Menu

Search … Search

- Home
- About
- Further Reading
- Skip to menu toggle button

[[WM:TECHBLOG]]

Open Source for Open Knowledge

Wikidata Query Service graph database reload at home, 2025 edition

Posted on: April 8, 2025 Last updated on: December 22, 2025 Categorized in: Wikidata Adam Baso

This post is about importing Wikidata into the graph database technology used for hosting the Wikidata Query Service (WDQS). The post includes details on how you can perform your own full Wikidata import to Blazegraph in about a week if you have a nice desktop computer, which was one of the nice takeaways from the analysis.

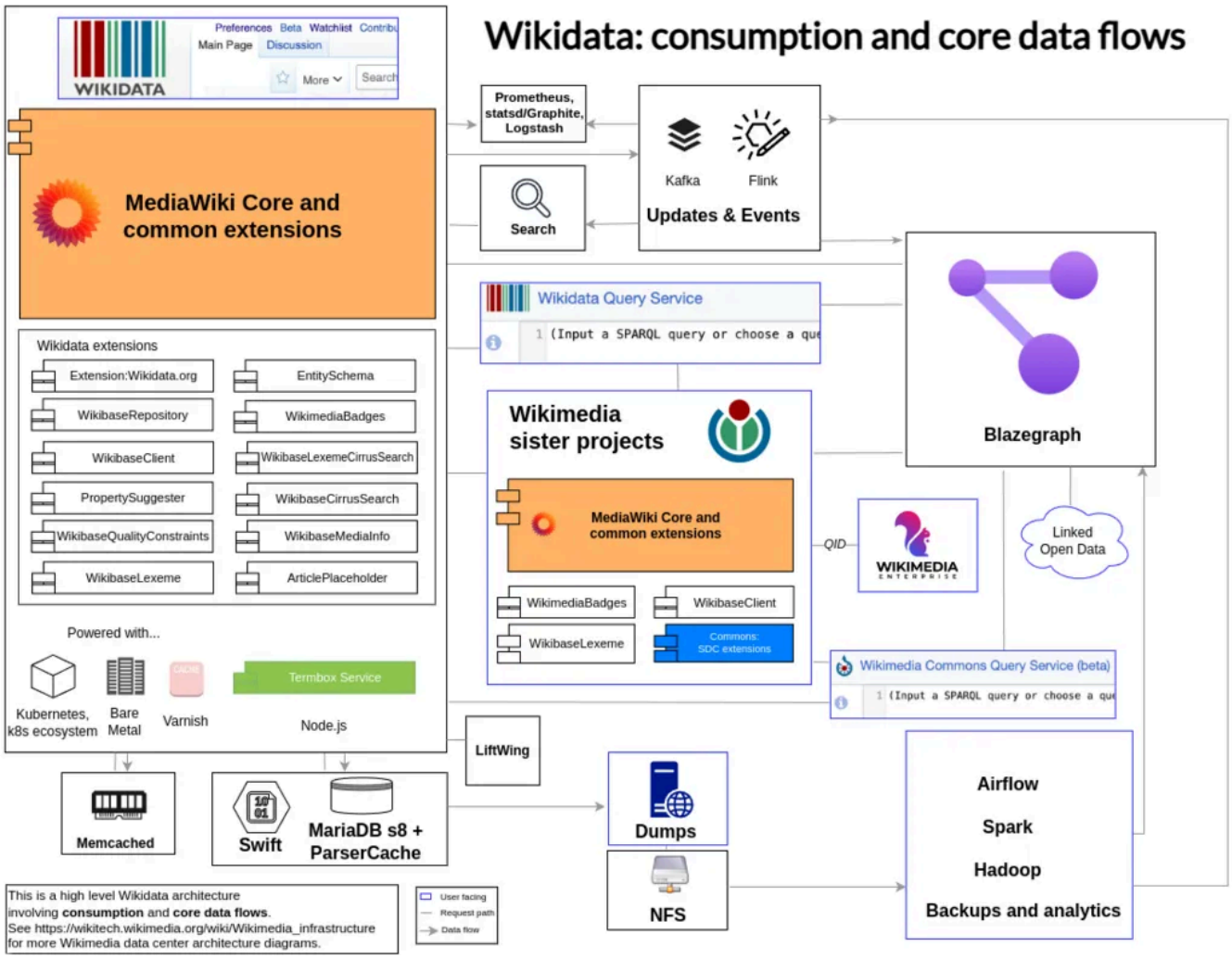
System utilization around file 20 of 2583 of Wikidata import to local WDQS

