CS425, Distributed Systems: Fall 2025
Machine Programming 4: Stream Processing

Released Date: Nov 11, 2025
Due Date (Hard Deadline): Sunday, Dec 7, 2025 (Code+Report due at 11.59 PM)
Demos on Monday Dec 8, 2025

This is a very intense and time-consuming MP! So please start early! Start now!
(Disclaimer: Like all MPs and HWs, all references to all companies and people in this spec are purely fictitious and are intended to bear no resemblance to any persons or companies, living or dead.)

ExSpace (MP3) miraculously merged with two companies to form a new conglomerate called…MartWall Inc. MartWall loved your previous work at ExSpace (and they're also aware of your great work on the mission to Saturn in HW3), so they've hired you as a "MartWall Fellow". That's quite prestigious! Congratulations!

You must work in groups of two for this MP. Please stick with the groups you formed for MP1. (see end of document for expectations from group members.)

MartWall needs to fight off competition from its three biggest competitors: Pied Piper Inc., Hooli Inc., and Amazing.com! So, they've decided to build a stream processing system that is faster than Storm. There are two parts to this MP.

1. Use HyDFS (from MP3) and the failure detector (MP2) to build a new stream-processing framework called RainStorm**,** which bears similarities to stream processing frameworks such as Storm, Spark Streaming, and MillWheel (https://static.googleusercontent.com/media/research.google.com/en//pubs/archive/41378.pdf).
2. **Compare** RainStorm's performance against the latest version of Spark Streaming (this involves deploying and running Spark Streaming on the VMs).

Downloading and deploying Spark to run on the VMs will take some time, and can be done in parallel, so please start early, and plan your progress with the deadline in mind. Starting even 2 weeks before the deadline will be too late, and you're unlikely to finish in time.

MartWall has provided detailed requirements for and some design elements of RainStorm. However, they want you to fill in the gaps in the design and to implement the system. Be prepared to improvise, to deploy new systems, and for the unknown. Move Fast and Break Things! (i.e., build some pieces of code that run, and incrementally grow them. Don't write big pieces of code and then compile and run them all!) RainStorm shares similarities with some of the above-mentioned stream processing systems, except that it is simpler. You can look at the docs and code of those systems, but not reuse any code; we will check using Moss. Also, it's faster to write your own

RainStorm than borrow and throw out code. In this MP, you should look to use code from MP1, MP2, and MP3.

For this MP, **you are allowed to use LLMs**. If you do, please first come up with a design and use LLM as a sounding board to improve your design. Uploading the entire MP spec as an input is likely to generate unmanageable code that will be difficult to debug. Instead, once you have a design you could use LLMs to generate small pieces of code. Your use and experience with the LLM must be included in the Report (below); you must also cite its use and include your sessions as files with your submitted code. Given the increased use of LLMs to generate code in the industry, these relaxed rules for MP4 serve as an educational purpose for you. However, note that (as in your future job) you and only you are responsible for the correctness and efficiency of any code that you obtain from LLMs. If something fails in the demo, it is your responsibility, not the LLM's. Proceed cautiously and judiciously.

RainStorm is a stream-processing framework where users specify a sequence of transformations on input data to produce one or more streams of output data. Unlike MapReduce, where one waits for the entire computation to complete, stream processing frameworks like RainStorm allow one to perform real-time analytics on a continuous stream of input data. The user-defined transformations run in parallel, and are not separated by barriers like in MapReduce. RainStorm can run on an arbitrary number of machines, but most of your experiments will be limited by the maximum number of VMs you have.

