Enterprise Rails

by Dan Chak

O'REILLY®

Enterprise Rails

by Dan Chak

Table of Contents

O'REILLY

14

Materialized Views

When you aren't caching anything, every page load incurs the penalty of the queries required to make up that page. Initially, when you don't have much data, and you don't have many users requesting pages, your application will be snappy. Unfortunately, with any amount of success, you eventually get hit with three problems seemingly all at once:

1. Your application becomes popular and the traffic you need to handle has grown by orders of magnitude.
2. As you sign new customers, gather data, and even simply exist, the amount of data in your database grows by orders of magnitude.
3. Your application grows in complexity and more queries are required to render any given page.

Although most people would be envious of these problems (and the business side of your company would term them "successes"), you nonetheless have to deal with them.

Caching – the act of saving some queried or calculated result for future use – is not as simple and clear-cut as it sounds. A number of subtle issues surround correct caching, which go beyond picking a cache key and storing data in the cache behind that key.

The first issue is of freshness. Can your cache lag behind the true values of your data, or does it need to reflect the latest values?

Next is correctness. If your goal is to keep the cache up to date, have you accounted for every situation where your cache needs to be invalidated or rebuilt? How do you know you've hit all of these cases?

Finally is the cost amortization of keeping the cache accurate. The purpose of caching is to reduce database load and to speed up requests, but someone still must has pay the price for cache updates. Either the requestor who invalidates the cache, or the next person to request the invalidated items will pay part or all of the cost. Choosing the wrong strategy for rebuilding can erase all of the gains that caching was meant to achieve in the first place.

In Chapter 13, we saw that a database view can be thought of as a named query. Even though a complex query can be hidden behind a simple view name, whenever you select from that view, you pay the price of database joins, subselects, filters, and functions that may be required to calculate the view results. A *materialized view* is a cached representation of a database view, stored in a regular table. Rather than query from the view with arbitrary complexity, with view materialization, an indexed table can be queried instead, with O(1) response time.

Chapter 21 contains an overview of caching at all layers of the application. In this chapter, we'll look in depth at caching at a single layer where we have already gained some experience: the database. The principles we will encounter at this layer are the same as those present at other layers of the application, but the database is the layer with the best tools for guaranteeing cache correctness. It is also the most mature and stable layer, so what you learn in this chapter can be applied to other caching problems for years to come, even as feature sets and APIs change for application layer caching solutions.

One way to look at database view materialization is that it is like "wax on, wax off" in *Karate Kid*. It can appear painful and tedious, but when you are ready for your ultimate caching battle, where the tools may not be as thorough and you need to rely on your wits, having a thorough background in caching via wax on, wax off practice will help you identify what elements may be missing from other caching at other layers, so you can be nimble as you come up with your own solutions. As you read this chapter, think beyond the database layer and identify analogs at the application layer to each situation, problem, and technique described.

Before proceeding, I'd like to give credit where credit is due, and pay tribute to Jonathan Gardner, who laid the groundwork for many who tread these waters in his online article, *Materialized Views in PostgreSQL*.

Materialized View Principles

A materialized view is a *cache-complete* copy of your view. This means that every record in the original view appears in the materialized view. This is unlike an LRU cache, where items may expire from the cache if they are not used frequently, of it the set of data being cached exceeds the memory set aside for caching. In a cache-complete implementation, if an item is not in the cache, the application can assume it does not exist. Sometimes these caches are also referred to as *write-through caches*, meaning that you always update the cached copy when you update data in its primary location.

In this chapter, we will build a cache-complete materialized view for the `current_movie_showtimes` developed in the previous chapter. To create a materialized view, we put together a number of building blocks, which will be described in detail throughout this chapter.

The first building block is an initial view to be materialized, which ideally abides by some guidelines that ease materialization. We'll go through some slight modifications to our original view to get it into proper form before we begin.

Next is a target table in which we'll store the cached copy of the view. Unlike a view, which acts like a table, this is a fully fledged physical table, which means we can take advantage of indexing and other features only available with physical tables. In the game of performance enhancements, materializing the view is a big win in and of itself, but adding appropriate indexes hits a home run with the bases loaded.

After we have an initial snapshot in our target table, we'll need a *refresh function* that can update a single record in the materialized view when we detect a change in the base view. Sometimes we don't want to refresh right away – we instead want to put the compute cycles needed to refresh the cache off for the future – and for these cases we'll create an *invalidation function* that marks a record as stale, but doesn't actually do the work of updating it.

We'll detect changes to the view by adding *triggers* to the base tables that make up the view. These triggers will – as the name implies – trigger either a refresh or an invalidation of the rows in our materialized view that are about to become out of sync.

Finally, we'll add some auxiliary views, including the *reconciler view*, on top of our target table to hide the fact that we've materialized the view at all. In addition to hiding our implementation from end-users, the reconciler view will ensure that accurate information is always returned, even if parts of the target table have gone stale or are marked invalid.

A view to materialize

First, we need something worth materializing. We'll start with our view from the previous chapter, but we'll make it a bit more complex so that we can explore a variety of caching techniques. Example 1 shows an extended version of our view, which incorporates the number of seats available in a theatre as `seats_available`, and the number of tickets purchased thus far as `tickets_purchased`. Since the purpose of this view is to show movie showtimes for which we can sell tickets, a filter has been added to the `where` clause to filter out showtimes that are sold out. Additions to our original view are shown in bold.

Example 14-1. A slightly more complex version of our original view from Chapter 13

```
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);
```

Let's pick apart some finer parts of this query to introduce some database concepts you may not be familiar with.

