CAMGRAPH: Distributed Graph Processing for Camera Networks

Steffen Maass, Kirak Hong, Umakishore Ramachandran
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
steffen.maass@gatech.edu, khong9@cc.gatech.edu, rama@cc.gatech.edu

Abstract—With the proliferation of sensors of various kinds, especially cameras, large-scale situation awareness applications employing camera networks will become common place. These applications are inherently distributed, dynamic, interactive, run 24×7, and generate spatiotemporal events that need to be stored and retrieved in a timely manner to satisfy real-time constraints. To address these challenges, we present CAMGRAPH, a distributed graph processing system for storing and querying events and event relationships generate by camera networks. CAMGRAPH presents a simple, easy to use high-level API for developers of situational awareness applications to store new events and query existing events. Under the covers, CAMGRAPH does all the heavy lifting to efficiently handle the events generated by the camera network. CAMGRAPH uses a graph abstraction to store the events and their relationships. The CAMGRAPH graph processing system is a distributed architecture embodying heuristics for automatic re-partitioning of the graph to ensure load balancing, and careful placement of vertices on the nodes of the distributed system to ensure good edge locality which is important for efficient low latency query processing. We perform controlled experiments to showcase the low latency and scalability properties of CAMGRAPH.

I. INTRODUCTION

Large-scale camera networks have increased in recent times to support situation awareness applications [18], which are inherently distributed, interactive, dynamic, stream-based, computationally demanding, and needing real-time or near real-time guarantees. These applications run 24×7 and generate a large amount of data. The data is continually processed to generate events of interest. Distributed video-based surveillance is a good canonical example of this application class.

Events generated by situation awareness applications are spatiotemporal in nature. Thus every event could be represented by a 4-tuple: (x, y, t, signature), signature, wherein x, y denote the spatial coordinates, t denotes the time, and signature is the application-specific feature of interest. For example, in a video-surveillance application, the signature may be the feature-set representing the face of an individual outputted by a vision algorithm that processed a camera image. To efficiently support such applications a system infrastructure is needed to store and retrieve events of interest to the application. The infrastructure should store the events in a manner that lends itself to satisfying queries efficiently. The infrastructure should scale well for supporting several simultaneous requests for storing new events as well as entertaining queries to retrieve events of interest.

Graphs have become a convenient way of representing large data sets, and applications that use the graph abstraction span social networks, web search engines, network science, network management, and health management. Recently, Xu et al. [22] have proposed using a graph abstraction for capturing and retrieving events generated by a distributed camera network. The focus of their work is on the mathematical modeling to construct a graph consisting of object links and context links from independent observations extracted from the distributed cameras. The intellectual contribution of this work is in the formulation of the mathematical models that allows creation of the object links and context links from the independent camera observations. The graph thus constructed (in a central server) would aid in precisely answering high-level spatiotemporal queries.

CAMGRAPH, the system to be presented in this paper uses a similar graph abstraction to represent the spatiotemporal events and the relationships among the events in large-scale situation awareness applications. The focus of our work is on the system architecture for the efficient implementation of such a large-scale graph abstraction in a distributed setting.

Recognizing the limitations of programming models such as map-reduce [4] for dealing with datasets represented as graphs, several new graph processing systems have been proposed in recent times including Pregel [14], GPS [20], Giraph [6], PowerGraph [7] and X-Stream [19], employing either a vertex-centric or edge-centric computation model. Typically, the computation needed at the vertex for the graphs manipulated by these systems is quite small (e.g., counting the number of outgoing links from a webpage for pageranking algorithm). For this reason, Google’s Pregel [14] and its descendants such as GPS, use Valiant’s Bulk Synchronous Processing (BSP) [3] model to increase the coarseness of computations done at a computation node between communication steps with other nodes. Other approaches like Chronos [9] and Grace [15] focus on exploiting the temporal and spatial properties of temporal graphs and work well for temporal graph mining, e.g., understanding the development of pageranks of a web-graph (e.g., twitter) over time. Chronos and Grace use snapshots of graphs to understand their development over time and do not deal with continually evolving graphs.

Contrary to the workload assumed by these graph processing systems, the computation at the vertex of a graph used by the camera networks in a situation awareness application is a heavy-weight operation even for a single camera frame (e.g., a robust face recognition algorithm may take around 45 ms to recognize a single face). On the other hand, the cost for moving a vertex from
one computational node to another is significantly smaller by as much as a factor of 45\(^1\). Further, as mentioned earlier, situation awareness applications have real-time constraints for query processing. Moreover, the graph catering to the camera network is continually evolving and there could be hotspots depending on the activity being monitored (e.g., traffic congestion in a particular geographical area). All of these considerations call for a new approach to maintaining the continually evolving graph to ensure timeliness for query processing, load balancing to avoid hotspots, and scalability to deal with simultaneous requests for event generation and event retrieval via queries.

