SLAMinDB: Centralized graph databases for mobile robotics

Dehann Fourie*, Samuel Claassens*, Sudeep Pillai, Roxana Mata, and John Leonard (*Fellow, IEEE*)

***Abstract*— Robotic systems typically require memory recall mechanisms for a variety of tasks including localization, mapping, planning, visualization etc. We argue for a novel memory recall framework that enables more complex inference schemas by separating the computation from its associated data. In this work we propose a shared, centralized data persistence layer that maintains an ensemble of online, situationally-aware robot states. This is realized through a queryable graph-database with an accompanying key-value store for larger data. In turn, this approach is scalable and enables a multitude of capabilities such as experience-based learning and long-term autonomy. Using multi-modal simultaneous localization and mapping and a few example use-cases, we demonstrate the versatility and extensible nature that centralized persistence and *SLAMinDB* can provide. In order to support the notion of life-long autonomy, we envision robots to be endowed with such a persistence model, enabling them to revisit previous experiences and improve upon their existing task-specific capabilities.**

I. Introduction

Realizing tangible value from robotic data requires a versatile and highly-accessible data representation. Novel database technologies provide advantages in representation, manipulation, and extraction. We argue the benefits of graph databases in robotics by exploring spatio-temporal representation and centralized persistence. This enables situationally-aware querying and inference of the robot's task-specific state at any point in its history. We view this as a critical component in the robot's ability to learn newer representations from previous experiences.

We propose a two-tier persistence architecture as depicted in Fig. 1 that maintains independent databases for: (i) A graph that provides both the master data index as well a store for relational, queryable data (e.g. the robot's Maximum-A-Posteriori state estimate); (ii) A key-value store retaining arbitrarily large sensor data (e.g. RGB and Depth (RGB-D) imagery and laser scans).

Through potential use-cases and experiments, we emphasize the flexibility of graph databases for a multitude of concurrent tasks: (i) Post-hoc loop closure detection and incorporation; (ii) Continuous factor-graph solving for simultaneous localization and mapping; (iii) Incorporation of post-processed information such as per-pixel semantic segmentation of individual keyframes; (iv) Querying relevant scene entities as shown in Fig. 5.

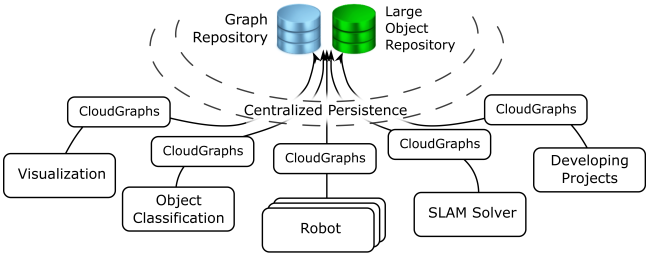

Fig. 1. Conceptual description of a navigation graph centralized robot data persistence and recall system. The graph database provides an efficient spatio-temporal search index while the larger sensor data is retained in the key store.

Next we present three major themes relating to a central navigation and data store system. We showcase three data queries which illustrate the benefits of a central database system in robotics, followed by related work and limitations. We present an implementation discussion and experimental results using a Turtlebot robot [1].

II. Principal Themes

Our approach emphasizes three core requirements in a robotics system: *Random access via query* relating to interaction with collected data in unpredictable ways; A *Centralized architecture* relating to in-situ interaction with the data from multiple, weakly independent processes; and *Horizontal scalability* relating to offloading large computation in real-time.

A. Random Access Queries

Insertion, modification, and extraction from any persistence framework should be as efficient as possible. The underlying complexity of this requirement is that an architect has to address the variety of different roles. We are proposing that a pose-keyed graph fulfils a good balance of ease-of-access with insertion speed. The natural structure of the graph also allows custom keying to both index and retrieve specialized datasets from the collective data (such as keying data by tracked features).

*Equal contributors, dehann@csail.mit.edu is with the Joint Program at Massachusetts Institute of Technology and Woods Hole Oceanographic Institution; spillai@csail.mit.edu, {rmata, jleonard}@mit.edu are with Massachusetts Institute of Technology, Cambridge, MA, 02139, USA; sclaassens.ad@gmail.com is a senior software engineer with General Electric.

