**Technical University of Crete**
School of Electrical and Computer Engineering

**PLH 613 – Special Topics in Database Systems** – 2025

*Fall Semester 2025* **Semester Project** **Due: 08.01.2026**

# Distributed Skyline Query Processing

with Apache Flink

**Student Team:**

**Spyridon Stamou**
2021030090

**Georgios Angelos Gravalos**
2021030001

# Contents

**Introduction** . . . . . **ii**

**Algorithms** . . . . . **iii**
MR-Dim . . . . . iv
MR-Grid . . . . . iv
MR-Angle . . . . . vi

**System Architecture** . . . . . **vii**

**Configurations** . . . . . **viii**

**Implementation** . . . . . **x**

**Experiments** . . . . . **xiv**
Experiment 1 . . . . . xiv
Experiment 2 . . . . . xv
Experiment 3 . . . . . xvi

**Conclusion** . . . . . **xviii**

# Introduction

The Skyline Operator is a fundamental tool for multi-criteria decision-making, particularly in Web Service selection where users balance conflicting attributes like cost and response time. Given a dataset $S$, a point $P$ is part of the Skyline if it is not dominated by any other point which means that no other point is better or equal in all dimensions and strictly better in at least one.

However, calculating the Skyline is a difficult task when dealing with huge amounts of data. As the volume of data increases, traditional algorithms that run on a single computer become too slow and cannot provide results in real-time. To solve this scalability problem, we need to split the work across multiple computers using distributed systems.

In this project, we build a distributed Skyline query processing system using Apache Flink and Apache Kafka. Our approach is based on the MapReduce model, where data is partitioned into smaller groups and processed in parallel. Specifically, we implement and compare three partitioning strategies proposed in the paper (`MapReduce Skyline Query Processing with A New Angular Partitioning Approach`): MR-Dim, MR-Grid, and MR-Angle. The MR-Dim algorithm splits data based on a single dimension, MR-Grid divides the space into cells, and MR-Angle uses angular coordinates.

The goal of this project is to reproduce these three algorithms and evaluate which one performs best, in order to verify the results reported in the paper

# Algorithms

We have to define the `Skyline Query`. Let $P$ be a set of data points in a $d$-dimensional space, where each point $p \in P$ consists of $d$ attributes $\{p_1, p_2, \ldots, p_d\}$. Assuming that for all dimensions smaller values are preferred, we say that a point $p$ dominates another point $q$ if the following two conditions are met:

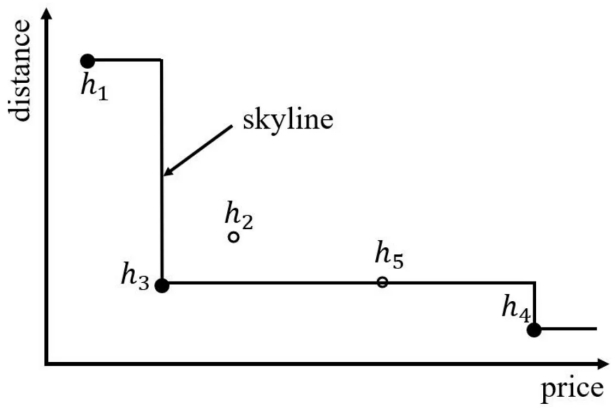
- $p$ is as good or better than $q$ in all dimensions:

$$p_i \leq q_i$$

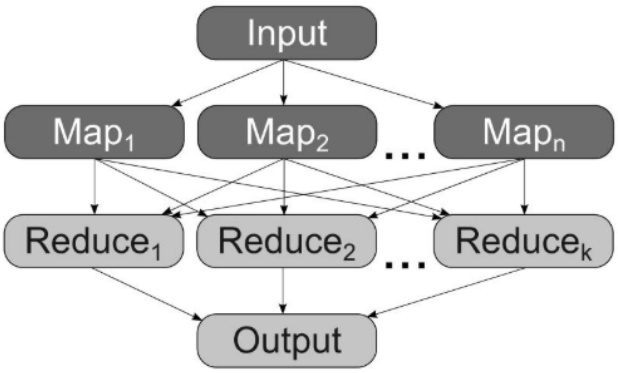
- $p$ is strictly better than $q$ in at least one dimension:

$$p_k < q_k$$

So, the `Skyline` of $P$ is the subset of points that are not dominated by any other point in the dataset.




Before describing the three specific algorithms, it is important to explain the general `MapReduce` logic that all of them follow. The main idea is to split the massive dataset into smaller pieces using a partitioning strategy. The process works in two simple phases:




- **Map Phase:** First, the dataset is split into smaller groups, partitions. Each worker takes one group and finds the no-dominated points within that specific group, (Local Skyline). Any point that is dominated here is thrown away immediately, as it cannot be part of the final result.

- **Reduce Phase** The surviving points from the Local Skylines from all workers are sent to a central node. Here, we compare them one last time to check if they are dominated by points from other previous groups. The points that survive this final check constitute the final Global Skyline of the dataset.

We can proceed to analyze the three specific partitioning algorithms

## MR-Dim

The MR-Dim algorithm operates in two distinct stages: the Partitioning Job and the Merging Job. In the first stage, the dataset is divided into subspaces based strictly on the values of a single dimension. The number of partitions, denoted as $N_p$, is set to be the number of available worker nodes.

