download pdf

- http://chak.org/matviews.pdf

But don’t read ahead yet! :-)

(oh, and use Acrobat, not Mac Preview,
well, check pg 16, got checkmarks? good)

important pages for now: 8 & 16

Materialized Views that Work

Dan Chak (dan@chak.org)

PGCon 2008

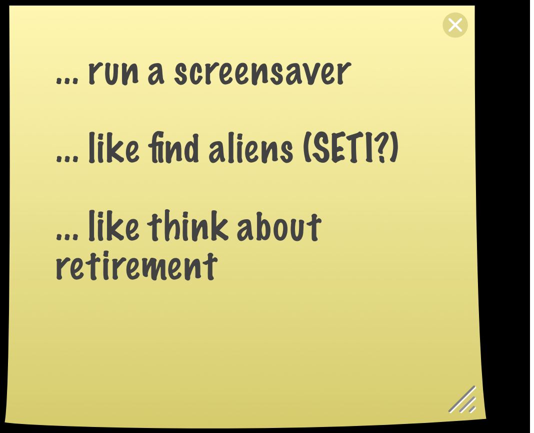
Materialized Views that Work Hard

(so you don't have to)

Dan Chak (dan@chak.org)
PGCon 2008

Materialized Views that Work Efficiently

(so your database can do other things)



... run a screensaver

... like find aliens (SETI?)

... like think about retirement

Dan Chak (dan@chak.org)

PGCon 2008

But let's talk about me

(aka, who is this guy, anyway?)

"You're probably wondering who I am and why you should be listening to me for 3 hours.

Don't worry, there will be coffee breaks."

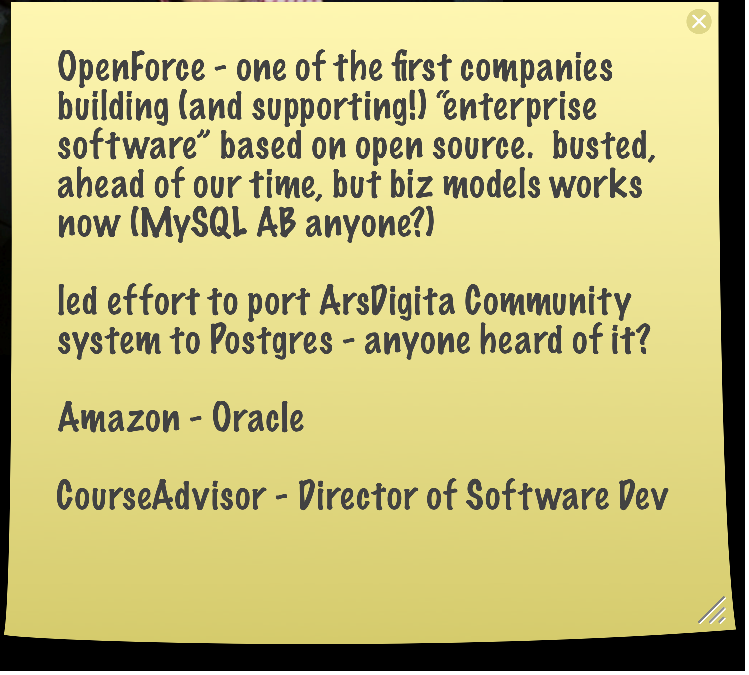
Dan Chak's

lightning fast résumé

CourseAdvisor (Boston) 2005-?
Amazon.com (Seattle) 2003-2005
OpenForce (NYC) 2000-2002

MIT Computer Science & Engineering Bachelors
MIT Human Computer Interfaces Masters

O'Reilly *Enterprise Rails*
due out in October!



OpenForce - one of the first companies building (and supporting!) "enterprise software" based on open source. busted, ahead of our time, but biz models works now (MySQL AB anyone?)

led effort to port ArsDigita Community system to Postgres - anyone heard of it?

Amazon - Oracle

CourseAdvisor - Director of Software Dev

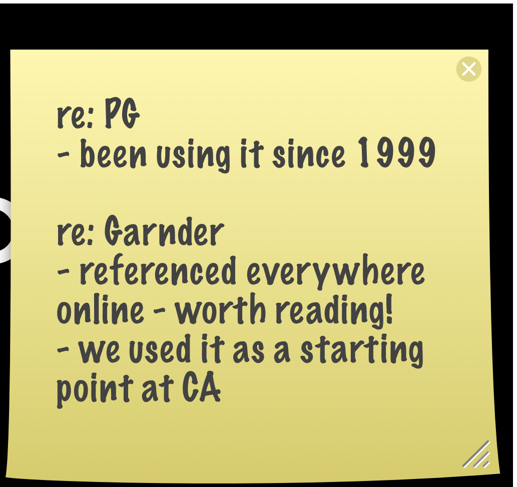
“The Orbitz of Education”

over 200k visits per day
5-6 million per month

~20%, or 1 million users, do an
“orbitz-style” search
for “what’s available for me?”

one Postgres database

Thank you’s to



re: PG
- been using it since 1999

re: Garnder
- referenced everywhere online - worth reading!
- we used it as a starting point at CA

- PG Team for creating a great database
- Jonathan Gardner - wrote the authoritative “Materialized Views in Postgres”
- Kristof Redei - CourseAdvisor intern who put these ideas into practice in our production application

what about you?

application developers

web?
dw?

pg developers?

expertise -
expert,
intermediate,
newbie?

materialized a view before?

Agenda

1. W's: What, why, when?
2. Some PG Basics
3. An end-to-end implementation
4. Getting Advanced (ie, even faster)
5. Repeatable Process

Part I:
What, Why, When?

1. Performance,
   Performance,
   Performance
2. Definitions
3. Applications
4. Expectation Setting

All About Performance

- $O(f(n))$ becomes $O(1)$
- Attack from all angles

makes queries slow:
joins,
function evaluation f(n)

data warehouse land:
- memoize reporting queries
history doesn't change, usually
- summary tables

attack from all angles means:

- vacuum
- query planning

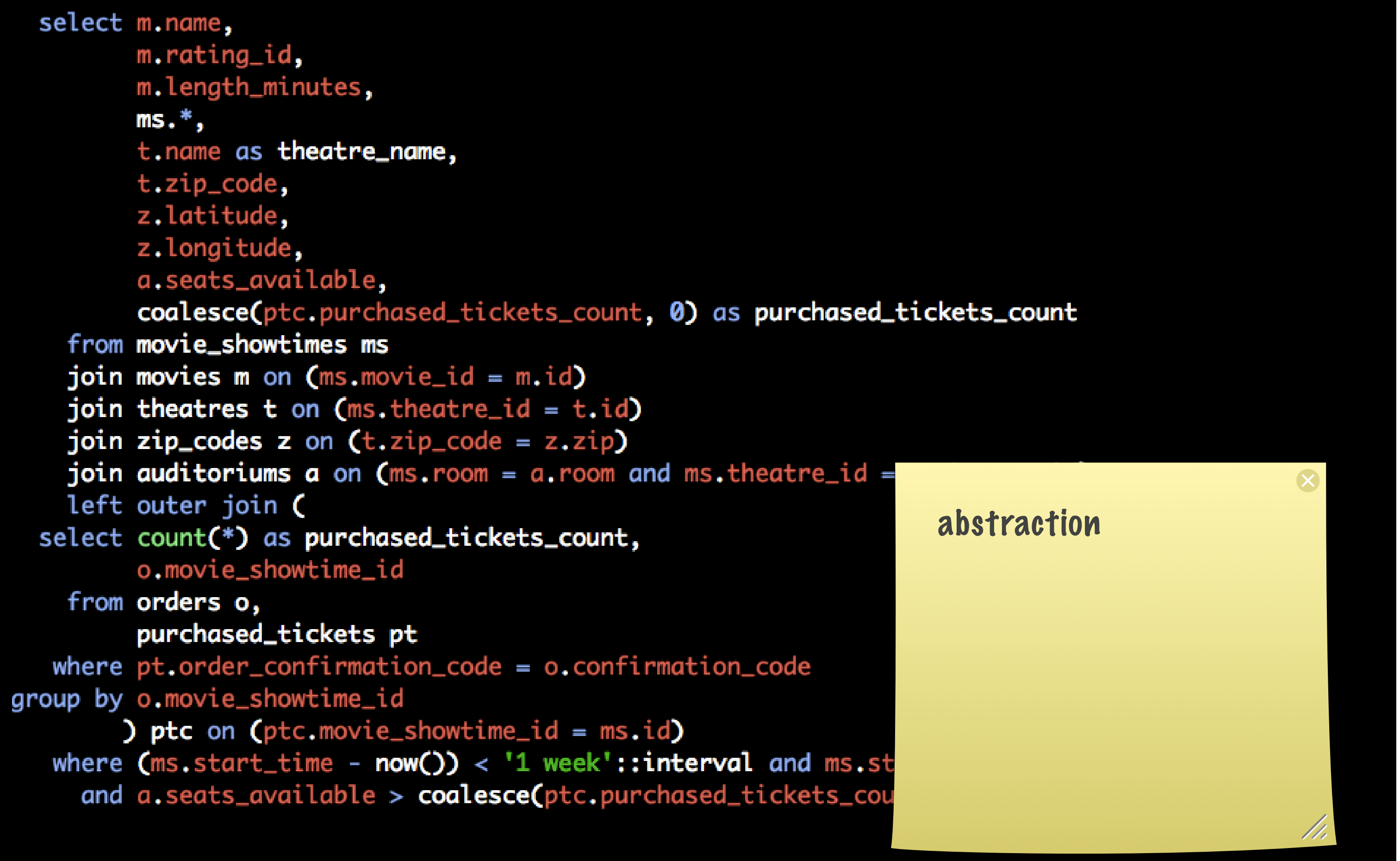
not just view materialization
but also:
- view optimization
- configuration tweaking

Definitions

a repetitive query

```
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where (ms.start_time - now()) < '1 week'::interval and ms.start_time > now()
     and a.seats_available > coalesce(ptc.purchased_tickets_count, 0);
```

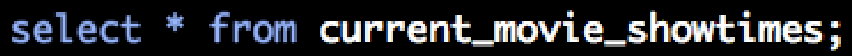
a view is a *named query*



```
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id =
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where (ms.start_time - now()) < '1 week'::interval and ms.st
     and a.seats_available > coalesce(ptc.purchased_tickets_cou
```

abstraction

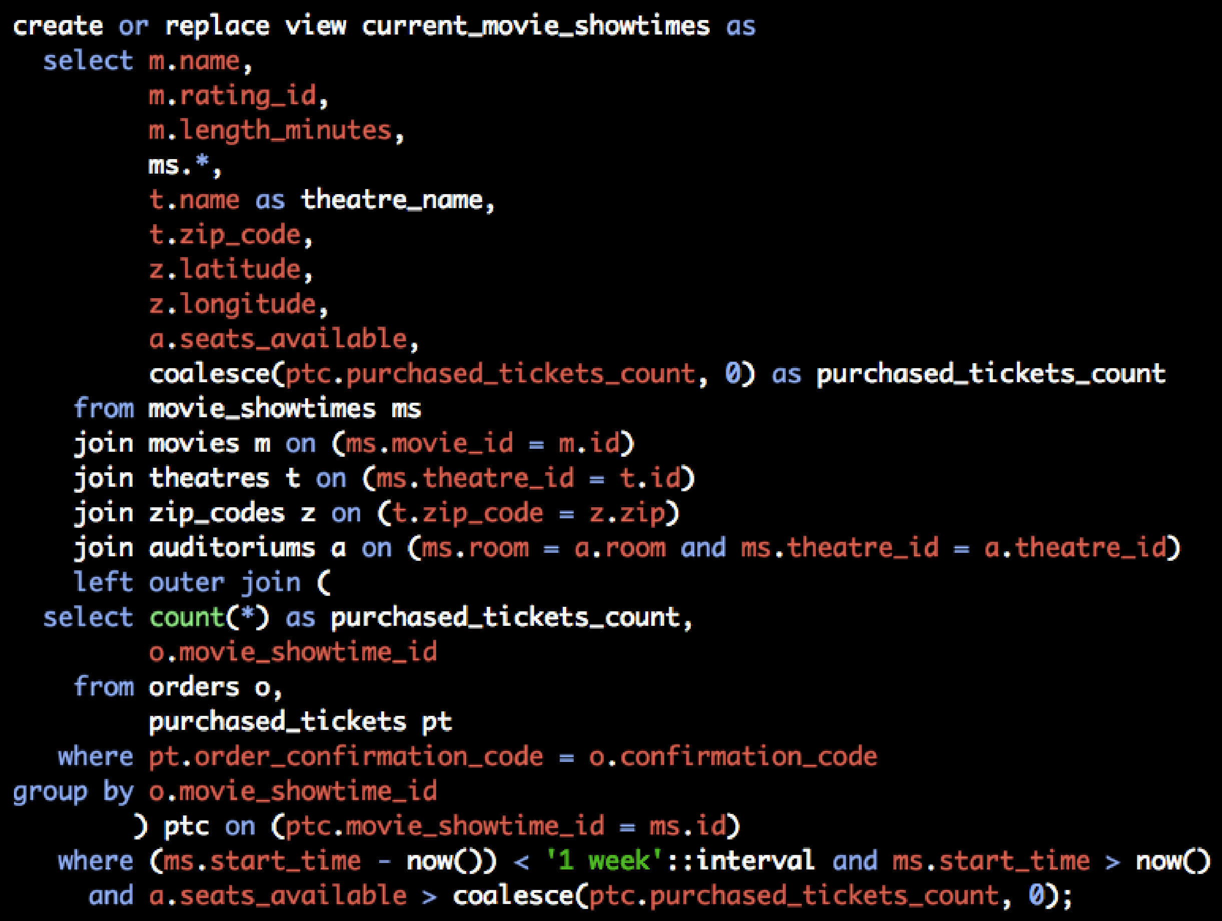
selecting from a view



```
select * from current_movie_showtimes;
```

equivalent to...

```
select * from (
```



```
  create or replace view current_movie_showtimes as
    select m.name,
           m.rating_id,
           m.length_minutes,
           ms.*,
           t.name as theatre_name,
           t.zip_code,
           z.latitude,
           z.longitude,
           a.seats_available,
           coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
      from movie_showtimes ms
      join movies m on (ms.movie_id = m.id)
      join theatres t on (ms.theatre_id = t.id)
      join zip_codes z on (t.zip_code = z.zip)
      join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
      left outer join (
    select count(*) as purchased_tickets_count,
           o.movie_showtime_id
      from orders o,
           purchased_tickets pt
     where pt.order_confirmation_code = o.confirmation_code
  group by o.movie_showtime_id
          ) ptc on (ptc.movie_showtime_id = ms.id)
     where (ms.start_time - now()) < '1 week'::interval and ms.start_time > now()
       and a.seats_available > coalesce(ptc.purchased_tickets_count, 0);
```

```
) current_movie_showtimes;
```

View Roundup

What’s good

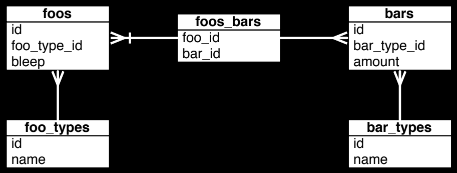
- Abstracts a query behind a view name
- Mentally efficient
- Reusable
- Less prone to error

What’s not so good

- Entire query is executed for each access
- Calculated columns re-calculated on each access
- Looks like a table, but slow like a query

Materialized Views?

(sounds like an oxymoron to me)



foos
id
foo_type_id
bleep

foos_bars
foo_id
bar_id

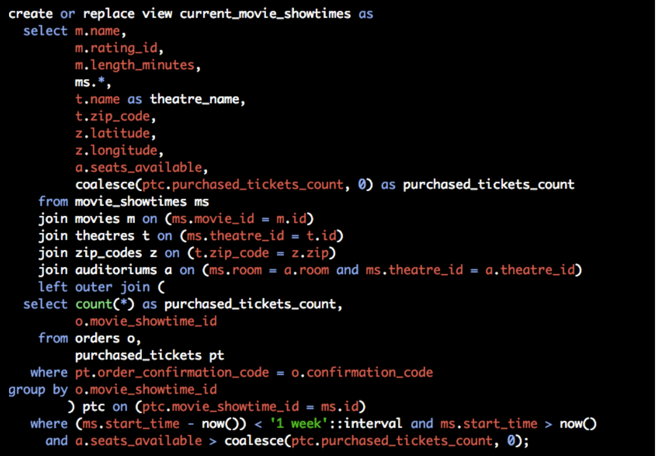
bars
id
bar_type_id
amount

foo_types
id
name

bar_types
id
name

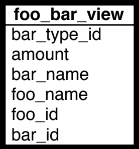
base tables
(physical)

views



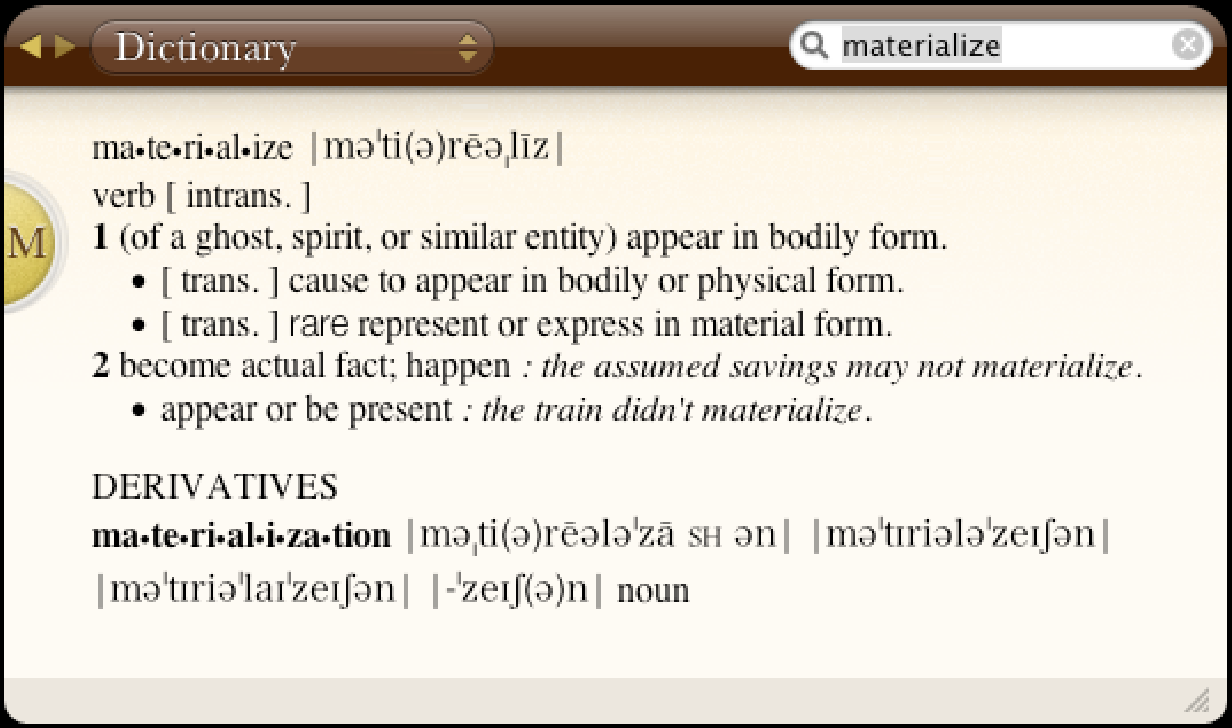
```
create or replace view current_movie_showtimes as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where (ms.start_time - now()) < '1 week'::interval and ms.start_time > now()
     and a.seats_available > coalesce(ptc.purchased_tickets_count, 0);
```

materialized view
(physical)



foo_bar_view
bar_type_id
amount
bar_name
foo_name
foo_id
bar_id

“materialize”



ma•te•ri•al•ize |məˈti(ə)rēəˌlīz|

verb [intrans.]

1 (of a ghost, spirit, or similar entity) appear in bodily form.

- [trans.] cause to appear in bodily or physical form.
- [trans.] rare represent or express in material form.

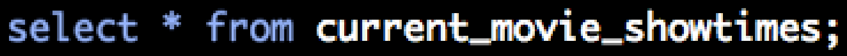
2 become actual fact; happen *: the assumed savings may not materialize.*

- appear or be present *: the train didn't materialize.*

DERIVATIVES

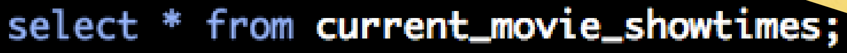
ma•te•ri•al•i•za•tion |məˌti(ə)rēələˈzā sh ən| |məˈtɪriələˈzeɪʃən| |məˈtɪriəˈlaɪˈzeɪʃən| |-ˈzeɪʃ(ə)n| noun

result:



```
select * from current_movie_showtimes;
```

really means...



```
select * from current_movie_showtimes;
```

physical table!

- No "subquery" to compute on each access
- A physical table can be indexed, partitioned, etc. to improve performance further

Types

- Snapshot
- Very Lazy
- Lazy
- Eager

Snapshot

- Creates a physical table as the result of selecting everything out of a view
- Refresh at a given interval
- Pro: Easy to set up
- Con: Gets out of sync quickly
- Con: Full refresh can be very expensive

Very Lazy

- Like snapshot, but only out of sync rows get updated at refresh time
- Requires keeping track of which rows are out of sync
- Pro: Lighter refresh than snapshot
- Con: Still gets out of sync quickly
- Con: Need an ancillary table to implement (or can use dirty column)

Lazy

- Start with a snapshot
- Refresh rows that are out of sync at the end of each transaction
- Pro: Always in sync*
- Pro: Only affected rows are updated
- Con: There’s no “after transaction” trigger in Postgres

* *mutable functions excluded*

Eager

common question: can you use statement level triggers to get around N:1 relationships?

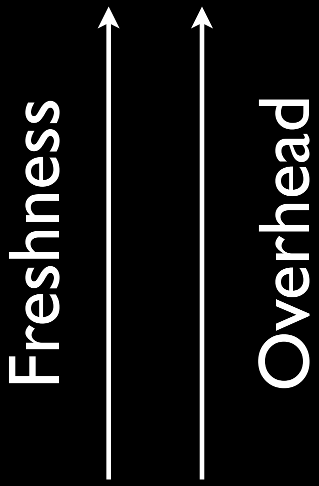
statement level doesn't tell you which rows are updated.

- Like Lazy, but update materialized each statement.
- Uses triggers after update, insert, and delete on all referenced tables
- Pro: Always in sync*
- Con: Bad in one-to-many relationships updates. Updating rows that feed into an aggregate cause N refreshes rather than 1.

* *mutable functions excluded*

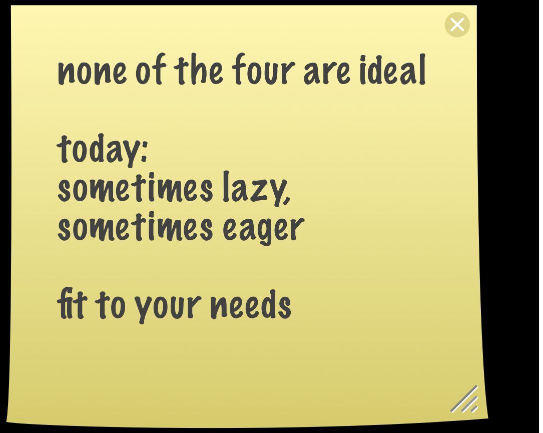
Refresh strategies

- Eager
- Lazy
- Very lazy
- Snapshot



Today’s Tutorial

- Snapshot
- Very Lazy
- Lazy
- Eager



none of the four are ideal

today:
sometimes lazy,
sometimes eager

fit to your needs

also:

- Solve mutable function problem for f(time)
- Mimic a post-transaction trigger

Applications

- High throughput web sites
- Data warehousing
- Reporting memoization

memoization can be tricky

will be discussed, but...

focus in talk will be on real-time production applications

Data Warehousing ETL

- Automatically build summary tables
- Automatically keep summary tables up to date
- Memoize results of recurring queries

High Performance Production Sites

- Reduces bottleneck $O(f(n))$ query to $O(1)$.

Expectation Setting

- Billions of dollars
- 6-pack abs
- 100-1000x performance increase typical

Really!

- *100-1000x performance increase typical*

Depends on how slow your query is to begin with.

Also depends on how heavily loaded your database is.

Compare

- executing arbitrarily complex query on a loaded database

- selecting a single row out of an indexed table

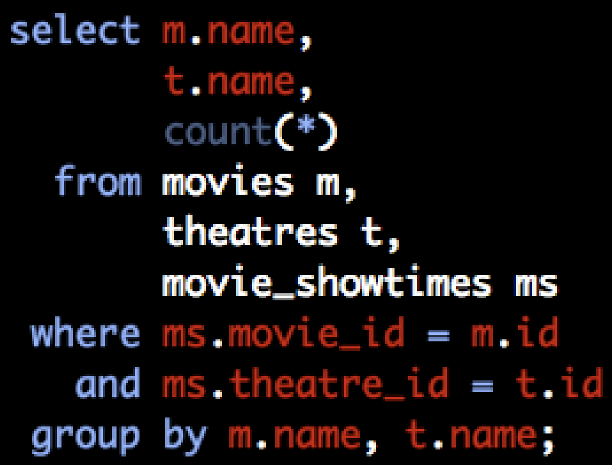
1, 2, 3... 5s? → 5ms

Part II: PG Basics

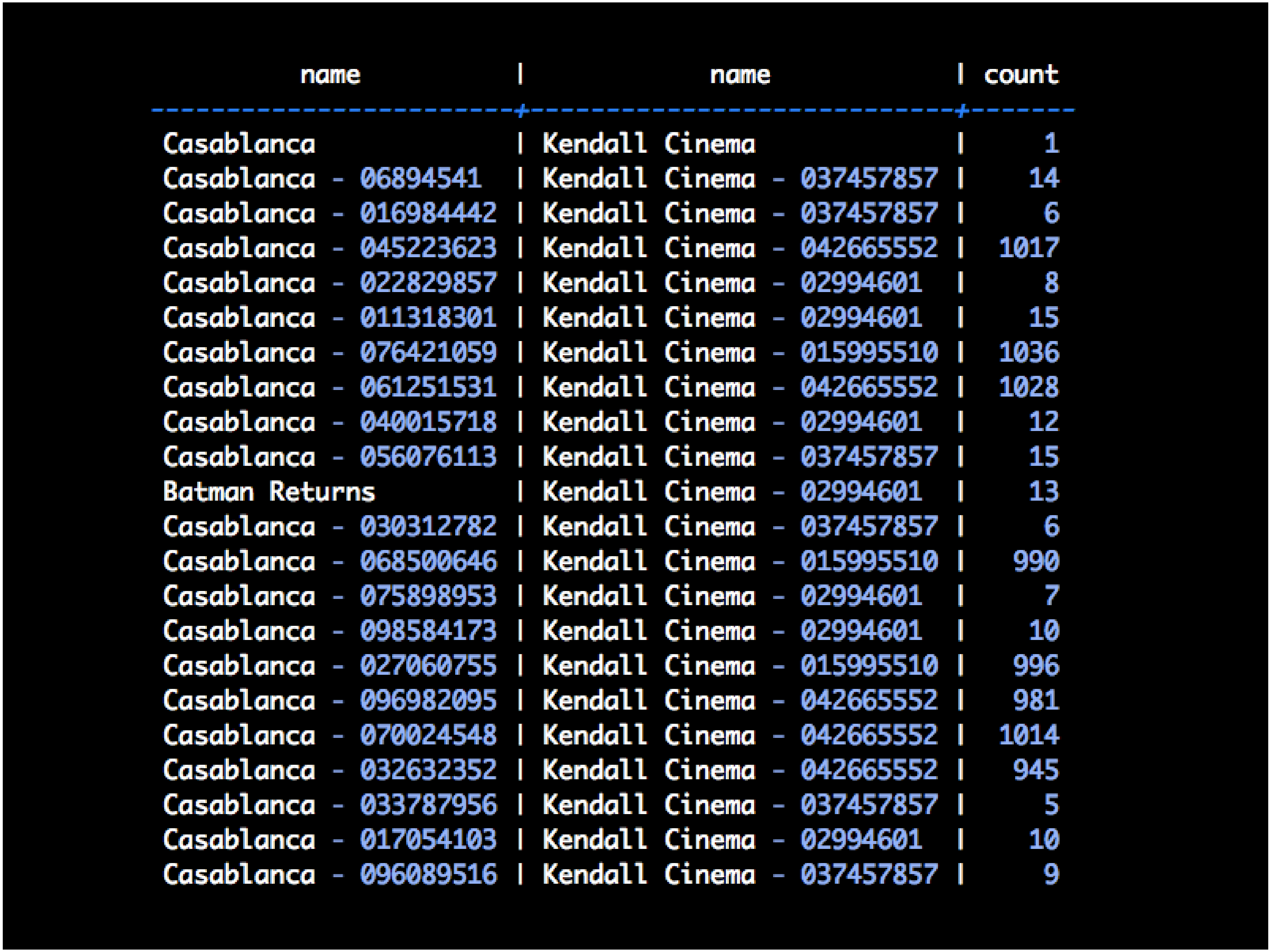
1. Query Planner
2. Stored procedures
3. Triggers

