Story UFC-199: Transactions Date Filtering

The following document will go over the design of XXX, as outlined in Story ticket UFC-XXX.

Acceptance Criteria

For reference, the following are the acceptance criteria for this ticket:

- When I go to the Transactions page, I can see the current month's transactions.
- I can change to see the next or previous month's transactions.
- I can change the size of the range of transactions that I can see between daily, weekly, monthly, yearly, all time, and custom.
- I can specify manually specify the start and end dates of the date range.
- When I manually specify the start/end dates, the date range size changes to 'custom' unless my custom dates form a size that matches an existing range size.
- When I specify a 'daily' range size, the start and end dates are both set to the same date.
- When the range size changes, the changing between different intervals of dates should happen based on the range size (i.e. when the range size is 'weekly', the 'next interval' should move ahead 1 week, etc).
- When the date range size is 'all time', the interval changing buttons should be disabled, since they don't make *any* sense in this case.
- See Transactions Page for the logic of determining ranges when switching between range sizes.

Design Brainstorming

As far as I can tell, there are going to be 3 big pieces to implementing date filtering:

i. The transactions date index in the store, that manages indexing and searching transactions given a certain queries.

ii. The date range Context hook, that manages the state for the currently selected date range 'query'.

iii. The transactions date range Context hook, that takes the query from the above hook and combines it with the store's date index to actually search up the transactions.

Together, these three pieces will enable an easy interface for very quickly querying up ranges of transactions.

Now let's discuss each these pieces in detail.

Transactions Date Index Slice

As mentioned, this piece will be a new slice in the Redux store that will store an index mapping dates to transactions. It will be very similar to the existing transactions index -- which indexes using descriptions -- in how it is used.

I've already done a large amount of research and thinking about how this index should be implemented; unfortunately, I can't link between Nuclino workspaces, but this research can be found in the `Personal Workspace` of the `Dilgess`, under "Indexing Time Series Data".

tl;dr: We can use the following schema:

```
{
    2020: {
        may: [
            [] * 31
        ]
    }
}
```

Essentially, this will allow us constant time lookups by date (technically O(1 + n), where n = the number of transactions on the date, since they're stored in an array and need to be scanned) and linear time lookups for date ranges (O(n + m), where n = the number of days in the range and m = number of transactions).

This was the best I could come up with that combines speed and simplicity, since a date range query (the most common kind of date query we're gonna have) just requires looking up the start date and scanning every day from the start till the end.

Update: OK, so you know how it says "may" up there? Yeah, that's stupid, it should be a number (0-indexed, cause JavaScript dates are stupid).

But more importantly, did you know that `Object.keys` *always* returns the keys in a sorted order, if the keys are numeric?!? I did not know this when I first designed this data structure. So if we want to be space efficient, we can leverage a map *for the days* and still maintain our sorted requirement, while also having the index be *a lot* more sparse. It would look like this:

```
{
    2020: {
        5: {
            3: ["id123"]
        }
```

```
    }
}
```

So now our range scanning time is reduced back to being linear on the number of transactions, instead of the number of days + transactions.

Update 2: I have made a terrible mistake.

Ya know how, below, I wanted to pass the whole transaction as part of the deletion action? And then decided against that in favour of building a separate index for the Account Balances Date index?

Well, we *might* still need to pass transactions along in deletion. Cause, otherwise, how are we supposed to lookup the transaction in the date index to delete it? We can't, cause we don't have the date! I mean, obviously, we could do a full index scan to find it, but I feel like that might be *slightly* inefficient.

So we're just changing the damn deletion action to pass along the whole transaction!

Update 3: I have made *another* terrible mistake.

Ya know how, below, we needed to create *an entire extra map* just to handle updates? Because we need the old values?

Well how do you think I feel now, knowing that that's *also* a problem for *this* index!??

How are we supposed to update a transaction to a new date, if we don't have a super fast map of ID → date already??

By realizing that date updates are a relatively operation and that introducing *another* map to handle this would be a poor idea. As such, we're just going to do a linear scan of all IDs in the index to find the ID and then remove it.

But seriously, it's a bad idea to add a new index *just* to optimize this case.

However, that doesn't mean we can't optimize it. For example, a good heuristic we can use is that if the user is updating the date of a transaction, they're *likely* not changing the month. And even then, they're *likely* not changing the year.