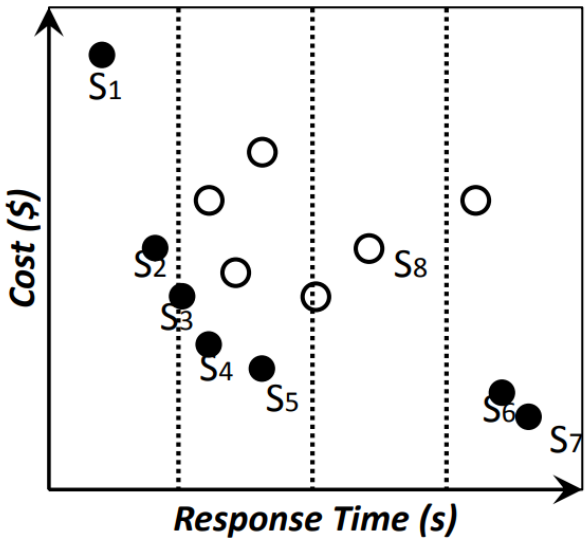
$V_{max}$ represents the maximum value of the selected dimension, the range of each partition is defined as

$$Range = \frac{V_{max}}{N_p}$$

Every data point is assigned to a specific partition ID based on the formula:

$$PartitionID = \left\lfloor \frac{value}{Range} \right\rfloor$$

After the data is split, each worker independently calculates the Local Skyline using the standard Block Nested Loop algorithm. Finally, all these local results are gathered in one central place. There, they are compared against each other one last time to remove any remaining dominated points and produce the final Global Skyline.




## MR-Grid

As mentioned in various discussions during class, we did not have to modify any code in terms of architecture for Flink, we only had to modify the `Algorithms.scala` file to provide the algorithm's implementation, as well as the `Job.scala` to properly call the algorithm, same as

MR-Dim.
The MR-Grid also operates in the Partitioning Job and Merging Job stages. Our objective is to decompose the original dataset into smaller subspaces, compute the local skylines, then merge these results to produce the global skyline.
In constrast to MR-Dim, which parittions the dataset using one dimension, the MR-Grid partitions the dataspace in multiple dimensions simultaneously (2, in our case). In the 2D case, the dataspace split results in rectangular partitions.
Let us denote as $N_p$ the number of workers, then to construct a balanced grid, each dimension is divided in $g = \lfloor \sqrt{N_p} \rfloor$ equal ranges, resulting in total $g^2$ grid cells (partitions).
Since we have 4 partitions for this job, $g = 2$ cells per dimension.

Assuming the max value of each dimension is $V_{max}$, the size of each grid cell along a dimension is

$$Cellsize = \frac{V_{max}}{g}$$

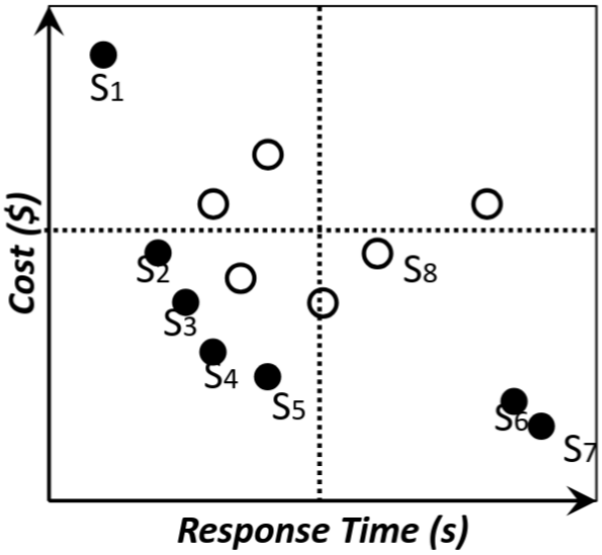
Each data point $(x, y)$ is assigned to a cell based on its coordinates. The cell indices in each dimension are computed as:

$$i_x = \lfloor \frac{x}{Cellsize} \rfloor, i_y = \lfloor \frac{y}{Cellsize} \rfloor$$

The final partitionID is obtained by linearizing the 2D cell coordinates:

$$partID = i_x \cdot g + i_y$$

With this scheme, each point is assigned to exactly one cell, therefore to one worker node.
After partitioning, each worker independently produces the local skyline of its own points, using the BNL skyline algorithm. Due to the tighter bounds imposed by the grid partitions, the number of skyline points per partition tends to be less compared to MR-Dim, which means more efficient local processing.
Once all the local skylines have been computed, and the trigger has been given to Input2, the results are collected in a central merging phase. The union of all local skylines is processed one more time to eliminate any dominated points, and the global skyline is returned.



## MR-Angle

The MR-Angle follows the same 2-stage execution model as the previous two algorithms. The key distinction between them is the way that the dataspace is partitioned. MR-Angle partitions the dataset based on the angular orientation of the data points.
The logic behind this, is that partitioning the data according to angular direction groups "similar" points together (about the same $y/x$ ratio), which can lead to smaller local skylines.

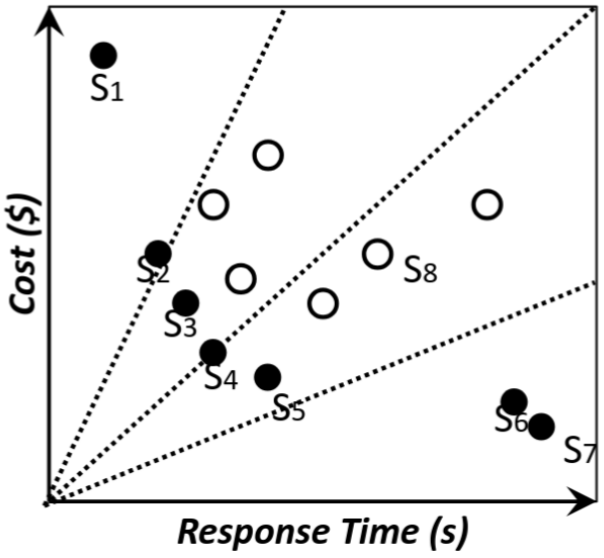
Let's once again denote $N_p$ the number of worker nodes.
In this implementation, the data points are assumed to all lie in the first quadrant ($y \geq 0, x \geq 0$), since both coordinates are generated in $[0, V_{max}]$. Therefore, the angle domain is limited to $\theta \in [0, \pi/2]$.
For a data point $(x, y)$, its angular coordinate $\theta$ to the x-axis is $\theta = arctan(\frac{y}{x})$.
In practice, to properly handle all edge cases (such as those who would return an undefined result for arctan), we compute the angle using the 2-argument arctan function $\theta = atan2(y, x)$.
All negative angles are fixed to 0, ensuring that all points are mapped in $[0, \pi/2]$.
This range is then divided into $N_p$ same size sectors, each with angular width $width = \frac{\pi}{2N_p}$.
Each data point is assigned to a partition ID, based on the sector that its angle is in :

