Berger *et al*. *Intell Robot* 2024;4(1):87-106
DOI: 10.20517/ir.2024.06

Intelligence & Robotics

Research Article

Open Access

Leveraging active queries in collaborative robotic mission planning

Cyrille Berger, **Patrick Doherty**, **Piotr Rudol**, **Mariusz Wzorek**

Department of Computer and Information Science, Linköping University, Linköping 58183, Sweden.

Correspondence to: Dr. Cyrille Berger, Department of Computer and Information Science, Linköping University, Buildings B & E, Campus Valla, Mäster Mattias väg, Linköping 581 83, Sweden. E-mail: cyrille.berger@liu.se

How to cite this article: Berger C, Doherty P, Rudol P, Wzorek M. Leveraging active queries in collaborative robotic mission planning. *Intell Robot* 2024;4(1):87-106. http://dx.doi.org/10.20517/ir.2024.06

Received: 12 Oct 2023 **First Decision:** 15 Dec 2023 **Revised:** 1 Feb 2024 **Accepted:** 28 Feb 2024 **Published:** 18 Mar 2024

Academic Editors: Jianjun Ni, Simon X. Yang **Copy Editor:** Dong-Li Li **Production Editor:** Dong-Li Li

Abstract

This paper focuses on the high-level specification and generation of 3D models for operational environments using the idea of *active queries* as a basis for specifying and generating multi-agent plans for acquiring such models. Assuming an underlying multi-agent system, an operator can specify a request for a particular type of model from a specific region by specifying an active query. This declarative query is then interpreted and executed by collecting already existing data/information in agent systems or, in the active case, by automatically generating high-level mission plans for agents to retrieve and generate parts of the model that do not already exist. The purpose of an active query is to hide the complexity of multi-agent mission plan generation, data transformations, and distributed collection of data/information in underlying multi-agent systems. A description of an active query system, its integration with an existing multi-agent system and validation of the active query system in field robotics experimentation using Unmanned Aerial Vehicles and simulations are provided.

Keywords: Autonomous robots, active queries, automated planning, situational awareness, 3D reconstruction, unmanned aerial vehicles

1. INTRODUCTION

Collaborative robotic scenarios often involve teams of heterogeneous robotic systems collaborating among themselves and with human agents to achieve complex goals. Mission planning in this context has a high

© The Author(s) 2024. **Open Access** This article is licensed under a Creative Commons Attribution 4.0 International License (https://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, sharing, adaptation, distribution and reproduction in any medium or format, for any purpose, even commercially, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

degree of complexity due to a number of factors:

- Teams of robotic and human agents are highly dynamic, where teams often reconfigure themselves during ongoing missions based on their accessibility, scheduling, and energy limitations;
- Due to the heterogeneity of the robotic systems, appropriate suites of sensors must be chosen for particular missions;
- Missions are highly distributive due to the nature of collaboration, the complexity of the operational environment, and the use of multiple agents;
- No single agent has a real understanding of the collective situation awareness of a participating team due to the distribution of data, information, and knowledge among team members;
- And due to the above considerations, human operators responsible for planning missions can be overwhelmed by the different degrees of complexity involved and would seem to operate more efficiently by abstracting away from the details of mission execution.

The motivation for this paper is to propose an interaction tool for assisting in planning collaborative robotic missions based on the use of "active queries". The intent is to deal with the list of complexities specified above by automating much of the process of certain types of mission planning through the use of active queries.

In this context, a human operator can specify an active query that, from the perspective of the operator, states high-level goals that need to be achieved. Such goals are often informational and are intended to acquire additional situational awareness of a specific operating environment, but the approach is much broader than that.

An active query can automatically initiate many different agent processes in order to answer a query such as finding and delegating tasks to robotic team members required to achieve the specific goals implicit in the query. These processes make up the "active" part of the query, making it distinct from a standard query to a database system. Since each agent on a collaborative team has its own knowledge base, an active query would consist of a combination of both standard queries in a distributed context, in addition to the use of active processes invoked to achieve overarching goals implicit in the original active query.

More specifically, in this paper, we consider information gathering missions that increase the degree of situation awareness human and robotic agents have in emergency rescue operations, where knowledge of the physical environment is essential in ensuing rescue operations. Situational awareness is dynamic and one has to continually acquire and re-acquire such awareness, often actively through interaction with the operational environment.

In the system proposed, team agents can collect, process, store, and exchange data, information, and knowledge in a distributed manner. Agents can also delegate tasks to each other and execute them to achieve goals. Information acquisition and task allocation among team agents will be determined in part using active queries that assume the use of an underlying distributed task- and information-based system [1,2]. The underlying system has been developed in previous work and deployed in field robotic experimentation. The active query functionality is part of an interactive system intended to be used by human operators in mission planning and leverages this complex system.

Figure 1 provides an overview of the active query system and its intended use in the context of a broader multi-agent system. Assume the multi-agent system consists of a team of agents, where each agent has its own local knowledge base and data repositories, and as a member of a team, an agent has the ability to delegate tasks to other agents on the team to achieve more complex goals. Such goals can include joint information gathering tasks, which are the focus of this paper, in particular, the generation of 3D models for parts of the operational environment. In this context, a typical active query might ask for a 3D model of a specific type and resolution, for a specified region of the operational environment.

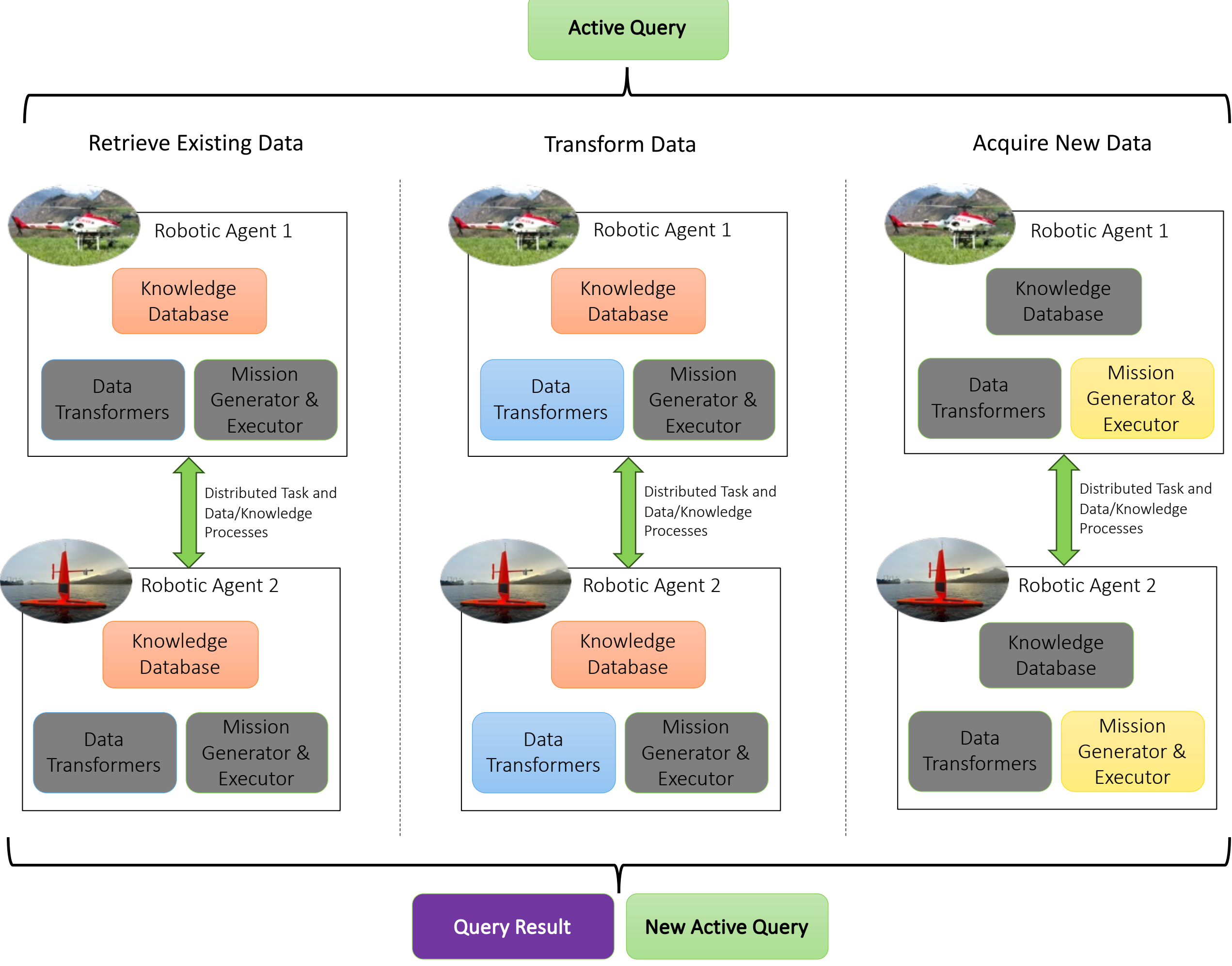

Figure 1. High-level overview of the proposed active query system. Colored boxes (non-gray) visualize system components actively used during three different types of mechanisms the system utilizes to generate a query result.

