1 Design with Azure

Main components:

- Azure functions
- Azure durable functions
- Azure blob storage, Azure cache for Redis

Main problems encountered:

- RAM limitations in each function
- Intermediate aggregations of data structures after each phase
- Azure functions can be written in Rust using custom handlers, but Azure durable functions cannot be written using Rust. (We are not 100% sure but, by looking on the internet we did not find any official information about writing durable functions using Rust in Azure)
- It seems to be impossible to use an orchestrator for example written in python to orchestrate functions written in Rust since we are in the same function app, thus same runtime (See link)
  Possible solutions:
  - Use only Azure functions written in Rust and orchestrate them without using durable functions but exploiting other Azure services such as message queues to re-create the fan-in fan-out pattern we want to have. (We haven't yet digged deep in this solution)
  - Write Azure functions in Rust using custom handlers and trigger them using HTTP requests. We can run a client orchestrator on a docker container hosted in the clouds that act as a stateful function and orchestrate the workload by triggering our functions with HTTP requests.
  - Switch to Java

2 Azure serverless design for parallelizing the algorithm

2.1 Main issues in parallelizing

- Passing data between functions that are in different phases of the algorithm:
  - Since data structures may be quite big, we can't simply exploit the single orchestrator to handle all the different partitions and pass them as parameters to functions of different phases. [1]
    To solve this, we thought about using **Azure cache for redis**(in-memory key-value noSQL database) as storage medium to pass data through different phases.
- Why entity-based partitions and not simple file partitioning ?
  - Assuming we have to scale to big graphs
  - Mainly because this allows workers to work on local versions of data structures. If we simply partitioned the file in N split without any semantics, then we would have had to keep all the structures global, thus use something like a queryable DB to share them (read-only) with all our functions (since a single function cannot host in memory the full Ψ_{ETD} data structure for example)
  - Still, we may have to deal with some memory problems in the functions where we merge the data structures. However, in our algorithm, we will never have to merge the full Ψ_{ETD} data structure with also the full information about Ψ_{ETPD}, which we think is the most memory critical one.

2.2 Phase 0 - How to partition the starting file

Generate N **entity based partitions**, i.e. each of our parallel worker (function) will be assigned a file containing only triplets where the **subject** is part of the set of entities assigned to the partition (e.g. Worker 1 will be assigned with set of entities {Alice, Luca, Michele}, thus the partition will contain only triplets with Alice, Luca, and Michele as **subject**). We generate these **entity based partitions** in such a way that each worker works roughly on the same amount of data.

2.3 Phase 1

Each worker will perform computations on its assigned split to generate local data structures Ψ_{ETD}, Ψ_{CEC}
After phase 1, however, we are forced to merge all the different local Ψ_{ETD} data structures, because information about the mapping **Entity→Set of types** is needed during phase 2 for the entity constraints extraction. Anyway, this should not be too much memory consuming, since in Ψ_{ETD} we still don't have any data in the Ψ_{ETPD} map in each entity (this will be populated locally during phase 2).

[1] We haven't implemented anything yet, using our common sense we simply thought that passing ≈GB data structures as function parameters was not a good idea

2.4 Phase 2

Up to now we have N different local Ψ_{CEC} (one for each worker), and Ψ_{ETD} which is the same for each worker. After this phase we will have again different local versions of Ψ_{ETD}(with the additional local informations about entity constraints (Ψ_{ETPD})), Ψ_{CEC}, and Ψ_{CTP}

- Do it as on paper for computing a local version of Ψ_{ETD}

2.5 Phase 3

Now each worker will use its local Ψ_{ETD} to compute local versions of Ψ_{SUPP} and Ψ_{PTT}. We still cannot compute Ψ_{CONF} during this phase as in the algorithm described in the paper, since we still have to merge together all the local versions of Ψ_{CEC}.
After phase 3 we now need to merge together the local versions of Ψ_{SUPP}, Ψ_{PTT}, Ψ_{CEC}, Ψ_{CTP} and use the merged Ψ_{SUPP} and Ψ_{CEC} to compute Ψ_{CONF}. These 3 data structures (SUPP, PTT, CEC) should not be memory critical anymore.