Graph databases and Wikidata

Graph databases are a useful technology for data mining relationships between all kinds of things and for enriching knowledge seeking via retrieval-augmented generation ("RAG") and other AI. Within the Wikimedia content universe, we have a powerful graph database offering called the *Wikidata Query Service* ("WDQS") which is based on a mid-2010s technology called *Blazegraph*.

Wikidata community members model topics you might find on Wikipedia, and this modeling makes it possible to answer all kinds of questions after importing Wikidata's data into WDQS. Our colleague Trey wrote a nice post describing WDQS that you should check out.

The Wikidata and WDQS architecture spans a number of components and technologies.



Wikidata and Wikidata Query Service high level diagram at it pertains to data flows that may be involved in consumption

Big data growing pains

As Wikidata has grown, the WDQS graph database has become pretty big, with about 16.6 billion records (known as *triples*) as of this writing, with many intricate relationships between those records that ultimately result in complex and large data structures on disk and in memory. Unfortunately, the WDQS graph database has also become unstable as a result, and this seems to be getting worse as the database gets larger. The last time a data corruption occurred it rippled through the infrastructure and it took about 60 days to reload the graph database to a healthy state across all WDQS graph database servers (part of this had to do with repeated failed imports; hopefully techniques in this here post are instructive to others encountering failed imports).

The long recovery time was a prompt to further enhance the data reload mechanisms and to figure out a way to manage the growth in data volume. Over the course of the last year, the Search Platform Team, which is part of the Data Platform Engineering unit at the Wikimedia Foundation, worked on a project to improve things.

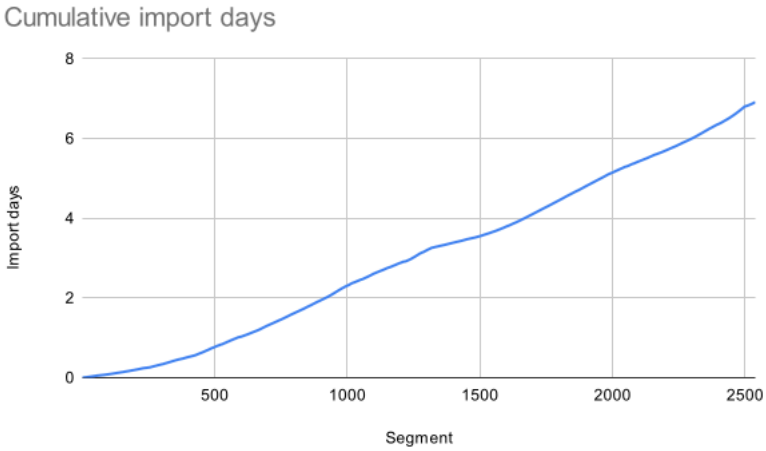
As part of its goal setting, the team determined it should make it possible to support more graph database growth (up to 20 billion rows in total) while being able to recover more reliably and more quickly in the event of a database corruption (within 10 days). The idea being that complex queries are useful – WDQS is one of the most important tools in the Wikidata system – but only if the system is up!

In order to support more database growth, it was pretty clear that either the backend graph database would need to be completely replaced or it would be necessary to split the graph database to buy some time, as the clock had run out on the graph database being stable. A full backend graph database replacement *is* necessary, but this is a rather complex undertaking and would push timelines out considerably; the replacement is an area for further analysis.

