Editor: Diomidis Spinellis
dds@aueb.gr

Engineering Data Processing Workflows

Diomidis Spinellis

DATA SCIENCE, BUSINESS analytics, and machine learning applications routinely churn large volumes of data. Often, you can rely on domain-specific tools, such as Prometheus for performance monitoring or Snort for intrusion detection. In other cases, you need to develop bespoke data processing workflows. I have noticed that experts typically follow specific best practices when engineering such workflows, which they have learned through tedious experimentation and expensive mistakes. Here, based on my experience in applying data analytics in diverse fields, ranging from software[1] and the environment[2] to bibliometrics[3] and finance,[4] I am distilling the most important practices for engineering data processing workflows. When building such workflows, your goal is to derive correct results in a reproducible, maintainable, and efficient manner.

Split Data and Tasks

You may be tempted to create a program that performs in one go all the required processing, but this is a mistake. Workflows based on such programs are difficult to test, to run reliably, to troubleshoot, and to parallelize. Instead, structure your input data, your calculations, and your output into chunks (shards) that can be processed in reasonable amounts of time, for example, a few hours. Choose the size so that the fixed overhead of running a task is only a tiny part of the total execution cost.

At the point where each task finishes the processing of a chunk, consider adding an execution hook that will allow you to modify or gracefully terminate the running process. This can be a file that a Python script will read and evaluate if it exists or a file that a Unix shell script will source. For example, if each shell script loop finishes with the command "`source loop-hook.sh`," you can terminate the loop at its next iteration by writing "`break`" in the "`loop-hook.sh`" file or see a timestamp after each iteration by writing "`date`" in it.

Avoiding overly large files simplifies their transfer and backup operations, especially when you deal with failures. Splitting tasks into multiple independent processes makes it trivial for you to utilize multiple cores or even multiple hosts with readily available Unix programs such as *xargs* and *GNU parallel*. Also, if one of the processes fails, you have a manageable amount of data to examine, and you can easily rerun just the failed part.

On the other hand, also avoid working with many overly small files (of a few hundred bytes each). Processing a file involves a fixed overhead, so ensure that files carry their weight. Group small amounts of data into larger files or in a database. For simple unstructured columnar data, such as numbers, identifiers, and dates, delimiter-separated text files can be processed more easily and efficiently than structured formats, such as JSON and XML. If you can benefit from the flexibility and power of SQL queries but do not require the ACID (atomicity, consistency, isolation, durability) guarantees of a full-fledged relational database system, then using the SQLite embeddable database will make your workflow easier to deploy and maintain.

When dealing with large numbers of files, avoid storing them all in a single directory as this may hinder their efficient processing. Instead, organize them in a simple tree structure, for example, by year or the first digits of their Universally Unique Identifier. Add further tree levels until you have hundreds,

Digital Object Identifier 10.1109/MS.2024.3385665
Date of current version: 12 June 2024

rather than many thousands, of entries in each directory.

Furthermore, consider splitting your processing into intermediate steps. Thus, you can develop and test each step independently. As with split data, if a step fails, after fixing it, you can continue processing from that step onward. Often, Unix pipelines automatically allow a multistep job to utilize simply and efficiently the several cores of modern processors.

Splitting adds costs for interprocess communication, for file input/output (I/O), and for combining the upstream results, so consider the tradeoffs to avoid overburdening your workflow. Remember that keeping your data sorted on appropriate keys allows you to compare, merge, and join them efficiently with *O*(*N*) operations.

Keeping your data sorted on appropriate keys allows you to compare, merge, and join them efficiently with *O(N)* operations.

Work With Data Subsets

Rapid feedback is important in all engineering workflows. Big data, with processing latencies that can range from hours to months, can obviously derail this feedback process. To prototype, test, simulate a full run, and troubleshoot your solution without long delays, implement your tasks so that they can easily process only a subset of the data. The splitting I described in the previous section obviously helps here, but other approaches are also possible. For example, a command-line switch or a configuration item can set up the task to process only one day each month or only customers whose identification number ends in 242. A related approach is to sample a pseudorandom number generator to select only a subset of the records. For example, a condition based on `random() < 0.01` will sample 1% of the data. Seeding the random number generator with a fixed value ensures that the process is reproducible. This helps both your debugging efforts and your colleagues who might want to verify your findings.

While at it, consider whether processing all the data is actually required. For many tasks, a small random sample may offer you the results you require. Verify this through the cross-validation of the results' stability across multiple samples or by taking increasingly larger samples and looking at the results' quality, for example, through statistical hypothesis testing.

Log

Logging your workflow's progress allows you to monitor and forecast progress, to detect stuck jobs and sources of inefficiency, and to pinpoint errors. Log liberally: as a minimum, when a task starts, ends, the same for each independent processing step, and all errors. On the other hand, do not log within hot loops to avoid the corresponding performance impact. Instead, monitor a counter and log at intervals that provide you with reasonable progress feedback. Include in your log output all identifiers needed to unambiguously determine when, where, and what went wrong. This may include a timestamp (in UTC if you work across multiple time zones), hostname, the process identifier, and input or output file names. When working with a high-level language, it may be worth using a logging library so that you can easily configure the required logging at runtime. In shell scripts, you can achieve some of this functionality by performing logging through a common function. For shell scripts with infrequent command invocations, it can be sufficient to configure the shell to log them (`set -x`) together with a prefix specified with Bash's `PS4` variable. Keep log files along with the generated data so that you can refer to them if you find a problem months or even years later.