$$partID = \lfloor \frac{\theta}{width} \rfloor$$

Following this strategy, each data point is assigned to one angular sector, and therefore one worker node.

After the partitioning part, each worker once again independently computes the local skyline of the points in its own angular sector, using the BNL algorithm.
Having grouped together points with similar directional characteristics, the MR-Angle reduces the number of mutually "incomparable" points in a partition. As a result, the local skyline produced by each worker tends to be smaller than those generated by MR-Dim or Grid.

Once again, when all the local skylines have been computed, their results are collected furing the merging phase, and their union is processed a final time to remove any dominated points. The result is the global skyline.



# System Architecture

Our system architecture follows a traditional pipeline pattern, consisting of data sources, processing operations, and a sink.

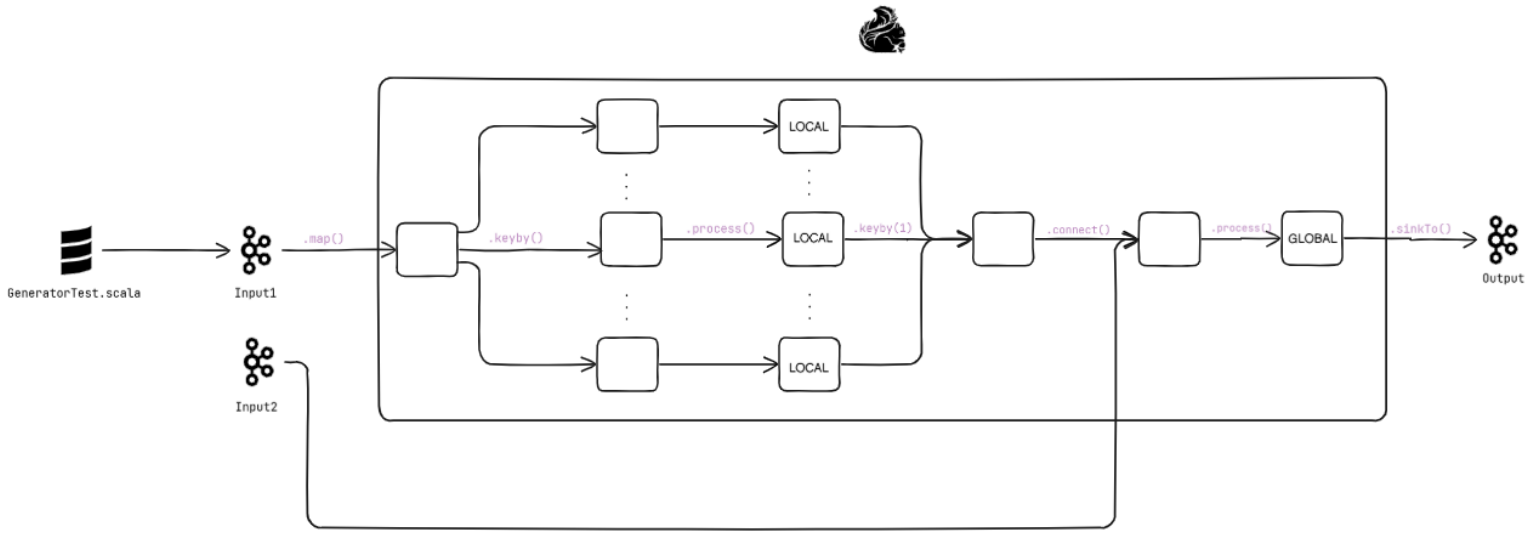
As input, we use two Kafka Topics. The first Kafka topic, `Input1`, is connected to the `GeneratorTest.scala` module. This component produces approximately 15,000,000 random tuples $(x, y)$, which are fed into the system. These points are then distributed to the workers to compute the local skyline before being forwarded to the global operator.

There is also a second input, `Input2`, which is connected directly to the Global Operator. This stream essentially "waits" for a user request to trigger the calculation of the skyline query.

After the final skyline is computed, the results are transferred to the output, which is another Kafka Topic named `Output`. The query results appear there at the specific moment the Global Operator receives the "query" from `Input2`.

In the intermediate stage, we have the operations performed within Flink. The data is ingested through `map()`, and using the `keyBy()` operator, it is partitioned among workers according to the specific algorithm. The local skyline is executed and computed at this stage. These local results are then gathered, connected with the `Input2` stream, and sent to the Global Operator. As mentioned, the Global Skyline is calculated only when triggered.

The pipeline described above is represented in the following figure, which illustrates our system's implementation. Having established this architecture, we proceeded to implement the three algorithms as defined in the paper:



# Configurations

We implemented our project in `Scala`, utilizing `JDK 11` as the underlying environment. We defined the `build.sbt` file, which serves as the blueprint for our project. Within this file, we set the Scala version to `2.12` in order to be compatible with Apache Flink. We also set the Flink version to `1.16.2` to ensure consistency across all dependencies.

