Special Topics in Databases - COMP 622

MapReduce Skyline Query Processing with A New Angular Partitioning Approach

Implementation of Algorithms and testing

Project Documentation

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ
TECHNICAL UNIVERSITY OF CRETE

Students:

Karalis Asterinos	ID: 2020030107
Niaropetros Emmanouil	ID: 2019030168

Contents

1 Introduction

In this project we implemented and evaluated three distributed Skyline query algorithms: MR-Dim, MR-Grid, and MR-Angle. Using the Apache Flink DataStream API, we developed a program that ingests tuples via Kafka and executes global skyline computations. Finally, we benchmarked the performance and throughput against a data generator producing over 10 million records to compare the efficiency of the previous algorithms with various parameters such as parallelism, dimensionality and type of data (uniform, correlated and anti correlated).

2 Setup

2.1 System environment

This project was deployed and tested on Windows 11 environment with the following components:

- **Java SDK:** Version 11
- **WSL:** Ubuntu 24.04
- **Apache Flink:** Version 1.20.0
- **Apache Kafka:** Version 3.7.2

2.2 Maven dependencies

Below are the key maven dependencies defined in the project configuration (pom.xml):

Package ID	**Artifact ID**	**Version**	**Description**
org.apache.flink	flink-streaming-java	1.20.0	Core Streaming API
org.apache.flink	flink-clients	1.20.0	Local execution support
org.apache.flink	flink-connector-kafka	3.3.0-1.20	Kafka Source/Sink
org.apache.flink	flink-connector-base	1.20.0	Connector dependencies
org.apache.logging.log4j	log4j-slf4j-impl	2.17.1	Logging bridge
org.apache.logging.log4j	log4j-api	2.17.1	Log4j2 API
org.apache.logging.log4j	log4j-core	2.17.1	Log4j2 Implementation
com.fasterxml.jackson.core	jackson-databind	2.17.2	JSON parsing

3 Code implementation

3.1 Structure

The project consists of the following files. **FlinkSkyline.java** is the main program and will be analyzed below.

```
root-directory/
├── java/org.main/
│   ├── FlinkSkyline.java ........... The main program that processes data and computes queries
│   └── ServiceTuple.java ..................... A class representing a service with its attributes
├── python/
│   ├── kafka-producer.py ........................................ The data generator script
│   ├── query-trigger.py ................................. A script used for triggering queries
│   ├── unified-producer.py ............... A combination of the previous scripts for experiments
│   ├── graph-ingestion-parallelism.py .............. Generates performance dashboard plots
│   ├── graph-paper-figures.py ................. Replicates paper figures using aggregated results
│   ├── graph-skyline-points-2d.py ........................... Visualizes 2D Skyline points
│   └── metrics-collector.py .................................... Saves Kafka results to CSV
└── pom.xml .................................................. Project's Maven dependencies
```

3.2 Parameters

In our program (FlinkSkyline.java) you can set the following parameters:

- **`parallelism`**: Sets the Flink job parallelism.
- **`algo`**: Selects the partitioning strategy (`mr-dim`, `mr-grid`, or `mr-angle`).
- **`input-topic`** / **`query-topic`** / **`output-topic`**: Defines Kafka topic names for data ingestion, triggers, and results.
- **`domain`**: Defines the maximum value for data dimensions.
- **`dims`**: Specifies the dimensionality of the input tuples.

3.3 Initialization

First we set the **numOfParitions** to be 2 x parallelism (like in the paper) to better distribute data in the workers. We initialize two **KafkaSource** instances to ingest string records from a local broker. The **tupleSrc** is set to process all QoS data (arrive as string), while **querySrc** consumes only the query triggers that arrive after the job starts.

```java
final int numPartitions = 2 * parallelism;
KafkaSource<String> tupleSrc = KafkaSource.<String>builder()
    .setBootstrapServers("localhost:9092")
    .setTopics(inputTopic)
    .setStartingOffsets(OffsetsInitializer.earliest())
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();
```

```java
KafkaSource<String> querySrc = KafkaSource.<String>builder()
    .setBootstrapServers("localhost:9092")
    .setTopics(queryTopic)
    .setStartingOffsets(OffsetsInitializer.latest())
    .setValueOnlyDeserializer(new SimpleStringSchema())
    .build();
```

Then we initialize a data stream called **rawData** from **tupleSrc** and parse the input strings into ServiceTuple objects, and remove any invalid entries (nulls). Lastly we create a placeholder variable, the **partitioner** an object of PartitioningLogic class, that will be assigned later based on the `algo` (more on that later).

```java
DataStream<ServiceTuple> rawData = env.fromSource(tupleSrc, org.apache.flink.api.
   common.eventtime.WatermarkStrategy.noWatermarks(), "Data")
                .map(ServiceTuple::fromString)
                .filter(Objects::nonNull);

DataStream<ServiceTuple> processedData = rawData;
PartitioningLogic.SkylinePartitioner partitioner;
```