An active query is parsed and interpreted by an active query engine and then executed based on its interpretation. The execution process is recursive in nature since it may generate new active queries in order to satisfy the activities implicit in the original active query. During the interpretation process, an active query may be directed to execute one of three processes or a combination of these processes [Figure 1]:

- **Retrieve existing data/information** – This is the most straightforward process. Each team agent member is assumed to have a knowledge base and data repositories. If the data/information required already exists in one or more agent knowledge bases or repositories, then this information can be retrieved in a standard manner using a specific query mechanism such as SQL or SPARQL. Note that the query may be required to fuse distributed data/information from several agents [Figure 1, left].

- **Transform data** - Agent systems may use various data types or resolutions in collecting data/information. An active query may ask for data/information in a specific format or resolution. Transform processes are built into the execution mechanism that may first retrieve data/information from one or more agents in a specific format or resolution and then transform it to the desired format or resolution.

For example, agents may have raw Light Detection and Ranging (LIDAR) scans but the active query asks for a point cloud of a specific region. In this case, the LIDAR scans must be fused and transformed into a specific point cloud format of a specific resolution [Figure 1, center].

- **Acquire new data** - This is the most complex process, and it is the main reason why an active query is, in fact, called "active". An active query may include a request for data/information that is not part of any of the team agent knowledge bases or data repositories or only exists partially. In this case, the active query engine has to determine what part of the desired query result already exists and what part needs to be actively acquired. The part that already exists can be retrieved through the retrieval processes and possibly transformed using the mechanisms previously considered. The missing data/information has to then be acquired by generating a mission that may include using several agent systems on the team to collect the missing data/information.

  For example, an active query may ask for a 3D model for a specific region where some portion of the model can already be provided by different agents on the team. The rest of the 3D model has to then be generated through an agent mission where one or more agents with appropriate sensors go out and collect the missing part of the 3D model. This has to then be fused with the existing part of the model [Figure 1, right].

It should be noted that the active query mechanism described herein is an independent module that can be integrated with any type of single- or multi-agent system based on specification of a particular underlying data model and the use of underlying distributed delegation and planning systems. In this particular case, we use an underlying system, the SymbiCloud Framework[1], to do this, but it is not a pre-requisite for using the active query mechanisms in other systems or contexts.

1.1. Contributions

The main contributions of this paper are:

- A specification of an "active query language" and engine tailored to the automatic acquisition of situation awareness by a multi-agent system. In this context, situation awareness will be associated with the acquisition of 3D models for an operational environment.
- An "evaluation strategy" (as a set of rules for evaluating expressions in programming languages[3]) and associated algorithms for the active query language which permits the interpretation and execution of active queries.
- Description of a larger prototype system that utilizes the proposed active query language and engine and leverages an existing multi-agent system used primarily for generating and executing collaborative plans.
- Experimental validation of the prototype system in multi-agent "field-test experiments" using unmanned aerial vehicles (UAVs) and in simulation experiments.

The paper is organized as follows. In Section 2, the query language and the algorithms for its evaluation strategy are presented. Section 3 showcases the results of field-test experiments. Related work and conclusions are described in Sections 4 and 5, respectively.

2. ACTIVE QUERIES

This section presents a specification of an Active Query (AQ) language and a framework for AQ evaluation and execution. The proposed system module, the Active Query Engine (AQE), allows users and agents to generate and execute AQs that acquire information from a team of collaborating autonomous agents operating in a shared operational environment. The AQ language, scQL (scQL stands for SymbiCloud Query Language, as it is an extension of the SymbiCloud Framework[1] which is integrated with and leveraged by the AQE), is a query language tailored for asking questions about situation awareness, with syntax inspired by SPARQL[4]. User queries in text format are processed by the AQE which consists of three components: a "parser", a "compiler",

and an "executor". The "parser" extracts a symbolic representation of the query. The "compiler" instantiates, implements and prioritizes data source operators for each of the relevant data sources associated with the active query. The "executor" uses the instantiated data source operators and its access to the relevant "data sources", instances of which are distributed among team agents, to generate an answer to the AQ by reasoning about (1) existing data; (2) existing data that can be transformed or fused; and (3) missing data that needs to be acquired or generated by executing new exploration missions [Figure 1]. The AQE is generic in nature, where the AQ types that can be evaluated depend on the underlying data model used within the team of collaborating agents. The underlying data model used in this work is found in existing work with distributed knowledge-based systems[1], where agents store data, information, knowledge, and metadata about collected raw sensor data in the form of RDF Graphs[5] that are automatically synchronized among all team members. The AQE executes queries using this metadata model.

An overview of the AQE is shown in Figure 2. The engine has three main functional components, namely, parsing the AQ, query compilation, and its execution. Data source operators are associated with each of the AQ data sources which allow the query engine to retrieve data from a data source, transform its data, or acquire data actively from the environment. When data sources require acquisition of new data (e.g., to transform or acquire), new active sub-queries are generated and processed recursively by the AQE. Execution of new exploration missions is coordinated by underlying task execution frameworks such as the delegation system presented in ref[2].

The syntax of the proposed scQL language is specified in Section 2.1, while its semantics is defined by the underlying data model associated with AQ data sources (Section 2.2) and the scQL evaluation strategy (Section 2.3).

2.1. AQ Syntax

The proposed "scQL query" language uses the following syntax:

```
1 SELECT <type> AS name WHERE conditions
```

where `<type>` is a URI identifying the requested type of data (e.g., `<lidar_scan>`, `<image>`). The `name` is an identifier used in `conditions` to specify constraints the returned data should satisfy. The `conditions` take a form similar to SQL[6] and SPARQL[4] queries:

```
1 name.parameter_0 op value_0 [AND name.parameter_1 op value_1] ...
```

where `parameter_i` is a named parameter specific to the type of data (e.g., `point_density` for a point cloud). The `op` is an operator, such as (in)equality, set (inclusion), *etc*. The `value_i` is a value defining a constraint.

Parameter names and their datatypes are dependent on the model of data that is queried (`<type>`). For instance, in the case of a point cloud type (`<http://askco.re/scql/types#point_cloud>`), the parameters include a `timestamp` of when the data was acquired, a `point_density` of the point cloud, and the `geometry` condition describing a bounding polygon of the point cloud. The query result is a list of objects.

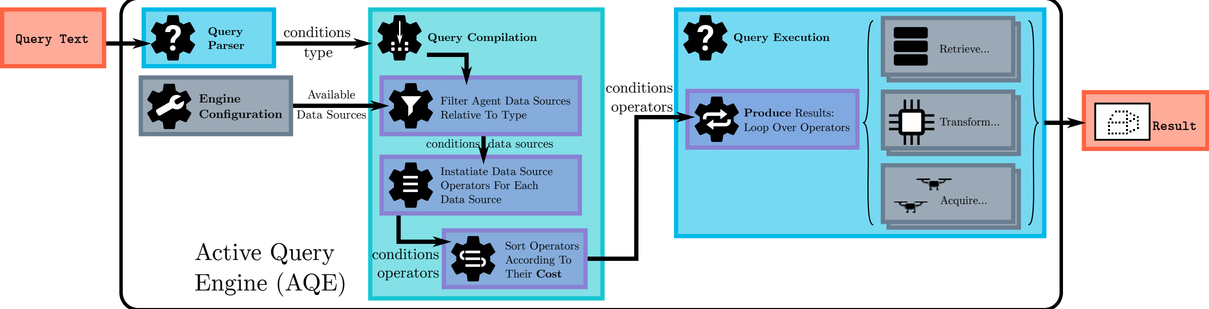

Figure 2. Overview of the AQE. The input AQ in text form is parsed and executed by the query execution algorithm to generate results by combining outputs of different data sources, which contain pre-existing, acquired, or transformed datasets. AQE: Active Query Engine; AQ: Active Query.