**CAMGRAPH** is a graph maintenance and query processing system to cater to the unique needs of situation awareness applications that deal with continuous data generated from camera networks. We present a simple API for use by domain experts to store and retrieve (via queries) events generated by these applications as a result of processing data from the camera networks. Under the cover, **CAMGRAPH** does all the heavy lifting in terms of launching the computations for identifying the relationships between the stored events, maintaining the dynamically evolving graph of events and their relationships, and performing latency sensitive query processing for retrieving events of interest. At the core of **CAMGRAPH** is a distributed architecture that employs heuristics for dynamically repartitioning the graph to increase the locality and ensure scalability.

To the best of our knowledge, **CAMGRAPH** is the first graph processing system for camera networks for use in situation awareness applications.

**B. APIs for Manipulating the Graph Abstraction**

**CAMGRAPH** provides a very simple API (a la map-reduce) for the applications. The API consists of the calls summarized in Listing 1. We give the applications a way to easily insert new vertices and query for those already stored in **CAMGRAPH**. We also expose an API for domain experts to provide their own algorithm for determining whether two vertices should share an edge. This allows developers to create their own semantics on how the graph structure shall evolve.

The bulk of the intellectual contribution of this work lies in the algorithmic details of graph maintenance, the fully distributed system architecture that ensures the scalability of the APIs exposed to the domain experts, the implementation of the system, and performance results that show the scalability of **CAMGRAPH**.

In the rest of this section, we describe the algorithmic and architectural design of **CAMGRAPH**.

Listing 1: "The API for developers to submit and query data in **CAMGRAPH**."

```java
struct Vertex {
    3DPosition position;
    Feature f;
}
option Area {
    3DBox,
    TimeCone
}
struct Query {
    Area area;
    Feature f;
}
void put(Vertex v);
list <Vertex> query(Query q);
bool haveEdge(Vertex v1, Vertex v2)=0;
```

**C. Algorithmic Design**

We first present the graph partitioning methods used in **CAMGRAPH**, followed by query optimization, and finally vertex insertion into the existing graph. In the description below, we assume that the graph is spread over a number of worker nodes in the distributed system. We use workers and partitions interchangeably to mean the same thing.

1) **Graph Partitioning:** Multiple heuristical approaches are used to partition the graph in an online-fashion, that is, concurrent with vertex insertion and query processing. Some of the approaches do not require global knowledge while some do. But, even the strategies requiring global knowledge (usually the size of each partition) are based on eventual consistency [2] as **CAMGRAPH** will repartition continuously throughout the

---

\(^1\)In our experimental setup with Microsoft Azure the vertex movement from one node to another takes roughly 1 m.s.
execution of the system; any suboptimal decisions taken due to stale knowledge will eventually get rectified. These partitioning strategies are by design simple to enable a continuous execution, similar to the strategies proposed by Stanton and Kliot [21].

**Hashing / Round-Robin.** With this baseline approach, vertices are assigned in a round-robin-fashion to worker nodes once and for all when they are inserted into the system. This strategy allows for evenly balanced partitions but doesn’t provide for edge-locality of a vertex. However, it doesn’t require any more updates when the system is running.

**Greedy.** The greedy strategy checks for all vertices affected by changes since the last partitioning. If a given vertex has more edges to vertices in another worker node $w_n$, (i.e., a different partition), it moves this vertex to $w_n$.

On one hand, this algorithm can easily be executed in parallel on every worker node and scales linearly with the number of vertices per worker in terms of execution time of the strategy. However, this heuristic does not provide for an evenly load-balanced system as there might be a large number of vertices that are being sent to one worker node in the system to improve edge-locality, potentially overloading that worker node.

At each worker node ($w_i$), the decision criterion to move a local vertex to a different worker is given by:

$$\max_{w_i} \left\{ |e = \{v_1, v_2\}| v_1 \in w_i, v_2 \in w_i \right\}$$

**Greedy-Weighted.** The greedy method is extended by balancing the raw number of edges to a specific other worker (i.e., partition) with the number of vertices already present on that worker. At each worker node ($w_i$), the new decision criterion to move a local vertex to a different worker is given by:

$$\max_{w_i} \frac{|e = \{v_1, v_2\}| v_1 \in w_i, v_2 \in w_i}{|\text{partition}(w_i)|}$$

The intuition for this strategy is to favor smaller partitions over larger partitions when they have about an equal number of local edges, thus enabling the smaller partition to grow.

