
Table of Contents

Welcome to Fetchq	1.1
Yet another queue system?	1.1.1
Features you will enjoy »	1.1.2
Getting Started	1.2
Setup Postgres	1.2.1
Initialize Fetchq	1.2.2
Create a new queue	1.2.3
Push a document	1.2.4
Consume a queue	1.2.5
Glossary	2.1

Welcome to Fetchq

`fetchq` is a **queue system** that we think is easy to integrate in your existing project:

- `fetchq` is 70% a Postgres extension
- `fetchq` is 15% a Client library
- `fetchq` is 15% a REST server

We are using `fetchq` in production to handle queues up to **60 millions unique repetitive tasks** (web scraping) with throughputs of easily thousands of jobs / minute, in a **real life project** where we have to deal with unreliable HTTP requests, **horizontal scaling across 70 workers server** (real concurrent access) and the constant fight for ever changing APIs or our targets (which I'm not going to disclose here).

Yet another queue system?

Yes, indeed. When I feel the urge to write some code... I google first. And I did. A lot.

I tried different systems:

- [RabbitMQ](#)
- [AWS SQS](#)
- [celery](#)
- countless NPM modules

I experienced some problems...

My problem was simple: I had to scrape the *WebsiteXXX* which hosted *Iron Man*'s profile. And I had to scrape *Iron Man*'s info every second day because the guy's so cool and his profile changes a lot.

repetitive tasks was problem n.1

Every time my [worker](#) did scrape *Iron Man*'s profile, it found references to other super heroes: *The Hulk*, *Captain America*, *Black Widow*, ... and of course I wanted them too. Soon enough I noticed that **they were often referencing each others**, so duplicates begun to appear in my solution. Ouch! My system was doing the job more than once every second day!

unique tasks was problem n.2

(with RabbitMQ you need to use a Redis - or similar - server to check uniqueness before pushing)

Some time went by and my system did work well. Too well. So much "well" that **it founds millions of superheroes**. Damn, they are everywhere! Soon enough **my server could not cope with the amount of profiles it needed to check** every second day. There was not enough time (given *WebsiteXXX* is so slow). At this point I figured: "*why don't just start copied of my system on multiple machines?*" Best idea ever, easy to think, easy to do (just throw money at AWS, right?). Few minutes into horizontal scaling I noticed that my many machines were all processing the same super hero!

exclusive distributed tasks was problem n.3

(all the existing systems solved this problem quite well)

Once I solved the concurrent access problem I was set for success. Or so I thought. It was about Christmas time so my optimism was sky-rocketing. Well, well, well... After a quick check I found out that *Iron Man* removed his profile from *WebsiteXXX* but my system kept trying to access the profile, getting a non surprising 404 back. Then I thought: "*shouldn't the system be smart enough to stop trying?*" Of course it should be! But AI/ML is so expensive so I simply

task failure threshold was problem n.4

There have been many other small and big challenges in getting together this *Postgres* extension, performances are always on top of my mind and I am now investigating new ways of using *table inheritance* and *data partitioning* to create an even more efficient system.

But at its core *Fetchq* is simply a stable queue system that can run millions of documents in a very small small machine, with a memory footprint that is basically ridicolus.

Fetchq Features

Multiple distributed workers

The whole point of developing *Fetchq* was to scale a crawler app that I was working on.

My target website (often in this pages I refer to it as the "superheroes" website) did slow down their response time and I had to put more servers (with different public IPs) into the job.

I soon found out that by simply picking up a task from a sorted list was not enough. I needed some sort of exclusive access to it that was resilient to multiple concurrent access.

To date I am using *Fetchq* in production with ~90 machines pointing to it as single source of truth for the question "what do I do next?".

Endless (~) reliable storage

I admit I am no expert in highly available distributed queue systems that do not fail. I know it is possible to build a custom queue system with a couple of existing open source solution, but I got to face the hard fact that is expensive in knowledge and infrastructure.

Postgres is a reliable relational database with a solid set of tools to perform maintenance operations (backup/restore) that are familiar to most developers. Among *Postgres* wonderful internal features I can mention the table namespacing, basically you can route data to specific logic discs based on the table's name. Wonderful. You can distribute data the way you want to the disk you want, based on need.