First, although normally you join against a table or view, you can also join against a named query. Indeed, recall from the previous chapter that a view is also nothing more than a named query. In this example, we've created a named query called `ptc`, for "purchased tickets count." It is a single-use named query. Unlike a view, this named query – `ptc` – has no meaning outside of this single place it is used; outside of the `current_movie_showtimes` view it is out of scope. Of course, we could also cast `ptc` as fully-fledged view of its own with `create view`, and then we could join directly against the view. That would make `current_movie_showtimes` more readable, and would also be a good idea if we wanted to use this subquery elsewhere. For now, we'll leave it as is and return to this idea when we talk about cascading materialized views.

Next, we've done a *left outer join* in our join against `ptc`. Unlike a regular join, which removes items for which there is no match between the two join tables, in a left outer join, every row from the table on the left remains regardless of whether there is a matching row from the table on the right. When there is no match, columns from the table on the right are filled with null values. In this example, if there are no tickets purchased for a given showtime, there would be no result row in the `ptc` subquery. However, we don't want to lose the fact that a showtime is current and has tickets available for purchase just because no one has purchased any tickets yet! That brings us to the third finer point of this query, `coalesce`.

The `coalesce` function takes an arbitrary number of arguments, and returns the first one that is not null. Here, we're coalescing the number of ticket purchases – which will be null if none have been purchased yet – with 0, which is the actual value we want output when there aren't any tickets sold. So although a left outer join normally returns nulls when there's no match in the right-hand table, we're substituting a value that makes sense for our domain.

Getting into form

Although it is not technically mandatory to do so, it makes it a bit easier to implement a materialized view if every row from the view's main table is present in the view to be materialized. In order to accomplish this, we re-cast elements of the where clause into Boolean columns in the table itself. Rather than filter out showtimes that aren't current, those showtimes will have a `false` entry in the `current` column, and movies that are current will have `true`. Likewise for sold out shows; they'll have a `true` entry in the `sold_out` column, and shows with seats available will have a `false` value there. Example 2 shows our rewritten view the new columns in bold.

Example 14-2. `where` clause recast as Boolean columns to ensure every row from main base table is always represented

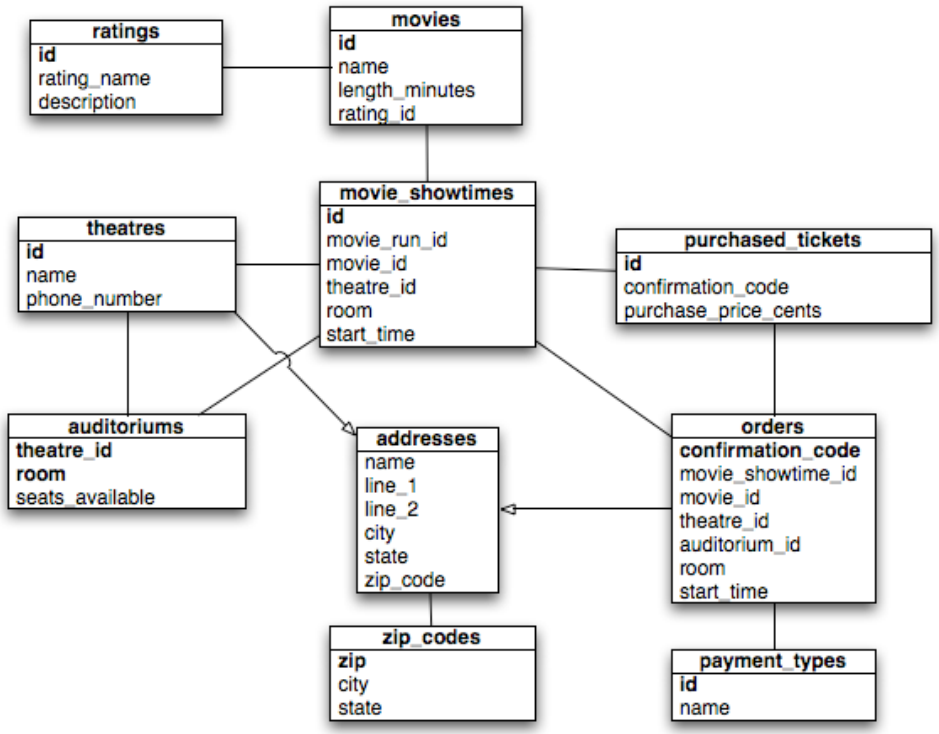
```
create or replace view movie_showtimes_with_current_and_sold_out_unmaterialized as
  select m.name,
         m.rating_id,
         m.length_minutes,
         ms.*,
         t.name as theatre_name,
```

```
        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);
```

Note that in Example 2, we renamed the view from `current_movie_showtimes` to `movie_showtimes_with_current_and_sold_out_unmaterialized`. The element of the new name `with_current_and_sold_out` refers to the fact that we've shifted the where clause filters into columns on which we can later apply filters. We've also added the suffix `_unmaterialized` to signify that this is the version of the view that is still just a named query. In keeping with idea that the caching implementation should be transparent to the user, by the end of this chapter, we'll have a new entity in our database called `current_movie_showtimes` which will look and act just like our original view, but will be orders of magnitude faster.

