Bundled References

Release 0.1

Carr

Dec 02, 2022

RESEARCH

Bundled References Documentation serves as the wiki and documentation for an independent research study being conducted by Alexander Carr under Professor Ahmed Hassan at Lehigh University. This site holds both information about the Bundled References code base as well as accompanying background information that helped advance or contribute to the project.

The goal of this independent research study is to take the Bundled References linked data structure and adapt it to be used in a graph database. Range queries across nodes within a graph database are common in graph query workloads, which creates an opportunity to increase graph database performance with Bundled References.

Note: This project is under active development.

CHAPTER

ONE

RESEARCH

1.1 Papers

1.1.1 Bundling Linked Data Structures for Linearizable Range Queries

The Bundled References implementation described in this paper serve as the inspiration for applying Bundled References to a graph database. The goal of my project is to take this implementation of Bundled References and add additional update functionality. Then, using an augmented Bundled References data structure, adapt a graph database that uses linked data structures to use Bundled References instead. The goal is for this adaptation to increase the performance of range queries on graph database nodes. Bundling Linked Data Structures for Linearizable Range Queries was authored by Jacob Nelson-Slivon, Ahmed Hassan, and Roberto Palmieri.

Abstract

We present bundled references, a new building block to provide linearizable range query operations for highly concurrent lock-based linked data structures. Bundled references allow range queries to traverse a path through the data structure that is consistent with the target atomic snapshot. We demonstrate our technique with three data structures: a linked list, skip list, and a binary search tree. Our evaluation reveals that in mixed workloads, our design can improve upon the state-of-the-art techniques by 1.2x-1.8x for a skip list and 1.3x-3.7x for a binary search tree. We also integrate our bundled data structure into the DBx1000 in-memory database, yielding up to 40% gain over the same competitors.

1.1.2 Black-box Concurrent Data Structures for NUMA Architectures

Besides applying Bundled References to graph databases, there are other ways of achieving higher performance. One of those ways is through taking advantage of NUMA architectures. The reason for reviewing this paper was to create a better understanding of how data structures could be adapted to take full advantage of NUMA machines. Black-box Concurrent Data Structures for NUMA Architectures was authored by Irina Calciu, Siddhartha Sen, Mahesh Balakrishnan, and Marcos K. Aguilera.

Abstract

High-performance servers are Non-Uniform Memory Access (NUMA) machines. To fully leverage these machines, programmers need efficient concurrent data structures that are aware of the NUMA performance artifacts. We propose Node Replication (NR), a black-box approach to obtaining such data structures. NR takes an arbitrary sequential data structure and automatically transforms it into a NUMA-aware concurrent data structure satisfying linearizability. Using NR requires no expertise in concurrent data structure design, and the result is free of concurrency bugs. NR draws ideas from two disciplines: shared-memory algorithms and distributed systems. Briefly, NR implements a NUMA-aware shared log, and then uses the log to replicate data structures consistently across NUMA nodes. NR is best suited for contended data structures, where it can outperform lock-free algorithms by 3.1x, and lock-based solutions by 30x. To show the benefits of NR to a real application, we apply NR to the data structures of Redis, an in-memory storage system. The result outperforms other methods by up to 14x. The cost of NR is additional memory for its log and replicas.

1.1.3 NUMASK: High Performance Scalable Skip List for NUMA

To dive a bit deeper on NUMA performance improvements, I reviewed a paper that used NUMA to speed up implementation of a skip list, which is a linked data structure that could also be adapted for Bundled References. Therefore, my goal was to consider the benefits of using Bundled References and NUMA aware linked data structures in graph databases. NUMASK: High Performance Scalable Skip List for NUMA was authored by Henry Daly, Ahmed Hassan, Michael F. Spear, and Roberto Palmieri.

Abstract

This paper presents NUMASK, a skip list data structure specifically designed to exploit the characteristics of Non-Uniform Memory Access (NUMA) architectures to improve performance. NUMASK deploys an architecture around a concurrent skip list so that all metadata accesses (e.g., traversals of the skip list index levels) read and write memory blocks allocated in the NUMA zone where the thread is executing. To the best of our knowledge, NUMASK is the first NUMA- aware skip list design that goes beyond merely limiting the performance penalties introduced by NUMA, and leverages the NUMA architecture to outperform state-of-the-art concurrent high- performance implementations. We tested NUMASK on a four-socket server. Its performance scales for both read-intensive and write-intensive workloads (tested up to 160 threads). In write- intensive workload, NUMASK shows speedups over competitors in the range of 2x to 16x.