Query 1: An example query for point cloud data of density superior to 20 point/m^2.

```
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT <http://askco.re/scql/types#point_cloud> AS pc
    WHERE pc.geometry = "POLYGON(...)"^geo:wktLiteral
          AND pc.point_density >= 20 point/m^2
```

Query 1 shows an example query where data for a point cloud is requested with a point density superior or equal to 20 point/m^2 covering the `"POLYGON(...)"^geo:wktLiteral`. The compact URI `geo:wktLiteral` (line 3) is expanded into a full URI with the `PREFIX` definition to `<http://www.opengis.net/ont/geosparql#wktLiteral>`, and it indicates that the string `"POLYGON(...)"` should be interpreted as a Well-Known Text (WKT) representation [7], a standard used for representing geometric and geographical features. The result of the query would be a list of point clouds.

2.2. AQ data sources and operators

Integrating the AQE in a practical system is done by configuring a list of available "data sources" associated with team agents and then implementing "data source operators" associated with each data source. These data source operators are instantiated relative to the data type during the active query compilation process. Data sources provide the origins encompassing the agent teams in question and are used collectively to answer active queries. Examples include databases, sensors, repositories, *etc*. For the purpose of this paper, data sources are databases associated with each team member. These databases contain pre-existing, acquired, or transformed datasets, among other entities.

"Data Source Operators" (DSOs) are functional units used to access or transform data from data sources or acquire new data. Figure 2 shows a detailed schematic of the processes associated with the AQE that leverage data sources and operators to answer complex active queries. A DSO implements an interface for each data source in the form of three functions: (1) a cost function; (2) a data-producing function; and (3) a condition update function. The cost function, in the form $cost(conditions)$, returns a cost value for providing data from each DSO associated with a data source given a set of conditions. Typically, a cost can be associated with the time it takes to provide an answer. The output of the cost functions is used to sort the relevant DSOs during query compilation. The data-producing function, in the form $produce(conditions)$, returns data from a data source using a DSO and a given set of conditions applied during query execution. The condition update function, in the form $update(result, conditions)$, updates the current set of conditions. For example, the update may be used to exclude partial data that has already been added to the final result of a query in the recursive processing of the algorithm to exclude overlapping areas.

The current version of the proposed AQE supports three types of DSOs:

- $retrieve$ - a data source operator that can query a local or federated database for data of a given $type$. We denote it as $retrieve(type)$ for the instantiation of a retrieve operation for $type$.
- $transform$ - a data source operator that can transform source types ($source_types$) to another data type ($type$). We denote it as $transform(source_types..., type)$ for the instantiation of a transform operation specific from $source_types...$ to $type$. An example would be fusing raw LIDAR scans into a point cloud.
- $acquire$ - a data source operator that can acquire data of a particular $type$, for example, by executing an exploration mission using a robotic system equipped with an appropriate sensor. We denote it as $acquire(type)$ for the instantiation of an exploration operation specific to acquiring data of the given $type$.

The AQE can use multiple instances of each data source operator. This feature provides the flexibility to access and retrieve data from different databases and allows for the inclusion of various transforming methods (e.g., generating point clouds based on LIDAR scans or image data).

The AQE relies on $cost$ functions to prioritize the use of data source operators, allowing the system to adapt to diverse operational conditions (Figure 2 - Query Compilation). A practical example includes accounting for typical distances between robotic agents. In some scenarios, agents may need to be nearby to reliably exchange large amounts of data due to limitations of communication links. Thus, the cost functions can be adjusted to prioritize the execution of new exploration missions over exchanging large amounts of existing data, potentially involving commanding the agents to move closer to each other. In the most common case, however, when agents operate in environments with reliable communication links, the $cost$ functions generally follow a generic order:

$$cost(retrieve) < cost(transform) < cost(acquire) \tag{1}$$

2.3. AQE

The AQE execution algorithm is presented in Algorithm 1. The input to the algorithm is an AQ in text form and a configuration. The configuration includes, among other information, a list of accessible data sources associated with the team of agents. The first step of the algorithm involves parsing the input query and extracting $type$, $name$, and $conditions$ (line 1) from it. The $conditions$ are a list of tuples ($name.parameter_i$, op_i, $value_i$). The result of parsing Query 1, previously considered, would result in $type$ equal to `http://askco.re/scql/types#point_cloud`, $name$ equal to `pc` and $conditions$ equal to $\{(pc.geometry, =, POLYGON(...)), (pc.point_density, \geq, 20 \text{ point/m}^2)\}$.

In the next step, the $FilterDataSources$ function (line 2) returns a list of possible $datasources$ that can be used to produce data for the given $type$ and AQE $configuration$. On line 3, the data source operators are instantiated for each data source. The $Sort$ function (line 4) uses the $cost$ function of each operator to sort the operators. The algorithm's main loop (lines 6-9) updates the query $result$ by retrieving relevant data using the different data source operators associated with the data sources. Simultaneously, the $conditions$ are updated by the $UpdateConditions$ function (line 8). The condition update is necessary to prevent retrieving unnecessary data from several sources if the associated data in the data sources overlap. The $geometry$ condition is also updated to exclude the area already covered by the current data source. The $AddToResult$ and $UpdateConditions$ functions (lines 7 and 8) internally utilize the $produce$ and $update$ functions of each data source (Section 2.2).

Figure 3 depicts the AQE execution flow, assuming the system uses the cost function order defined in Equation (1). First, the AQE attempts to reuse existing data from a database (step 1 in the figure), which may require downloading the data from a different agent's database (step 2). Next, the executor checks if it is possible to

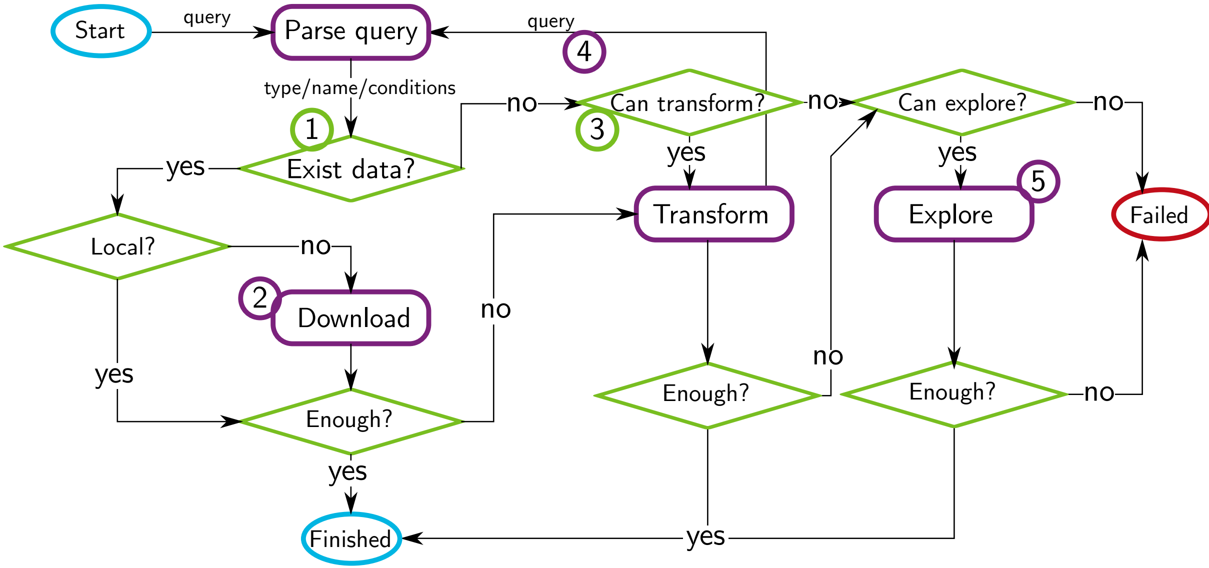

Figure 3. The AQE execution flow, assuming the cost of retrieving data is lower than performing data transformation.

transform existing data (step 3). If this is the case, the AQE will generate a new active sub-query (step 4) to access any data needed for the transformation. The generated sub-query is then evaluated by a recursive call to the ExecuteQuery algorithm (Algorithm 1). Finally, if data is still missing, the AQE generates and executes an exploration mission (step 5), which results in the acquisition of the missing data.