Another caveat worth mentioning is that the view to be materialized should be capable of having a primary key. This is another way of saying that there should be a one-to-one correspondence between the view and its primary base table, and that the primary base table needs to have a primary key. We've already helped guarantee this in our example by moving the where clause filters into columns, and having only one row in the view per record from `movie_showtimes`. The `id` column from `movie_showtimes` will become the primary key of the materialized view. Figure 1 is a reproduction of our schema diagram from Chapter 13, which you can refer to as we go along.

Figure 14-1. Schema from Chapter 10, for reference



The Target Table

A materialized view is created by taking an initial snapshot of the data in the unmaterialized view. Later we'll add triggers to monitor all of the tables that make up the view and update the view whenever there is a change. In this way, our materialized view always stays up to date.

To create the initial materialized view, we execute the following SQL:

```
create table movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry as
select *,
       false as dirty,
       null::timestamp with time zone as expiry
  from movie_showtimes_with_current_and_sold_out_unmaterialized;
```

This statement creates a new table called `movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry` that is pre-filled with all of the data from our view. Two columns have been added: `dirty` and `expiry`. The `dirty` column will be used to implement deferred refresh via the invalidation trigger. The `expiry` column will be used to deal with special cases where we can't count on a database event to trigger a refresh. How to use both of these columns will be explained in detail, but for now you can ignore these columns and think of the target table as a plain old table that happens to contain the result of our view. Example 3 shows the table described from a `psql` prompt.

Example 14-3. The physical table definition of our materialized view

```
movies_development=# \d
movie_showtimes_with_current_and_sold_out_with_dirty_and_expiry
    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                  |
 dirty                   | boolean                  |
 expiry                  | timestamp with time zone |
```

Refresh and Invalidation Functions

The next piece of the puzzle is the refresh function. The refresh function takes as its argument the primary key of the materialized view. In this case, that key corresponds to the primary key of the `movie_showtimes` table. Whenever we detect that a row in our view is invalid, we run the refresh function on that row.

Example 4 shows our first pass at a refresh function. It accepts an integer parameter, the primary key of the materialized view. First, it deletes the old row keyed on that id. Then, it re-selects the row with the same id from the unmaterialized view – which is real-time, and thus guaranteed to be accurate – and inserts it back into the materialized view. It also replaces the values in the `dirty` and `expiry` columns.

Example 14-4. A simple refresh function for a materialized view

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

Remember that the materialized view is just a table. You can modify it, thus invalidating the contents, and then run the refresh function on the modified rows to test that it sets them back to the correct values. Example 5 shows just that. We first find the movie name for the showtime with id of 1, *Casablanca*. Next, we invalidate that record in the materialized view by changing the movie name to be *The Godfather*. We check, and the materialized view indeed did allow us to change the record to an invalid value. We run our refresh function on that row, and when we select the name again, it has been restored to *Casablanca*.

Example 14-5. The refresh function patches an invalid row so that it matches the view

```
movies_development=# select name
movies_development-#   from
movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
movies_development-#  where id = 1;
    name
------------
 Casablanca
(1 row)

movies_development=# update
movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
movies_development-#    set name = 'The Godfather'
movies_development-#  where id = 1;
UPDATE 1

movies_development=# select name
movies_development-#   from
movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
movies_development-#  where id = 1;
     name
---------------
 The Godfather
(1 row)

movies_development=# select movie_showtimes_refresh_row(1);
  movie_showtimes_refresh_row
------------------------------

(1 row)

movies_development=# select name
movies_development-#   from
movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry
movies_development-#  where id = 1;
    name
------------
 Casablanca
(1 row)
```

Of course, in this case, we knew that the record was invalid because we invalidated it ourselves. In practice, it won't be the materialized view that changes to bring the two out of sync, but the unmaterialized one. We'll need to detect changes by watching all of the tables that make up the view with database triggers.

However, there are certain circumstances when observing changes in tables won't alert us to a change in our view. Such unobservable changes can arise from mutable functions

being part of the original view definition. For example, if we had a column based on the `random()` function, our materialized view would always be out of sync. Such cases are rare, though. The most common mutable function is `now()`, which appears in our view in the definition of the `current` column. Before we build any triggers, we'll first see how to deal with these unobservable, time-based events.

Time Dependency

Although a seemingly random mutable function can be tricky to deal with, dealing with a time dependency in a view is straightforward.

The problem we are facing is that it is not a change in the contents of any table that changes the value of the `current` column in our view, but simply the passage of time. In our original view, we have defined current to mean a showtime is in the future, and starts within one week. So with all else staying constant in our database, a showtime that is two weeks away should have a `false` value in `current`. After the passage of one week, it should switch to `true`. Another week later, back to `false`.

Because time always marches forward at the same pace, we know ahead of time the moment when the Boolean value in our materialized view needs to flip. If the showtime is far in the future, then `current` will become true one week before the start time of the showing. If the showtime is already current, it will become false when the present time is equal to the start time. And if the showing was in the past, it will never become current.

With this application-specific knowledge in hand, we can write a function that will tell us when a row in our materialized view should be considered invalid and in need of a refresh due to the need to update `current`. Example 6 shows this function. It takes an integer parameter referring to the primary key of our view. A local variable `start_time` is defined which will hold the start time of the showtime in question. Then, within the function body, we select the start time from the view and put it in that variable. Then we run through the logic above to determine the moment in time that our record should be invalidated.