1.1.4 Harnessing Epoch-based Reclamation for Efficient Range Queries

Bundled References uses an epoch based global timestamp to order references, which greatly speeds up range queries. A competing linked data structure described in this paper also utilizes an epoch based global timestamp to speed up range queries as well. I reviewed this paper in order to get a better understanding of how this implementation differs from Bundled References, and consider if there are any graph database applications for this linked data structure. Harnessing Epoch-based Reclamation for Efficient Range Queries was authored by Maya Arbel-Raviv and Trevor Brown.

Abstract

Concurrent sets with range query operations are highly desirable in applications such as in-memory databases. However, few set implementations offer range queries. Known techniques for augmenting data structures with range queries (or operations that can be used to build range queries) have numerous problems that limit their usefulness. For example, they impose high overhead or rely heavily on garbage collection. In this work, we show how to augment data structures with highly efficient range queries, without relying on garbage collection. We identify a property of epoch-based memory reclamation algorithms that makes them ideal for implementing range queries, and produce three algorithms, which use locks, transactional memory and lock-free techniques, respectively. Our algorithms are applicable to more data structures than previous work, and are shown to be highly efficient on a large scale Intel system.

1.1.5 Constant-Time Snapshots with Applications to Concurrent Data Structures

Another competing linked data structure optimized for range queries is versioned CAS (vCAS), which is described in this paper. This implements a unique way of subsituting new verions of linked nodes during a compare and swap operation. I reviewed this paper to analyze the performance of vCAS in relation to Bundled References. Constant-Time Snapshots with Applications to Concurrent Data Structures was authored by Yuanhao Wei, Naama Ben-David, Guy E. Blelloch, Panagiota Fatourou, Eric Ruppert, and Yihan Sun.

Abstract

Given a concurrent data structure, we present an approach for efficiently taking snapshots of its constituent CAS objects. More specifically, we support a constant-time operation that returns a snapshot handle. This snapshot handle can later be used to read the value of any base object at the time the snap- shot was taken. Reading an earlier version of a base object is wait-free and takes time proportional to the number of successful writes to the object since the snapshot was taken. Importantly, our approach preserves all the time bounds and parallelism of the original data structure.

Our fast, flexible snapshots yield simple, efficient imple- mentations of atomic multi-point queries on a large class of concurrent data structures. For example, in a search tree where child pointers are updated using CAS, once a snapshot is taken, one can atomically search for ranges of keys, find the first key that matches some criteria, or check if a collection of keys are all present, simply by running a standard sequential algorithm on a snapshot of the tree.

To evaluate the performance of our approach, we apply it to three search trees, one balanced and two not. Experiments show that the overhead of supporting snapshots is low across a variety of workloads. Moreover, in almost all cases, range queries on the trees built from our snapshots perform as well as or better than state-of-the-art concurrent data structures that support atomic range queries.

1.1.6 G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture

G-Tran is a graph database implementation that uses a Multi-Version Optimistic Concurrency Control (MV-OCC) mechanism to address large read or write sets, very similar to a range query. This MV-OCC system acts very similarly to Bundled References, just implemented in a different way. Reviewing this paper provided a deep dive into a graph database that is adapted for increased performance on range queries, which is the goal of using Bundled References in a graph database implementation. G-Tran: A High Performance Distributed Graph Database with a Decentralized Architecture was authored by Hongzhi Chen, Changji Li, Chenguang Zheng, Chenghuan Huang, Juncheng Fang, James Cheng, and Jian Zhang.

Abstract

Graph transaction processing poses unique challenges such as ran- dom data access due to the irregularity of graph structures, low throughput and high abort rate due to the relatively large read/write sets in graph transactions. To address these challenges, we present G-Tran, a remote direct memory access (RDMA)-enabled distributed in-memory graph database with serializable and snapshot isolation support. First, we propose a graph-native data store to achieve good data locality and fast data access for transactional updates and queries. Second, G-Tran adopts a fully decentralized architec- ture that leverages RDMA to process distributed transactions with the massively parallel processing (MPP) model, which can achieve high performance by utilizing all computing resources. In addition, we propose a new multi-version optimistic concurrency control (MV-OCC) protocol with two optimizations to address the issue of large read/write sets in graph transactions. Extensive experiments show that G-Tran achieves competitive performance compared with other popular graph databases on benchmark workloads.

