Building, Curating, and Querying Large-scale Data Repositories for Field Robotics Applications

Peter Nelson, Chris Linegar and Paul Newman

Abstract Field robotics applications have some unique and unusual data requirements – the curating, organisation and management of which are often overlooked. An emerging theme is the use of large corpora of spatiotemporally indexed sensor data which must be searched and leveraged both offline and online. Increasingly we build systems that must never stop learning. Every sortie requires swift, intelligent read-access to gigabytes of memories and the ability to augment the totality of stored experiences by writing new memories. This however leads to vast quantities of data which quickly become unmanageable, especially when we want to find what is relevant to our needs. The current paradigm of collecting data for specific purposes and storing them in ad-hoc ways will not scale to meet this challenge. In this paper we present the design and implementation of a data management framework that is capable of dealing with large datasets and provides functionality required by many offline and online robotics applications. We systematically identify the data requirements of these applications and design a relational database that is capable of meeting their demands. We describe and demonstrate how we use the system to manage over 50TB of data collected over a period of 4 years.

1 Introduction

Lifelong learning for robotic systems requires large quantities of data to be collected and stored over long periods of time. As these data accumulate, they become increasingly difficult to manage and query. Without a scalable system in place, finding useful data becomes ever more complex and this undermines our goal of achieving long-term autonomy.

Many publications focus on the mechanics of lifelong autonomy but very few explicitly deal with the problem of storing and accessing the required data in a way

Peter Nelson, Chris Linegar, Paul Newman
Mobile Robotics Group, Department of Engineering Science, University of Oxford, UK
e-mail: {peterdn,chrisl,pnewman}@robots.ox.ac.uk

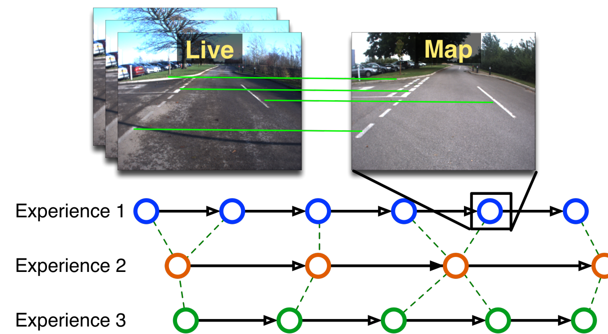

Fig. 1 Experience Based Navigation (EBN) [1] is a state-of-the art visual navigation framework for autonomous robots. Maps, in the form of 'experiences' with associated graph structure, camera images, and landmarks accumulate over many years. This paper aims to answer the question: 'how do we store and retrieve this kind of data in a flexible and efficient way?'

designed to aid long-term, large-scale mobile autonomy. Given the need for field roboticists to build coherent systems, it is time for this subject to be addressed.

Mobile robotics applications have some unusual data needs that cannot always be anticipated in advance. An example is illustrative. We often want to evaluate the efficacy of a new feature detector for visual odometry and thus testing under differing lighting conditions is vital. What we actually want to do is automatically collate image sequences that satisfy complicated compound queries such as 'find sequences $>$ 50m of stereo images, captured while driving into the sun over wet ground in the absense of dynamic obstacles'. This should run over all data ever collected and return a pristine dataset as if this had been the sole purpose of our experimentation over the past 4 years.

As another example, we need images of traffic lights to train a new state-of-the-art traffic light detector. Instead of wasting time collecting a whole new set of data for this specific purpose, we should first look to our existing data. It is probably the case that we inadvertently have images of traffic lights from previous data collection missions, and therefore we would like the ability to search for them.

To aid in solving this problem, we have designed and implemented a relational database framework that is applicable to a wide range of robotics applications. A data and query model is presented which cleanly distinguishes between sensor data and user-defined metadata. This makes it trivial for a user to decorate the database with their own contributions, and makes those contributions accessible to other users in a consistent way. For example, if someone builds the aforementioned traffic light detector and runs it over 100,000 images, they are then able (and encouraged) to add those results to the database for others to use in the future.