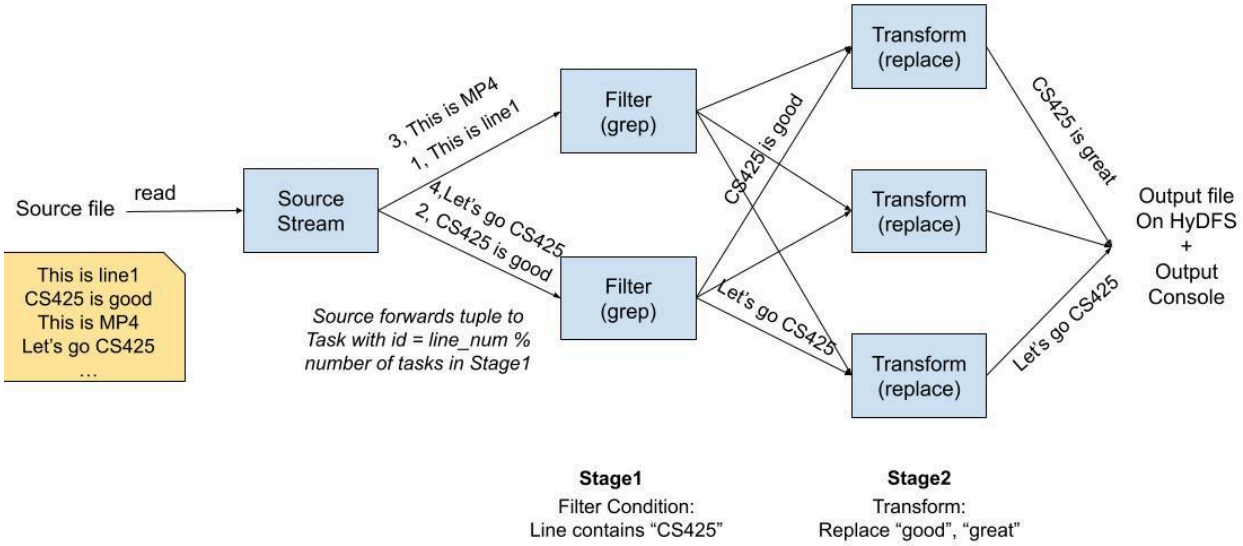
A stream is a sequence of <key, value> tuples. Stream processing frameworks support many stages of transformations. However, RainStorm supports no more than three processing stages. Additionally, RainStorm must read input files from HyDFS and use them to produce the source stream of <filename:linenumber, line> tuples for the first stage to consume.

Like MapReduce, each stage must have parallel tasks. Each task in a stage must process one or more input streams from the previous stage and produce one or more streams for tasks in the next stage. The stream from the final stage must be output to the console (continuously) **and** be appended to a single output file in HyDFS. Data streaming into a stage is partitioned among the tasks of the stage uniformly (you could use hash partitioning).

RainStorm tasks support three types of operators:

1. Transform: apply a user-defined function on each tuple, and send to the next stage (e.g., Replace)
2. Filter: filter out tuples that don't satisfy a condition, and send the remaining to the next stage (e.g., Grep)
3. AggregateByKey: maintain a running aggregate statistic of all records seen so far and pass the aggregate to the next stage (e.g., count grouped by a key). Note that this type of operator must remember the state (i.e., the aggregate statistic) across invocations, unlike the previous two.

Given the above, a grep+replace streaming application can be implemented as follows:



First, a source process continuously reads from the source file on HyDFS, produces a stream of lines, and sends it to tasks in the first stage based on the line number. Each of multiple parallel tasks in the first stage (Grep) filters out lines that do not contain the user-specified grep pattern, and forwards only lines with the pattern to the second stage. Each of multiple parallel tasks in the second stage (Replace) replaces a user-specified grep pattern in each line with another user-specified replace pattern, and continuously outputs the lines to the console and to an output file in HyDFS.

Like in MapReduce, each stage must have multiple tasks. The stream between two stages is partitioned by a key. For example, the key for the input to the Grep stage is file:linenumber, whereas the key for the input to the Replace stage is the line. You must *partition* the incoming stream to a stage such that each task receives approximately the same number of tuples. You could use hash partitioning on the key modulo the number of tasks in the next stage. Each key must be assigned to exactly one task in a stage. Unlike MapReduce, there is no barrier between stages. Each task forwards any tuple it has processed immediately to the next stage, and tasks in the last stage must output tuples to the console and append them to the HyDFS output file.

We will invoke RainStorm as follows.