**Probabilistic.** This strategy is comparable to the greedy strategy in terms of execution time. It does not just pick the worker with the highest edge-locality for a given vertex, but will pick a worker proportional to the number of edges emanating from a given worker. This strategy reduces the chance of overloading the worker nodes (thus creating hotspots) and enables a more evenly balanced partitioning among the different workers/partitions.

Thus, the probability for, e.g., a vertex $v_1$, to be assigned to the worker $w_i$ is defined as:

$$\Pr(w_i = v_1) = \frac{|e = \{v_1, v_2\}| v_1 \in w_i \lor v_2 \in w_i}{|E_{w_i}|}$$

In this formula, $E_{w_i}$ denotes the edge-set that has at least one of the vertices assigned to the worker $w_i$.

**Probabilistic-Weighted.** This strategy extends the probabilistic strategy with the same idea as the greedy-weighted strategy, namely, take the size of the partition into account in the re-partitioning decision. Thus the probability distribution for a worker node $w_i$ is defined as:

$$\Pr(w_i = v_1) = \frac{|e = \{v_1, v_2\}| v_1 \in w_i \lor v_2 \in w_i}{|\text{partition}(w_i)|}$$

2) **Query Optimization:** When processing a query, CAMGRAPH uses the very nature of the graph structure as well as the semantics of a query: If there is a match between a vertex and the query-contents given by the user, the vertex in question is being put into the result set along with its neighbors (to whom this vertex has edges) matching the spatiotemporal bounds.

3) **Worker-Selection Strategies:** We propose two different modes for inserting vertices, with the first approach being a general one, while the second heuristic is designed to exploit edge locality.

**Round-Robin.** Insert vertices in a round-robin-fashion which assures that an even balance of vertices is initially given to every worker node.

**Guided-Insert.** Maximize the edge-locality during the insertion: For each worker node, we maintain a mean $(x, y)$ position index calculated from the $(x, y)$ coordinates of all the vertices that it presently houses. In this strategy, a vertex is assigned to the worker node which mean position index most closely matches the given vertex, thereby maximizing the potential edge locality for the newly inserted vertex. This strategy does not worry about load balancing since the vertex re-partitioning strategy we discussed earlier takes care of load balancing.

**D. Architectural Design**

We now present system architecture for CAMGRAPH. With reference to Figure 1, there is one ordained node in the system architecture that hosts the message-queue, which is the entry-point to CAMGRAPH. The message-queue holds tasks, i.e., to-be-added vertices or queries. All the other nodes (denoted by DMW Figure 1) in the system architecture can take on the role of a master or a worker. The message-queue dispatches a task to a ready DMW node, i.e., a DMW which currently is not processing any other task. A node receiving a new task from the message-queue will assume the role of a master for this specific task. As a master, the DMW has to distribute queries to the appropriate worker nodes in the system or find a potentially good worker node to push a vertex to. Every DMW can act as the master for a query or a new vertex to be inserted and will only be executing one of these tasks at a time. A DMW reports as ready only after finishing a task it has been assigned by the message-queue.

In the system architecture, there exists a distributed spatial index (DSI), to gather the knowledge as to which DMW governs which area at a specific point in time and what its mean position index is. The mapping of (vertexID ⇒ workerID), i.e. which DMW a vertex is assigned to, is stored in a distributed hash table. This is necessary due to potential reassignment of vertices. The DMWs are connected to each other via a peer-to-peer network. For the purpose of broadcasting messages we use a pub-sub-network with all DMWs subscribing to it on startup.

The algorithm for inserting a vertex is given in algorithm 1.

1) **Types of Queries:** CAMGRAPH supports three different types of queries:

**BoxAreaQuery.** This query contains a field for specifying a spatiotemporal "box", boxCoordinates as well as a feature to match against. This query will carve a 3D-box out of the graph with the $x$ and $y$ axes being represented by the box given in the query and the $z$ axis being represented by the time-frame of the query. All features of the matching vertices of the result set have to be similar to the reference image.

**TimeConeQuery.** This query contains the following fields:
Algorithm 1 The algorithm running on a master node for inserting a vertex

```pseudo
procedure ON_RECEIVE(AddFeatureVertexMessage message)
  w_i ← GET_RESPONSIBLE_WORKER(message)
  id ← GEN_ID(message)
  message.vertex.id ← id
  DHT.insert([id, w_i])
  SEND(w_i, message)
  BROADCAST(workerInserted)
end procedure
procedure RESPONSIBLE_WORKER(AddFeatureVertexMessage message)
  if mode = Guided Insert then
    return nextWorker % |workers|
  else if mode = Round Robin then
    return distributedSpatialIndex.findNearestWorker(message.vertex.spatialCoordinates)
  end procedure
procedure GEN_ID(AddFeatureVertexMessage message)
  return CONCATENATE(workerID, localID++)
end procedure
```

Fig. 1: High-level overview of the distributed system architecture. DMW denotes a Distributed-Master & Worker. New vertices and queries will be put into the Message Queue for execution by a distributed master which in turn will handoff computation to a worker.