3.4 Partitioning strategy selection

A switch statement determines the partitioning logic based on the `algo` parameter. Inside the PartitioningLogic class we have created a class for every algorithm (DimPartitioner, GirdPartitioner and AnglePartitioner), each extending Flink's KeySelector and overriding their getKey() function to assign tuples to their corresponding partition according to the paper.

```java
switch (algo) {
    case "mr-dim":
        partitioner = new PartitioningLogic.DimPartitioner(numPartitions, domainMax);
        break;
    case "mr-grid":
        processedData = rawData.filter(new PartitioningLogic.GridDominanceFilter(
   domainMax, dims));
        partitioner = new PartitioningLogic.GridPartitioner(numPartitions, domainMax,
    dims);
        break;
    default:
        partitioner = new PartitioningLogic.AnglePartitioner(numPartitions, dims);
        break;
}
```

3.5 Physical partitioning

The `.keyBy(partitioner)` method is called on the data stream which acts as the shuffle barrier. Flink physically redistributes `ServiceTuple` objects across worker nodes so that all tuples with the same partition ID (0 to `numPartitions-1`) are routed to the same task slot using our custom functions.

```java
KeyedStream<ServiceTuple, Integer> keyedData = rawData.keyBy(partitioner);
```

3.6 Query broadcasting

Query triggers are read from Kafka and processed to ensure they reach every data partition. For every incoming query ID, the code generates `numPartitions` trigger events. Each event is a `Tuple3` containing a target partition ID (`f0`), the query ID (`f1`), and a timestamp (`f2`). The trigger stream is keyed by the target partition ID (`f0`). This ensures the trigger for ”Partition X” arrives at the same worker node holding the data for ”Partition X”.

```java
KeyedStream<Tuple3<Integer, String, Long>, Integer> keyedTriggers = env
    .fromSource(querySrc, org.apache.flink.api.common.eventtime.WatermarkStrategy.
   noWatermarks(), "Queries")
    .flatMap(new FlatMapFunction<String, Tuple3<Integer, String, Long>>() {
        @Override
        public void flatMap(String rawPayload, Collector<Tuple3<Integer, String, Long
   >> out) {
            long startTime = System.currentTimeMillis();
            for (int i = 0; i < numPartitions; i++) {
                out.collect(new Tuple3<>(i, rawPayload, startTime));
            }
        }
    })
    .keyBy(t -> t.f0);
```

3.7 Local skyline computation

The data stream and query stream are connected via a **CoProcessFunction** Class called **SkylineLocalProcessor** which contains the logic for processing each local skyline.

```java
DataStream<Tuple6<Integer, String, Long, Long, List<ServiceTuple>, Long>>
   localSkylines = keyedData
    .connect(keyedTriggers)
    .process(new SkylineLocalProcessor())
    .name("LocalSkylineProcessor");
```

3.8 Global aggregation

The stream of local skylines is re-keyed by **Query ID**. This routes all local results for a specific query to a single reducer instance.

```java
DataStream<String> finalResults = localSkylines
    .keyBy(t -> t.f1)
    .process(new GlobalSkylineAggregator(numPartitions))
    .name("GlobalReducer");
```

3.9 Data sink

The final results are written to the `output-topic` using a `KafkaSink`. The property `max.request.size` is increased to ≈10MB to hold large skyline result sets (this also needs to be changed internally in Kafka's configuration).

```
finalResults.sinkTo(KafkaSink.<String>builder()
    .setBootstrapServers("localhost:9092")
    .setProperty("max.request.size", "10485760")
    .setRecordSerializer(KafkaRecordSerializationSchema.builder()
        .setTopic(outputTopic)
        .setValueSerializationSchema(new SimpleStringSchema()).build())
    .build());
```

3.10 SkylineLocalProcessor

The `SkylineLocalProcessor` is the main computation unit of the program. It extends `CoProcessFunction` to handle the data stream containing `ServiceTuple` objects and the control stream containing query triggers. It implements the Block-Nested Loop (BNL) algorithm to maintain a local skyline for its specific partition and manages synchronization barriers to ensure result consistency.

The class utilizes Flink's managed state to persist the current local skyline candidate points and track synchronization metrics. A transient buffer is used to batch incoming records before processing to optimize state access.

```
public static class SkylineLocalProcessor extends CoProcessFunction<...> {
    // Persistent state for the local skyline points
    private transient ListState<ServiceTuple> localSkylineState;
    // Buffer to optimize BNL comparisons
    private transient List<ServiceTuple> inputBuffer;
    // Synchronization state: tracks progress of data stream
    private transient ValueState<Long> maxSeenIdState;
    // Queues queries arriving before data requirements are met
    private transient ListState<Tuple3<Integer, String, Long>> pendingQueriesState;
    @Override
    public void open(Configuration config) {
        localSkylineState = getRuntimeContext().getListState(new ListStateDescriptor
   <>("localSky", ServiceTuple.class));
        inputBuffer = new ArrayList<>();
        maxSeenIdState = getRuntimeContext().getState(new ValueStateDescriptor<>("
   maxId", Long.class));
        pendingQueriesState = getRuntimeContext().getListState(new
   ListStateDescriptor<>("pendingQs", TypeInformation.of(new TypeHint<Tuple3<Integer,
    String, Long>>() {})));
        // ... (Timing state initialization)
    }
}
```