Example 14-6. A function to determine when a time-dependent row should expire

```
create or replace function movie_showtime_expiry(
  id integer
) returns timestamp with time zone
security definer
language 'plpgsql' as $$
declare
  start_time timestamp with time zone;
begin
  select into start_time ms.start_time
    from movie_showtimes_with_current_and_sold_out_unmaterialized ms
   where id = id;
  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
$$;
```

Armed with this new method, `movie_showtime_expiry`, we can construct a better refresh function that will insert the correct record expiration time into the `expiry` column, rather than the null placeholder used in Example 4. Example 7 shows our new function, with the new elements in bold. Note that we've also modified the return type of the refresh function to return the expiration time. We'll use this later when we come to the reconciler view.

Example 14-7. A refresh function that calculates row expiry based on a showtime id

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

We have introduced an inefficiency here. Can you see it? Because we want to return the expiry value – again, why we do this will become apparent later in this chapter – we have evaluated our costly unmaterialized view twice: first, in the call to `movie_showtime_expiry`, which selects from the unmaterialized view; second, in the refresh function itself in the predicate of the select. Since our overarching goal here is to optimize, we'd rather not do this twice what could be done once. Correcting this inefficiency is left as an exercise for the end of this chapter.

One last problem with time dependency is that our initial snapshot did not contain any expiration information. It would have helped to have our expiry function ready before we generated the snapshot, but we can update the entire materialized view with the following:

```
select movie_showtimes_refresh_row(id)
  from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry;
```

If we had the function available from the start, we could also have created our initial materialized view with a SQL statement that took the expiry into account:

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

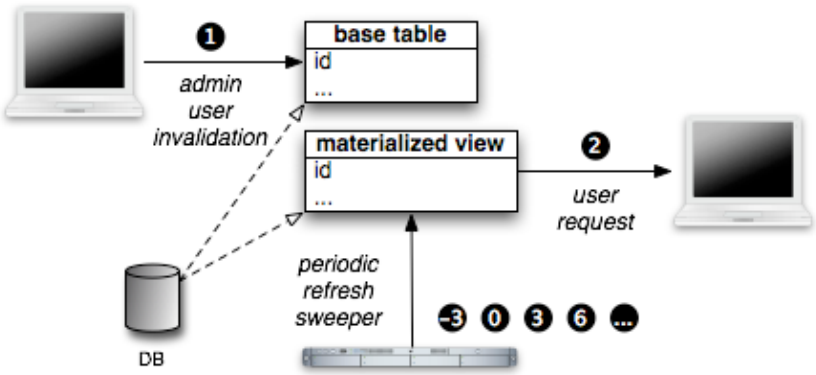
Who pays the price?

Nothing comes for free. Even if queries against a materialized view are fast – O(1) time – you still have a price to pay to keep the materialized view accurate. O(1) is nothing to shout about if the data you are retrieving is stale or invalid, and refreshing can be expensive. In fact, our refresh function makes it clear that the cost of refreshing is as expensive as the cost of querying our complex, slow, unmaterialized view. Therefore, it's important to minimize the amount of time you spend refreshing, and also to refresh at times that are least likely to be burdensome.

When any table involved in the view definition changes – whether rows are inserted, updated, or deleted – you may have an event that requires an update to the materialized view. However, it may not be wise to update the materialized view at the first opportunity we have to do so.

Figure 2 shows three possible times when we can update our materialized view. The first is at the exact moment when a change to a base table record causes corresponding records in the materialized view to become invalid. These events are easily detectable through database triggers, and are the topic of the next section. The second detectable time we can refresh is when a user is making a request to the materialized view. If we know records are invalid, we can refresh them just before returning data to the user. The third opportunity for refresh is not detected but is forced through periodic update of rows known to be invalid. This method alone is not enough to ensure cache correctness, but in conjunction with the first two methods it can help reduce the user-visible lag of refreshing invalid rows.

Figure 14-2. Timeline of refresh opportunities



Before discussing in detail how to decide which refresh scheme is best for which circumstances, we need to clarify how, in the latter two refresh schemes, you would know whether a record is invalid or not. We've already seen how the `expiry` column can be used for this purpose for records with a time dependency. We need an analogous way to know when rows which we chose not to refresh at the time of invalidation are indeed invalid. This is where the `dirty` column, which we created in our initial snapshot, comes into play.

Example 8 shows a new method, `movie_showtimes_invalidate_row`, which sets the dirty column of a particular row to `true`. For triggered events where we decide not to call the expensive refresh function immediately, we can instead call this method, which runs in O(1) time. Now our second and third refresh opportunities – just before

returning results to a user and the periodic refresh sweep – can check that a row is either dirty or has expired, and refresh only those rows.

Example 14-8. A function to mark a materialized view row as invalid

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

There are a number of considerations that go into choosing when to refresh. The first way of looking at the problem is in terms of who should pay the price. If you can split your users into two classes, admin users who are making the most changes to data, and customers, who are viewing that data, then the choice is clear. You're paying for customers and your employees are being paid to do their jobs, so the burden of update should in general be on the employee's shoulders.

However, in some cases the relationships between the tables themselves plays a role in determining whether it's appropriate to refresh immediately – while the actor is waiting – or to defer the burden to the viewer or the periodic sweeper. These relationships fall into three categories: 1:1 (*"one to one"*), 1:N, and N:1.

1:1 updates

In the first category, 1:1, every row you update in a base table corresponds to a unique row in the view. The most obvious example is the table from which the view gets its primary key, in this case `movie_showtimes`. If you update five records in the `movie_showtimes` table, five records change in the view and thus five records need to be refreshed or invalidated in the materialized view.

In this case, it usually makes sense to refresh the row in question right away, especially if only employees have the ability to invalidate the data. The additional time spent by the employee translates directly into time saved when rendering pages for your users.

