Performance & Scalability in an Agile World

By Jason Carlson
CTO, LiquidPlanner

Copyright ©2010 LiquidPlanner, Inc. All rights reserved. LiquidPlanner, the LiquidPlanner logo, and all other LiquidPlanner names and slogans are trademarks of LiquidPlanner, Inc. in the US and/or other countries. All other trademarks belong to their respective owners.

When developing software there are constant decisions and trade-offs being made. often summed up by the glib, but frequently heard phrase:

> *"You can have it **Fast**, **Cheap** or **Good** - choose any **2**"*

This typically refers to the project triangle relationship between time, cost and scope. For a startup company with limited finances (think bootstrapping or angel investors), the priority has to be on time and cost, meaning that scope is the natural area for cutbacks, leading to many startups choosing to build it *"Fast, Cheap and **Good-Enough**"*.

In the software world, this fits in well with an agile development process, where short iterations allow us to build fast and cheap, while scope is cut back to a minimal viable feature set that is ***good enough*** to ship to customers, elicit feedback, and provide a base for further iterative releases. While cutting scope to a minimal viable feature set is necessary, it is important to maintain the quality of each release. Techniques such as automated testing and test driven development are typically used to ensure that while the feature set may be small, it maintains a high quality bar.

Within this agile environment, in addition to cutting back feature scope, we typically want to postpone dealing with performance and scalability issues until absolutely necessary.

> *"The First Rule of Program Optimization: Don't do it. The Second Rule of Program Optimization (for experts only!): Don't do it **yet**." - Michael A. Jackson*

There are reasons for this at many different levels.

- Avoid wasting time optimizing areas that are not bottlenecks
- Avoid wasting time optimizing features that the customer doesn't want.
- Avoid wasting time optimizing features that will change considerably in the near future.
- Avoid wasting time optimizing a product that might not survive in the marketplace.

While you could spend time and resources identifying the bottlenecks to avoid the first issue, the latter issues cannot be identified until after you take your product to market and therefore you want to postpone these until **just before** they become problems for your customer base[1].

This article describes an example of one such scalability issue that emerged while developing our online project management web application liquidplanner.com.

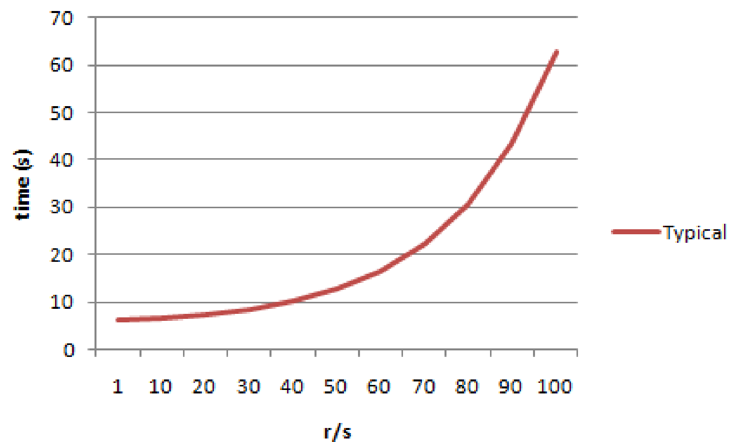
[1] Of course, this is a broad generalization, there will always be instances where performance and scalability are #1 priorities (e.g. huge sporting events like the Olympics) but these are the exceptions rather than the rule.

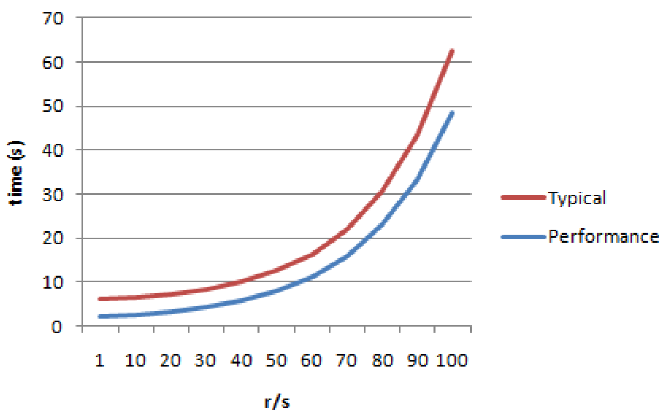
Performance vs. Scalability

Let's take a brief timeout to differentiate between performance and scalability, because they are two related but different aspects of web applications. I like to think of it this way:

- **Performance** - how fast your application responds to a user (e.g. page load times)
- **Scalability** - how is performance impacted as the size of your application (e.g. users) grows[2].