- **spatioTemporalCoord**: a 3D \((x, y, t)\) starting point
- **maxSpeed**: the maximum speed of the object of interest
- **feature**: a reference feature to match against

This query carves a 3D-cone out of the graph, allowing tracking of an object through time by only returning the results an object of interest can actually reach.

**InteractiveQuery.** This query has only one field, the **vertexID** of the vertex, along with its neighbors. The semantics of this query are to retrieve the vertex **vertexID**, with its feature, as well as all of the vertices connected to its outgoing edges.

This query uses the graph-structure in itself for fetching results: It allows for exploring the similarity graph by retrieving a vertex and its neighbors. The proposed use of this query is to allow an operator to explore the graph, step by step, e.g., to track a target throughout space and time.

2) **Sequence of actions while inserting a vertex**: The general course of actions is depicted in Figure 2. The camera network delivers the vertex \(v\) to the message-queue which further routes the message to a node \((DMW_i)\) functioning as the master for inserting \(v\). After querying the DSI to determine the best match, \(DMW_i\) will pass \(v\) to a specific worker \((DMW_k)\), which will query its neighbors for similar vertices, add its local and remote edges, insert the mapping of \(v \Rightarrow W_k\) into the DHT. \(DMW_k\) also updates the DSI with the new set of bounds and a new mean position index (if applicable).

3) **Executing a query**: Upon reception of a new query at a worker, it will look up the vertices (in its local spatial index) that match the spatiotemporal constraints set forth by the query and will then perform the similar-function. If there is a match, the matching vertex is put into the result set.

**Optimization.** As introduced before, we can use the edges of the graph to speed up query executions by circumventing the costly image-analysis for all neighbors of an already matched vertex.

4) **Consistency**: We will briefly discuss consistency for the various operations in CAMGRAPH. The graph is eventually consistent, as the graph will for some time during the insertion not contain all edges that are actually in the graph: When inserting a remote edge, this edge will at first only be inserted on the local worker which executes the query associated with the insertion (see Figure 2 for details) before sending the response containing the remote-vertex of the remote edge back to the host currently responsible for the vertex. But, this kind of inconsistency is not a problem, as remote edges will only be used for re-balancing purposes or for fulfilling InteractiveQueries. The key property of the operations carried out by CAMGRAPH is the fact that the activities at each worker in response to either vertex insertion, query, or re-partitioning is entirely based on local knowledge and does not depend on other worker nodes. Consequently, while the returned result for a query may be inexact (e.g., a new vertex insertion is not complete) but always correct with respect to some temporal snapshot of the system.

Algorithm 2 The algorithm running on a worker node for inserting a vertex

```pseudo
procedure ON_RECEIVE(AddFeatureVertexMessage message)
  v ← addToLocal(message.vertex)
  GENERATE_LOCAL_EDGES(v)
  FIND_REMOTE_EDGES(v)
  RUN_PARTITIONING(v)
end procedure
procedure RUN_PARTITIONING(Vertex v)
  message ← PartitioningMessage(v.neighbors)
  BROADCAST(message)
end procedure
procedure ON_RECEIVE(PartitioningMessage message)
  vertices ← selectLocalVertices(message.vertices)
  for all vertices as v do
    RUN_PARTITIONING_STRATEGY(v)
  end procedure
```

Camera Network Queue W_i DSI W_k DHT

Fig. 2: A detailed overview of messages being sent during an insert-operation of a vertex. The nodes in the network participating in this examples are depicted on the top, where \(W_i\) and \(W_k\) are two workers, DSI is a distributed spatial index and DHT is a distributed hash table.
Most importantly, by design there is no possibility of oscillation, instability, or deadlock in the operation of CAMGRAPH.

There are two more components in the system architecture which we should describe:

**Vertex to Worker-Mapping.** : For this function, the system can allow eventual consistency. The only part where this functionality is critical is an InteractiveQuery (see §II-D1 for details). But, if the vertex in question is not found in the local data-store, it will simply be forwarded to the worker actually in charge of the vertex to fulfill the query, allowing the system to rely on eventual consistency.

**Distributed Spatial Index.** : For this component, the system can also allow eventual consistency, at least for the location of the mean position index for every worker: Even though this information might change frequently, it will not change by a large margin in most cases. Even if it changes by a large amount, it will still not be an issue as this information is mainly being used for the insertion-mode to find a suitable worker that is close to the vertex which is being inserted. Even if incorrect decision is taken due to stale data, it will get rectified during the rebalancing/repartitioning phase.