1:N updates

For the second category, 1:N, an update of a single row in one of the base tables changes multiple rows in the view. For example, if we make a change to a movie or theatre, all of rows for that movie or theatre change in our view. An update to a single row could require 100, 1000, or even more refreshes or invalidations in the materialized view.

In this case, it's not obvious whether you should refresh rows immediately, or invalidate them for a deferred refresh. Clearly, refreshing so many rows at once could seem beyond reasonable, even for someone on payroll. The statement, "They don't pay me enough for this @!#%!" comes to mind. Factors affecting this decision are the number of rows likely to be invalidated at once, how many of these invalid rows are likely to be requested in a single request by a user, and how patient your employees are. If many rows are

invalidated, but they're requested by visitors one by one, it may make sense to have your site's visitors pay one row at a time rather than force employees to sit around waiting for a thousand records from a complex view to be refreshed. In the proceeding discussion of the reconciler view, we'll see how we can limit request-time refreshes to only to those records that are being requested.

N:1 updates

Finally, in N:1 relationships, a number of updates to some base table corresponds to just a single row in the view. This type of relationship is generally found in aggregate functions. For example, in our view, each showtime record has a count of the number of tickets purchased in the `tickets_purchased_count` column. Whether we add 10 tickets to an order for a given showtime (one row per ticket), delete ten tickets, or modify each of those tickets in some way, only one row in the view changes, and therefore only one record in the materialized view needs to be updated.

In this case, it is cruel and unusual to have the actor, generally your employees, pay the price of refresh *N* times when there is only one change to be made to the materialized view. Because each base table modification results in a trigger firing, if that trigger calls the complex refresh function, you will pay for the refresh function *N* times before the transaction is complete. Clearly, the result of refreshing once after all modifications are made is the same as refreshing *N* times, except that the latter case is a waste of time and more importantly, database resources. Therefore in this case the triggers related to N:1 relationships should invalidate materialized view rows for deferred refresh rather than perform the refresh in place. The cost to invalidate is negligible, and the heavier cost of refreshing need happen only once.

Triggered Refreshes and Invalidations

So far we have built a snapshot of our view at a given point in time, and we have created stored procedures that can be used to invalidate and to refresh rows in the materialized view snapshot. Now it's time to build triggers that will refresh or invalidate rows automatically as changes are made to underlying tables.

In general, triggers follow these steps:

1. Determine if any change to the materialized view is necessary, and quit early if not.
2. Determine which rows, by primary key, need to be refreshed or invalidated.
3. Call the refresh or invalidate function on those primary keys.

Writing these triggers can be a tedious process, because we need to account for inserts, updates, and deletes on all tables that make up the view. In this case, nearly all of our tables – six – are involved in the view in some way. With three functions per table for each of insert, update, and delete, this could mean we need to write eighteen trigger functions. Luckily, with some proper analysis, we can eliminate the need for more than half of these.

To facilitate this analysis, we create a reference table as shown in Table 1. We list each table involved in the view. For each table, we determine its relationship to the view: 1:1, 1:N, or N:1. Then, for each operation on the table, we first determine whether any action is needed at all, and if so, we choose whether we will refresh immediately, or defer the

refresh by performing an invalidation operation. We'll examine each table in turn to see how we came up with the entries in this table.

Table 14-1. A summary of view base tables and how they relate to invalidation or refresh

Table	Relationship to view	Operation	Action Needed?	Refresh	Invalidation
movies	1:N	insert			
		update	√	√	
		delete			
theatres	1:N	insert			
		update	√	√	
		delete			
movie_showtimes	1:1	insert	√	√	
		update	√	√	
		delete	√	√	
orders	N:1	insert			
		update	√		√
		delete			
ticket_purchases	N:1	insert	√		√
		update	√		√
		delete	√		√
auditoriums	1:N	insert			
		update			
		delete			

Movie Showtimes

As we've already discussed, the `movie_showtimes` table shares its primary key with the view, and therefore has a 1:1 correspondence. When a new showtime record is inserted, we need to add that record to the materialized view. When a record is deleted, we need to delete the record. And when a record is updated, we need to update the corresponding record.

Example 11 shows three functions to handle each case of update, insert, and delete. There are three new features of PL/pgSQL relating to trigger functions worth noting here.

First, functions intended to be used in conjunction with table triggers must have a return type `trigger`.

Second, functions called from triggers can implicitly receive two parameters: `old` and `new`. On an update, both of these are present, and `old` refers to the record before the

update, and `new` refers to the record after the update. On an insert, only `new` is provided, and on delete, only `old`.

Finally, we have used the keyword `perform` in our trigger functions. `perform` is used when you don't intend to store the result of a select statement. In these cases, you replace the keyword `select` with `perform`.

Note that in general, updates are a special case. If the record changes in a way where the referenced primary key changes, we need to take action on both the old and new primary key.

Also note that the method names follow a particular pattern. First, we prefix them in a way that identifies the materialized view they are for: `ms_mv_` for "movie showtimes materialized view." Then we identify the table this trigger function is for, here `showtime` as a shortened version of `movie_showtimes`. Finally, we append to the function name an identifier of whether this is the insert, update, or delete trigger with `_it`, `_ut`, or `_dt`, respectively. This pattern will be followed for all tables and triggers.

Example 14-9. Triggers functions for the `movie_showtimes` table

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

The naming convention above intended to help you identify which function is for what purpose, but from the database's perspective, these are just arbitrary names. For each function, we also need to add a corresponding trigger to the table itself so that the database knows which method to call when each particular event occurs. Example 12 shows how we add these triggers.