RainStorm <Nstages> <Ntasks_per_stage> <op1_exe> <op1_args> … <opNstages_exe> <opNstages_args> <hydfs_src_directory> <hydfs_dest_filename> <exactly_once> <autoscale_enabled> <INPUT_RATE> <LW> <HW>

The first parameter specifies the number of stages in the application. The second specifies the number of tasks in each stage. The next is a series of parameter op_exe and op_args pairs, one per stage, which are the user-specified executable and arguments for that stage. Each exe takes as input a <key, value> tuple and produces zero or more <key,value> tuples. In our example application, op1_args can be the grep pattern to be

used by op1_exe from stage1. The two parameters with the hydfs_ prefix specify the locations of the input files and the final output file. The next two parameters (exactly_once & autoscale_enabled) are booleans to control the toggling of two features. And the last 3 parameters need to be specified only if autoscale_enabled is true. More on them below.

Note that op1_exe, op2_exe, etc., are custom programs that you will write based on the stream processing application (similar to how one writes custom Map and Reduce functions). We will specify the exact stream processing application when we release the demo instructions. Thus, your framework must take arbitrary op_exe programs and op_args that filter/transform a tuple and produce one or more tuples for each stage.

Design: The RainStorm cluster has N (up to max number of VMs) server machines. One of them is the leader, and the remaining N-1 are worker servers. Similar to MapReduce, the tasks are scheduled on a worker machine. The leader is responsible for all critical functionalities, including receiving commands, scheduling tasks, determining the hash partition functions, and handling failures of the workers. In short, any coordination activity is done by the leader. The leader should distribute the tasks approximately evenly among the VMs; the number of overall tasks in RainStorm is not limited by N. Additionally, we assume that the leader will never fail.

Worker failures must be tolerated. When a worker fails, the leader must restart the task quickly and add it to the appropriate stage. Worker failures must not result in incorrect outputs (more below).

Mandatory #1: Exactly-once semantics. RainStorm supports **exactly-once semantics even in the presence of failures**, i.e., each input tuple is processed exactly once by each stage. This feature is toggled by the exactly_once command-line parameter. A failure is triggered by killing a specific task of a stage. Note that only the specific RainStorm task is failed, and not HyDFS tasks. That is, **HyDFS can be assumed to be always available.**

To provide this semantics, each stage must track which tuples it has processed and which tuples have been processed by the next stage; the Source Stream "stage" tracks only the latter. Each stage also acknowledges the (sender task from the) previous stage once it has processed a tuple. If a sender task doesn't receive an ack on time, it re-sends the tuple until it receives an ack. A receiver task must check if a (re-sent) input tuple is a duplicate to avoid processing it a second time; it must discard duplicates but ack them. How can a task detect duplicate tuples? One method is for a sender task to associate a unique ID with each outgoing tuple and for the receiver task to use that ID for such detection.

To support exactly-once semantics across task failures and restarts, each task must persist the aforementioned tracking; in other words, each task maintains this tracking in its own append-only log file in HyDFS. Each task has an identity, and the leader must ensure that a restarted task inherits the corresponding failed task's identity, even when

the task is restarted on a VM that is different from the failed task's VM. The restarted task can use that identity to locate and open its log file in HyDFS. A restarted task can look at (replay) its log file[1] to figure out which input tuples it has processed (and ack'ed), and which output tuples have been ack'ed by the next stage. If a task fails, the leader must restart the task to bring the total number of tasks back to the level it was before the failure.

Note that we will fail only tasks that implement filter or transform operators and **not fail stateful ones that implement aggregation**. RainStorm must still filter incoming duplicates to aggregation tasks, but need not worry about handling failure of the aggregation tasks themselves.

If writing to HyDFS after processing every tuple is slow, you may consider batching as an optimization. For example, you can batch multiple tuples and interact with HyDFS once for a batch of input tuples.

Mandatory #2: Autoscaling. RainStorm supports dynamic adjustment of #tasks in each stage based on processing load. This feature is toggled by the autoscale_enabled command line parameter. It is not easy to design exactly-once semantics to interact accurately with dynamic changes in the number of tasks in a stage. Therefore, to keep things simple, **we will disable exactly-once semantics when the autoscaling feature is enabled**. However, we expect at-least-once delivery semantics for tuples during any autoscale-triggered changes to #tasks in a stage. In other words, we expect that tuples are never dropped. Note: you can also assume that no failures will be triggered (task or VMs) during any runs with autoscale_enabled set to true.