The behavior of the ExecuteQuery algorithm can be illustrated by studying the evaluation of Query 1 based on the example scenario depicted in Figure 4, where the user requests point cloud data with at least 20 point/m^2 density. At the start of the evaluation, three existing datasets (A, B, and C) that overlap with the geometric area defined in the query are available. These datasets may exist locally on different agent systems. A federated database that can access all datasets residing in different agent databases may be viewed as a data source in this example. We denote this data source as *DS-Fed*. Datasets A and B contain point cloud data, with densities of 30 and 10 point/m^2, respectively. Dataset C includes raw LIDAR sensor data scans with a 20 point/m^2 density. Thus, data included in dataset A can be used directly, while data from dataset B lacks density. Dataset C contains data that can be transformed into a point cloud with sufficient density. Assuming the cost function order defined in Equation (1), the algorithm will prefer reusing existing data (retrieve) over performing transformation or acquiring new data. Consequently, retrieving the requested data type directly from a database (using

Algorithm 1 ExecuteQuery($query$)

Require: $query$ a valid text query, $configuration$ the AQE configuration

1: $type, name, conditions \leftarrow Parse(query)$
2: $datasources \leftarrow FilterDataSources(type, configuration)$
3: $datasourcesoperators \leftarrow Instantiate(datasources)$
4: $datasourcesoperators \leftarrow Sort(datasourcesoperators, conditions)$
5: $result \leftarrow \emptyset$
6: **for all** $operator \in datasourcesoperators$ **do**
7: $\quad result \leftarrow AddToResult(result, operator, name, conditions)$
8: $\quad conditions \leftarrow UpdateConditions(result, operator, name, conditions)$
9: **end for**
10: **return** $result$

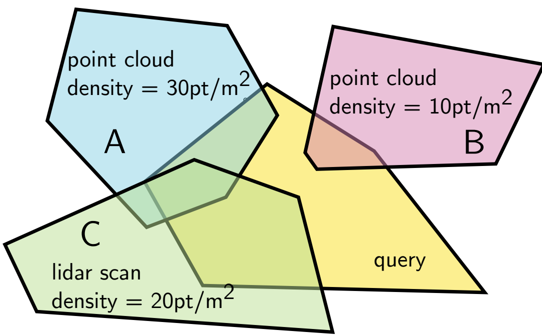

Figure 4. Example scenario for Query 1 evaluation. The yellow area represents the geographic area defined in the query. The blue (A) and purple (B) datasets are existing point clouds with respective densities of 30 and 10 point/m², while the green (C) dataset is raw LIDAR sensor data that can be transformed into a point cloud with a density of 20 point/m². Datasets A, B, and C are part of the data source associated with the federated database accessible by all agents.

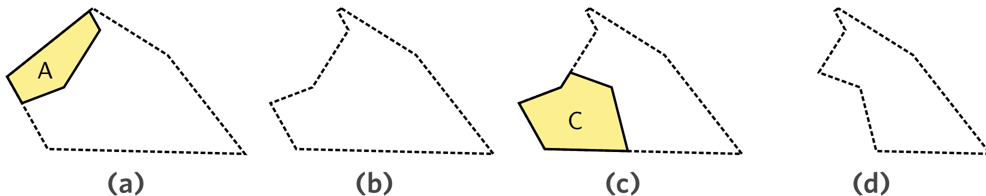

Figure 5. Evolution of the `geometry` condition during evaluation of the example query presented in Figure 4. (a) depicts the intersection of the query geometry with region A; (b) shows `geometry` condition after retrieving data from region A; (c) demonstrates the intersection of remaining `geometry` condition with region C, since that data can be used for generating a point cloud; (d) indicates the remaining region that will be used to execute an exploration mission to acquire data that is not present in existing databases.

dataset A) will be prioritized over transforming (using dataset C) or acquiring new data. In the following, we denote a point cloud type as pc and a LIDAR scan type as ls. The function $FilterDataSources$ will return two data source operators associated with the data source DS-Fed, $retrieve(pc)$ and $transform(ls, pc)$; the latter transforms LIDAR scans to point clouds. Note that the point cloud data required cannot be acquired directly by executing an exploration mission. Thus, an $acquire(pc)$ data source operator for DS-Fed does not exist and is not generated in this case.

Given the two previously instantiated data source operators $retrieve(pc)$ and $transform(ls, pc)$, the ExecuteQuery algorithm will first consider $retrieve(pc)$ and reuse part of dataset A and discard dataset B, as it does not satisfy the point cloud density constraints. After retrieving the data from the data source DS-Fed using the operator $retrieve(pc)$, the algorithm updates the `geometry` condition [Figure 5]. Figure 5A depicts the initial overlap of dataset A with the query region, and Figure 5B represents the updated `geometry` condition after the reuse of dataset A. Next, the ExecuteQuery algorithm will consider the second data source operator, $transform(ls, pc)$, associated with the data source DS-Fed. To perform the transformation, the data source operator first needs to retrieve LIDAR scans (`<http://askco.re/scql/types#lidar_scan>`). Hence, the algorithm generates a new active sub-query (Query 2) that is evaluated in a direct recursive call to the ExecuteQuery algorithm.

Query 2: New sub-query for LIDAR scans with point density superior to 20 point/m^2.

```
1 PREFIX geo: <http://www.opengis.net/ont/geosparql#>
2 SELECT <http://askco.re/scql/types#lidar_scan> AS scan WHERE
3     scan.pose in "POLYGON(...)"^geo:wktLiteral
4     AND scan.fuse_to_point_density >= 20 point/m^2
```

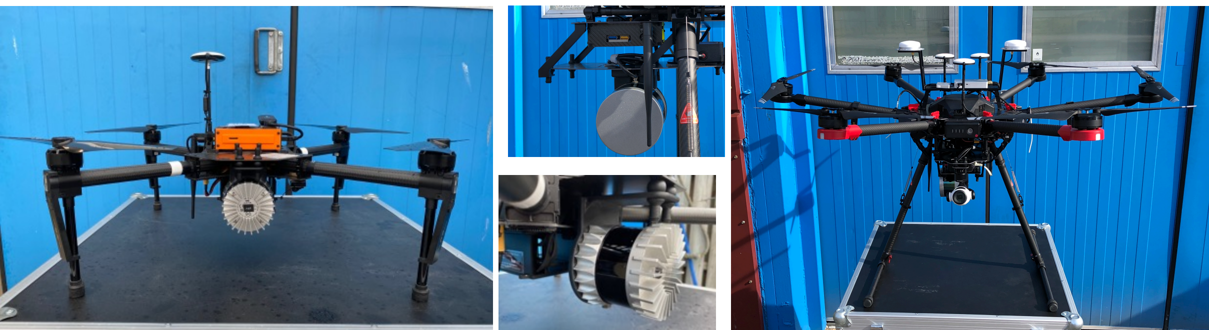

Figure 6. Experimental platforms: DJI Matrice 100 equipped with an Ouster OS1 sensor (left), DJI Matrice 600 Pro equipped with a Velodyne Puck sensor (right).

The ExecuteQuery algorithm will consider two new data source operators associated with the data source, DS-Fed, during evaluating the generated sub-query: $retrieve(ls)$ and $acquire(ls)$ operators. The $retrieve(ls)$ operator will be considered first based on the cost function order, as accessing existing data is deemed faster than acquiring new sensor data. Consequently, after retrieval of overlapping data from dataset C, the algorithm updates the `geometry` condition. Figure 5C exhibits the initial overlap of dataset C with the remaining query region before the sub-query evaluation, and Figure 5D represents the updated `geometry` condition after the reuse of dataset C. Data for the remaining area needs to be acquired by executing an exploration mission which will result from the evaluation of the second data source operator, $acquire(ls)$. The sub-query result will be returned once the remaining data is acquired, ending the recursive call. The algorithm continues with the execution of the LIDAR scan fusion using the operator $transform(ls, pc)$, after which the final result to the original query is returned.

3. EXPERIMENTAL EVALUATION

3.1. Evaluation with deployed robotic systems

In this section, an example use of the AQE in a field-test experiment is presented. The mission is similar in type to the example presented in Section 2.3. The mission has been performed using actual robotic platforms operating in an outdoor operational environment. It includes three agents: a ground operator and two UAVs [Figure 6]. One UAV is a DJI Matrice 600 (https://www.dji.com/se/matrice600-pro) equipped with a 3D LIDAR sensor from Velodyne (https://velodynelidar.com/products/puck/). The other is a DJI Matrice 100 (https://www.dji.com/se/matrice100) equipped with a 3D LIDAR sensor from Ouster (https://ouster.com/products/scanning-lidar/os1-sensor/). Both UAVs are equipped with Intel NUC computers (https://en.wikipedia.org/wiki/Next_Unit_of_Computing#Tenth_generation) that host the software. The communication between agents and software modules is handled using ROS[8] middleware. Each agent system runs several software modules that include the proposed AQE, a "Knowledge DataBase" (KDB) associated with the underlying "SymbicCloud System"[1], and a "Delegation System"[2] responsible for the generation and execution of multi-agent tasks. The latter two are existing frameworks available on our robotic systems and operate independently of the AQE.