A stopgap solution seemed best. So, the team pursued the approach of splitting the graph database from one monolithic database into separate databases partitioned by two coarse grained knowledge domains: (1) scholarly article entities and (2) everything else. As fate would have it, these two knowledge domains are roughly equivalent in size.

After many changes to the data ingestion, streaming, network topology, and server automation the migration of the WDQS servers to the split-based architecture is happening now in the spring of 2025 (around the time of this blog post, coincidentally).

Now, while working through the split of the graph database, although initial testing suggested that it should be possible to achieve a data reload of a graph of 10 billion rows within ten days and reloads for both knowledge domains could run in parallel (thus allowing for 20 billion rows in total), there still wasn't a lot of room for error. What happens if a graph database corruption happens right when the weekend starts? What if some other sort of server maintenance is blocking the start of a reload for a day or two? We wanted to be certain that we could reload and still have some breathing room to stay within 10 days.



Cumulative Wikidata dump import time using the approach detailed in this post on publicly available dump data

Hardware to the rescue?

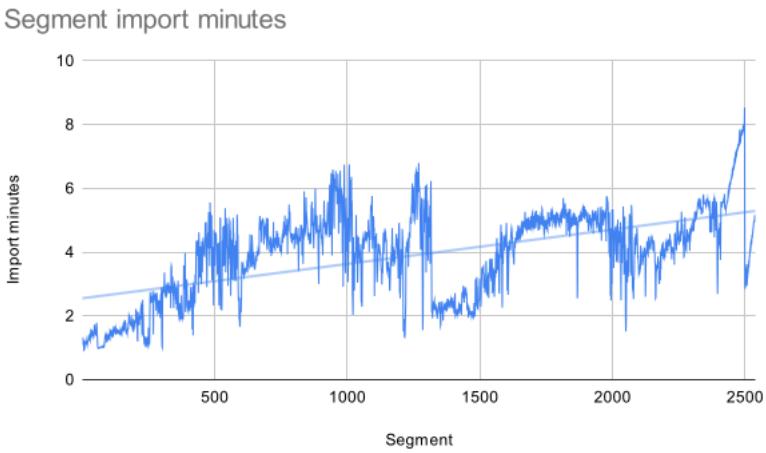
From previous investigations it seemed that more powerful servers could speed up data reloads. Obvious, right?

Well, yes and no. It's a little more complicated. People have tried.

- Adam Shorland shared his observations in testing WDQS Blazegraph data load performance.
- Several researchers, including a previous consultant to Wikimedia Foundation, Andrea Westerinen, profiled getting and hosting your own copy of Wikidata.
- Ghislain Auguste Atemezing analyzed Wikidata imports with Amazon Neptune, which is the commercial SaaS successor to Blazegraph (Blazegraph is no longer actively maintained), as well as other alternatives.
- Chalupsky et al looked at more narrowly defined extraction for the purpose of Cypher-like queries (the Graph Query Language, or GQL for short, has roots with Cypher).

The legacy Blazegraph wiki has some nice guidance on Blazegraph performance optimization, I/O optimization, and query optimization (some of this knowledge is evident in a configuration ticket from as early as 2015 involving one of the original maintainers of Blazegraph). Some of it is still useful and seems to apply, although some changes backported in the JDK plus the sheer scale of Wikidata make some of the settings harder to reason about in practice. Data reloads have become so big and time consuming (think on the order of weeks, not hours) that it is impractical (and expensive) to profile every permutation of hardware configuration and Blazegraph, Java, and operating system configuration.

This said, after noticing that a personal gaming-class machine I bought in 2018 for a machine learning workflow (cross-compiling and applying transfer learning ultimately for an offline Raspberry Pi application) was able to do much faster WDQS imports than what we were seeing on our data center servers, I wanted to understand if there were advances with CPU, memory, and disk in the wild that might point the way to even faster data reloads and wanted to understand better if any software configuration variables could yield bigger performance gains.