Query Planner

- explain
- explain analyze



```
select m.name,
       t.name,
       count(*)
  from movies m,
       theatres t,
       movie_showtimes ms
 where ms.movie_id = m.id
   and ms.theatre_id = t.id
 group by m.name, t.name;
```



```
          name          |            name            | count
------------------------+----------------------------+-------
 Casablanca             | Kendall Cinema             |     1
 Casablanca - 06894541  | Kendall Cinema - 037457857 |    14
 Casablanca - 016984442 | Kendall Cinema - 037457857 |     6
 Casablanca - 045223623 | Kendall Cinema - 042665552 |  1017
 Casablanca - 022829857 | Kendall Cinema - 02994601  |     8
 Casablanca - 011318301 | Kendall Cinema - 02994601  |    15
 Casablanca - 076421059 | Kendall Cinema - 015995510 |  1036
 Casablanca - 061251531 | Kendall Cinema - 042665552 |  1028
 Casablanca - 040015718 | Kendall Cinema - 02994601  |    12
 Casablanca - 056076113 | Kendall Cinema - 037457857 |    15
 Batman Returns         | Kendall Cinema - 02994601  |    13
 Casablanca - 030312782 | Kendall Cinema - 037457857 |     6
 Casablanca - 068500646 | Kendall Cinema - 015995510 |   990
 Casablanca - 075898953 | Kendall Cinema - 02994601  |     7
 Casablanca - 098584173 | Kendall Cinema - 02994601  |    10
 Casablanca - 027060755 | Kendall Cinema - 015995510 |   996
 Casablanca - 096982095 | Kendall Cinema - 042665552 |   981
 Casablanca - 070024548 | Kendall Cinema - 042665552 |  1014
 Casablanca - 032632352 | Kendall Cinema - 042665552 |   945
 Casablanca - 033787956 | Kendall Cinema - 037457857 |     5
 Casablanca - 017054103 | Kendall Cinema - 02994601  |    10
 Casablanca - 096089516 | Kendall Cinema - 037457857 |     9
```

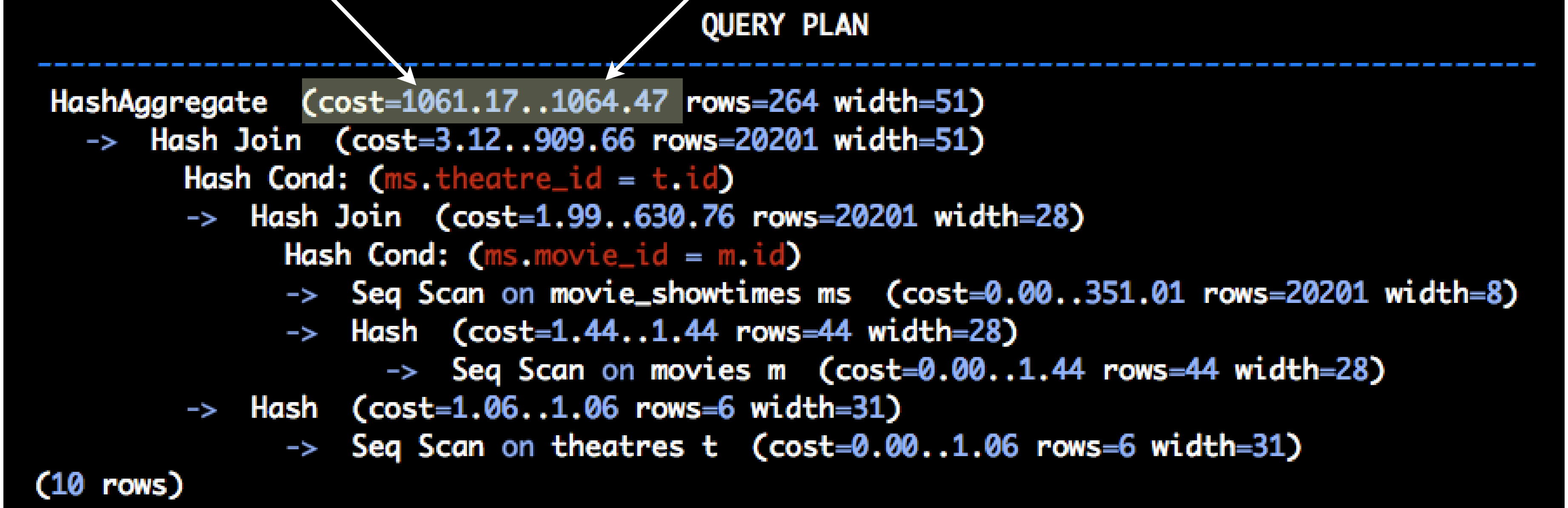
explain

- returns the query plan
- fast
- units are mythical

```
explain
select m.name,
       t.name,
       count(*)
  from movies m,
       theatres t,
       movie_showtimes ms
 where ms.movie_id = m.id
   and ms.theatre_id = t.id
 group by m.name, t.name;
```

Time to first result record

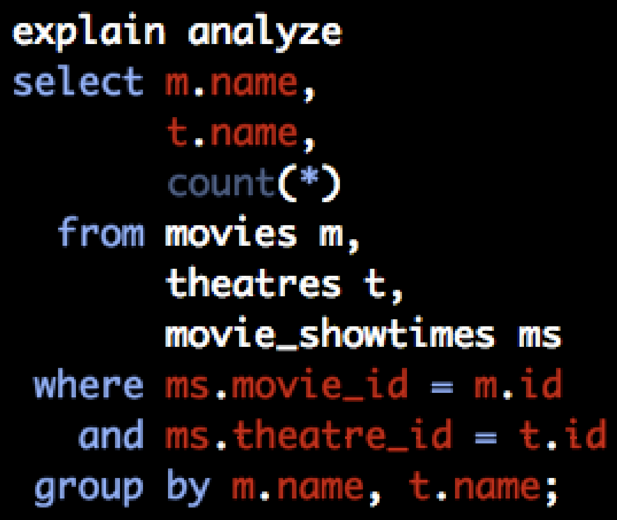
Time to last result record



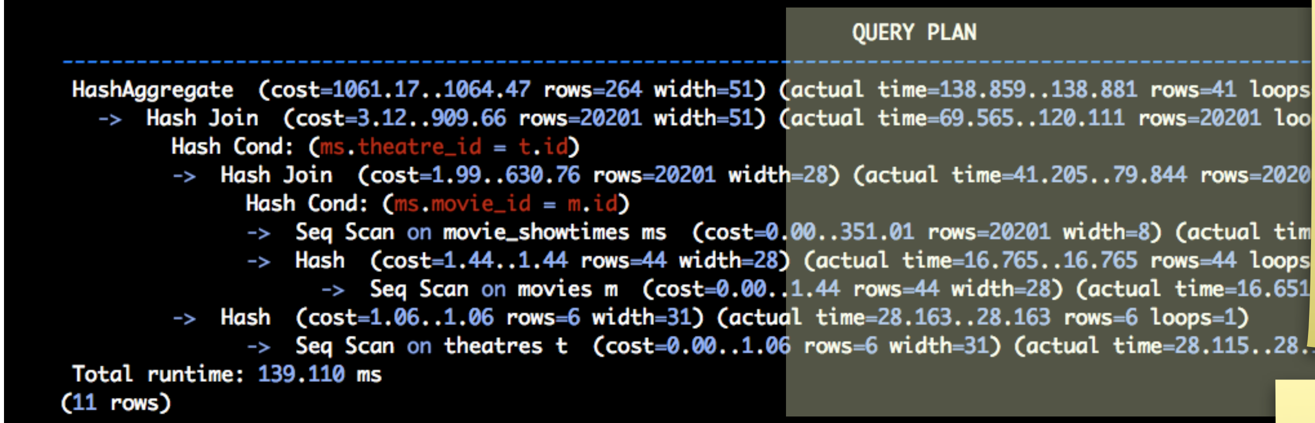
```
                                        QUERY PLAN
------------------------------------------------------------------------------------------
 HashAggregate  (cost=1061.17..1064.47 rows=264 width=51)
   ->  Hash Join  (cost=3.12..909.66 rows=20201 width=51)
         Hash Cond: (ms.theatre_id = t.id)
         ->  Hash Join  (cost=1.99..630.76 rows=20201 width=28)
               Hash Cond: (ms.movie_id = m.id)
               ->  Seq Scan on movie_showtimes ms  (cost=0.00..351.01 rows=20201 width=8)
               ->  Hash  (cost=1.44..1.44 rows=44 width=28)
                     ->  Seq Scan on movies m  (cost=0.00..1.44 rows=44 width=28)
         ->  Hash  (cost=1.06..1.06 rows=6 width=31)
               ->  Seq Scan on theatres t  (cost=0.00..1.06 rows=6 width=31)
(10 rows)
```

explain analyze

- explain “plus”
- actually runs the query (without commit)
- adds time in milliseconds



```
explain analyze
select m.name,
       t.name,
       count(*)
  from movies m,
       theatres t,
       movie_showtimes ms
 where ms.movie_id = m.id
   and ms.theatre_id = t.id
 group by m.name, t.name;
```



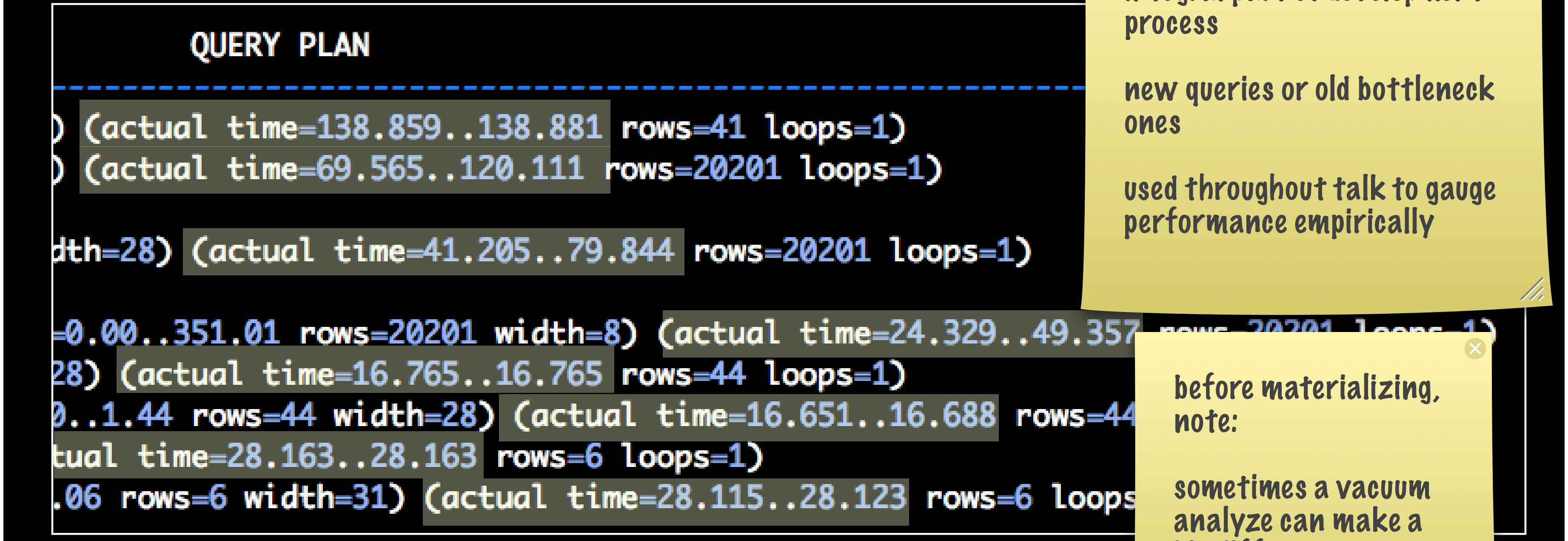
```
                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 HashAggregate  (cost=1061.17..1064.47 rows=264 width=51) (actual time=138.859..138.881 rows=41 loops
   ->  Hash Join  (cost=3.12..909.66 rows=20201 width=51) (actual time=69.565..120.111 rows=20201 loo
         Hash Cond: (ms.theatre_id = t.id)
         ->  Hash Join  (cost=1.99..630.76 rows=20201 width=28) (actual time=41.205..79.844 rows=2020
               Hash Cond: (ms.movie_id = m.id)
               ->  Seq Scan on movie_showtimes ms  (cost=0.00..351.01 rows=20201 width=8) (actual tim
               ->  Hash  (cost=1.44..1.44 rows=44 width=28) (actual time=16.765..16.765 rows=44 loops
                     ->  Seq Scan on movies m  (cost=0.00..1.44 rows=44 width=28) (actual time=16.651
         ->  Hash  (cost=1.06..1.06 rows=6 width=31) (actual time=28.163..28.163 rows=6 loops=1)
               ->  Seq Scan on theatres t  (cost=0.00..1.06 rows=6 width=31) (actual time=28.1
 Total runtime: 139.110 ms
(11 rows)
```

find the slow operation

seq scan? add index

operations out of order?
-> explicit join syntax

can't solve? matview!



```
       QUERY PLAN
--------------------------------------------------------
) (actual time=138.859..138.881 rows=41 loops=1)
) (actual time=69.565..120.111 rows=20201 loops=1)

dth=28) (actual time=41.205..79.844 rows=20201 loops=1)

=0.00..351.01 rows=20201 width=8) (actual time=24.329..49.357 rows=20201 loops=1)
28) (actual time=16.765..16.765 rows=44 loops=1)
0..1.44 rows=44 width=28) (actual time=16.651..16.688 rows=44
tual time=28.163..28.163 rows=6 loops=1)
.06 rows=6 width=31) (actual time=28.115..28.123 rows=6 loops
```

explain/analyze should be an integral part of development process

new queries or old bottleneck ones

used throughout talk to gauge performance empirically

before materializing, note:

sometimes a vacuum analyze can make a big difference

vacuum analyze

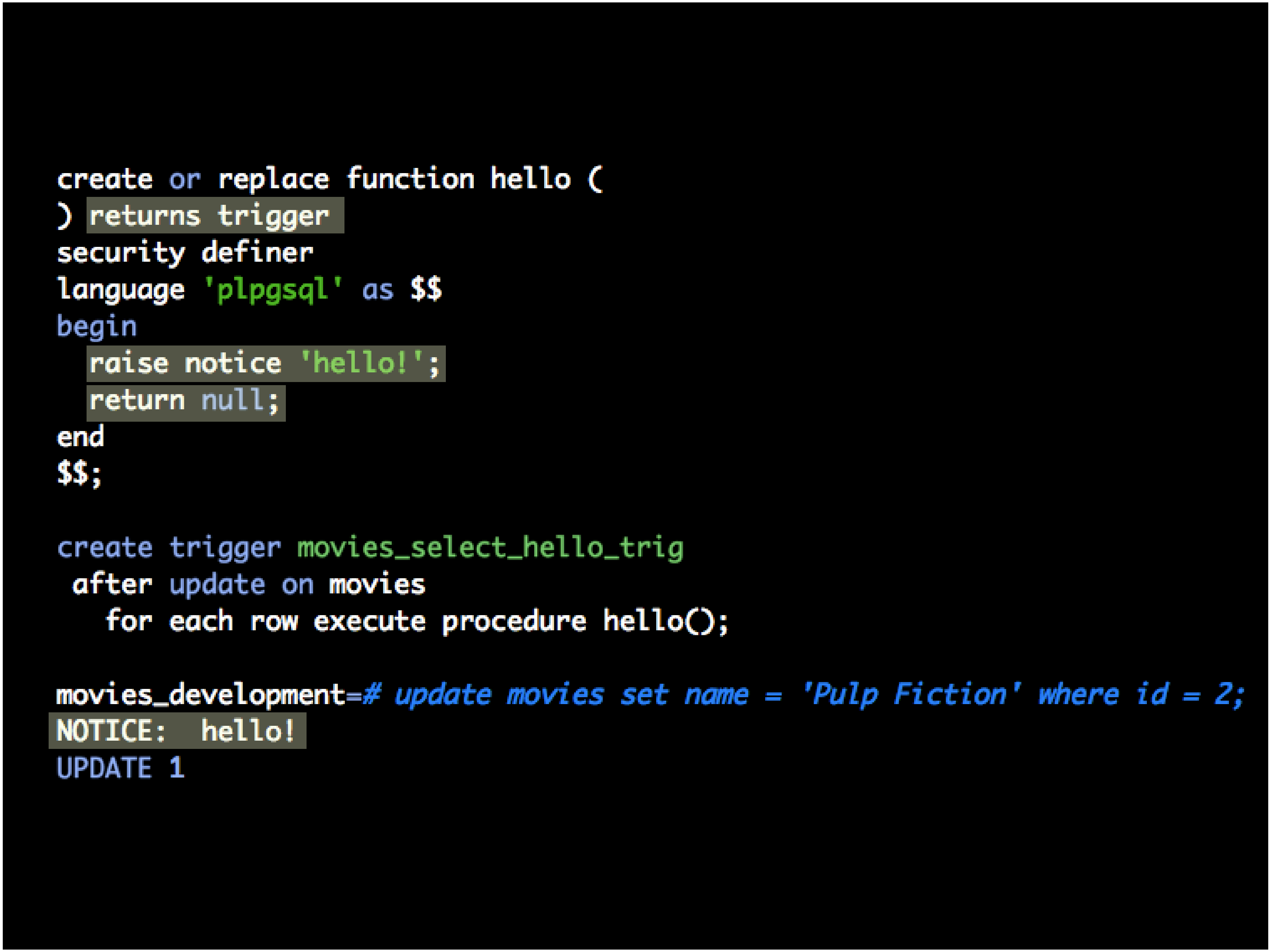
- each query generates statistics
- vacuum compacts database -- run nightly!
- vacuum analyze does same, also re-orders data on disk to improve performance based on statistics
- do everything you can to avoid materializing a view!

Stored Procedures

- procedural programming inside the DB
- PL/pgSQL, PL/TCL, PL/Java, etc.
- This talk: learn through examples

Triggers

```
CREATE TRIGGER name { BEFORE | AFTER } { event [ OR ... ] }
    ON table [ FOR [ EACH ] { ROW | STATEMENT } ]
    EXECUTE PROCEDURE funcname ( arguments )
```



```
create or replace function hello (
) returns trigger
security definer
language 'plpgsql' as $$
begin
  raise notice 'hello!';
  return null;
end
$$;

create trigger movies_select_hello_trig
 after update on movies
   for each row execute procedure hello();

movies_development=# update movies set name = 'Pulp Fiction' where id = 2;
NOTICE:  hello!
UPDATE 1
```

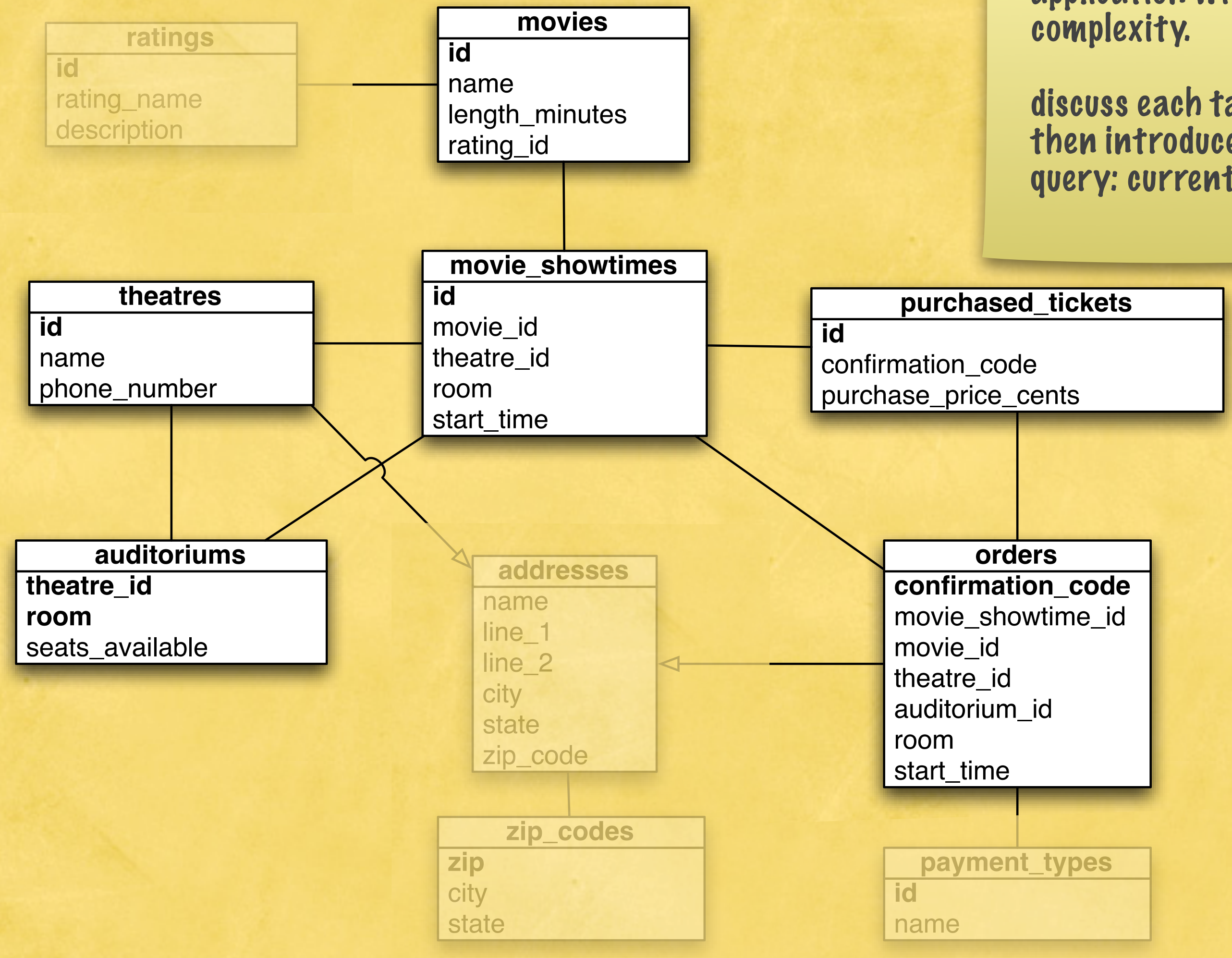
Part III: End to End

1. Considerations
2. Getting into form
3. The initial snapshot
4. Refresh function
5. Triggered Refresh
6. Indexing
7. Performance

Note: This will be an eager implementation!

Warning

- Although magical, obvious in retrospect
- Couple aha! moments, but easy once you know how



ratings
id
rating_name
description

movies
id
name
length_minutes
rating_id

movie_showtimes
id
movie_id
theatre_id
room
start_time

theatres
id
name
phone_number

purchased_tickets
id
confirmation_code
purchase_price_cents

auditoriums
theatre_id
room
seats_available

addresses
name
line_1
line_2
city
state
zip_code

orders
confirmation_code
movie_showtime_id
movie_id
theatre_id
auditorium_id
room
start_time

zip_codes
zip
city
state

payment_types
id
name

to make this interesting, need to use a real application with some complexity.

discuss each table in turn, then introduce common query: current showtimes

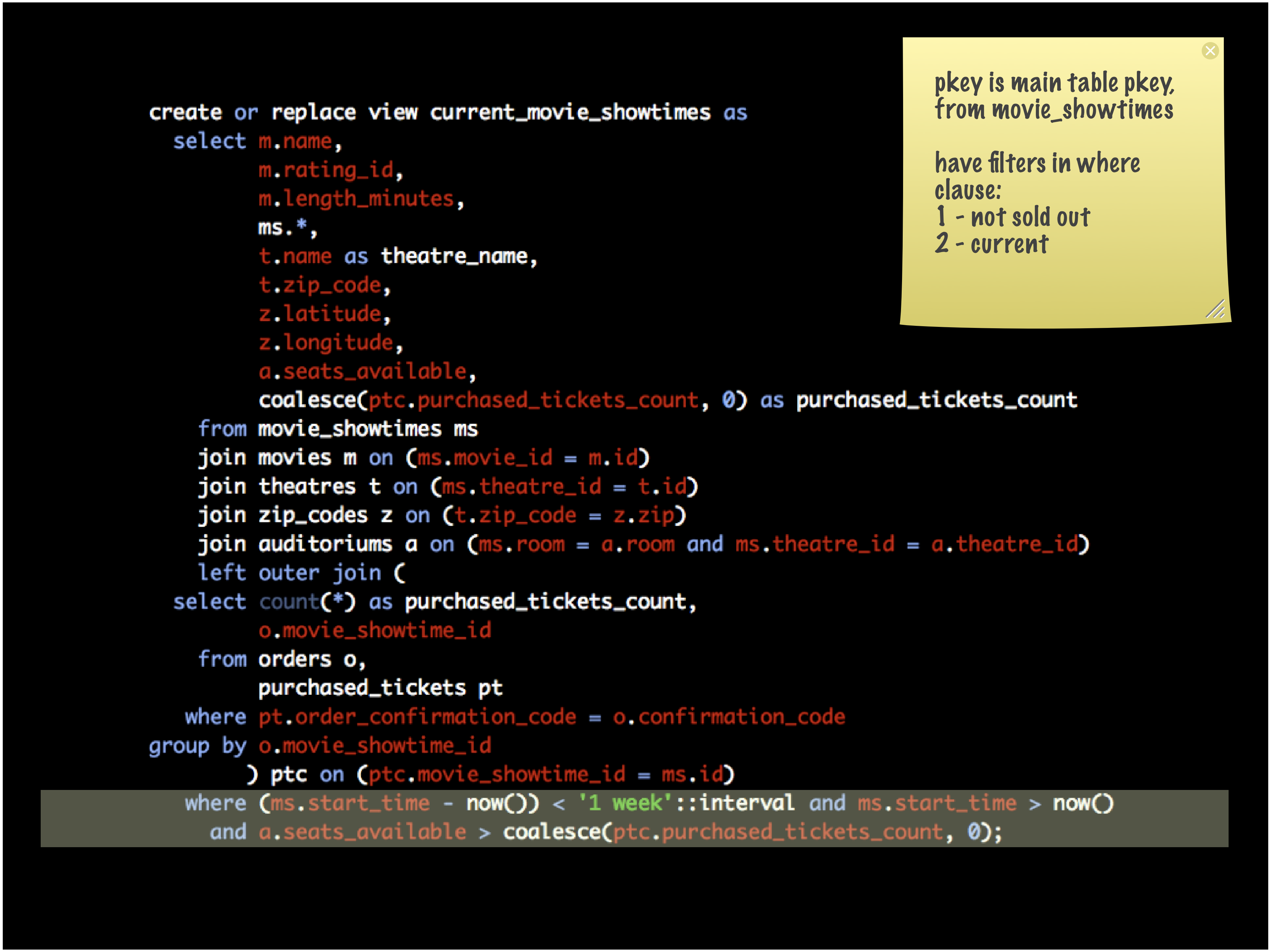
```
create or replace view current_movie_showtimes as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where (ms.start_time - now()) < '1 week'::interval and ms.start_time > now()
     and a.seats_available > coalesce(ptc.purchased_tickets_count, 0);
```

Considerations

- Should be transparent to end-user, drop-in replacement.
- Always accurate, up to date

Getting into form

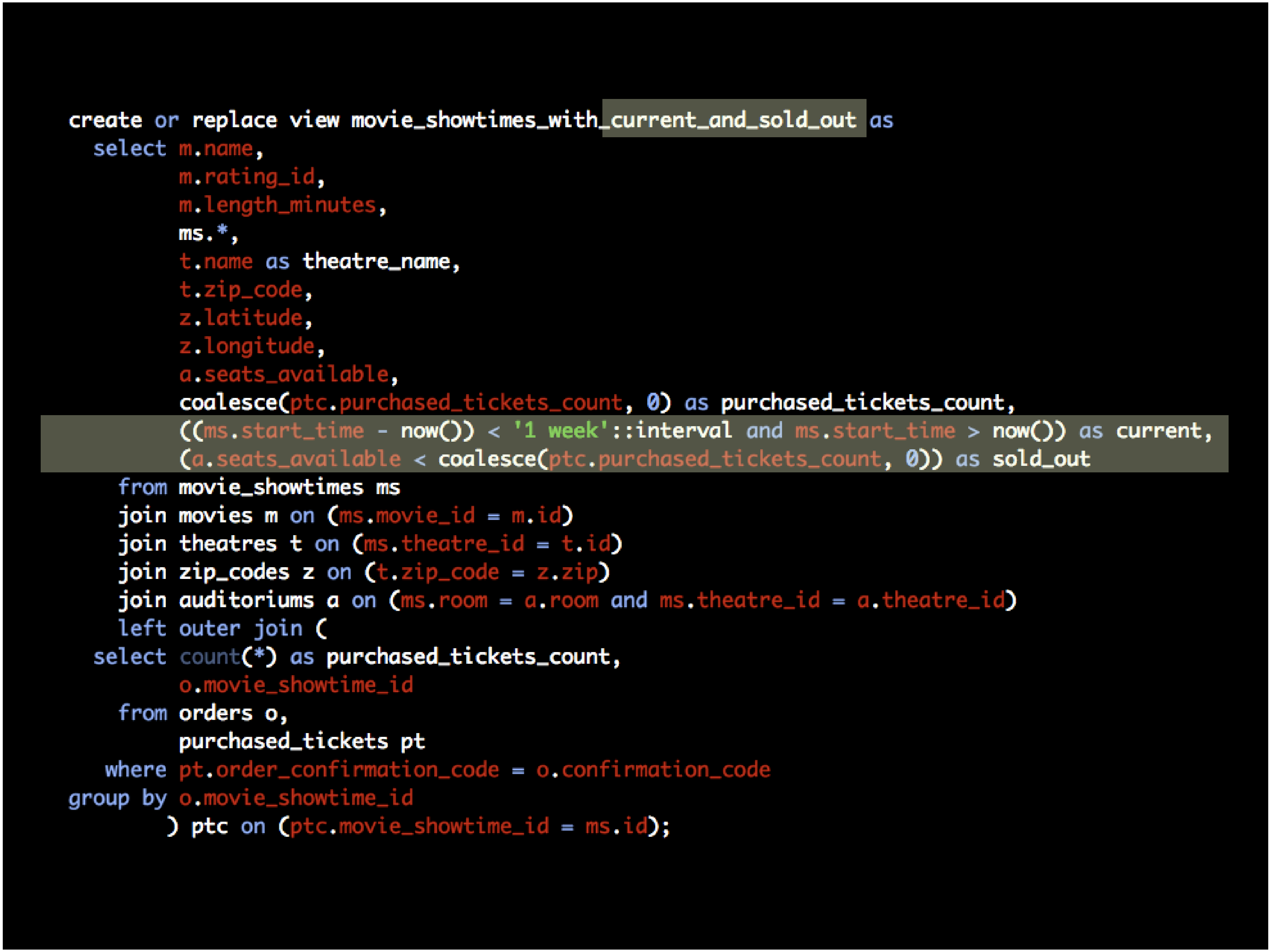
- view should have primary key
- recast filters as columns
- rename as _unmaterialized



