Graph Record Layout Research Environment

A graph database with some layout tools & methods

Fabian Klopfer

Databases and Information Systems Group
Department of Computer Science
University of Konstanz

Abstract:
The here described software is focussed on finding new methods to layout the records of a graph database such that the least possible accesses are necessary to satisfy a certain benchmark. A benchmark is here usualy a set of queries, for example traversal-based queries. The current state-of-the-art methods use the graph structure to statically layout the graph on disk using clustering, partitioning or community detection methods to form blocks along with some ordering algorithm. The methods that are to be explored are dynamic — either in terms of the query or in terms of a changing database. This document provides a high-level user guide to the structure of the database and the methods and tools that are implemented. Furthermore the specification as well as a more extensive design document are captured, along with some benchmark and visualization techniques.

Contents

Chapter 1

User Guide

1.1 Introduction

1.2 Database Architecture

1.2.1 Overview

1.2.2 Records

1.2.3 Access

1.2.4 IO

1.2.5 Cache

1.2.6 Query

1.2.7 Import

1.2.8 Layout

1.2.9 Visualization

1.2.10 Building, Tests, Coverage

1.3 Record Layout Methods

1.3.1 Theoretical Aspects

1.3.2 Static Layout

1.3.3 Dynamic Layout

1.4 Conclusion

1.4.1 Summary

1.4.2 Future Work

Chapter 2

Technical Documentation

2.1 Software Requirement Specification

2.1.1 Introduction

Purpose

The purpose of this Software Requirements Specification is to describe the features, constraints and demands of a research environment for the optimal layout of graph records on disk in detail. This document is intended for both the stakeholders and the developers of the system and will be proposed to the Dr. Theodoros Chondrogiannis, the supervising postdoctoral researcher.

Scope

This Software system shall implement a graph record layout research environment, that consists of a graph database along with tools, that rearranges the recrods of the database based on a predefined format, measure the number of disk accesses needed to service a certain query and that provide other layouts to compare against.

Definitions, Acronyms, Abbreviations

word	shortform	meaning
database	db	a software system to store and alter data in an organized manner.
Operating System	os	An operating system is system software that manages computer hardware, software resources, and provides common services for computer programs.
Portable Operating System Interface	POSIX	A specification for a set of OSes that covers for example Linux, macOS and BSD-style operating systems.
C Programming Language	C	a programming language.
Input/Output	IO	the notion of loading and storing information to media other than RAM and CPU registers. In this document hard drive and solid state disk are meant primarily.
Create, read, update, delete	CRUD	The basic database operations, that allow to create, read, update and delete a record.
Databases and Information Systems	DBIS	The name of the group at the university of Konstanz, at which the software system is build.
Stanford Network Analytics Project	SNAP	A porject of the university of stanford that hosts many large scale graph data sets.
Least Recently Used	LRU	A strategy when evicting pages from a pool of memory.

Breadth-first Search	BFS	A graph traversal scheme, where all neighbours of the current node are visited before continuing with the next node.
Depth-first Search	DFS	A graph traversal scheme, where the next node is considered before visiting all neighbours of the current node.
Single Source Shortest Path	SSSP	The problem of finding the shortest path to all nodes in a graph from one source node.
...	...	...

Overview

In the second chapter several conditions, assumptions and circumstances will be mentioned, that help charachterizing the software's special use case. In the thrid chapter the concrete requirements are listed.

2.1.2 Overall description

Product Perspective

The product shall rely on functions of unixoid OSes. That is it uses the interfaces specified by the OpenGroup in the POSIX.1-2017 specification (also called IEEE Std 1003.1-2017) [1]. There are no relations to other software systems than the operating system during the runtime of the environment.