As such, we can at least start the search with the month of the updated transaction, and then scan backwards/forwards through the year before deciding that we need to scan the rest of the index.

That'll likely get us pretty good performance without having to add extra indexes.

You know what else this means? We're nullifying **Update 2** and *not* changing the transaction deletion action to include the whole object.

But there's a Catch - Account Balances Date Index

While this state structure works very well for querying transactions in a given range, there's another problem we have to deal with: querying and calculating the total of the transactions *before* the range.

See, a number of different features in uFincs depend on showing the amount that a value is coming 'from'; for example, the Account Summaries on the Dashboard or the Type filters on the Transactions page.

These types of elements require two calculations: a summation of all the transactions in the given date range, *on top of* a summation of all the transactions *before* the date range.

Obviously, given the structure we have above, the first calculation is fairly trivial, and while the second calculation is *technically* trivial (in that we know *how* to calculate it), it's not necessarily the most *efficient* way to calculate it.

Because really, what we need is a 'starting balance' for every date range query to then add onto with the transactions in the query. So what are our possible solutions for this?

- Well, the most naive solution is the one stated above: for every date range query, just lookup every transaction before the start of the query and calculate the total.
    - Again, this is obviously very inefficient, because it means that every date range query will always be linear in the number of transactions (on top of the number of days, so O(n + m)).
- The second most naive solution is to *cache* (or precompute) balances for every account, for *every* day.
    - Side note: You might think that we could compute balances for every account, for every *transaction* (since we'll need this information when we go to implement the Account Details view), but I think this overkill when the per-transaction balance can be trivially computed on the fly by taking the day-before's balance as a starting point.
        - Then again, for very large date ranges, they're probably both going to be pretty intensive.
    - This solution obviously gives us the best performance, *at query time*, at the expensive of having a (relatively) expensive pre-computation step app boot.
- I can't think of any better solution.

So, if we're going with a precomputed approach, what might that look like?

- Well, we need to compute balances for each account, so starting with indexing the data by account ID is probably a good idea.

- Then we need presumably need to index by date. Well, we already have a nice date indexing format (detailed above), with year -> month → day → (balance), but I think this date index should be slightly different.
    - Because we don't need to linearly scan these values, we can store the days as an object mapping "day number → account balance" instead of an array.

OK, so we have our data structure. It should look something like this:

```
{
    accountId: {
        2020: {
            may: {
                3: {
                    12345
                }
            }
        }
    }
}
```

So now how do we interact with this data structure? What are our algorithms? Well, we'll need algorithms for the following:

- Account created
    1. Add ID with an empty object.
- Account deleted
    1. Remove ID.
- Accounts set
    1. Delete the entire index and add all the account IDs.
- Transaction created
    1. Using date of new transaction, find* first date that comes before it.
        - Hmm, I think this process of 'finding' the previous date might be slightly less optimized than it could be since we decided to use an object instead of an array for the days.
        - In order to 'find' the previous date, we'd need to do a scan on the year/month/day to find it.
            - I suppose we're probably going to need some custom algorithms just for interacting with this data structure (insert, find, find previous, find next, delete, update, etc).

        - Also, we need to consider the case where the new transaction has the same date as an existing balance calculation. I suppose that's then the trivial case for 'find'.
        - But whatever, we'll figure it out.
    2. Add the transaction's amount to that balance.
    3. Insert a new* date with new balance.
        - Obviously, we also have to consider the case when the new transaction has the same date as an existing balance calculation.
        - In that case, we just need to update the balance with the new amount.
    4. Update all balances that come after the new transaction's date.
        - This could get slightly tricky with our data structure. Obviously, this is going to be a linear operation (O(n) where n = the number of days after the new transaction), but scanning the data structure is where it gets tricky.
        - Like I said above, we'll need to develop some algorithms for this data structure; a complete forwards and backwards scan will just be part of it.

- Transaction deleted
    - Hmm, at first glance, this seems a bit tricky. Because the transaction deletion event only has the ID of the transaction in the payload, I don't know how we're gonna relate the ID back to the date to look the transaction up in the index. I gotta lookup how the description index works... I'm sure *it* uses IDs in the index... yep, it does.
    - So I guess this means that we need an ID → date, amount map? But then we'd be duplicating data... but I guess we're kinda already doing that with the descriptions index? Because the 'words' are technically duplicate data of the descriptions. Hmm...
    - I mean, the *really* easy solution would be to have the whole transaction be passed as the payload for the deletion event, but I don't know if that seriously breaks anything or not.
        - Looks like it shouldn't? The only places where transaction deletion happen are the transactions sagas (duh), the existing transactions index (duh), and the accounts saga. Of these three, only the accounts saga instance needs to be slightly modified, since it currently pulls in the non-populated account; change that to the populated account to get the full

transactions, dispatch deletion actions with full transactions, and hey presto.