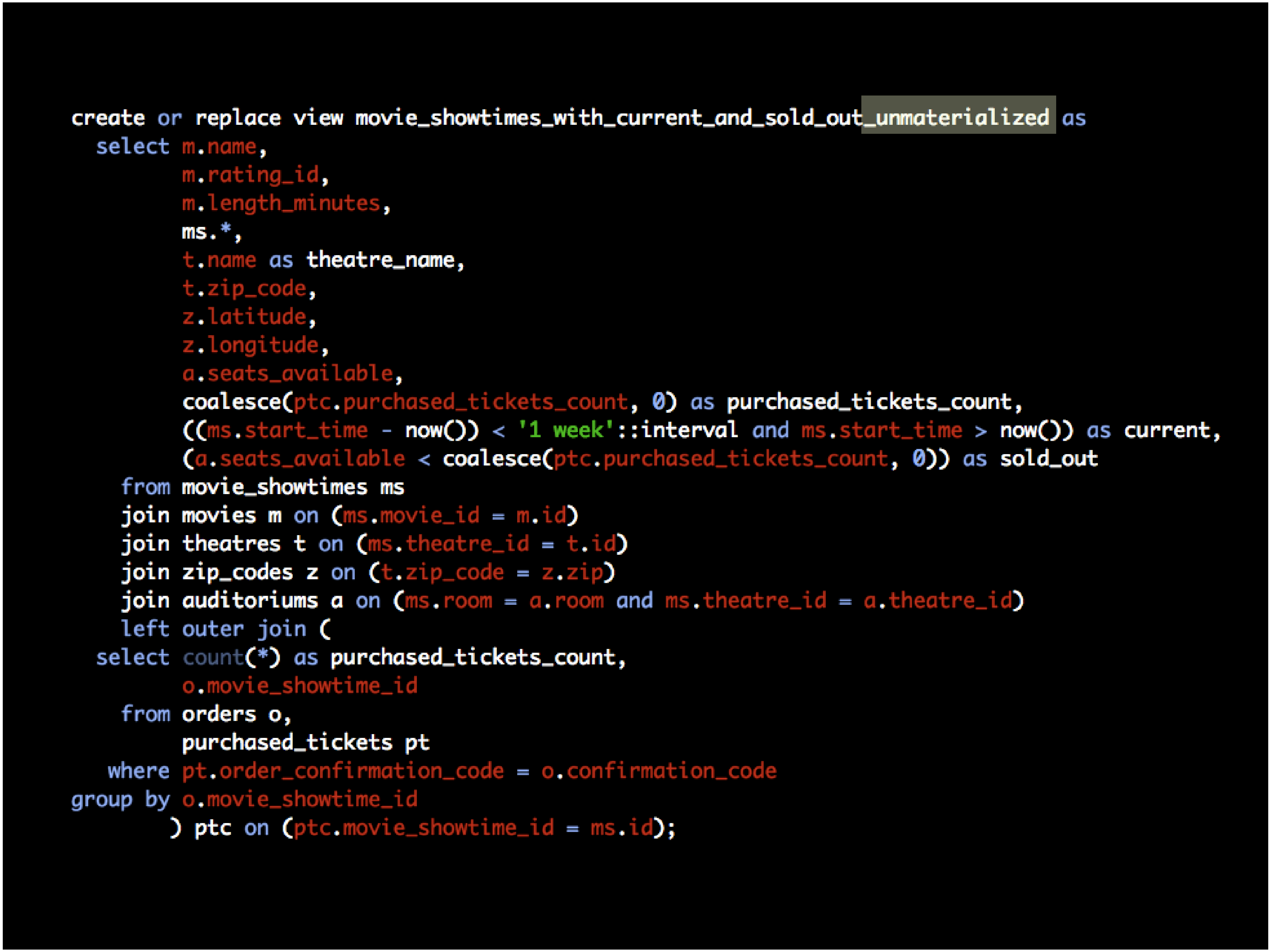
```
create or replace view current_movie_showtimes as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where (ms.start_time - now()) < '1 week'::interval and ms.start_time > now()
     and a.seats_available > coalesce(ptc.purchased_tickets_count, 0);
```

pkey is main table pkey,
from movie_showtimes

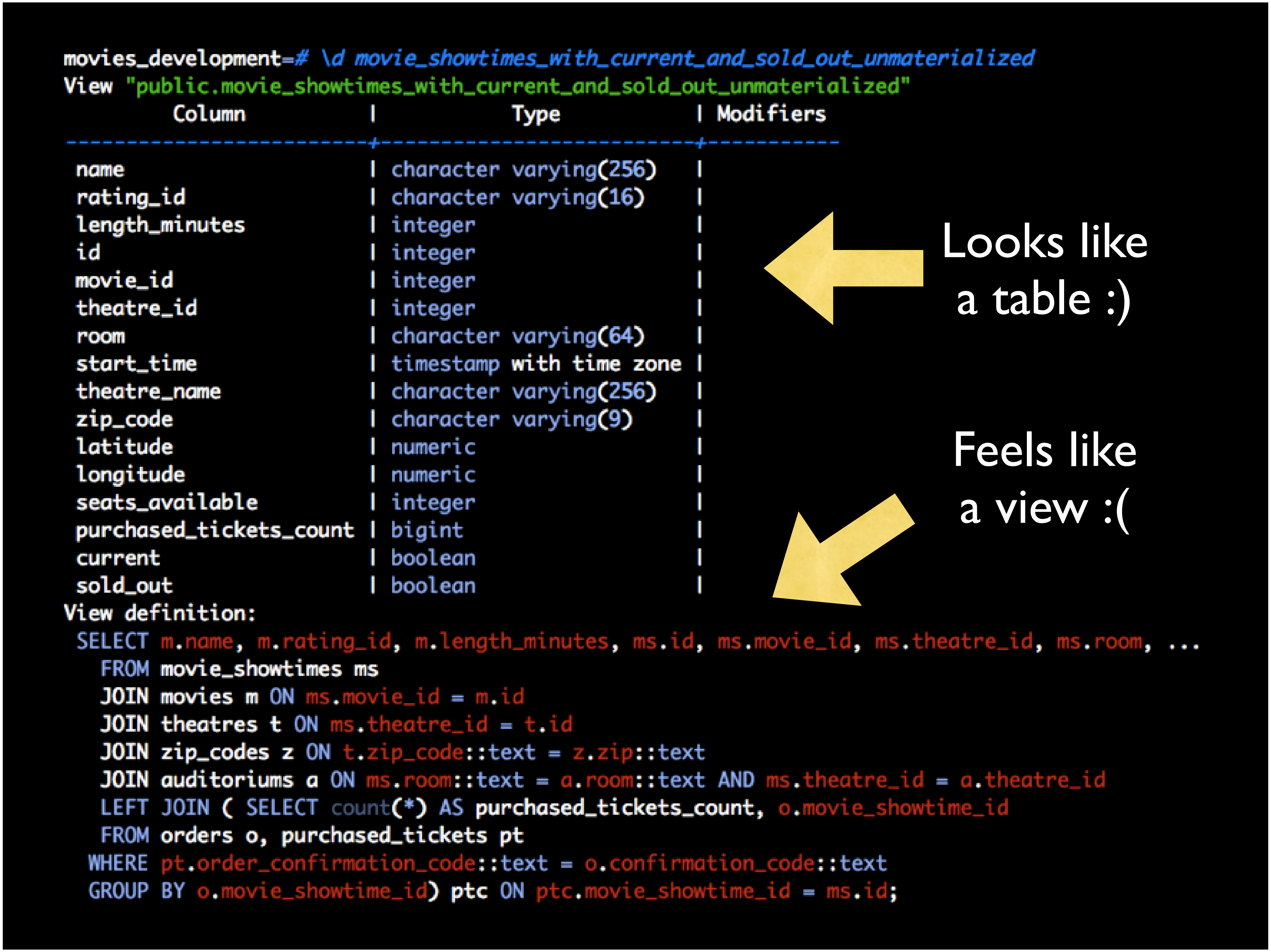
have filters in where
clause:
1 - not sold out
2 - current



```
create or replace view movie_showtimes_with_current_and_sold_out as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count,
         ((ms.start_time - now()) < '1 week'::interval and ms.start_time > now()) as current,
         (a.seats_available < coalesce(ptc.purchased_tickets_count, 0)) as sold_out
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id);
```



```
create or replace view movie_showtimes_with_current_and_sold_out_unmaterialized as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
         t.zip_code,
         z.latitude,
         z.longitude,
         a.seats_available,
         coalesce(ptc.purchased_tickets_count, 0) as purchased_tickets_count,
         ((ms.start_time - now()) < '1 week'::interval and ms.start_time > now()) as current,
         (a.seats_available < coalesce(ptc.purchased_tickets_count, 0)) as sold_out
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id);
```

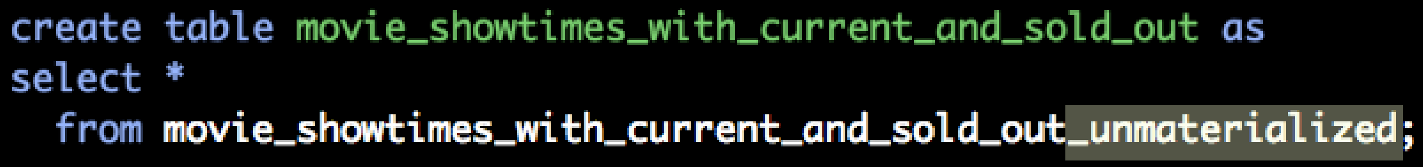


```
movies_development=# \d movie_showtimes_with_current_and_sold_out_unmaterialized
View "public.movie_showtimes_with_current_and_sold_out_unmaterialized"
         Column           |           Type           | Modifiers
--------------------------+--------------------------+-----------
 name                     | character varying(256)   |
 rating_id                | character varying(16)    |
 length_minutes           | integer                  |
 id                       | integer                  |
 movie_id                 | integer                  |
 theatre_id               | integer                  |
 room                     | character varying(64)    |
 start_time               | timestamp with time zone |
 theatre_name             | character varying(256)   |
 zip_code                 | character varying(9)     |
 latitude                 | numeric                  |
 longitude                | numeric                  |
 seats_available          | integer                  |
 purchased_tickets_count  | bigint                   |
 current                  | boolean                  |
 sold_out                 | boolean                  |
View definition:
 SELECT m.name, m.rating_id, m.length_minutes, ms.id, ms.movie_id, ms.theatre_id, ms.room, ...
   FROM movie_showtimes ms
   JOIN movies m ON ms.movie_id = m.id
   JOIN theatres t ON ms.theatre_id = t.id
   JOIN zip_codes z ON t.zip_code::text = z.zip::text
   JOIN auditoriums a ON ms.room::text = a.room::text AND ms.theatre_id = a.theatre_id
   LEFT JOIN ( SELECT count(*) AS purchased_tickets_count, o.movie_showtime_id
   FROM orders o, purchased_tickets pt
  WHERE pt.order_confirmation_code::text = o.confirmation_code::text
  GROUP BY o.movie_showtime_id) ptc ON ptc.movie_showtime_id = ms.id;
```

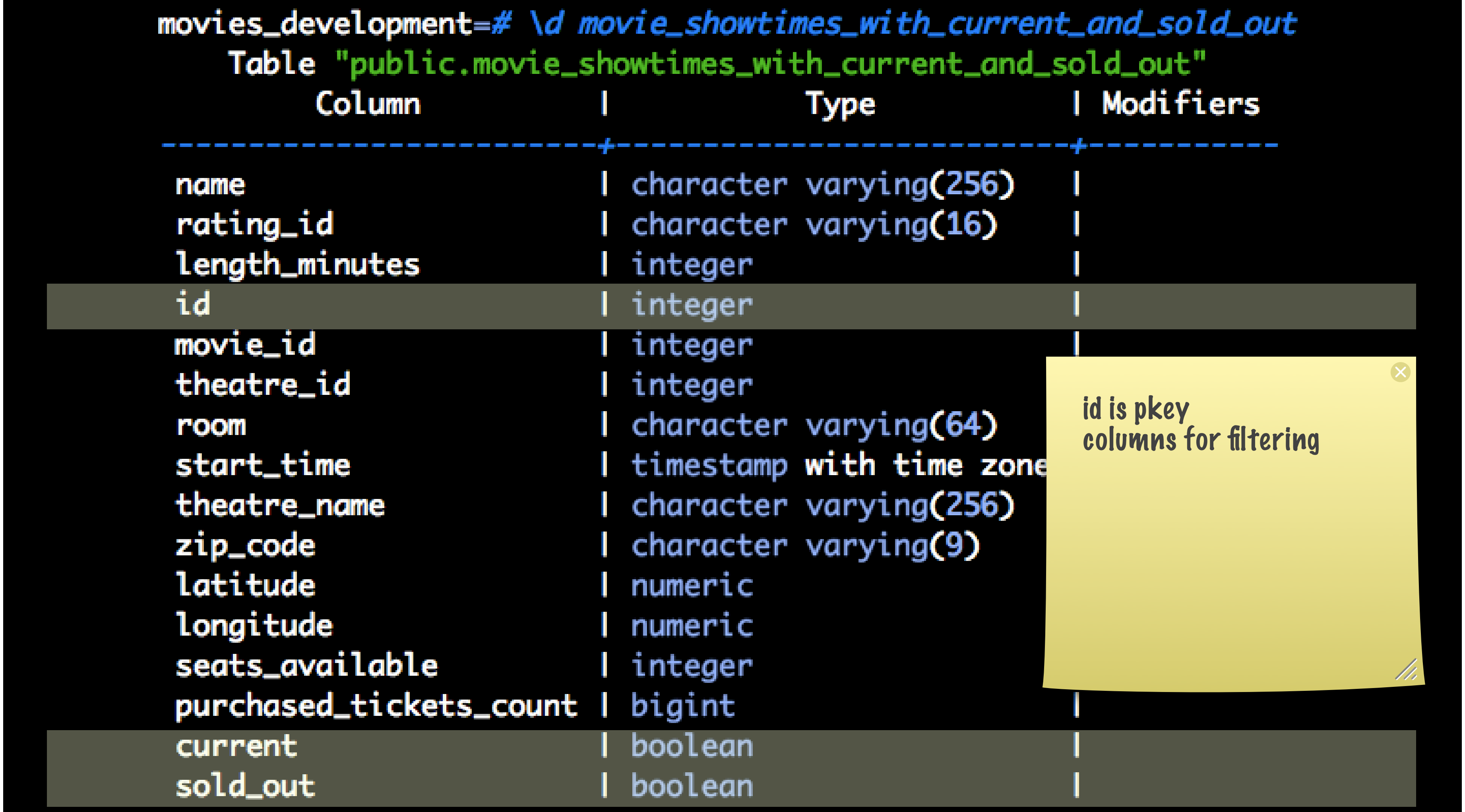
Looks like
a table :)

Feels like
a view :(

Initial Snapshot



```
create table movie_showtimes_with_current_and_sold_out as
select *
  from movie_showtimes_with_current_and_sold_out_unmaterialized;
```



```
movies_development=# \d movie_showtimes_with_current_and_sold_out
    Table "public.movie_showtimes_with_current_and_sold_out"
        Column          |           Type           | Modifiers
-------------------------+--------------------------+-----------
 name                    | character varying(256)   |
 rating_id               | character varying(16)    |
 length_minutes          | integer                  |
 id                      | integer                  |
 movie_id                | integer                  |
 theatre_id              | integer
 room                    | character varying(64)
 start_time              | timestamp with time zone
 theatre_name            | character varying(256)
 zip_code                | character varying(9)
 latitude                | numeric
 longitude               | numeric
 seats_available         | integer
 purchased_tickets_count | bigint                   |
 current                 | boolean                  |
 sold_out                | boolean                  |
```

id is pkey
columns for filtering

nothing here because this is a real table!

Indexing

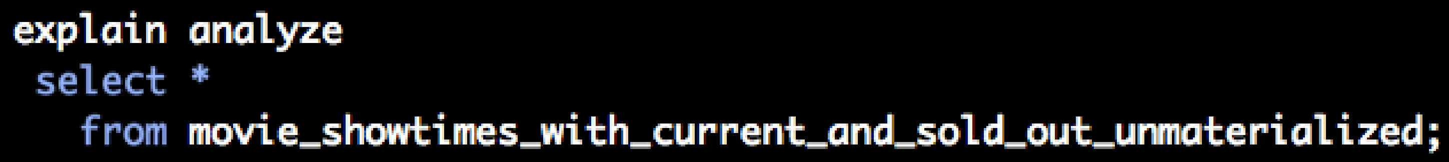
- materialized view is a regular table, so benefits greatly from indexes
- index minimally: pkey, filter columns
- also index: anything you may search on
- avoid over-indexing -- performance performance performance!

```
alter table movie_showtimes_with_current_and_sold_out add primary key (id);

create index movie_showtimes_with_current_and_sold_out_current_idx
  on movie_showtimes_with_current_and_sold_out(current);

create index movie_showtimes_with_current_and_sold_out_sold_out_idx
  on movie_showtimes_with_current_and_sold_out(sold_out);
```

Constraints? RI?

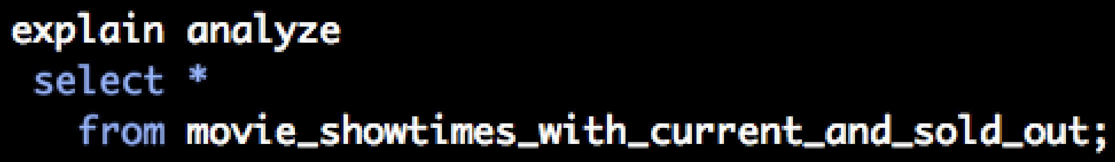
- materialized view should not have constraints or enforced foreign key references
- MV can be temporarily stale
- base tables should have these already, so just slows things down

Initial Comparisons

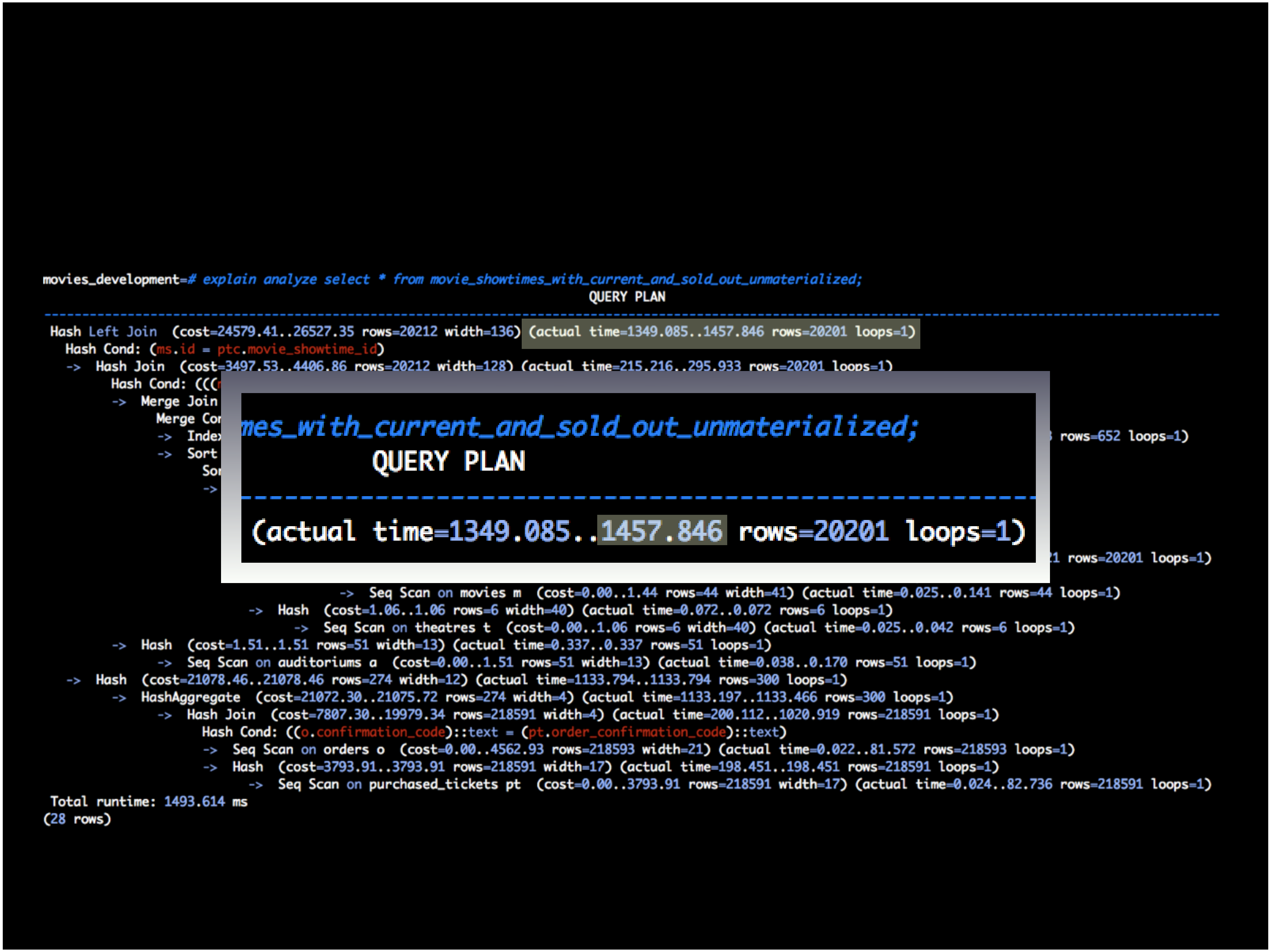


```
explain analyze
 select *
   from movie_showtimes_with_current_and_sold_out_unmaterialized;
```

versus

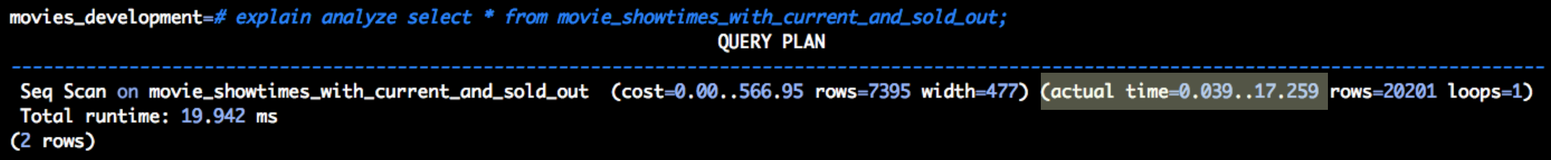


```
explain analyze
 select *
   from movie_showtimes_with_current_and_sold_out;
```

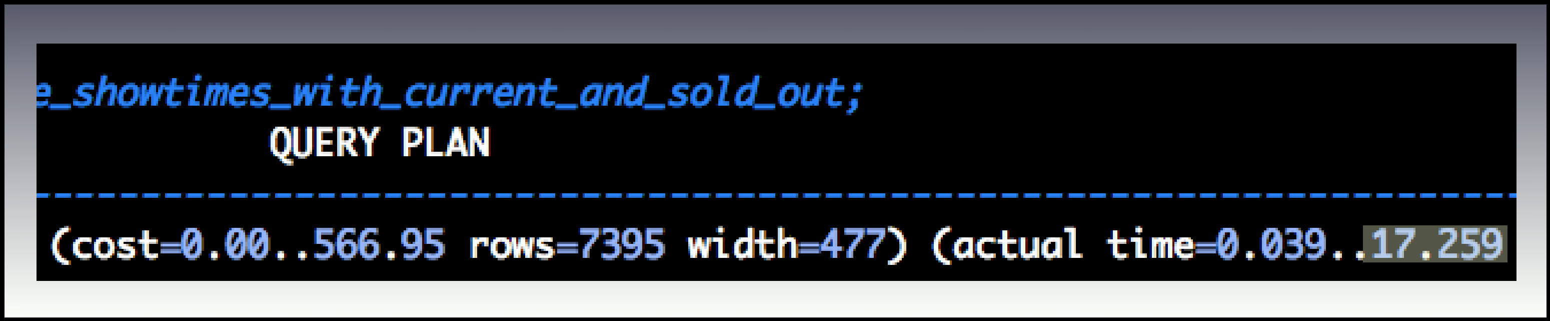


```
movies_development=# explain analyze select * from movie_showtimes_with_current_and_sold_out_unmaterialized;
                                                                     QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------------
 Hash Left Join  (cost=24579.41..26527.35 rows=20212 width=136) (actual time=1349.085..1457.846 rows=20201 loops=1)
   Hash Cond: (ms.id = ptc.movie_showtime_id)
   ->  Hash Join  (cost=3497.53..4406.86 rows=20212 width=128) (actual time=215.216..295.933 rows=20201 loops=1)
         Hash Cond: (((
         ->  Merge Join
               Merge Cor
               ->  Index                                                                                          rows=652 loops=1)
               ->  Sort
                     Sor
                     ->


                                                                                                               1 rows=20201 loops=1)
                           ->  Seq Scan on movies m  (cost=0.00..1.44 rows=44 width=41) (actual time=0.025..0.141 rows=44 loops=1)
                     ->  Hash  (cost=1.06..1.06 rows=6 width=40) (actual time=0.072..0.072 rows=6 loops=1)
                           ->  Seq Scan on theatres t  (cost=0.00..1.06 rows=6 width=40) (actual time=0.025..0.042 rows=6 loops=1)
         ->  Hash  (cost=1.51..1.51 rows=51 width=13) (actual time=0.337..0.337 rows=51 loops=1)
               ->  Seq Scan on auditoriums a  (cost=0.00..1.51 rows=51 width=13) (actual time=0.038..0.170 rows=51 loops=1)
   ->  Hash  (cost=21078.46..21078.46 rows=274 width=12) (actual time=1133.794..1133.794 rows=300 loops=1)
         ->  HashAggregate  (cost=21072.30..21075.72 rows=274 width=4) (actual time=1133.197..1133.466 rows=300 loops=1)
               ->  Hash Join  (cost=7807.30..19979.34 rows=218591 width=4) (actual time=200.112..1020.919 rows=218591 loops=1)
                     Hash Cond: ((o.confirmation_code)::text = (pt.order_confirmation_code)::text)
                     ->  Seq Scan on orders o  (cost=0.00..4562.93 rows=218593 width=21) (actual time=0.022..81.572 rows=218593 loops=1)
                     ->  Hash  (cost=3793.91..3793.91 rows=218591 width=17) (actual time=198.451..198.451 rows=218591 loops=1)
                           ->  Seq Scan on purchased_tickets pt  (cost=0.00..3793.91 rows=218591 width=17) (actual time=0.024..82.736 rows=218591 loops=1)
 Total runtime: 1493.614 ms
(28 rows)
```

```
mes_with_current_and_sold_out_unmaterialized;
         QUERY PLAN
---------------------------------------------------
(actual time=1349.085..1457.846 rows=20201 loops=1)
```

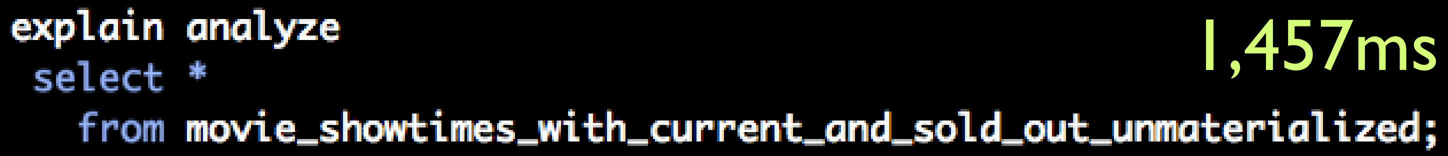


```
movies_development=# explain analyze select * from movie_showtimes_with_current_and_sold_out;
                                                                  QUERY PLAN
----------------------------------------------------------------------------------------------------------------------------------------------------
 Seq Scan on movie_showtimes_with_current_and_sold_out  (cost=0.00..566.95 rows=7395 width=477) (actual time=0.039..17.259 rows=20201 loops=1)
 Total runtime: 19.942 ms
(2 rows)
```



```
e_showtimes_with_current_and_sold_out;
               QUERY PLAN
------------------------------------------------------------
 (cost=0.00..566.95 rows=7395 width=477) (actual time=0.039..17.259
```