Wikidata dump segment import times using the approach detailed in this post

This was explored in T359062, where you'll find an analysis and running log of import performance on various AWS EC2 configurations, a MacBook Pro (2019 Intel-based), my desktop (2018 Intel-based), our bare metal data center servers, and Amazon Neptune. The takeaways from that analysis were that:

- Cloud virtual machines are sufficiently fast for running imports. They may be an option in a pinch.
- Removal of CPU governor limits on data center class bare metal servers significantly improved performance. In other words, allowing the CPUs to run at their maximum published clock rates sped up imports.
- Removal of CPU governor limits didn't confer an advantage on prosumer grade computers.
- A Blazegraph buffer configuration variable increase significantly improved import speed.
- Higher grade hard drives (fast consumer NVMe at home and data center class RAIDed SSDs in production) confer a noticeable performance advantage.
- The Amazon Neptune service was by far the fastest option for import. It's unclear if free or near-free data ingestion observed during the free cloud credit period would extend for additional future imports, though. It is a viable option for imports, but requires additional architectural consideration post an import.

- The N-Triples file format (`.nt`) dramatically improved import speed. It should be (and now, is) used instead of the more complicated Turtle (`.ttl`) format for imports.

Computing configuration and initial setup

My 2018 personal gaming-class machine with a 6-CPU configuration (up to 4.6 GHz turbo boost) after several years of upgrades has 64 GB of DDR4 RAM and a 4 TB NVMe.

A *full* Wikidata graph import into Blazegraph took 5.22 days with this configuration and our optimized N-triples files in August 2024.

I had the benefit of pre-split N-triples files produced from our Spark cluster as part of an Airflow DAG that runs weekly, where there are no duplicate lines in the files and there are some additional simplifications compared to the N-triples files produced by legacy jobs in our data dumps infrastructure. If you're doing this at home without a large Spark cluster, though, you can fetch `wikidata-<YYYYMMDD>-all-BETA.nt.bz2` from a datestamped directory in the Wikidata dumps and run some shell commands to prepare files to achieve something similar (do note that the data is less optimized, but it works).

You can at present import somewhat reliably and peformantly with one 4 TB NVMe internal drive and one 2 TB external (or SATA) SSD drive if you are willing to script some file compression to avoid running out of disk. In the example that follows, I assume that you have three drives, though: one 4 TB NVMe drive (let's say this is your primary drive), one SATA or external 2+ TB SSD (that's `/media/ubuntu/EXTERNAL_DRIVE` in the example), and another SATA or external 2+ TB SSD (that's `/media/ubuntu/SOME_OTHER_DRIVE` in the example).

The commands

Here are the commands you'll need to download the Wikidata dump, break it up into smaller files that Blazegraph can handle, and import within a reasonable timeframe.

