Communication Papers of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 9–17
DOI: 10.15439/2025F8243
ISSN 2300-5963 ACSIS, Vol. 45

# Flexible and Scalable Results Collecting in Distributed Spatial Simulations

Piotr Aksamit
AGH University of Krakow
paksamit@student.agh.edu.pl

Mateusz Najdek
AGH University of Krakow
najdek@agh.edu.pl

Wojciech Turek
AGH University of Krakow
wojciech.turek@agh.edu.pl

***Abstract*—Among many distributed spatial simulation systems each has its own approach to the problem of results collecting and analysis. The volume of results can be huge, while not all results are finally needed. The presented solution is to provide a unified form of defining the range of data to be collected and the methods for efficiently collecting them during the simulation runtime. Simulation results can be represented as a stream of records, where every record has the same structure. This observation means, that simulation can specify one or more data schema, equivalent of the $CREATETABLE$ command in an SQL database. Then data selection and analysis comes down to writing proper $SELECT$ statements. The paper describes three main parts of the proposed, SQL-inspired results collecting method: parsing and query analysis, distributed computing and integrating all parts together. The method has been integrated with the $HiPUTS$, a distributed urban traffic simulator.**

## I. Introduction

COMPUTER simulation is a powerful method for conducting research on spatial-temporal systems. Spatial simulation can be successfully used for understanding behaviors of individual animals forming flocks [1], for predicting crowded locations during buildings evacuations [2], for understanding geological structures created ages ago by microscopic sea shellfish [3], for optimizing urban traffic [4] or public transport [5]. In general, these methods are based on mathematical models of a physical environment and physical entities existing in the environment. The simulation algorithm updates the state of the entities and the environment as the simulated time passes.

All spatial simulations generate large volumes of data during the simulation process, as the state of all the entities changes over the simulated time. Storing the complete record of these changes can significantly influence the performance od the computation. Considering an exemplary simulation of 1 million people in a city, each characterized with e.g. 30 bytes of variable state, we can expect over 1 terabyte of results after simulating only 1 hour of their life at 10 frames per second. Such volumes cannot be stored in memory and require time-consuming disk operations.

Typically, in order to draw the necessary conclusions from the simulation, not all the results are needed. The researcher may need only the state changes record of a selected subset of entities, which is an option available in popular simulators [6]. More often the desired result of the simulation computation is an aggregated value of selected state changes. This includes simple statistics and distributions of observed states over time and space, which are typically the basis for drawing valuable conclusions. Such aggregations can be computed after the simulation process finishes, however, this approach again requires all the results to be collected. It would be far more convenient to be able to compute the needed aggregates during the simulation. This approach is relatively hard to generalize, therefore it typically requires implementing specific extensions to the simulator's source code.

The problem becomes even more complex when the distribution of the simulation computation is considered. Simulating complex models of large-scale scenarios requires using many computing nodes simultaneously in order to receive the results in a reasonable time and to fit the model in the available memory. Many contemporary spatial simulation tools support distributed computing [7], [8], [9], [10]. Although distributed collection of results can be an efficient and scalable solution, calculations of aggregated values by separate computing nodes, which are responsible for different fragments of the simulated environment, is a significant challenge.

In the presented work we address the problem of simulation results collecting and aggregating during distributed computing of spatial simulations. We propose a general approach, based on the Structured Query Language, SQL, for defining the range of required data. The proposed approach does not require the user to extend the simulator's source code. In fact it does not require knowing any programming language. It is based on a standard and relatively simple query syntax, used for defining a data model and querying the model afterwards. The query syntax allows for defining simple and complex queries, including spatial and temporal limits, selecting entities of specific features or states, identifying relations between entities and computing many types of aggregates.

We present an abstract, Java-based implementation of this approach, which can be integrated with many different simulation tools. For evaluation purposes it has been integrated with an urban traffic simulator. In this important field, the simulation is a basis for the majority or research [4].

The proposed solution is also capable of working in distributed environments. While preserving the simplicity of queries, all the features have been successfully implemented for working with distributed processing of simulation models.

In the following section we present the existing approaches to the problem of results collecting in distributed spatial sim-



**Topical area:** Computer Science & Systems

ulations. The next two sections describe the proposed solution from the user perspective and its internal implementation details.

## II. Existing Approaches

The problem of results collecting, aggregating and processing is inextricably connected with all large-scale simulations, therefore it is not novel. An in-deep analysis of its elements and possible solutions can be found in [11], where authors identify four crucial challenges: selection, collection, storage and retrieval. The two considered strategies for data selection: all vs partial, have their advantages and drawbacks. Collecting all data can be justified only when rolling-back the simulation is required or detailed after-action review is needed. In other cases selection should be performed during the simulation runtime. In the context of data collection the problem of scalability has been pointed out with distributed collection as the possible solution.

An agent-based architecture for collecting simulation data has been presented in [12]. It provides services that facilitate data collection and analysis within a distributed simulation. The discrete event simulation models are considered in this case. The proposed approach is compared to the “baseline” methodology, where sub-models report the data to a central database for output analysis. Not surprisingly, a significant improvement over the centralized approach has been reported.