  - So, if we change transaction deletion (making a note in the slice *why* transaction deletion uses the full transaction object, as opposed to everything else that expects an ID -- actually, that makes me think that the `crudSliceReducerFactory` will need to be adapted to support a different payload for the delete action), then implementing date index deletion is a lot easier.

  - OK, so after thinking about how to implement update (see below), I've decided to just implement the extra ID → date, amount map.

  - So here's the flow:

1. Look up the date of the transaction.
2. Subtract the amount of the transaction from the balance of the date.
3. Find the balance of the previous date.
4. If the transaction's date's balance is equal to the previous date's balance, then remove the transaction's date from the index (this is because it was the only transaction on that date)
5. If they're not equal, then leave it be; deletion is done.

- Transaction (amount) updated

  - OK, *now* it gets really complicated. How are supposed to update a date's balance if we don't know what the transaction's old amount is? In the action, we'd only have the new amount. Hmm...

    - My first thought was to lookup the transaction in the store as part of the saga, but that doesn't work since the transaction will get updated before we can look it up (due to the reducer/saga action ordering).

    - My second thought goes back to one of the solutions for deletion: having a separate map for ID → date, amount. This way we can know what the old amount was. Then we can just lookup the date, subtract the old amount, add the new amount, and done.

      - But man, I can't help but feel like having all of this duplicated data is bad, both from a normalization standpoint and just a 'sheer amount of data' standpoint. There's going to eventually be a breaking point for how many transactions we can store (which should hopefully be obvious; it's just that adding more indices just brings us closer to that amount).

        - Although, I suppose at one point in the future, we'll *eventually* hit a point where it becomes infeasible to send *all* of the transaction data to the browser, to the point where uFincs can't be considered truly offline first. But hopefully that day doesn't come anytime soon.
            - Thankfully, we'll have good mitigation tactics for this: mainly storing account balance snapshots so that only the X last (years) of transactions have to be sent to the client before it requests more separately.
    - I mean, at least one benefit of having this extra mapping is that we don't have to change transaction deletion! Right?
    - Ah screw it, let's just have the extra mapping.
- So here's the update flow:

1. Lookup the date and amount of the transaction.
2. Lookup the balance on the date of the transaction.
3. Subtract the old amount of the transaction.
4. Add the new amount of the transaction.

So yeah, there's a lot to get right here.

One thing that I've been thinking of is that, since we've now decided to include the ID → date, amount data in this index, should we change the main data structure to include this data or should this data be a separate data structure?

I kinda just want to clump it all together in the one data structure, such that days are broken down into transactions. This way we can then have running balances at the transaction level, instead of the day level.

However, if we do that, that means that we'd need to keep the transactions ordered (by updatedAt/createdAt time) so that the running balance is strictly correct. I think that might be *too* much.

So let's just go with an extra, separate map.

So we've had all this discussion, but I completely forgot that this section was under the date index; I think this account balances index is now a completely different slice, separate from the transactions date index slice.

Let's call it the `accountBalancesDateIndex` slice.

Also, it doesn't really make sense to build this new index as part of *this* story, since this story is only focused on filtering the transactions list/table, which only needs the transactions date index. So we've leave building this index for the Type filtering story.

Something to note about querying this index:

- We'll need selectors that use the index and certain configurations of accounts (most notably, accounts by type) since we very often want to aggregate several account balances as opposed to individual accounts.
- However, we definitely still need to be able to query each account, since something like the Accounts scene will need the balances for each account for the Accounts list.

This actually makes me think of something:

- We're gonna need to completely revamp how account balances are calculated at every point in the app to leverage this index. I think this primarily means that we need to throw out the balance calculation that's part of populating an account, and then derive the balance from the index.
- So where are account balances even used? At the current point of implementation for the redesign, I think it's only the Accounts List.
- And I think, going forward, all uses of account balances are going to be under a date range filter, so this should be fine.
- However, since adding the date range filter to the Accounts scene is outside the scope of *this* story, we can leave the current account population algorithm as is.