Note that you'll need to have a copy of the logback.xml file downloaded to your home directory.

```
# Download some dependencies
sudo apt update
sudo apt install bzip2 git openjdk-8-jdk-headless screen
git clone https://gerrit.wikimedia.org/r/wikidata/query/rdf
cd rdf
./mvnw package -DskipTests
sudo mkdir /var/log/wdqs
mkdir /home/ubuntu/wtemp
# Your username and group may differ from ubuntu:ubuntu
sudo chown ubuntu:ubuntu /var/log/wdqs
touch /var/log/wdqs/wdqs-blazegraph.log
cd dist/target/
tar xzvf service-0.3.*-SNAPSHOT-dist.tar.gz
cd service-0.3.*-SNAPSHOT/
cd /media/ubuntu/EXTERNAL_DRIVE
mkdir wd
cd wd
# Run the next multiline command before the weekend - be sure to verify
# that your computer will stay awake without reboot. The server throttles
# somewhat, so the download takes a while. And the deflate and split-sort-split
# also take a while. There are faster ways, but this is easy enough. In case
# you were wondering, wget seems to work more reliably than other options.
# Torrents do exist for dumps, but be sure to verify their checksums against
# dumps.wikimedia.org and verify the date of a given dump. In the following
# command pipeline we just print out the checksum for manual verification later,
# as it's nice to let this run over a weekend and come back on a Monday to
# verify instead of potentially having to wait longer; it usually works fine.
date && \
wget https://dumps.wikimedia.org/wikidatawiki/entities/20241216/wikidata-20241216-all-BETA.nt.bz2 && \
date && \
wget https://dumps.wikimedia.org/wikidatawiki/entities/20241216/wikidata-20241216-sha1sums.txt && \
grep wikidata-20241216-all-BETA.nt.bz2 wikidata-20241216-sha1sums.txt && \
sha1sum wikidata-20241216-all-BETA.nt.bz2 && \
date && \
bzcat wikidata-20241216-all-BETA.nt.bz2 | split -d --suffix-length=4 --lines=7812500 --additional-suffix='.nt' - 'wikidata_full_with_duplicates.' && \
date && \
sort wikidata_full_with_duplicates.*.nt --unique --temporary-directory=/home/ubuntu/wtemp | split -d --suffix-length=4 --lines=7812500 --additional-
suffix='.ttl.gz' --filter='gzip > $FILE' - 'wikidata_full.' && \
date
# Let's head back to where you were:
cd ~/rdf/dist/target/service-0.3.*-SNAPSHOT/
mv ~/logback.xml .
# Using runBlazegraph.sh like production, change heap from 16g to 31g and
# point to logback.xml by updating HEAP_SIZE and LOG_CONFIG to look like so,
# without the # comment symbols, of course.
#  HEAP_SIZE=${HEAP_SIZE:-"31g"}
#  LOG_CONFIG=${LOG_CONFIG:-"./logback.xml"}
vim runBlazegraph.sh
# Modify the buffer in RWStore.properties so it looks like this (1M, not 100K),
# without the # comment symbol, of course.
#   com.bigdata.rdf.sail.bufferCapacity=1000000
vim RWStore.properties
# Let's get Blazegraph running in the background.
screen
# Wait a few seconds after running the next command to ensure it's good.
 ./runBlazegraph.sh
# Then CTRL-a-d to leave screen session running in background.
# You can chain the following commands together with && \ if you like.
# Let's import the first file to make sure it's working (takes about 1 minute).
time ./loadData.sh -n wdq -d /media/ubuntu/EXTERNAL_DRIVE/wd -s 0 -e 0 -f 'wikidata_full.%04d.ttl.gz' 2>&1 | tee -a loadData.log
# If it worked, let's import another 9 files (maybe another ~10 minutes).
time ./loadData.sh -n wdq -d /media/ubuntu/EXTERNAL_DRIVE/wd -s 1 -e 9 -f 'wikidata_full.%04d.ttl.gz' 2>&1 | tee -a loadData.log
# Let's see how long it took to import the first ten files, just sum and then
# divide by 1000 for seconds (sum / 1000 / 60 / 60 / 24 for days).
grep COMMIT loadData.log | cut -f2 -d"=" | cut -f1 -d"m"
```

```
# Now let's handle the rest of the files. This could take a week or so - again
# be sure to verify that your computer will stay awake, without reboot.
time ./loadData.sh -n wdq -d /media/ubuntu/EXTERNAL_DRIVE/wd -s 10 -f 'wikidata_full.%04d.ttl.gz' 2>&1 | tee -a loadData.log
# Hopefully that worked. Go to http://localhost:9999/bigdata/#query and run the
# following query:
#   SELECT (count(*) as ?ct) WHERE { ?s ?p ?o }
# For this example it was 19,827,410,787 with the non-optimized dump.
# As of March 2025 you might expect 16.6B for an optimized dump, as here:
# https://query.wikidata.org/#select%20%28count%28%2a%29%20as%20%3Fct%29%20where%20%7B%3Fs%20%3Fp%20%3Fo%7D
# Celebrate!
# Let's close Blazegraph and make a backup of the Blazegraph journal.
screen -r
# CTRL-c to stop Blazegraph
exit
# Okay, screen session ended, let's look at the size of the file
ls -alh wikidata.jnl
cp wikidata.jnl /media/ubuntu/SOME_OTHER_DRIVE/
```

You'll notice here I don't take time to make intermediate backups of the Blazegraph journal file. It's a good exercise for the reader!