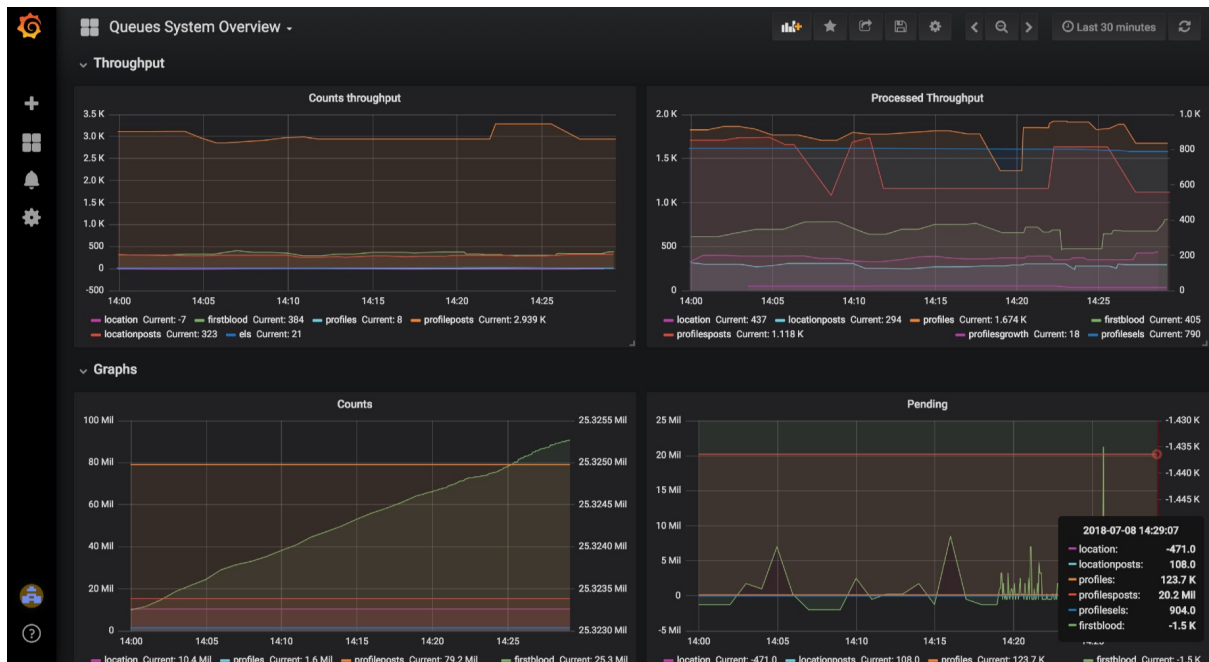
To date my production db is ~700Gb on a relatively small *AWS m4.medium* instance. There are ~160M documents in it, and I can scale the data storage to multiple disks before I even start to think to clusterize *Postgres* itself (which is in the roadmap for *Fetchq* anyway).

It is ridiculously easy to dump single (or multiple) queues into compressed files and automatically send them up to S3 for long term restorable backup. It doesn't sweat!

Near real-time metrics

`SELECT COUNT(*)` works only up to few hundred thousand records. Beyond that point you have to build your own guns.

Much much work went into (and still is) building a solid and reliable metric system that represent a *near real time* status of your queue. But today you can track your entire system progress through a set of *Grafana boards* that give an immediate visual understanding. Cool stuff.



Of course you can also programmatically access any metric, plus you can **track your own metrics as well**.

Say you want to **measure and track how long a piece of code takes to execute**, this is not only doable, but it integrates seamlessly with the core metrics and you can plot a custom chart for that too.

Best effort policy, with a twist

Fetchq is not just another *FIFO* queue. Documents are **sorted by date from the older to the newer**.

When you insert a document in the queue you can freely choose a **next execution date**, *Fetchq* guarantees that no actions will be taken until that date comes.

This single feature allows you to inject a document in any point-in-time. Say you really need to get a document processed right away, you can simply inject it with a `nextExecution = 0000-00-00`, it probably get picked up right away (unless you use that setting for all your documents!)

Queue partitioning

More often than not you end up having some stuff that is more important than others. You may want to prioritize the execution of documents for a specific customer over another, and so forth.

Fetchq allows you to detail a **priority number**, higher priority gets processed first.

A **priority** will create a partition in the queue, all the **pending documents** of priority "1" will be processed before the **pending documents** of priority "0".

Queue versioning

God knows if **data change through time**! It's just a fact of Life, hence *Fetchq* accept it and tries to work for the best!

When you insert a document in a queue you can set a **version number** which is just an integer. There are no particular rules for that, I normally start from "0" and bump it by one unit any time I feel I need a new version.

When you want to pick a document for processing you must specify which version you are targeting.

This is so cool because it allows you to run multiple concurrent versions of the same **worker** that targets different versions of the document.

Just imagine you need to process some *XML* files to extract info, but they might come in in different formats. If you use the version number wisely it's going to be an easy thing for you!