Example 14-10. Actual trigger declaration for the `movie_showtimes` table

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

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

create trigger ms_mv_showtime_it after insert on movie_showtimes
  for each row execute procedure ms_mv_showtime_it();
```

Movies

The `movies` table, as noted previously, has a 1:N correspondence with our view. A change to a single movie affects all of the showtime records associated with that movie. Our trigger function must select a column of showtime ids for refresh. Because only an employee could change a movie record, an immediate refresh was chosen rather than an invalidation, which would make users viewing the site pay for the refresh.

Because a movie has no impact on our view until it has showtimes, we do not need a trigger on insert or delete of a movie. On insert, there can be no showtime records yet for that movie. On delete, there can be no records because the referential integrity constraint which guarantees that a showtime reference a valid movie. All of the `movie_showtimes` records would have already been deleted or updated to reference a different movie before a delete on `movies` could succeed, and the refreshes triggered after modifications to that table would have cleared all of the records from the materialized view.

Example 14-11. Triggers for the `movies` table

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

Theatres

Our treatment of the `theatres` table exactly matches the treatment of `movies`. The theatres table also has a 1:N correspondence, so our trigger function must select a column

of showtime ids for refresh. Just as with the `movies` table, referential integrity constraints prevent a theatre from being deleted while showtimes reference it, so we do not need a delete trigger function. Similarly, when a theatre record is first inserted, it has no showtimes and therefore cannot impact the view. Our single update trigger function is defined in Example 12.

Example 14-12. Triggers for the `theatres` table

```
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 after update on theatres
  for each row execute procedure ms_mv_theatre_ut();
```

Orders

The `orders` table has an interesting relationship with the materialized view. It does not directly affect the view at all, but the records in `purchased_tickets`, which are linked to the `movie_showtimes` table through the `orders` table, do. Therefore, adding or removing an order record has no effect on the view, but a modification to the order which would alter the showtime it is for – and transitively its associated tickets – does have an effect. Therefore, we need only an update function, and we only need to perform the refreshes or invalidations if the `movie_showtime_id` foreign key reference changes. Any other changes have no affect on the view. In this case, we have chosen to invalidate the row.

Example 14-13. Triggers for the orders table

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

create trigger ms_mv_orders_ut after update on orders
```

```
for each row execute procedure ms_mv_orders_ut();
```

Purchased Tickets

Ticket purchases impact the `purchased_ticket_count` column of the view, but only the presence or absence of any given row is relevant. Therefore, we certainly need an insert and delete trigger for the `purchased_tickets` table. We do also need an update trigger, but it is constrained to take action only if there is a possibility that the tally for the ticket needs to be moved from one showtime to another. This is only possible if the ticket purchase is re-associated with a different order, which might be for a different showtime. Therefore, the update trigger takes action conditionally based on whether the `order_confirmation_code` foreign key column undergoes a change.

Because ticket purchases have an N:1 relationship with a showtime in our view – all the purchases for a showtime are counted and affect a single column in a single record – we call the invalidation function in our triggers rather than the refresh function. If ten tickets are purchased, we want to refresh only once, not ten times, since the end result is simply to increase the `purchased_tickets_count` column by ten. Calling the refresh function ten times to increment the count one ticket at a time is a waste of time and resources.

Example 14-14. Triggers for the `purchased_tickets` table

```
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 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 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_ut after update on purchased_tickets
  for each row execute procedure ms_mv_ticket_ut();

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

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

Hiding the Implementation with the Reconciler View

Now that we have our triggers defined, we have added yet another way for the materialized view to decay. The first decay mechanism was the expiry column, which allows rows to declare when they should be treated as irrelevant. The second mechanism is the `dirty` column, which our invalidation function sets to true when certain tables receive updates.

Slowly but surely, our materialized view will become a minefield full of stale records we need to avoid if we aim to present accurate data to database clients. Such a table is by no means a drop-in replacement for the original view. Not only is the materialized view slowly turning into garbage, but also the interface is different. If selecting directly from this table, a client must be careful to avoid stale or invalid rows. The logic that was neatly contained within the view's `where` clause filters is now contained in columns, which the client must explicitly filter on.

We will now plug up these holes, first ensuring that the data returned to clients is always up-to-date. Then we'll give back to the client the original interface provided by the original view. We'll hide the `dirty` and `expiry` columns, and transform the `current` and `sold_out` columns back into a filter.

We accomplish the first goal of always returning accurate data with the *reconciler view*. We give this view the same name as our original, "well formed" view from Example 2, but without the suffix `_unmaterialized`. It is simply called `movie_showtimes_with_current_and_sold_out`. This new view is shown in Example 15. It is the union of two select statements. The first returns rows from our physical materialized view table which are neither dirty nor expired.

The second part of this view contains the magic. It returns records that *look like* data from the materialized view, complete with a false dirty column, and an accurate expiry time. Recall when we built our refresh function in Example 7, we constructed it so that it would return the expiry time of the new row being inserted. That was not without purpose; we make use of that behavior here to create a complete yet functional façade for the materialized view, which hides the mixture of accurate, expired, and dirty rows. When we call the refresh function, it both fills in the expiry column accurately, but more importantly, it refreshes the expired or dirty row.

This is as close as we get in this book to pure magic. *The process of requesting rows from the reconciler view refreshes the expired rows in the materialized view.* The

reconciler view, then, that is nearly our end-state view. Although it does have two additional columns that our original view did not have – expiry and dirty – it is essentially a drop-in replacement for the view we started out with in Example 2, as it is always accurate.

Example 14-15. The reconciler view provides just-in-time cache-correctness