This work was partially supported by the Office of Naval Research under grants N00014-11-1-0688 and N00014-13-1-0588 and by the National Science Foundation under grant IIS-1318392, which we gratefully acknowledge.

Three example queries are provided in Section V to demonstrate how the graph database enables random-access queries.

B. Centralized Architecture

Local processing of sensory data reduces the need for large-bandwidth communication systems, however it also isolates the data and places large demands on the processing power encapsulated in the robot. This is relevant in exploratory robotics, where data volumes, bandwidth and latency are fundamental concerns during design. Domestic and urban environments, however, offer greater communications bandwidth but large scale robotic processing power remains expensive.

The question becomes one of balancing local robot processing with large scale centralized processing. In these settings, centralizing the data would allow: (i) long-term data accessibility by any number of agents; (ii) minimization of local, client-side processing; (iii) sharing of data and analysis between different sessions and robots; (iv) aggregation and refinement of the collective data (e.g. summary maps).

C. Horizontal Scalability and Concurrency

Requiring horizontal scalability enforces that the underlying data structure may be processed by multiple independent applications. We argue that a horizontally scalable system is advantageous as we are not constraining the system to a single application. Rather, new agents may be introduced at will, and the underlying structure should support operation by any number of concurrent applications.

III. Related Work

Automation of a robot in unpredictable environments requires a map, or contextual reference, to be generated from sensor data in a automated manner. A graphical depiction (or factor graph [2]) is a good representation for collecting and inferring variables of interest over all the data [3]. A variety of factor graph based Simultaneous Localization and Mapping (SLAM) solutions exist, in particular [4], [5], which utilize parametric representations of the sensor readings and the world.

Our work on a database-centered approach to SLAM follows from, among others, three major requirements on SLAM system. Firstly, we do not know ahead of time which data or situations may be of interest. As a result, robot systems capabilities are limited in our ability to utilize and infer traits, patterns, trends or things from gathered sensor data. For example, automated discovery of physical objects can be improved by leveraging optimized spatial estimates for a series of images [6] rather than treating spatial and image information separately.

Secondly, our abilities are constrained by available computational power, ultimately resulting in less robust 'artificial intelligence'. This work is partly motivated by ongoing research into non-parametric solutions for SLAM [7] which requires higher computational load, but offers more robust solutions than previous methods. In general, the method's ability to work with and interact with data should not be hindered by the physical location of the captured data, or by some complicated access process to recover a specific seemingly random piece of data.

Thirdly, large volumes of data and robust inference techniques require larger computation resources, such as multi-processor architectures, which in turn places large concurrency demands on the design of a robotic computational system. Previous work by Newman et al. [8] used a database and querying language, in the context of SLAM, as an aid to finding loop closures from robot data. Consider an even bigger dataset [9] where common locations are searched by working through large amounts of camera image data from a car driving multiple kilometers. The requirement for efficient storage and concurrent computation becomes critical as long-term, multi-vehicle solutions are considered [10].

An objective assessment of SLAM solutions suggest that many applications do not necessarily require a high speed SLAM solution to achieve low latency navigation. Concurrent smoothing and filtering work by [11] makes deeper assessment on how fast a large SLAM solution needs to be, and finds a robust, elegant analytical method to separate the low latency estimate from the large data fusion (or SLAM) process. This line of reasoning allows us to investigate slower, but much more versatile data management strategies, such as a database-centric approach.

Database systems have traditionally been used for storing large amounts of transactional data, in which tabulated data mostly adheres to a set relational structure. Table and relations are designed beforehand, so that an operational system can add new rows and manipulate values in existing columns.

Relational databases perform well for a variety of applications [12], but are limited when the connections between data are rapidly changing; a dominant feature in SLAM systems being used and developed in the robotics community today.

Recent work by Nelson et al. [13] strongly argues for database driven long-term storage and data retrieval in robotics. Such data accumulation systems show performance benefits when using a database for searching over temporal and spatial cues, or integrating the back-end navigation solution to a database server [14], [10].