The most common performance/ scalability measurements of a web application is page response time and requests per second. The following graph shows a hypothetical application that serves up pages in the 5 to 10 second range at low loads, but as the number of requests per second grows beyond about 40 the response time climbs exponentially.

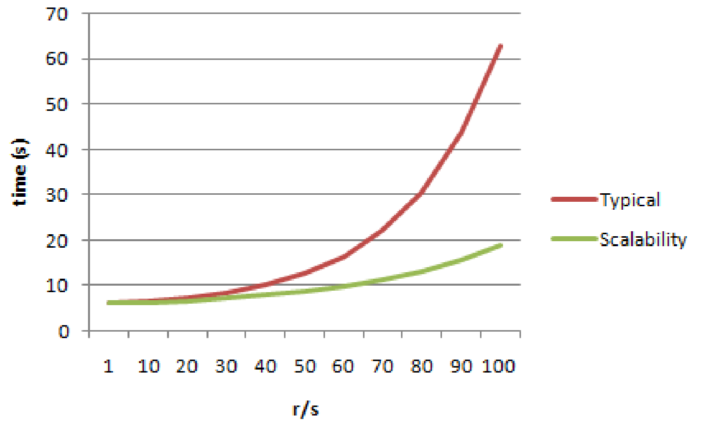




If we were to profile our pages and do some performance optimizations we might move that curve down. Improving our page response times from 5 seconds to 1 second is fantastic, but without tackling scalability issues this only benefits us under light load, under heavy load, the improvements are less impressive. Yes 20 seconds is less than 30 seconds, but both are unaceptable.

[2] There are also scalability issues with a products design, for example, large flat lists become unwieldy and need grouping mechanisms, but for the purposes of this article we are focused on scalability issues with performance.

So tackling scalability issues, in addition to pure performance optimizations, helps us reduce the gradient of the performance curve and move it to the right.



LiquidPlanner's First Scalability Issues

LiquidPlanner is online project management software. In addition to many other features, the core of our product is a hierarchical grid of projects and tasks along with their schedules:

Clients, Projects and Folders	Owner	Remaining Effort	hours	Nov Dec Jan 2011 Feb Mar Apr
Quantum FX	Charles	[4863h - 5192h]	5028h	
Big Fish Client	Jake	[217h - 293h]	255h	
Super SliderZ	Jake	[217h - 293h]	255h	
US Team	Jake	[117h - 177h]	147h	
SliderZ: Character Design	Frank	[32h - 72h]	52h	
SliderZ: Level Maps	Charles	[3h - 11h]	7h	
SliderZ: UI Dev	Jake	[24h - 56h]	40h	
SliderZ: UNIT Testing	Jake	[32h - 64h]	48h	
European Team	Jake	[85.02h - 131h]	108h	
Little Fish Client	Jake	[1124h - 1282h]	1203h	
Paint Ball Shooter	Jake	[294h - 378h]	336h	
Lunar Ballons	Liz	[294h - 378h]	336h	
Gummi Gems	Jake	[294h - 378h]	336h	
Snow Crash	Charles	[165h - 225h]	195h	
Red Fish Client	Jake	[2539h - 2785h]	2662h	
iBeerPong V3	Jake	[294h - 378h]	336h	
Jungle Rumble	Mary Ellen	[352h - 464h]	408h	
Logic Blox	Jake	[367h - 467h]	417h	

Although this is a web application, we needed to provide desktop like functionality (e.g expand/collapse, fixed columns, resizable columns, right-click context menu's etc) that are way beyond the scope of a normal HTML table. Therefore we built our own grid control that is a combination of server side (Ruby on Rails) code and client side html & javascript.