Our framework not only makes offline batch processing tasks easier, but also supports the needs of online tasks, for example the storage and retrieval of maps used by a robot's navigation system at runtime. This massively reduces the overhead of implementing and testing new navigation systems as data back-ends do not need to be written from scratch. A motivating use case that demonstrates this is our Experi-

ence Based Navigation (EBN) system [1], a visual navigation framework designed to deal with vast maps that grow continuously over a robot's entire lifetime (see Figure 1). EBN utilises our database framework in order to fulfil these challenging data storage and retrieval demands.

In the following sections we present the design and implementation of this framework. We analyse the performance of the system and demonstrate its real-world use by EBN.

2 Related Work

In the context of lifelong learning for autonomous robots, very little work has so far addressed the problem of organising, maintaining, and utilising the vast amounts of data that will accumulate over long periods of time. Traditionally, (relatively) small datasets are collected for specific purposes and are often discarded or forgotten about once they have served their purpose.

Various datasets have been released to the community along with associated tools and documentation. Some of these have been widely referenced in other publications and serve as convenient benchmarks for comparing performances of related techniques. Examples include the New College Vision and Laser dataset [2] and the DARPA Urban Challenge dataset [3]. Although these resources are invaluable to the robotics and computer vision communities, they are also utterly static – the data are a snapshot of a single period in time and adding to them in a way that is accessible for others to use is difficult. Additionally, as these datasets are often formatted differently and require non-standard tools to use, it is cumbersome to mine data from several at once. To be useful in the context of lifelong learning, we wish to move away from the idea of disjoint, immutable datasets and towards a living, growing repository of everything we record and have ever recorded.

RoboEarth [4] is an ambitious project that proposes to solve some of these problems by building a 'World Wide Web' for robots. A distributed cloud database is used to store machine-readable semantic data about environments, objects, and actions. Generic robots can access this prior knowledge to help complete a task and can upload their own knowledge once they succeed. A modular software architecture enables generic actions (e.g. moving, grasping) to be realised on specific hardware. A subset of RoboEarth's vision is close in spirit to what we want to achieve, however it places more of an emphasis on the storage, retrieval, and reconciliation of knowledge required for high-level planning and reasoning tasks.

3 Requirements

Our own condition serves as motivation for what is to follow, and we suspect these requirements are not unique to us. Figure 2 shows how our data have accumulated exponentially over the past 4 years. As of March 2015 we have amassed over 50TB

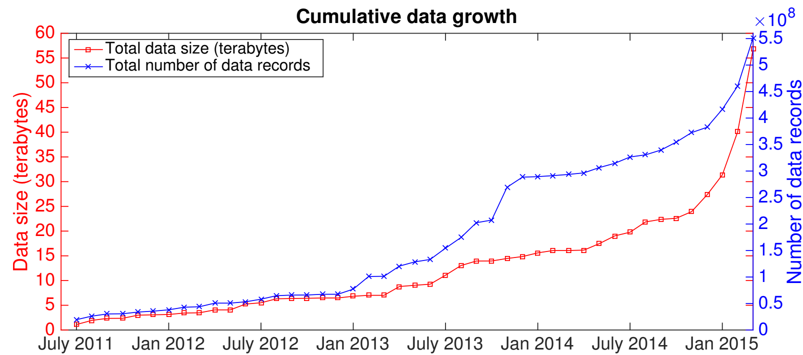

Fig. 2 Cumulative amount of sensor data we have amassed each month from July 2011 to March 2015. Shown in red is the total size of these data in terabytes and shown in blue is the approximate number of 'individual' data records (i.e. single camera images, GPS readings, laser scans).

of data, comprising of more than 500 million 'individual' records, and more are added on a daily basis. It is now intractable to manage all of this by hand.

Firstly there is the problem of reuse. In a previous time, data were collected for a specific purpose, used by one or two people, then forgotten about. Almost no semantic information about the data was stored and when it was, it was usually done so in a non-standard ad-hoc way (such as in notes on a wiki, or in a readme). Information and metadata (annotations) extracted from processed datasets suffered from the same problems. When gigabytes of data are collected and processed like this it becomes increasingly difficult to reuse what is considered useful and instead is easier to collect and process new data for each new purpose. To address this problem, we require that existing useful data can be found effectively and we therefore need a way to index it, as well as efficient ways to add new data when necessary. We also require a standard method for data annotation and require that these annotations can be easily traced to their underlying data.