However, past work has focused on using a relational database structure as the primary mechanism of data storage, together with SQL queries for data retrieval. The required Entity-Relationship model for designing such databases assumes a specific mechanism of mapping and localization acting on the raw data, a mechanism which might not satisfy the interface needs of a new technique [15]. Moreover, these database methods

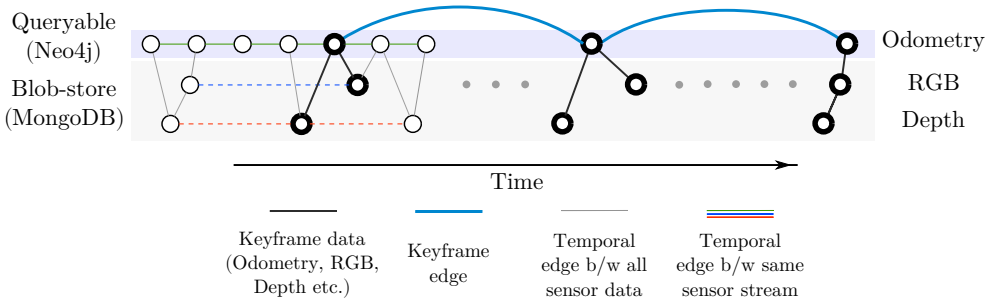

Fig. 2. An illustration of the two-tier architecture that is maintained and stored in a graph-based data layer with nodes represented by the keyframes (consisting of RGB and Depth imagery from the Turtlebot) and edges (in blue: connecting the keyframes via odometry estimates from the wheel encoders on the Turtlebot). The SLAM estimates are stored in a lightweight and queryable Neo4j layer, while the large sensory information such as RGB and Depth imagery corresponding to the associated keyframes are stored in a MongoDB key-value store with simple key-based lookup from the Neo4j layer.

require validity checking for each new datum and are not able to integrate new data into the map online [10]. This method doesn't scale because of the necessary re-incorporation of all data for each query of metric computation, thus precluding online augmentation of the map [15].

IV. Graph Databases for Robotics

We argue that relational databases are a critical step forward as they provide: (i) a rich scalable data structure where robots and task-specific processing are not required to store the complete graph in-memory; (ii) centralization of the data persistence layer so interacting-processors may operate concurrently on the shared data; (iii) a relational query language for powerful data extraction where only the relevant data is returned to the client.

However, using a relational database for data storage has drawbacks, as: (i) it requires the graph be represented as tables and tabular relationships, which causes unnecessary, constant translation from graph to tabular structure; (ii) it limits flexibility as the relational structure must be defined beforehand and is difficult to modify in situ; (iii) large sensory data will bloat the database with binary blob elements, negatively affecting the overall database performance.

Classic relational databases represent data and relationships using a tabular structure. Similarly, graph databases represent data and relationships using a native graph datastructure. Unlike statically typed tables in relational databases, graph databases allow for richer data types and complex relationships [16]. Accompanying the richer elements is a query language that can interrogate and traverse the data structure.

We propose a native graph database and two-tier storage structure to offload large sensory data to a NoSQL key-value store. NoSQL key-value stores are designed for efficient insertion and extraction, which is highly applicable for persisting the larger sensory data. This is done to realize a 'starved graph', i.e. a graph that can be efficiently leveraged for querying often-used data and relationships, with the ability to extract the larger binary on demand (image and sensor data). The keys are stored in the graph, providing a link between the two systems. Users of the persistence system choose which data should be included in the graph and which should be offloaded, and this can change dynamically. This architecture allows additional systems and data to be bound to the graph without bloating the structure.

Prior to discussing implementation or experimental results, we wish to highlight the advantages of the centralized robot graph database with a few working examples. Consider that in all cases the query execution is performed on the server-side, resulting in a small but relevant fraction of the complete dataset being transferred to the client.

V. Working Examples

Three example queries are chosen to illustrate how the structure can simplify otherwise complicated random access queries. We use Cypher syntax, a declarative graph database language that is native to Neo4j [17].