2.6 Phase 4

During phase 4 we compute Ψ_{CONF} using Ψ_{SUPP} and Ψ_{CEC}. We don't have memory issues since there is no need to keep the full Ψ_{ETD} data structure in memory, which was the most critic one.
Then we perform shapes extraction as in the QSE-exact algorithm using Ψ_{CTP}:

- Easily parallelizable by running functions over a subsets of triplets that check if the triplet is higher than ϵ or ω based if we are running on confidence or support

2.7 Graphical explanation

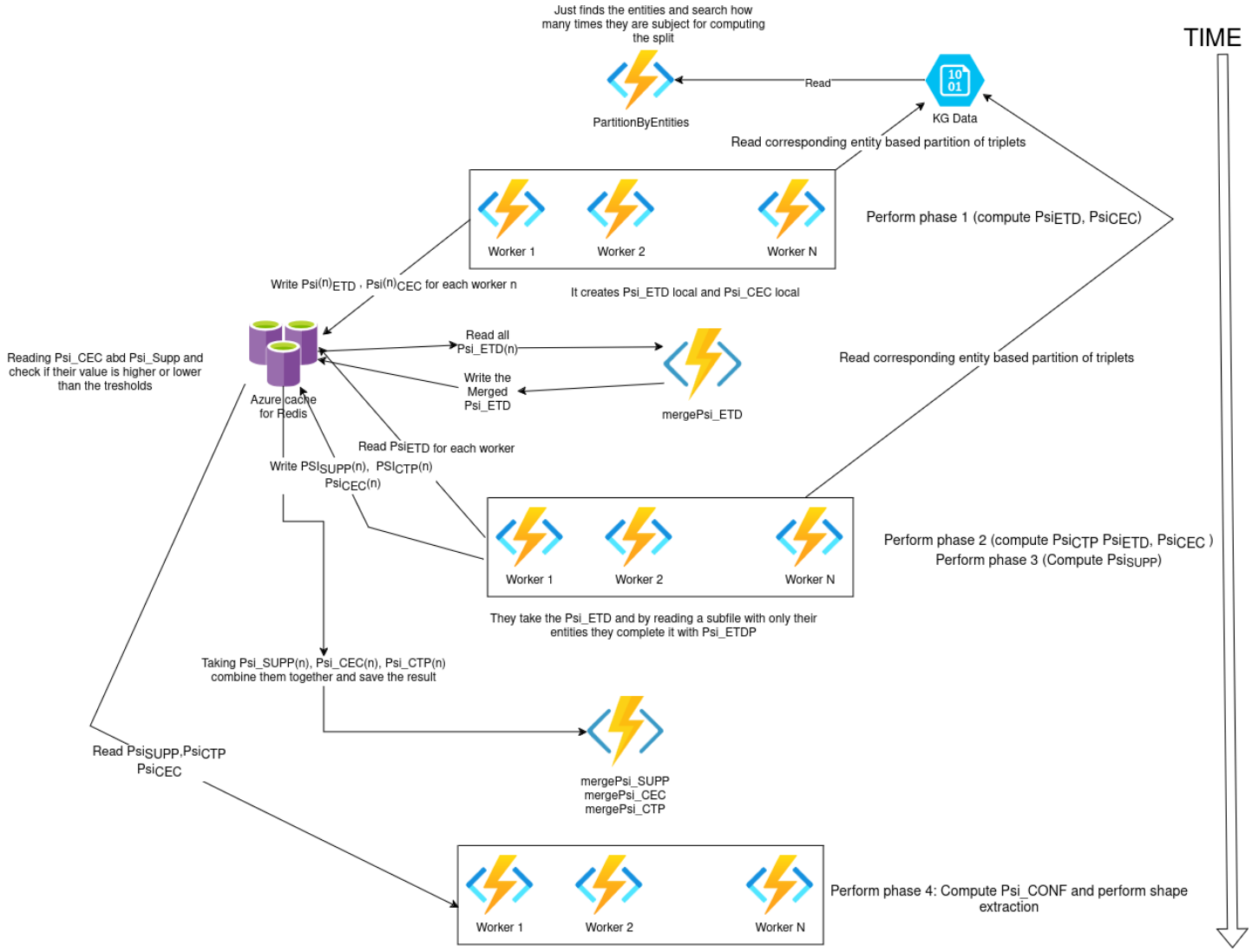

Figure 1: A sample image

3 LINKS

3.1 PAPER

- Original paper
- Summary of the algorithms

3.2 WHY RUST

https://blog.scanner.dev/serverless-speed-rust-vs-go-java-python-in-aws-lambda-functions/

3.3 AWS

- AWS RUST
- AWS itembatcher
- AWS dynamic parallelism used
- AWS parallel

3.4 AZURE

- Durable Functions
- HOW TO create durable functions Java with VS code