### III. Implementation

CAMGRAPH has been implemented as a prototype in C++. Coupling of the various modules is being done via message-passing using Google Protobuf [8] as the serialization-solution as well as @MQ[23] as the networking-layer. The broadcast-layer that was introduced in §II is being implemented by the means of pub-sub-network using @MQ. Image processing and analysis is done using the OpenCV-library and the Fisherface-algorithm is being used for face-recognition [1]. The database, described in §II which is being used to store location and time for each vertex for easier query-retrieval is a MongoDB. To enable fast querying, a compound index consisting of the location and the time for each vertex is being used inside the respective MongoDB-instances.

All communication happening in the system has been implemented in a non-blocking fashion: messages that have been received in a node will be put in a processing-queue, allowing the receiving function to immediately acknowledge the receipt of the message without having to wait for the message to be processed. The processing of these messages is being done in an event-based manner, ensuring efficient usage of the given system resources. This is implemented by waiting on a socket to receive messages, thus avoiding active polling.

The distributed spatial index has been implemented using a sharded collection in MongoDB, distributing the dataset among a set of client-instances.

For storing the association of which vertex resides on which worker, a distributed hash table is being employed. The implementation chosen to do this is a zero-hop hash-table (ZHT) [13].

### IV. Evaluation

#### A. Overview

We will start with an overview of what to expect in this evaluation and how to put the results into context. Even though the sizes of the graphs seem to be somewhat limited (up to $\approx 3000$ vertices), the number of vertices in the graph is actually not the main cost-driver: To support the queries a naive algorithm would compare every vertex to a query-reference. This will take about $50 \text{ ms}$ to $100 \text{ ms}$ per vertex, a naive implementation on a single core would take about $100 \text{ s}$ to $200 \text{ s}$ to fulfill one query (given a graph size of 2000 vertices). Such a system can obviously not support real-time-queries.

#### B. Setup

We performed the evaluation on Microsoft Azure, utilizing eight D14-instances, with 16 CPU-cores (Intel Xeon E5-2660, 2.20 GHz) each, and 112GB RAM each. Every participating Azure-instance ran a local instance of MongoDB, running a sharded collection of the distributed spatial index and a number of local collections for the workers of that specific instance. Also, every instance ran a distributed server for the distributed hash table. Per Azure-instance, we started up to 16 workers to ensure a 1-to-1 setting between workers and CPU-cores.

#### C. Methodology

We used sample components inserting vertices and posting queries to the system, both components used the “Labeled Faces in the Wild” (LFW) [11] database of faces. To enable a realistic workload this component also generates traces of walking-movements of persons on a given 2D-topology which were used to tag the vertices with the 2D-location. The movement of up to 15 persons was generated at a time, on a $100m \times 100m$-map. Every person took 1 to 3 steps in the same direction until a new direction is chosen and generated a new vertex at every step. A new location for a given person was generated on average every $750 \text{ ms}$, while the walking speed of a person was randomly varied from $1.2 \text{ m/s}$ to $2.0 \text{ m/s}$ [17]. 200 vertices were inserted into the graph for every person, simulating a scenario of about 2.5 minutes for every person.

Every scenario was run with two different random-seeds, simulating different workloads. Also, and each scenario will be run two times, we will take the better of the two runs and the average of both scenarios in the later sections.

#### D. Metrics & Variables

The metrics and control variables used in the evaluation are listed in Table I.

**Metrics.** Metrics include the time to insert one vertex into CAMGRAPH, as well as the time to execute a query and return the result. The latter is being determined by submitting a set of InteractiveQueries to CAMGRAPH, measuring the latency. We also measured the maximum rate of vertex-insertions and queries which can be handled by CAMGRAPH to determine the scaling property of CAMGRAPH. Another metric is the overall number of remote edges (edges between two vertices on different workers) in the system and the number of vertices assigned to each worker.

**Variables.** Variables include the number of workers, varying between 2 and 64, and the chosen partitioning algorithm. Another variable is the number of persons being tracked in the system, which also determines the number of vertices per seconds being inserted into the system as the insertion rate can be seen as a constant per person. The insertion rate will be chosen uniformly at random with a mean of $500 \text{ ms}$ and a maximum of $1000 \text{ ms}$ for every insertion.


