Introduction

Pique is an automated engine for gathering, sorting, and displaying trending web content. It is designed to streamline
content consumption, allowing users to access top and trending content from multiple social media platforms quickly.

Content is queried from multiple sources and streamed into a Redis data queue, sorted by their source location. Each
post is assessed based on the number of different types of interactions, such as likes, comments and shares and given a
“popularity score”. The posts from different all sources are then sorted according to their popularity scores and
velocity (the change in their score over time) and placed back in a different Redis queue, in order and ready for
consumption. The front end is accessed as a web app and simply sends GET requests to our back end which serve up the
data from the appropriate queue.

Verification Strategy

Our product is being developed to meet a perceived need based on our understanding of the way that many users interact
with social media. This is the need to be able to quickly view the most relevant (top, trending, or hashtag-related)
posts on social media at that moment. In order to ensure that our product is actually meeting this need for the users,
we will need to get regular feedback from our targeted user base regarding the usability and helpfulness of our product.
This will include feedback on the content we’re providing (is it the content they want to see) as well as how easy and
intuitive our site is to use.

Our user base is anyone who uses social media, so an easy way for us to get feedback will be to ask our friends and
classmates to look at our product and see what they say. Since we want to get feedback regularly and early in the
development process, there are many stages at which we will seek this feedback:

i. At the beginning: when we tell them about our idea for the product and how it will work, do people think that this
   is something they would use? Which social media sites would they like to see posts from on our site? Is there an
   existing service that meets this need, and what do they like about that particular product?

ii. With mock-ups: do they like the way the website looks? Do they understand how they would navigate the site naturally,
   or do they need instructions on how to navigate the site?

iii. With an initial UI prototype: is it easy to understand the content they’re seeing? Is anything

confusing to them? Are
the main features of our product (searching between top, trending, and hashtag-related posts) immediately obvious to
them when they go to the site?

iv. With the full product: is the content relevant to what they want to see? Is there a good mix of content that they're
interested in? Is anything missing?

Hopefully receiving and using this feedback from our targeted user base during development will ensure that our final
product is successful at meeting the needs of social media users.

Non-functional Testing & Results

A list of the Pique's non-functional requirements, as outlined in our Requirements & Use Cases document is listed
below, followed by a test plan to address each.

Performance Requirements

i. Pique must provide users with content access at a lower latency (from content creation to consumption) than accessing
each individual social media site for the same trending content

ii. Pique must be significantly easier to use than browsing multiple social media sites simultaneously

iii. Database Access Object must respond to all queries within 0.1 seconds during normal operation

iv. The Pique landing page must respond and load within 0.5 seconds of a request, barring latency due to legacy client
hardware

v. The Pique data cache must not exceed a reasonable storage limit

Performance requirements by nature are dissimilar from pass/fail unit test scenarios and are not quantifiable in the
same way. We feel that trying to constrain subjective or scale outputs such as those described above to Bernoulli Trial
cases is counter productive and have therefore elected to describe them in a different way.

Our first two performance requirements are subjective goals dealing with ease of use and user experience. That is, users
must feel as though our product is easier to use and that they gain some sort of benefit of using it. Therefore the
testing procedure for these is similar to steps 2-4 of the verification strategy outlined above. We can gather
information from a few different types of scenarios, document the results, and use them to optimize our product to
meet the goals.

		Outcome

Scenario	Reasoning	Notes
Full walkthrough of product	Gathers feedback on **missing** desired features	
Guided walkthrough of product	Gathers feedback on **difficult to use or find** features	
“Hands off” demo (let users walk themselves through the product)	Gathers feedback on **ease of learning** the product.	

The remaining three non-functional requirements are purely related to the desired performance of the system. While real
goals, the product can not be written off as working or not working based on their results, unlike pass/fail
functionality tests. Rather, following the adage “First we make it work, then we make it beautiful, then we make it
fast”; we test a complete and correct system to see if it meets our performance goals, and make decisions based on the
results. For these goals we will do iterations upon our completed product, following a format similar to as follows:

Iteration	Data Access (ms)	Page load time (ms)	Cache Size (MB)	Other Goals (unit)
1	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:
2	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:
…	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:
n	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:	Worst:Best:Avg:

Where upon iteration n we have decided that our performance either meets our goals or cannot reasonably be improved
further within the project budget constraints.