Migration workers

It's not just it yet with versioning. You can also decide to **migrate a document** from version "0" to version "1" (maybe you need to update some data in the document's payload).

If you have few thousands documents it's a no brainer, but if you - like me - deal with millions or hundreds million of those, you know you can't just run an update or a `forEach` loop.

Fetchq allows you to **define a special type of worker that is employed at the sole purpose of upgrading a document to a new version**, in preparation for further data proddessing.

We used that feature a lot with our *superheroes* data scraping project as they were changing their APIs and data format A LOT in a very short amount of time and without notice.

If you scrape a document and you get a data processing error you can decide to "promote" that document to a next version, meaning that you need to investigate further and probably update your **worker** so to match the new data format.

Errors threshold

Your workers are going to screw up in some way. You know that, I know that. Bugs exist.

But it's quite tricky to debug in a system that is possibly running on multiple servers and handle millions of documents each day. A **worker** can simply crash for many many reasons.

Fetchq allows you to resolve a document processing with some different actions, one of those is `reject` where you can specify an error message and a detail. You can even reference a *processId* (I use it to find out which *Docker container* in which *EC2 instance* threw the error).

Fetchq has the knowledge of **orphan documents**, documents that got picked by a **worker** but were never resolved in time because the **worker** crashed.

In both cases we are facing a document processing error, that may or may not repeat through time. Just imagine if it was a temporary networking issue, do you want to loose your document because of that? No you don't!

For this reason *Fetchq* allows for a **try again** policy. By default each document is given 5 chances to resolve, but you can customize this threshold queue by queue.

Errors logging

All the errors that are generated by a **worker** or from **orphan documents** are logged in a queue specific data partition (fancy name for "a table") for performance reasons.

You can read from this table to find out what is going not so well about your data processing.

Error retention policy

Error logging can be very expensive in terms of data storage, for that reason *Fetchq* automatically drops old logs.

By default we retain the last 24h, but you can customize this policy for each queue.

Getting Started

This tutorial will guide you through the setup of a *Fetchq* project.

NOTE: You are going to use *Docker*, *Postgres* and *NodeJS* to set this up. I assume you have at least a brief working with those tools.

Follow those steps:

1. [Setup Postgres](#)
2. [Initialize Fetchq](#)
3. [Create a new queue](#)

Setup Postgres database

The first step is to boot a *Postgres* instance that can run the *Fetchq* extension.

Fetchq crew maintains an [image on Docker HUB](#) that is basically *Postgres* with the *Fetchq* extension's files shipped within.

Run With Docker...

To quickly run *Fetchq* you can copy paste this command:

```
docker run --name fetchq -p 5432:5432 fetchq/fetchq:9.6-1.2.0
```

...or With Docker Compose

I personally like to run it with *Docker Compose* as it feels easier to modify and reproduce across my projects. Try this snippet in your `docker-compose.yml` :

```
# ./docker-compose.yml
services:
  postgres:
    image: fetchq/fetchq:9.6-1.2.0
    ports:
      - 5432:5432
    container_name: fetchq
    network_mode: bridge
```

`container_name` and `network_mode` are important for the successful execution of the ``psql`` commands in this page.

Connect to the database

This will start *Postgres* 9.6 with the default configuration as described in the [Postgres's Docker Hub page](#). Anyway use "*postgres*" as setting for *username*, *password* and *database* **when you connect** using *psql* or a visual client such [Postico](#).

Here I'm going to use *Docker* to run *psql* and connect to the database.

```
# Start a psql session:
docker run -it --rm --link fetchq:postgres postgres psql -h postgres -U postgres

# Launch a single query:
docker run -it --rm --link fetchq:postgres postgres psql -h postgres -U postgres \
-c 'SELECT * FROM current_database()'
```

IMPORTANT: from now on I will only write *SQL QUERIES*, assuming that you will run them with the "single query" command. But suits yourself and use whatever method you like best!

Initialize Fetchq

So far you have a plain *Postgres* running, as it comes out from the factory. Fetchq extension files are shipped within the container, but the extension itself needs to be created:

```
CREATE EXTENSION IF NOT EXISTS "fetchq";
```

You may also want to create the extension `uuid-oss` which is a core extension used to generate unique document names, useful for appending unique documents into a queue.

```
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

At this point you are ready to initialize *Fetchq* and create the basics *schema* for your database:

```
SELECT * FROM fetchq_init();
```

Test your installation by listing the tables in the *public schema*:

```
SELECT * FROM pg_catalog.pg_tables WHERE schemaname = 'public';
```

× 3 docker