Additionally, we imported some necessary libraries. These allow us to use the DataStream API and to submit jobs to the cluster using the Web UI. We also included specific libraries to handle the connection with Apache Kafka.

build.sbt

```
name := "FlinkSkyline"

version := "0.1"

scalaVersion := "2.12.15"

val flinkVersion = "1.16.2"

resolvers += Resolver.mavenCentral

libraryDependencies ++= Seq(
  "org.apache.flink" %% "flink-streaming-scala" % flinkVersion,
  "org.apache.flink" %  "flink-clients"        % flinkVersion,
  "org.apache.flink" %  "flink-runtime-web"    % flinkVersion,
  "org.apache.flink" %  "flink-connector-kafka"% flinkVersion


)

libraryDependencies += "org.apache.flink" % "flink-connector-kafka" % "3.0.1-1.17"

Compile / run / fork := true
```

We leverage Docker to implement the services needed for the execution. These include Zookeeper and Kafka, which handle the data sources of the system. Furthermore, we set up the JobManager and TaskManager for the execution in Apache Flink.

We define these services in a `docker-compose.yml` file. This setup allows us to easily start the cluster and scale the TaskManagers to 4 nodes for parallel processing.

A special parameter that we define is the `taskmanager.numberOfTaskSlots` in the TaskManager configuration. We set this value to 1. By doing so, we ensure that each Docker container corresponds to one worker node. This setup is the key to achieving true parallelism in our experiments.


The experiments to check the algorithms' performance took place on a machine with an `Intel Ultra 7 155H` processor and 32GB of RAM.

# Implementation

## Data generation

Data generation is handled in `Generator.scala`. The generator produces 15 million 2D points $(x, y)$, with each coordinate being double values, in the range $[0, 1000]$.

The generator supports three distinct data distributions: *Uniform*, *Correlated*, and *Anti-Correlated*. For the correlated and anti-correlated datasets, points are generated along the diagonal ($y \approx x$) or anti-diagonal ($y \approx 1000 - x$) respectively, with a Gaussian noise factor of $\pm 5\%$.

Each point is serialized as a comma-separated string, and is sent to the Kafka "Input" topic. We ran the generator as a standalone Scala application, from Intellij, and it continuously produces and publishes the points until the designated number of tuples is reached.

## Skyline operators

The skyline computation logic can be found in `Operators.scala`.
A custom Point class consists of a unique identifier, and a list of numerical values representing the dimensions.
In our project, the points have two dimensions $x, y$.

The Dominance logic is implemented with a a weak and a strict constraint:

- **Weak dominance** : A point $p$ dominates $q$ if $p_i \leq q_i$ in all dimensions i
- **Strict dominance** : A point $p$ strictly dominates $q$, if it weakly dominates q, and $p_j < q_j$ in at least one dimension.

We have incorporated these constraints in Scala and they are used across local and global skylines.

## Local Operator (Operator1)

The local skyline operator Operator1 runs locally in each Flink partition, uses the Block Nested Loop algorithm, maintains an in-memory collection od candidate skyline points, and for each incoming point it removes any points that are dominated by another existing skyline point.
This operator continuously produces an updated local skyline per partition.

## Global Operator (Operator2)

The global skyline operator Operator2 is lazy, it runs upon a trigger.
It receives the union of all local skylines, from all partitions, as well as the trigger signal from the Kafka topic Input2.
ONLY upon receiving the trigger, all collected skyline points are compared, all remaining dominated points are removed, and the final skyline is produced and pushed to the Kafka topic Output, which can be seen in console.

## Partitioning Algorithms

The `Algorithms.scala` file contains the implementation of the 3 partitioning algorithms, MR-Dim/Grid/Angle.
We will not look at them in depth here, since this has been done in the "Algorithms" section of the report.
Each algorithm computes a partitionID used by Flink's keyBy() operator. A short recap is:

- **MR-Dim:**
  - Partitions the data space using one dimension
  - Uses the maximum value for all x,y : $V_{max} = 1000$
  - The partition range = $V_{max}/N_p$
  - PartitionID = $\lfloor value/range \rfloor$
- **MR-Grid:**
  - Uses both dimensions, not just one, in constrast to MR-Dim
  - Computes number of equal ranges $g = \lfloor \sqrt{N_p} \rfloor$
  - Divides each dimension in $g$ grid cells
  - Each point is mapped to a grid cell $partID = i_x \cdot g + i_y$, point assigned to exactly one partition
- **MR-Angle :**
  - Computes the angular coordinate $\theta = atan2(y, x)$
  - points generated with $x, y \geq 0$, therefore angles are restricted to 1st quadrant. fix negative angles to 0, $\theta \in [0, \pi/2]$
  - Divides the circle into $N_p$ angular sectors (number of workers)
  - $width = \pi/2N_p$
  - PartitionID = $\lfloor \theta/width \rfloor = \lfloor \frac{\theta \cdot N_p}{2\pi} \rfloor$

All 3 algorithms return an integer partition key and share the exact same execution pipeline, nothing else changes in the project to serve the execution of a specific algorithm.

## Job composition

`Job.scala` defines the Flink streaming job and connects all the components.
Key responsibilities include:

- Parsing the arguments in the terminal (Kafka bootstrap server, Input/Input2/Output topics, algorithm selection dim/grid/angle, parallelism, max value).
- Initialization of the Flink execution environment
- Configuration of Kafka producers/consumers

The job reads from input (generator), parses each tuple to a Point, applies keyBy() using the selected algorithm, runs the local skyline operator continuously in all partitions, collects local skylines, awaits for the trigger from topic Input2. Once the trigger is sent, it executes the global skyline, and publishes results to topic Output.