The AQE presented in this paper is generic and stand-alone in the sense that it could be interfaced to any underlying multi-agent system architecture. A minimal constraint for the underlying multi-agent system would be that each agent has a database that can contain one or more datasets and that the system has the ability to generate multi-agent task plans.

The KDB is a distributed knowledge database system used for multi-agent teams, which allows agents to generate, retrieve, and synchronize RDF Graphs contained in databases and exchange sensory and other datasets

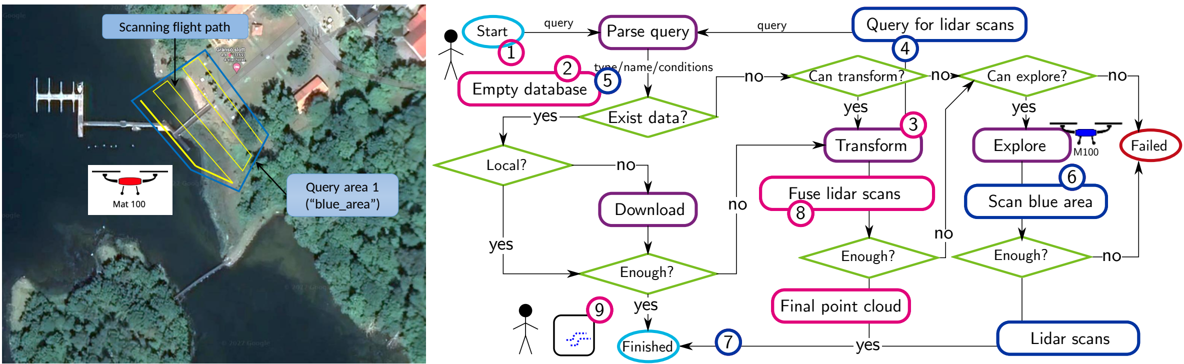

Figure 7. Execution flow for AQ 1 (similar to Query 1) where a ground operator requests a point cloud for a selected (blue) region. The numbers highlight the important steps. The pink numbered circles indicate the execution flow for the original point cloud query. The blue numbered circles indicate the execution flow for the generated sub-query for the LIDAR scans required to obtain the originally requested point cloud. The resulting flight paths for the executed exploration mission are shown in yellow. AQ: Active Query; LIDAR: Light Detection and Ranging.

also contained in databases. The system enables agents to synchronize metadata information regarding available datasets and their properties. For example, this enables every agent to be aware of the existence and attributes of datasets A, B, and C in the example scenario presented in Figure 4, regardless of whether the data is stored locally in different agent systems. Each agent can use the "KDB Manager" to request a dataset transfer, allowing them to get a local copy of datasets A, B, or C. The "Delegation System" handles task execution for an agent and enables task distribution among multiple agents when required. For example, an exploration task can be delegated to a single agent or distributed among many agents.

The AQE uses both the "KDB Manager" and "Delegation System" to produce answers to active queries. The "KDB Manager" is used for accessing existing knowledge shared among agents [$retrieve(type)$ data source operator], while the "Delegation System" is used for executing data transformations [$transform(source_types..., type]$ data source operator) and exploration missions [$acquire(type)$ data source operator]. Conceptually, integration between the AQE and the KDB is made by assuming each agent on a team has a database, that database is a data source, and each database consists of one or more datasets.

In the field robotics experiment presented in this section, a ground operator agent first generates two AQs requesting 3D models in the form of point clouds for two overlapping areas in the environment. This requires the exploration of two regions and the fusion of the acquired LIDAR data into a single point cloud per query. As a result, the AQE automatically generates and coordinates the executions of two exploration missions using the two UAVs on the team. An overview of the two generated missions is presented in Figures 7 and 8. Recall that these query types have been previously specified and discussed in the example presented in Section 2.3, although for the field experiment, different parameterizations of queries are used. In this experiment, data sources used by the AQE include databases associated with each individual UAV agent and the ground operator agent. Each database contains pre-existing, acquired, or transformed datasets.

Figure 7 shows the execution process of the first AQ and highlights the important steps. The ground operator selects a region in a user interface (the blue area in the figure) and executes a data request query (similar to Query 1 in Section 2.1) for a point cloud data type (step 1). The system checks for existing data in the database associated with the ground operator (step 2). Since no existing data is available to answer the operator's request, the system checks if it is possible to transform other existing sensor data into a point cloud (step 3). To generate a point cloud, the system needs to fuse LIDAR scans. Consequently, to access LIDAR scans, the system generates an active sub-query (step 4) for these scans (similar to Query 2 in Section 2.3) which will be delegated to the UAV agents.

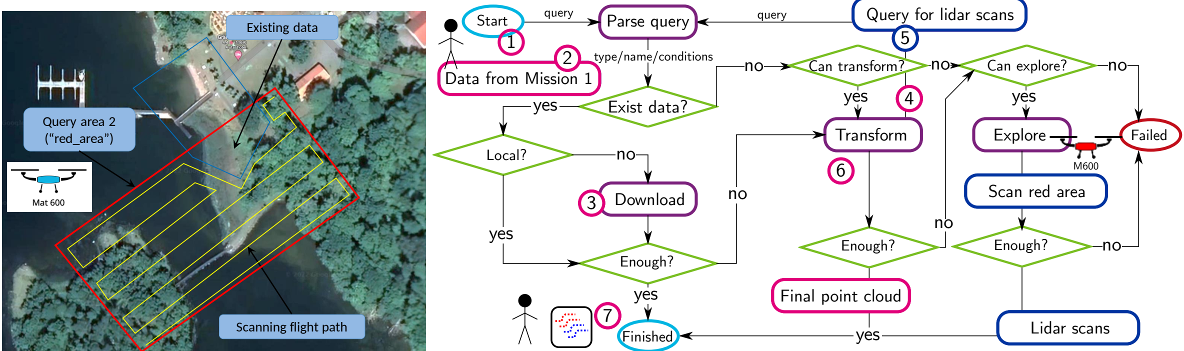

Figure 8. Execution flow for AQ 2 (similar to Query 1), where a ground operator requests a point cloud for a selected (red) region. The numbers highlight the important steps. The pink numbered circles indicate the execution flow for the original point cloud query. The blue numbered circles demonstrate the execution flow for the generated sub-query for the LIDAR scans required to obtain the originally requested point cloud. The resulting flight paths for the executed exploration mission are shown in yellow. AQ: Active Query; LIDAR: laser imaging detection and ranging.

The sub-query execution starts with a check for existing LIDAR scans in the database associated with the ground operator (step 5). Since the data does not exist, and LIDAR scans cannot be generated from other data types, the data must be acquired directly from sensing (step 6). An exploration mission for the region is automatically generated and delegated to a UAV Matrice 100 to acquire raw LIDAR data ($acquire(type)$ data source operator). The acquired data is saved in the database associated with the UAV Matrice 100, ending the execution of the sub-query (step 7).

To complete the interpretation and execution of the original point cloud active query (AQ 1, similar to Query 1 in Section 2.1), the LIDAR scans need to be fused into a point cloud (step 8). For this purpose, a computation task (a *transform()* data source operator associated with the Matrice 100 database) is delegated to the Matrice 100, resulting in the generation of a point cloud covering the blue area. The data is then synchronized between the Matrice 100 and the ground operator using the SymbiCloud and Delegation systems, and it is then displayed in the user interface (step 9).

Figure 8 shows the process of executing the second active query (AQ 2) and highlights the important steps. In this case, the ground operator selects a region in the user interface (the red area in the figure) and executes a data request query for a point cloud data type (step 1, similar to Query 1 in Section 2.1). The system checks for existing data in the database associated with the ground operator (step 2). Since the query area (red) overlaps with the area requested in the previous AQ (blue), existing data can be reused. The data for the point cloud of the previous AQ is available in the Matrice 100 database and needs to be downloaded to the ground operator's database. The point cloud can be shown directly to the user after the data is downloaded (step 3). The remaining steps required to complete the execution of the AQ are similar to the first query from the ground operator. To generate the remaining point cloud data (step 4), it is necessary to transform LIDAR scans using a $transform(ls, pc)$ data source operator, which results in an execution of a sub-query (step 5). The sub-query (similar to Query 2 in Section 2.3) execution triggers an exploration mission for part of the query area (the red area, excluding the overlap between the red and blue regions). The mission is delegated to the UAV Matrice 600, and the acquired LIDAR scans are combined in a point cloud (step 6). Finally, the two point clouds are combined for visualization (step 7). The resulting point cloud data is shown in Figure 9.