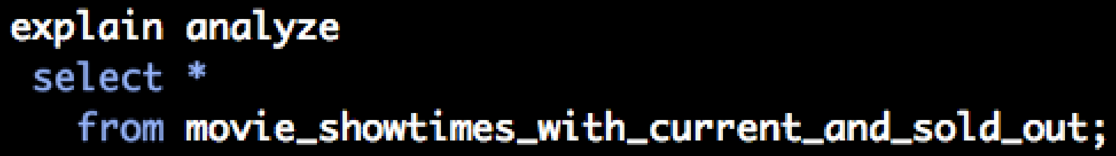
Initial Comparisons



```
explain analyze
 select *
   from movie_showtimes_with_current_and_sold_out_unmaterialized;
```

1,457ms

versus



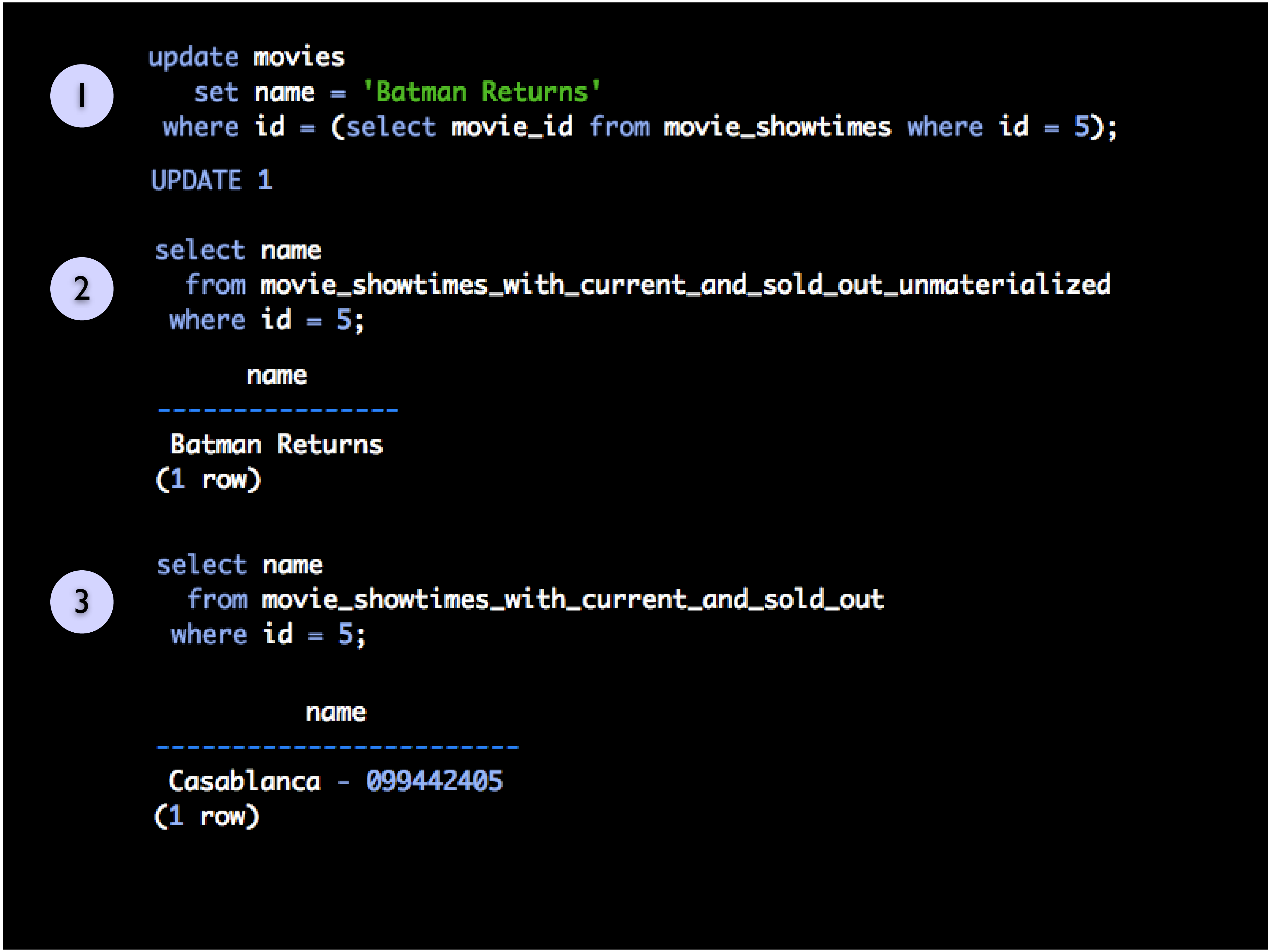
```
explain analyze
 select *
   from movie_showtimes_with_current_and_sold_out;
```

17ms

materialized view is 85 times faster!

Refresh Function

The materialized view is fast,
but it’s not accurate



1

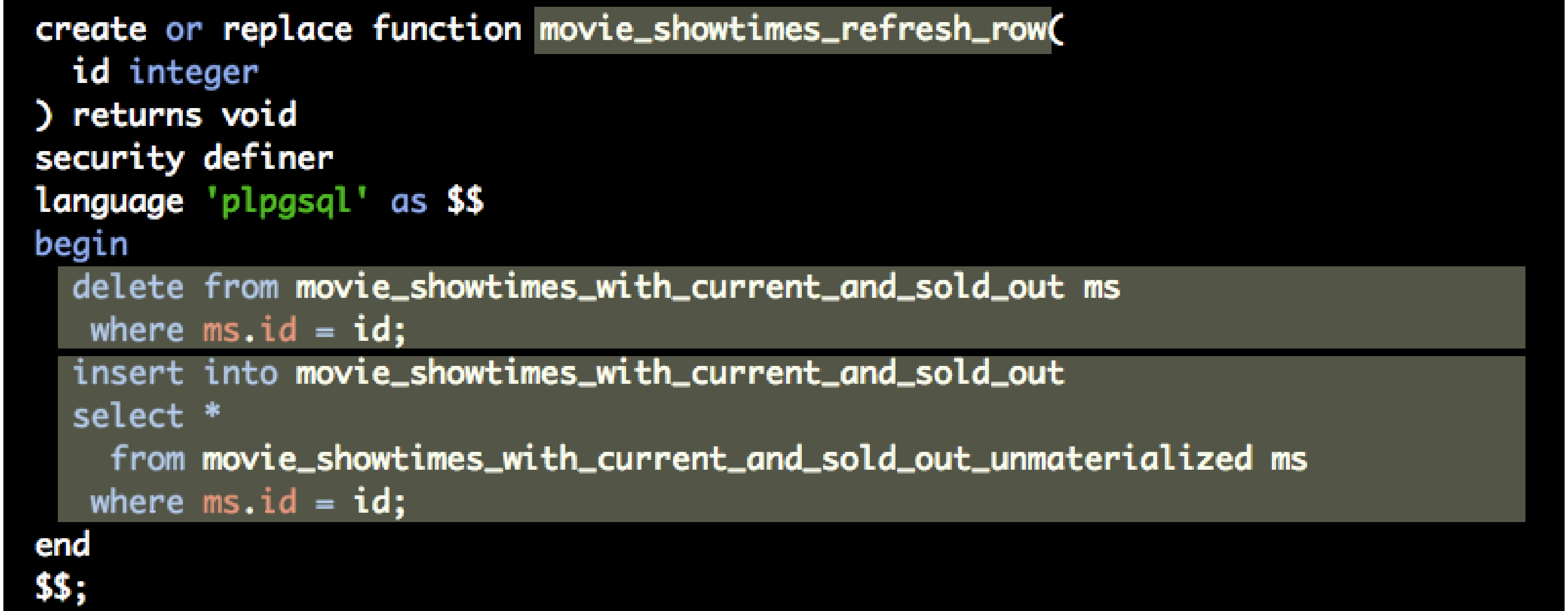
```
update movies
   set name = 'Batman Returns'
 where id = (select movie_id from movie_showtimes where id = 5);

UPDATE 1
```

2

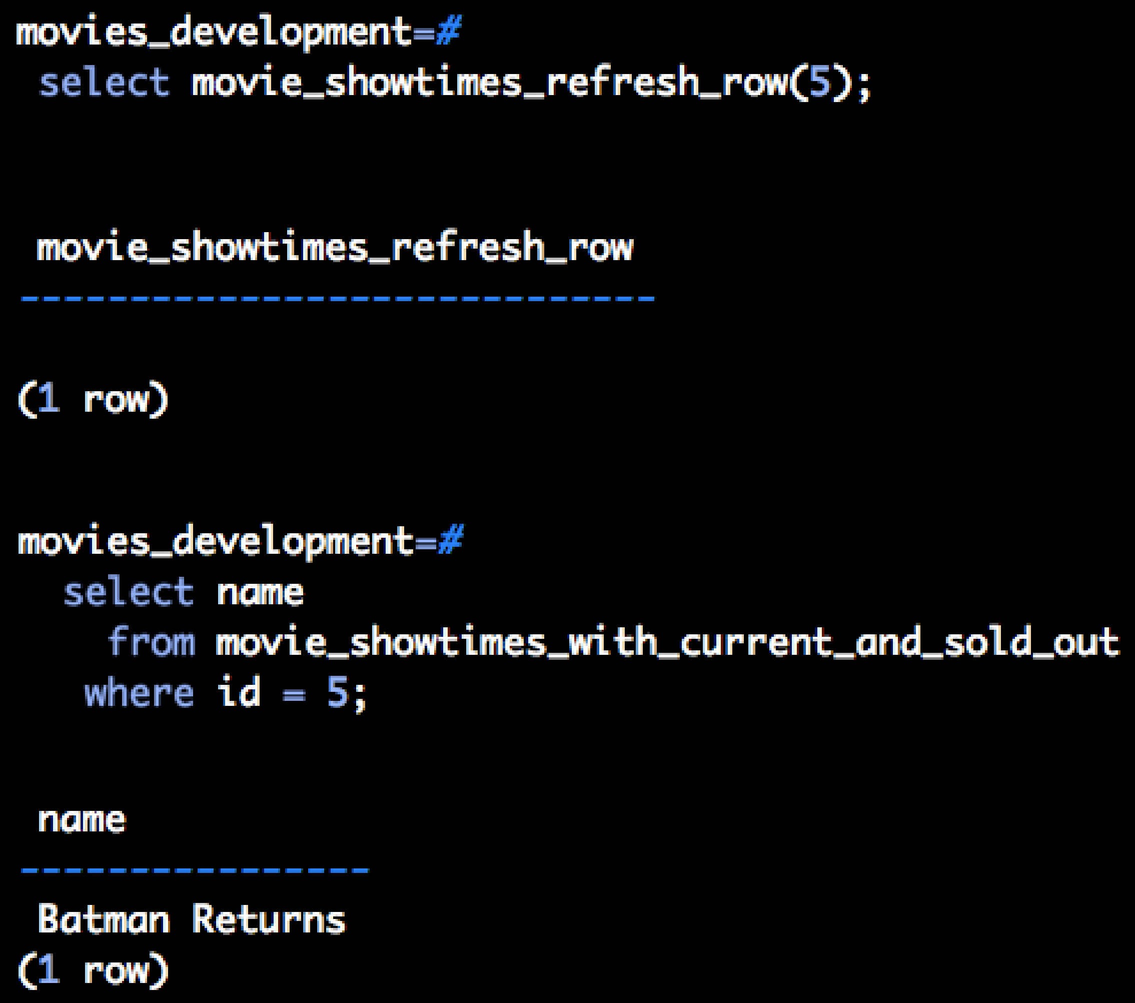
```
select name
  from movie_showtimes_with_current_and_sold_out_unmaterialized
 where id = 5;

      name
----------------
 Batman Returns
(1 row)
```

3

```
select name
  from movie_showtimes_with_current_and_sold_out
 where id = 5;

          name
------------------------
 Casablanca - 099442405
(1 row)
```



```
create or replace function movie_showtimes_refresh_row(
  id integer
) returns void
security definer
language 'plpgsql' as $$
begin
  delete from movie_showtimes_with_current_and_sold_out ms
   where ms.id = id;
  insert into movie_showtimes_with_current_and_sold_out
  select *
    from movie_showtimes_with_current_and_sold_out_unmaterialized ms
   where ms.id = id;
end
$$;
```



1

```
movies_development=#
 select movie_showtimes_refresh_row(5);


 movie_showtimes_refresh_row
-----------------------------

(1 row)
```

2

```
movies_development=#
  select name
    from movie_showtimes_with_current_and_sold_out
   where id = 5;


 name
----------------
 Batman Returns
(1 row)
```

Triggered Refresh

- refresh function works great, but we need it to happen automatically
- accomplished with triggers attached to all base tables

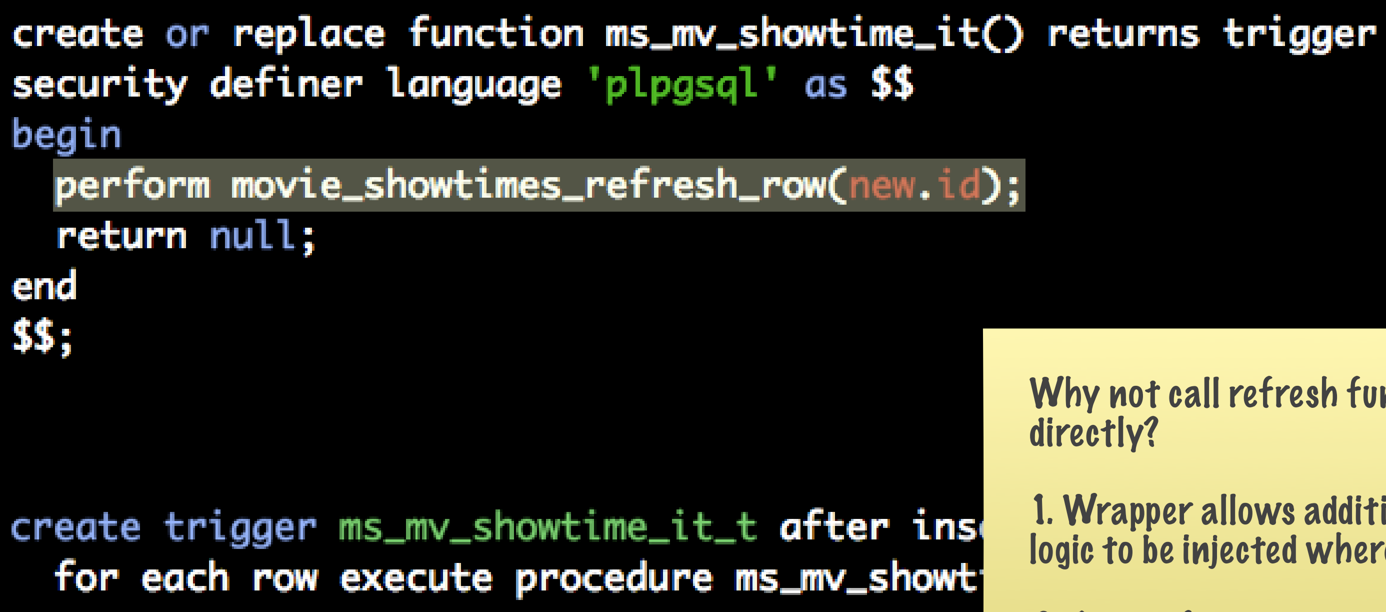
Refresh Triggers 101

- is a refresh needed for this operation?
- is it only needed under certain conditions?

Triggers - old, new

- insert trigger:
  refresh new row
- delete trigger:
  refresh old row
- update trigger:
  if pkey changes, refresh old, new; else either

movie_showtimes insert



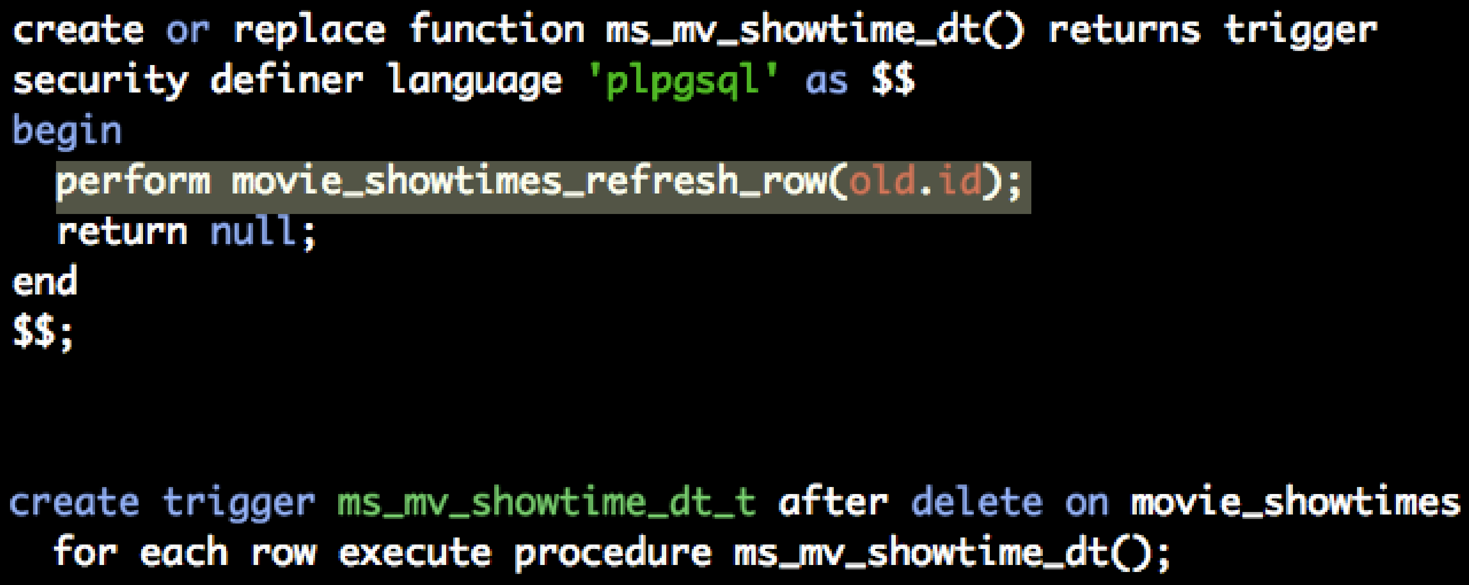
```
create or replace function ms_mv_showtime_it() returns trigger
security definer language 'plpgsql' as $$
begin
  perform movie_showtimes_refresh_row(new.id);
  return null;
end
$$;


create trigger ms_mv_showtime_it_t after ins
  for each row execute procedure ms_mv_showt
```

Why not call refresh function directly?

1. Wrapper allows additional logic to be injected where needed.

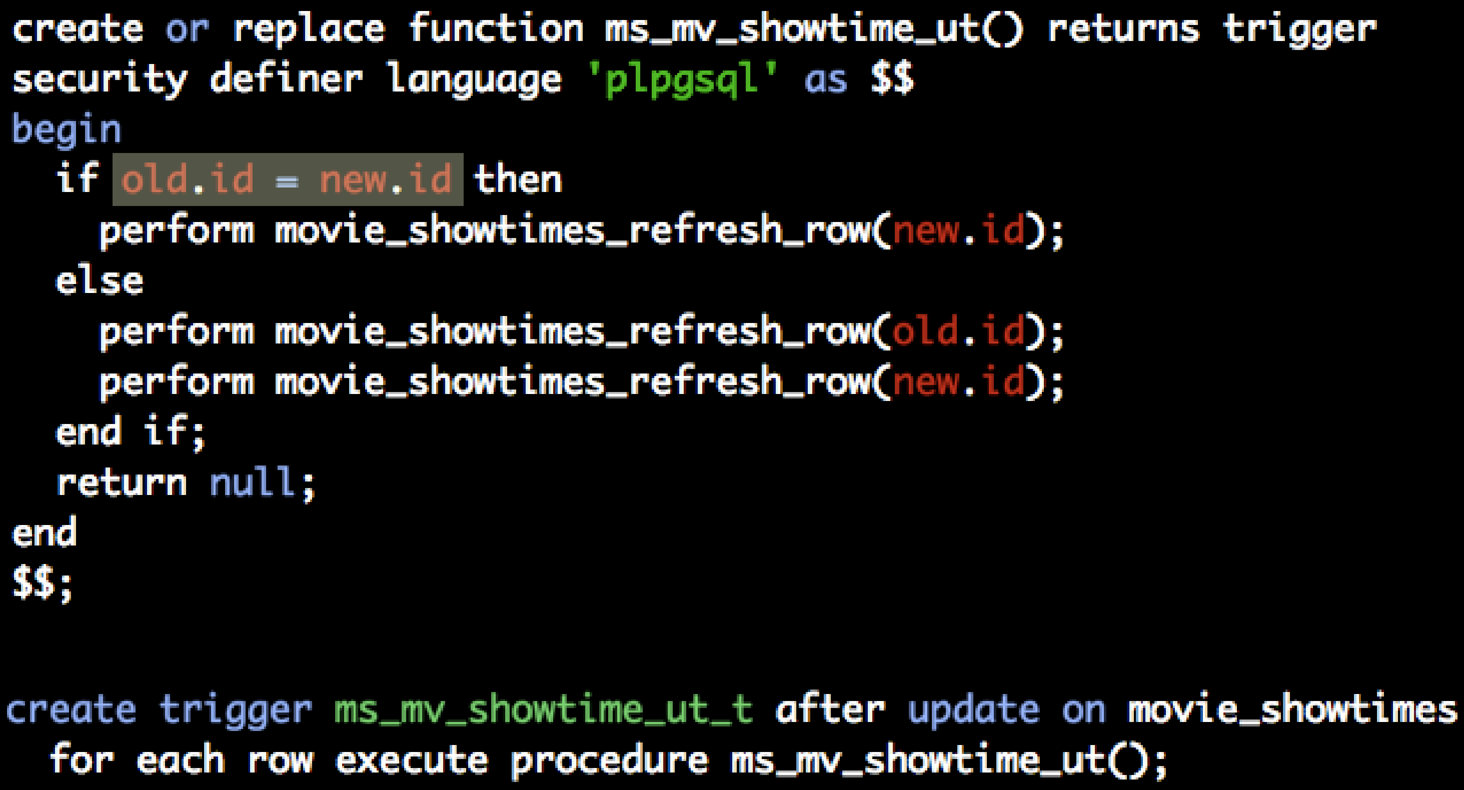
2. Trigger functions must return null or row. We're going to play w/ return val of refresh function soon.

movie_showtimes delete



```
create or replace function ms_mv_showtime_dt() returns trigger
security definer language 'plpgsql' as $$
begin
  perform movie_showtimes_refresh_row(old.id);
  return null;
end
$$;


create trigger ms_mv_showtime_dt_t after delete on movie_showtimes
  for each row execute procedure ms_mv_showtime_dt();
```

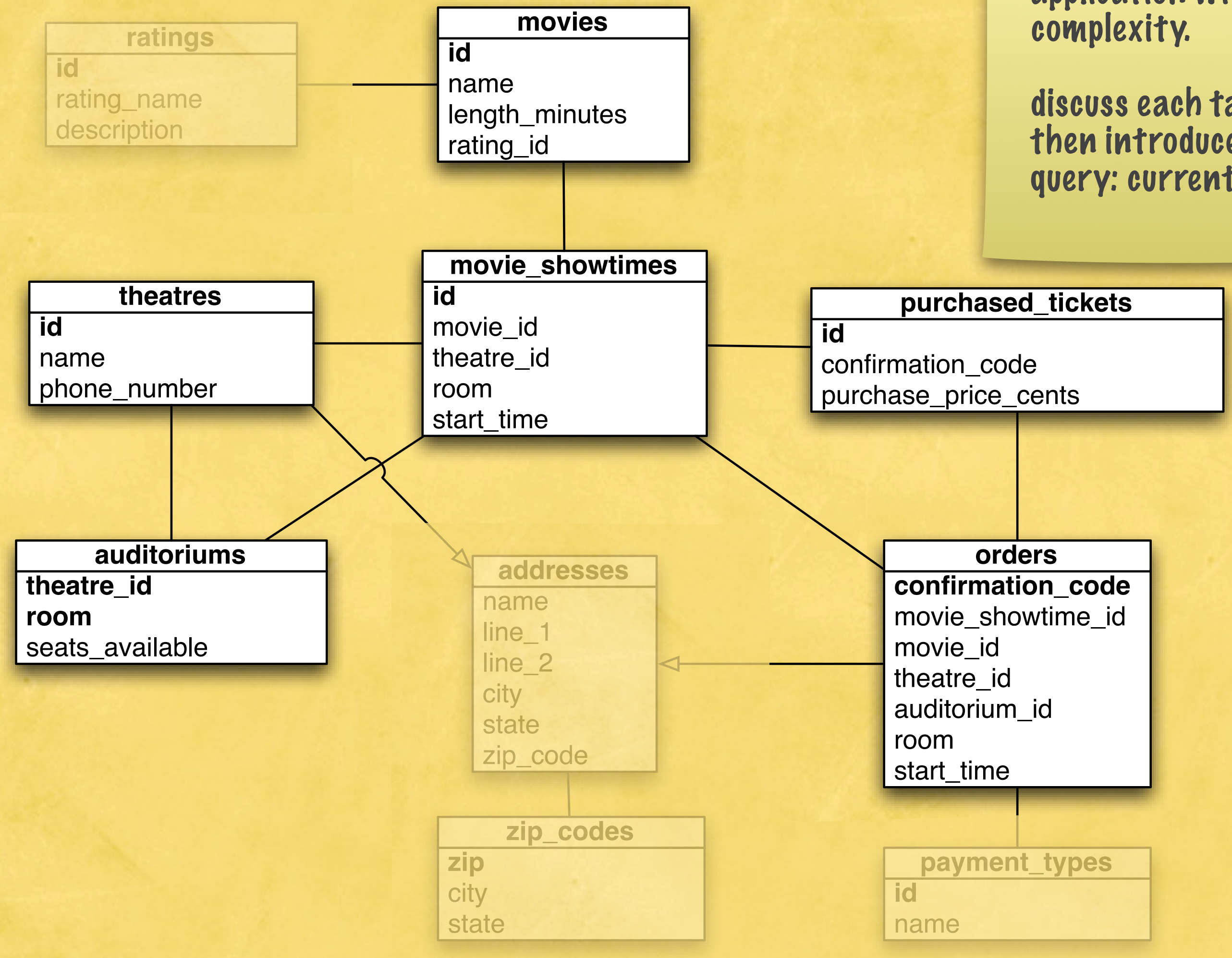
movie_showtimes update



```
create or replace function ms_mv_showtime_ut() returns trigger
security definer language 'plpgsql' as $$
begin
  if old.id = new.id then
    perform movie_showtimes_refresh_row(new.id);
  else
    perform movie_showtimes_refresh_row(old.id);
    perform movie_showtimes_refresh_row(new.id);
  end if;
  return null;
end
$$;


create trigger ms_mv_showtime_ut_t after update on movie_showtimes
  for each row execute procedure ms_mv_showtime_ut();
```

and repeat...