Production, in practice

We were a little surprised that my desktop could perform faster imports than what we were seeing in our data center servers. Our colleague Brian King in Data Platform SRE had a hunch, which turned out to be correct, that we could adjust the CPU governor on the production servers. This helped dramatically on the production servers, and when coupled with the graph split it makes recovery much faster. We don't need to use the buffer size configuration trick as described above, but we also have that as an option should it become necessary.

Considerations

It would be nice to have no hardware limitations, but there are some practical limitations.

CPU: Although CPU speed increases are still being observed with each new generation of processor, much of the advances in computing have to do with parallelizing computation across more cores. And although WDQS's graph database holds up relatively well in parallelizing *queries* across multiple cores, it's difficult to optimize perfectly for many-cores architecture for data *import*.

Memory: Although more memory is commonly beneficial to large data operations and intuitively you might expect a graph database to work better with more memory, the manner in which memory is used by running programs can drive performance in surprising ways, ranging from good to bad. WDQS runs on Java technology, and configuration of the Java heap is notoriously challenging for achieving performance without long garbage collection ("GC") pauses. We deliberately use a 31 GB heap in production for our Blazegraph instances. It's also important to remember that a large Java heap requires a lot of RAM, which can become expensive.

Nevertheless, more memory can be helpful for filesystem paging operations. Taking the hardware configuration guidance at face value suggests that we would need about 12 TB of memory for the scale of data we have today for an ideal server configuration (we have about 1200 GB of data with about 16.6 billion records). We're getting by with 128 GB of memory per server, which is much less than 12 TB of memory. We've also heard of people using several hundred GB of memory and having reasonable success. More memory would be nice, but today it's too expensive in a multi-node setup built for redundancy across multiple data centers.

Disk: NVMe disks have brought increased speed to data operations. But backpressure on CPU or memory can also mask what might otherwise be able to manifest with speedier NVMe throughput. NVMEs did show a material performance gain during testing, although presently in production we're thankfully doing okay with RAIDed data center class SSDs (6 TBs). NVMes would most likely be an improvement in the future in the data center, but they are priced higher for data center quality devices, whereas prosumer grade NVMes for personal computers are reasonably priced; due to the risks of hardware failure we prefer to avoid prosumer grade NVMes in the data center.

Caveats

A few things to remember if you're running Blazegraph at home:

- Be mindful of `SERVICE wikibase:mwapi` syntax, as it uses external Wikimedia APIs; be sure to avoid rapid repeat queries with this syntax.
- Beware of exposing on the network: it doesn't have the same load balancing and firewall arrangement, as well as other security controls, as the real Wikidata Query Service.