```
postgres=# SELECT * FROM pg_catalog.pg_tables WHERE schemaname = 'public';
```

schemaname	tablename	tableowner	tablespace	hasindexes	hasrules	hastriggers	rowsecurity
public	fetchq_sys_queues	postgres		t	f	f	f
public	fetchq_sys_metrics	postgres		t	f	f	f
public	fetchq_sys_metrics_writes	postgres		t	f	f	f
public	fetchq_sys_jobs	postgres		t	f	f	f

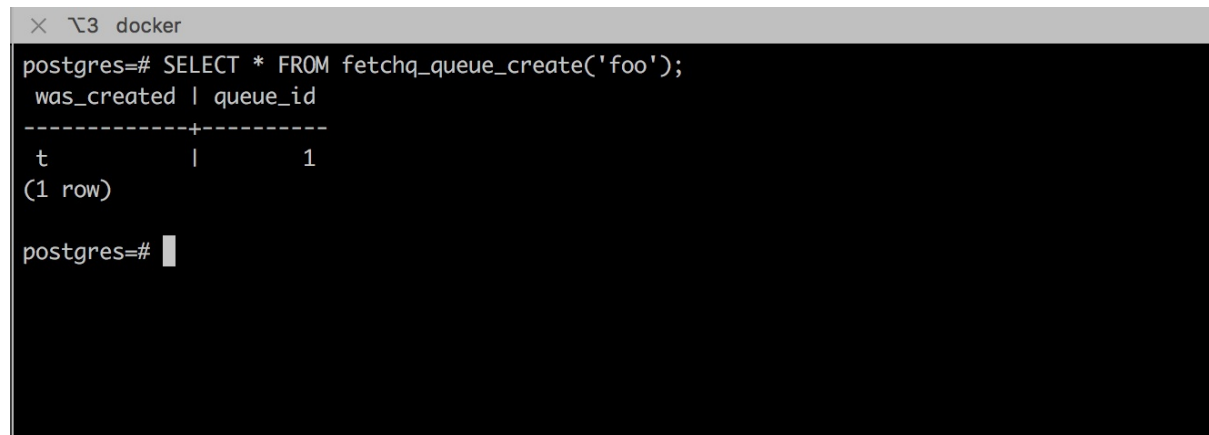
(4 rows)

```
postgres=#
```

Create a new queue

Fetchq provides you with queues for you to manage few-to-millions documents. Hence your next step is to *create* a new queue:

```
SELECT * FROM fetchq_queue_create('foo');
```



```
× 3 docker
postgres=# SELECT * FROM fetchq_queue_create('foo');
 was_created | queue_id
-----+-----
 t          | 1
(1 row)

postgres=#
```

This command will *upsert* a queue. If the queue didn't exist then it's being created; else not so much is happening, but the command will not fail. You have a boolean indicator that tells you whether the queue had been created or updated.

NOTE: in future versions the "update" mode might trigger reindexing or version migration schema updates on a specific queue.

Queue Tables

If you list the tables again (`SELECT * FROM pg_catalog.pg_tables WHERE schemaname = 'public';`) you will notice that 3 new tables were created:

- `fetchq__foo__documents` - will store the queue's documents
- `fetchq__foo__errors` - errors will be logged here
- `fetchq__foo__metrics` - timeserie metrics will let you plot the status in time for this queue

Queue Data

Fetchq keeps some queue related data around the system tables.

You can change some queue settings in `fetchq_sys_queues` like metrics and error logs retention period, details about how often the maintenance jobs on the queue are performed, generic configuration about other jobs, or possibly extensions.

For sure you want to keep an eye on the `fetchq_sys_metrics` table which contains info regarding the current status of all the queues: number of documents, size of the [pending documents](#), etc.

Drop a Queue

In any moment you are entitled to **completely and irreversibly delete** a queue and all its related contents: documents, jobs, metrics, ...

```
SELECT * FROM fetchq_queue_drop('foo');
```

Push a Document

Now that you have a queue ready, the first action you want to look into is how to push data into it.

There are 3 possible ways to add a document into a queue:

- `fetchq_doc_push()` - add a unique document by subject
- `fetchq_doc_upsert()` - add a unique document by subject, or update an existing document
- `fetchq_doc_append()` - add a new document with an auto-generated unique subject

In this document I'm going to briefly describe how and in which conditions you may want to use those methods.

fetchq_doc_push

```
# Function signature
fetchq_doc_push(
  queue::string      // queue name, es: "foo"
  subject::string     // your unique id, es: "john"
  version::int        // the version of the document, es: 0
  priority::int        // an arbitrary priority, es: 0
  nextIteration::date // when to execute the document, es: "2081-06-30 15:05"
  payload::jsonb       // a json payload, consider it a cookie. es: '{"age":22}'
)