- same process for every table
- except when not needed



to make this interesting, need to use a real application with some complexity.

discuss each table in turn, then introduce common query: current showtimes

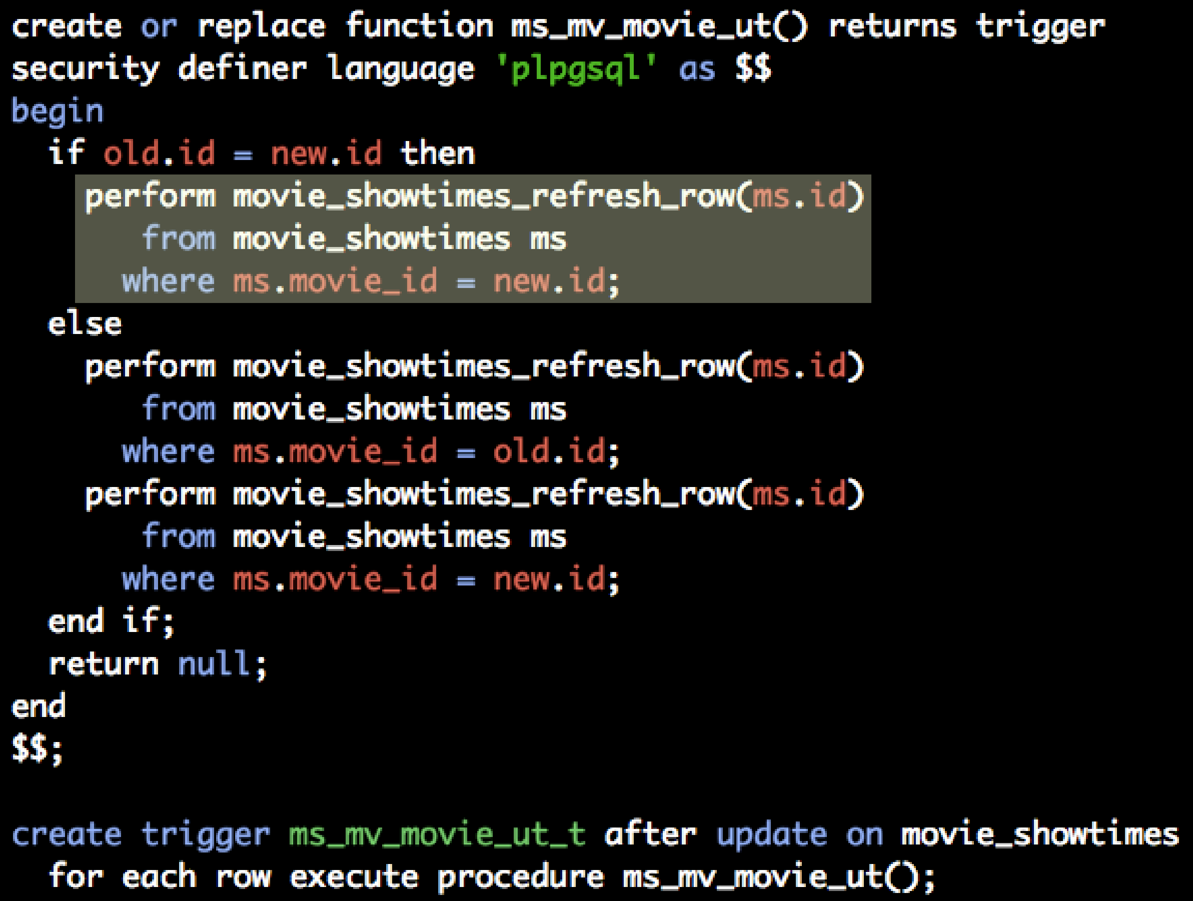
table	action	refresh needed?
movie_showtimes	insert	x
	update	x
	delete	x
movies	insert	
	update	x
	delete	
theatres	insert	
	update	x
	delete	
	insert	
	update	x
	delete	
ases	insert	x
	update	x
	delete	x
auditoriums	insert	
	update	
	delete	

18 possible triggers,
only 9 needed

building this table is a big help.

fewer refreshes = faster db,
faster user-perceived performance

table	action	refresh needed?
movies	insert	
	update	x
	delete	



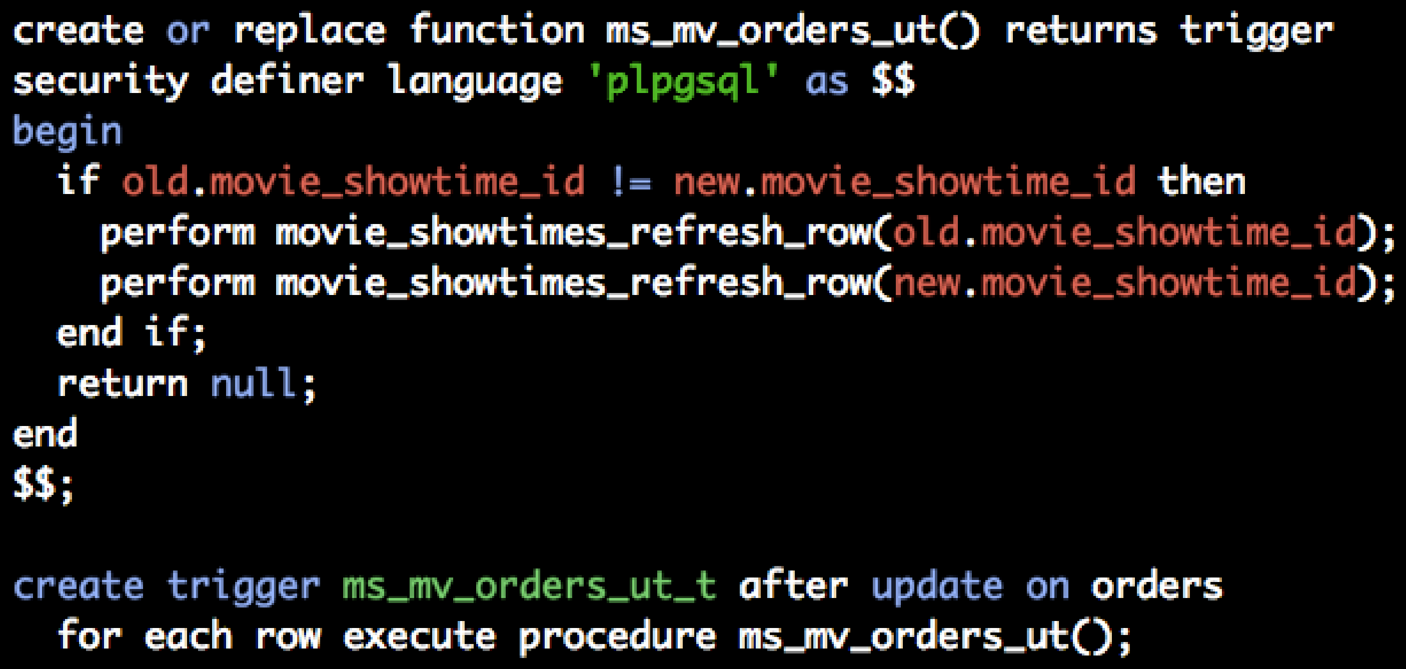
```
create or replace function ms_mv_movie_ut() returns trigger
security definer language 'plpgsql' as $$
begin
  if old.id = new.id then
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.movie_id = new.id;
  else
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.movie_id = old.id;
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.movie_id = new.id;
  end if;
  return null;
end
$$;

create trigger ms_mv_movie_ut_t after update on movie_showtimes
  for each row execute procedure ms_mv_movie_ut();
```

table	action	refresh needed?
theatres	insert	
	update	x
	delete	

```
create or replace function ms_mv_theatre_ut() returns trigger
security definer language 'plpgsql' as $$
begin
  if old.id = new.id then
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.theatre_id = new.id;
  else
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.theatre_id = old.id;
    perform movie_showtimes_refresh_row(ms.id)
       from movie_showtimes ms
      where ms.theatre_id = new.id;
  end if;
  return null;
end
$$;

create or replace trigger ms_mv_theatre_ut_t after update on theatres
  for each row execute procedure ms_mv_theatre_ut();
```

table	action	refresh needed?
orders	insert	
	update	x
	delete	

only if the showtime changes

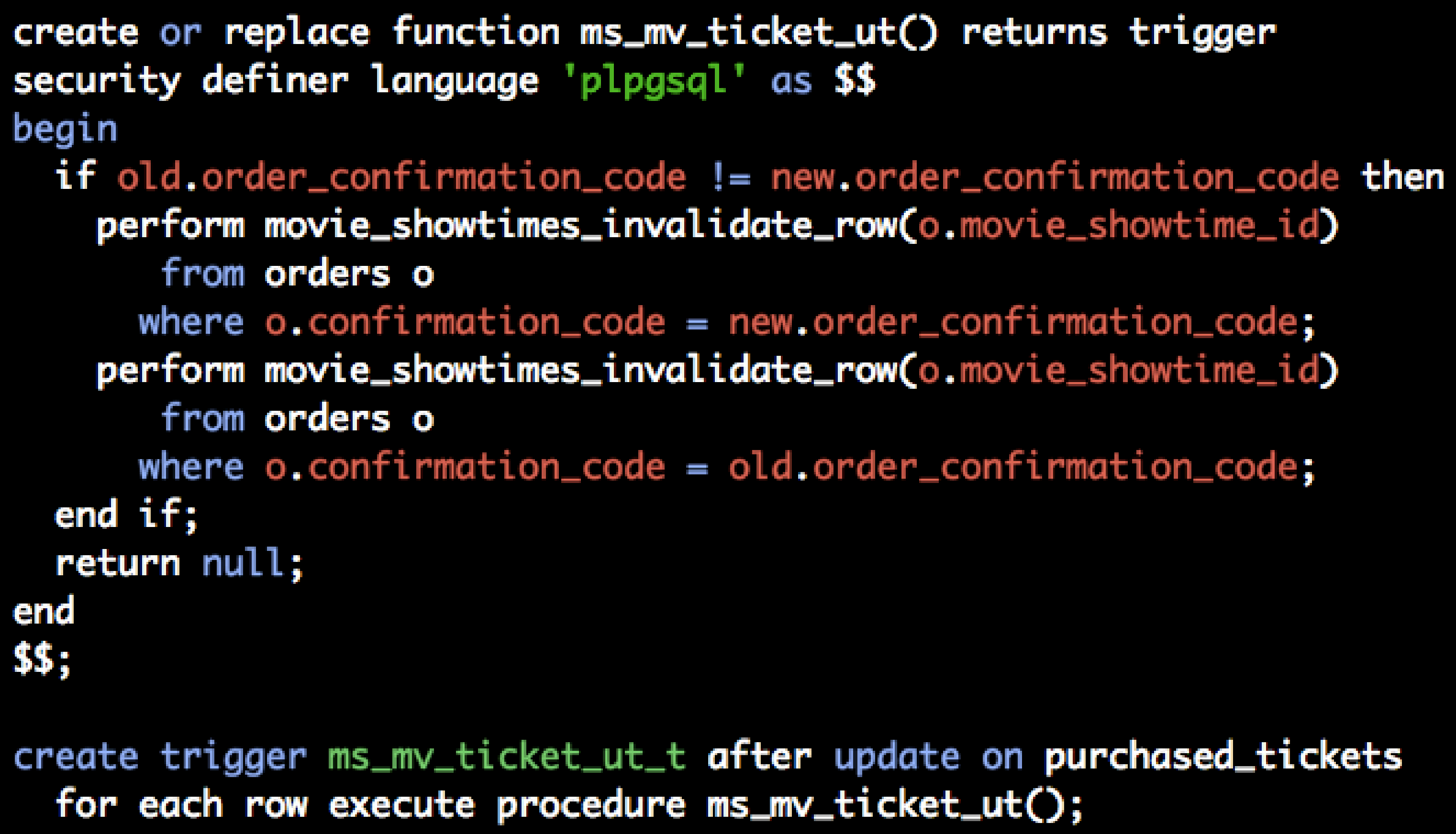


```
create or replace function ms_mv_orders_ut() returns trigger
security definer language 'plpgsql' as $$
begin
  if old.movie_showtime_id != new.movie_showtime_id then
    perform movie_showtimes_refresh_row(old.movie_showtime_id);
    perform movie_showtimes_refresh_row(new.movie_showtime_id);
  end if;
  return null;
end
$$;

create trigger ms_mv_orders_ut_t after update on orders
  for each row execute procedure ms_mv_orders_ut();
```

table	action	refresh needed?
ticket_purchases	insert	x
	update	x
	delete	x

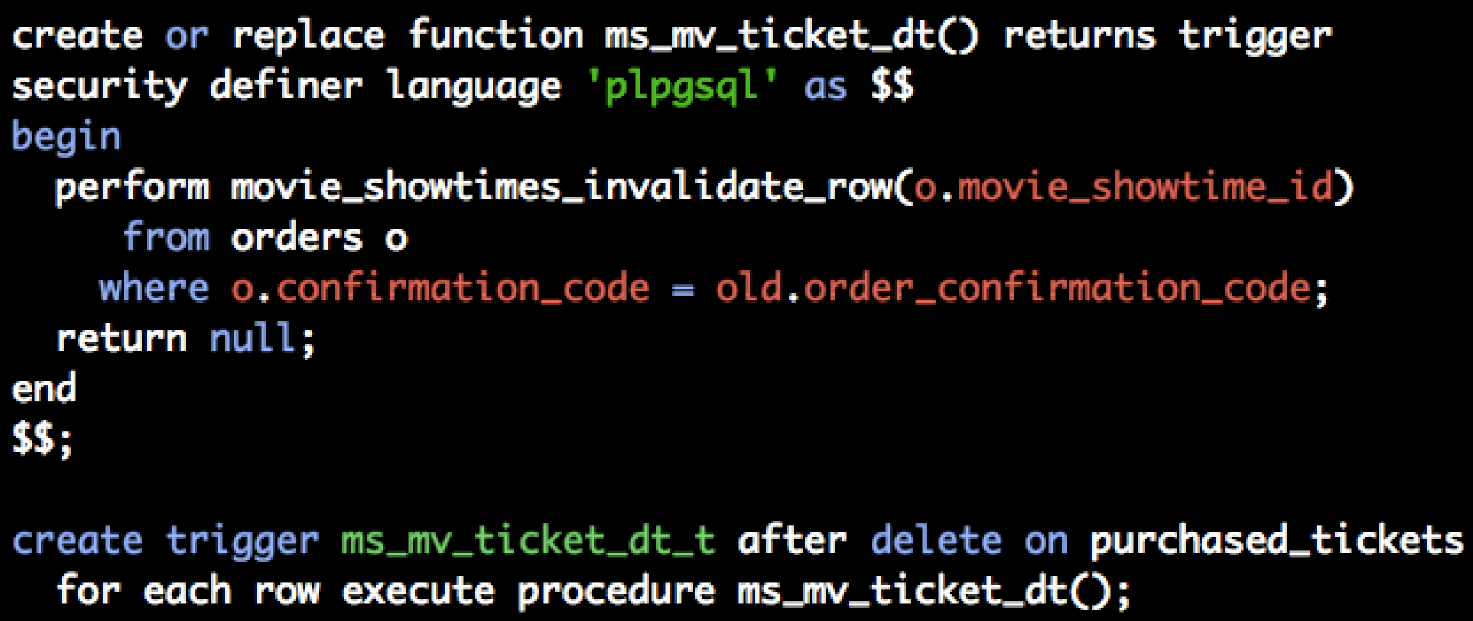
```
create or replace function ms_mv_ticket_it() returns trigger
security definer language 'plpgsql' as $$
begin
  perform movie_showtimes_invalidate_row(o.movie_showtime_id)
     from orders o
    where o.confirmation_code = new.order_confirmation_code;
  return null;
end
$$;

create trigger ms_mv_ticket_it_t after insert on purchased_tickets
  for each row execute procedure ms_mv_ticket_it();
```

table	action	refresh needed?
ticket_purchases	insert	x
	update	x
	delete	x



```
create or replace function ms_mv_ticket_ut() returns trigger
security definer language 'plpgsql' as $$
begin
  if old.order_confirmation_code != new.order_confirmation_code then
    perform movie_showtimes_invalidate_row(o.movie_showtime_id)
       from orders o
      where o.confirmation_code = new.order_confirmation_code;
    perform movie_showtimes_invalidate_row(o.movie_showtime_id)
       from orders o
      where o.confirmation_code = old.order_confirmation_code;
  end if;
  return null;
end
$$;

create trigger ms_mv_ticket_ut_t after update on purchased_tickets
  for each row execute procedure ms_mv_ticket_ut();
```

table	action	refresh needed?
ticket_purchases	insert	x
	update	x
	delete	x



```
create or replace function ms_mv_ticket_dt() returns trigger
security definer language 'plpgsql' as $$
begin
  perform movie_showtimes_invalidate_row(o.movie_showtime_id)
     from orders o
    where o.confirmation_code = old.order_confirmation_code;
  return null;
end
$$;

create trigger ms_mv_ticket_dt_t after delete on purchased_tickets
  for each row execute procedure ms_mv_ticket_dt();
```

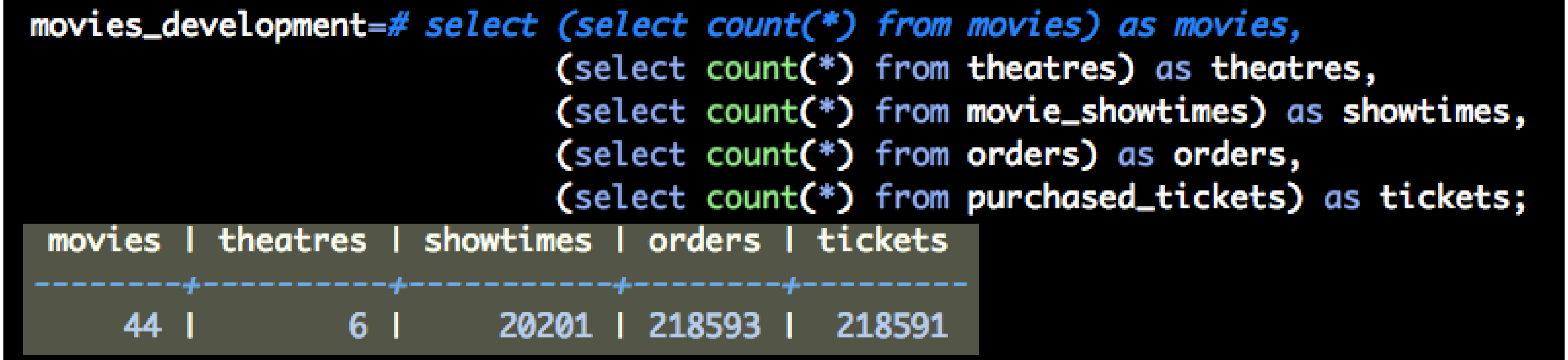
table	action	refresh needed?
auditoriums	insert	
	update	
	delete	

none needed

Performance

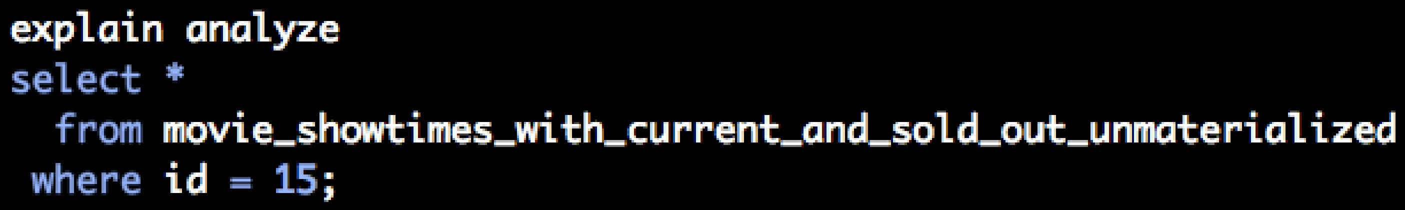
- How good is it?

Sample Data Set



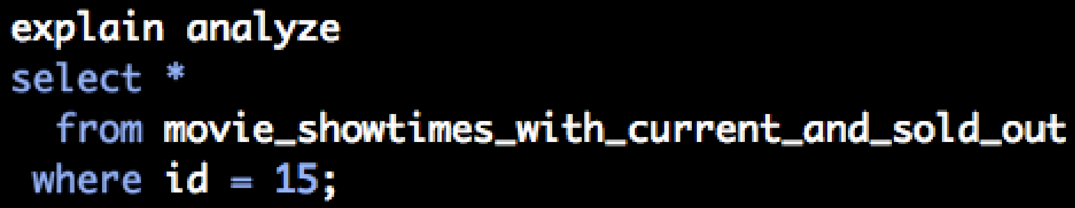
```
movies_development=# select (select count(*) from movies) as movies,
                            (select count(*) from theatres) as theatres,
                            (select count(*) from movie_showtimes) as showtimes,
                            (select count(*) from orders) as orders,
                            (select count(*) from purchased_tickets) as tickets;
 movies | theatres | showtimes | orders | tickets
--------+----------+-----------+--------+---------
     44 |        6 |     20201 | 218593 |  218591
```

by *id*



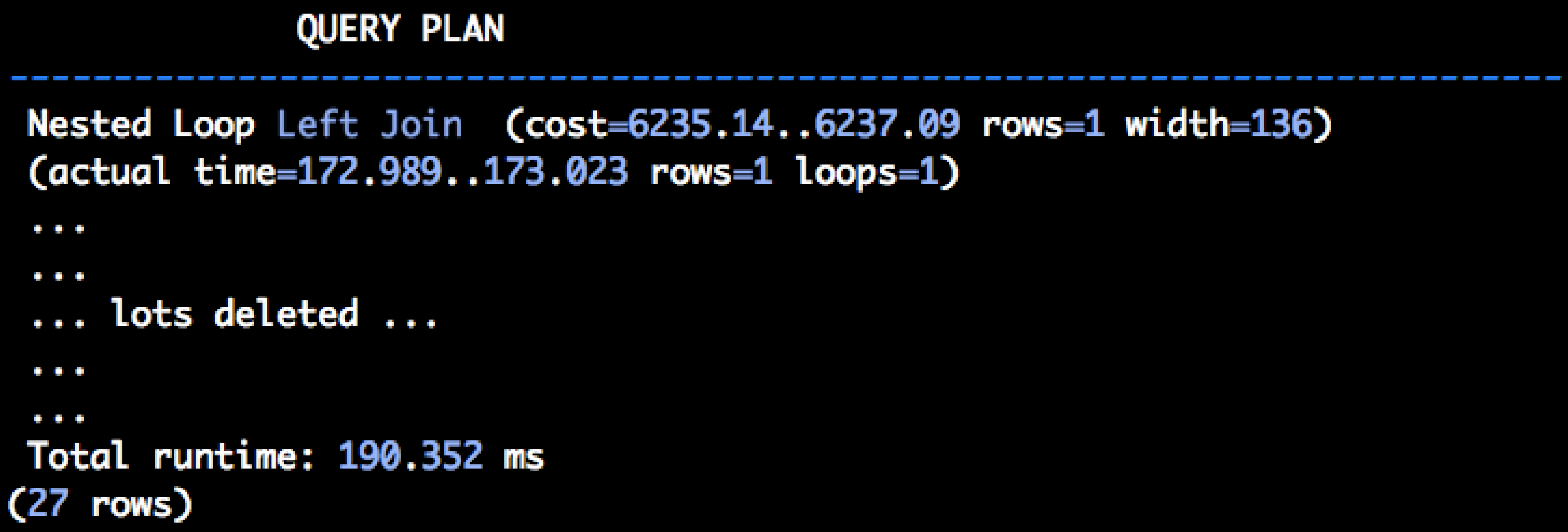
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out_unmaterialized
 where id = 15;
```

versus



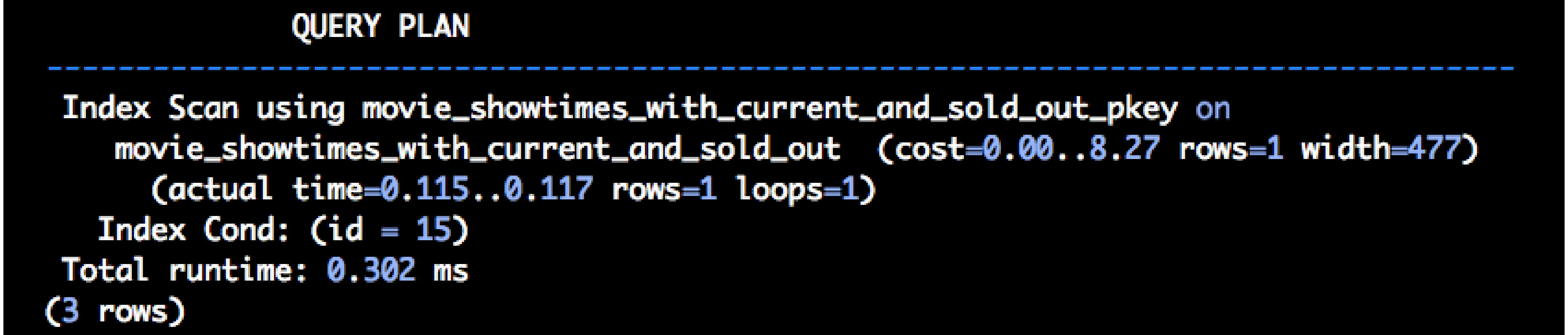
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out
 where id = 15;
```

unmaterialized



```
              QUERY PLAN
------------------------------------------------------------------------------
 Nested Loop Left Join  (cost=6235.14..6237.09 rows=1 width=136)
 (actual time=172.989..173.023 rows=1 loops=1)
 ...
 ...
 ... lots deleted ...
 ...
 ...
 Total runtime: 190.352 ms
(27 rows)
```

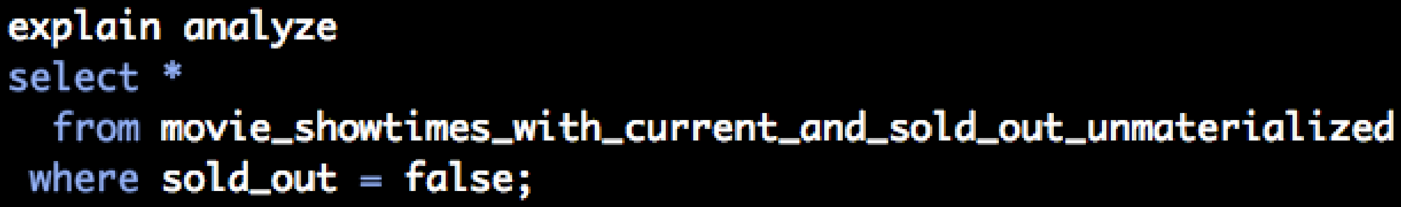
materialized



```
              QUERY PLAN
----------------------------------------------------------------------------------
 Index Scan using movie_showtimes_with_current_and_sold_out_pkey on
    movie_showtimes_with_current_and_sold_out  (cost=0.00..8.27 rows=1 width=477)
      (actual time=0.115..0.117 rows=1 loops=1)
   Index Cond: (id = 15)
 Total runtime: 0.302 ms
(3 rows)
```

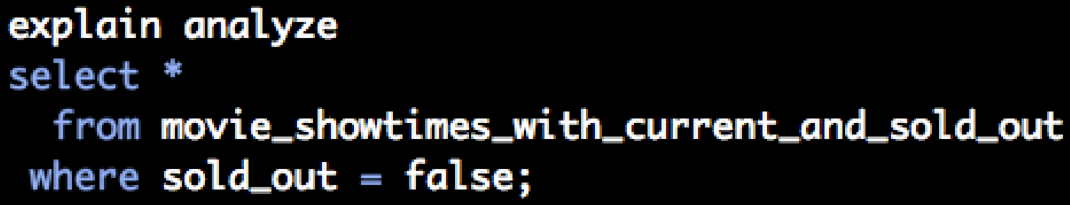
190 / 0.3 = 633 times faster!

by *sold_out*



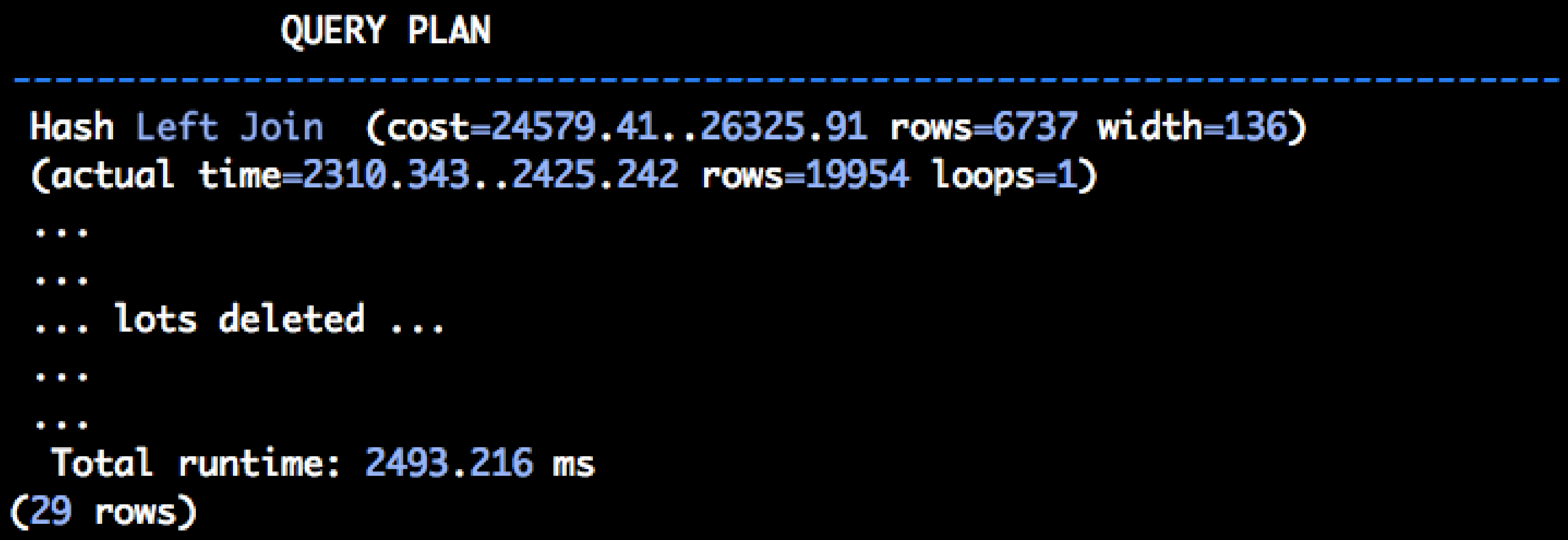
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out_unmaterialized
 where sold_out = false;
```

versus



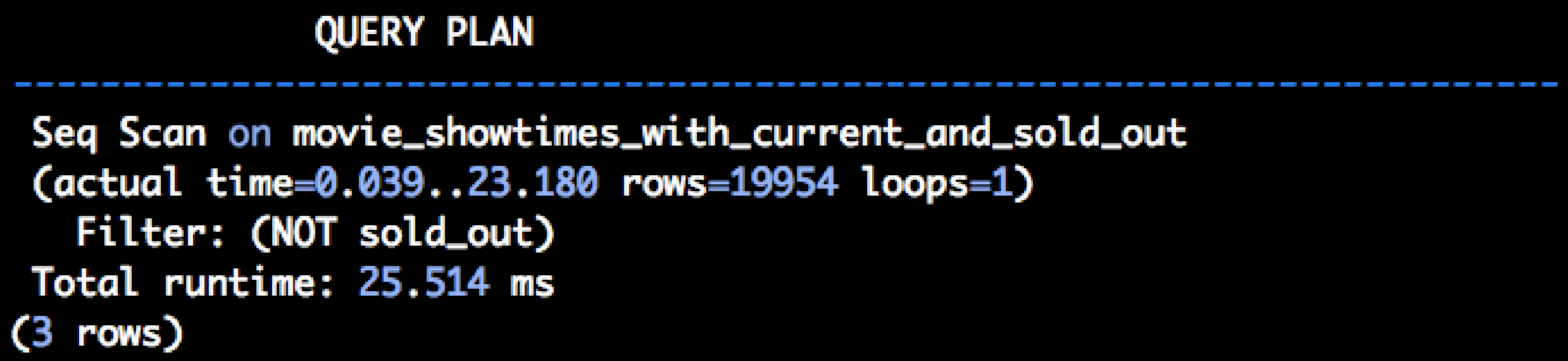
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out
 where sold_out = false;
```

unmaterialized



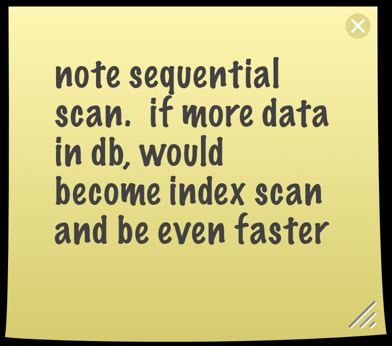
```
             QUERY PLAN
-----------------------------------------------------------------------------
 Hash Left Join  (cost=24579.41..26325.91 rows=6737 width=136)
 (actual time=2310.343..2425.242 rows=19954 loops=1)
 ...
 ...
 ... lots deleted ...
 ...
 ...
  Total runtime: 2493.216 ms
(29 rows)
```

materialized



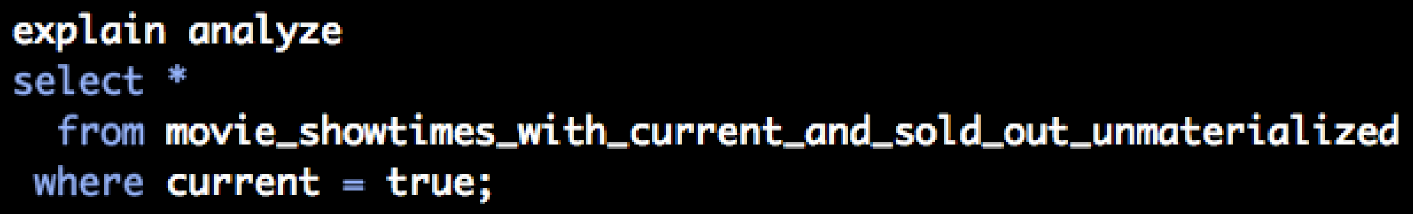
```
              QUERY PLAN
------------------------------------------------------------------------
 Seq Scan on movie_showtimes_with_current_and_sold_out
 (actual time=0.039..23.180 rows=19954 loops=1)
   Filter: (NOT sold_out)
 Total runtime: 25.514 ms
(3 rows)
```

2493 / 25.5 = 98 times faster!