Next, we have the problem of retrieval. Once we know a dataset contains relevant data, we must somehow retrieve them. For batch tasks, this would have been done by manually inspecting the dataset, chopping out the required parts and discarding the rest – a time-consuming and boring process. For online tasks, data would typically be accessed through a custom-written ad-hoc back-end, leading to bugs and time wasted on re-implementing common components.

Lastly, we have the problem of physical storage and organisation of data. Previously, datasets were organised only by directory structure and filename. This reliance on physical location presents many problems when datasets must be moved. Users should not need to care which machines their data physically reside on and so aim to build a modular system that is platform and tool agnostic. Users should be able to issue queries and access data using any programming language on any operating system. Ideally, queries should run online on robotic platforms with little or no modification.

4 Data and Query Models

Here we describe the main data and query models that underpin our framework. At the lowest level is *raw sensor data*: typically recorded directly from physical hardware and subjected to minimal, if any, processing. *Annotations* exist as a layer above this and are related directly to the data they annotate. Data that do not fall neatly into either of these two categories follow the standard relational model.

In many ways these categories are arbitrary and some data can fall into multiple categories depending on context. However, they help us identify common types of queries which we take advantage of to reduce complexity. Returning to our traffic light example: raw data include images captured by a camera during a data collection mission. Annotations include the output of a traffic light detector on these images (perhaps bounding boxes and labels). Other data include everything else that isn't directly causally related, for example an OpenStreetMap [5] road network map.

The following subsections make some use of relational algebra, a comprehensive introduction to which is available in [6].

4.1 Raw sensor data

Raw sensor data form the bedrock of our data model and many of our subsequent processing needs. As the collection of sensor data is such a fundamental part of our workflow, it seems appropriate that they be given special consideration.

Raw data are characterised by the fact that they represent part of the state of the world, as observed by some robotic platform, at a particular moment in time. A fundamental query we wish to support is to find the *entire* state of the world, as observed by a platform, at a particular moment in time. For example, our detector finds a traffic light in some camera image that was captured at time t. We therefore may want to find, from our collection of raw data, the corresponding GPS reading giving the location of the robot at time $t' \approx t$. This will be the first and most coarse step in many applications' processing pipelines.

Consider a robotic platform $r \in R$ representing an entity equipped with a set of sensors $S(r)$. The output datum of sensor $s^i \in S(r)$ at time t is denoted s^i_t. We assume that a single sensor can be mounted on more than one platform, though not at the same time. We also assume that, by definition and without loss of generality, sensors cannot produce multiple data at the same time (i.e. in cases such as stereo cameras, we treat both images as a single datum). Given these constraints, it can be seen that $\langle$platform, sensor, timestamp$\rangle$ tuples map to individual data via $\bar{\Omega}$:

$$s^i_{t'} \leftarrow \bar{\Omega}(r, s^i, t) \tag{1}$$

Where t' is the nearest[1] discretised timestamp for which a sensor reading exists. It follows that $\langle$platform, timestamp$\rangle$ tuples map to the state of the world as observed by the entire platform via Ω:

[1] The definition of 'nearest' may differ between queries but note that it is beyond the scope of the system to, for example, interpolate between records or verify annotation correctness – how that is handled is up to individual client applications.

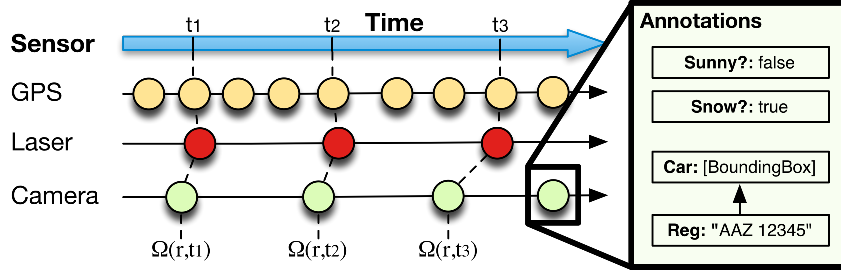

Fig. 3 Visualisation of some raw GPS, laser, and camera data and the relative times they were captured. An example annotation hierarchy for a single image datum is shown – note that the 'Car' annotation has a child that contains the detected registration number. Links between data records that are temporally joined at times t_1, t_2, and t_3 are shown as dashed lines.

$$\{s^i_{t^i}\} \leftarrow \Omega(r,t) \tag{2}$$

Where t^i is the nearest discretized timestamp for which an output from s^i exists.