The core logic for BNL resides in `processBuffer`, which is triggered when the input buffer reaches a defined size or a query is executed. It compares new candidates against the existing skyline state, removing dominated points and discarding invalid candidates.

```java
private void processBuffer() throws Exception {
    Iterable<ServiceTuple> stateIter = localSkylineState.get();
    List<ServiceTuple> currentSkyline = new ArrayList<>();
    if (stateIter != null) {
        for (ServiceTuple s : stateIter) currentSkyline.add(s);
    }
    for (ServiceTuple candidate : inputBuffer) {
        boolean isDominated = false;
        Iterator<ServiceTuple> it = currentSkyline.iterator();
        while (it.hasNext()) {
            ServiceTuple existing = it.next();
            if (existing.dominates(candidate)) {
                isDominated = true;
                break;
            }
            if (candidate.dominates(existing)) {
                it.remove();
            }
        }
        if (!isDominated) {
            currentSkyline.add(candidate);
        }
    }
    localSkylineState.update(currentSkyline);
    inputBuffer.clear();
}
```

To ensure consistency, the processor checks a ”barrier” condition. The query trigger contains a required record count and the processor executes the query only if the partition has processed enough records (‘maxSeenId-State‘). Otherwise, the query is cached in ‘pendingQueriesState‘.

```java
@Override
public void processElement2(...) throws Exception {
    String[] parts = trigger.f1.split(",");
    long requiredCount = (parts.length > 1) ? Long.parseLong(parts[1]) : 0;
    Long maxIdWrapper = maxSeenIdState.value();
    long currentMaxId = (maxIdWrapper != null) ? maxIdWrapper : -1L;
    //check data consistency
    if (currentMaxId >= requiredCount || currentMaxId == -1L) {
        processQuery(trigger, out);
    } else {
        pendingQueriesState.add(trigger);
    }
}
```

The processQuery method is invoked once the synchronization barrier is passed. It forces a processing of any records remaining in the inputBuffer to ensure the local skyline is up-to-date and it calculates the total CPU time spent on the BNL algorithm for this specific partition. Then it retrieves the calculated skyline from the state, tags each tuple with its originPartition, and emits the result as a Tuple6.

```java
private void processQuery(...) throws Exception {
    // 1. Flush remaining buffer
    long startNano = System.nanoTime();
    if (!inputBuffer.isEmpty()) {
        processBuffer();
    }

    // 2. Update CPU Metrics
    long duration = System.nanoTime() - startNano;
    Long currentCpu = accumulatedCpuNanosState.value();
    long totalCpuNanos = (currentCpu == null ? 0 : currentCpu) + duration;
    accumulatedCpuNanosState.update(totalCpuNanos);

    // 3. Prepare Output Tuple
    int partitionId = trigger.f0;
    String queryPayload = trigger.f1;
    Long triggerDispatchTime = trigger.f2;

    // Retrieve final local skyline
    List<ServiceTuple> results = new ArrayList<>();
    if (localSkylineState.get() != null) {
        for (ServiceTuple s : localSkylineState.get()) {
            // Tag tuple with origin for Optimality calculation later
            s.originPartition = partitionId;
            results.add(s);
        }
    }

    // Emit Tuple6: <PartitionID, Query, TriggerTime, StartTime, Results, CPUTime>
    out.collect(new Tuple6<>(
            partitionId,
            queryPayload,
            triggerDispatchTime,
            startTimeState.value(),
            results,
            totalCpuNanos / 1_000_000L // Convert to ms
    ));
}
```

3.11 GlobalSkylineAggregator

The GlobalSkylineAggregator functions as the final reducer. It is a KeyedProcessFunction keyed by the unique query identifier. It collects local skyline subsets from all parallel partitions, perform a final global reduction, calculate performance metrics, and emit the results as string.
The aggregator maintains a global buffer to store the consolidated skyline points and a counter to track how many partitions have completed their local processing for the current query.

```
public static class GlobalSkylineAggregator extends KeyedProcessFunction<...> {
    private final int totalPartitions;
    private transient ValueState<List<ServiceTuple>> globalBuffer;
    private transient ValueState<Integer> arrivedCount;
    //metrics
    private transient ValueState<Long> minStartTimeState;
    private transient MapState<Integer, Integer> localSkylineSizes;

    public GlobalSkylineAggregator(int totalPartitions) {
        this.totalPartitions = totalPartitions;
    }
    @Override
    public void open(Configuration config) {
        globalBuffer = getRuntimeContext().getState(new ValueStateDescriptor<>("
    gBuffer", TypeInformation.of(new TypeHint<List<ServiceTuple>>() {})));
        arrivedCount = getRuntimeContext().getState(new ValueStateDescriptor<>("cnt",
     Integer.class));
        // ... (metric state initialization)
    }
}
```

