIN
        IF TG_OP = 'DELETE' THEN
            PERFORM calendar_manage(OLD.room_id, OLD.available_date);
            RETURN OLD;
        END IF;
    END;
$trigger$
LANGUAGE plpgsql;

CREATE OR REPLACE FUNCTION unavailability_manage()
RETURNS trigger
AS $trigger$
    BEGIN
        IF TG_OP = 'DELETE' THEN
            PERFORM calendar_manage(OLD.room_id, OLD.unavailable_date);
            RETURN OLD;
        END IF;
    END;
$trigger$
LANGUAGE plpgsql;


/** And the triggers, applied to everything */
CREATE TRIGGER availability_manage
AFTER DELETE ON availability
FOR EACH ROW
EXECUTE PROCEDURE availability_manage();

CREATE TRIGGER unavailability_manage
AFTER DELETE ON unavailability
FOR EACH ROW
EXECUTE PROCEDURE unavailability_manage();
```

Replacing the Triggers

- We will have a Python script that reads from a logical replication slot and if it detects a relevant change, take an action
- *Similar* to what we did with triggers, but this moves the work to OUTSIDE the transaction
- BUT...we can confirm whether or not the work is completed, thus if the program fails, we can restart from last acknowledged transaction ID

Reading the Changes

```
import json
import sys

import psycopg2
import psycopg2.extras

SQL = {
    'availability': {
        'insert': """SELECT calendar_manage(%(room_id)s, %(available_date)s)""",
        'update': """SELECT calendar_manage(%(room_id)s, %(available_date)s)""",
    },
    'availability_rule': {
        'insert': True,
        'update': True,
    },
    'room': {
        'insert': """
            INSERT INTO calendar (room_id, status, calendar_date, calendar_range)
            SELECT
                %(id)s, 'closed', calendar_date, tstzrange(calendar_date, calendar_date + '1
day'::interval)
            FROM generate_series(
                date_trunc('week', CURRENT_DATE),
                date_trunc('week', CURRENT_DATE + '52 weeks'::interval),
                '1 day'::interval
            ) calendar_date;
        """,
    },
    'unavailability': {
        'insert': """SELECT calendar_manage(%(room_id)s, %(unavailable_date)s)""",
        'update': """SELECT calendar_manage(%(room_id)s, %(unavailable_date)s)""",
    },
}
```

Reading the Changes

```
class StreamReader(object):
    def _consume_change(self, payload):
        connection = psycopg2.connect("dbname=realtime")
        cursor = connection.cursor()
        for data in payload['change']:
            sql = SQL.get(data.get('table'), {}).get(data.get('kind'))
            if not sql:
                return
            params = dict(zip(data['columnnames'], data['columnvalues']))
            if data['table'] == 'availability_rule':
                self._perform_availability_rule(cursor, data['kind'], params)
            else:
                cursor.execute(sql, params)
        connection.commit()
        cursor.close()
        connection.close()

    def _perform_availability_rule(self, cursor, kind, params):
        if kind == 'update':
            cursor.execute("""DELETE FROM availability WHERE availability_rule_id = %(id)s""", params)
        if kind in ['insert', 'update']:
            days_of_week = params['days_of_week'].replace('{', '').replace('}', '').split(',')
            for day_of_week in days_of_week:
                params['day_of_week'] = day_of_week
                cursor.execute(
                    """
                    SELECT availability_rule_bulk_insert(ar, %(day_of_week)s)
                    FROM availability_rule ar
                    WHERE ar.id = %(id)s
                    """, params)
```

Reading the Changes

```
def __init__(self):
    self.connection = psycopg2.connect("dbname=schedule",
        connection_factory=psycopg2.extras.LogicalReplicationConnection,
    )

def __call__(self, msg):
    payload = json.loads(msg.payload, strict=False)
    print(payload)
    self._consume_change(payload)
    msg.cursor.send_feedback(flush_lsn=msg.data_start)
```

Reading the Changes

```
reader = StreamReader()
cursor = reader.connection.cursor()
cursor.start_replication(slot_name='schedule', decode=True)
try:
    cursor.consume_stream(reader)
except KeyboardInterrupt:
    print("Stopping reader...")
finally:
    cursor.close()
    reader.connection.close()
    print("Exiting reader")
```

Testing Our Application

Lessons

- Logical decoding allows the bulk inserts to occur significantly faster from a transactional view
- DELETEs are tricky if you need to do anything other than using the PRIMARY KEY
- Based on implementation, changes applied serially
  - Potential bottleneck for long running queries
  - Use a distributed streaming tool like Kafka to perform follow-up queries

Conclusion

- PostgreSQL is robust
- Triggers will keep your data in sync but can have significant performance overhead
- Utilizing a logical replication slot can eliminate trigger overhead and transfer the computational load elsewhere
- Not a panacea: still need to use good architectural patterns!

Questions?

jonathan.katz@crunchydata.com
@jkatz05