Software Quality Attributes

i. We will need to keep in mind adaptability for adding future media sources that Pique will pull from, or removal of
existing social media sources from Pique’s lineup

ii. The Pique landing page will remain available 99% of the time during operation

iii. Pique source code will remain fully tested and maintainable with every release, ensured via peer code and test suite
review

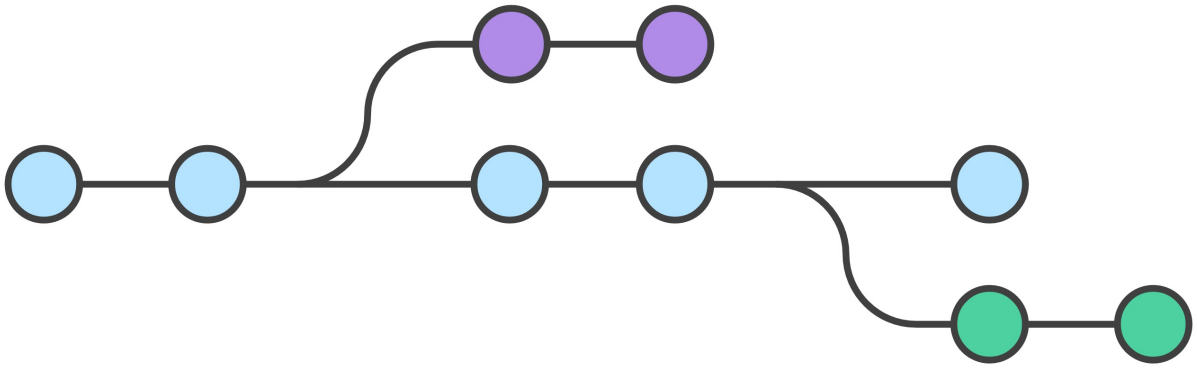
iv. Pique source code will remain intrinsically portable on the JVM

Although media source adaptability was addressed in the system design, we will need to ensure that the system remains
adaptable throughout Pique’s iterative development lifecycle. This is addressed both with a well-

documented design
process (so that a developer or tester knows to maintain adaptability), and an intensive version control workflow and
code review process (to ensure that one developer's implementation does not break from this design), which is detailed
below.

We've decided to maintain a feature branch version control workflow throughout Pique development. This allows us to
deter defects from reaching the master branch, and encourages frequent testing and code review.

An invariant of our particular process is that not one developer can commit code directly to master. The developer must
create a feature branch off master, and then create a pull request to merge changes into master. Before the pull request
can be merged, however, the feature branch must be up to date with master, be reviewed and approved by a peer, and pass
all tests in the automated test suite.



From https://www.atlassian.com/git/tutorials/comparing-workflows/centralized-workflow

While the application will be intrinsically portable on the Java Virtual Machine, the development, automated build
process, and system tests we've planned will ensure that Pique can successfully run on a multitude of machines,
including workstations, dedicated servers, and temporary virtual machines.

Functional Testing Strategy

Throughout development, we will perform unit, integration, and system tests. Testing tools provided within the Play
framework allow us to create integration tests to run alongside unit tests, all of which will be run as part of our
deployment test suite. This test suite can be run manually within a development environment, but will also be run with
each push to our remote Github repository on all branches, and when a pull request is made to pull a branch into
master in pursuit of Continuous Integration.

At least once a week, the latest master build will be deployed to a remote test environment, on which we will perform
longer-term system tests. A fully documented load test at production scale will be performed before release, so that we
can understand the limits of our system within the particular production environment.

Testing Approach

In order to improve software testability, Pique software is designed and developed with the SOCK model for testability
in mind. The SOCK model is defined as follows:

- **S**imple: Simple components and applications are easier (and less expensive) to test.
- **O**bservable: Visibility of internal structures and data allows tests to determine accurately whether a test passes
  or fails.
- **C**ontrol: If an application has thresholds, the ability to set and reset those thresholds makes testing easier.
- **K**nowledge: By referring to documentation (specifications, Help files, and so forth), the tester can ensure that
  results are correct

From *"How we Test Software at Microsoft"*, Page, Johnston, and Rollison

Following this model will ensure that Pique testing is simple, streamlined, that testers create meaningful tests, and
that we receive the maximum value from our testing procedure as a result.