A. Temporal Queries

Computing the start and end positions from a ROS bag [18] or datafile involves scanning the complete dataset. The *SLAMinDB* graph representation allows a succinct query to perform server-side searches from indexed properties (such as timestamp). Given that `n.timestamp` is the timestamp parameter and `n.SLAM_Estimate` we wish to extract:

Listing 1 Retrieving the latest refined pose estimate

```
MATCH n:POSE
WHERE n.timestamp = max(n.timestamp)
AND exists(n.SLAM_Estimate)
RETURN n.label, n.SLAM_Estimate
```

The solver is computing the SLAM solution concurrently—captured by the `exists(n.SLAM_Estimate)`—if this exists then we have a valid SLAM estimate. Omitting it would retrieve the latest raw pose in the SLAMinDB implementation. As

the solution is calculated in-situ, the simple conditional creates an index-searched operation that is arguably difficult with flat file datasets.

Similarly we can retrieve the start location of the refined graph by simply changing our search criterion:

Listing 2 Retrieving the initial refined pose estimate

```
MATCH n:POSE
WHERE n.timestamp = min(n.timestamp)
AND exists(n.SLAM_Estimate)
RETURN n.label, n.SLAM_Estimate
```

In multi-agent scenarios, the graph retains the complete history of all agents. Queries can be constructed that relate information of interest (such as identified objects, proximity, or time) to the cumulative history of the robots. Additional indexing allows the results to be efficiently extracted. An example of such an extraction would be retrieval of the latest pose (`n.SLAM_Estimate`) and sensor data (`n.bigData`) for `ROBOT1` during its fifth session, `SESSION5`:

Listing 3 Retrieving the latest refined pose estimate for ROBOT1 during SESSION5 run

```
MATCH (n:POSE:ROBOT1:SESSION5)
AND n.timestamp = max(n.timestamp)
AND exists(n.SLAM_Estimate)
RETURN n.label, n.SLAM_Estimate, n.bigData
```

B. "Foveation" and Spatial Queries

In addition to temporal queries, the graph can leverage position as a filter. A useful example of such a query would be to extract all nodes within a vicinity and within view. This is referred to as a foveate query and can be implemented by modifying the WHERE clause. Additionally we are making use of two server-side user-defined functions which augment the query language with SLAMinDB-specific functionality. The following query will return all factor graph nodes with a [2,5] meter range and within a 45^o field-of-view:

Listing 4 Foveation calculation as Cypher query

```
WITH [0, 0] as position, 3.14159/4.0 as fov
MATCH (n)
WHERE
// Region filtering
       cg.withinDist2D(n, position, 2, 5)
AND
// Frustum cutoff filtering
       cg.withinFOV2D(n, position, fov)
RETURN n
```

User-defined functions and procedures can address scenarios where often-used queries should be encapsulated, or in cases where procedural steps are required. For reference, we have included a simplified form of the `cg.withinFOV2D` function:

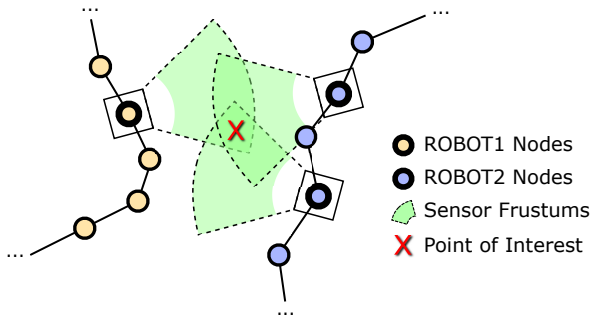

Fig. 3. Illustration of foveation query and node selection for multiple robots in the centralized SLAM-aware database.