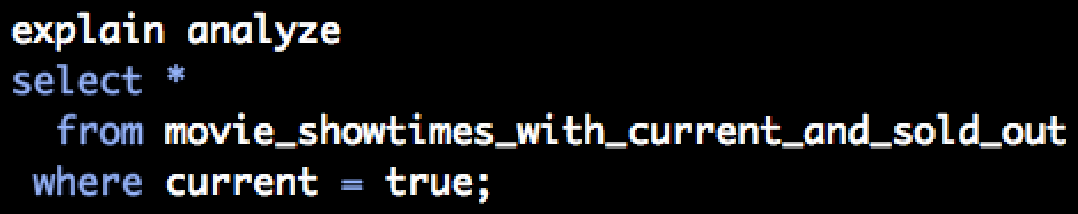
note sequential
scan. if more data
in db, would
become index scan
and be even faster

by *current*



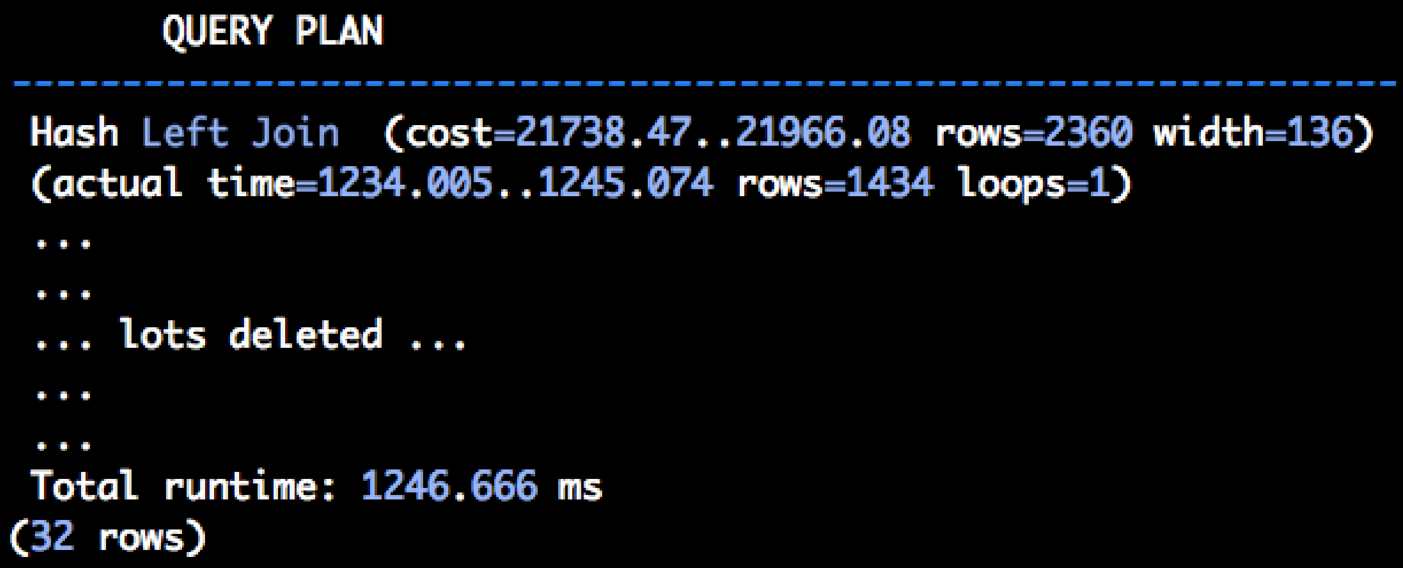
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out_unmaterialized
 where current = true;
```

versus



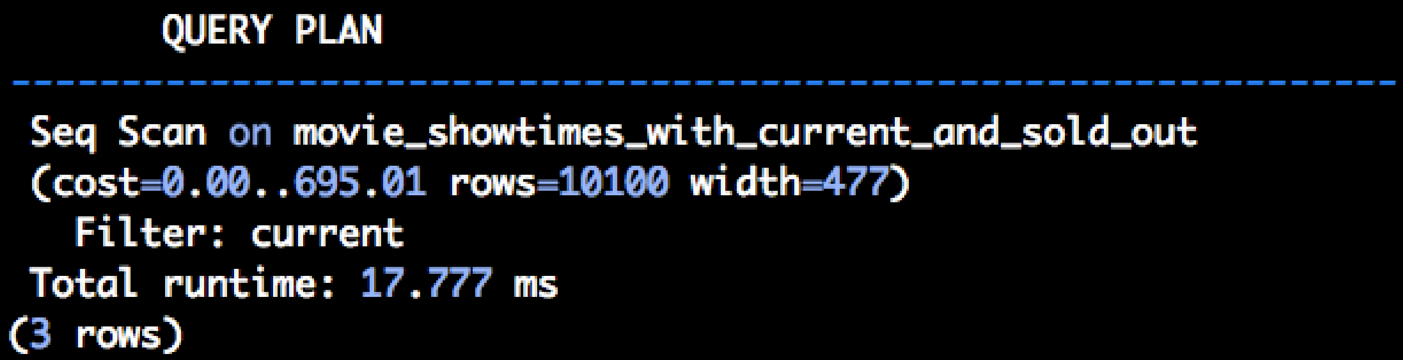
```
explain analyze
select *
  from movie_showtimes_with_current_and_sold_out
 where current = true;
```

unmaterialized



```
       QUERY PLAN
-----------------------------------------------------------------
 Hash Left Join  (cost=21738.47..21966.08 rows=2360 width=136)
 (actual time=1234.005..1245.074 rows=1434 loops=1)
 ...
 ...
 ... lots deleted ...
 ...
 ...
 Total runtime: 1246.666 ms
(32 rows)
```

materialized



```
       QUERY PLAN
-----------------------------------------------------------------
 Seq Scan on movie_showtimes_with_current_and_sold_out
 (cost=0.00..695.01 rows=10100 width=477)
   Filter: current
 Total runtime: 17.777 ms
(3 rows)
```

1246 / 17.7 = 70 times faster!

Performance Roundup

- All rows: ***85x***
- By id: ***633x***
- By filter 1 (sold out): ***98x***
- By filter 2 (current): ***70x***

Not too shabby!

Part IV: Getting Advanced

1. Time dependencies
2. Reconciler view
3. Deferring payment with invalidation
4. Periodic Refreshes
5. Cascading materialized views

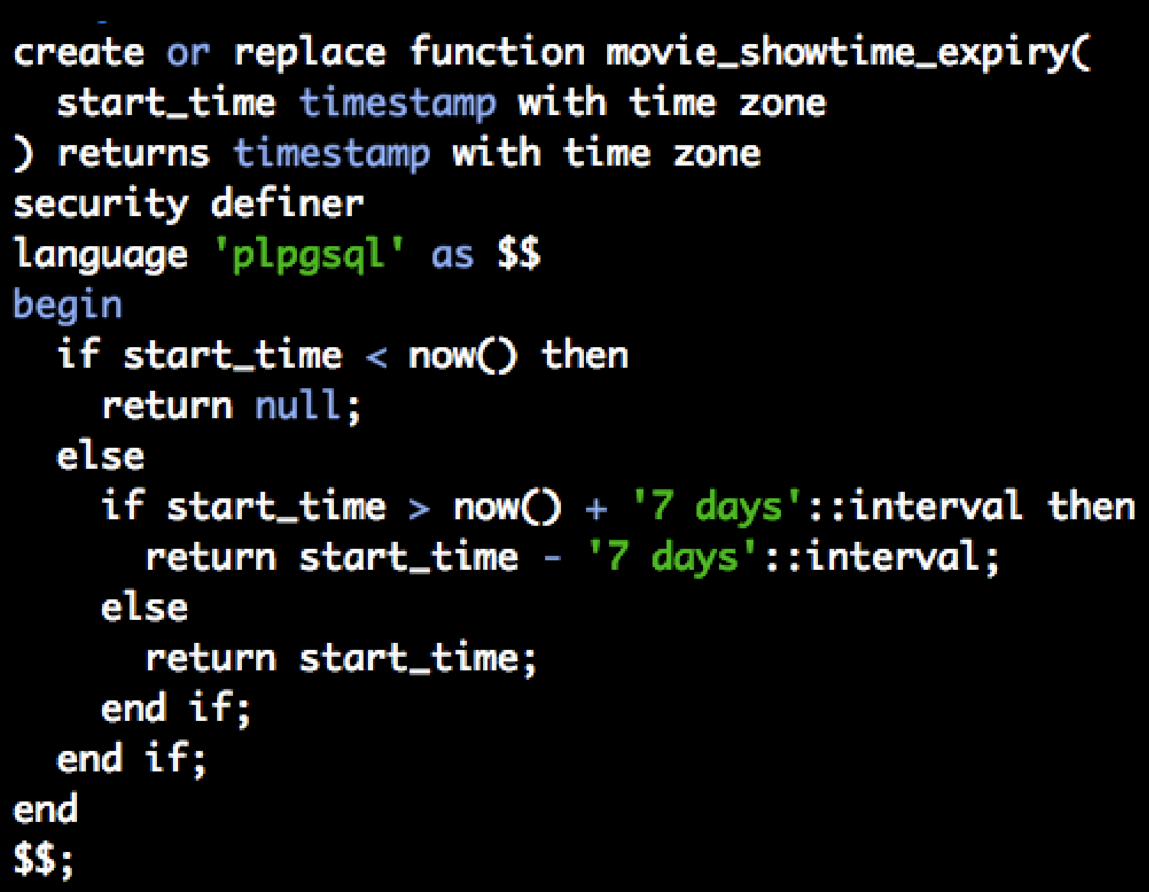
Time Dependency

- mutable functions mess everything up
- most common is time: *e.g., now()*
- no triggerable event, just the march of time

expiry column

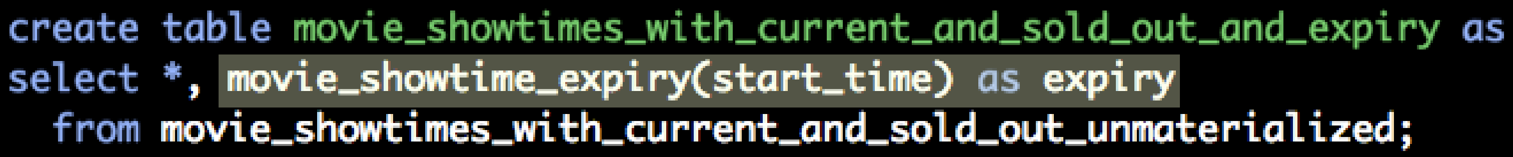
- we know our domain
- we know when a filter will flip polarity
- put expiry time in a new column

expiry function



```
create or replace function movie_showtime_expiry(
  start_time timestamp with time zone
) returns timestamp with time zone
security definer
language 'plpgsql' as $$
begin
  if start_time < now() then
    return null;
  else
    if start_time > now() + '7 days'::interval then
      return start_time - '7 days'::interval;
    else
      return start_time;
    end if;
  end if;
end
$$;
```

new snapshot



```
create table movie_showtimes_with_current_and_sold_out_and_expiry as
select *, movie_showtime_expiry(start_time) as expiry
  from movie_showtimes_with_current_and_sold_out_unmaterialized;
```

new refresh

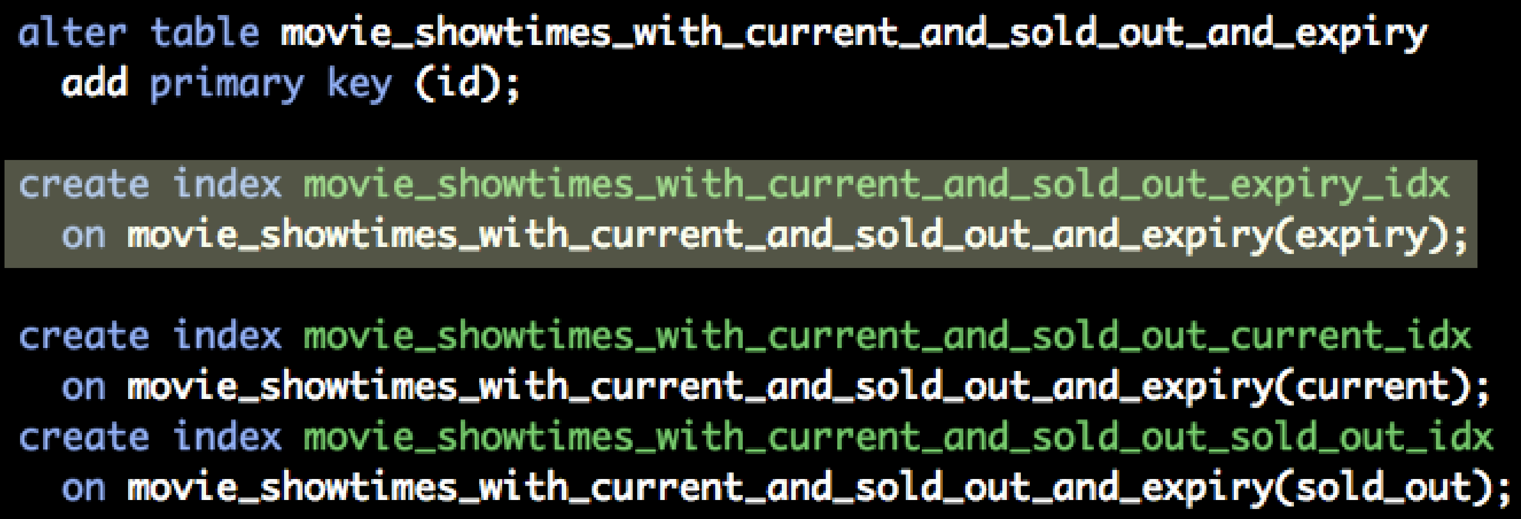
some rigamarole to return expiry.

why do we do this?

we'll see soon in reconciler view.

```
create or replace function movie_showtimes_refresh_row(
  id integer
) returns timestamp with time zone
security definer
language 'plpgsql' as $$
declare
  entry movie_showtimes_with_current_and_sold_out_and_expiry%rowtype;
begin
  delete from movie_showtimes_with_current_and_sold_out_and_expiry ms
   where ms.id = id;
  select into entry
         *, movie_showtime_expiry(ms.start_time)
    from movie_showtimes_with_current_and_sold_out_unmaterialized ms
   where ms.id = id;
  insert into movie_showtimes_with_current_and_sold_out_and_expiry
  values (entry.*);
  return entry.expiry;
end
$$;
```

indexes...

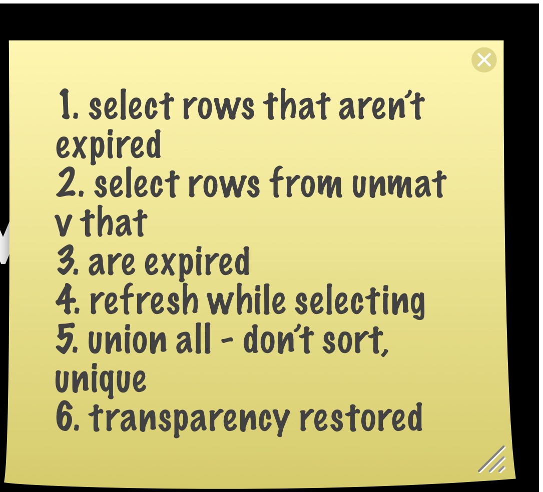


```
alter table movie_showtimes_with_current_and_sold_out_and_expiry
  add primary key (id);

create index movie_showtimes_with_current_and_sold_out_expiry_idx
  on movie_showtimes_with_current_and_sold_out_and_expiry(expiry);

create index movie_showtimes_with_current_and_sold_out_current_idx
  on movie_showtimes_with_current_and_sold_out_and_expiry(current);
create index movie_showtimes_with_current_and_sold_out_sold_out_idx
  on movie_showtimes_with_current_and_sold_out_and_expiry(sold_out);
```

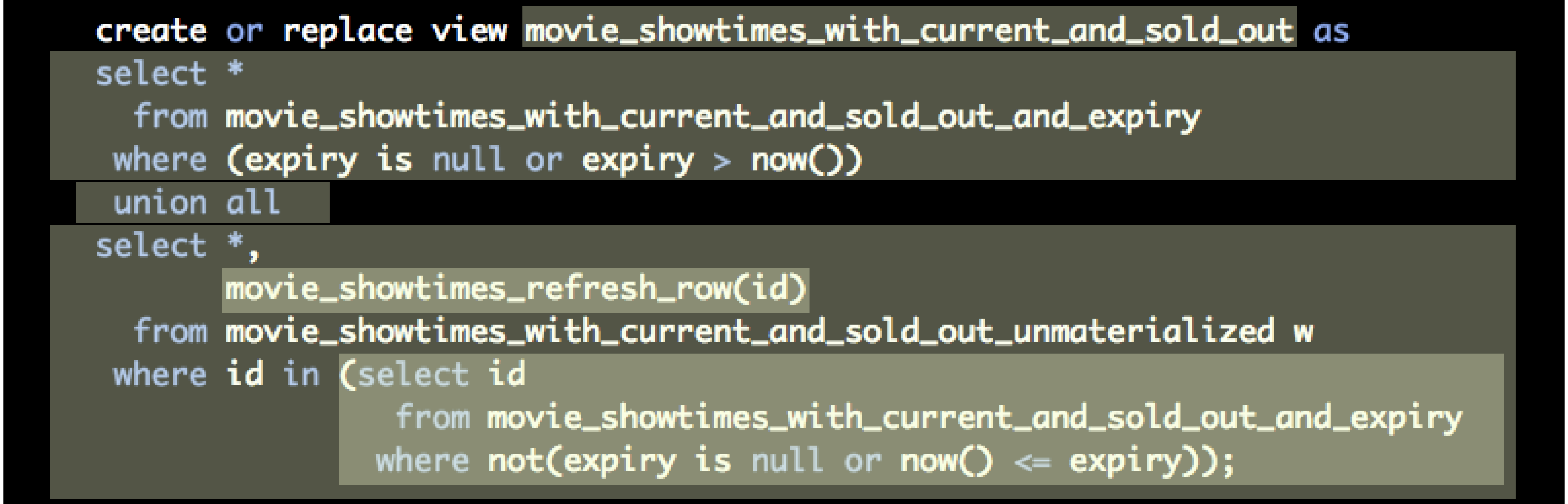
reconciler view

- expiry column exposes implementation
- don’t want clients to filter on it, or know about it

reconciler viev



1. select rows that aren't expired
2. select rows from unmat v that
3. are expired
4. refresh while selecting
5. union all - don't sort, unique
6. transparency restored



```
create or replace view movie_showtimes_with_current_and_sold_out as
select *
  from movie_showtimes_with_current_and_sold_out_and_expiry
 where (expiry is null or expiry > now())
 union all
select *,
       movie_showtimes_refresh_row(id)
  from movie_showtimes_with_current_and_sold_out_unmaterialized w
 where id in (select id
                from movie_showtimes_with_current_and_sold_out_and_expiry
               where not(expiry is null or now() <= expiry));
```

FAQ

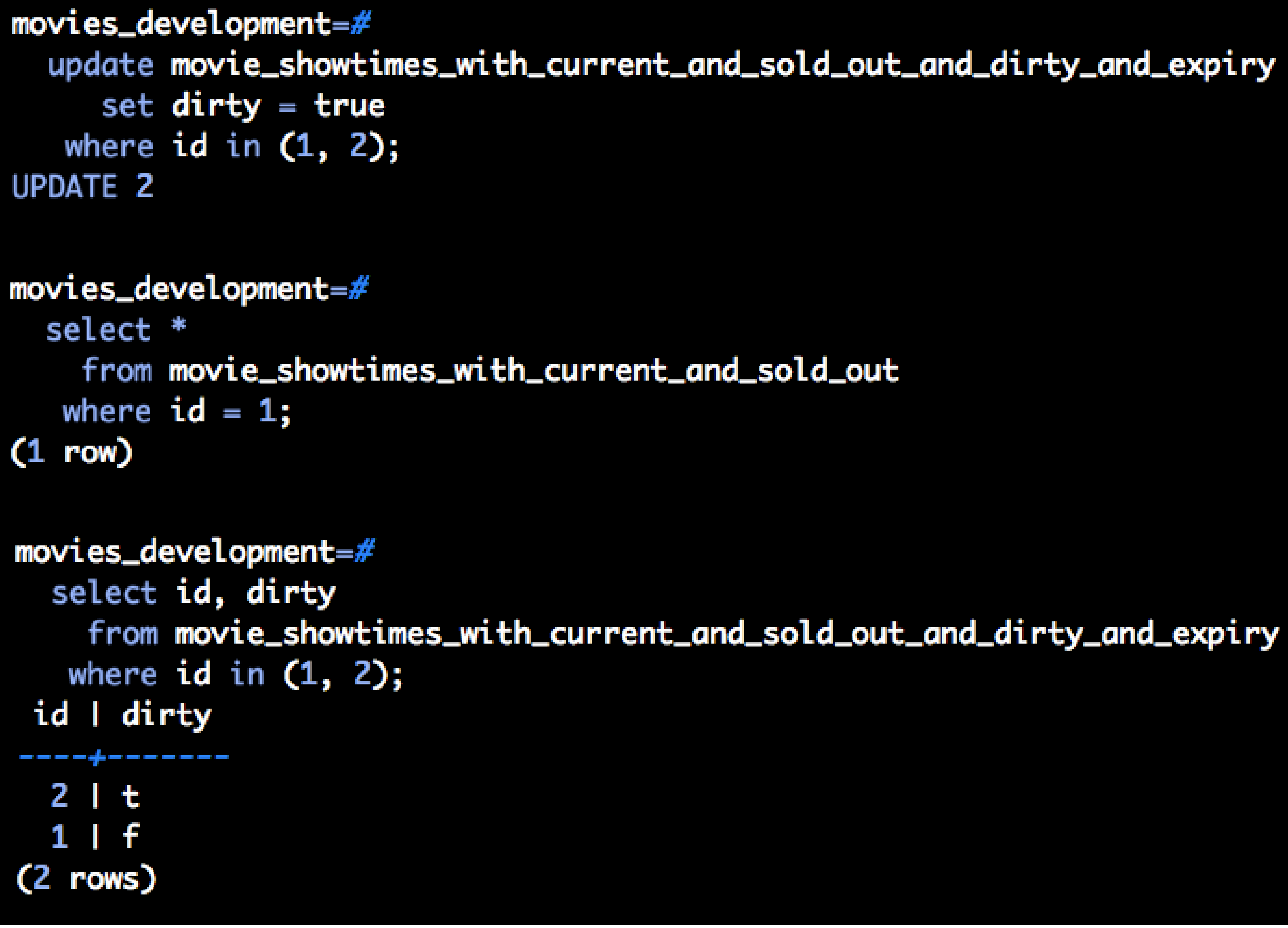
- What happens when filtering on columns that may be invalid?
- Do all expired rows get refreshed, or just those returned by the query?
- Is an outer where clause applied to the unmaterialized or materialized view?
- Is this truly magical?

It didn’t work :(

sometimes, after doing all this work,
selecting from the materialized view is just as slow...

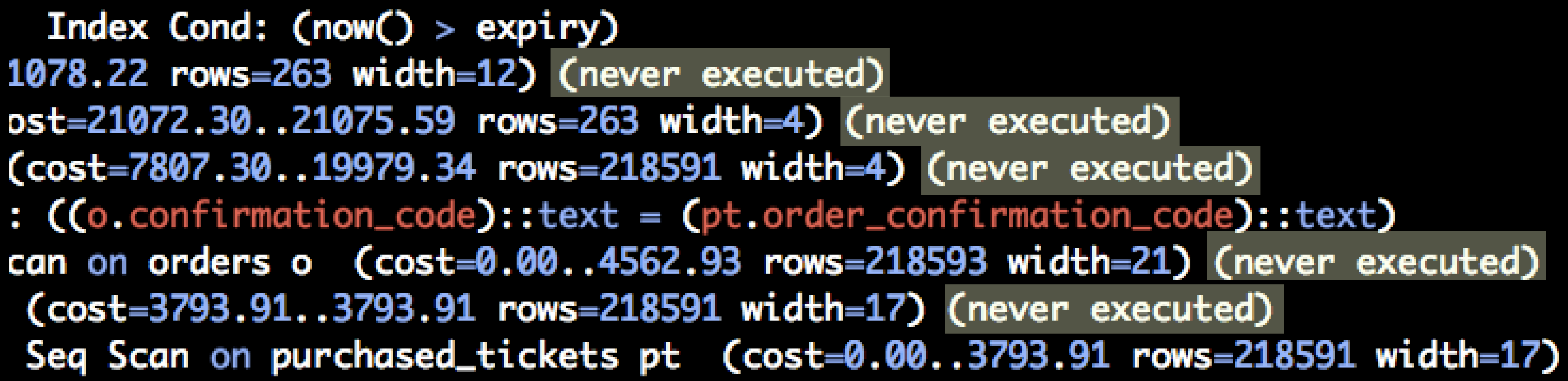
run ‘vacuum analyze’ and try again.

incremental refresh