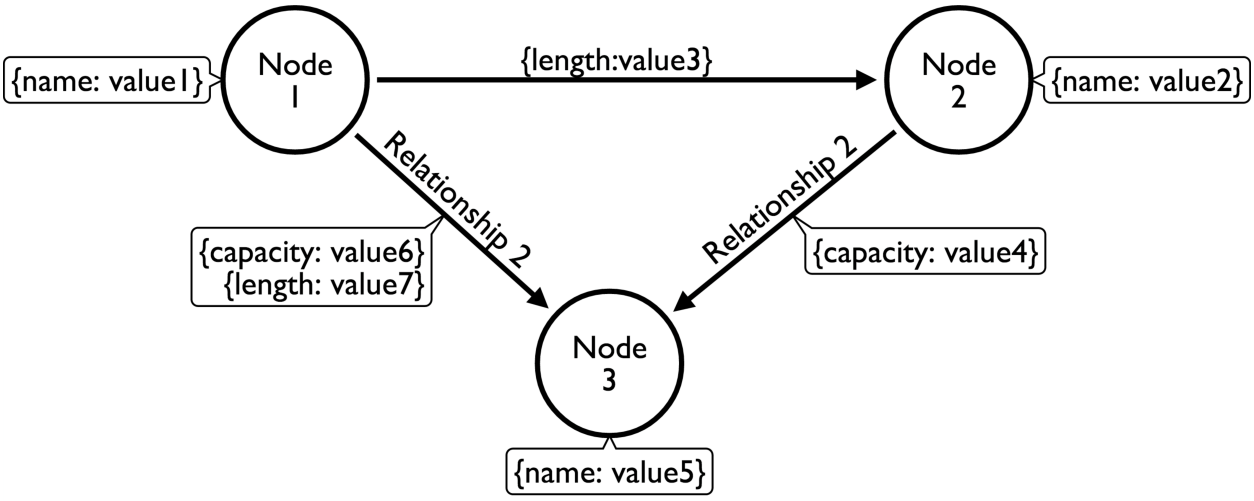
1.2 Graph Databases

1.2.1 What is a Graph Database?

A graph database is a specialized database platform designed specifically for storing and manipulating graphs. Graphs are made up of vertices, edges, and properties, which together can represent many different forms of data. Due to the various shapes and sizes of graphs, most graph databases are implemented using NoSQL, which means they are not using a traditional relational database model. In fact, a major benefit of not being forced into a structured relational model is that there have become two leading methods for storing and representing graph databases, the property graph model and resource description framework (RDF) model. These two methods are both very popular among industry leading graph database technologies, proving they both can be viable when used to represent graph data structures.

Property Graph Model

Property graphs model graphs as a set of nodes and relationships, where either nodes or relationships can have additional attributes, called properties. This way of representing a set of vertices and edges is straightforward, as it adheres closely to the visual representation of a traditional graph. In memory, a property graph is usually stored using some adaptation of a linked data structure, such that nodes and relationships contain pointers to each other.



Resource Description Framework Model

The RDF model is a bit more difficult to grasp than the property graph model. RDF is a method of representing highly interconnected data. Therefore, it can easily be applied to graph databases. In the RDF model, data is stored in triples: subject, predicate, and object. This is very similar to a node and a relationship in the property graph model. As triples are created, a network of subjects, predicates, and objects is formed. This network looks exactly like a graph, with vertices made up of subjects and objects and predicates as edges. The biggest difference between the property graph model and the RDF model is that the unique identifier in an RDF model is the triple, whereas in the property graph model, nodes and relationships can be uniquely identified individually.

1.2.2 Popular Graph Databases

The market for graph databases is large, and there exist many different companies and open source projects that have developed different graph database products. In order for my adaptation of a graph database with Bundled References to be relevant, it must adapt an already common graph database product. To find the currently most used graph databases, I consulted a market research report conducted by Forrester Research. This report indicated that Neo4j and Amazon Web Services (Amazon Neptune) are the industry leaders, which is unique in the fact that they use opposite graph models. Neo4j is an open source project utilizing the property graph model, whereas Amazon Neptune is a proprietary graph database implemented using the RDF model. To learn more, I investigated both of these technologies.