Listing 5 Simplified form of the cg.withinFOV2D user-defined function in Java

```
@UserFunction
public boolean withinFOV2D(
  @Name("node") Node node,
  @Name("position") List<Double> pos,
  @Name("fov") double fovRad
) {
  double[] pose = (double[])node.getProperty("MAP_est");
  double poseAng = pose [2];
  //Calc pose-forward and pose-to-position vectors
  SimpleMatrix
    pose2POI = toVec(pos.get(0)-pose[0],pos.get(1)-pose[1]),
    poseFor = toVec(Math.cos(poseAng),Math.sin(poseAng));
  pose2POI = pose2POI.divide(pose2POI.normF());
  //Use dot product to determine if within FOV
  return Math.acos(pose2POI.dot(poseFor)) <= fovRad;
}
```

User-defined procedures allow more comprehensive code to be embedded server-side and if (as in the case above) we can optimize the search on the server, the foveate query can be succinctly expressed as:

Listing 6 Foveation calculation as user-defined procedure

```
CALL cg.foveate2D([0, 0], 2, 5, 3.14159/4.0)
```

C. Interactive SLAM

Graph relationships provide rich functionality for generating and traversing elements. Consider two concurrent processes in SLAMinDB: one processing the sensor data and suggesting potential loop closures, and the second processing a parallel solver consuming the changes when suitable. The two agents can be on different systems, operating with minimal interaction as fully decoupled applications.

The production of the loop closures could, for example, be a supervised application. In the event that a loop closure is confirmed, an edge can be introduced to indicate to the solver that it has new relationships to process. This can be done in the graph with the following update query:

Listing 7 Introducing edges in the case of a loop closure

```
MATCH (n:LANDMARK), (m:LANDMARK)
WHERE n.label="l10" and m.label="l84"
SET (n)-[r:SAMELANDMARK]-(m)
```

When the SLAM solver is in a suitable state to integrate the new relationships, it can consume the unique edges and produce the correct function nodes:

Listing 8 Solver retrieval of edges during client-side graph update

```
MATCH (n)-[r:SAMELANDMARK]-(m)
RETURN n.label, m.label, r
```

The refined graph is updated in place, resulting in incremental improvement of the dataset without processing off-line or requiring complex handshaking.

VI. Implementation

The experimental implementation makes use of a Neo4j graph database [17] for the graph persistence and a MongoDB NoSQL database [19] for the key-value store[1] as shown in Fig. 2. The codebase was developed principally in Julia [20] with the robot front-end and image recognition in Python [2]. The integration of both the Multi-Modal iSAM solver (Caesar.jl) [7] and the underlying data persistence layer (CloudGraphs.jl) is referred to as SLAMinDB.

On insertion, data partitioned into four categories: (i) Labels for indexed searching; (ii) Properties for searching and filtering; (iii) Packed data for local vertex storage; (iv) Large object storage for MongoDB storage.

Labels and properties can be used to build relational queries and traverse the graph in task-specific ways. Vertices, for example, can be labelled when AprilTags [21] are detected, and the properties can contain more detail about the specific tag. The packed data permits critical binary data to be stored in the vertices - a compromise between the large data store and simple properties. Custom labelled edges can also be inserted, developers to insert task-specific traversals which augment the existing factor graph.

Large sensor data is trimmed from the vertices and persisted in MongoDB. The vertices are appended with the MongoDB keys to maintain the relationship between the two stores. The separation is opaque to graph consumers, which operate via the CloudGraphs API, splitting and re-splicing as the data as required.

VII. Experiments

We demonstrate the key elements discussed above through a concrete example of robot navigation and mapping. This choice led us to develop a SLAM-style factor graph solver directly at the database-level, admitting concurrent access, search, visualization and computation.

A. Illustrating Concurrency and Random Access

We are also particularly interested in the ability to offload computation from the robot, whereby more post-hoc / in-situ agents, or human operators, can interact with the data in a rich manner.

In our experiments, we used a Turtlebot outfitted with a RGB-D structured light camera and demonstrate tangible experimental results relating to the three main themes discussed in Section II: Random query access, centralized and atomic transactional structure, and horizontal scalability. The robot is tele-operated in an AprilTag-laden office environment for benchmarking purposes. Through an interactive procedure, we incorporate cross-session navigational loop closures—via AprilTags—as constraints to the SLAM factor-graphs from each session.