These observations give rise to three relations (tables) whose rows respectively represent individual robotic platforms, individual sensors, and individual data:

$$\text{Platforms}(\underline{platform_id}, \ldots)$$

$$\text{Sensors}(\underline{sensor_id}, \ldots)$$

$$\text{Data}(\underline{data_id}, platform_id, sensor_id, timestamp, data, \ldots)$$

Expressed in relational algebra, Equation (1) becomes a simple selection over rows in the Data relation:

$$\sigma_{\text{platform_id}=r \wedge \text{sensor_id}=s^i \wedge \text{timestamp}\approx t}(\text{Data})$$

Equation (2) is given by:

$$\sigma_{\text{timestamp}\approx t}(D_{r,s^1} \bowtie_T \ldots \bowtie_T D_{r,s^N})$$

Where $\{s^1,\ldots,s^N\} = S(r)$, $D_{r,s^i} = \sigma_{\text{platform_id}=r \wedge \text{sensor_id}=s^i}(\text{Data})$ and $\bowtie_T$ is a *temporal join* operator that associates records whose timestamps are close together.

4.2 Annotations

Annotations are characterised by the fact that they mandatorily relate to other data – whether raw data, other annotations, or otherwise. In other words, they are 'meaningless' without context. To expand on our traffic light example: one detector may

create annotations consisting of bounding boxes that identify the locations of traffic lights in camera images. Another detector might annotate *these* annotations, indicating the state of the traffic lights, or other higher-level properties. Metadata, such as the version of the algorithm used to generate the labels, might also be included.

Annotations for raw data are stored in their own recursive relations:

$$\text{Annotations}(\underline{annotation_id}, data_id, parent_id, annotation_data, \ldots)$$

The *data_id* field references a row in the Data relation, and the *parent_id* field references a lower-level annotation. One and only one of these fields must be non-empty. Queries such as 'find all raw data with a particular annotation θ' are simply a selection over the join of the Data and Annotations relations (more complex compound queries can have any number of conditions chained together):

$$\sigma_{\text{annotation_data}=\theta}(\text{Data} \bowtie \text{Annotations})$$

Figure 3 shows graphically how annotations and raw data are related.

4.3 Other data

This class encompasses any other arbitrary data that do not fall neatly into the two aformentioned categories. For example, it includes completely standalone data, such as OpenStreetMap maps and sensor calibration information. Harnessing the power of the relational model, these data can still link to raw sensor data and annotations if the need arises. For example, one of our traffic light annotations could link to the OpenStreetMap intersection ID representing where the light is located.

4.3.1 Multigraph maps

One ubiquitous pattern that we have explicitly considered is the use of a directed multigraph structure (*GraphDB*) to represent maps for use by localisation and navigation systems, for example EBN. The graph is defined by two relations and is encoded in an edge-list representation. Two additional relations store links between nodes, edges, and annotations:

$$\text{Nodes}(\underline{node_id})$$

$$\text{Edges}(\underline{edge_id}, node_from_id, node_to_id)$$

$$\text{NodeAnnotations}(\underline{node_id, annotation_id})$$

$$\text{EdgeAnnotations}(\underline{edge_id, annotation_id})$$

An example of how a map might be represented using this structure is shown in Figure 4.

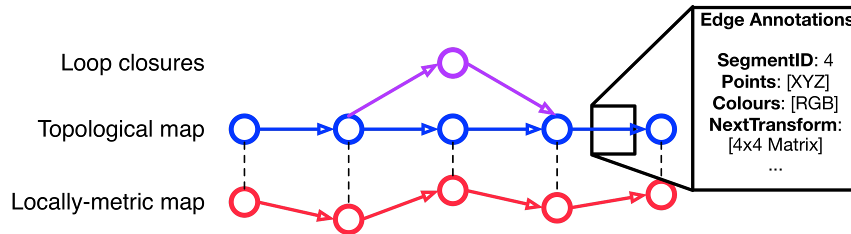

Fig. 4 Example GraphDB structure representing a metric map, topological map, and loop closures. Example annotations linked to a single edge are shown.