Test cases will be created using a medley of functional test methods, including equivalence class partitioning, boundary
value analysis, and combinatorial analysis wherever appropriate. Equivalence class partitioning allows us to break down
a large number of potential tests into a subset of meaningful test cases with the same amount of coverage, reducing test
redundancy. Boundary value analysis will help us to understand whether methods handle edge cases correctly; we will
stick to the 3*(BC) model — in which each boundary, as well as its left and right neighbors are all tested — wherever
possible. In the case that a method has multiple inputs, combinatorial analysis will allow us to pair down test inputs
so that we maintain a manageable number of non-redundant test cases.

Bug Tracking

In order to streamline development efforts, bugs will be tracked as issues within the Pique Github repository. Bugs are
evaluated on two criteria: severity and priority, each with four distinct levels, defined as follows:

Severity		**Priority**	
Critical	All is lost. This bug breaks everything.	Urgent	This bug needs to be fixed. Yesterday.

Major	The system *technically* operates with this defect, but remains unusable in some major facet.	High	This defect must be resolved as soon as possible.
Minor	This defect affects minor functionality, and is relatively simple to fix	Medium	This can be fixed throughout the regular course of development.
Trivial	This issue does not affect functionality whatsoever, merely an inconvenience or light cosmetic change.	Low	This defect should be resolved, but can be deferred until higher priority work is completed.

Normally (but not always) higher severity defects will correspond to higher priority, and vice versa.

Tasks and issues throughout Pique development is tracked using Github's internal issue tracker. When a task is
completed, the issue is tagged in the pull request. Another team member will review the code in the pull request for
various criteria, including the fulfillment of each issue tagged. Once the pull request is approved, and all unit tests
pass, the feature branch can then be merged into the master branch. This will automatically close the associated issues.
If the issues are not tagged, the reviewer will close the issue manually.

Adequacy Criterion

Testing of a component evolves as follows:

i. Unit test the individual component
ii. Integration test the component with each component with which it interacts

iii. Test the component as part of the whole system

iv. Unit Testing

v. Each method should have at least one test for a reasonable input, and one for each edge case input
vi. Negative cases are tested at this level
vii. Integration Testing
viii. Each behavioural interaction (determined by specifications) is tested, once for a reasonable input and once for
each edge case
ix. System Testing
x. Each use case path through the system has an automated test that mimics the UI requests to the backend
xi. Each use case path is tested end to end driven by actual UI interactions by manual testing
    i. A written guide should be produced that is designed to mimic the automated system tests substituting the mimicked
    UI actions for actual UI actions
    ii. Manual testing should follow the guide systematically and results documented, as to be

repeatable and verifiable

Rationale

The testing adequacy criterion were designed to maximize the efficiency and capabilities of testing at each level. Unit
testing is the quickest and should be done as to ensure a class behaves correctly for all reasonable or reasonably
predicted edge case inputs. Once this is complete, we can test interactions between components to ensure they interact
as we expect, according to their specifications. That is, we ensure that the components are not just functional, but
correctly functional. Finally, once all this is complete we run system tests which imitate use case paths through the
system in two ways. The automated component will not use the UI, but imitate the requests it would send in code. The
one and only manual component of our testing is the one that involves our UI, testing that it behaves the same way as
the automated tests that mimic its behaviour. The reason that we chose to do this is that the comparative cost for the
overhead of a UI test framework and code writing (and its inherent fickleness) versus the benefit on our simple UI was
deemed unrewarding.

Test Cases & Results

As a complete set of passing unit tests is required for a feature branch to be pulled into master, all unit tests are
passing.

API Testing

These tests will check that our APIs are correctly getting data.