Some constructs make it difficult to inspect the data you process, but there are workarounds. You can inspect data flowing through Unix pipes by placing a *tee* command at the point you want to inspect. A file argument to *tee* will be the file receiving a copy of the data flowing through it. In vectorized processing, via Python's *NumPy* package or languages that deal directly with *n*-dimensional arrays, peeking inside array processing is also tricky. You can inject logging by structuring your code in the form of *apply(function, array)* and then instrument the function with the (temporary) logging you need.

Document

Processing workflows can be difficult to understand because they are often built around many diverse technologies: bespoke high-level language code, specialized tools, scripting and shell glue, databases, cloud services, and file system hierarchies. Help your colleagues and your future self who has forgotten how you designed the workflow by documenting all aspects. Provide a high-level overview in a README file as well as comments in each bespoke tool or script describing its purpose, expected input, generated output, and available configuration elements.

Internally, attach metadata to the data you generate so that you can easily determine their provenance. These can take the form of file names incorporating timestamps and other identifiers, comments in XML documents, a metadata object in a JSON file, or corresponding columns in plain text files or database tables. These metadata should allow you to trace back faulty data to the processes that generated them and the associated log files.

Expect Invalid Data and Failures

I have often posited the following Law of Big Data Analytics, "*Any sufficiently large dataset will contain elements that will trip your analysis tools*," and its obvious corollary, "*In any sufficiently large collection of positive integers, at least one value will be –1 or null.*" Data at any stage within your processing can be corrupt (violating a specification) or invalid (in allowed but unanticipated ways). Structure your processing to deal with these issues. One approach is to log such instances and handle them gracefully (for example, skipping or imputing the corresponding records) to avoid a few expected bad apples spoiling an expensive processing task. (The logging helps you ensure that the problematic records are indeed few and unfixable.)

When dealing with floating point numbers, you can prevent your accidental use of missing data by utilizing the hardware's support for handling invalid values. On input, rather than using zero as a stand-in for missing values, set these to the processor's not a number (NaN) representation. Arithmetic operations involving NaNs will propagate them to their result, making the use of missing values apparent in the output.

Another approach involves stopping the processing and providing enough data so that you can fix the issue and repeat the corresponding step. Add assertions liberally in your code to catch data-breaking invariant failures as early as possible. This is especially worthwhile when you initially prototype your solution and later when the polished workflow is running in a production environment.

For the case of invalid data or software failures, set up your workflow to provide you with all the troubleshooting data you need. Configure native code to include debug symbols and to record a core dump on failure, which you can inspect with a debugger. Configure higher level code, written, for example, in Java or Python, to provide a complete stack trace. Set up Unix shell scripts to terminate in the case of unanticipated errors (`set -eu -o pipefile`) and terminate them by exiting with a nonzero code when you encounter an error so that their callers will also fail.

Problems will also arise from hardware faults. Depending on your environment and setup, the most common ones are likely to be power failures and network outages. Both can result in interrupted processes and truncated output files. Deal with these by ensuring that your results appear ready through an atomic operation that takes place only when all processing completes correctly. When using a database, committing a transaction is an obvious solution. When writing files, generate them with a temporary name and rename them to the final one only when the task completes. Operating systems rename files atomically, ensuring your task's transaction integrity.

When dealing with thousands of hosts, networking, and storage devices, you can also bet on these failing or behaving in unpredictably wrong ways, often corrupting your data or results. Addressing these issues is beyond the scope of this article.

Automate a Reproducible Incremental Workflow

Manually running your workflow's steps is a recipe for disaster. You may miss or misconfigure steps, incorporate stale data, or entirely forget how the process worked when you revisit it in the future. Instead, express the required steps so that a single command can execute the entire process. You can do this with a script, or better, with a specialized tool, such as Unix *make*. Both approaches will also serve as your process's documentation. You can bootstrap the automation process with a log of executed statements, which allows you to recall the order of processes and their arguments. For this task, Python notebooks and the shell's history are your friends.

Implement your workflow to run tasks incrementally so that if you want to rerun only some downstream steps, you will not need to pay the price of running the upstream ones. At a very simple level, you can do this by skipping the generation of output files that already exist. A better approach is to represent all dependencies between tasks as a directed acyclic graph and run only those whose outputs are missing or are stale with respect to their dependencies. You can easily implement this idea with *Makefile* rules. Also, create a "clean" rule or process that will delete all intermediate and cached data so that you verify your process from the beginning to the end.

Ensure that your workflow remains reproducible over time by pinning the versions of dependencies you are installing through a package manager; by cloning Git repositories through release tags or specific commit hashes; by seeding random number generators with fixed values; by using persistent resource identifiers, such as digital object identifiers; and by maintaining local copies of fungible third-party data. Ideally, you should be able to run your process without using any external data sources, relying only on locally cached data.