As each local skyline arrives, it is merged into the global buffer. The algorithm performs a standard skyline comparison: points in the global buffer may be dominated by the new incoming points (which come from a different partition), or the incoming points may be dominated by existing global points.

```
@Override
public void processElement(...) throws Exception {
    List<ServiceTuple> currentGlobal = globalBuffer.value();
    if (currentGlobal == null) currentGlobal = new ArrayList<>();
    //...metrics stuff...
    List<ServiceTuple> incoming = input.f4;
    //merge incoming local skyline into global skyline
    if (incoming != null && !incoming.isEmpty()) {
        for (ServiceTuple candidate : incoming) {
            boolean isDominated = false;
            Iterator<ServiceTuple> it = currentGlobal.iterator();
            while (it.hasNext()) {
                ServiceTuple existing = it.next();
                // Cross-partition dominance checks
                if (existing.dominates(candidate)) {
                    isDominated = true;
                    break;
                }
                if (candidate.dominates(existing)) {
                    it.remove();
                }
            }
            if (!isDominated) {
                currentGlobal.add(candidate);
            }
        }
    }
    globalBuffer.update(currentGlobal);
}
```

The aggregation concludes only when the ‘arrivedCount‘ matches the total number of partitions. At this stage, the system calculates the ”Optimality” metric (ratio of surviving points to local skyline sizes) and processing timestamps before clearing the state for the next query.

```
Integer count = arrivedCount.value();
if (count + 1 >= totalPartitions) {
    //..metrics calculation (Optimality, Latency)
    //formatting the output string
    StringBuilder sb = new StringBuilder();
    sb.append("{");
    sb.append("\"query_id\": \"").append(qId).append("\", ");
    sb.append("\"skyline_size\": ").append(currentGlobal.size()).append(", ");
    sb.append("\"optimality\": ").append(String.format(java.util.Locale.US, "%.4f",
    optimality));
    sb.append("}");
    out.collect(sb.toString());

    globalBuffer.clear();
    arrivedCount.clear();
    localSkylineSizes.clear();
} else {
    arrivedCount.update(count + 1);
}
```

3.12 Partitioning Algorithm Details

3.12.1 MR-Dim (DimPartitioner)

It partitions the domain of the first dimension (d_0) into equal intervals, using the formula: $P_{id} = \lfloor \frac{\text{value}[0]}{\text{maxVal}/\text{partitions}} \rfloor$. Dimensions 2 through N are ignored for partitioning purposes according to the paper.

```java
public static class DimPartitioner implements SkylinePartitioner {
    private final int partitions;
    private final double maxVal;
    public DimPartitioner(int partitions, double maxVal) {
        this.partitions = partitions;
        this.maxVal = maxVal;
    }
    @Override
    public Integer getKey(ServiceTuple t) {
        int p = (int) (t.values[0] / (maxVal / partitions));
        return Math.max(0, Math.min(p, partitions - 1));
    }
}
```

3.12.2 MR-Grid (GridPartitioner)

Treats the data space as a hyper-cube divided into 2^D hyper-octants. A bitmask integer is created. For dimension i, the i-th bit is set to 1 if value$[i] \geq$ threshold. The bitmask is mapped to a partition using modulo: GridID $\pmod{\text{partitions}}$.

```java
public static class GridPartitioner implements SkylinePartitioner {
    private final int partitions;
    private final double[] mids;
    public GridPartitioner(int partitions, double maxVal, int dims) {
        this.partitions = partitions;
        this.mids = new double[dims];
        for (int i = 0; i < dims; i++) {
            this.mids[i] = maxVal / 2.0;
        }
    }
    @Override
    public Integer getKey(ServiceTuple t) {
        int mask = 0;
        for (int i = 0; i < t.values.length; i++) {
            // If dimension i is in the upper half, set bit i to 1
            if (t.values[i] >= mids[i]) {
                mask |= (1 << i);
            }
        }
        return mask;
    }
}
```

Note: another class was implemented for filtering the dominated areas (**GridDominanceFilter**) but was not used (commented out) since the code was computationaly heavy and complex.

3.12.3 MR-Angle (AnglePartitioner)

It maps multi-dimensional angular coordinates to a single scalar value. Calculates $D - 1$ hyperspherical angles ($\phi_1 \ldots \phi_{n-1}$). Then it normalizes all angles to $[0, 1)$, computes their arithmetic mean, and scales the result by the number of partitions. Points with generally ”small” angles across dimensions group into lower partitions; points with ”large” angles group into higher partitions.

```
public static class AnglePartitioner implements SkylinePartitioner {
    private final int partitions;
    private final int dims;
    public AnglePartitioner(int partitions, int dims) {
        this.partitions = partitions;
        this.dims = dims;
    }
    @Override
    public Integer getKey(ServiceTuple t) {
        int numAngles = dims - 1;
        if (numAngles < 1) return 0;

        double[] angles = new double[numAngles];
        for (int i = 0; i < numAngles; i++) {
            double v_i = t.values[i];
            double sumSqRest = 0.0;
            for (int j = i + 1; j < dims; j++) {
                sumSqRest += t.values[j] * t.values[j];
            }
            double hyp = Math.sqrt(sumSqRest);
            angles[i] = Math.atan2(hyp, v_i);
        }
        double maxAngle = Math.PI / 2.0;
        long linearizedID = 0;
        double normalizedSum = 0.0;
        for(int k=0; k < numAngles; k++) {
            normalizedSum += (angles[k] / maxAngle);
        }
        double avgPosition = normalizedSum / numAngles;
        int p = (int) (avgPosition * partitions);
        return Math.max(0, Math.min(p, partitions - 1));
    }
}
```

4 Execution Workflow

To replicate the experiments, the components must be launched in a specific order to ensure proper synchronization between the data stream, the query triggers, and the Flink engine.