```
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(w.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));
```

> In the reconciler view, we use `union all` rather than `union`. A SQL `union` returns only unique rows. To do so, the result rows must be first be sorted, followed by a unique operation to filter out any duplicate rows. Since we aren't expecting any duplicate rows, using `unique` can be much more efficient if we're requesting a large number of rows from the reconciler view at once.

You may have noticed that the way we implemented the reconciler view, with it's selection from the unmaterialized view *and* a call to the refresh function for each invalid row, actually evaluates the unmaterialized view twice for each invalid row that needs to be updated. This is unfortunate, but is still a vast improvement over an evaluation of the unmaterialized view for every page request. Of course, nothing is impossible, and therefore writing a reconciler view that does not evaluate the unmaterialized view twice is not impossible. However, such an implementation is beyond the scope of this book, and strays from our purpose here: that is, the fundamentals of materialization and cache correctness. A more complete materialized view implementation is available at this book's web site, located at http://enterpriserails.chak.org.

You may also be wondering, when you select from the reconciler view, how many rows are refreshed? Are all of the expired and invalid rows refreshed, which could be quite costly, or just the ones that influence the query? In fact, it is the latter. Example 16 shows a set of SQL queries to illustrate this. First, two rows in the materialized view are manually set to be dirty. Then, one of those rows is requested from the reconciler view. Finally, the dirty column of both rows is selected directly from the materialized view. Only the one we selected from the reconciler view has been refreshed, and it now has a `false` value in the `dirty` column.

This property of the reconciler view plays a big role when you're choosing between invalidation or refresh for 1:N table relationships. If it's common for rows to be selected from the materialized view in small chunks rather than all at once, you can amortize the full refresh of a 1:N invalidation over a number of future site visitors. Alternatively, the user who invalidated the rows – often an employee – must pay the price of refreshing N rows all at once.

Example 14-16. When selecting from the reconciler view, you only pay for refreshing the rows you select

```
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)
```

Periodic refreshes

In Figure 1, I alluded to a periodic refresh activity. Now that we have seen the reconciler view, such an activity is trivial to implement. That activity is simply the variation on the bolded portion of the reconciler view in Example 15. We simply need to select the refresh function on all of the rows that are expired or dirty. This can run via a cron job at a given internal to alleviate some of the burden imposed on site visitors whose requests go through the reconciler view. If the refresh function is called between an invalidation operation (either explicit or implicit due to an expiration date passing) and a request, then cron pays rather than the next visitor.

Example 14-17. A refresh function to be run periodically

```
select movie_showtimes_refresh_row(w.id)
 from movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry w
where dirty is true
   or not(expiry is null or now() <= expiry));
```

Completing the circle

Our reconciler view, although it is a nearly identical drop-in replacement for our well-formed view from Example 2, is not the same as the view we began with in Example 1, `current_movie_showtimes`.

At this point, it is not hard to create a new view with the same name that has the same output, but is based on the materialized view. We just request all rows except dirty and expiry from the reconciler view. We also omit our Boolean columns from the new view, and instead we use them as filters. Example 18 shows the new definition of our original view. Now we've come full circle.

Example 14-18. A view indistinguishable from our original `current_movie_showtimes` *view, but based off the reconciler view.*