Figure 4 shows a visualization of the merged map reconstructed from the Turtlebot data. The local processes, designed to be light-weight, communicates relevant measurements back to the central database for SLAM solver consumption. Typically, the central database is hosted on a more powerful server computer, while the the robot connects to it through a low-bandwidth network. The local robot processes incrementally pushes new marginalized measurements to the centralized database, all while the database consumes and solves with these added constraints independent of each other. Larger key-value entries such as raw color and depth imagery from the RGB-D sensor are stored in a local MongoDB instance. These entries are synchronized with the server MongoDB data store at lower priority as permitted by network availability. Additionally, we generate globally unique identifiers at the initial commit to the local store, which will become globally available as the network availability allows.

At any point in time, the robot may choose to run its own queries against the central database. For example, the latest available Maximum-A-Posteriori (MAP) location estimate (independently computed by the SLAM agent) can be required using the query presented in Section V-A. Depending on network traffic and length of network between the robot and server, this query nominally takes on the order of tens to hundreds of milliseconds to run. We'd like to emphasize that the query sent and returning result are only small single line text strings, with most computation happening on the central server.

The server computes and then returns the latest available SLAM pose estimate. The ability to make such queries has made an otherwise complicated processes remarkably simple. The robot may now incorporate these return results, registered against a previously

[1]The code for the experiment is available at https://github.com/dehann/Caesar.jl and https://github.com/GearsAD/CloudGraphs.jl

[2] Pybot is available at https://github.com/spillai/pybot

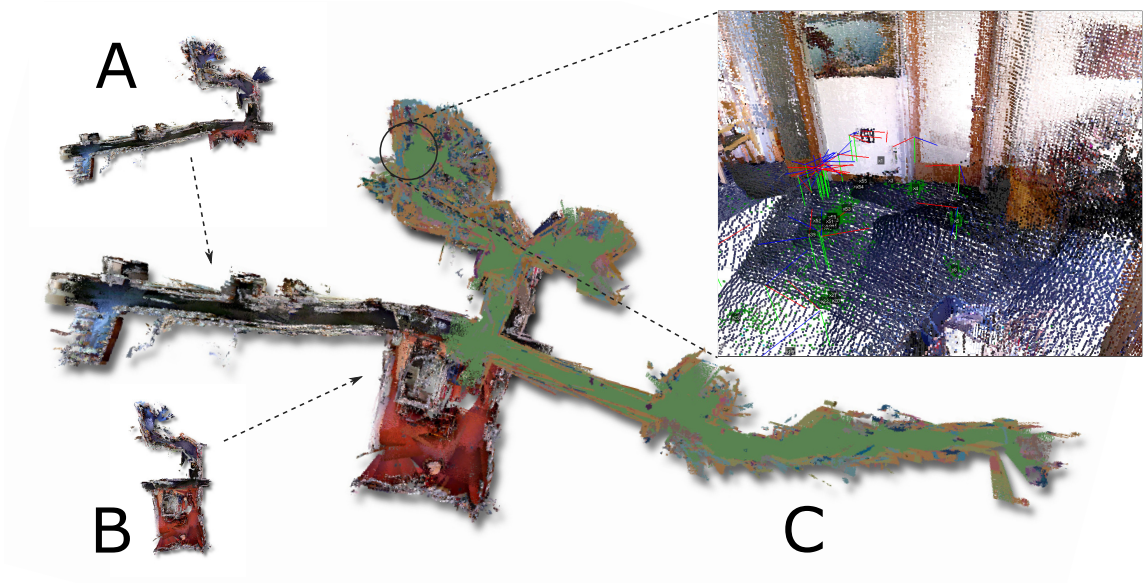

Fig. 4. Composite 3D reconstruction by a "decoupled" visualization process (as depicted in Fig. 1) from three Turtlebot trajectory sessions; where individual keyframe structured-light point clouds are projected from multi-modal SLAM [7] optimized trajectories via the graph database. The SLAM solution was computed on a server while the robot produced data. A database query similar to Listing 1 would all low bandwidth transport of latest unique pose SLAM estimates to the robot. Loop closure constraints internal to and across sessions are affected by visual AprilTags sparsely placed in the scene. The session to the right has color labels given by SegNet segmentation [22], showing floors, walls, objects and more. The top-right image shows a close up area with two doors and a painting in RGB color palette directly from recorded key frame images.

known pose ID to improve its own location estimate. A process not entirely dissimilar from how the Apollo spacecraft navigated to the moon and back, resetting the spacecraft's local estimates using Earth-based radio navigation and computation.