```
movies_development=#
  update movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
     set dirty = true
   where id in (1, 2);
UPDATE 2

movies_development=#
  select *
    from movie_showtimes_with_current_and_sold_out
   where id = 1;
(1 row)

movies_development=#
  select id, dirty
    from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
   where id in (1, 2);
 id | dirty
----+-------
  2 | t
  1 | f
(2 rows)
```

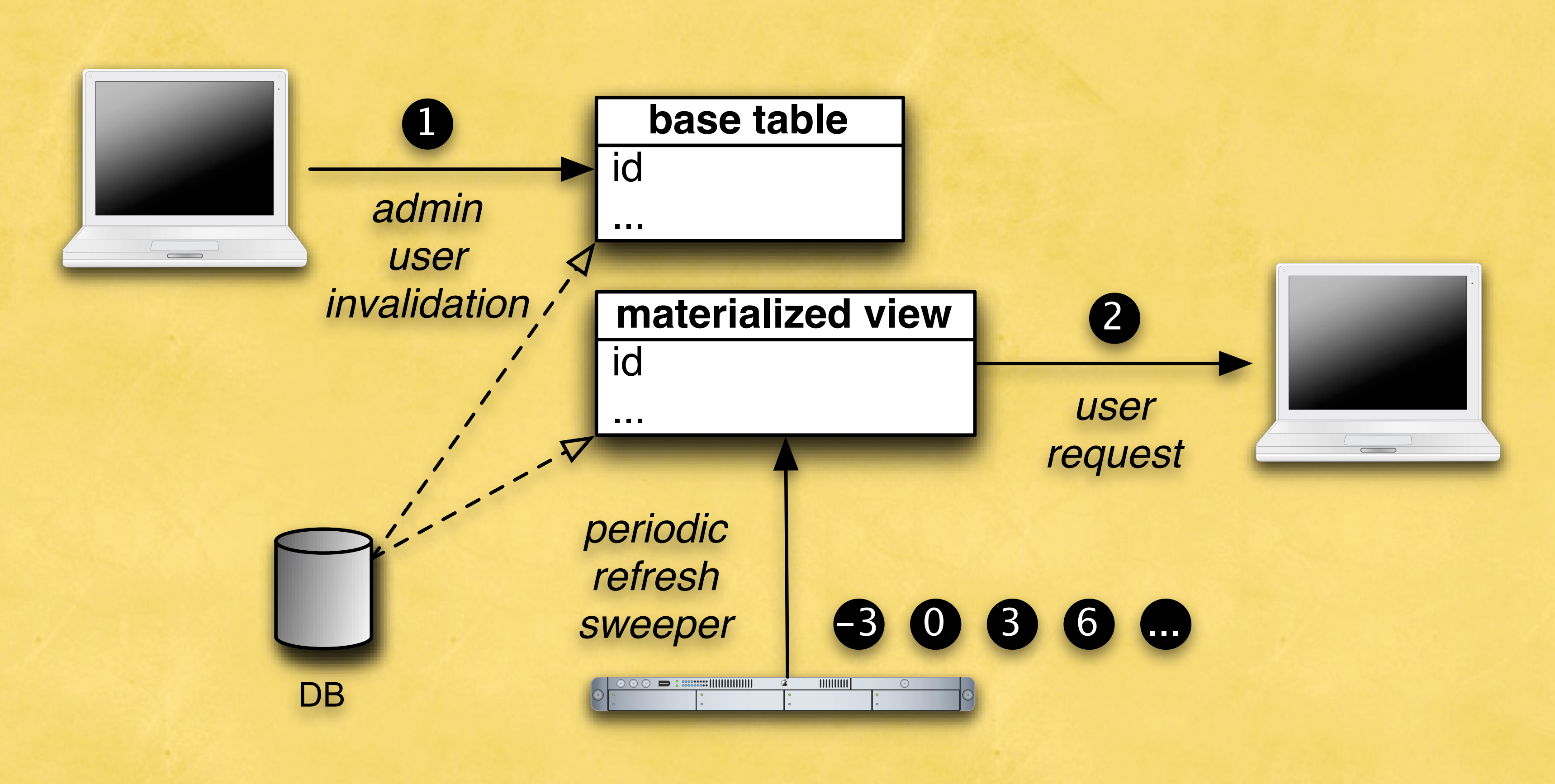
query planner smarts



```
  Index Cond: (now() > expiry)
1078.22 rows=263 width=12) (never executed)
ost=21072.30..21075.59 rows=263 width=4) (never executed)
(cost=7807.30..19979.34 rows=218591 width=4) (never executed)
: ((o.confirmation_code)::text = (pt.order_confirmation_code)::text)
can on orders o  (cost=0.00..4562.93 rows=218593 width=21) (never executed)
 (cost=3793.91..3793.91 rows=218591 width=17) (never executed)
 Seq Scan on purchased_tickets pt  (cost=0.00..3793.91 rows=218591 width=17)
```

Filter conditions on unmaterialized view enable piecemeal refresh.

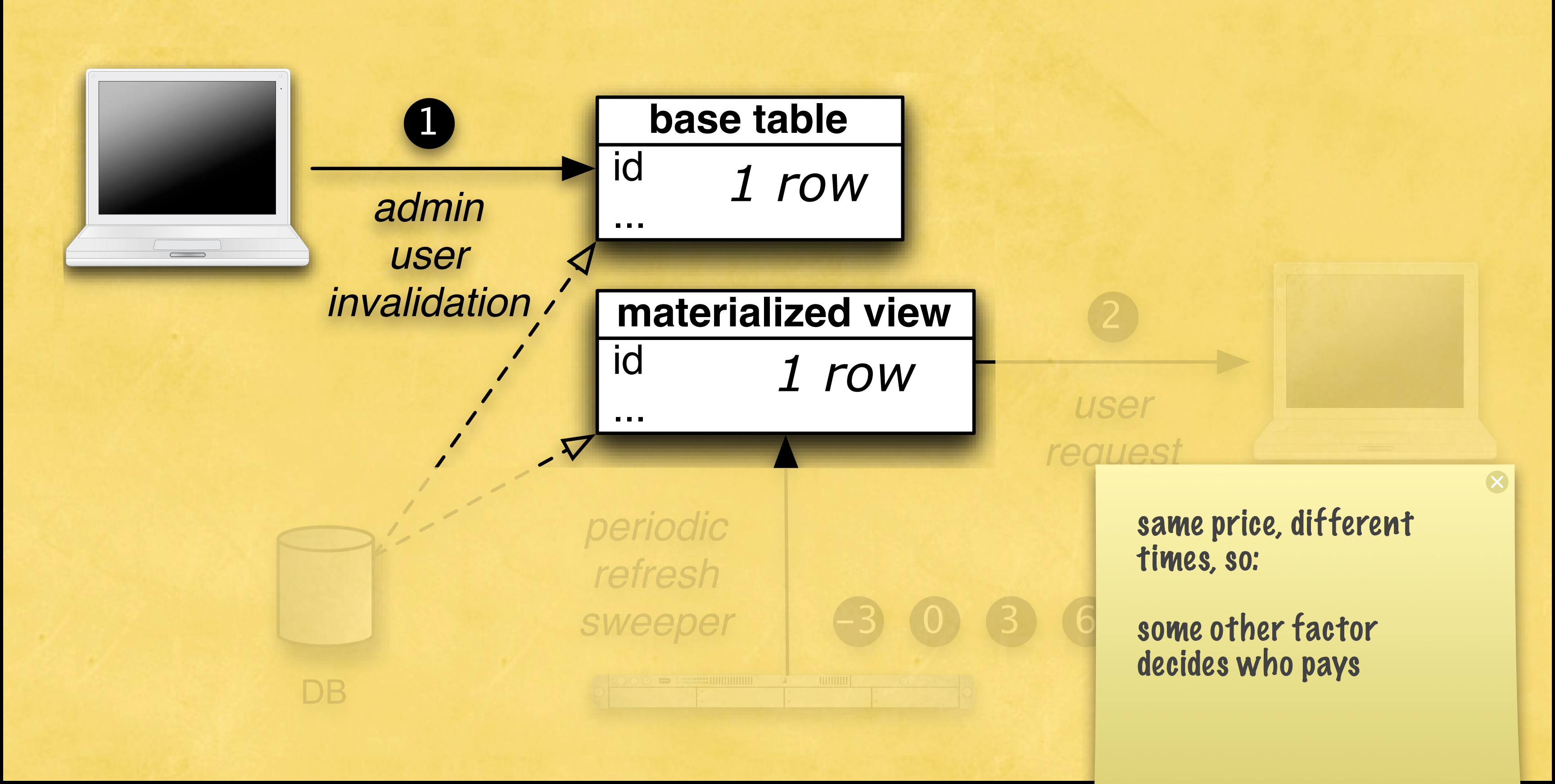
Who Pays?


1
admin
user
invalidation
base table
id
...
materialized view
id
...
2
user
request
periodic
refresh
sweeper
-3
0
3
6
...
DB

It depends

- data relationship
- who are the users?
- invalidation : read proportion

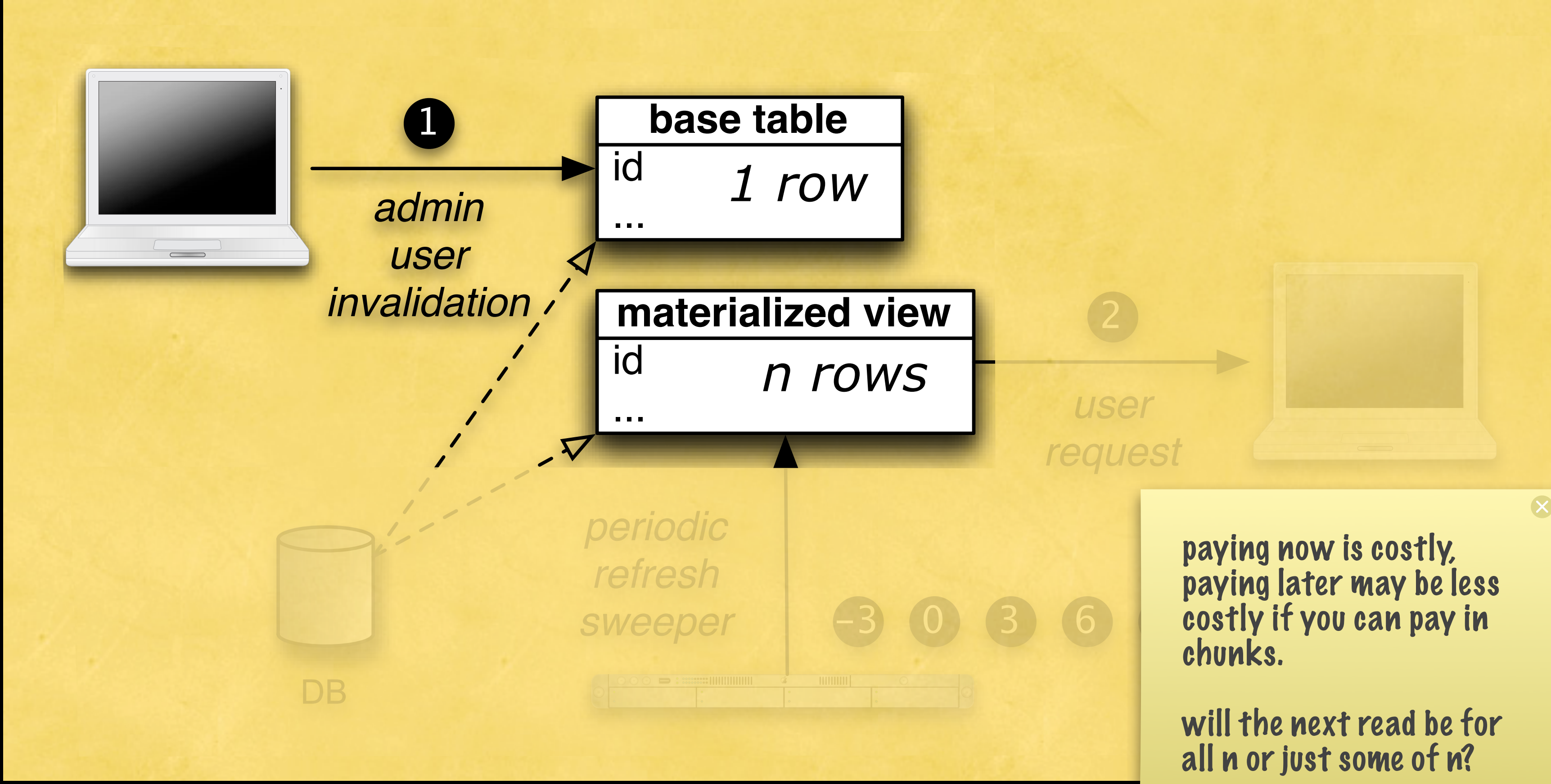
1 : 1 relationship



same price, different times, so:

some other factor decides who pays

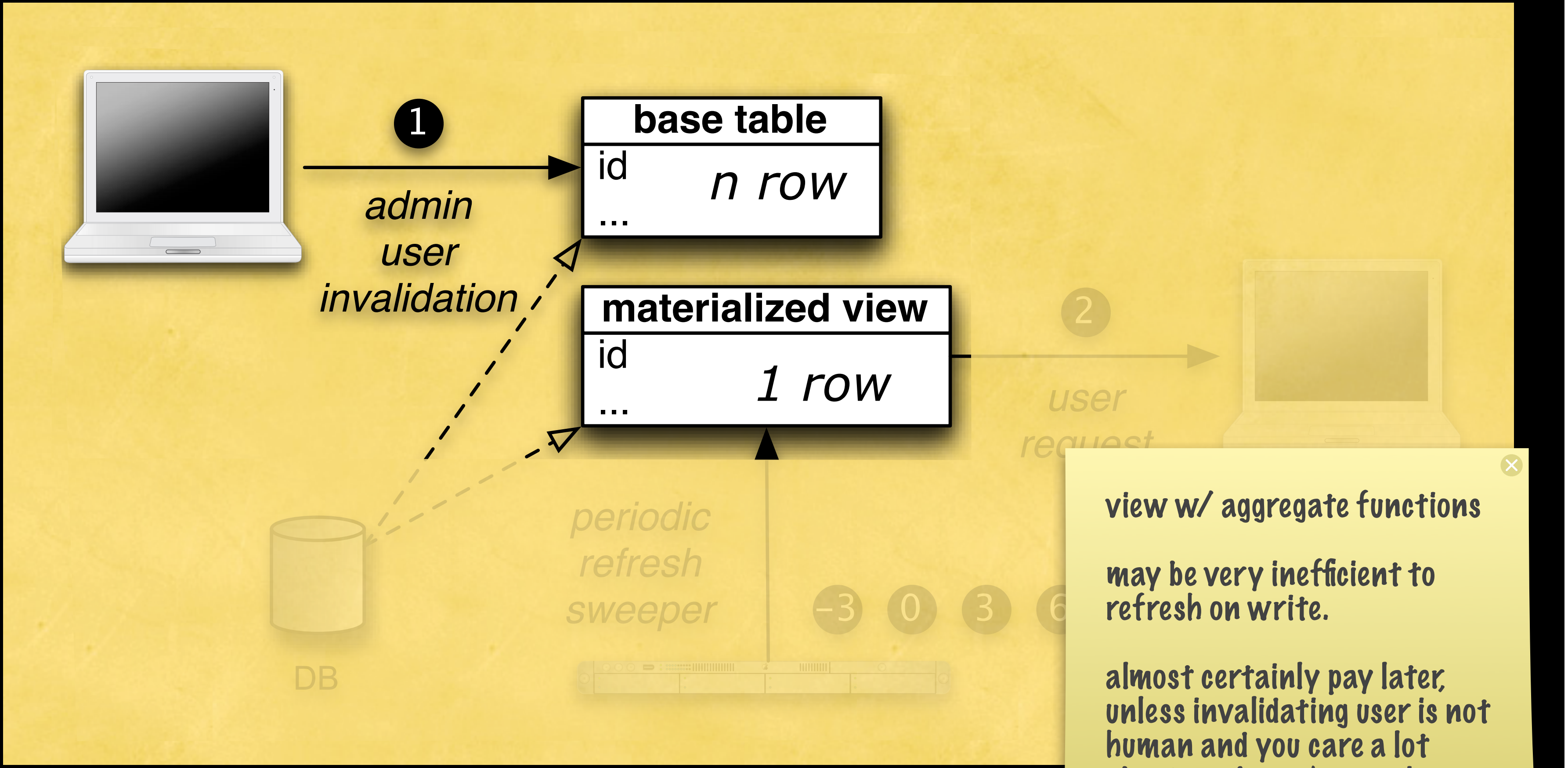
1 : n relationship



paying now is costly, paying later may be less costly if you can pay in chunks.

will the next read be for all n or just some of n?

n : 1 relationship



view w/ aggregate functions

may be very inefficient to refresh on write.

almost certainly pay later, unless invalidating user is not human and you care a lot about read user's experience

invalidating users

- admin users (who you pay)
- visitors (who you pay for)
- ETL in a data warehouse

read users

- admin users (who you pay)
- visitors (who you pay for)
- report generation

in general, people you pay should pay

...unless it's unreasonably costly to database resources.

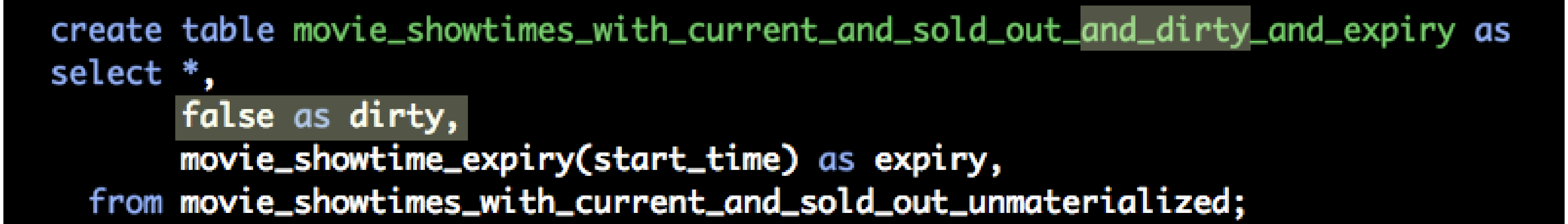
invalidations : reads

- ETL to update/backfill reports:
  will the report ever be read?
- blog entry:
  lots of edits before any reads?
- long tail data:
  widespread invalidation but infrequent reads
  of *most* of it

invalidation

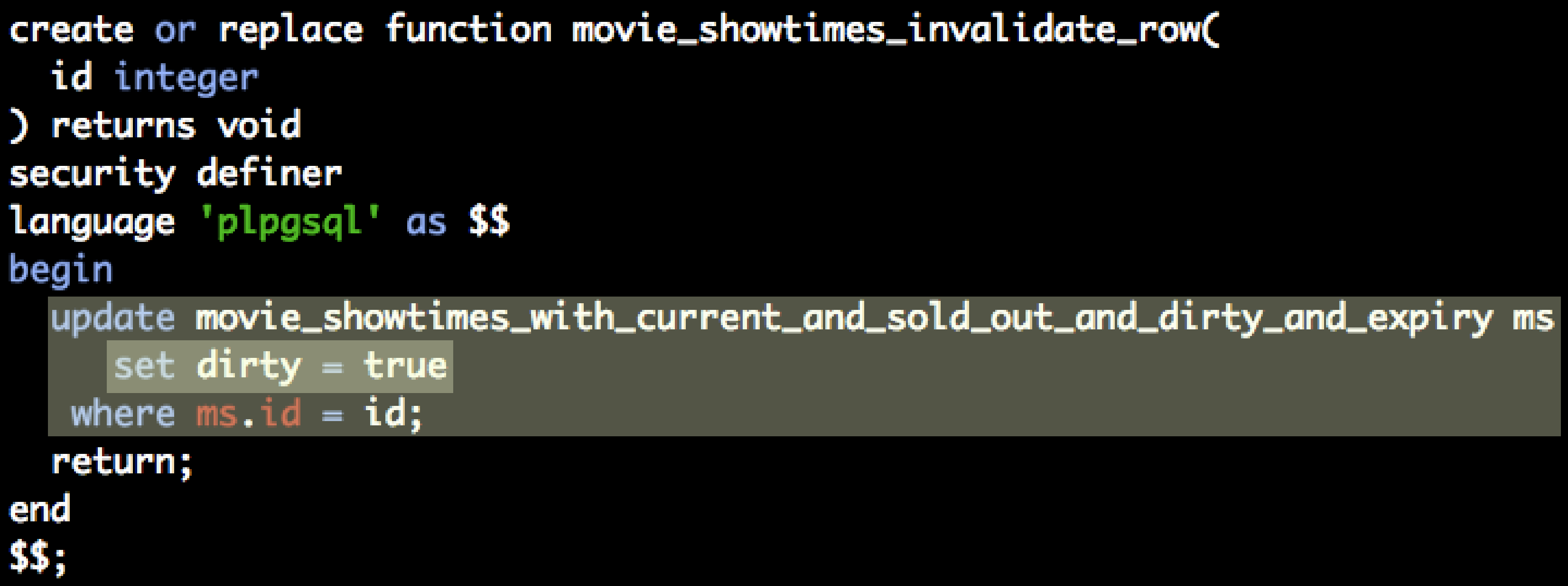
- similar to expiry, add a “dirty” column for bookkeeping
- refresh is O(*f*(*n*)), marking dirty is O(1)
- 1 refresh for N:1 relationships
- “end of transaction”

new snapshot



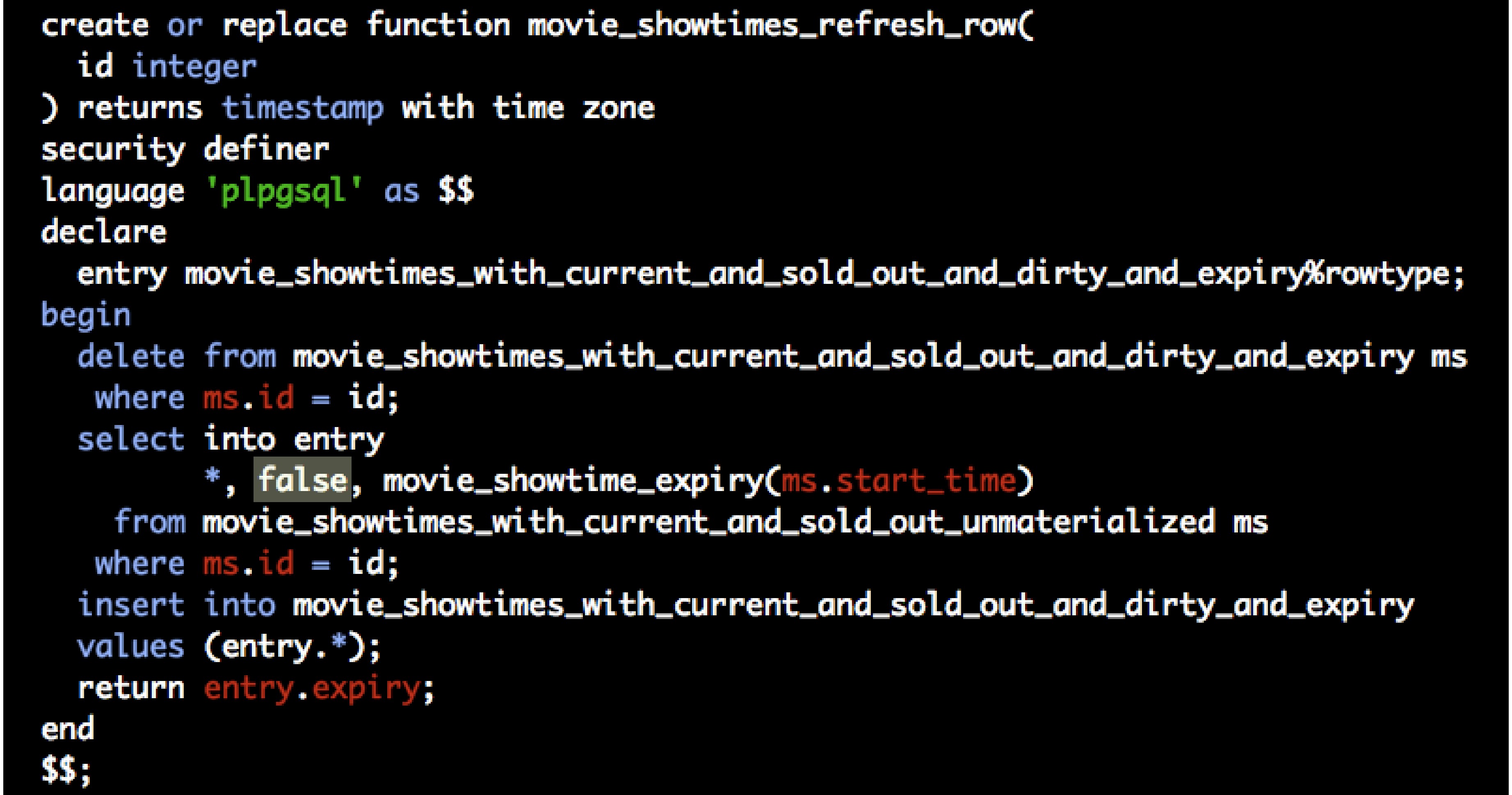
```
create table movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry as
select *,
       false as dirty,
       movie_showtime_expiry(start_time) as expiry,
  from movie_showtimes_with_current_and_sold_out_unmaterialized;
```

invalidation function



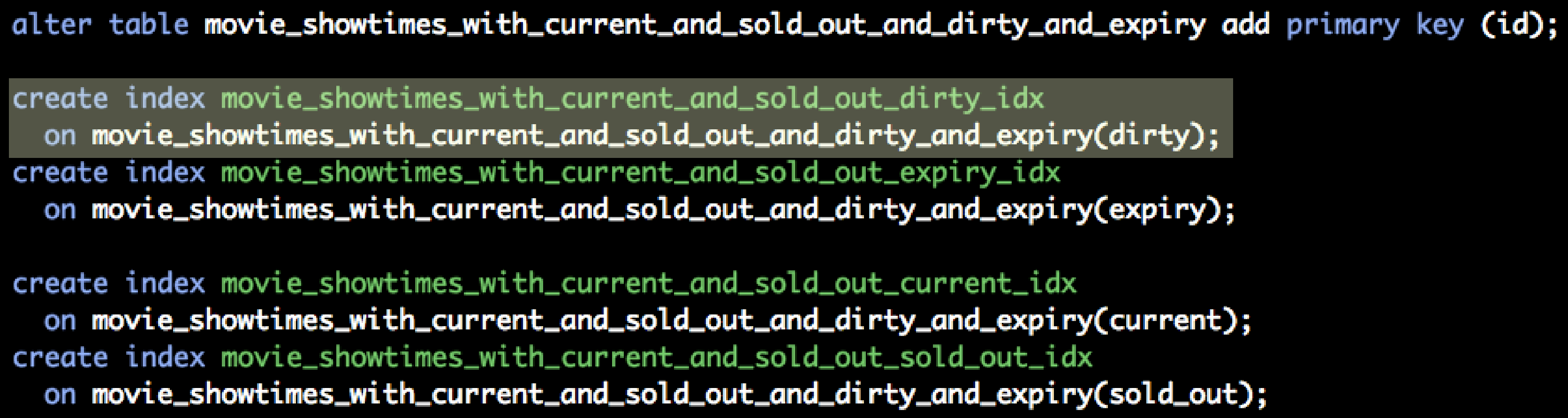
```
create or replace function movie_showtimes_invalidate_row(
  id integer
) returns void
security definer
language 'plpgsql' as $$
begin
  update movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry ms
     set dirty = true
   where ms.id = id;
  return;
end
$$;
```

new refresh function



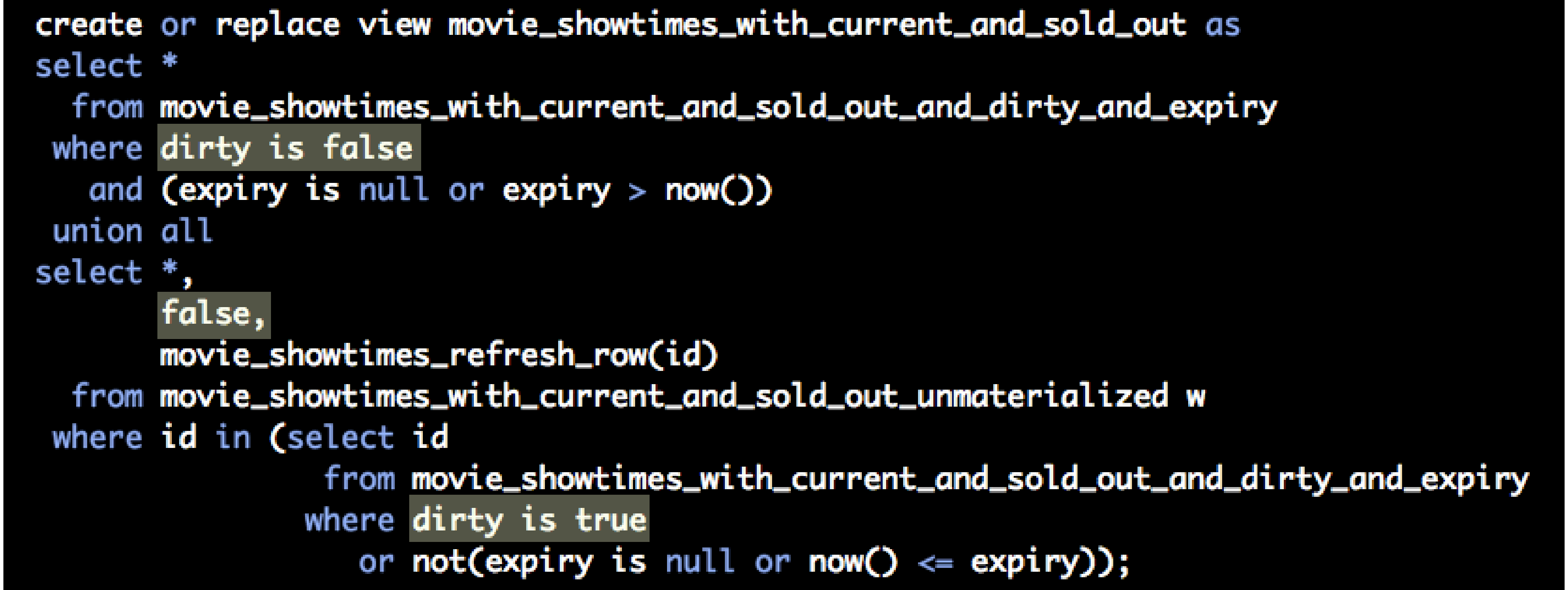
```
create or replace function movie_showtimes_refresh_row(
  id integer
) returns timestamp with time zone
security definer
language 'plpgsql' as $$
declare
  entry movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry%rowtype;
begin
  delete from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry ms
   where ms.id = id;
  select into entry
         *, false, movie_showtime_expiry(ms.start_time)
    from movie_showtimes_with_current_and_sold_out_unmaterialized ms
   where ms.id = id;
  insert into movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
  values (entry.*);
  return entry.expiry;
end
$$;
```

indexes...



