1 — Avoid Server Round-Trips

Moving as much processing as possible away from the database and application server and into the browser always results in better performance and ultimately a better user experience. 3 high level areas can help navigate the issue — *database operations, filtering and caching*.

Database Operations

Multiple requests for data from the browser kills performance. I decided to execute one logical query and return all the data I needed to display and interact with the timeline.

Solution: Execute one logical query, augment with additional information as needed and return a list to the browser….once.

Instead of multiple calls to the database for each child record I combined them into one call. When researching the best way to query multiple child records I went back to basics.

```
SELECT Account.Name,(SELECT Contact.Name FROM contacts), (SELECT Case.Subject FROM cases),(SELECT <Object>.
<Field> FROM <RelationshipName>) FROM Account
```

One logical query that joins the child objects will allow my solution to scale — when we need to query more records we just add the object to the join.

We also had a requirement to display key fields when a user hovers over each record on the timeline. Issuing a server call to retrieve detailed information from the database for each record would be super slow. Instead I'll use 2 techniques.

The first is to use the Lightning Data Service (LDS). I came across lightning-record-form. This allows me to specify a layout-type and record-id and salesforce automagically returns the fields and data I have access to and owns caching the data.

```
<lightning-record-form
    record-id="001XXXXXXXXXXXXXX"
    object-api-name="Account"
    layout-type="Compact"
    columns="2"
    mode="readonly">
```

The second is to allow an administrator to specify a field to use for the tooltip in a custom metadata type. I've designed for this so that for objects not supported by the UI API it will fallback to using a fields value that I have already retrieved in my original database query. It will ensure the hover events are super snappy.

Filtering in Browser

Our requirements asked for the ability to filter records displayed on the timeline. We could issue a query for each request but this would have an associated delay as it made its way to the application server — then the

database and back again.

Solution: To avoid a server roundtrip to query for the data each time we ensure all filtering is done client side. This will mean our filter will be applied almost instantaneously.

D3.js has filtering capabilities built in. Let's use them.

Caching

The timeline component uses images, JavaScript libraries, CSS and database records. Where possible these need to be cached to reduce load and render times. It's important to review the options available to your use cases.

Solution: Salesforce provides a number of ways to cache information. I have chosen the best fit for each of my use cases below.

Topic	Use Case	Options	Decision
Display Images	Images used in the timeline	a) Use Content Delivery Network	Yes. Use Salesforce CDN
Cache Static Data	Data retrieved to determine which records to plot	a) Use Platform Cache	No. Will use a custom metadata type
		b) Use a Custom Metadata Type	Yes. Cached on first query.
Cache Dynamic Data	Data retrieved about individual records	a) Use Platform Cache	No. Data is updated regularly
		b) Use Lightning Data Service	No. We're retrieving multiple child records across objects
		c) Use Custom Cache	No. Data is updated regularly
		d) Don't cache	Yes. We'll use an Apex imperative callout but don't need to cache the data
JavaScript and CSS	Third party scripts and CSS	-- Only one option	Yes. Use static resources and minify.

Caching options and decisions

2 — Browsers have limits

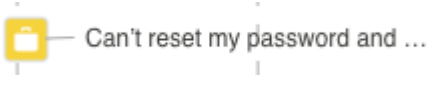
Avoid Large Complex Layouts

D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. Our original design had each record needing multiple SVGs to plot its position (5 in total) and for the bottom axis it requires on SVG per record too. If we want to scale to 3,000 records then that's a lot of SVGs. I would almost certainly run into performance issues.

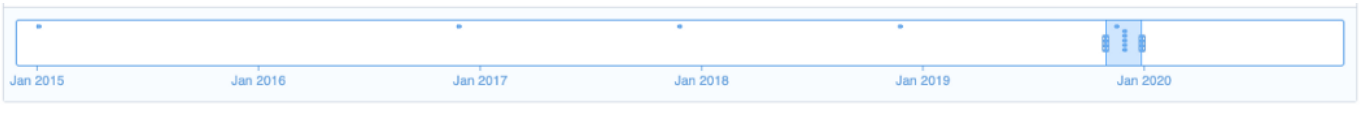
Solution — Reduce the number of elements to a minimum and aggregate records for the timeline map instead of plotting individual records

---

Bad: This looks great but each shape adds to the complexity of the layout



Better: A compromise. It removes unnecessary shapes.