This was one of the first components that had to be in place for our minimal viable feature set to take to customers, and, being an angel-funded startup, needed to be built "fast, cheap and good enough", where **good enough** meant functional and high quality - but not necessarily fast or scalable.

Like all good agile startups, we built the simplest thing possible. Load, from the DB, all the items that match the current filter[3] and are in expanded containers, generate a HTML table with one row for each item and stream the whole thing back to the browser to render. This was simple, fast and cheap to code and worked great if the total number of items was kept under, roughly, 500 items. Beyond that loading more items from the DB was slow, generating larger and larger tables was slow, streaming all that HTML back to the client was slow, and browsers themselves cannot deal very well with huge complex tables that have hundreds or thousands of rows. Our performance characteristic was, at best, O(n), put simply, the more items that needed to be in the table, the slower it was.

Under certain circumstances, this can actually work ok for a little while. At any given point in time, the user probably only cares about one little part of the project plan. If our product design could guide users to filter aggressively, or drill in to a single project within the workspace, or keep most containers collapsed, then we can keep the total number of items needed in the grid down to a reasonable amount. Internally, we use our own product for our project planning, and we have over 10,000 items in our workspace, but since we have learnt to filter aggressively we rarely, if ever, need to look at more than a couple hundred at any one time.

However, this relies on trying to dictate user behavior. Not everyone wants, or likes to filter. We built a product that looks and feels a little bit like a desktop application, it's entirely reasonable for users to want to be able to expand all and see their entire plan in all its glory. If we did not want users to "expand all" then we should not have provided a button that lets them do it!

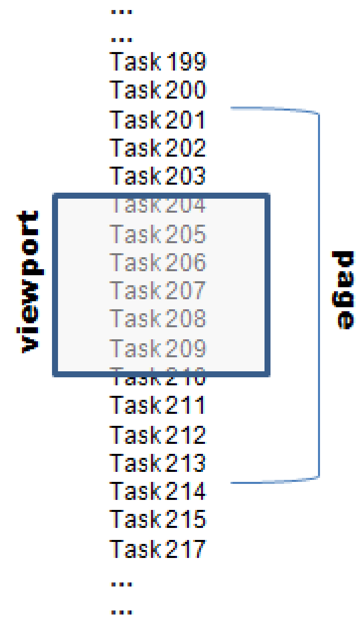
Asking users to filter appropriately is not a long term solution, and as our customers grew their workspaces larger and larger it became apparent that we had reached that time where this particular scalability issue was a problem for our larger customers and another approach was needed.

LiquidPlanner's Virtual Grid

The most obvious approach to creating a scalable grid is to recognize that, even if the user is looking at a grid with 10,000 items, their monitor has a limited physical presence and even the largest of LCD screens is only going to have room to display 50, 100, maybe 200 rows at once. If we call this the **viewport**, we can imagine a **virtual grid** that only downloads and displays the visible rows, loading more rows only when necessary as the user scrolls the viewport.

[3] LiquidPlanner has a filtering mechanism to allow the user to focus on only those items they care about, e.g., within a specific project, or assigned to a specific owner, or between 2 dates, etc.

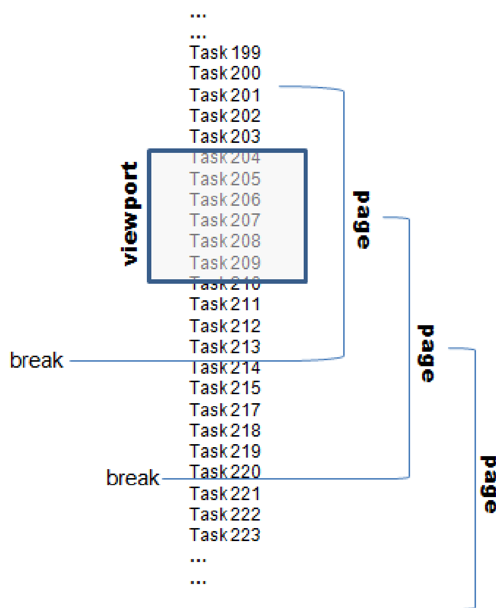
We know that going back to the server to load more rows is not going to be instantaneous, and we don't want the user to get frustrated with a stop & start jerky scrolling experience so we want to load slightly more rows than are actually visible in the viewport. Ideally just enough for the user to scroll around in the area they care about, but not so much to cause the slow page load problem we are trying to solve. We call this the **current page**.