Getting Content from Reddit

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Can authenticate	Oauth credentials	Authentication successful	Authentication successful	P	
2	Can retrieve bulk content	N/A	Content is retrieved and parsed into our format	Content is retrieved and parsed into our format	P	
3	Can store content retrieved	N/A	Posts are successfully added into the data tier	Posts are successfully added into the data tier	P	
	Can request					

4	certain post	Post ID	Post is retrieved	Post is retrieved	P	

Getting Content from Twitter

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Can authenticate	Oauth credentials	Authentication successful	Authentication successful	P	
2	Can retrieve bulk content	N/A	Content is retrieved and parsed into our format	Content is retrieved and parsed into our format	P	
3	Can store content retrieved	N/A	Posts are successfully added into the data tier	Posts are successfully added into the data tier	P	
4	Can request certain post	Post ID	Post is retrieved	Post is retrieved	P	

Getting Content from Imgur

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Can authenticate	Oauth credentials	Authentication successful	Authentication successful	P	
2	Can retrieve bulk content	N/A	Content is retrieved and parsed into our format	Content is retrieved and parsed into our format	P	
3	Can store content retrieved	N/A	Posts are successfully added into the data tier	Posts are successfully added into the data tier	P	
4	Can request certain post	Post ID	Post is retrieved	Post is retrieved	P	

Testing for Sorting Posts

These tests will check that the data we get from our APIs is getting correctly sorted into Redis queues.

Identifying and Sorting Top Posts

Test						

#	Purpose	Input	Expected	Actual	P/F	Notes
1	Data from multiple sources is sorted correctly according to score	Posts to be sorted	Posts are sorted in descending order by popularity score	Posts are sorted in descending order by popularity score	P	
2	Sorted data is stored back in the data tier	Posts to store	The posts are put correctly back into the data tier	The posts are put correctly back into the data tier	P	
3	Data below top popularity threshold is filtered out	Posts to be sorted with zero sum of comments, shares, and likes	empty list	empty list	P	
4	Sorted data contains posts with popularity score greater than threshold	Posts to be sorted	List of sorted posts contains only posts with popularity score greater than threshold	List of sorted posts contains only posts with popularity score greater than threshold	P	

Identifying and Sorting Trending Posts

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Data from multiple sources is sorted correctly according to velocity	Posts to be sorted	Posts are sorted in descending order by popularity velocity	Posts are sorted in descending order by popularity velocity	P	
2	Sorted data is stored back in the data tier	Posts to store	The posts are put correctly back into the data tier	The posts are put correctly back into the data tier	P	
3	Run trending sorter on empty input	empty list	no posts	no posts	P	

Identifying and Sorting Posts by Hashtags

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Hashtag sorter sorts posts by hashtags	posts to be sorted	Posts from multiple sources are sorted into respective hashtag channels	Posts from multiple sources are sorted into respective hashtag channels	P	
2	Hashtag sorter sorts posts by hashtags on empty input	empty list	Posts from multiple sources are sorted into respective hashtag channels	Posts from multiple sources are sorted into respective hashtag channels	P	
3	Store posts under hashtag channels	Map of sorted posts	postLists stored under hashtag channels	postLists stored under hashtag channels	P	
4	Store empty map of posts	empty map of posts	no posts sorted in hashtag channels	no posts sorted in hashtag channels	P	

Identifying and Sorting Top Hashtags

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Identify hashtag trends from stored hashtags	null	The top X hashtags are identified and sorted	The top X hashtags are identified and sorted	P	
2	Identify hashtag trends from stored hashtags	empty list	The top X hashtags are identified and sorted	The top X hashtags are identified and sorted	P	
3	Identify hashtag trends from specified hashtags	list of hashtags	The top X hashtags of those specified are identified and sorted	The top X hashtags are identified and sorted	P	
4	Identify hashtag trends with no stored hashtags	null	empty list	empty list	P	
5	Identify hashtag trends	empty	empty list	empty list	P	

	with no stored hashtags	list				
6	Load top hashtags with none currently stored	list of top hashtags	new list is loaded in data store	new list is loaded in data store	P	
7	Load top hashtags	list of top hashtags	new list replaces old list	new list replaces old list	P	
8	Load top hashtags with empty input	empty list	old list is cleared	old list is cleared	P	