B. Illustrating Third Party Interactions

Now lets focus on operations done elsewhere in the system. Dedicated, larger computational tasks may very well be placed near or on the same server computers. The SLAM solution is one key example. We use a powerful non-parametric SLAM solver [7] to query, infer and insert current best Maximum-a-Posteriori estimates back into respective vertices of the database. The estimates provided by the SLAM service therefore become available for global consumption via the database.

In Fig. 4, we see pose-slaved point clouds from three Turtlebot robot trajectories. These multi-session maps contain internal loop closures, and thanks to the common factor graph persistence, can also share information among sessions. By using sightings of AprilTags and current best estimates of robot keyframe pose locations, we can manually introduce new relationships in the graph to affect cross-session loop closures.

As a further interaction example, we can easily retrieve two or three RGB keyframe images from the large key store corresponding to poses of interest, at which point we could confirm or ignore that loop closure would be inserted.

We are able to make cross-session modifications, also discussed earlier in Section V-C, with a single query. These modifications can occur as the SLAM service is solving the factor graph, while the robot is simultaneously injecting more information, and even more processes are affecting in-session loop closures. Concurrency fields in the database vertices allow simultaneous and atomic changes to the shared graph structure.

Furthermore, since the database acts as a central store of data, it becomes fairly simple to attach other batch processes. In our case, we use SegNet [22] of all keyframe images for image segmentation. We also insert the segmentation output image into the Mongo key store with a unique identifier, co-registered in the thin graph database.

The visualization is another process, as depicted earlier in Fig. 1, which attaches to the database and recovers relevant information. Pose state, SLAM solver values of interest, depth point cloud, RGB & segmentation color palette for each pose can then be be visualized as required. Again, this process is greatly simplified by the ability to query over all the information and only retrieve (transport) the bits relevant to that specific user query.

The visualizations in Fig. 4 shows three robot trajectory sessions. The right-hand side segment shows one session's point cloud with segmentation color palette for floors, walls, and objects. We can now use the common data persistence to enhance training on the segmentation system, leveraging the SLAM-aware information available in the database.

Fig. 5. Foveation + Pixel-wise segmentation (SegNet).: Specify 3 out of the 5 queried nodes. Cropped and color modified keyframe SLAM-aware images, which were recovered through the *foveate* query (Section V-B), to ease the detection of a common object in the world. This test case does not yet use the common sighting of a painting in the SLAM solution. An independent process may now manipulate the persistent database centralized factor graph to introduce new loop closure constraints based on object detections of the painting.

C. Specialized Queries

A centralized graph database system can help us develop *dreaming* capabilities by handling the low-level data indexing and searching. For our final illustration we will focus on the idea of foveation, as shown in Fig. 3, where we use SLAM inferred (or optimized) keyframe pose geometry as meta information to aid our search. The geometry enables efficient server-side spatial searches over potentially large volumes of sensory data.

Using the query described in Section V-B, we inspect poses on the database which might see a specific point in space. Querying poses of interest, we can then retrieve the larger data, i.e. RGB, depth or segmented images, from the Mongo key store.

As a further step, we now superimpose the SegNet *artwork* labels for images shown in Fig. 5 and can review such sightings: either to include new loop closure measurements for the SLAM solution, or improve training data for segmentation of *artwork*.

VIII. Conclusion

This paper demonstrates the applicability of a centralized factor-graph for data persistence. By leveraging the natural structure of the factor graph as the principal data representation in a robotic mobile system we realize a model that supports: (i) Random access for data; (ii) Centralization of data; (iii) Horizontal scalability. There are several advantages of relating most robotic data to a single versatile index, which are demonstrated in the examples. While centralizing the factor graph in such a manner may at first not seem an optimal choice, we show that the advantages of using dedicated data layer technologies far outweigh the marginal increase in complexity to existing robotic navigation and recall systems, and that this is superior to local, independently operating solutions, especially with regard to cooperative agents.