A few years later a similar problem was addressed using the Web Services, which apparently gained popularity in this period of time. The authors point out that "The data collection system should place minimal stress on the simulation infrastructure from both a computational load and communications overhead perspective".

A comparison between centralized, partially and fully distributed data collection methods has been presented in [13]. The evaluation is limited to the recommended, fully decentralized method, which achieves O(N) complexity, guaranteeing good scalability.

The issue of aggregating simulation results has been identified in [14], where a geo-distributed simulation has been considered. The proposed solution involved using Hadoop, a map-reduce computational framework for processing the data. The same platform has been used in [15], where data processing was executed in distributed manner without the need for centralization. Obviously not all types of aggregations can be computed this way.

Unfortunately, the conclusions, design concepts and particular solutions described above, have not been incorporated into the existing spatial simulation tools. The problem of data collecting and processing seems to receive less attention than the modeling, simulation and computations distribution. In the Flame simulator [8], a user can configure two types of outputs: "snapshots" of the complete simulation state or "agents", which results in saving selected entities only. In addition, the number of iterations between saved states can be specified. No aggregations are available. In the D-MASON framework [7], each computing node (each model partition) writes its results to its own file. One can use a statistics output streams to save self-implemented aggregates. The Pandora simulation system [10] saves all the data from of the environment and agents collected during each time step in two files. The authors provide a separate program (implemented in Python) to further analyze the data.

In the REPAST HPC [9] the most advanced data collection functionalities can be found. A user can define so called "Aggregate data collection", which use several reduction functions (sum, min, max, etc) to compute basic aggregates in the fly. The aggregates can compute the values from remote computing nodes using MPI and store them in a single file. Unfortunately, the configuration of data collection has to be defined in the source code, using the provided API.

In the field of urban traffic simulation, which is an exemplary use case considered in this work, the problem of data collection and analysis also receives relatively little attention. One of the most popular simulators, SUMO [16], offers a wide variety of so called *outputs*, which cover selected types of entities, e.g. all cars states or all lane-change events. It also offers a fixed list of aggregates, like average trip speed or route length. The user cannot select specific fragments of space or specify more complex operations on the simulation results.

Another popular tool for urban traffic simulation is MATSim [17], which is constructed over a simpler, queue-based traffic model. It allows storing the simulation results in several csv files, containing information about all simulated trips. Further analysis is to be performed outside the simulation.

SMARTS [18] is an urban traffic simulator designed to perform distributed computations. It implements several advanced features related to model partitioning and local communication between workers. It demonstrates good scalability of simulation distribution. However, the results collecting mechanisms assumes sending cars’ trajectories to a centralized server, which stores these in files. Such approach turns out to be a scalability bottleneck when hundreds of cores are used simultaneously.

The problem of results collecting and aggregating in distributed spatial simulations is clearly visible in the domain. Unfortunately, it does not receive enough attention, leaving a user with a complex problem to solve.

## III. Proposed Results Collecting Subsystem

Every simulation generates different types of data. This means that the simulation defines its own data schema, composed of the state description of all simulated entities, which changes over time. This schema can be queried using SQL-like $SELECT$ statements.

The proposed solution is abstract and can be applied in various simulations. In order to adopt it to a particular case, we need to define multiple "dictionary" tables, where data do not change during simulation, and one "events" table that keeps both references to "dictionary" tables and changeable variables. The concept will be presented using a specific case of a urban traffic simulation.

We have combined our idea with the $HiPUTS$ Simulator [19], which is a distributed urban traffic simulator[1]. It allows us to divide the environment model of simulation (road map) into separate fragments and assign the responsibility of computation over those fragments to separate processes, which can use many nodes of a computing cluster. The simulation process is divided into small time quanta called steps. In each step, the number of cars can be different in separate workers, as they can move freely from one to the other.

The environment model in the simulation is composed of two key components: lanes and junctions. It is depicted as a directed graph, where the edges represent the lanes and the nodes correspond to the junctions. Every lane starts and ends with junction. Each junction has a fixed, known position, while each lane is a straight segment linking junctions. As a result, every lane has a defined length, and junctions are aware of the circular arrangement of incoming and outgoing lanes. From the vehicle's perspective, it tracks details such as its current lane and position, speed, acceleration, and other characteristics like its length, maximum speed, and designated route, enabling it to navigate accordingly. The state of vehicle changes during the simulation due to natural movement, so parameters such as e.g. speed, acceleration and position on lane are modified and represent singular state of car in a specific time unit of the simulation. This specific car is simulated by the corresponding worker, depending on its current location.

Here we have three "dictionary" tables - $Cars$, $Junctions$ and $Lanes$ and one "events" table - $Events$ that tracks all changes. The dictionary tables contains static data that are constant during the simulation and they are not modified. The event table, in turn, contains dynamic data, such as information about changes of individual vehicles in simulation. The simulation model definition of SQL "dictionary" and "events" tables is presented in Listing 1.