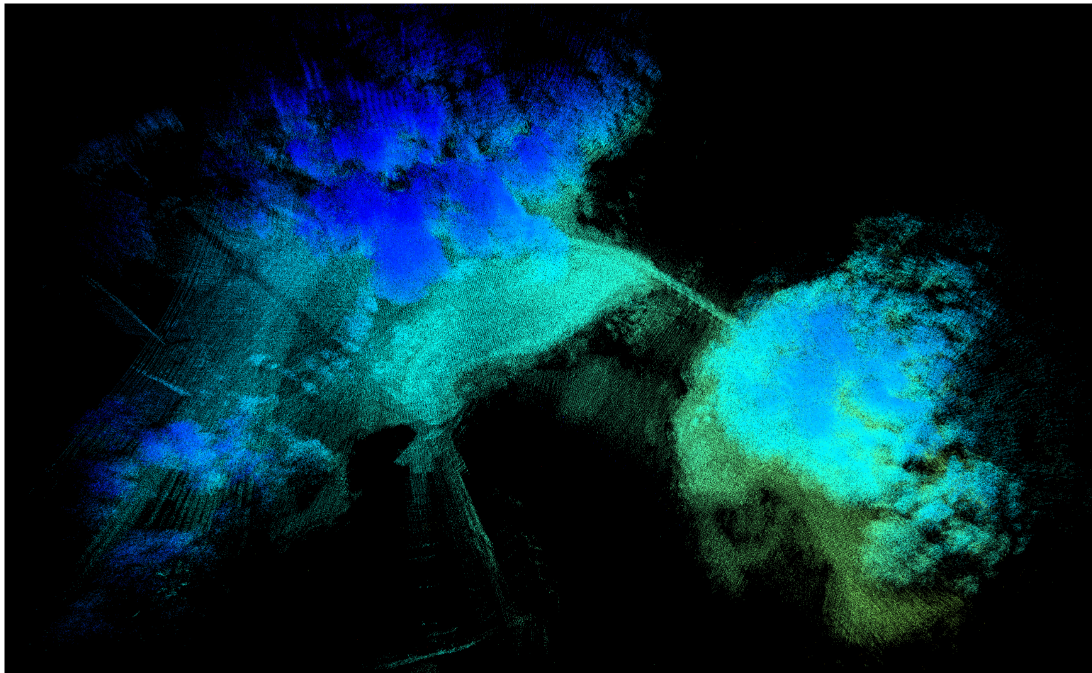

Figure 9. Final results of the field-test experiment generated by executing two active queries.

3.2. Comparison with other approaches and scalability evaluation

Related research with sophisticated multi-agent systems consisting of operational, heterogeneous autonomous systems interacting with multiple ground operators is sparse. The state-of-art in this respect still involves a great deal of manual steps at the task level needing to be specified for each individual agent system in a larger mission. Progress at the automated multi-agent path planning level is somewhat more developed [9], although a broader solution to robust and useful in-field multi-agent systems requires development at both the multi-agent task planning and path planning levels.

This paper focuses on suitable query interfaces for mission task planning for multi-agent systems, in particular, generation of 3D models, where existing approaches for solving multi-agent exploration and distributed situation awareness usually involve many manual steps. Current state-of-the-art puts a heavy burden on human operators tasked to use teams of autonomous systems. Human operators need to explicitly check for existing information in a database, set up exploration missions often specifying each step, and then manually combine newly acquired data with existing information or manually use systems that would do such types of fusion.

The novelty and use of active queries and the AQE focuses on reducing the amount of manual work required from human operators when specifying multi-agent task planning for teams of agents in this context. The goal is to reduce cognitive load and make the process more efficient and automated, allowing the operators to focus on other important tasks and reduce operator errors. The human operators specify the *what* that is required using active queries, and the AQE system generates the *how* through the combined use of retrieval, transformation, and acquisition through multi-agent plans.

To summarize the intent and value of the approach to mission planning that uses active queries, a generic comparison can be made between most existing approaches to the problem and an approach using active queries. Table 1 shows how a single active query can automate many of the manual processes human operators currently use in generating multi-agent mapping missions. The active query technique can be generalized to other application areas in a straightforward manner.

Table 1. List of actions performed by an operator when using and not using the AQE approach

	Using the AQE	Not using the AQE
Operator Actions	Select a region in the environment	Query the database for existing information Set up scanning missions for missing information Download raw sensor data from UAVs Process the raw sensor data Fuse acquired data with existing information

AQE: Active Query Engine; UAVs: unmanned aerial vehicles.

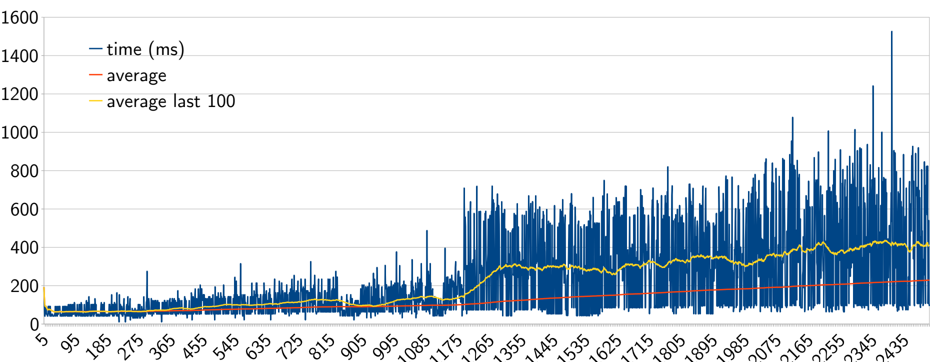

Figure 10. Evolution of the execution time for the active query scalability evaluation (timings do not include performing explorations nor data processing). The blue line shows the timing of individual queries. The red line represents the average over the queries since the origin. The yellow line is an average over the last 100 queries.

In order to evaluate the scalability of the AQE, an experiment was performed where 2,500 queries were executed. The number of queries for the experiment was chosen to be large enough to draw conclusions about the time complexity of the system. To that end, a timing measurement for each query was recovered and analyzed. Additionally, to show how much the AQE system reuses already acquired data, an exploration rate metric was introduced, recorded, and analyzed for each query. The queries were randomly generated and could result in either directly returning a set of LIDAR scans or point clouds. To exclude the influence of actual exploration and data processing times, these steps were not performed. For each query, its execution time and the exploration rate were measured. The exploration rate τ^i_{exp} for query i is defined as follows:

$$\tau^i_{exp} = \frac{\mathcal{A}^i_{exp}}{\mathcal{A}^i_{req}} \quad (2)$$

where $\mathcal{A}^i_{exp}$ is the area of a polygon that needs to be explored to answer a query, while $\mathcal{A}^i_{req}$ is the area of the polygon that was requested in the query i. In other words, if no data is available to answer a query the rate equals 1, and if all data is already available the rate is equal to 0 meaning no additional exploration missions are required.

The timing results are shown in Figure 10. The results show a linear increase of the execution time with the number of queries performed. This is because as more queries are executed, more data is added to the database. In practice, the AQE is as scalable as the underlying data storage. Additionally, we can see a timing jump around the execution of the 1,200th query, caused by hitting the limits of database index caches. After this jump, the timing reverts again to a linear increase. The exploration rate results are shown in Figure 11. The figure shows that the number of required explorations decreases as consecutive missions are performed. This is caused by the fact that the environment has already been increasingly explored.