4.1 Infrastructure Setup

Before submitting the Flink job, the Kafka environment must be active with the necessary topics created.

```
#creating topics
--create --topic input-tuples --bootstrap-server localhost:9092
--create --topic queries --bootstrap-server localhost:9092
--create --topic output-skyline --bootstrap-server localhost:9092
```

4.2 Job Submission

The `FlinkSkyline` job is submitted to the cluster. The partitioning strategy and dimensionality are configured via CLI arguments.

```
#example
#these are also the default values if the parameters are not manualy set.
$ ./bin/flink run -c org.main.FlinkSkyline target/flink-skyline-1.0.jar \
    --algo mr-angle \
    --dims 2 \
    --parallelism 4 \
    --input-topic input-tuples \
    --query-topic queries \
    --output-topic output-skyline
```

4.3 Experiment orchestration

Once the job is running, the Python scripts are used to generate data and collect metrics. The `metrics_collector.py` should be started first to ensure no results are missed.

```
#o n terminal A start the metrics collector
#listens for JSON results and appends them to 'results.csv'
$ python python/metrics_collector.py results.csv

#On terminal b start the Unified Producer (Source + Control)
#streams 2D anti-correlated data and triggers a query every 1.000.000 records
$ python python/unified_producer.py \
    input-tuples \
    anti_correlated \
    2 \
    0 10000 \
    queries
```

Note: inside unified-producer.py there is the variable QUERY-THRESHOLD that makes the script trigger a query everytime the amount of records sent is equal to that. We set this to 1.000.000 but you may change this

4.4 Visualization

After the experiment concludes, the collected metrics can be visualized using the replication scripts.

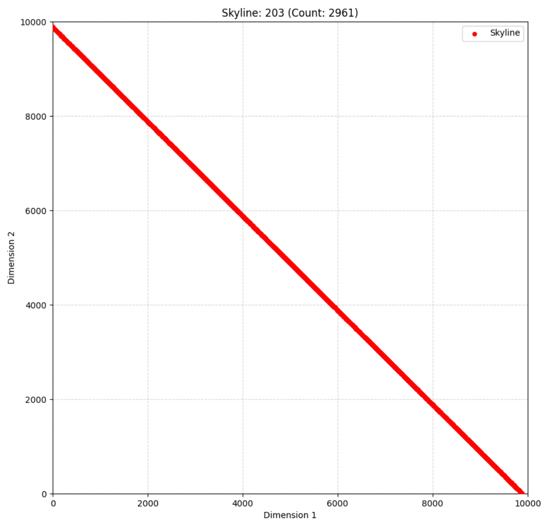
```
#generate the dashboard (Ingestion, Scalability, Optimality)
$ python python/graph_ingestion_parallelism.py MR-Angle=results.csv
```

Listing 1: Generate Performance Figures

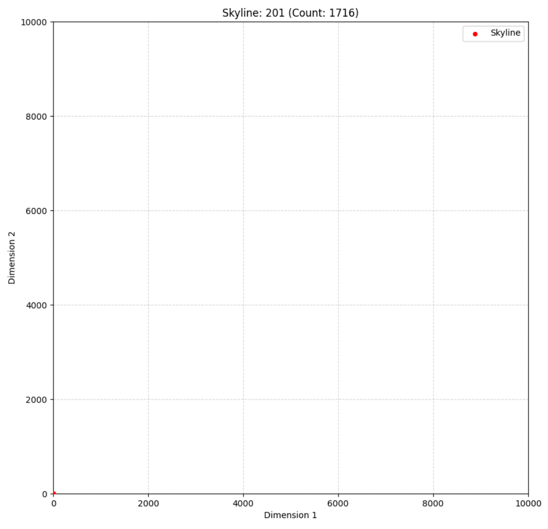
5 Experimental results

5.1 Data generator test

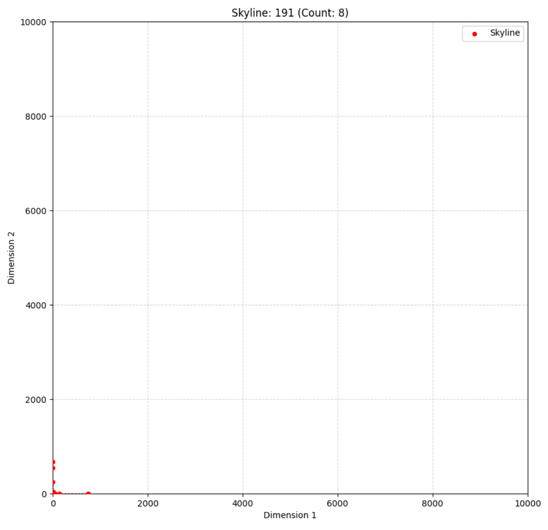
Below we demonstrate our data generator script to confirm its functionality. For this we generated points with domain values from 0 to 10000, triggering a query at 200.000 tuples.