```
create view current_movie_showtimes as
  select name,
         rating_id,
         length_minutes,
         id,
         movie_id,
         theatre_id,
         room,
         start_time,
         theatre_name,
         zip_code,
         latitude,
         longitude,
         seats_available,
         purchased_tickets_count
    from movie_showtimes_with_current_and_sold_out
   where current is true and sold_out is false;
```

Figure 3 shows the progression of views, tables, and wrapper views we created to completely hide the implementation of our materialized view from clients.

Cache Indexes

Our materialized view implementation is actually rather useless if we do not add indexes. Although we don't need to evaluate the complex view query when we query the materialized view, without indexes, each request would need to run a full table scan in order to return any results. Indexing makes queries by id, or by one of our original filters close to instantaneous.

There is a minimal set of indexes we need on a materialized view. First, we need to index the primary key column. It's fine to do this by creating an explicit primary key. Next, we need to add indexes on the `dirty` and `expiry` columns, since they are part of the `where` clause of the reconciler view. Indexing these columns keeps that part of our implementation fast. Finally, we should index the filters that we recast as columns, `current` and `sold_out`, since it's likely we'll be filtering on these columns frequently. Apart from this set – the primary key, the invalidation implementation columns, and the filter columns, you can index any columns in your materialized view that your application will select or filter on. The creation of our primary key and indexes is shown in Example 19.

Example 14-19. A minimal set of indices on a materialized view

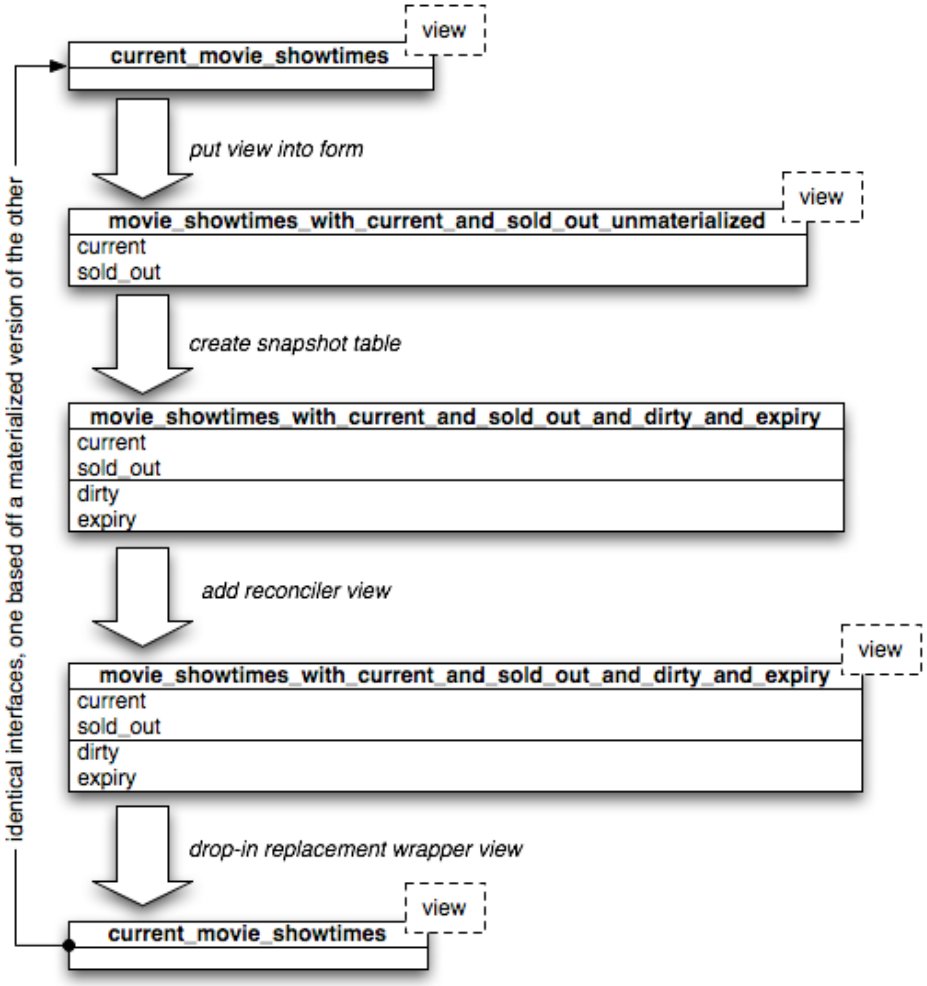
```
alter table movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry add
primary key (id);

create index movie_showtimes_with_current_and_sold_out_dirty_expiry_idx
  on movie_showtimes_with_current_and_sold_out_and_dirty_and_expiry(dirty, 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);
```

Figure 14-3. The progression of views and tables to abstract the materialized view implementation from clients



Results

The results of using a materialized view rather than an unmaterialized one are quite impressive. In Example 20, a select from both for current, non-sold out showtimes is analyzed.

First, the count of records in each table is selected to give you an idea of how much data we are dealing with. In fact, it is not too much data compared to what a real production site selling tickets for all theatres and all movies nationally might have in its database. Our data set likely accounts for a day or at most a week's worth of accumulated data on a real system.

Next, we select all of the current, non sold out showtimes from the unmaterialized view. On a Dual Core 2.1 Ghz MacBook Pro, the query takes 1.16 seconds. Next we issue the same select against the materialized view. It takes 0.013 seconds. With this dataset,

Selecting from the materialized view is almost 100 times faster. As the dataset grows, the time required to select from the unmaterialized view continues to grow, while the time to request from the materialized view remains nearly constant.

Example 14-20. Comparison of runtimes on a view versus a materialized view

```
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
movies_development=# explain analyze
   select id
     from movie_showtimes_with_current_and_sold_out_unmaterialized
    where current = true
      and sold_out = false;
Total runtime: 1158.617 ms

movies_development=# explain analyze
   select id
     from movie_showtimes_with_current_and_sold_out
    where current = true
      and sold_out = false;
Total runtime: 12.553 ms
```

Cascading Caches

In Example 2, we joined against a named query subselect to create our view. We noted that this query could also be recast as a fully-fledged view in its own right. By extension, it could also be recast as a materialized view.

If that were the case, we would be cascading two materialized views. The inner materialized view would be concerned with orders and ticket purchases, and the outer view would no longer need to watch those tables directly. Instead, the outer materialized view would maintain triggers on the inner materialized view.

Working this way can reduce the complexity of any given materialized view. It also speeds up the inner view, for cases where there are other uses for that data. Implementing a cascading materialized view is left as an exercise.

Exercises

1. Write a stored procedure that verifies a one-to-one correspondence between rows in the materialized "reconciler" view and the unmaterialized view.
2. Using the stored procedure from Exercise 1, write a series of unit tests that modify records – one at a time as well as multiple at a time – and then assert the validity of the reconciler view.
3. It was noted that our refresh function does extra work to calculate and return the `expiry` value. Write a new `movie_showtime_expiry` stored procedure that

takes the `start_time` as a parameter, and returns an expiry value *without* selecting from the unmaterialized view.

4. Rewrite the snapshot creation query and the refresh function to use the new procedure you wrote in Exercise 3. Time the snapshot creation and refresh operations with both methods. Which set is faster, and by how much?
5. Following the procedure outlined in this chapter, create a materialized view for ptc, the named subquery from Example 2. Then, rewrite the triggers for the main materialized view to use this new physical table. How do you propagate invalidations?