To unambiguously define car in time and place we need to use triple:

$$(WorkerId, StepNumber, CardId)$$

```
CREATE TABLE Cars
(
    CarId bigint PRIMARY KEY,
    MaxSpeed numeric NOT NULL,
    Length numeric NOT NULL
);

CREATE TABLE Junctions
(
    JunctionId bigint PRIMARY KEY,
    Longitude numeric NOT NULL,
    Latitude numeric NOT NULL
);

CREATE TABLE Lanes
(
    LaneId bigint PRIMARY KEY,
    IncomingJunctionId bigint NOT NULL
        references Junctions(JunctionId),
    OutgoingJunctionId bigint NOT NULL
        references Junctions(JunctionId),
    Length numeric NOT NULL
);

CREATE TABLE Events
(
    WorkerId text NOT NULL,
    StepNumber bigint NOT NULL,
    CarId bigint NOT NULL
        references Cars(CarId),
    LaneId bigint NOT NULL
        references Lanes(LaneId),
    PositionOnLane numeric NOT NULL,
    Speed numeric NOT NULL,
    Acceleration numeric NOT NULL,
    PRIMARY KEY(WorkerId, StepNumber, CarId)
);
```

Listing 1. Equivalent simulation model definition in SQL tables.

For simplicity let's define the following view (Listing 2), that combines all listed table above 1. Data analysis is ultimately reduced to executing $SELECT$ queries on the SimulationDatas view. As previously mentioned simulations can be run on multiple computing instances, referred to as Workers, with each instance tasked with processing a specific portion of the map. The configuration file allows to define the format in which the analysis results should be exported. Currently the module supports exporting data in CSV [20] and Parquet [21], [22] formats, which has been found to be well-suited for data visualization thanks to their structure and the ability to facilitate detailed analysis across different contexts.

```
CREATE VIEW SimulationDatas AS

SELECT
    e.*, c.MaxSpeed, c.Length,
    l.Length AS LaneLength,
    ji.JunctionId AS IncomingJunctionId,
    ji.Longitude AS IncomingJunctionLongitude,
    ji.Latitude AS IncomingJunctionLatitude,
    jo.JunctionId AS OutgoingJunctionId,
    jo.Longitude AS OutgoingJunctionLongitude,
    jo.Latitude AS OutgoingJunctionLatitude
FROM Events e
JOIN Lanes l ON l.LaneId = e.LaneId
JOIN Junctions ji
    ON ji.JunctionId = l.IncomingJunctionId
JOIN Junctions jo
    ON jo.JunctionId = l.OutgoingJunctionId
JOIN Cars c ON c.CarId = e.CarId
```

Listing 2. Definition of SQL view for data in simulation.

Example. A trivial simulation has 2 steps. In first step there are 2 cars ($C1$, $C2$), in second step 3 cars ($C2$, $C3$, $C4$). Cars $C1$, $C2$ are always on worker $W1$, rest on worker $W2$. Then $Events$ table has 5 rows, two rows from first step and three rows from second step. Then $Events$ table looks as presented in Table I (for simplicity we present only columns: $WorkerId$, $StepNumber$, $CarId$):

Available syntax:

```
SELECT [expression [ [ AS ] column_name ] ] [, ...]
[FROM SimulationDatas]
[WHERE condition]
[GROUP BY grouping_element [, ...] ]
[HAVING condition]
```

[1]https://github.com/hiputs/HiPUTS

TABLE I
EXEMPLARY OVERVIEW OF FRAGMENT OF SOME COLLECTED EVENTS DATA

| WorkerId | StepNumber | CarId | ... |
|---|---|---|---|
| W1 | 1 | C1 | ... |
| W1 | 1 | C2 | ... |
| W1 | 2 | C2 | ... |
| W2 | 2 | C3 | ... |
| W2 | 2 | C4 | ... |

```
[ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST }] [, ...] ]
[LIMIT count]
[OFFSET start]

where grouping_element is one of:
    ()
    expression,
    ( expression,  [,  ...] )

where expression is one of the following:
- number, text,  NULL,
- scalar function e.g. COALESCE, CONCAT, ABS
- aggregates e.g. MIN,  MAX,  SUM,  AVG,  COUNT
- arithmetic operators: +, -, *, /, %
- text operators: ||
- conditional operators: OR, AND
- relational operators: =, <, >, <=, >=, <>,
    LIKE, REGEX, BETWEEN, IN, IS [NOT] NULL,
    IS [NOT] DISTINCT
- conditional statement:
    CASE
        WHEN expression_1 THEN result_1
        [WHEN ...]
        [ELSE else_result]
    END
- column name: (defined by simulation)
- casting: expression::type or CAST(expression,
    type)
- parentheses expression: (expression)
- signed expression: +, -, ~
```

Listing 3. Available syntax for collecting and aggregating data using this method.

## IV. SYNTAX DESCRIPTION

The syntax for our system is inspired by PostgreSQL[2] and follows a familiar structure for querying and manipulating data. In the following, we describe each clause supported by our solution and provide explanations for the available expressions and operators.