Amazon Neptune

Amazon Neptune is a fully managed graph database service designed specifically for highly connected graph datasets. It is implemented using a derivative of the RDF model, in which instead of using triples to identify data, it uses a quad: subject, predicate, object, and graph. It functions the same way as the RDF model, where a network of these quads develops into a graph. However, the fourth graph element provides additional metadata that Amazon Neptune uses for named graph identifiers, which add support for multiple graphs. Amazon Neptune has also added support for popular query language Gremlin, which is used for querying.

The major problem I encountered with Amazon Neptune was that it is proprietary. This means that if I am looking to add additional functionality using Bundled References, I would have to add on to the service as a third party instead of being able to integrate directly. Therefore, although this is an industry leader in graph databases, it does not fit into the scope of this project.

Neo4j

Neo4j is an open source native graph database using the property graph model. Because it is open source, its exact implementation is public. In fact, I was able to utilize a textbook called Graph Databases, which describes in depth the storage model of Neo4j.

In memory, Neo4j stores nodes, relationships, and properties using linked lists. Each of these object types are stored separately as a fixed size object. Nodes contain pointers to the head of their relationship list and the head of their property list. The property list acts as a singly linked list where each property points to the next property in the list. The relationship list is where most of the querying for Neo4j is implemented. Therefore, each relationship in the relationship list has a pointer to the source node and destination node for itself, the previous relationship in the list, and the next relationship in the list. This allows for queries to only need to traverse the relationship list in order to find both nodes and relationships.

Due to the reliance on linked data structures for the implementation of Neo4j and the fact that it is open source, it seems like the optimal graph database to add Bundled References to. In fact, because the code base is available on GitHub, I have the ability to implement Bundled References directly into the linked lists themselves. Therefore, this project will add Bundled Reference functionality to Neo4j in order to increase its performance with aggregate and range queries.

1.3 Graph Benchmarks

1.3.1 Benchmarking Graph Databases

In order to understand the benefits of implementing Bundled References in a graph database, measuring performance will be required. However, there are many graph database benchmark options available, and they are not created equal. Each benchmark tests different algorithms and is compatible with a specific set of graph databases. To best evaluate the effect Bundled References has on a graph database, graph benchmarks will need to be analyzed as well.

1.3.2 Existing Graph Database Benchmarks

Graph databases are not new in Computer Science, and as such, there exist many graph database benchmarks. However, they each differ on the algorithms they implement and the databases they are compatible with. After discovering over 10 possible graph database benchmarks, I narrowed it down to the two most comprehensive options, the Linked Data Benchmark Council (LDBC) and Graph Benchmark.

Linked Data Benchmark Council

LDBC is a group of individuals that have created a set of test suites that can be used to evaluate the performance of various graph databases. Although they have many different test suites with varying data sets, their Social Network Benchmark (SNB) is the most relevant for testing a graph database implemented with Bundled References.

The SNB is split into two main workloads, a business intelligence workload focused on aggregation queries and a interactive workload focused on simpler queries with constant inserts and updates. Some of the supported queries include:

- Breadth First Search
- PageRank
- Single-source Shortest Path
- Local Clustering Coefficient

With extensive coverage of features, LDBC is a strong option for benchmarking an adapted graph database with Bundled References.

Graph Benchmark

Graph Benchmark was developed by Matteo Lissandrini, Martin Brugnara, and Yannis Velegrakis. This benchmark is highly customizable and provides functionality for a wide range of queries and graph databases. Just in the example test cases, over 70 different individual and bulk queries are highlighted. In addition, these tests also incorporate many different graph databases, including:

- Neo4j
- OrientDB
- Titan
- ArangoDB

Another major benefit of Graph Benchmark is the wide variety of data sets it supports as well. In fact, Graph Benchmark supports some data sets created by LDBC, like the SNB data. Therefore, due to Graph Benchmark's flexibility, it is the best choice to benchmark a graph database implementation with Bundled References.

Papers
: Research papers that contain relevant information to Bundled References.

Graph Databases
: Overview of current graph databases and how they are structured.

Graph Benchmarks
: Analysis of graph benchmarks that can be used to benchmark graph database performance.

CHAPTER

TWO

BUNDLED REFERENCES

2.1 Getting Started

2.1.1 Implementation Notes

This work graciously builds on a benchmark developed by Arbel-Raviv and Brown's work (https://bitbucket.org/trbot86/implementations/src/master/cpp/range_queries/) to provide linearizable range queries in linked data structures. We use their codebase as a starting point and implement our technique on top of the existing framework. The core of our bundling implementation is contained in the 'bundle' directory, which implements the global structures as well as necessary functions of bundling. The three data structures we implement are found in 'bundled_*' directories. The scripts necessary to produce the plots found in our paper are included under the 'microbench' directory.