One possible approach to a virtual grid is to add more rows to the top or bottom as the user scrolls close to the edge of the current page. For LiquidPlanner this approach has 2 basic problems.

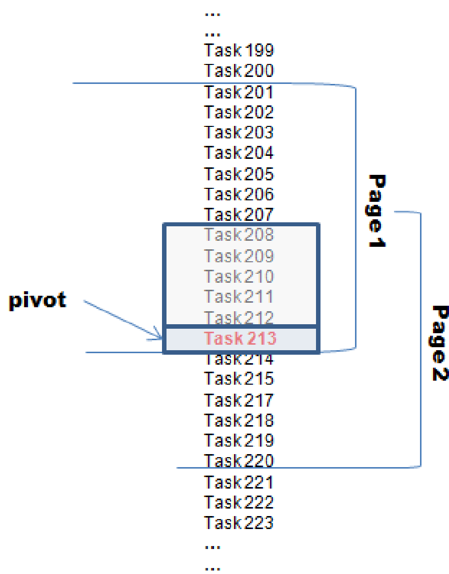
- Continuously adding more rows to the HTML table will eventually cause the browser to slow down and become sluggish. Ideally we would like to maintain a fairly constant number of rows loaded, as the user scrolls down we add more rows to the bottom and remove older rows that are no longer visible from the top.

- LiquidPlanner is a multi-user system. By the time it takes for a user to view the current page and decide to scroll down further, other users of the system may have re-organized the hierarchy. If we leave the old rows in place and simply add new rows we are likely to end up displaying an inconsistent view of the user's project plan.

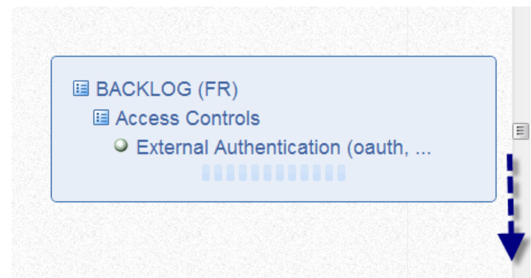


In order to handle both these issues, the LiquidPlanner virtual grid waits until the user scrolls down to the edge of the current page (we call this a **page break**) and then swaps in an entirely new page centered around the last item that was visible in the previous page. This overlapping page mechanism allows the user to scroll back up a fair amount before hitting another page break.

The key to this page swapping algorithm is known as the **pivot item**. As the user scrolls the viewport down and approaches the bottom of page 1, we find the last item in that page and it becomes the pivot item which will be the center of the newly loaded page 2.



When scrolling with the mouse wheel or the up/down arrow keys this page-by-page pivot scheme works well, but what happens if a user is currently viewing items near the top of the grid and needs to scroll down all the way to view items at the bottom? We don't want to make them sit through hundreds of small page breaks loading each page in between. For this reason scrolling with the scrollbar itself requires a slightly different mechanism. Instead of swapping in each page as the user scrolls past it, a small preview window is displayed near the scrollbar thumb indicating, roughly, where you are in the larger project plan. The actual final page is not swapped in until you stop scrolling and let go of the scrollbar thumb.



Loading the right page from the DB

The LiquidPlanner grid represents a project plan hierarchy. There are a number of ways to manage a hierarchy of items in a SQL database. We chose to use postgres ARRAY columns to indicate the items position in the tree (priority) and the items ancestors, e.g.

id	name	priority	ancestors
400	Project X	[1]	[ ]
401	Task X1	[1, 1]	[400]
402	Task X2	[1, 2]	[400]
403	Project Y	[2]	[ ]
404	Task Y1	[2, 1]	[403]
405	Task Y2	[2, 2]	[403]
406	Sub-Project	[2, 3]	[403]
407	Task Y3	[2, 3, 1]	[403, 406]

Given this schema we can load the hierarchy in the correct order, ignoring children of collapsed containers using the postgres <@ subset operator:

```
select *
  from items
 where ancestors <@ ARRAY[ ids of expanded containers ]
 order by priority
```

If we want to load a filtered view of the hierarchy, we include some additional query clauses[4]

```
select *
  from items
 where ancestors <@ ARRAY[ ids of expanded containers ]
   and some additional query
 order by priority
```

How do we go from loading the entire hierarchy to loading just the current page ?