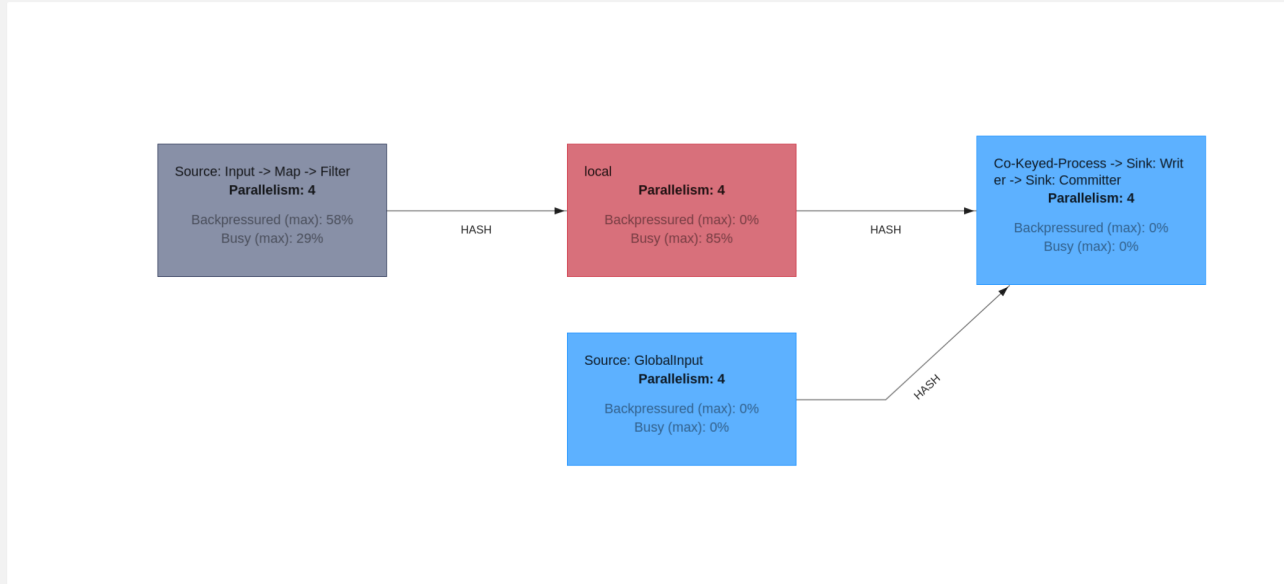
In the Scala code, a switch-case is used to select the algorithm at runtime without changing the pipeline structure.

For example, a typical execution follows these steps:

- Start the Flink cluster and Kafka brokers using Docker, in terminal #0
- In terminal #1, submit the job with the desired MR algorithm
- In terminal #2, open a Kafka consumer on the Output topic
- In terminal #3, open a Kafka producer on the Input2 topic.
- Run the Generator.scala from Intellij, and wait a bit
- When you're ready, send ANY string via terminal #3 to topic Input2, which will trigger the computation of the global skyline
- Head over to terminal #2 and see the result of the global skyline at topic Output's terminal

## Execution Graph

When launching a job, we can head to localhost:8081 (Flink Web GUI) to see the job execution graph.
For example, when **testing with MR-Dim**:



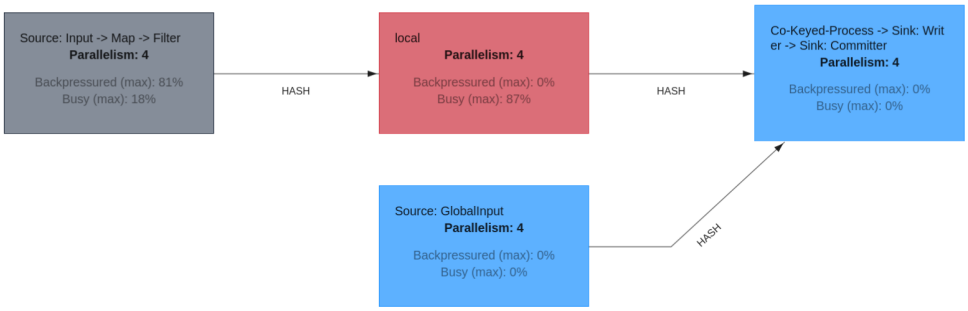

The graph can be explained from our code in `Job.scala`.
In the **left box**:

- "`Source:Input`" is the Kafka consumer reading from the Input topic.
- "`-> Map`" is the mapping we do to each Kafka message x,y into Point(id, List(x,y)).
- Immediately after, "`->Filter`" is the filtering we do to discard the failed parsed x,y.

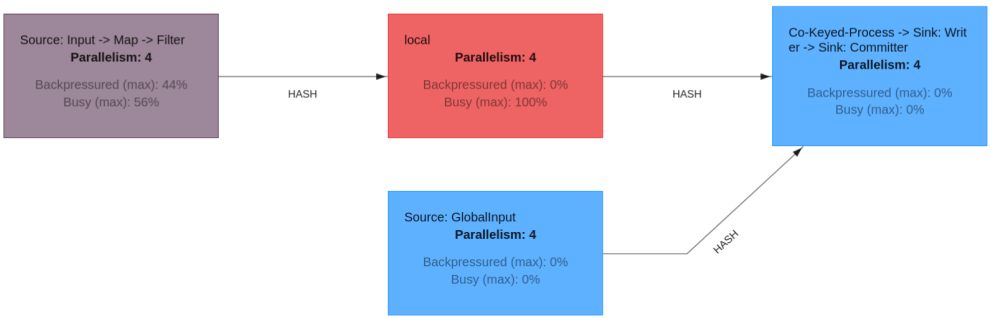
Then, the arrow **HASH** is the `keyBy()`. This is the partitioning stage.
Then, the two boxes in the middle reflect our implementation. **The upper box (Local)** is our local skyline operator (Operator1), which applies the BNL logic and runs the local skyline continuously. This is why it appears red, declaring it's the busy operator (constantly doing work).
**The lower box (Input2)** showcases our trigger input topic, the lazy logic behind the execution of the global skyline.
**The right box** is the global skyline operator (Operator2). Co-keyed process means that we have two streams : localSkylines (output of Operator1), and globalInput (Kafka Input2, the trigger) and we connect them. This forces all events into one logical key group, so that all local skyline points and trigger instances arrive at the same operator instance.
The percentages we see in the boxes are:

- **Busy (%):** time spent actively processing records
- **Backpressured (%) :** time spent blocked, because downstream operators can't accept more data.

The same logic holds for **MR-Grid**:




and for **MR-Angle**:



# Experiments

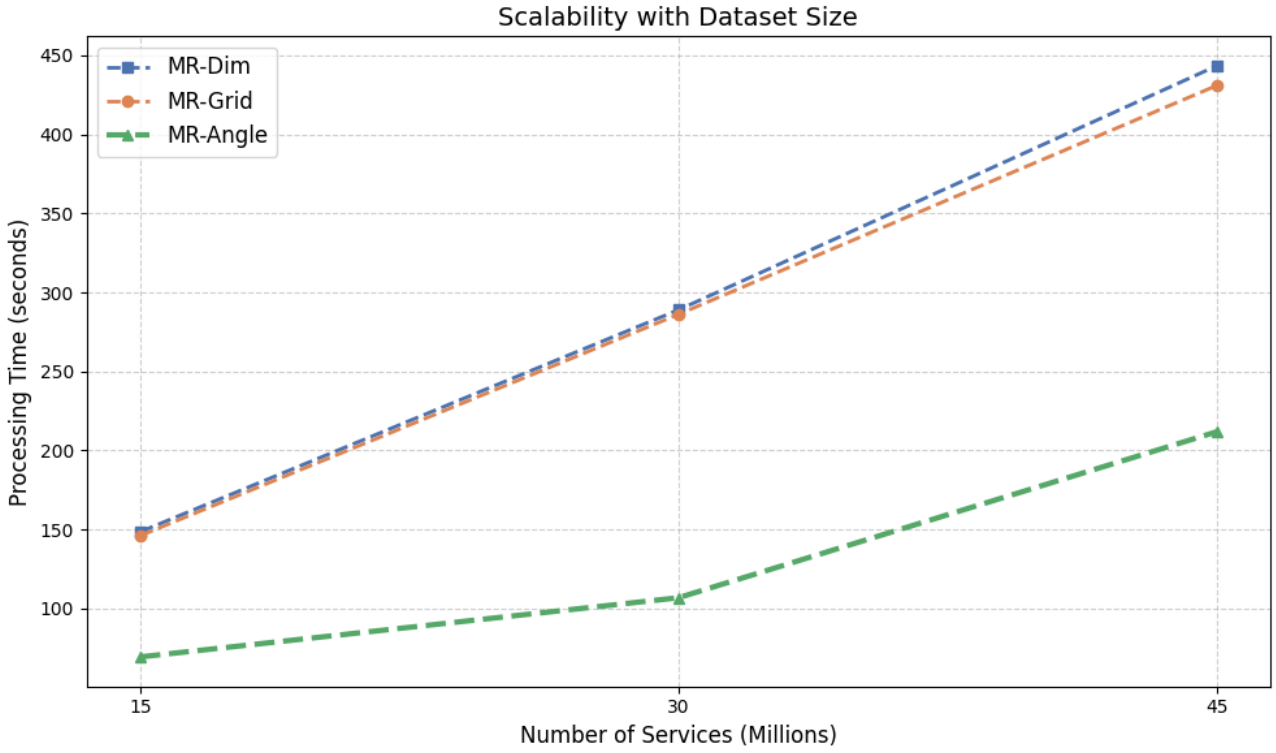
In this section, we present three experiments to evaluate and compare the three algorithms. First, we test how they perform as we increase the dataset size (15M, 30M, 45M). Second, we test different data distributions to see which algorithm works best in each situation. Finally, we verify that increasing parallelism results in better performance for our algorithm.

## Experiment 1

In this experiment, we evaluate the scalability of the three algorithms (MR-Dim, MR-Grid, and MR-Angle) by increasing the size of the dataset. The goal is to observe how the processing time behaves as the data grows. We generated three synthetic datasets containing 15 million, 30 million, and 45 million records. The data follows a Uniform distribution. All tests were executed on a Flink cluster with a parallelism of 4.

| Dataset Size | MR-Dim (s) | MR-Grid (s) | MR-Angle (s) |
|---|---|---|---|
| 15,000,000 | 148.61 | 146.17 | 69.47 |
| 30,000,000 | 288.93 | 286.11 | 106.88 |
| 45,000,000 | 430.25 | 443.79 | 211.82 |

Table 1: Processing Time comparison with increasing Dataset Cardinality (Uniform Distribution, Parallelism=4).