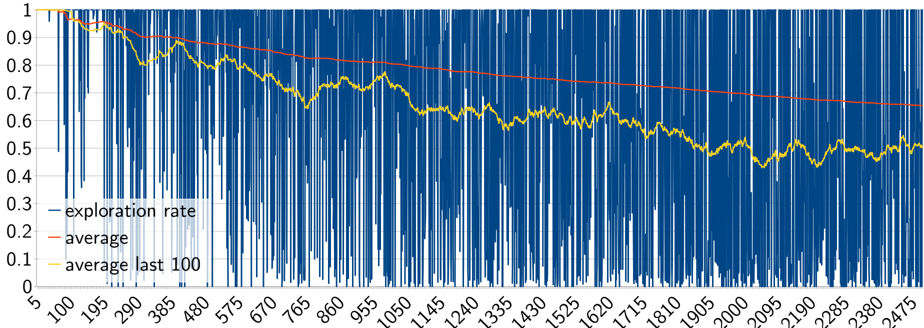

Figure 11. Evolution of the exploration rate for the active query scalability evaluation. The rate represents the ratio between the explored and the total areas requested in a query. The ratio starts at 1.0 when the database is empty and decreases over time. Exploration rates of individual queries are drawn with blue lines. The red line represents the average over the queries since the origin. The yellow line is an average over the last 100 queries.

4. RELATED WORK

Concepts related to active queries have been previously studied in multiple research areas. However, to the best of the authors' knowledge, they have never been considered in the context of actively acquiring situational awareness by teams of collaborative robotic systems. The research relevant to the ideas described in this paper is presented in this section.

Active Sensor Networks. Work presented in this paper is closely related to Wireless Sensor Networks (WSNs)[10], in particular regarding the concept of distributed information gathering for situational awareness. The early methods used in WSNs involved a sense-push approach where a sensor collects data and sends it to a central database, which a user can query. In ref[11], the authors propose a mechanism that allows for reconfiguring a WSN depending on the user's query. Instead of collecting all possible data types, the main aim is to decrease power usage by optimizing the network to collect only relevant information. The key principle of this work is conceptually similar to the approach proposed in this paper in the sense that the system adapts itself to answer a question.

Active vision[12] is a sub-field of active sensor networks, where planning algorithms are used to decide the best position for a camera to make an observation. The algorithms used for active vision are very relevant to implementing $explore$ functionality for the proposed AQE.

Sensing Using Multi-Agent Systems. Conceptually, Multi-Agent Systems (MASs) are similar to WSNs as they include systems equipped with sensors and computing capabilities. However, they typically have more capabilities and can move and act in the environment. In ref[13], a system for acquiring satellite images with an adequate resolution is presented. Low-resolution images obtained by the satellites are sent to a ground station for point-of-interest detection. If high-resolution images are required, based on the user's request, the system uses an automated planner to schedule and select the best satellite to acquire the requested image. The approach has been extended to handle more complex satellite constellations and requests in ref[14]. The idea of choosing the most suitable system to perform observations bears similarities to the work presented in this paper.

Previous work presented in ref[15] focused on using a team of UAVs to build 3D models of the environment and a data/knowledge management infrastructure intended to support distributed, collaborative collection of data and knowledge and its shared use in multi-agent systems[1]. The work presented in this paper extends these distributed exploration frameworks by providing a novel way to represent and execute exploration tasks. Ex-

ploration missions in ref[15] are created manually. In ref[1], the system uses static scripts to check the database for existing data and trigger exploration tasks when required. This paper explores a new general approach to replace manual or static script-based mission generation. This novel technique opens the way to more flexibility in the automatic generation of exploration tasks when necessary, where data queries can include both symbolic and non-symbolic values - for example, querying for the 3D model of buildings in an area.

Query of Sensor Streams. Database query languages have been extended to work on sensor data streams. For example, the SPARQL was extended in ref[16] to query sensor data from a stream using virtual RDF triples. There are similarities between stream querying and our work, where the resulting data may or may not be in a database. Stream querying involves querying continuously produced data, while our work involves reconfiguring the system to produce the desired result.

Active Queries. In ref[17](ACQUIRE), AQs were introduced to minimize the energy cost of executing queries within a sensor network. The typical approach was to broadcast the query causing overloading of the network. ACQUIRE defines a novel strategy to optimize which node to query. The AQ presented in this paper indirectly follows some principles used in ACQUIRE. It uses the hastily formed knowledge network (HFKN) framework[1] to store metadata about collected raw sensor data in the form of RDF Graphs[5] automatically synchronized among all team members. This method allows us to limit the number of queries to agents to those relevant to AQ execution. AQs are often used in the context of systems that require user inputs. These can range from simple suggestions of data to acquire to making requests for help from the user. In ref[18], AQs are used to answer a query about the location of a mobile phone. Automatic calculation of locations based on user-submitted pictures leads to incorrect results in half the cases. The authors propose a mechanism that suggests to the user where to take an additional picture to increase the calculated location certainty.

Another example of AQs was proposed in ref[19] where the system helps a user refine an incomplete query to an image database. AQs are also used in machine learning, where training algorithms can request labels of interesting data points[20]. Similar active strategies have been used in remote sensing to improve the accuracy of image segmentation[21]. When the algorithm is uncertain about a query, it can ask the user to label the data. The AQs presented in this paper do not necessarily involve human-agent interaction other than what the multi-agent system may need for operation in the context of mixed-initiative interactions[22].

Knowledge Generative Queries. In this area, researchers examine using “intentional information” (e.g., metadata, relationships, inference rules) to enhance query outcomes over “extensional information” (actual data in a database). The most common methods consist of algorithms for converting extensional data to generate “intentional answers”[23]. Several techniques have been further developed for “cooperative answering” in databases, including a dialog with the user in case no answer to the query can be provided. Typically, these systems modify the original query and present several alternative answers that users may find satisfactory[24]. All these systems involve some form of knowledge processing and are relevant to this paper, as these approaches can be used to implement $transform()$ data sources in the proposed AQE.

In the area of automated planning, executing a plan can generate knowledge useful to its completion. This is referred to as the concept of the “knowledge producing actions”[25]. The challenge for the planning algorithms is to reason about available knowledge and the knowledge that can be acquired during the plan execution. The “knowledge producing actions” are a means to an end of accomplishing the goal, i.e., what knowledge is needed to reach a goal. In contrast, the AQs presented in this paper focus on producing requested knowledge. Consequently, our work can be integrated as a “knowledge-producing action” executor.

Query Language Syntax. Several existing database query languages offer features similar to those of the scQL language. The most prominent language is SQL[6]. As shown in ref[11] SQL can be used for sensors instead of

databases. Query types presented in this study can be expressed with an SQL syntax, as shown in Query 3.

Query 3: General form of an SQL equivalent to the scQL query.

```
SELECT name FROM type WHERE conditions
```

Consequently, the example query for point clouds (Query 1) could be rewritten as Query 4.

Query 4: Example of an SQL query for a point cloud.

```
SELECT pc FROM pointcloud WHERE
    convert(density, density_unit, "point/m^2")> 20
    AND intersects(area, from_wkt("POLYGON(...)"))
```

where density, density_unit and area are fields of a virtual table called pointcloud. In situational awareness applications, however, one may require combining different data sources – for example, a query for the point cloud of a set of buildings in an area. Using SQL to represent such queries requires the use of join statements, which makes both the interpreter and the writing of the query more difficult. scQL can be easily extended to support such queries without introducing these additional challenges.

Another database query language alternative feasible for representing queries for semantic data is SPARQL[4], along with the use of virtual graphs[16]. The example point cloud query (Query 1) can be rewritten using SPARQL as Query 5.

Query 5: Example of SPARQL for point cloud.

```
PREFIX scql_properties: <http://askco.re/scql/properties#>
PREFIX scql_func: <http://askco.re/scql/functions#>
PREFIX geo: <http://www.opengis.net/ont/geosparql#>
SELECT ?pc_data WHERE {
    :object a <http://askco.re/scql/types#point_cloud> ;
            scql_properties:geometry ?pc_geometry ;
            scql_properties:density ?pc_density ;
            scql_properties:density_unit ?pc_density_unit .
    FILTER (scql_func:intersects(?pc_geometry,
            "POLYGON(...)"^geo:wktLiteral)
            && scql_func:convert(?pc_density, ?pc_density_unit,
                <http://askco.re/scql/units#pt_per_m_2>)> 20 ) }
```

SPARQL is designed for querying RDF triples. However, the AQ system presented in this paper requires a more general mechanism that includes queries for objects. A detailed specification of the relationship between the proposed scQL and SPARQL will be presented in future work.