Conclusion

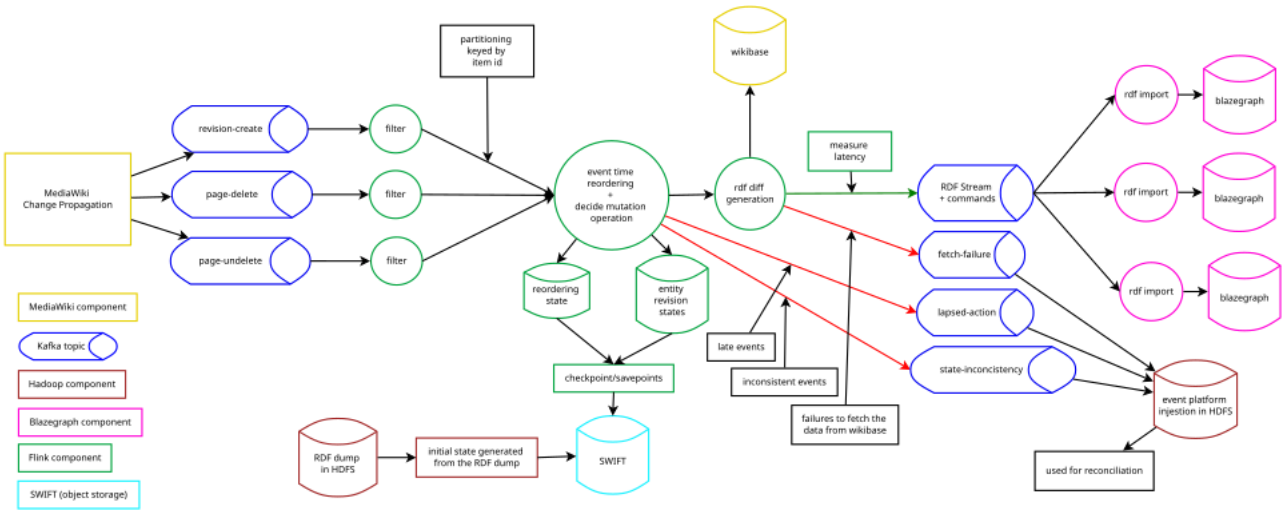
If you are looking to host your own Blazegraph database of Wikidata data *without* having two graph partitions (i.e., if you want to have the full graph in one partition) you might try the following:

1. Get a desktop with the fastest CPU possible and acquire a speedy 4 TB NVMe plus 64 GB or more of DDR5 RAM; get a couple larger internal SATA SSD or faster throughput external SSDs if you can, too. As of this writing consumer grade 4 TB NVMes can be had with reasonable price-performance tradeoffs; perhaps 6 GB or 8 GB NVMes with the same level of performance will become available in the next year or two.
2. Import using the N-Triples format split into multiple files.
3. Consider scripting the batch import operation to make a backup copy of the graph database for every 100 files imported. That way if your graph database import fails at some point you can troubleshoot and resume from the point of backup. The intermediate backup will slow things down a little but it may save you many days in the end.
4. If you can't build or upgrade a desktop of your own, consider use of a cloud server to perform the import, then copy the produced graph database journal file to a more budget friendly computer; remember that in addition to cloud compute and storage costs, there may be data transfer costs.

As you saw up above, there are a few variables in configuration files that you need to update in order to speed an import along.