References

[1] Open Source Robotics Foundation Inc., "Turtlebot 2: Open-source development kit for apps on wheels." 2016. [Online]. Available: http://www.turtlebot.com/

[2] F. R. Kschischang, B. J. Frey, and H.-A. Loeliger, "Factor graphs and the sum-product algorithm," *IEEE Transactions on information theory*, vol. 47, no. 2, pp. 498–519, 2001.

[3] T. Bailey and H. Durrant-Whyte, "Simultaneous localization and mapping (SLAM): Part ii," *IEEE Robotics & Automation Magazine*, vol. 13, no. 3, pp. 108–117, 2006.

[4] M. Kaess, A. Ranganathan, and F. Dellaert, "iSAM: Incremental smoothing and mapping," *IEEE Transactions on Robotics*, vol. 24, no. 6, pp. 1365–1378, 2008.

[5] M. Kaess, H. Johannsson, R. Roberts, V. Ila, J. J. Leonard, and F. Dellaert, "iSAM2: Incremental smoothing and mapping using the bayes tree," *The International Journal of Robotics Research*, p. 0278364911430419, 2011.

[6] S. Pillai and J. Leonard, "Monocular SLAM supported object recognition," in *Proceedings of Robotics: Science and Systems (RSS)*, Rome, Italy, July 2015.

[7] D. Fourie, J. Leonard, and M. Kaess, "A nonparametric belief solution to the Bayes tree," in *IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems, IROS*, Daejeon, Korea, Oct 2016, to appear.

[8] P. Newman and K. Ho, "SLAM-loop closing with visually salient features," in *proceedings of the 2005 IEEE International Conference on Robotics and Automation*. IEEE, 2005, pp. 635–642.

[9] M. Cummins and P. Newman, "Appearance-only SLAM at large scale with fab-map 2.0," *The International Journal of Robotics Research*, vol. 30, no. 9, pp. 1100–1123, 2011.

[10] P. Mühlfellner, P. Furgale, W. Derendarz, and R. Philippsen, "Designing a relational database for long-term visual mapping," 2015.

[11] M. Kaess, S. Williams, V. Indelman, R. Roberts, J. J. Leonard, and F. Dellaert, "Concurrent filtering and smoothing," in *Information Fusion (FUSION), 2012 15th International Conference on*. IEEE, 2012, pp. 1300–1307.

[12] J. J. Miller, "Graph database applications and concepts with neo4j," in *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA*, vol. 2324, 2013.

[13] P. Nelson, C. Linegar, and P. Newman, "Building, curating, and querying large-scale data repositories for field robotics applications," in *Field and Service Robotics*. Springer, 2016, pp. 517–531.

[14] W. Churchill and P. Newman, "Experience-based navigation for long-term localisation," *The International Journal of Robotics Research*, vol. 32, no. 14, pp. 1645–1661, 2013.

[15] P. Mühlfellner, M. Bürki, M. Bosse, W. Derendarz, R. Philippsen, and P. Furgale, "Summary maps for lifelong visual localization," *Journal of Field Robotics*, 2015.

[16] "Apache Tinkerpop: The benefits of graph computing," http://tinkerpop.apache.org/, accessed: 2017-02-20.

[17] "Neo4j: The world's leading graph database," https://neo4j.com/, accessed: 2017-02-20.

[18] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "ROS: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3. Kobe, Japan, 2009, p. 5.

[19] "MongoDB: For giant ideas," https://www.mongodb.com/, accessed: 2017-02.

[20] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *arXiv preprint arXiv:1411.1607*, 2014.

[21] E. Olson, "Apriltag: A robust and flexible visual fiducial system," in *Robotics and Automation (ICRA), 2011 IEEE International Conference on*. IEEE, 2011, pp. 3400–3407.

[22] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: A deep convolutional encoder-decoder architecture for image segmentation," *arXiv preprint arXiv:1511.00561*, 2015.