Balances for Income/Expense

So something just occurred to me... whenever we want to calculate the balances for income/expense accounts, we only want to view their amount *for the current* date range *or the previous* date range.

For example, take the Type filters on the Transactions page. When we calculate the current amount, obviously we need to calculate for the date range that the user has currently selected. But for the 'From' amount, we want to calculate the amount *for the same amount of type in the current date range*, but just the interval *before it*. As in, the 'From' amount for income/expense accounts *is not* a summation of all transactions from the beginning of time.

Now, this might seem strange considering the current implementation of Income/Expense balances (for the Account page, in original uFincs) *does* show the total balance from the beginning of time. However, for that use case, I'm saying we should change it so that it reflects against the selected date range.

So, what does this all mean? It means that we *can't* pre-compute balances for the income/expense accounts. There'd be no point to it.

So then... what do we do? Well, I guess the only thing we *can* do is compute their balances on the fly. Cause their balances are *literally* dependent on the date range; the only thing we *could* cache is the balance for an exact date range, but that doesn't *really* seem worth it.

Although, I *guess* it would help a bit because it would speed up date range interval increments/decrements, since we'd have a whole interval already calculated, but I feel like it'd be more work do deal with the cache busting than just re-computing it all the time. And anyways, interval level calculations *should* (on average) be a lot faster, considering the intervals should be generally relatively small (compared to, ya know, 'all time').

So I guess that means the only thing that goes into this 'account balances date index' is the balances for assets and liabilities. Well, at least that'll speed up boot-up time?

Date Range Context Hook

So the `useDateRange` hook uses a custom Context provider to store the state needed for specifying a date range query.

So what state do we need?

- start date
- end date
- range size

And what actions do we need?

- set range size
- set start date
- set end date
- increment date range
- decrement date range

And then there's all the business logic needed when changing between range sizes and incrementing/decrementing ranges. As stated in the Acceptance Criteria, it's all documented in Transactions Page, but here's the tl;dr:

- When changing range sizes:
  - `Daily`: Take the last day in the last range.
  - `Weekly`: Align the end with the end of the last range, such that the start is 1 week before the end.

- `Monthly`: Set the range to the full month of whatever month the last range was in.
    - If the last range spanned two months, take the latest month.1
    - If the month is the current month, set the range only up to today.
        - **We're not doing this.** See below.
- `Yearly`: Set the range to the full year of whatever year the last range was in.
    - If the last range spanned two years, take the latest year.
    - If the year is the current year, set the range only up to today.
        - **No**. See above then see below.

- When increment/decrementing ranges:
    - `Daily`:
        - A range is considered to be `Daily` in the following cases:
            i. The start and end date are the same.
        - Switching between days works like this:
            i. Incrementing adds 1 day to the start/end date; decrementing subtracts 1 day from the start/end date.
    - `Weekly`:
        - A range is considered to be `Weekly` in the following cases:
            i. The number of days between the start and end date is 6 days (i.e. the range comprises 7 days). That's it.
        - Switching between weeks works like this:
            i. Incrementing adds 7 days to the start/end date; decrementing subtracts 7 days from the start/end date.
    - `Monthly`:
        - A range is considered to be `Monthly` in the following cases:
            i. The start date is the first of a month and the end date is the last of a month.
            ii. The start date and the end date share the same day #, with exactly a month in between (e.g. "May 13" and "June 13" is a

'month', but "May 14" and "June 13" is not, and would be marked as `Custom`).

iii. The start date is the first of the current month and the end date is the current date.

- Actually, you know what, ***screw this***. I'm not gonna have this special "today" case for monthly -- the default `Monthly` view will just be the whole current month. That way it's easier to implement, and we get the future benefit of being able to show users their upcoming transactions.

- Switching between months works like this (matching the above cases):

i. Incrementing a month gets you to the first/last of the next month; decrementing a month gets you to the first/last of the previous month.

ii. Incrementing a month sets start date to the previous end date and sets end date to end date + 1 month; decrement a month sets start date to start date - 1 month and sets end date to previous start date.

– `Yearly`:

- A range is considered to be `Yearly` in the following cases:

i. The start date is January 1st of a year and the end date is December 31st of the same year.

ii. The start date and end date share the same month/day, but their year is 1 apart.