### A. SELECT Clause

The `SELECT` clause specifies the data to retrieve from the database. Each expression can optionally be renamed using the `AS` keyword:

- `expression [ [ AS ] column_name ]`: Defines what data to retrieve (e.g., column names, computed values).
- `AS column_name`: Renames the result of the expression.

[2]https://www.postgresql.org/docs/current/sql-select.html

### B. FROM Clause

The `FROM` clause specifies the source of the data. In our case, `SimulationDatas` defines the combined "dictionary" tables and "events" table.

### C. WHERE Clause

The `WHERE` clause filters the rows returned by the `SELECT` clause according to a specified condition:

- `WHERE condition`: Applies filters using relational (=, <, >, etc.) and logical operators (`AND`, `OR`, etc.).

### D. GROUP BY Clause

The `GROUP BY` clause aggregates rows with the same values in specified columns:

- `GROUP BY grouping_element`: Groups the rows by one or more columns (or expressions), often used with aggregate functions.

### E. HAVING Clause

The `HAVING` clause filters the aggregated results produced by `GROUP BY`:

- `HAVING condition`: Similar to `WHERE`, but operates on aggregated data.

### F. ORDER BY Clause

The `ORDER BY` clause specifies the sorting order of the results:

- `ORDER BY expression [ ASC | DESC ] [ NULLS { FIRST | LAST }], [, ...]`: Orders the results by an expression or multiple expressions in ascending (`ASC` - default) or descending (`DESC`) order, and determines the placement of `NULL` values.

### G. LIMIT and OFFSET Clauses

- `LIMIT count`: Limits the number of rows returned by the query.
- `OFFSET start`: Skips the first `start` rows before returning the rest.

### H. Expressions and Operators

Various expressions and operators can be used within the above clauses to perform more complex calculations and conditions:

*1) Scalar Functions:* Scalar functions perform operations on individual values. Currently implemented functions:

- `ABS(expression)`: Returns the absolute value of a numeric expression. e.g. `ABS(-5)` returns `5`.
- `CEIL(expression)`: Rounds a numeric expression up to the nearest integer. e.g. `CEIL(4.3)` returns `5`.
- `COALESCE(expression1, expression2, ...)`: Returns the first non-`NULL` expression from the list. If all expressions are `NULL`, it returns `NULL`. e.g. `COALESCE(NULL, 'default', 'value')` returns `'default'`.
- `CONCAT(expression1, expression2, ...)`: Concatenates multiple expressions into a single string.

e.g. `CONCAT('Hello', ' ', 'World')` returns `'Hello World'`.
- `CONCAT_WS(separator, expression1, expression2, ...)`: Concatenates multiple expressions into a single string skipping `NULL` values, with a specified separator between them. e.g.`CONCAT_WS('-', '2024', '10', '23')` returns `'2024-10-23'`.
- `FLOOR(expression)`: Rounds a numeric expression down to the nearest integer. e.g. `FLOOR(4.7)` returns `4`.
- `LENGTH(expression)`: Returns the number of characters in a string. e.g. `LENGTH('OpenAI')` returns `6`.
- `LOWER(expression)`: Converts all characters in a string expression to lowercase. e.g. `LOWER('Hello World')` returns `'hello world'`.
- `ROUND(expression, decimal_places)`: Rounds a numeric expression to the specified number of decimal places. e.g. `ROUND(3.14159, 2)` returns `3.14`.
- `SQRT(expression)`: Returns the square root of a numeric expression. e.g. `SQRT(16)` returns `4`.
- `SUBSTR(expression, start_position, length)`: Extracts a substring from a string expression starting at `start_position` and continuing for `length` characters. e.g. `SUBSTR('HiPUTS', 2, 3)` returns `'iPU'`.
- `UPPER(expression)`: Converts all characters in a string expression to uppercase. e.g. `UPPER('hello world')` returns `'HELLO WORLD'`.

*2) Aggregate Functions:* Aggregate functions operate on sets of rows and return a single result:
- `MIN()`, `MAX()`: Return the minimum or maximum value.
- `SUM()`: Adds numeric values.
- `AVG()`: Computes the average.
- `COUNT()`: Counts rows.

*3) Operators:* **Arithmetic Operators**:
- +, −, *, /, %: Perform mathematical operations.

**Text Operators**:
- `||`: Concatenates strings.

**Conditional Operators**:
- `AND`, `OR`: Combine conditions.

**Relational Operators**:
- =, <, >, <=, >=, <>: Compare values.

**Specialized Operators**:
- `LIKE`, `REGEX`: Perform pattern matching.
- `BETWEEN`, `IN`, `IS [NOT] NULL`: Check ranges, membership, or null values.
- `IS [NOT] DISTINCT`: Check for distinction between two values.

### I. Conditional Statements

The `CASE` expression allows conditional logic:
- `CASE WHEN expression THEN result [WHEN ...] [ELSE else_result] END`: Evaluates conditions and returns the corresponding result.

### J. Casting and Parentheses

**Casting**:
- `expression::type`: Casts an expression to a specific type.
- `CAST(expression AS type)`: Another way to perform the cast.

**Parentheses and Signed Expressions**:
- Parentheses control the order of operations within expressions.
- Signed expressions: +, −, ~: Apply positive, negative, or bitwise complement to an expression.