(a) Anti-correlated Data



(b) Correlated Data



(c) Uniform Data

As expected, for plot (a) we can see the points concentrated in an anti diagonal line (2961 in total). For plot (b) we see them clustered towards the bottom left of the plot (1716 in total, all duplicates with point[0,0]). Uniform (c) has the least amount of points (8) scattered really close to the [0.0] point. This is expected when generating so many number of tuples.

5.2 Scalability Analysis across Dimensions

We first evaluate the scalability of the three algorithms (MR-Dim, MR-Grid, MR-Angle) across different dimensionalities. Figure 2 presents the processing time as a function of data cardinality for 2, 3, and 4 dimensions respectively.

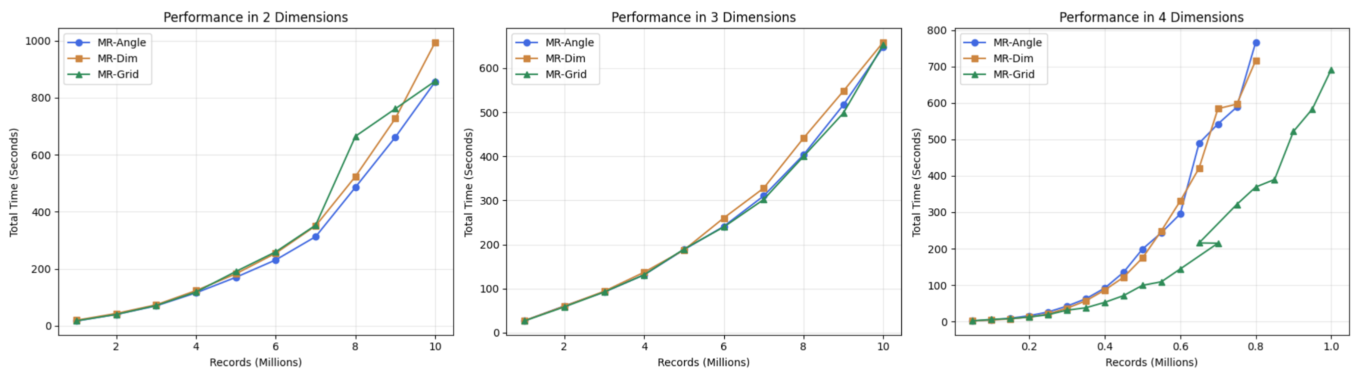


Figure 2: Scalability of distributed skyline algorithms in 2D, 3D, and 4D.

In 2D (left) and 3D (center), all algorithms exhibit linear scalability up to 10 million records, with processing times remaining under 400 seconds. The transition to 4 dimensions (right) reveals a sharp divergence: the workload becomes significantly more compute-intensive, forcing us to reduce the cardinality range to 800k-1m. Here, **MR-Grid** (Green) outperforms the others. However, this performance edge should be interpreted with caution; it may stem from stochastic variances in the synthetic anti-correlated data generation that coincidentally favored grid boundaries (clustering points in fewer partitions), or it could indicate that the current implementation of **MR-Angle** lacks specific high-dimensional optimizations—such as more efficient hyperplane projections—that would otherwise allow it to remain competitive in 4D space. In general,in lower dimensions the mr-angle outperforms the other algorithms as expected.

5.3 Impact of Dimensionality on Latency

To quantify the ”Curse of Dimensionality,” we aggregated the total processing time for a fixed cardinality (1 million records) across increasing dimensions. Figure 3 illustrates this exponential growth.

The graph demonstrates that while the time difference between 2D and 3D is manageable, the jump to 4 dimensions causes a massive spike in latency (approaching 800,000ms). This confirms that the complexity of dominance checks—combined with the inability to effectively prune points locally—dominates the execution time. The skyline size grows with dimensionality, degrading the effectiveness of the Block-Nested Loop (BNL) algorithm from average-case linear behavior to its worst-case quadratic complexity.

5.4 Partitioning Efficiency (Local Optimality)

A critical metric for distributed skyline algorithms is *Local Optimality*—the ratio of local skyline points that survive as part of the global skyline. A ratio of 1.0 implies perfect partitioning (no false positives sent to the reducer).