- Switching between years works like this:

i. Increment/decrement the year of the start/end date.

ii. ^^

Setting Start/End past each other

Seems like there's one case that I completely neglected... the user setting the start date past the end date or setting the end date before the start date. Here's how I think these should be handled:

i. Start past end

– Set the end date to the start date and change the range size to daily

ii. End before start

- Set the start date to the end date and change the range size to daily

Transactions Date Range Context Hook

Really, all this is is a Context hook that uses the `useDateRange` state to query up transactions in the date range using a selector (ala how `useTransactionsSearch` currently works). I don't think it even needs any internal state or actions itself; we can just re-expose the ones from `useDateRange`.

Actually...

I don't really think the `useTransactionsDateRange` hook actually makes much sense. There's no need for it to 're-expose' the state/actions from `useDateRange`; the consumer can just *use* `useDateRange`.

So then where do we put the logic for grabbing the date filtered transactions from the store? I think it should actually just be in the pipeline of `useFilteredTransactions`. Instead of that hook receiving the transactions from somewhere else (currently, the connected Transactions scene), why doesn't the hook just fetch the transaction itself from the store? In this case, the date filtered transactions.

That way, the logic for filtering the transactions (by date, by pagination, etc) is all self-contained and can be re-used on a whim, very easily.

It would also mean that the Transactions scene no longer needs to connect to grab the transactions; it just needs to make sure it has all the right Context providers for the subcomponents (the Transactions table and list) to filter the transactions themselves.

I think this could work quite nicely.

Actually, actually...

I've changed my mind again. We're going with the `useTransactionsDateRange` hook (except now it's called `useDateRangeTransactions`) and we're keeping it separate from `useFilteredTransactions`.

`useFilteredTransactions` will have the role as the hook that the Transactions list/table uses themselves to filter whatever transactions they receive.

`useDateRangeTransactions`, since it connects to the store, is used at the scene level to replace the `connect` of the Transactions scene; as such, now the entire scene gets wrapped in a `DateRangeProvider`.

I decided to go with this setup instead because I wanted to maintain the ability to pass transactions in as a prop to the table/list. Not only because I was really lazy and didn't want to rewrite the stories, but also because I think it gives us a level of flexibility that we'll appreciate in the future.

Speaking of the future, there are two more 'filtering' type transaction 'hooks' that will be coming up soon: type filtering and search filtering.

Type filtering falls in the `useFilteredTransactions` camp; since it just filters a list without accessing the store, it can be used at the list/table level.

However, search filtering falls in the `useDateRangeTransactions` camp; it'll be used at the scene level and have its results merged with those of `useDateRangeTransactions` (likely via a literal union of the two sets) before having the transactions passed down to the table/list.

Date Range Picker

I think the most noteworthy about this component to discuss is the `CustomFormatDateInput`. Basically, since we've decided that we're going to be using regular `input[type=date]`, well we can't specify a custom display format for it.

So what I've decided is that we'll make it ourselves! The `CustomFormatDateInput` will show our custom formatted date just as a regular text field when it's not focused, but when it does become focused, it changes into an `input[type=date]`, so that users can then change the text. When it becomes unfocused, it just goes back to the custom formatted date.

Simple, elegant solution that keeps us from having to use a custom date input.

Component Breakdown

Atoms

- IntervalSwitcher (the container around the PaginationSwitcher and the date switcher with the side chevrons)
- SelectInput (just a regular old `select`; used for the interval size picker on mobile)

Molecules

- CustomFormatDateInput (a date input that shows a custom formatted date when not focused, but a regular date input when focused)
- DateSwitcher (IntervalSwitcher + CustomFormatDateInput)
- DateRangeSizePicker (SelectInput [on mobile] + RadioGroup with custom buttons [on desktop])

Organisms

- DateRangePicker (DateSwitcher + DateRangeSizePicker + useDateRange hook)

Scenes

- [modification] Transactions (add DateRangePicker; integrate `useTransactionsDateRange` hook)

Tasks

- Build the `transactionsDateIndex` slice in the Redux store.
- Build the `useDateRange` Context hook.
- Build the `useDateRangeTransactions` Context hook.
- Build the `DateRangePicker` and all its subcomponents.
- Integrate the `useTransactionsDateRange` hook and `DateRangePicker` with the `Transactions` scene.