5 Implementation

5.1 How do we store data?

We have standardized the way raw sensor data are stored in an attempt to eliminate fragmentation of our tools and methods. A *monolithic file* contains a sequence of atomic data records. Each record consists of an XML header and a binary blob containing a serialised Google Protocol Buffer[2] message. The XML header contains sensor-agnostic metadata about the message such as a timestamp indicating when it was collected, its length in bytes, and the name of the corresponding message format. As sensor data are being collected, monolithic files are constructed on the fly. Raw sensor data are streamed through a driver which converts them into messages of the appropriate format and both the message and header are appended to the output monolithic. As they are simply an unlinked sequence of distinct and atomic records, monolithics can be constructed, split, and concatenated easily, and their length is limited only by the underlying file system.

Raw sensor data are stored in monolithic files in a well-defined directory structure that is organised by the platform and sensors used to collect them, and time of collection. The top-level directory is located on a network drive that allows users to mount it to anywhere in their own filesystem.

5.2 Relational data and query framework

We use a relational database management system (DBMS) – either PostgreSQL[3] or SQLite[4], depending on our particular use case. PostgreSQL is designed for a multi-user environment and has full transactional and locking abilities, allowing many concurrent connections to the database without the risk of data corruption. These

[2] https://developers.google.com/protocol-buffers/
[3] http://www.postgresql.org/
[4] http://www.sqlite.org/

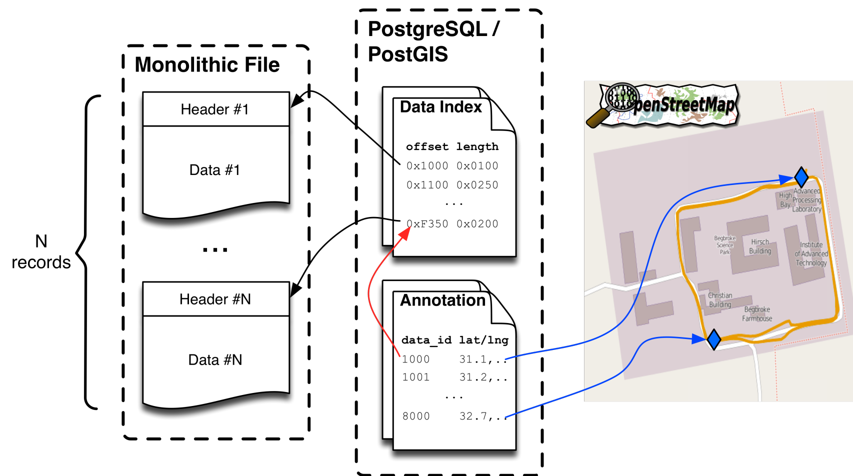

Fig. 5 Overview of the concrete data model structure. At the lowest level, raw data are stored in sequential records in flat 'monolithic' files. An index table in the DBMS holds offsets that point to these records (represented by black arrows). Annotations exist in separate tables with each individual annotation pointing to some record in the index (red arrow) and optionally to any other data – in this example, to locations in OpenStreetMap (blue arrows).

guarantees come at the cost of performance – queries must be sent over pipe or network to the PostgreSQL server process which executes them and returns results over the same medium. For real-time, online applications, we therefore prefer SQLite which stores all necessary data in a single file. Applications access this file via a shared library so no costly inter-process or network communication is required.

As SQL is a concrete superset of relational algebra, the schemas and queries described in Section 4 can be translated trivially (see [6] for an overview) into appropriate `SELECT` statements. In particular, any 'linking' of data is done using foreign key constraints. Compound queries are implemented using joins or subqueries.

As raw data are kept in monolithic files, we only store references to them in the DBMS. They do not contain a built-in index, meaning searching them for specific data requires a slow linear scan. Therefore, we store an index in an SQL table which holds offsets into monolithic files for every record. This table corresponds to the Data relation described in Section 4.1.

Annotations are stored in one global table which corresponds to the Annotation relation described in Section 4.2. This encodes the annotation heirarchy. Annotation-specific data (for example, the bounding boxes from our traffic light detector) are then stored in their own tables. Records in these tables link to their corresponding records in the global table. Annotations can then be created, updated, and deleted using SQL `INSERT`, `UPDATE`, and `DELETE` statements, respectively.