Bad: Plotting individual records will require a large complex layout for large numbers of records

Better: Aggregating records by week reduces complexity for large data volumes (one bar per week vs one circle per record)

Avoid Layout Thrashing and Forced Synchronous Layouts

When you change styles the browser has to check to see if any changes require layout to be calculated. Running style calculations and layout synchronously and earlier than the browser would do normally are potential bottlenecks.

```
function resizeParentDivsBasedOnTimeline() {

  // Puts the browser into a read-write-read-write cycle causing layout thrashing.
  for (var i = 0; i < parentDivs.length; i++) {
    parentDiv[i].style.width = timeline.offsetWidth + 'px';
  }
}
```

Given that the D3.js library will require me to manipulate the DOM, change height and colours at runtime and position tooltips relative to parent. I will likely run into this issue.

Solution — Read style values prior to writing them.

```
// Read timeline width first.
var timelineWidth = timeline.offsetWidth;

function resizeParentDivsBasedOnTimeline() {
  for (var i = 0; i < parentDivs.length; i++) {
    // Now write.
    parentDiv[i].style.width = timelineWidth + 'px';
  }
}
```

3 — Salesforce has limits

Resources for the Salesforce Lightning Platform are shared between customers. In order to make sure each customer is using their fair share of resources Salesforce applies allocations (or governor limits). Typically these are generous in nature but you can hit these limits if you don’t research your approach up front.

I recommend reviewing the documentation and listing the limits that may apply to your use case.

For the timeline component the most likely limits I could hit would be per-transaction limits related to database operations. It’s best to review the ones that you think will apply to your design and design to mitigate potential issues.

Limit	Risk	Mitigation
Total number of SOQL queries issued: 100	SOQL query to retrieve more than 99 child records will throw an exception	For now it's unlikely this will happen. Ideally we could programmatically check the total number of child records and display an error
Total number of records retrieved by SOQL queries: 50,000	SOQL query to retrieve records across all child objects could exceed 50,000 and throw an exception	We plan for up to 3,000. Ideally we could check for the total number of records returned and display an error
Maximum SOQL query run time before Salesforce cancels the transaction: 120 seconds	SOQL query to retrieve records could exceed 120 seconds and throw an exception	Review the query plan and indexes used by default

Governor limits and mitigation strategies. Make sure you've considered them in your designs.

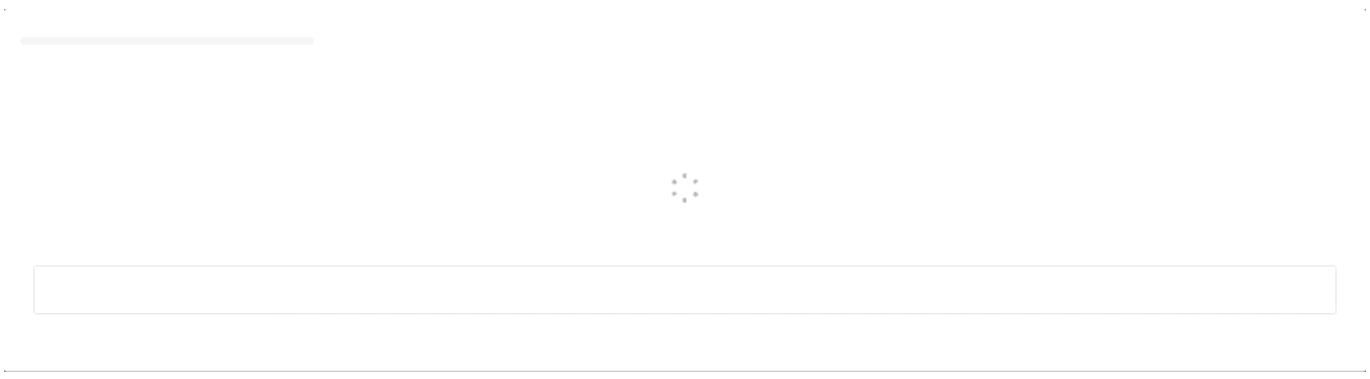
Solution: Review the documentation and ensure your code adheres to best practices. Refactor code where necessary.

4— Avoid blank screens

In the last post we covered the proposed user experience. What we didn't cover is that a blank screen — typically whilst your component is rendering — doesn't provide a great first impression.

Skeleton screens (aka stencils in the Lightning Design System), when used to indicate that a screen is loading, are perceived the screen as being shorter in duration when compared to a blank screen. Whilst it doesn't actually make your component load faster….it will make it *seem*faster.

Solution — I'll use the styles suggested by LDS to indicate my component is loading. Since the component look and feel is bespoke I'll keep it simple.



Stenciled timeline design shown when loading

Summary

Lightning Web Components have a number of resources and techniques you can use to tune performance. Since performance is critical to the user experience it needs as much attention to detail as the user interface design.

Thinking about scalability prior to writing code will help you anticipate challenges and allow you to mitigate them