Production

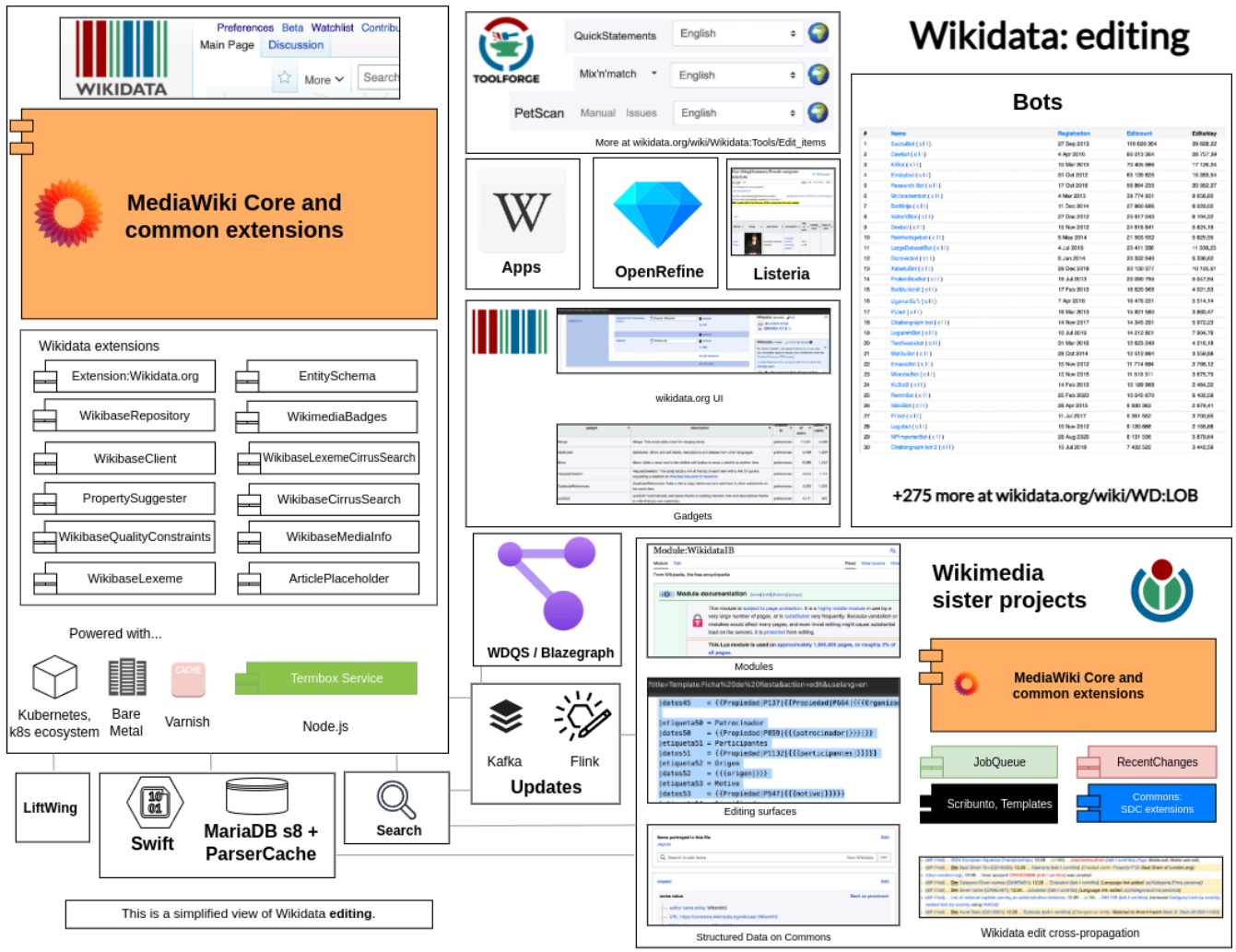
After splitting the Wikidata graph database in two and removing CPU throttling for Wikidata Query Service *production data center* nodes, we're now able to import the WDQS database and catch Blazegraph up to the Wikidata edit stream in less than a week. The way Wikidata updates are applied in the production environment is an interesting topic unto itself, but this diagram gives you an idea of how it works.



Production Wikidata Query Service Kafka Flink-based updater

Acknowledgments

Thank you for reading this post. I'd like to thank the wonderful colleagues in Search Platform, Data Platform SRE, Infrastructure Foundations, Traffic, and Data Center Operations for the solid work on the graph split, and more specifically regarding this post, the support in exploring opportunities to improve performance. I'd like to especially express my gratitude to David Causse (WDQS tech lead and systems thinker), Peter Fischer (Flink-Kafka graph splitter extraordinaire), Erik Bernhardson (thank you for the Airflow environment niceties), Ryan Kemper & Brian King & Stephen Munene & Cathal Mooney (cookbook puppeteers who balance networks and servers with a keyboard), Andrew McAllister (thank you for your analysis of query patterns!), Haroon Shaikh & Renil Thomas (much appreciated on the AWS configs) and Willy Pao & Rob Halsell & Sukhbir Singh (thank you for helping investigate NVMe options). Thank you to my Engineering Director, Olja Dimitrijevic, for encouraging this post, as well as to Tajh Taylor for review. And as ever, major thanks to Guillaume Lederrey, Luca Martinelli, and Lydia Pintscher (WMDE) for partnership in WDQS and Wikidata and its amazing community.



High level overview of Wikidata editing data flows and how it relates to components involved in production

Previous Post How Many Languages Does Wikimedia Search Support?

Next Post Wikimedia Cloud VPS: IPv6 support