The GraphDB structure is implemented as tables which correspond to the relations described in Section 4.3.1. A dedicated API is provided that handles graph-

Query	Execution Time
Insert approx. 250,000,000 index records	12 hours approx.
Select single datum by ID	0.06ms
Select single datum by platform, sensor, and timestamp	0.06ms
Temporal join to select data by platform and timestamp	7.5ms
Temporal join for every record in an approx. 60GB dataset	14.7s

Table 1 Measured execution times for some example queries. Performed using a test PostgreSQL instance containing a 250,000,000-record subset of our data (approximately 20TB).

based operations such as creating, querying, and deleting nodes, edges, and their respective annotations.

In addition, we keep a subset of OpenStreetMap [5] consisting of all roads and regions in England. This is stored locally in our centralised PostgreSQL instance using the PostGIS[5] extension, allowing us to perform geospatial queries very efficiently.

Figure 5 shows an example of how the aforementioned components of the database interact.

6 Performance

The importance we ascribe to run-time performance of the database system depends on the specific use case in question. For example, batch tasks that process our entire 50TB of data have very different performance expectations to those of navigation systems running on live robots.

Both PostgreSQL and SQLite support indexes on data tables which we use to improve the performance of certain queries. These are implemented using well known B-Tree [7], R-Tree [8], and hash table data structures. For example, we create a B-Tree index on the *data_id* field in the Data table. This reduces the search for a record identified by a specific *data_id* from $O(n)$ to $O(\log n)$. For most long-running batch tasks, the speedups afforded by these indexes are sufficient. Table 1 shows measured execution times for some example queries of this nature.

In the rest of this section we analyse some of the performance characteristics of PostgreSQL and SQLite under particular configurations.

6.1 PostgreSQL vs SQLite

PostgreSQL is a server-based DBMS, meaning queries are issued via relatively costly interprocess communication or network calls. In return it provides full transactional and fault-tolerant behaviour and therefore allows many users to read and write data simultaneously without interfering with each other.

[5] http://postgis.refractions.net/

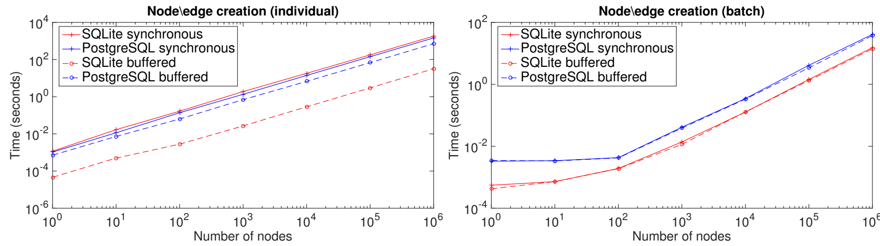

Fig. 6 Log-log plots showing time taken to create linear chains of various lengths in PostgreSQL and SQLite, in both synchronous and buffered modes. On the left, nodes were all created individually – i.e. n calls to a `CreateNode` API function. On the right, nodes were created in batch using a single API call.

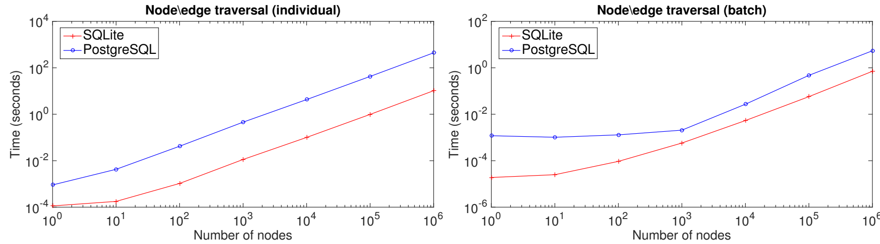

Fig. 7 Log-log plots showing time taken to read linear chains of various lengths in both PostgreSQL and SQLite. On the left, nodes were all read individually – i.e. n calls to a `GetNextNode` API function. On the right, the entire chain was read in batch using a single API call.

For real-time use cases, we instead prefer SQLite. It is linked statically and stores all of its data, metadata, and indexes in a single file which vastly reduces the overhead of its API calls.

Here we give performance results for the creation and traversal of a ‘worst case’ linked list graph structure. Although somewhat contrived, this example allows us to identify certain bottlenecks and observe how different configurations perform relative to one another. In the real world, access patterns and disk speed should also be major considerations.