- **Load only the ID** of each item (matching the filter and expanded) in hierarchy order.
- Find the index of the pivot item
- Calculate how many items above & below the pivot are required to make a complete page.

[4] It's actually much more complex once you filter because you must deal with the possibility that a child task matches the filter but its ancestor container does not. Solving that problem involves a complicated subquery that is beyond the scope of this article.

- **Load the full model** for only those items in the page.
- Generate the HTML table structure for only those items in the page.

The first step sounds like a potential bottleneck since we may still be returning thousands of records, but since we are only returning the ID's for each record, and it can be done in a single optimized query with appropriate indices it can be very fast. Handling an array of thousands of integers can be done extremely fast. The important part is that we quickly cut that number down to a sensible amount **before** we need to load full DB models, build up any HTML or stream anything back to the client.

Given an array of item ID's loaded with a query like those described above, we can cut down to the current page with an algorithm like this:

```
def get_page(items, pivot, size = DEFAULT_PAGE_SIZE)

  index = items.index(pivot)

  # calculate number of items above (ulimit) and below (dlimit) pivot
  ulimit = (size/2) - 1
  dlimit = (size/2)

  min_index = 0
  max_index = items.length-1

  min = index - ulimit
  max = index + dlimit

  # if there is not enough to show above/below the pivot, then take the difference
  # and add it to the other end (trying to keep the page size constant)
  if min < min_index
    max = max + (min_index - min)
    min = min_index
  elsif max > max_index
    min = min - (max - max_index)
    max = max_index
  end

  # be sure to bound up and down limits by min/max indices of the items array
  min = [min_index, min].max
  max = [max_index, max].min

  # put together some helpful information for the caller
  return {
    :items => items[min..max],                           # PAGE OF ITEMS
    :above => min,                                       # number of items above the page
    :below => (items.length - (max + 1)),                # number of items below the page
    :prev  => (min > min_index) ? items[min-1] : nil,    # last item on previous page (if any)
    :next  => (max < max_index) ? items[max+1] : nil     # first item on next page (if any)
  }

end
```

A real implementation would have much more to consider:

- what if the pivot was not found in the array of items ?
- what if the total number of items is only slightly larger than the page size ? are we going to force a page break or can we have a "greedy" algorithm that enlarges the page size
- what if I want the pivot item to be the top or bottom of the page instead of the middle ? This might be useful if we switch to a UI that has the user clicking "next" or "prev" links in order to jump over the page break boundaries.
- ... and more ...

Fine-tuning the page size

So, the virtual grid is swapping in overlapping pages at appropriate moments. How large should each page be? The ideal page size will be small so that it loads fast, but should also be large enough to minimize the number of page breaks that occur under typical use. We knew from the existing grid implementation that we would need to keep the page size below, about 500 items, and from observing our typical usage we, subjectively, felt that 200 items was enough for most common use cases to minimize the impact of the page break loading delay. Ultimately this will require further fine-tuning as we learn from our customers, we will measure page load times and number of page breaks being hit and, subjectively, try to find a healthy balance between the two.

Conclusion

At LiquidPlanner we have to tackle issues of performance and scalability from many different angles, we minify our javascript files, we gzip our pages and edge cache them on the Akamai network, we analyze our DB query plans, we optimize our CSS sprite images and profile our ruby on rails actions, and on and on... There is a lot that goes into making a web application performant, and we would be the first to admit that we still have a long way to go to meet user expectations in this "Google instant" world.

Historically, software architecture tends to swing like a pendulum between centralized and distributed paradigms. From mainframes, to native desktop applications, to client server, to the web, each swing of the pendulum seems to be smaller and destined to unify the power and performance of native desktop applications with the distribution and management benefits of web applications[5]. Tackling performance and scalability issues will continue to be part of that evolution.

[5] HTML 5 and related technologies being a stepping stone towards that unification.