Below are presented some example of usage of this solution:

1) Problem 1: What was speed of car with id = 3.

```
SELECT Speed, WorkerId, StepNumber
FROM SimulationDatas
WHERE CarId = 3
```

Listing 4. Query for problem 1.

2) Problem 2: Collect velocity of car with id = 3 when it was on lanes L1, L2 and L3.

```
SELECT LaneId, Speed
FROM SimulationDatas
WHERE LaneId IN ('L1', 'L2', 'L3')
```

Listing 5. Query for problem 2.

3) Problem 3: For each car what was its average speed in every 10 steps.

```
SELECT
    CarId,
    StepNumber / 10 AS Start,
    StepNumber / 10 + 9 AS End,
    AVG(Speed)
FROM SimulationDatas
GROUP BY CarId, StepNumber / 10
```

Listing 6. Query for problem 3.

4) Problem 4: Calculate average/minimum/maximum speed separately for very slow - up to 5m/s, slow - to 10m/s, medium - to 15m/s, fast - to 20m/s, very fast - above 20m/s

```
SELECT
    CASE
        WHEN Speed < 5 THEN 'Very slow'
        WHEN Speed < 10 THEN 'Slow'
        WHEN Speed < 15 THEN 'Medium'
        WHEN Speed < 20 THEN 'Fast'
        ELSE 'Very fast'
    END,
    MIN(Speed) AS min,
    AVG(Speed) AS avg,
    MAX(Speed) AS max
FROM SimulationDatas c
GROUP BY
```

```
    CASE
        WHEN Speed < 5 THEN 'Very slow'
        WHEN Speed < 10 THEN 'Slow'
        WHEN Speed < 15 THEN 'Medium'
        WHEN Speed < 20 THEN 'Fast'
        ELSE 'Very fast'
    END
```

Listing 7. Query for problem 4.

5) Problem 5: Which cars had average speed greater than 100 in 10 subsequent steps

```
SELECT
    CarId, AVG(Speed) AS speed,
    StepNumber / 10 AS stepnumber_from,
    StepNumber / 10 + 9 AS step_number_to
FROM SimulationDatas c
GROUP BY CarId, StepNumber / 10
HAVING AVG(Speed) > 100
```

Listing 8. Query for problem 5.

### K. Optimalization

*1) Skipping simulation step:* Consider the following example.

```
SELECT StepNumber, AVG(Speed)
FROM SimulationDatas
WHERE StepNumber % 100 = 0
GROUP BY StepNumber
```

$WHERE$ clause in this example does not depend on vehicle, lane or junction parameters. When for the given simulation step $WHERE$ clause is always false, we can skip whole step without checking condition for each vehicle in this step. Before every simulation step we can execute the following procedure.

**Algorithm 1** Procedure checking whether $WHERE$ clause is always $false$

```
condition ← WHERE clause
if condition is empty then
    return false
end if

for each column ∈ SimulationDatas do
    change all occurences of column in condition with "NULL"
end loop

condition ← evaluated condition
if condition is empty then
    return false
end if

return condition
```

By conducting a preliminary check before starting the calculations for each simulation step, it can greatly enhance the program's performance. This approach allows the aggregator to determine whether collection of results for a particular step are necessary. If they are not essential—for example, when it is only interested in data calculated every 100 simulation steps—the aggregator can skip unnecessary calculations during the intervening steps. This optimization conserves computational resources, reduces execution time, and improves overall efficiency by focusing only on the simulation steps that yield relevant data.

*2) Short-circuit evaluation:* Our module stops calculating boolean expression combined by $AND$, $OR$ operators, when value can be predicted by already computed sub-expressions for given patterns:

- **OR** - when expression or part of expression consists of multiple sub-expressions joined by $OR$ we can predict result whenever any or sub-expression is $true$
- **AND** - when expression or part of expression consists of multiple sub-expressions joined by $AND$ we can predict result whenever any or sub-expression is $false$

In most programming languages, this optimization is a common practice to improve computational performance when evaluating logical expressions with multiple conditions connected by logical operators like **AND** or **OR**. By implementing short-circuit evaluation, the program determines the result of the entire expression based on the initial conditions. If the outcome is already clear after evaluating the first few conditions, it skips the unnecessary computation of the remaining ones. This not only saves processing time but also enhances the overall efficiency of the program. For instance, in an **AND** operation, if one condition evaluates to false, the entire expression is false, and there is no need to check the subsequent conditions. This technique is especially beneficial in complex logical statements where some conditions might be resource-intensive to evaluate.

## V. Implementation

Query computation consists of several steps:

1) parsing - check, whether query has valid syntax
2) analyze - check query type, it can be $InlineResult$ - which means that result of data does not depends from simulation e.g. $SELECT 1 + 2$. Otherwise we have to split the query into two parts. The first one, that will be calculated during simulation, and the second one that will combine results together.