2.1.2 Getting Started Guide

Requirements

The experiments from the paper were executed on a 4-socket machine with Intel Xeon Platinum 8160 processors running Ubuntu 20.04. However, we also verified our results on a dual-socket machine with Intel Xeon E5-2630 v3 processors running Ubuntu 18.04. The C++11 libraries are requried to build and run the experiments, while the Python libraries are used for plotting results.

Unix Utilities:

```
make (e.g., sudo apt install make)
dos2unix (e.g., sudo apt install dos2unix)
bc (e.g., sudo apt install bc)
```

C++ Libraries:

```
libnuma (e.g., sudo apt install libnuma-dev)
libjemalloc (e.g., sudo apt install libjemalloc-dev)
libpapi (e.g., sudo apt install libpapi-dev)
```

Python libraries:

```
python (v3.10)
plotly (v5.1.0)
psutil (v5.8.0)
requests (v2.26.0)
pandas (v1.3.4)
absl-py (v0.13.0)
```

The above libraries can be installed with Miniconda, whose installation instructions can be found here (https://docs.conda.io/en/latest/miniconda.html). Generally speaking, you will download the appropriate installer for your machine, run the install script, and follow the prompts. After it is installed, use the following commands to prepare the environment needed for plotting output:

```
conda create -n paper63 python=3
conda activate paper63
conda install plotly psutil requests pandas absl-py
```

In order to reproduce our results, it is necessary to link against jemalloc. For convenience, our scripts assume that a symbolic link is available in the lib subdirectory. If you already have it installed on your system you can create a link to the shared library in lib so that the scripts can locate it. Otherwise, you can use the following command to install it in the previously configured conda environment and link it. Note that we assume the commands are run from the project's root directory:

```
conda install -c conda-forge jemalloc
ln -s $(conda info --base)/envs/paper63/lib/libjemalloc.so ./lib/libjemalloc.so
```

Docker. As a convenience, we also include a Dockerfile that will handle all of the setup steps outlined above. Note that running in a Docker container may inhibit performance, so we advise running on bare metal if you wish to reproduce our results. The Dockerfile can be used as a guide when doing so. Please see the comments in the Dockerfile for more detailed information. Once Docker is installed on your machine following the instructions here (https://docs.docker.com/engine/install/), you can build and run the container in the background by running the following commands from the root directory, where the Dockerfile is located:

```
docker build -t bundledrefs .
docker run -d -p 8022:22 bundledrefs
```

After, you can ssh into the container using:

```
ssh -p 8022 bundledrefs@localhost
```

and provide the password bundledrefs. From there, you will be able to follow the rest of this README as usual.

Implementation

`./bundle` implements the bundling interface as a linked list of bundle entries. In addition to the linked list bundle, there is an experimental cirular buffer bundle (not included in the paper) as well as an unsafe version that eliminates the overhead of ensuring bundle consistency for comparison.

`./bundle_lazylist`, `./bundle_skiplistlock` and `./bundle_citrus` each implement a data structure to which we apply bundling. Note that we do not apply our technique to the remaining data structures (which are lock-free) because our current bundling implementation would impose blocking.

`./vcas_lazylist`, `./vcas_skiplist_lock`, and `./vcas_citrus` each implement our porting of vCAS to lock-based data structures for the evaluation.

Experiments

`config.mk` is the primary configuration file and is used across all experiments. Users will need to update this file to match their system (more on this later).

The experiments that we report in the paper are located in the following directories.

- `./microbench` tests each data structure in isolation.
- `./macrobench` ports DBx1000 to include the implemented data structures.

Both of the above experiments are run by using their respective `./runscript.sh` scripts.

Generated Files

`experiment_list.txt` is generated by the microbenchmark, resides in `./microbench` and contains the list of experiments to run.

`./microbench/data` and `./macrobench/data` are the destination directories for all saved data during experiments. This is also where automatically generated .csv files will be saved when plotting the results.

`./figures` stores all generated plots. It is split first into the respective experiments and then into the data structures used. Note that the generated plots are saved as interactive HTML files and should be opened using a browser.

Getting Started
: How to get started with running the benchmark for Bundled References.