### TABLE I: The various metrics and control variables being used in the evaluation.

<table>
<thead>
<tr>
<th>Control Variables</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Time to Insert One Vertex</td>
</tr>
<tr>
<td></td>
<td>Time to Execute Query</td>
</tr>
<tr>
<td>Workers</td>
<td>Maximum Insertions per Second</td>
</tr>
<tr>
<td>Hash</td>
<td>Number of Remote Edges</td>
</tr>
<tr>
<td>Greedy</td>
<td>Number of Vertices per Node</td>
</tr>
<tr>
<td>Weighted Greedy</td>
<td></td>
</tr>
<tr>
<td>Probabilistic</td>
<td></td>
</tr>
<tr>
<td>Weighted Probabilistic</td>
<td></td>
</tr>
</tbody>
</table>


**TABLE II: Timing analysis of one vertex insertion with 2000 vertices already inserted, with 64 workers and the weighted greedy partitioning strategy.** “Other” includes serialization, sending, and receiving of messages as well as network delay.

![Fig. 3](image)

**Fig. 3:** The insertion time in milliseconds for a new vertex, using the weighted greedy partitioning strategy, varying the number of workers from 2 to 64.

![Fig. 4](image)

**Fig. 4:** Maximum insertion time/rate which results in a non-overloaded system, given in milliseconds, between two vertices, using 2 to 64 workers and the weighted greedy partitioning strategy.


### E. Questions

We will focus on answering the following set of questions throughout the rest of the evaluation:

- Does the distributed architecture scale with size and number of workers?
- Do our partitioning schemes, especially the weighted greedy scheme, perform well, both in terms of insertion time as well as edge locality and load-balancing.
- Does the graph-optimization yield a shorter query-response time?
- Does the guided-insert-strategy improve the scalability characteristics of the system?

### F. Results

**Vertex Insertion & Scale.** We will start with the insertion-rate as well as response time for new vertices in CAMGRAPH, focusing on answering the question for the effectiveness and efficiency of our distributed architecture.

The setting for this experiment were: 10 persons being tracked (results in an expected mean time between two vertices of 75 ms and 2000 vertices), using the weighted greedy partitioning method with the round-robin insertion mode. The results show the time to insert one vertex, averaged across 50 vertices respectively, and can be seen in Figure 3 (a), from 50 to 1000 vertices, and in Figure 3 (b) from 1000 to 2000 vertices. These results show that initially inserting a vertex takes between 150 ms to 250 ms, depending on the number of workers participating in the system. It is also visible that increasing the number of workers from 2 to 16 results in an improvement on the insertion time with little benefit when increasing the number of workers beyond that. This is due to a small number of workers resulting in every worker covering a larger area, as compared to a setting with more workers, resulting in most (if not all) of the workers participating in every query, hindering parallel execution. Also, every worker is responsible for more vertices in general. The parallelism in settings with more workers, due to distinct worker-sets being accessed, results in faster response times and most notably an increase in maximum insertions per second. In this case, the insertion time is only bound by network delay and actual processing times as summarized in Table II for the case of 64 workers and 2000 already inserted vertices. Note that the processing time for a single face-recognition is bound by ≈ 45 ms in our environment.

But, taking Figure 3 (b) into account, we can also recognize the scalability-bounds with very few workers. For example, in a setting with two workers, inserting one vertex has to be finished within 150 ms when inserting a new vertex every 75 ms to avoid queueing. In general, the processing time should be bound by \( \#\text{workers} \times \text{insertion\_rate} \). The effect of missing deadlines can be seen in Figure 3 (b), as the settings with 2, 4, and 8 workers eventually start experiencing large insertion times.

To determine the maximum possible rate of vertex-insertions we will run CAMGRAPH varying the number of workers from 2 to 64, while the insertion rate, is being varied until CAMGRAPH doesn’t incur end up overloaded, i.e. response to insertion smaller than 500 ms, with the final amount of vertices present in the system (e.g. for 10 persons: 2000 vertices). Figure 4 (a) and Figure 4 (b) show the results of this experiment. It is interesting to note that the maximum insertion time is actually larger than the theoretical minimum of \( \#\text{workers} \times \text{insertion\_rate} \) (e.g., 150 ms for 2 workers). This is due to side-effects of other actions, e.g. moving vertices to other workers or executing queries, happening concurrently to the actual insertions, resulting in slightly larger maximum insertion times. Figure 4 also shows that the maximum insertion rate doesn’t scale perfectly with the number of workers which is due to multiple reasons: Running the partitioning incurs overhead when vertices have to be moved to a new worker. Also, the distributed indices incur some more latency.

We can conclude that the distributed design scales well with the number of workers and the size of the problem, while spending most of its execution time inside the image processing (which is not the scope of this paper) and thus yields an efficient and effective design.

**Partitioning Strategies.** We will now focus on answering,
whether the partitioning schemes help to render CAMGRAPH more scalable and faster. The strategies involved in here are: hash (no partitioning), greedy, weighted greedy, probabilistic and weighted probabilistic.