The result is that the MR-Grid algorithm performs slightly better than MR-Dim, yet both exhibit a similar sharp linear increase in processing time as the dataset grows. In contrast,

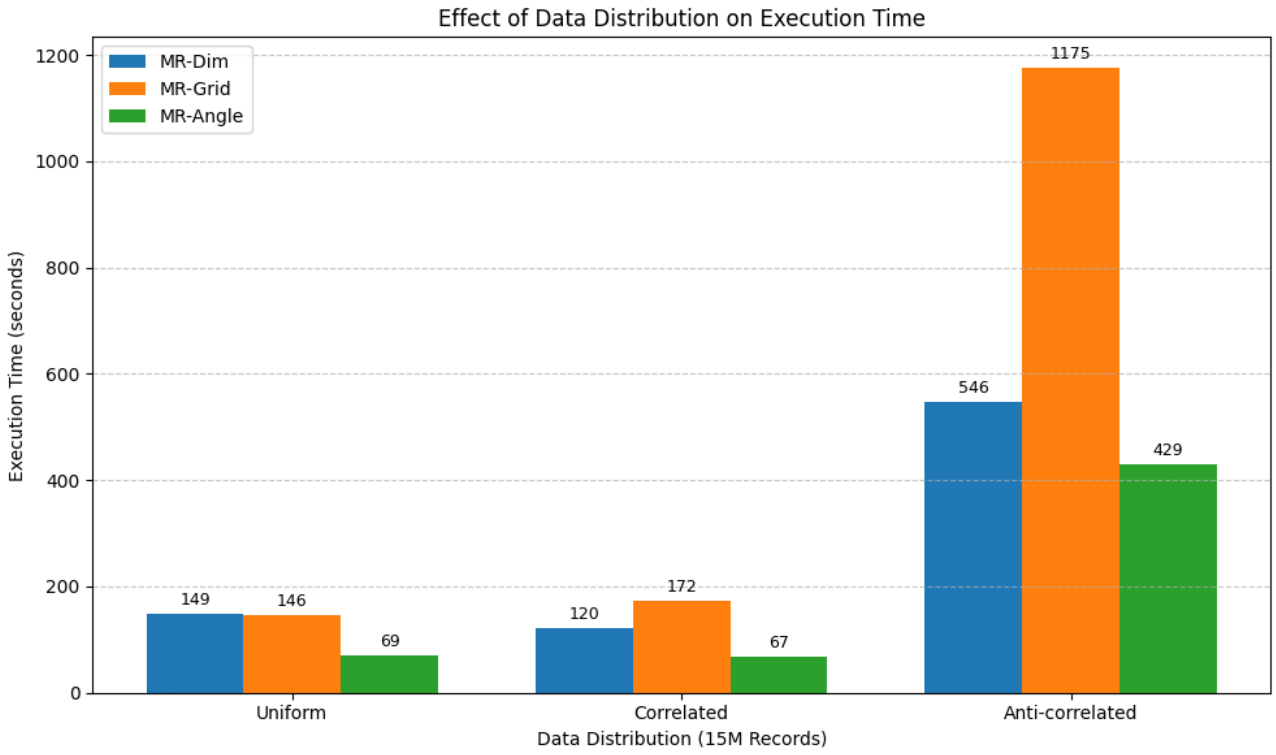
MR-Angle demonstrates superior performance and scalability, maintaining significantly lower execution times and a much more normal growth rate compared to the other two methods. In the MR-Angle and MR-Grid, we observed a notable data skew where the workload was not evenly distributed among the available resources. Specifically, due to the angular concentration of the data, certain workers processed the majority of the records while others remained idle.

## Experiment 2

In this experiment, we examine the behavior of the three algorithms against different data distributions. We fixed the size of the dataset at 15 million records. For the Uniform distribution, we leverage the measurements took in Experiment 1. For the Correlated and Anti-correlated, we generated the corresponding datasets of the same size and submitted the jobs to the Apache Flink cluster. We measured the total processing time, as we did before, for each algorithm under identical conditions.

| Distribution | MR-Dim (s) | MR-Grid (s) | MR-Angle (s) |
|---|---|---|---|
| Uniform | 148.61 | 146.17 | 69.47 |
| Correlated | 120.50 | 172.26 | 67.22 |
| Anti-correlated | 546.16 | 1175.28 | 429.47 |

Table 2: Processing Time for different Data Distributions.




For the Correlated dataset, we observed a clear difference in the performance of each algorithm. We begin with MR-Grid, which yielded the worst performance. his was expected due to the geometry of the data, where points are concentrated along the diagonal. As a result, most grid cells remain idle, and only the cells located on the diagonal receive data to process, leading to a heavy workload for specific workers and effectively losing the advantage of parallelism.

R-Dim performed better because it leverages the correlation by partitioning data using a single dimension, which is more efficient for this linear structure. Finally, the best performance was achieved by MR-Angle, which groups data based on angular partitioning, a method that proved that behaved exceptionally well for this type of data distribution.

For the anti-correlated dataset, we observe the most significant difference in execution times. Starting with MR-Grid, we see that it yielded the worst performance with a massive delay. This is due to the geometry of the data, which forms an inverse diagonal. A point with a better value in one dimension inevitably has a worse value in the other, meaning that very few grid cells can dominate others. Consequently, the algorithm fails to prune partitions and is forced to compare almost all data points, resulting in excessive computational overhead. MR-Dim performed better, as its partion approach is faster and behaves better. Finally, MR-Angle achieved the best result, demonstrating that partitioning based on angular sectors is the most effective strategy for processing these type of datasets.