Experiments were performed on an early 2011-era MacBook pro with 8GB RAM and a 500GB hard drive, with a PostgreSQL server running locally.

6.1.1 Experiments and Analysis

To test write performance, we create linear chains (successive nodes connected by single edges) of varying lengths n in the GraphDB. PostgreSQL and SQLite are tested in both *synchronous* (every write is fully flushed to disk) and *buffered* (writes are buffered in memory by the host operating system) modes. In the synchronous case, a write-ahead log is also used. In addition, we compare the creation of all nodes and edges in one operation (*batch*) versus creating them individually. The former

case is more likely to be the behaviour of an offline batch task that has all data already available to it, whereas the latter behaviour is more likely to be exhibited by an online task that is constantly processing new data from its environment.

Timing results for write operations are shown in Figure 6. Unsurprisingly, buffering individual writes is faster for both PostgreSQL and SQLite, although much more dramatically for the latter. Buffering batch writes makes virtually no difference. Batch writes of more than about 10-100 nodes are significantly faster than writing them individually. This is likely due to the overhead of API calls which begins to dominate in longer chains, particularly for PostgreSQL.

To test read performance, we load the previously created linear chains. Again we test how reading the whole graph in one operation (batch) compares with traversing the chain one node at a time.

Timing results for read operations are shown in Figure 7. Similarly we see that reading in batch is far quicker than traversing nodes individually. Additionally, SQLite is faster than PostgreSQL by almost 1-2 orders of magnitude in all cases.

Overall SQLite consistently outperforms PostgreSQL, although the difference is less marked in batch than individual cases. We believe this is because of the overhead associated with interprocess communication return trips, of which there are $O(1)$ in the batch case compared with $O(n)$ individually. We use this as justification for our use of PostgreSQL as our centralised DBMS, suitable for use with long-running offline tasks—where the multi-user and fault tolerance properties outweigh slight performance gains—and SQLite for real-time tasks.

7 Use Case: Experience Based Navigation

Experience-Based Navigation (EBN) [1] is a general-purpose framework for vast-scale, lifelong visual navigation. Outdoor environments exhibit challenging appearance change as the result of variation in lighting, weather, and season. EBN models the world using a series of overlapping ‘experiences’, where each experience models the world under particular conditions. New experiences are added in real-time when the robot is unable to sufficiently localise live camera images in the map of experiences. Additionally, the robot learns from its previous use of the experience map in order to more accurately retrieve relevant experiences from memory at run-time. The system has been tested on datasets totalling 206km of travel, successfully running in real-time at 20Hz [9].

Since EBN must operate over vast distances and throughout the lifetime of a robot, it is essential that the map of experiences is persisted in long-term storage. In order to maintain reasonable memory requirements, the system needs to be able to selectively load relevant portions of the map into memory, while leaving others on disk. To meet real-time constraints, these data must also be efficiently accessible.

The map of experiences is represented as a graph. Nodes describe the appearance of the environment at particular times and poses, while edges specify topological and metric links between these places (see Figure 1).

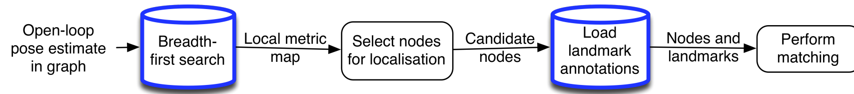

Fig. 8 Overview of the EBN localisation pipeline. This process is strictly required to run at 20Hz – the system has 50ms to complete, including data access tasks (highlighted in blue).

Fig. 9 Overview of the EBN experience creation pipeline. This process has less strict runtime requirements and runs in a background thread. Writes to the database are highlighted in red.

The GraphDB meets these requirements and therefore provides the mechanism for storing and retrieving data. Raw sensor data (e.g. images) and processed data (e.g. visual landmarks) are linked to corresponding nodes as annotations. Edges are annotated with 6DOF relative transformations, giving the graph a local metric structure.

The pipeline for localisation is shown in Figure 8. The breadth-first search enables EBN to load relevant portions of the map (nodes nearby the robot) into memory at runtime. Since this search may require a large number of read requests, the SQLite implementation of the GraphDB is used in buffered mode to maintain real-time performance.