The setting for this evaluation are 64 workers, tracking 10 persons, resulting in an expected time between vertices is 75 ms and 2000 total vertices. One experiment is depicted in Figure 8 (b) and displays the ratio of local and remote edges for the various strategies, demonstrating that the weighted greedy strategy exhibits the fewest remote edges. Note that the number of total edges is not the same for every experiment as both probabilistic partitioning strategies show overload of the system (see Figure 6 (a) for reference). This results in a different graph-structure as CAMGRAPH will only focus on spatiotemporally “close” vertices which due to the overload will result in less potential edges. Figure 8 (b) also shows that the weighted greedy strategy results in the fewest remote edges and shows a considerable improvement compared to the general greedy strategy. This is a result of taking the partition size of a possible new worker into account as it allows for better judging the possible clustering in that worker for a specific vertex. The same holds true for the weighted probabilistic method, but, the improvements are small over the general probabilistic strategy. The reason for this is the already apparent improvement of the probabilistic strategy compared to the greedy strategy, with the probabilistic strategy only having about a third of the remote edges as the greedy strategy.

We will also take a look at the number of vertices per worker to judge the load-balancing induced by the strategies, which is depicted in Table III. This shows that both the probabilistic strategies perform poorly in terms of load-balancing, with the probabilistic strategy performing worst by assigning almost 25% of all vertices to a single worker, while the weighted probabilistic strategy shows slight improvements. However, even the greedy strategies don’t perform well, as both of these strategies show considerable worse load-balancing compared to the hash strategy which exhibits a perfect load-balancing. To add to these results, Figure 7 (a) shows the number of vertices per worker with 2 to 64 workers after inserting 2000 vertices. This result shows that the good results for the weighted greedy strategy in terms of load-balancing work for a range of different numbers of workers. It is noticeable that the largest variance is visible at 8 and 16 workers. The explanation for this result can be found in the design of the scenario: Once again, 10 persons are being tracked, which naturally results in about 10 to 15 clusters of well connected vertices. With 8 or 16 workers, it is possible to put such clusters completely onto one worker and due to the low number of workers and thus the increased probability of a matching vertex to be put on the same machine, larger clusters may be completely put on a single worker, resulting in a skewed statistic in terms of load-balancing while also minimizing the number of remote edges.

The insertion times with all partitioning strategies are presented in Figure 6 (a). This shows that both probabilistic partitioning strategies do not perform well. The reason for this is excessive movements of vertices between workers with these strategies. Figure 8 (a) shows the number of movements for all partitioning strategies in the system and shows the excessive movements that the probabilistic strategies induce, especially the weighted probabilistic strategy. Moving a vertex requires updating the DHT, shipping the vertex across the network as well as processing time on the receiving worker. This encourages the exploration of the greedy strategies which is depicted in Figure 6 (b). It shows the improvements of the weighted greedy strategy over the general greedy strategy. Another interesting observation is the similar performance of the hash compared to the weighted greedy strategy, demonstrating the effectiveness of the latter.

For the partitioning schemes, we conclude that the weighted greedy- strategy performs as well as the hash-strategy in terms of insertion times while providing far better edge-locality (99% reduction in remote edges, 94% compared to greedy-scheme).

**Graph Optimization.**

As the graph-optimization is only active when executing queries the setting for this evaluation is a query-based workload consisting of TimeConeQueries for randomly selected vertices.

A single data-point aggregates five query-response times. We will use 16 workers, tracking 12 persons using the weighted greedy partitioning strategy.

The average query-response-times up to 1500 vertices are shown in Figure 5 (a). The results show a response time between 50 ms and 200 ms with a slight improvement visible for the graph-optimization. It is noticeable that the plain setting may result in faster response times as well (e.g. at 700 vertices) due to spurious effects (inserts, vertex-movements) throughout the system. But, overall, the graph optimization results in faster response times and especially results in a more consistent response time due to a smaller computational overhead.

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Min Vertices</th>
<th>Max Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hash</td>
<td>31</td>
<td>122</td>
</tr>
<tr>
<td>Weighted Greedy</td>
<td>3</td>
<td>170</td>
</tr>
<tr>
<td>Greedy</td>
<td>7</td>
<td>192</td>
</tr>
<tr>
<td>Weighted Probabilistic</td>
<td>8</td>
<td>233</td>
</tr>
<tr>
<td>Probabilistic</td>
<td>2</td>
<td>493</td>
</tr>
</tbody>
</table>

**TABLE III:** The minimal and maximal number of vertices with 64 workers, after inserting 2000 vertices into the system using a varying vertex partitioning strategy.
This can be explained with the less fortunate initial vertex-distribution of vertices with 2
(a)
(b)
Guided Insert. To evaluate this strategy, we will evaluate the insertion times as well as the query-times, the number of moved vertices and the edge-distributions with and without the guided insert-strategy.