Running the sorting node

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Run the sorting node in full	none (posts exist in source namespace)	pages exist in stringlist, hashtag, and display namespaces	pages exist in stringlist, hashtag, and display namespaces	P	
2	Run the sorting node with number of available posts under threshold	none	sorting node exits and does not sort anything	sorting node exits and does not sort anything	P	

Calculating Popularity Score & Velocity

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Calculate popularity score of post and rebuild	Post with no popularity score	Post has been rebuilt, now with popularity score field	Post has been rebuilt, now with popularity score field	P	
2	Calculate popularity score of post past lower score bound	Post with negative sum of comments, likes, shares	Post has been rebuilt, now with popularity score bounded to 0	Post has been rebuilt, now with popularity score bounded to 0	P	
	Calculate					

3	popularity score of post at lower bound	Post with zero sum of comments, likes, shares	Post has been rebuilt, now with popularity score of 0	Post has been rebuilt, now with popularity score of 0	P	
4	Calculate popularity score of post near lower bound	Post with near zero sum of comments, likes, shares	Post has been rebuilt, now with positive popularity score near 0	Post has been rebuilt, now with positive popularity score near 0	P	
5	Calculate popularity score of post past upper bound	Post with comments, likes, shares, all set to integer max value	Post has been rebuilt, now with popularity score bounded to integer max value	Post has been rebuilt, now with popularity score bounded to integer max value	P	
6	Calculate popularity score of all posts in list	List of posts without popularity scores	Each post has been rebuilt, now with a popularity score	Each post has been rebuilt, now with a popularity score	P	
7	Calculate popularity score of all posts in empty list	empty list of posts	empty list of posts	empty list of posts	P	
8	Calculate popularity velocity of a post and rebuild	Two posts, with the newer having a higher popularity score	New post has positive popularity velocity with correct magnitude	Positive popularity velocity with correct magnitude	P	
9	Calculate popularity velocity of a post and rebuild	Two posts, with the newer having a lower popularity score	New post has negative popularity velocity with correct magnitude	Negative popularity velocity with correct magnitude	P	
10	Calculate popularity velocity of a post and rebuild	Two posts with the same popularity score	New post has popularity velocity of 0	New post has popularity velocity of 0	P	

Preparing data for use by or produced from sorter classes

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Prepare postList from list of posts	list of posts	List of posts is converted to postList object	List of posts is converted to postList object	P	
2	Prepare postList from empty list of posts	empty list of posts	Empty list of posts is converted to empty postList object	Empty list of posts is converted to empty postList object	P	
3	Prepare pages from list of posts	list of posts with size greater than the number of posts in a page	List of posts is partitioned into (size / num posts per page) + 1 postList objects, in order	List of posts is partitioned into (size / num posts per page) + 1 postList objects, in order	P	
4	Prepare pages from empty list of posts	empty list of posts	empty list posts is converted to empty list of postlists	empty list posts is converted to empty list of postlists	P	
5	Expand pages into list of posts	list of postLists	List of postLists is converted to single list of posts, in order	List of postLists is converted to single list of posts, in order	P	
6	Expand empty list of pages into list of posts	empty list of postLists	empty list of postlists is converted to empty list of posts	empty list of postlists is converted to empty list of posts	P	

Testing for Storing Data

These tests will check that our post data is getting correctly stored to and served from Redis and its in-memory variant.
This set of tests is run on both the Redis access object and the In Memory Access Object.

Test #	Purpose	Input	Expected	Actual	P/F	Notes
	Add Post to					

1	empty channel	post	post is added to channel	post is added to channel	P	
2	Add Post to nonempty channel	post	post is added to channel	post is added to channel	P	
3	Add Posts to empty channel	list of posts	list of posts is added to channel	list of posts is added to channel	P	
4	Add Posts to nonempty channel	list of posts	list of posts is added to channel	list of posts is added to channel	P	
5	Pop Post from empty channel	channel name	empty optional	empty optional	P	
6	Pop Post from nonempty channel	channel name	post at front of channel is returned and now removed	post at front of channel is returned and now removed	P	
7	Add Lists of Posts to empty channel	lists of posts	lists are added to channel in order	lists are added to channel in order	P	
8	Check that posts are added in correct order	posts	posts are added to channel in order	posts are added to channel in order	P	
9	Get empty list of posts	channel name	empty list	empty list	P	
10	Get list of posts	channel name	list of posts	list of posts	P	
11	Peek at empty post list	channel name	empty optional	empty optional	P	
12	Peek at post list	channel name	post at front is returned, not removed	post at front is returned, not removed	P	
13	Add display postlist to empty channel	channel name, display postlist	postlist is added to channel	postlist is added to channel	P	