Consider the following query:

```
SELECT CarId / 2, 3 * AVG(Speed + 3)
FORM Simulation
WHERE CarId % 10 = 0
GROUP BY CarId / 2
HAVING MIN(Speed) > 10
```

Here we can specify the following parts:

1) $AVG(Speed + 3)$ - in order to calculate average, we will calculate separately sum and count of expression $Speed + 3$

2) $GROUPBYCarId/2$ - we have to calculate each aggregate independently by expression $CarId/2$.
3) $3 * AVG(Speed + 3)$, $MIN(CASEWHENCarId > 0THENSpeed ELSE2 * SpeedEND)$ - in this query we have two aggregate functions, during simulation we will not know value of this expressions until simulation finishes and we gather data together. After compacting data we can then execute remaining scalar expressions. For expression $3 * AVG(Speed + 3)$ we will calculate $SUM(Speed+3), COUNT(Speed)+3$ during simulation. After simulation finishes we will determine value of $AVG(Speed + 3)$ and after combining data from every computational node we can calculate $3 * AVG(Speed + 3)$
4) $WHERE\ CarId\ \%\ 10\ =\ 0$ - where clause can be calculated during simulation, because it contains only scalar expressions.

In general computational nodes will keep the following tuples:
$(GroupingKey, Aggregate_1, Aggregate_2, ..., Aggregate_N)$

$$(CarId/2, AVG(Speed + 3), MIN(Speed))$$

After simulation finishes we have to take all tuples and compact them together. Our aggregator uses following aggregate functions: $SUM, MIN, MAX, COUNT, AVG$. Compacting $SUM, MIN, MAX, COUNT$ is pretty straightforward, only for $AVG$ we need to keep separately $COUNT$ and $SUM$.

### A. Memory control

We cannot predict amount of data that will be gathered, so we have to exchange data between RAM and hard drive. Json/Xml serializers wouldn't be effiecient, that's why we decided to define custom binary serialization and deserialization.

Simple types like $integer, text, bigint, boolean$ have implemented serialization/deserialization methods in every language, so let's focus on more complicated data structures.

Consider the following query, where $CarId$ is $bigint$, $Speed$ is $double$.

```
SELECT CarId, AVG(Speed)
FROM SimulationDatas
GROUP BY CarId
```

During simulation computational nodes have to keep $CarId$ and $AVG(Speed)$. For $CarId$ = 5, current value of $AVG(Speed)$ = 2.5, where $SUM(Speed)$ = 10, $COUNT(Speed)$ = 4, tuple

$$(CarId, AVG(Speed))$$

will be represented by the following sequence of bits.

| Bits | Meaning |
|---|---|
| 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000101 | 5 - bigint |
| 01000000 00100100 00000000 00000000 00000000 00000000 00000000 00000000 | 10 - double |
| 00000000 00000000 00000000 00000000 00000000 00000000 00000000 00000100 | 4 - bigint |

### B. Configuration

The analyzer can be launched and configured using a dedicated configuration file that contains its parameters definition. Example definition is presented in Listing 9.

```
analyzerConfiguration:
  bufferSize: 1000
  storageType: in-memory
  numberOfSegments: 8
  nodeSizeOfIndexTree: 16
  levelsInIndexTree: 4
  export:
    enabled: true
    format: parquet
    path: /path/to/export/directory
    configExportEnabled: false
  queries:
    - SELECT MIN(Speed) + MAX(Speed)
      FROM SimulationDatas
      WHERE CarId::bigint IN (2, 34)
      GROUP BY CarId
      HAVING COUNT(1) > 5
    - SELECT AVG(Speed) FROM SimulationDatas
```

Listing 9. Example configuration file for analyzer.

The configuration file is a crucial component that allows users to customize and control the behavior of the application. It consists of different parameters that define how the module operates, enabling users to tailor the performance and functionality to meet specific needs. These parameters cover various aspects and are defined as:

1) **bufferSize** - The size of the buffer when writing data from the Worker to a temporary file and reading data from it by the PostMaster.
2) **storageType** - An enumerated value indicating whether the data processed by the Workers is stored on the hard drive (`drive`) or in RAM (`in-memory`).
3) **numberOfSegments** - A numerical value specifying the number of independent segments into which `HashMap` structures are divided, ensuring concurrent operation of the program. Each segment has separate locks for reading and writing. Too few segments will cause bottlenecks during parallel usage, while too many will increase memory overhead.
4) **nodeSizeOfIndexTree** - The `HashMap` structure uses an `IndexTree` instead of an array to store hash keys. `IndexTree` is similar to a `SparseArray`. This means unused elements do not occupy memory space. The structure does not perform rehashing operations, so it cannot be larger than the initial value.
5) **levelsInIndexTree** - The number of levels in the `IndexTree` structure.

6) **allocateStartSize** - The initial size of the `HashMap` structure in bytes.
7) **allocateIncrement** - The number of bytes by which the size of the `HashMap` structure is increased.
8) **queries** - An array of SQL queries.
9) **export**:
   - **enabled** - A boolean value indicating whether the results should be exported to a file.
   - **format** - An enumerated value specifying the data export format (`parquet` or `csv`).
   - **path** - The relative path to the folder where the results will be exported. In the specified folder, a new folder with the export date is created, into which a file named `result<query_number>` with the extension specified in the configuration file is generated.
   - **configExportEnabled** - A boolean value indicating whether the configuration file should be exported along with the data.