Rainstorm should support dynamic adjustment of tasks as follows. A source process feeds into the tasks of the first stage. The Source process must sustain the stream it sends into the first stage at INPUT_RATE tuples(lines)/second. If the input load to a stage drops below an average low-watermark of LW tuples/second per task, the system must decrease the number of tasks in that stage by one. And, if the input load to a stage goes above an average high-watermark of HW tuples/second per task, the system must increase the number of tasks in that stage by one. Any change made by the system to #tasks in a stage within N seconds of its input load crossing a watermark. Each experiment/demo will start with Ntasks_per_stage per stage, and the system will scale up or down the number of tasks in a stage based on the watermarks and the input rate to that stage. The system must detect the input rates of each stage and scale the stage automatically, and not just react to the INPUT_RATE parameter, which controls only the first stage's input rate. All three values, INPUT_RATE, LW, and HW, are specified as parameters to the RainStorm command line.

One way to design autoscaling is to have a central ResourceManager (RM) module running on the leader VM. RM could monitor the per-task load on each stage and react

[1] Since HyDFS tasks never fail, the RainStorm task can always access the latest log file. Please run a "merge" command before the RainStorm task replays the log.

to the per-stage input load by dynamically increasing or decreasing the number of tasks in a given stage. Note that we **don't require RainStorm to autoscale a stateful stage**; i.e., the RM need not monitor and autoscale a stage implementing aggregation .

You must use the code for MPs 1-3 in the RainStorm system. Use MP1 to log messages for debugging, MP2 to detect failures, and MP3 to store the inputs, outputs, and intermediate data for RainStorm.

Once you have your code working, your first test must confirm that your flow-rates measured by each task and reported to the RM track the INPUT_RATE from the source. In other words, create 2 tasks per stage, have each stage simply pass through each tuple (i.e., make `op_exe` an identity function), and ensure that the measured and reported (by RM) tuple/sec rate across all tasks in each stage is equal to the INPUT_RATE.

Create logs (of debug messages)---one per worker machine and one on the leader (including the RM) machine—that can be queried (via MP1 or via grep). You can make your logs as verbose as you wish (for debugging), but at the minimum, each machine must log when it starts/restarts a task, and the leader machine must log the start and end of the entire run, the start and end of each task, restart of a task after a task failure, and any autoscale-triggered start or stop of a task. We will request to see the log entries at demo time, either via local grep or using the MP1's querier.

Other parts of the design are open, and you are free to choose. Design first, then implement. Keep your design and implementation as simple as possible. Use the adage "KISS: Keep It Simple Si…". Otherwise, MartWall Inc. may, in their anger at your complex design, fire you into space (or worse, into the deep sea next to the Titanic), with only a book to read. That would be a very boring vacation, right?

We also recommend (but don't require) writing tests for basic scheduling operations. In any case, the next section tests some of the workings of your implementation.

Spark: Download Spark Streaming from https://spark.apache.org/streaming/ and run it on your VMs (this step will take some effort, so give it enough time!). Compare the performance of RainStorm with SparkStreaming, and **see if you can make RainStorm faster.** Make the comparison *fair*. In other words, run the job with the same settings for both systems, i.e., in the same cluster on the same dataset and for the same topology (#tasks per stage). Most of the inefficiencies in RainStorm will be in accessing storage, so think of how your files are written and read. To measure performance, use output tuples produced per time. Are you able to beat SparkStreaming?

Datasets: Use or create synthetic datasets that are at least 100s of MBs large in your experiments. Smaller datasets may suffice for testing, but ensure that run times are at least several tens of seconds, especially when comparing with Spark.