# SQL Example
SELECT * FROM fetchq_doc_push('foo', 'john', 0, 0, NOW() + INTERVAL '20s', '{}');
```

If you own the subject of your document you are going to use `fetchq_doc_push()`. The classic example is that you have a list of superheroes that you want to monitor by scraping the website `http://avengers.com/{superuser}`.

This project requires you to periodically scrape `avengers.com` on a specific page, which url can be generated automatically by knowing the name of the superuser. At this point the most important thing is to do not insert the same superhero twice or we are going to waste time!

In this context the `payload` does not have much of an importance, but can be used as storage for informations that need to be carried out between different iterations:

```
# those documents are good because the subject is unique
SELECT * FROM fetchq_doc_push('avengers', 'ironman', 0, 0, NOW(), '{ "lastScore": 20 }');
SELECT * FROM fetchq_doc_push('avengers', 'hulk', 0, 0, NOW(), '{ "lastScore": 18 }');
...
# this should be an error because "ironman" was already there!
SELECT * FROM fetchq_doc_push('avengers', 'ironman', 0, 0, NOW(), '{ "lastScore": 20 }');
```

If you attempt to push duplicates they will simply be ignored. The function returns the number of documents that got inserted without causing subject duplication.

NOTE: this behaviour allow for the fastest possible `INSERT` performance, expecially when using the `bulk insert` signature.

On an average mac I could measure insert speed of **18K documents per second** when pushing **5M documents!**

fetchq_doc_upsert

```
# Function signature
fetchq_doc_upsert(
  queue::string      // queue name, es: "foo"
  subject::string    // your unique id, es: "john"
  version::int       // the version of the document, es: 0
  priority::int      // an arbitrary priority, es: 0
  nextIteration::date // when to execute the document, es: "2081-06-30 15:05"
  payload::jsonb     // a json payload, consider it a cookie. es: '{"age":22}'
)

# SQL Example
SELECT * FROM fetchq_doc_upsert('foo', 'john', 0, 1, NOW() + INTERVAL '1m', '{"age":22}');
```

`upsert` shares the same signature as `push` but it has update capabilities in case the document already exists in the queue. The following properties will be updated:

- `priority`
- `nextIteration`
- `payload`

NOTE: the update will be applied to [active documents](#) only.

fetchq_doc_append

```
# Function signature
fetchq_doc_append(
  queue::string      // queue name, es: "foo"
  payload::jsonb     // a json payload, consider it a cookie. es: '{"age":22}'
  version::int       // the version of the document, es: 0
  priority::int      // an arbitrary priority, es: 0
)

# SQL Example
SELECT * FROM fetchq_doc_append('foo', '{"age":22}', 0, 1);
```

If you need to queue a document regardless of the subject univocity you append.

In this circumstance the `payload` becomes the most important piece of information as it is the only discriminant between documents. Document version and priority work as usual.

A classic example might be a `sendmail` queue to which we want to append documents that describe emails that need to be send. The [worker](#) will then process the queue, using the payload as source of the email informations:

```
SELECT * FROM fetchq_doc_append('sendmail', '{"to":"foo@foo.it", "template":"signup", "subject":"welcome!"}', 0, 0);
```

This document will be appended with the current `dateTime` and will be processed as soon as possible. Classic implementation of a FIFO queue. In this type of queue the document is likely to be deleted after a successful iteration, but you may have different needs!

pending document

A document who's `next_iteration` date is set in the past is called **pending** and it will be processed as soon as possible.

pending documents

see: [pending document](#)

active document

A document becomes active when is being selected using `fetchq_doc_pick()` , normally a [worker](#) process is responsible of this state.

Only [pending documents](#) can become active.

active documents

see: [active document](#)

worker

A [worker](#) is an app written in any language you may know (our examples are written in NodeJS) that pick a document from a queue and process it.

Fetchq is engineered so that you can run multiple workers concurrently and they will not cross each other or receive the same document twice!

orphan document

When a document gets picked by a [worker](#), the [worker](#) is given a certain amount of time to complete the data processing. It's a timeout tha the [worker](#) can set by itself and by default is set to 5 minutes.

If this time lapse and the worked takes no action, *Fetchq* assumes that the [worker](#) is dead and takes action on the document.

Based on the error threshold the document may be scheduled for another attempt or may become a [dead document](#).

orphan documents

see: [orphan document](#)

dead document

Means: "you don't need to process me ever again".

A document can be marked as "dead" by an explicitly [worker](#)'s action or because it went beyond the error threshold.

NOTE: the document is still in the queue so no other documents with the same subject will be admitted into the queue. If your want to allow for re-insert then you need to drop the document.