When choosing parameters, it is important to asses that the maximum number of elements in the `HashMap` structure will be equivalent to:

$$\text{numberOfSegments} \times \text{nodeSizeOfIndexTree}^{\text{levelsInIndexTree}}$$

## VI. Third party libraries

Our implementation uses two mainly used libraries $JSqlParser$ and $MapDb$

### A. JSqlParser

For parsing we used publicly available library called $JSqlParser$[3]. It is an open-source library written in Java that enables analysis, manipulation, and processing SQL queries in both text and object formats.

Some of the key features of this library include:

- SQL Query Parsing - the ability to convert SQL queries from their textual form into an object-oriented data structure that is easy to work with.
- Query Manipulation - allows to modify, delete, or add parts of SQL queries using an object interface.
- SQL Query Generation - enables to create SQL queries using objects and convert them back into text form.

With $JSqlParser$, developers can effortlessly manipulate and analyze SQL queries within their applications.

### B. MapDb

For memory control We use library $MapDb$[4] that provider Maps, Sets and other collections backed by off-heap or on-disk storage. It is a hybrid between java collection framework and embedded database engine. Library is concurrent-safe and provides support for $ACID$ transactions. Due to the fact that it can use on-disk storage, library requires to provide serialization & deserialization implementation of used items. Storage type and other parameters that MapDb enables can be configured through config file.

[3]https://github.com/JSQLParser/JSqlParser
[4]https://mapdb.org/

## VII. Conclusions and Further Work

In this paper, we presented a SQL-inspired method for flexible and scalable result collection in distributed spatial simulations. By representing simulation data as streams of records conforming to defined schemas, we enable users to specify the range of data to be collected and the computations to be performed using standard SQL query syntax. This approach abstracts the complexities of data collection and aggregation in distributed environments, allowing users to focus on analysis without modifying the simulator's source code or writing additional programs.

We implemented this method within HiPUTS, a distributed urban traffic simulator, demonstrating its practicality and efficiency. Our evaluation showed that the approach introduces minimal time and memory overhead while providing significant flexibility in data collection and aggregation. The optimization techniques, such as skipping simulation steps, short-circuit evaluation and custom binary serialization & deserialization, further enhance performance by reducing unnecessary computations.

The proposed method addresses a significant gap in existing spatial simulation tools, which often lack advanced mechanisms for data selection and aggregation, especially in distributed settings. By enabling SQL-like queries, our solution simplifies the process of obtaining meaningful insights from large-scale simulations, which is crucial for researchers and practitioners working with complex models and massive datasets.

For future work, we plan to explore several directions.

A significant area we intend to explore is the utilization of simulation results during runtime by different entities within the simulation, such as vehicles in a traffic model, or by the load balancer itself. By enabling entities to access aggregated or filtered simulation data in real-time, we can enhance the fidelity and adaptability of the simulation. For example, vehicles could adjust their behaviors based on current traffic conditions derived from aggregated data, leading to more realistic modeling of traffic flow and congestion patterns. This dynamic interaction would allow for the simulation of advanced scenarios, such as adaptive cruise control or real-time route optimization.

Similarly, the load balancer could use real-time simulation metrics to dynamically adjust the distribution of computational load across workers. By monitoring the simulation results, the load balancer can identify hotspots or regions with increased computational demands and reallocate resources accordingly. This approach can improve the efficiency and scalability of the simulation by ensuring balanced workloads and minimizing processing delays.

Machine Learning Integration, where exploring the incorporation of machine learning techniques for predictive analytics within the simulation framework could enable advanced func-

tionalities like anomaly detection and trend prediction directly during simulation runtime.

In conclusion, the proposed method addresses a critical challenge in distributed spatial simulations by offering a user-friendly, efficient, and scalable solution for data collection and aggregation. By empowering users to specify precisely what data they need and how it should be processed, we facilitate more effective and focused analysis. We believe that further development along the outlined directions will enhance the system's capabilities and broaden its adoption, ultimately contributing to more advanced and insightful simulation studies across various fields.

## ACKNOWLEDGMENTS

The research presented in this paper was funded by the National Science Centre, Poland, under the grant no. 2019/35/O/ST6/01806. We gratefully acknowledge Poland's high-performance Infrastructure PLGrid ACK Cyfronet AGH for providing computer facilities and support.

## REFERENCES

[1] R. De Nicola, L. Di Stefano, O. Inverso, and S. Valiani, "Modelling flocks of birds from the bottom up," in *Leveraging Applications of Formal Methods, Verification and Validation. Adaptation and Learning*, T. Margaria and B. Steffen, Eds. Cham: Springer Nature Switzerland, 2022. ISBN 978-3-031-19759-8 pp. 82–96.