The results of comparing the two insertion strategies can be seen in Figure 7 (b), using 16 workers and the weighted greedy partitioning strategy while inserting 2000 vertices. It shows two interesting trends: On one hand, it is immediately apparent that the round-robin setup is suffering from overload. This can be explained with the less fortunate initial vertex-placing selections that the round-robin strategy will incur as compared to the guided-insert strategy. This results in increased vertex-movements, resulting in less potential throughput for CAMGRAPH due to added overhead. It is also noticeable that the insertion times for both strategies in the non-overloaded case are comparable, with the guided-insert-strategy being a little slower as the round-robin strategy. This can be explained due to less-than-perfect initial load-balancing when using the guided-insert-strategy as compared to the round-robin scheme.

Comparing the vertex-movements shows the increased vertex-movements which have been identified as the main scalability-problem for the round-robin scheme. For the same scenario as before, the round-robin-strategy incurs 2982 movements while the guided-insert-strategy only incurs 2135 movements, almost a 30% reduction.

But, not all is good for the guided-insert-strategy: As discussed before, this strategy sacrifices on load-balancing and might thus result in a worse vertex-balancing among the workers. Albeit this statement is true, the situation is still acceptable as can be seen in Table IV, showing the minimal and maximal number of vertices on each worker for the same settings as before after inserting 2000 vertices. The results show that the guided-insert-strategy suffers from a slightly worse load-balancing compared to the round-robin scheme. But, this is expected from our earlier results, we thus conclude that the guided-insert is effective and results in superior performance-capabilities.

Lastly, we show that the guided-insert strategy is helpful in reducing the number of remote edges due to more informed initial vertex-placements. This is shown in Table V, with 16 workers, the weighted greedy partitioning scheme, and 2000 vertices inserted into CAMGRAPH. These results validate a reduction of remote edges by about 100 remote edges, which translates to almost a 100% reduction compared to the round-robin-strategy.

Thus, we see that the guided-insert-strategy results in better results (insertion-times, number of movements and remote edges) as the plain round-robin-strategy by sacrificing on load-balancing to improve the initial vertex-placement. The downside of the guided-insert-strategy is the slightly worse load-balancing compared to the round-robin scheme which can be accepted as a trade-off for better performance.

V. RELATED WORK

Other graph processing systems include Pregel [14], Stanford’s GPS [20], X-Stream [19] and PowerGraph [7]. These systems follow a Bulk Synchronous Parallel model [3, 5] as they optimize for static graph analysis.

Another set of graph engines of interest are temporal graph engines such as Grace [15] or Chronos [9]. These systems operate on a snapshot basis to allow graph algorithms to run in parallel with new insertions.

For partitioning the graph, this paper is using a similar set of heuristics as proposed by Stanton et al [21]. Other approaches include algorithms which retain equal-sized partitions [12, 16]. While these approaches work well for offline analysis, they need the system to be locked for updates to run their scheme rendering it unsuitable for real-time analysis.
The idea to use a graph-structure for a camera-network was inspired by Xu et al [22].

VI. CONCLUSION

We present an architecture for distributed graph processing for spatiotemporal queries in the context of camera networks which fully distributes the workload of inserting vertices and executing queries among worker nodes. We also provide a graph partitioning heuristic that provides for high edge-locality while still providing for a load-balanced vertex-distribution among the workers. The results show that the fully distributed system is a scalable solution when used with the weighted greedy strategy and scales well with the number of workers and the problem size. Among all heuristic partitioning strategies, the weighted greedy strategy performs best in the number of remote edges, the distribution of vertices among the workers and, most importantly, the effect of the partitioning strategy on the execution time of tasks in the system. The weighted greedy strategy exhibits comparable performance as a basic hashing strategy in terms of inserting a vertex into the system or querying the system despite doing the additional work of moving vertices between workers. Additionally, the weighted greedy strategy also offers the added benefits of resulting in a small number of remote edges which translates to very good edge-locality when compared to the hashing strategy which performs very poorly for both of these metrics. We thus conclude that the distributed architecture with the weighted greedy partitioning scheme offers a viable solution to the problem of efficient graph processing and online graph partitioning in the context of camera networks.

A possible extension of the architecture of CAMGRAPH would be a departure from a traditional datacenter-centric approach towards a P2P architecture. This would allow the use of the system in a widely distributed area, making use of nearby edge-resources or mobile devices carried by end-users and could be built on top of a previously developed abstraction for pervasive mobile computing [10].

VII. ACKNOWLEDGMENT

We thank NSF for supporting this work through the award NSF-CPS-1446801 and also Microsoft for providing access to the Azure platform for our experimentation.

REFERENCES