A core feature of EBN is the ability to add new experiences to the map in real-time (pipeline shown in Figure 9). These experience creation tasks are processed in an independent thread so that slow disk write speeds do not impact on real-time performance. Figure 10 shows timing measurements for both the localisation and experience creation pipelines.

8 Conclusion

In this work we have motivated the need to move away from privileged datasets and ad-hoc data storage and annotation. These habits do not allow us to fully utilise our resources. Data take valuable time to collect and annotate and we make the case that it is impossible to predict which will be useful as they are collected. It is often only after the fact—weeks or months later—that we realise the full potential of some data. Without a consistent framework like ours in place, we would likely forget these useful data, not to mention waste time reimplementing separate storage and retrieval back-ends. We firmly believe that a system like this is the way forward for robotics applications – text files, readmes and wikis are no longer sufficient for many of our data management needs.

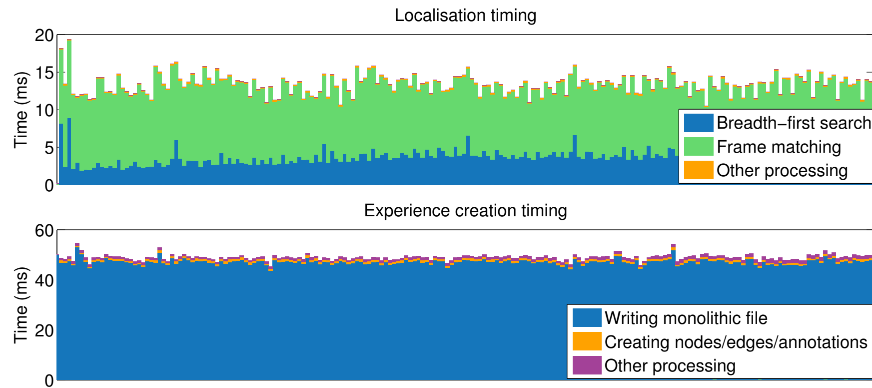

Fig. 10 Sampled timing measurements for the localisation and experience creation pipelines. Dominant database access periods are shown in blue. It can be seen that during localisation, these accesses comfortably meet the 50ms target. Experience creation takes in general longer however still runs under 50ms for majority of the time. For this task, occasional spikes in disk access latency do not impact on real-time performance as it is handled in an independent thread.

9 Acknowledgments

Peter Nelson is supported by an EPSRC Doctoral Training Account. Chris Linegar is supported by the Rhodes Trust. Paul Newman is supported by EPSRC Leadership Fellowship EP/I005021/1.

References

1. 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.
2. M. Smith, I. Baldwin, W. Churchill, R. Paul, and P. Newman, "The New College vision and laser data set," *The International Journal of Robotics Research*, vol. 28, pp. 595–599, May 2009.
3. A. S. Huang, M. Antone, E. Olson, L. Fletcher, D. Moore, S. Teller, and J. Leonard, "A High-rate, Heterogeneous Data Set From The DARPA Urban Challenge," *Int. J. Rob. Res.*, vol. 29, pp. 1595–1601, Nov. 2010.
4. M. Waibel, M. Beetz, J. Civera, R. D'Andrea, J. Elfring, D. Galvez-Lopez, K. Haussermann, R. Janssen, J. Montiel, A. Perzylo, *et al.*, "Roboearth," *Robotics & Automation Magazine, IEEE*, vol. 18, no. 2, pp. 69–82, 2011.
5. M. Haklay and P. Weber, "OpenStreetMap: User-generated street maps," *Pervasive Computing, IEEE*, vol. 7, no. 4, pp. 12–18, 2008.
6. R. Ramakrishnan and J. Gehrke, *Database management systems*. Osborne/McGraw-Hill, 2000.
7. R. Bayer, "Organization and maintenance of large ordered indexes," *Acta Informatica*, vol. 1, no. 3, pp. 173–189, 1972.
8. A. Guttman, *R-trees: a dynamic index structure for spatial searching*, vol. 14. ACM, 1984.
9. C. Linegar, W. Churchill, and P. Newman, "Work Smart, Not Hard: Recalling Relevant Experiences for Vast-Scale but Time-Constrained Localisation," in *Proc. IEEE International Conference on Robotics and Automation (ICRA2015)*, 2015.