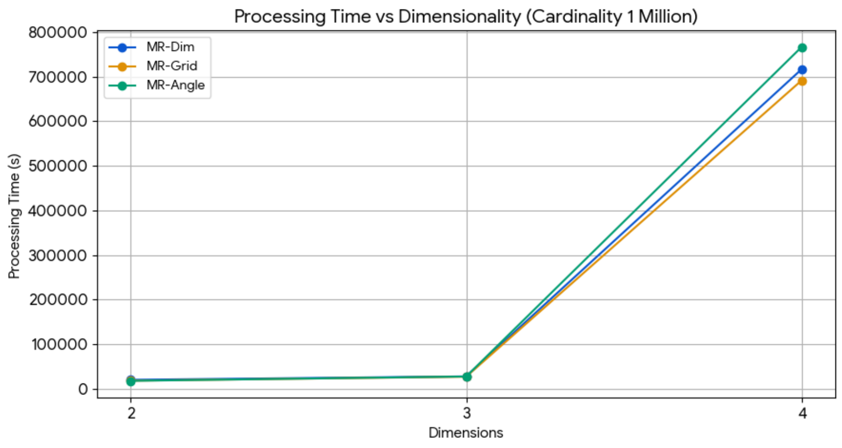


Figure 3: Impact of dimensionality on processing time (Fixed Cardinality: 1 Million).

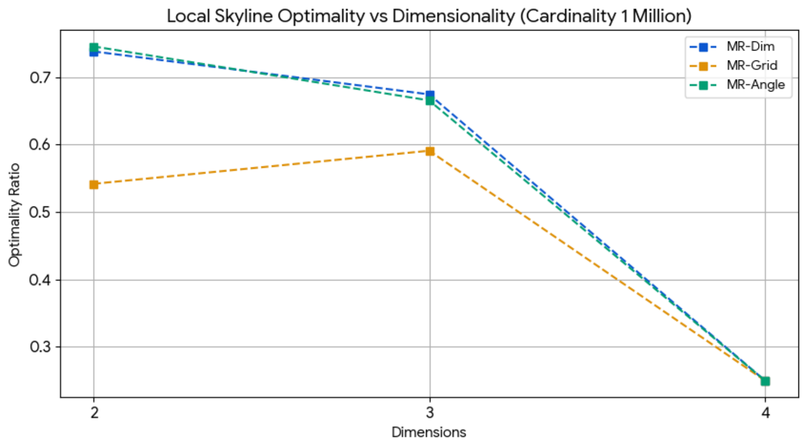


Figure 4: Degradation of Local Optimality as dimensions increase.

As shown in Figure 4, partitioning strategies that work well in low dimensions break down in high-dimensional space. In 2D, strategies like MR-Angle achieve near-perfect optimality ($\approx 0.75 - 0.99$). However, by dimension 4, all strategies converge to a baseline optimality of 0.25. This indicates that in high dimensions, points become ”non-dominated” in local subspaces more easily, forcing workers to transmit 75% ”false positive” points to the global reducer, thereby saturating the network and the final merge phase.

5.5 System Performance Dashboard & Bottleneck Analysis

Figure 5 provides a comprehensive breakdown of the system's internal metrics, allowing us to pinpoint performance bottlenecks and validation of the distributed architecture.

Ingestion vs. Processing Latency: The ”Ingestion Time” (top-left) exhibits a near-linear trajectory that remains stable regardless of data volume. This indicates that the Kafka-Flink connector acts as an efficient conduit, capable of absorbing tuples at network speed without saturation. Crucially, when we compare this to the ”Total Processing Time” (top-right), we can see that the ingestion time takes about 80% of the total processing time. This might be cause our producer is slow, or because our data are not that many and in only 2 dimensions so its easily computed.

Optimality & Time Breakdown: The ”Optimality Evolution” (bottom-left) tracks the ratio of local candidates that survive the global merge. The stability of this line (flatlining or degrading slowly) proves that our partitioning strategy (MR-Grid/MR-Angle) maintains consistent pruning power even as the stream grows. This is corroborated by the ”Time Breakdown” (bottom-right) of the final batch. The dominant bar is *Global Merge Time* (Orange), vastly outweighing the *Local CPU Time* (Blue). This signifies that the heavy lifting is correctly distributed among the parallel workers,making it last very little time, especially with more workers, and the global reducer is the one that has to do the heavy computational work.

Impact of Parallelism: Finally, the degree of parallelism (P) fundamentally dictates the shape of these graphs. Increasing P splits the input stream into smaller substreams, directly reducing the N in the local BNL's $O(N^2)$ complexity, thereby lowering the ”Local CPU Time” quadratically. We can see in the bottom left graph, that although generally it is better to increase the parallelism when you can, especially in scenarios where you don't have many data, you can make more local skyline points that aren't in the final global one and thus reducing optimality. This, doesn't necessarily mean though that the algorithm is slower, so scaling parallelism is the most effective lever for reducing the Total Processing Time shown in the top-right quadrant.