Do Not Overengineer

The abundance of big data processing facilities brings the temptation to overengineer. Steer away from it! Before launching a cloud-based Spark/Flink/Hadoop/Kubernetes cluster, driven by a message queue, and storing data into a multimaster distributed relational database, see how far you can get by following the KISS (keep it simple, stupid) principle. Complex tools bring with them dependencies that over time will introduce breakages and the need for expensive maintenance. Therefore, choose your approach pragmatically, adopting the simplest solution that can reliably and efficiently deliver the goods.

Experiment, prototype, and adopt for production (if viable) workflows using plain (often text) files and your bespoke code, together with simple, powerful, mature, and widely available Unix tools, such as *curl, xargs, parallel, awk, sed, find, sort, join, comm, grep*, and *make*, glued through shell scripting.[5] I recently experienced the robustness of the Unix tools approach when I reran an almost unmodified Unix documentation typesetting workflow written in the early 1970s.

Optimize the Heavy Lifting

In processing tasks, typically only a small part of the code is responsible for most of the runtime cost. This is the part you should focus on optimizing. If you have coded your workflow in Python, the heavy lifting should be performed by libraries written in performant C/C++, such *as NumPy, SciPy, Pandas, TensorFlow, PyTorch*, and *scikit-learn*. Similarly, in shell scripts, the core processing should be carried out by heavily optimized Unix tools, such as *sort* or *grep*, or more specialized command-line tools running on large volumes of data. You should avoid having your hot loop perform its core processing in pure Python, in the Unix shell, or invoke a new process in each iteration.

Multiple levels of caching can help you further reduce the cost of expensive processing. Keeping processed data, rather than reproducing them from scratch, buys you speed and efficiency (at the expense of increased storage) when you need to rerun only a subset of downstream tasks.

Reduce the cost of textual data storage, communication, and I/O by storing them in a streaming-friendly compression format. The Zstandard program and its *zstd* implementation are particularly well suited, offering a high compression ratio, fast decompression, the utilization of multiple processing cores, and a robust checksum to detect data corruption. The employed compression format is append-friendly, allowing you to add more data to an already compressed file simply by appending them to it. Furthermore, using the *zstd* program's `--rsyncable` flag allows the efficient (partial) transfer of your data files with *rsync* after they have been slightly modified.

Follow Software Engineering Practices

Data processing workflows can be a key part of your organization's operations, so treat them with the care you would give to any production software artifact. Follow the established guidelines for writing maintainable code,[6] such as organizing the workflow along short, simple, and loosely coupled units serving distinct concerns; incorporating automated tests; and avoiding the accumulation of technical debt. Put your workflow's elements under version control, review changes, and establish a continuous integration process for verifying the workflow's quality through formatting checks, static analysis, and tests.

Where possible, also version your data. For small amounts of textual data, you can use your process's revision control repository. To keep your source repository lean, version larger blobs in a separate repository, in the file system, or through your cloud provider's storage services.

In essence, the successful engineering of data processing workflows merges the expertise, insight, and ingenuity of a data scientist with the methodical and disciplined approach of a software engineer.

ABOUT THE AUTHOR

DIOMIDIS SPINELLIS is a professor in the Department of Management Science and Technology, Athens University of Economics and Business, Athens 104 34, Greece, and a professor of software analytics in the Department of Software Technology, Delft University of Technology, 2600 AA Delft, The Netherlands. He is a Senior Member of IEEE. Contact him at dds@aueb.gr.

References

1. D. Spinellis, "Tools and techniques for analyzing product and process data," in *The Art and Science of Analyzing Software Data*, T. Menzies, C. Bird, and T. Zimmermann, Eds., San Mateo, CA, USA: Morgan-Kaufmann, 2015, pp. 161–212.
2. D. Spinellis and P. Louridas, "The carbon footprint of conference papers," *PLoS One*, vol. 8, no. 6, May 2013, Art. no. e66508, doi: 10.1371/journal.pone.0066508.
3. D. Spinellis, "Open reproducible scientometric research with Alexandria3k," *PLoS One*, vol. 18, no. 11, Nov. 2023, Art. no. e0294946, doi: 10.1371/journal.pone.0294946.
4. T. Evgeniou, M. Pontil, D. Spinellis, and N. Nassuphis, "Regularized robust portfolio estimation," in *Regularization, Optimization, Kernels, and Support Vector Machines*, J. A. K. Suykens, M. Signoretto, and A. Argyriou, Eds., London, U.K.: Chapman & Hall, Oct. 2014, ch. 11, pp. 237–256.
5. B. Kernighan, "Sometimes the old ways are best," *IEEE Softw.*, vol. 25, no. 6, pp. 18–19, Nov./Dec. 2008. doi: 10.1109/MS.2008.161.
6. Visser, J. *Building Maintainable Software: Ten Guidelines for Future-Proof Code*. Sebastopol, CA, USA: O'Reilly Media, 2015.

Digital Object Identifier 10.1109/MS.2024.3399275