14	Add display postlist to channel	channel name, display postlist	postlist is added to channel	postlist is added to channel	P	
15	Add hashtag postlist to empty channel	channel name, display postlist	postlist is added to channel	postlist is added to channel	P	
16	Add hashtag postlist to channel	channel name, display postlist	postlist is added to channel	postlist is added to channel	P	
17	Get postlist out of current bounds	channel name, index out of bounds	empty optional	empty optional	P	
18	Get number of postlists in empty namespace	namespace	no postlists	no postlists	P	
19	Get number of postlists in namespace	namespace	number of postlists in that namespace	number of postlists in that namespace	P	
20	Get keys in namespace	namespace	number of keys in that namespace	number of keys in that namespace	P	
21	Get keys in empty namespace	namespace	number of keys in that namespace	number of keys in that namespace	P	
22	Delete N posts from list of posts	channel name, number of posts to be deleted	first N posts deleted	first N posts deleted	P	
23	Delete N posts from empty list of posts	channel name, number of posts to be deleted	no posts deleted	no posts deleted	P	
	Delete N >	channel				

24	size posts from list of posts	name, number of posts to be deleted	all available posts deleted	all available posts deleted	P	
25	Delete 0 posts from list of posts	channel name, number of posts to be deleted (0)	no posts deleted	no posts deleted	P	
26	Verify that post lists expire	post lists greater than number of allowed postlists	only newest max number of postlists remain	only newest max number of postlists remain	P/**D**	**D**eprecated due to long runtime in test enviroment
27	Add new hashtag post lists	channel name, post lists	postlists stored under channel name	postlists stored under channel name	P	
28	Replace hashtag post lists	channel name, post lists	postlists replace existing post lists	postlists replace existing post lists	P	
29	Replace hashtag post lists with empty list	channel name, empty list	no postlists remain in that hashtag channel	no postlists remain in that hashtag channel	P	
30	Add new top hashtag list	channel name, list of top hashtags	list of top hashtags added to channel	list of top hashtags added to channel	P	
31	Replace top hashtag list	channel name, list of top hashtags	list of top hashtags replaces old list of top hashtags	list of top hashtags replaces old list of top hashtags	P	
32	Replace top hashtag list with empty list	channel name, empty list	list of top hashtags is deleted	list of top hashtags is deleted	P	

Testing for Parsing and Displaying Data

These tests will check that our post data is getting correctly received from Redis and displayed on

the webpage.

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Posts are correctly retrieved	N/A	Post object is correctly retrieved	Post object is correctly retrieved	P	
2	Posts are correctly parsed	N/A	A parsed object's fields are all correct	A parsed object's fields are all correct	P	

UI Testing

These tests will check that our UI is working correctly.

Test #	Purpose	Input	Expected	Actual	P/F	Notes
1	Clicking on a post opens expanded view	Click on post	The post opens in expanded view	The post opens in expanded view	P	
2	Can click a hashtag to see more of that topic	Click on hashtag	Posts matching that particular hashtag are now shown	Posts matching that particular hashtag are now shown	P	
3	Can search a hashtag to see more of that topic	Search a tag in the search bar	Posts matching that particular hashtag are now shown	Posts matching that particular hashtag are now shown	P	
4	Can sort by top scoring	Click "Top"	Posts are shown and ordered in decreasing popularity score	Posts are shown and ordered in decreasing popularity score	P	
5	Can sort by fastest rising score	Click "Trending"	Posts are shown and ordered in decreasing popularity velocity	Posts are shown and ordered in decreasing popularity velocity	P	