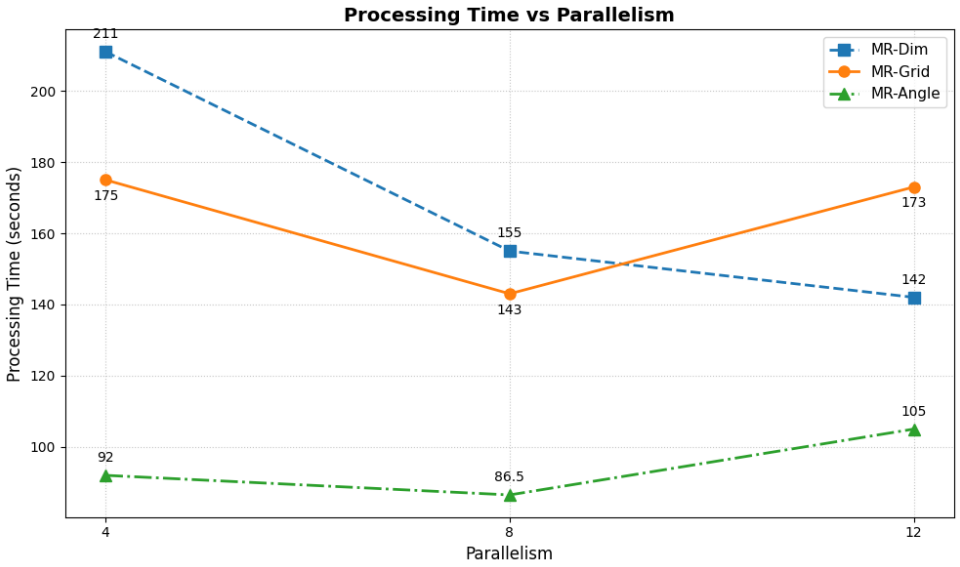
Generally, MR-Grid performed badly because of the data geometry. The algorithm's logic failed to handle the shape of the Correlated and Anti-correlated datasets, resulting in slow execution times. MR-Dim performed reasonably well fue to the simple approach of splitting data by one dimension. Finally, MR-Angle proved to be the best algorithm. Its method of grouping data by angles is very effective, making it the fastest and most stable solution for all dataset types.

## Experiment 3

In this experiment, we study the scaling behavior of the 3 skyline computation algorithms, under increasing values of parallelism. The goal is to evaluate how effectively each algorithm utilizes additional computational resources when processing a fixed-size dataset, and to identify potential scalability limits imposed by system and algorithmic overheads.
All experiments are conducted using the same dataset of 15 million points and the same Flink dataflow pipeline. The algorithms differ only in their partitioning strategy, which determines how data is distributed across workers and how local skyline computation is performed. By varying the number of parallel workers, while keeping the workload constant, we isolate the impact of parallelism on execution time and compare the scalability of the three algorithms.
Execution time was measured as mentioned above.

| **Parallelism (workers)** | **MR-Dim (s)** | **MR-Grid (s)** | **MR-Angle (s)** |
|---|---|---|---|
| 4 | 211 | 175 | 92 |
| 8 | 155 | 143 | 86.5 |
| 12 | 142 | 173 | 105 |

Table 3: Processing Time for different parallelism values.




As we can see, increasing parallelism from 4 to 8 improves performance for all algorithms, showing effective utilization of additional workers. However, pushing parallelism to 12 leads to performance degradation due to increased communication, synchronization, and scheduling overheads. This points us towards resource saturation.

We mentioned before that the experiments were conducted on a laptop with an Ultra 7 155H CPU. This CPU has 16 cores, but not homogeneous : 6 P-cores (performance), 8 E-cores (efficiency), and 2 LP-E cores (low-power efficiency), as well as 22 threads ($2 \cdot 8P$, $8E$, $2LPE$). Not all cores are equal, and scheduling multiple parallel Flink tasks means that tasks can land on any of the cores, and performance is not uniform.

With parallelism=4, all tasks are mainly scheduled on the P-cores, which guarantee good performance for computation-heavy tasks. We have minimal shuffle overhead, and MR-Angle is clearly the fastest, as expected.

Increasing parallelism to 8, we still have tasks mainly on the P-cores and some on the E-cores. This strikes a good balance between useful parallelism and acceptable coordination cost. This is clearly the optimal point, and MR-Angle's optimal performance.

Pushing parallelism to 12, we begin to see that we saturate our resources. Tasks are spread across all cores, including LP-E cores. We have increased context switching, state sync and shuffle costs. MR-Angle becomes overhead-dominated, and the execution time increases.

Overall, we see that MR-Angle consistently outperforms the other two algorithms, and optimal performance is reached at moderate parallelism values, beyond which the parallelism benefits start to diminish.

# Conclusion

Overall, we designed, implemented, and evaluated a distributed Skyline query processing pipeline using Apache Flink and Kafka. The focus was on reproducing and comparing three partitioning algorithms proposed in the paper : MR-Dim, MR-Grid, and MR-Angle, and validating their performance through 3 experiments.

Our experimental results clearly show that MR-Angle consistently outperforms both MR-Dim and MR-Grid across all scenarios. When scaling the dataset size from 15 million to 45 million points, MR-Angle had significantly lower execution times and a smoother growth rate, confirming its superior scalability. This behavior is due to the angular partitioning strategy, which groups points with similar dominance characteristics and significantly reduces the size of the local skylines, therefore lowering the cost of the global merge phase.

The impact of data distribution was also notable. While MR-Grid and MR-Dim were very sensitive to correlated and anti-correlated datasets (often suffering from severe workload imbalance and excessive dominance comparisons), MR-Angle remained robust. Its angular partitioning effectively handled these challenging geometries, leading to the best performance in all distributions tested. This confirms that MR-Angle is not only faster, but also more stable across different real-world data characteristics.

Finally, our parallelism experiments showed that increasing the number of workers/parallelism improves performance only up to a point. On the tested hardware, optimal performance was achieved at moderate parallelism levels, while higher values led to diminishing returns due to scheduling, communication, and synchronization overheads. This highlights the importance of aligning algorithmic parallelism with the underlying hardware architecture, especially on heterogeneous CPUs like ours.

Overall, this project validates the claims of the original MR-Angle approach in a practical Flink-based environment and confirms that angular partitioning is the most effective strategy for distributed skyline computation among the three examined methods.