[2] M. De Iuliis, E. Battegazzorre, M. Domaneschi, G. P. Cimellaro, and A. G. Bottino, "Large scale simulation of pedestrian seismic evacuation including panic behavior," *Sustainable Cities and Society*, vol. 94, p. 104527, 2023. doi: https://doi.org/10.1016/j.scs.2023.104527. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2210670723001385

[3] P. Topa, Łukasz Faber, J. Tyszka, and M. Komosinski, "Modelling ecology and evolution of foraminifera in the agent-oriented distributed platform," *Journal of Computational Science*, vol. 18, pp. 69–84, 2017. doi: https://doi.org/10.1016/j.jocs.2016.07.009. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750316301168

[4] S. S. S. M. Qadri, M. A. Gökçe, and E. Öner, "State-of-art review of traffic signal control methods: challenges and opportunities," *European transport research review*, vol. 12, pp. 1–23, 2020.

[5] M. B. K. Kubiak and R. Długosz, "Solutions for planning smart hybrid public transportation system–poznan agglomeration as a case study of satellite towns' connections," in *Communication Papers of the 2019 Federated Conference on Computer Science and Information Systems*, 2019, p. 67.

[6] P. A. Lopez, M. Behrisch, L. Bieker-Walz, J. Erdmann, Y.-P. Flötteröd, R. Hilbrich, L. Lücken, J. Rummel, P. Wagner, and E. Wießner, "Microscopic traffic simulation using sumo," in *2018 21st international conference on intelligent transportation systems (ITSC)*. IEEE, 2018, pp. 2575–2582.

[7] G. Cordasco, V. Scarano, and C. Spagnuolo, "Distributed mason: A scalable distributed multi-agent simulation environment," *Simulation Modelling Practice and Theory*, vol. 89, pp. 15–34, 2018. doi: https://doi.org/10.1016/j.simpat.2018.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1569190X18301230

[8] M. Holcombe, S. Coakley, and R. Smallwood, "A general framework for agent-based modelling of complex systems," in *Proceedings of the 2006 European conference on complex systems*, vol. 1. European Complex Systems Society Paris, France, 2006.

[9] N. Collier, J. Ozik, and C. M. Macal, "Large-scale agent-based modeling with repast hpc: A case study in parallelizing an agent-based model," in *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers 21*. Springer, 2015, pp. 454–465.

[10] X. Rubio-Campillo, "Pandora: a versatile agent-based modelling platform for social simulation," *Proceedings of SIMUL 2014, The Sixth International Conference on Advances in System Simulation*, pp. 29–34, 2014.

[11] P. A. Wilcox, A. G. Burger, and P. Hoare, "Advanced distributed simulation: a review of developments and their implication for data collection and analysis," *Simulation Practice and Theory*, vol. 8, no. 3, pp. 201–231, 2000. doi: https://doi.org/10.1016/S0928-4869(00)00023-9. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0928486900000239

[12] S. S. Y. Xu and F. Ciarallo, "An agent-based data collection architecture for distributed simulations," *International Journal of Modelling and Simulation*, vol. 24, no. 2, pp. 55–64, 2004.

[13] E. Kaya and F. E. Sevilgen, "A fully distributed data collection method for hla based distributed simulations," in *Proceedings of the 2009 Summer Computer Simulation Conference*, 2009, pp. 337–347.

[14] K.-T. Yao, R. F. Lucas, C. E. Ward, G. Wagenbreth, and T. D. Gottschalk, "Data analysis for massively distributed simulations," in *Interservice/Industry Training, Simulation, and Education Conference (I/ITSEC)*, 2009, pp. 2–32.

[15] Y. Wu and G. Gong, "A fully distributed collection technology for mass simulation data," 06 2013. doi: 10.1109/ICCIS.2013.438 pp. 1679–1683.

[16] H. Chen, K. Yang, S. G. Rizzo, G. Vantini, P. Taylor, X. Ma, and S. Chawla, "Qarsumo: a parallel, congestion-optimized traffic simulator," in *Proceedings of the 28th International Conference on Advances in Geographic Information Systems*, 2020, pp. 578–588.

[17] A. Horni, K. Nagel, and K. W. Axhausen, "Introducing matsim," in *The multi-agent transport simulation MATSim*. Ubiquity Press, 2016, pp. 3–7.

[18] K. Ramamohanarao, H. Xie, L. Kulik, S. Karunasekera, E. Tanin, R. Zhang, and E. B. Khunayn, "SMARTS: Scalable microscopic adaptive road traffic simulator," *ACM Trans. on Intelligent Systems and Technology (TIST)*, vol. 8, no. 2, pp. 1–22, 2016.

[19] W. T. Mateusz Najdek, Natalia Brzozowska, "Hiputs: Super-scalable simulation of microscopic continuous urban traffic model," in *Proceedings of the 39th ECMS International Conference on Modelling and Simulation*, 2025, pp. 476–485.

[20] Y. Shafranovich, "Common Format and MIME Type for Comma-Separated Values (CSV) Files," RFC 4180, Oct. 2005. [Online]. Available: https://www.rfc-editor.org/info/rfc4180

[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 330–339, 2010.

[22] D. Vohra and D. Vohra, "Apache parquet," *Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools*, pp. 325–335, 2016.