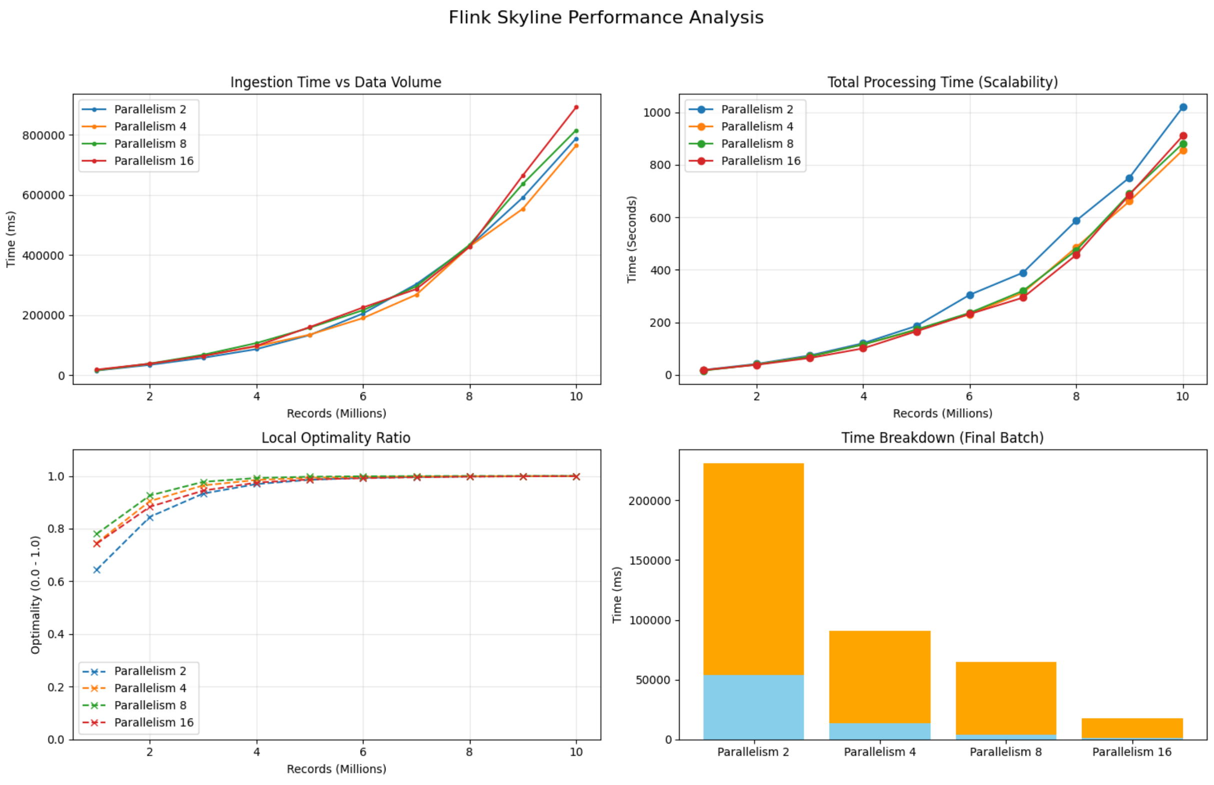


Figure 5: System Performance Dashboard: Ingestion, Optimality, and Time Breakdown.

5.6 Experimental Configuration & Reproducibility

Unless explicitly stated otherwise, all experiments presented in this section utilized **anti-correlated** data distributions. We selected this distribution as the default because it represents the theoretical worst-case scenario for Skyline queries: points clustered along the anti-diagonal exhibit minimal dominance relationships, resulting in the largest possible skyline size and maximum computational load. Evaluating our system against this distribution ensures that the reported performance metrics reflect the system's behavior under peak stress conditions.

For complete reproducibility of these results, including the exact Apache Flink cluster configuration, Kafka topic initialization, and Python environment setup, please refer to the supplementary documentation file `README_UBUNTU_SETUP.md` included with the source code.

6 Conclusion

In this project, we successfully designed, implemented, and evaluated a distributed Skyline Query processing system using Apache Flink and Apache Kafka. By porting the theoretical MapReduce-based algorithms (MR-Dim, MR-Grid, and MR-Angle) to a modern stream-processing architecture, we demonstrated the feasibility of real-time dominance analysis on large-scale datasets.

Our experimental results highlight the critical impact of dimensionality on system performance. In lower-dimensional spaces (2D and 3D), all three partitioning strategies exhibited linear scalability, with **MR-Angle** showing a slight advantage in pruning efficiency due to its angular clustering logic. The system maintained high local optimality scores (> 0.65), confirming that the partitioning logic effectively isolated skyline candidates to local workers, minimizing the load on the global reducer.

However, the transition to 4-dimensional anti-correlated data exposed the inherent ”Curse of Dimensionality.” Processing times increased exponentially, and local optimality degraded to a baseline of 0.25 across all algorithms. This collapse in pruning power indicates that in high-dimensional spaces, simple partitioning schemes become insufficient as points become increasingly sparse and non-dominated. Under these stress conditions, **MR-Grid** demonstrated unexpected robustness, outperforming other methods in execution time, something that was unexpected.

The ingestion throughput remained stable regardless of load, validating the efficiency of the Kafka-Flink connector. While increasing parallelism significantly reduced local processing times by distributing the quadratic complexity of the BNL algorithm, it also introduced a trade-off by slightly lowering optimality in smaller datasets and ”giving” more work to the global aggregator.

Ultimately, this project validates that stream-based distributed skyline computation is highly effective for low-to-medium dimensional data. For future work, incorporating more advanced indexing techniques within the local processing phase could further mitigate the performance degradation observed in high-dimensional scenarios.