```
alter table movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry add primary key (id);

create index movie_showtimes_with_current_and_sold_out_dirty_idx
  on movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry(dirty);
create index movie_showtimes_with_current_and_sold_out_expiry_idx
  on movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry(expiry);

create index movie_showtimes_with_current_and_sold_out_current_idx
  on movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry(current);
create index movie_showtimes_with_current_and_sold_out_sold_out_idx
  on movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry(sold_out);
```

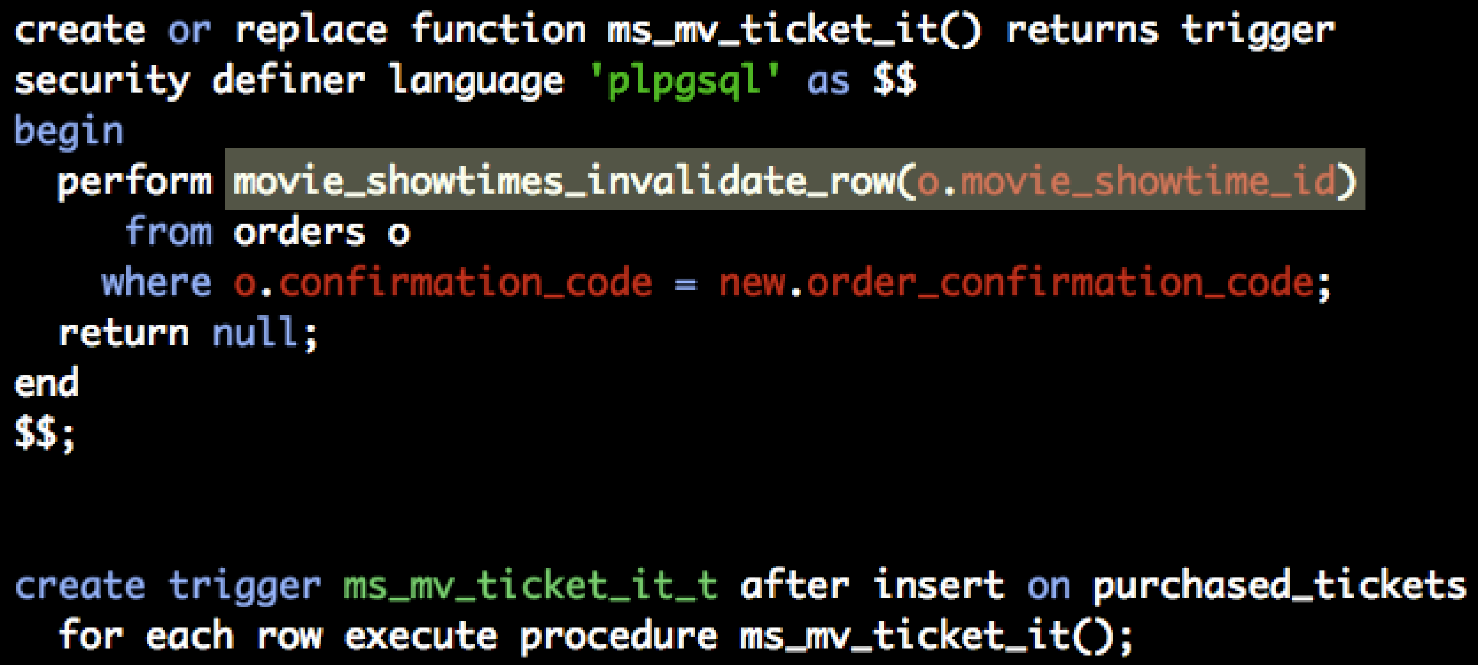
new reconciler view



```
create or replace view movie_showtimes_with_current_and_sold_out as
select *
  from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
 where dirty is false
   and (expiry is null or expiry > now())
 union all
select *,
       false,
       movie_showtimes_refresh_row(id)
  from movie_showtimes_with_current_and_sold_out_unmaterialized w
 where id in (select id
                from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
               where dirty is true
                  or not(expiry is null or now() <= expiry));
```

table	action	refresh needed?	relation	type
movie_showtimes	insert	x	1:1	eager
	update	x		eager
	delete	x		eager
movies	insert		1:N	
	update	x		either
	delete			
theatres	insert		1:N	
	update	x		either
	delete			
orders	insert		1:1	
	update	x		eager
	delete			
ticket_purchases	insert	x	N:1	lazy
	update	x		lazy
	delete	x		lazy
auditoriums	insert		-	
	update			
	delete			

lazy refresh



```
create or replace function ms_mv_ticket_it() returns trigger
security definer language 'plpgsql' as $$
begin
  perform movie_showtimes_invalidate_row(o.movie_showtime_id)
     from orders o
    where o.confirmation_code = new.order_confirmation_code;
  return null;
end
$$;


create trigger ms_mv_ticket_it_t after insert on purchased_tickets
  for each row execute procedure ms_mv_ticket_it();
```

pause, ask if anyone knows why for bullets 1 & 3.

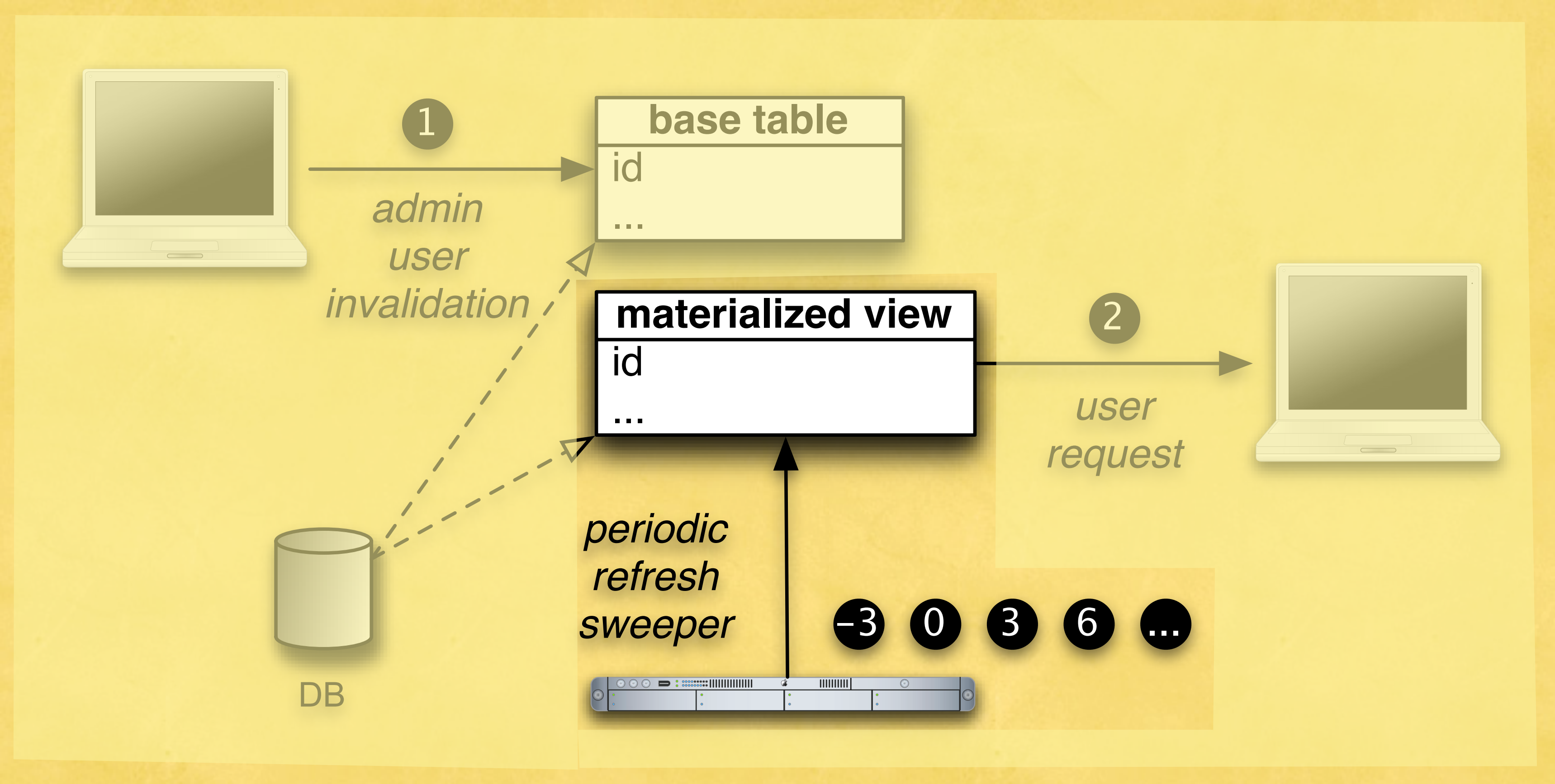
gotcha!

- Lazy via reconciler view will orphan rows that should be deleted.
- Solution: If you know which rows, you can delete them in trigger.
- But if you maintain the abstraction, doesn’t really matter. The orphaned rows will never be returned.

gotcha again!

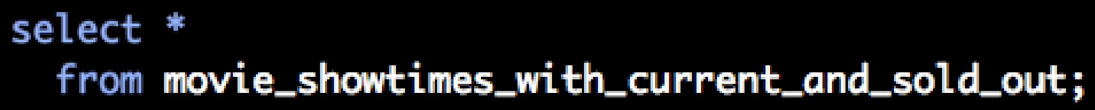
- Lazy does not work for MV inserts, period.
- No row exists yet to mark as dirty.
- Inserts to base tables that do not add rows to the MV are OK.

Periodic Refreshes



periodic refresh

simple: put this in crontab:



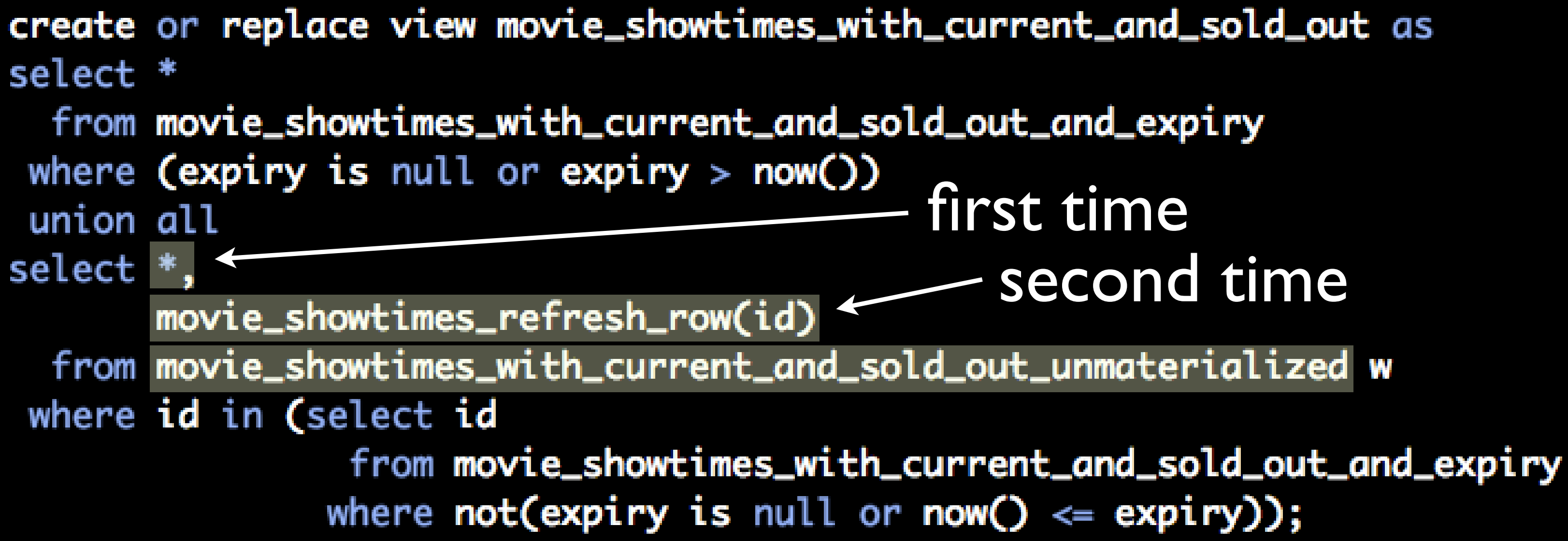
```
select *
  from movie_showtimes_with_current_and_sold_out;
```

note: refreshes one row at a time.
a more efficient refresh function can be built, too.

Other Inefficiencies?

Did you notice a problem with our reconciler view?

It evaluates the unmaterialized view twice.



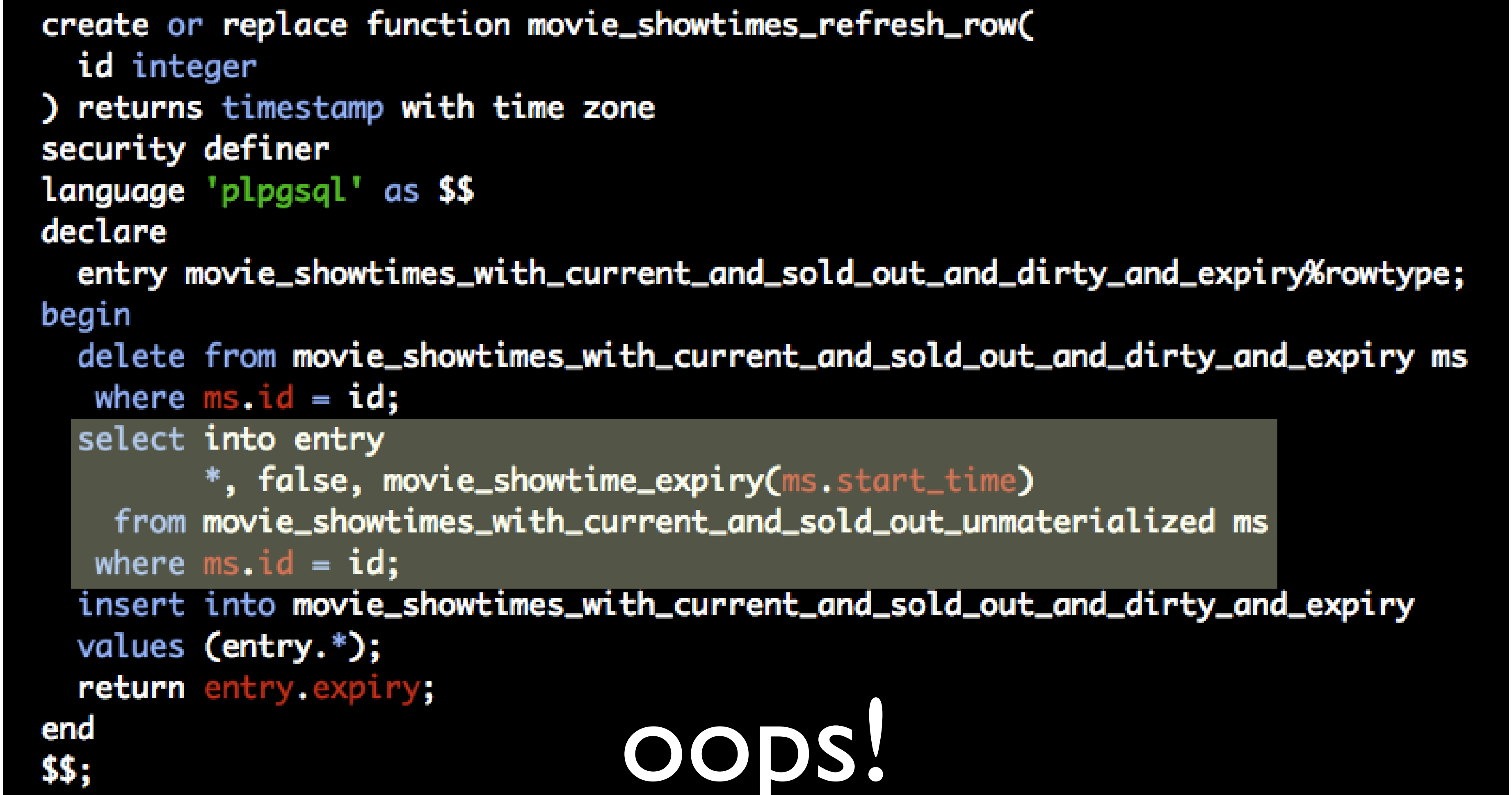
```
create or replace view movie_showtimes_with_current_and_sold_out as
select *
  from movie_showtimes_with_current_and_sold_out_and_expiry
 where (expiry is null or expiry > now())
 union all
select *,
       movie_showtimes_refresh_row(id)
  from movie_showtimes_with_current_and_sold_out_unmaterialized w
 where id in (select id
                from movie_showtimes_with_current_and_sold_out_and_expiry
               where not(expiry is null or now() <= expiry));
```

first time

second time

refresh function does the same lookup, by id

see?



```
create or replace function movie_showtimes_refresh_row(
  id integer
) returns timestamp with time zone
security definer
language 'plpgsql' as $$
declare
  entry movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry%rowtype;
begin
  delete from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry ms
   where ms.id = id;
  select into entry
         *, false, movie_showtime_expiry(ms.start_time)
    from movie_showtimes_with_current_and_sold_out_unmaterialized ms
   where ms.id = id;
  insert into movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
  values (entry.*);
  return entry.expiry;
end
$$;
```

oops!

- We returned expiry to facilitate “magic”
- Can we return the entire row?
- Default PG: “No.” Need to define our own return type.
- Or better, accept whole row, and insert it.
- Either way is challenging to avoid double work.

Cascading

- avoid cascading invalidations multiple times: update .. where dirty is false and not expired

Other tricks

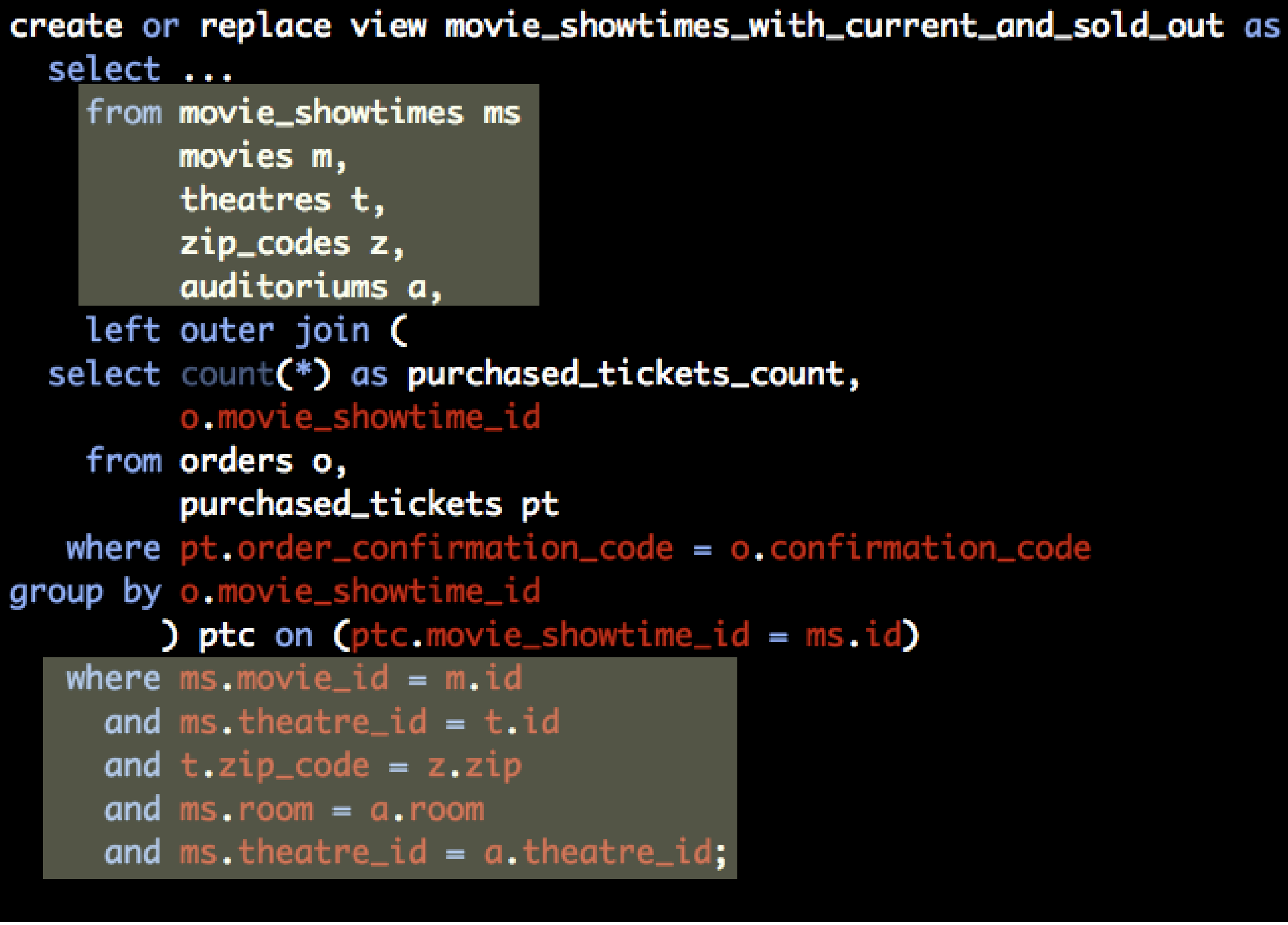
(esp for ETL and memoizing)

- create empty snapshot table, dirty = true
- report queries will fill materialized view up incrementally

Explicit Joins

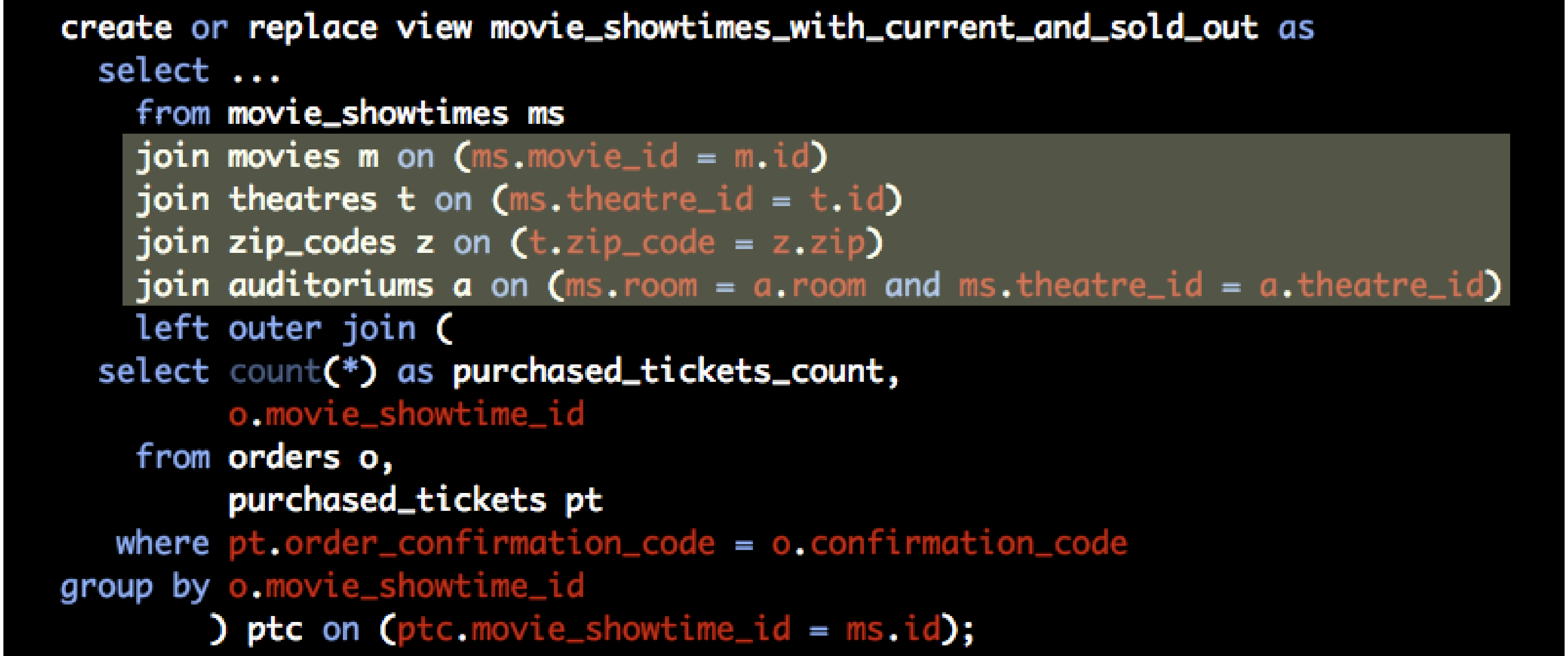
- Unmaterialized view should still be as fast as possible
- Become one with the query planner

implicit



```
create or replace view movie_showtimes_with_current_and_sold_out as
  select ...
    from movie_showtimes ms
         movies m,
         theatres t,
         zip_codes z,
         auditoriums a,
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id)
   where ms.movie_id = m.id
     and ms.theatre_id = t.id
     and t.zip_code = z.zip
     and ms.room = a.room
     and ms.theatre_id = a.theatre_id;
```

explicit



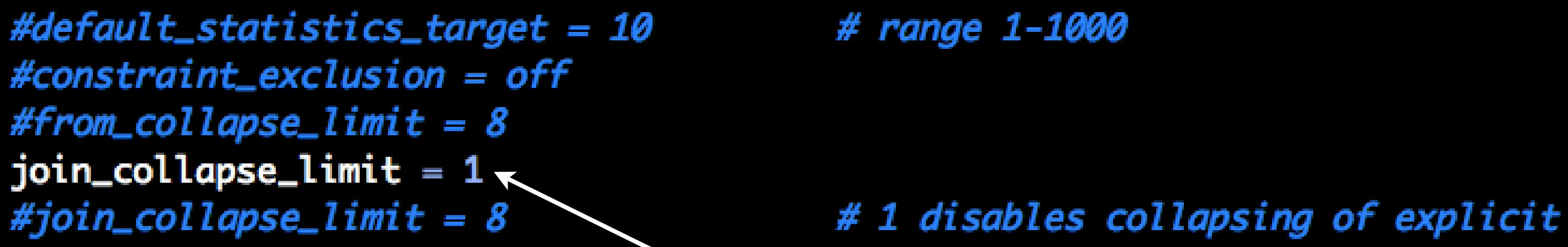
```
create or replace view movie_showtimes_with_current_and_sold_out as
  select ...
    from movie_showtimes ms
    join movies m on (ms.movie_id = m.id)
    join theatres t on (ms.theatre_id = t.id)
    join zip_codes z on (t.zip_code = z.zip)
    join auditoriums a on (ms.room = a.room and ms.theatre_id = a.theatre_id)
    left outer join (
  select count(*) as purchased_tickets_count,
         o.movie_showtime_id
    from orders o,
         purchased_tickets pt
   where pt.order_confirmation_code = o.confirmation_code
group by o.movie_showtime_id
        ) ptc on (ptc.movie_showtime_id = ms.id);
```

what’s the difference?

nothing

unless...

postgresql.conf



```
#default_statistics_target = 10         # range 1-1000
#constraint_exclusion = off
#from_collapse_limit = 8
join_collapse_limit = 1
#join_collapse_limit = 8                # 1 disables collapsing of explicit
```

training wheels off

training wheels on

Repeatable Process

1. What can be generated?
2. What's can't?

would be great to not have to do all of this work each time.

can generate:

- refresh function
- invalidation function
- control table:
  need action?
  which action?
- trigger definitions (but not functions)
- reconciler view

What’s not?

- expiry function - domain specific
- trigger functions - require domain knowledge to be efficient

But these could be stubbed out to make things easy.

questions?