Good places to look for datasets are the following (don't feel restricted by these):

- Champaign Map Databases are available at: https://gis-cityofchampaign.opendata.arcgis.com/search?collection=Dataset. Try queries like finding the number of parking meters or the number of apartment buildings with > 100 residents, etc. Look for other "GIS" databases.
- Stanford SNAP Repository: http://snap.stanford.edu/
- Web caching datasets: http://www.web-caching.com/traces-logs.html
- Amazon datasets: https://aws.amazon.com/datasets/
- Wikipedia Dataset: http://www.cs.upc.edu/~nlp/wikicorpus/

Machines: We will be using the CS VM Cluster machines. You will be using all your VMs for the demo. The VMs do not have persistent storage, so you are required to use git to manage your code. To access git from the VMs, use the same instructions as MP1.

Demo: Demos are scheduled on the Monday right after the MP is due. The demos will be on the CS VM Cluster machines. You will use up to the max VMs for your demo (details will be posted on Piazza closer to the demo date). Please make sure your code runs on the CS VM Cluster machines, especially if you've used your own machines/laptops to do most of your coding. Please make sure that any third party code you use is installable on CS VM Cluster. Further demo details and a signup sheet will be made available closer to the date.

Language: Choose your favorite language! We recommend C/C++/Java/Go/Rust/Python.

Report: Write a report of not more than three pages (12 pt font). Briefly describe the following (we recommend your report contain headings with bold keywords):

a) **Design**: Briefly describe your design (including architecture and programming framework) of RainStorm.

b) **Past MP Use**: (very briefly, 1-2 sentences) how MP2 and MP3 were used in MP4 and how useful MP1 was for debugging.

c) **LLM use**: If you used LLMs, describe in <100 words your usage and experience---especially how beneficial or not it was---and mention the filenames in your code submission that contain your chat sessions.

d) **Measurements** (measure real numbers, do not calculate by hand or calculator!): see below. Please label each plot/data clearly so your report is easy to read.

Show plots comparing RainStorm's performance to SparkStreaming for two different datasets. You can pick from the aforementioned datasets or create/find your own. In any case, your report must include a brief description of the two datasets used. If created, include in your git repo the script/code used to generate it. If found, include the source and web-link in your report. For each dataset, (describe in your report) and

implement TWO streaming applications on that dataset (each with at least 2 stages); the grep+replace described above is one such example. So, there should be a total of FOUR experiments. Your report should include a description of each of these applications.

For each data point on the plots, take at least 3 measurements, plot the average (or median) and standard deviation. Run each experiment (for each system) on all the VMs in your allocation. **Devote sufficient time for doing experiments** (this means finishing your implementation early!). Discuss your plots, don't just put them on paper, i.e., discuss trends, and whether they are what you expect or not, and why. (Measurement numbers don't lie, but we need to make sense of them!) Stay within the page limit.

Submission: (1) Submit your report to Gradescope BEFORE the deadline, and TAG your page(s) on Gradescope (there will be a penalty if you don't). Only one group member should submit – you must tag your partner.
(2) There will be a demo of each group's work. Signup sheets will be posted on Piazza.
(3) Submit your working code via gitlab sharing (including your group number in your project name). Please include a README explaining how to compile and run your code. Default submission is via gitlab sharing – please include your group number in your gitlab share names! Further submission instructions will be posted on Piazza.
Other submission instructions are similar to previous MPs.

When should I start? Start NOW. You already know all the necessary class material to do this MP. Each MP involves a significant amount of planning, design, and implementation/debugging/experimentation work. **Do not** leave all the work for the days before the deadline – **there will be no extensions**.

Evaluation Break-up: Demo [60%], Report (including design and plots) [30%], Code readability and comments [10%].

Academic Integrity: You cannot look at others' solutions, whether from this year or past years. We will run Moss to check for copying within and outside this class – first offense results in a zero grade on the MP, and second offense results in an F in the course. There are past examples of students penalized in both those ways, so just don't cheat. You can only discuss the MP spec and lecture concepts with the class students and forum, but not solutions, ideas, or code. If we see you posting code on the forum, that's a zero on the MP. MartWall Inc. is watching and will be very Sad!

Barring exceptional circumstances, we recommend you stay with your prior MP group. We expect all group members to contribute equivalently to the overall effort. If you believe your group members are not, please have "the talk" with them first, give them a second chance. If that doesn't work either, please approach one of the instructors.

Happy Streaming (from us and the fictitious MartWall Inc.)!