An important point to emphasize is that the scQL language is tailored to situational awareness applications where applying SQL or SPARQL directly is challenging. scQL follows a mix of SQL and SPARQL conventions. It is defined as an "object" language, similar to SQL, where the properties and data types are defined using URI, similar to SPARQL. Using "object" is natural in situation awareness applications, where answers often refer to a specific object, for example, a point cloud or a building. Using URI allows for the grounding of

the query language in RDF ontologies. Additionally, scQL allows for adding mechanisms for reasoning about the relationship between querying for data and the execution of the exploration missions. For example, in future work, we plan on introducing time limits on query response times. Such a mechanism would allow for executions of queries that return the highest quality point cloud within specified time limits rather than defining a specific condition on the density.

5. CONCLUSIONS

In many applications, such as emergency rescue, human operators interface with teams of heterogeneous robotic agents to get things done in highly complex and uncertain situations. In particular, the first response in a disaster requires rapid construction of situation awareness about unfolding events. One of the first tasks is reconstruction of the operational environment by generating 3D models which can then be used for mission planning. This paper has proposed the use of an active query interface to complex multi-agent systems consisting of teams of heterogeneous robotic systems. Rather than manually and tediously setting up detailed missions, human operators can instead query a system, where the query is a higher-level declarative description of what is required to enhance situation awareness.

The active query system includes a rich language for specifying requirements, where the details as to how to meet those requirements are constructed through the interpretation and execution of the original active query, thus relieving human operators of this burden. In this process, it is often the case that additional multi-agent missions are generated to meet the original requirements. Descriptions of an active query language and engine for interpretation and execution of such queries have been provided together with associated algorithms. Additionally, it has been shown how to integrate the stand-alone active query mechanism with a complex underlying multi-agent system. It is important to emphasize that the active query system is independent of any particular multi-agent system or application. The paper has focused on using active queries to retrieve and generate 3D models of operational environments, although the mechanism is much more general than this specific use. This application has been used to field test the active query system using teams of human and UAV agents to validate the idea. Many experiments have been successfully completed and one representative scenario is described in this paper. Empirical evaluation of the scalability of the active query system is also provided.

The proposed AQ system assumes that only one variable is used per query. Hence, the query result is restricted to one data type and does not allow executing operations (“join”) between multiple variables. However, use cases exist in situational awareness applications where querying information using multiple variables is desirable. The AQ system will be extended in future work to support “join” operations at the scQL language and the executor levels. This will involve extending the syntax of the scQL and the AQE algorithm.

In summary, we believe this is a very powerful and useful mechanism to relieve human operators of the need to manually program detailed instructions for the use of teams of robots and their associated sensors and capabilities. In the future, we will focus on graphical interfaces to describe active queries and extend their use to other application areas. Additionally, human factors studies would be useful for additional validation of the approach and demonstrating additional efficacy in the field.

DECLARATIONS

Authors’ contributions

Made substantial contributions to the conception and design of the active query system: Berger C, Doherty P
Performed development of the main software prototype used in the field-test experimentation: Berger C

Contributed to iterations on all ideas and were responsible for field robotics experimentation: Rudol P, Wzorek M
Contributed to the writing of the article: Berger C, Doherty P, Rudol P, Wzorek M

Availability of data and materials
Not applicable.

Financial support and sponsorship
This work has been supported by the ELLIIT Network Organization for Information and Communication Technology, Sweden (Project B09), the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, and Sweden's Innovation Agency Vinnova (Project: Autonomous Search System, AuSSys, 2022-00086, 2023-01035).

Conflicts of interest
All authors declared that there are no conflicts of interest.

Ethical approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Copyright
© The Author(s) 2024.

REFERENCES

1. Doherty P, Berger C, Rudol P, Wzorek M. Hastily formed knowledge networks and distributed situation awareness for collaborative robotics. *Auton Intell Syst* 2021;1:16. DOI
2. Doherty P, Heintz F, Kvarnström J. High-level mission specification and planning for collaborative unmanned aircraft systems using delegation. *Unmanned Syst* 2013;1:75-119. DOI
3. Turbak F, Gifford D, Sheldon MA, Sussman J. Design concepts in programming languages. MIT press; 2008. Available from: https://mitpress.mit.edu/9780262201759/design-concepts-in-programming-languages/. [Last accessed on 13 March 2024]
4. Prud'hommeaux E, Seaborne A. SPARQL query language for RDF. W3C Recommendation; 2008. Available from: http://www.w3.org/TR/rdf-sparql-query/. [Last accessed on 13 March 2024]
5. Gandon F, Schreiber G. RDF 1.1 XML syntax. Cambridge, Massachusetts, USA: World Wide Web Consortium (W3C); 2014. Available from: https://www.w3.org/TR/rdf-syntax-grammar/. [Last accessed on 13 March 2024]
6. BS ISO/IEC 9075-2:2023. Information technology. Database languages SQL - Foundation (SQL/Foundation). DOI
7. BS ISO/IEC 13249-3:2016 Information technology. Database languages. SQL multimedia and application packages - Spatial. DOI
8. Quigley M, Gerkey B, Conley K, et al. ROS: an open-source Robot Operating System. Available from: http://robotics.stanford.edu/~ang/papers/icraoss09-ROS.pdf. [Last accessed on 13 March 2024]
9. Antonyshyn L, Silveira J, Givigi S, Marshall J. Multiple mobile robot task and motion planning: a survey. *ACM Comput Surv* 2023;55:1-35. DOI
10. Tubaishat M, Madria S. Sensor networks: an overview. *IEEE Potentials* 2003;22:20-3. DOI
11. Levis P, Gay D, Culler D. Active sensor networks. Available from: https://www.usenix.org/legacy/events/nsdi05/tech/full_papers/levis/levis.pdf. [Last accessed on 13 March 2024]
12. Zeng R, Wen Y, Zhao W, Liu YJ. View planning in robot active vision: a survey of systems, algorithms, and applications. *Comp Visual Media* 2020;6:225-45. DOI
13. Chien S, Davies A, Tran D, et al. Using automated planning for sensorweb response. 2004. Available from: https://dataverse.jpl.nasa.gov/dataset.xhtml?persistentId=hdl:2014/39138. [Last accessed on 13 March 2024]
14. Wang P, Reinelt G, Gao P, Tan Y. A model, a heuristic and a decision support system to solve the scheduling problem of an earth observing satellite constellation. *Comput Ind Eng* 2011;61:322-35. DOI
15. Doherty P, Kvarnström J, Rudol P, et al. A collaborative framework for 3d mapping using unmanned aerial vehicles. In: Baldoni M, Chopra A, Son T, Hirayama K, Torroni P, editors. PRIMA 2016: Principles and Practice of Multi-Agent Systems. Cham: Springer; 2016. pp. 110-30. DOI
16. Calbimonte JP, Jeung H, Corcho O, Aberer K. Enabling query technologies for the semantic sensor web. *Int J Semant Web Inf Syst*

2012;8:43-63. DOI

17. Sadagopan N, Krishnamachari B, Helmy A. Active query forwarding in sensor networks. *Ad Hoc Netw* 2005;3:91-113. DOI
18. Yu FX, Ji R, Chang SF. Active query sensing for mobile location search. In: Proceedings of the 19th ACM International Conference on Multimedia. New York, NY, USA: Association for Computing Machinery; 2011. pp. 3–12. DOI
19. Cai G, Zhang J, Jiang X, et al. Ask&Confirm: active detail enriching for cross-modal retrieval with partial query. In: 2021 IEEE/CVF International Conference on Computer Vision (ICCV). Montreal, QC, Canada: IEEE; 2021. pp. 1815-24. DOI
20. Huang SJ, Zhou ZH. Active query driven by uncertainty and diversity for incremental multi-label learning. In: 2013 IEEE 13th International Conference on Data Mining. Dallas, TX, USA: IEEE; 2013. pp. 1079-84. DOI
21. Tuia D, Muñoz-Marí J, Camps-Valls G. Remote sensing image segmentation by active queries. *Pattern Recognit* 2012;45:2180-92. DOI
22. Burstein M, Ferguson G, Allen J. Integrating agent-based mixed-initiative control with an existing multi-agent planning system. In: Proceedings Fourth International Conference on MultiAgent Systems. Boston, MA, USA: IEEE; 2000. pp. 389-90. DOI
23. Motro A. Intensional answers to database queries. *IEEE Trans Knowl Data Eng* 1994;6:444-54. DOI
24. Minker J. An overview of cooperative answering in databases. In: Andreasen T, Christiansen H, Larsen HL, editors. FQAS 1998: Flexible query answering systems. Berlin, Heidelberg: Springer; 1998. pp. 282-85. DOI
25. Scherl RB, Levesque HJ. Knowledge, action, and the frame problem. *Artif Intell* 2003;144:1-39. DOI