Product Functions

The product shall support different tasks in graph record layout research:

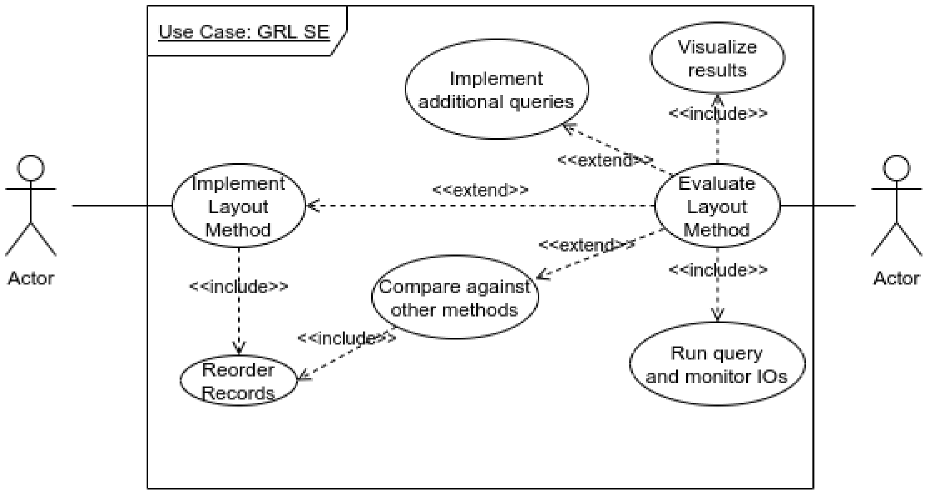
1. Import data into a graph database.
2. Query the database with a certain fixed function.
3. Rearange the record layout on disk with a layout given in a specific format.
4. Monitor the number of disk IOs.
5. Monitor the caching behaviour.
6. Provide interfaces for the implementation of new layout methods.
7. Provide a interfaces to ease the implementation of new queries.

This shall be done using a graph database, that shall save related information to disk and load them into a cache on access, as well as support the common CRUD functionality.

User Characteristics

The potential users are the staff and student assistants of the DBIS group. Therefore standard user will have technical knowldege, and have visited at least a basic course on databases. Futher the users are able to program in C. There are three different types of users of this research environment:

- Researchers: Implements new layout method, benchmarks the result of the method against other layouts, visualizes the result of the benchmark. Should at least know the theory of how records are stored.
- Supervisors: Works on the administration and coordination of the project. Eventually hires other researchers and developers. Also a researcher.
- (Future) Developers: Uses the existing framework to extend the functionality, for example to implement additional queries of a different type, add new features like storing properties and similar things. Needs to know details of database architecture and implementation, more advanced C programming and some software engineering.



Constraints

- The database is expected to be run on a standard notebook and desktop machine, that is it needs to run without momory-related errors on a machine with 8 GiB RAM, no matter the data set size.

Assumptions and Dependencies

The environment supports only POSIX-compliant OSes.

2.1.3 Specific Requirements

External interfaces

The POSIX interfaces.
The data format of some of the SNAP data sets.

2.1.4 Functional requirements

1 Data Storage and IO
The system shall

- 11 store a graph G consisting of nodes and edges (V, E).
- 12 store the records in a file on disk.
- 13 be able to grow and shrink the size of the file that is used.
- 14 be able to read from disk into main memory.
- 15 be able to write back to disk.

2 Data Caching and Memory
The system shall

- 21 not exceed a certain memory limit for both pages from disk and memory requirements from queries.
- 22 split an amount of memory into frames.
- 23 load data from disk into a frame on request.
- 24 maintain information on unused frames.
- 25 maintain a mapping from loaded data to frames.
- 26 evict data from memory when the memory limit is hit
- 27 be able to prefetch by reading more data than required in a neighbourhood

3 Data Access
The system shall

- 31 be able to calculate the location in the file of nodes and relationships efficiently
- 32 keep track of free record slots on disk.
- 33 be able to create, read, update and write nodes.
- 34 be able to create, read, update and write relationships.
- 35 provide an interface to easily retrieve and traverse data.
- 36 provide functions to retrieve common informations about nodes and relationships, like the node degree.
- 37 be able to monitor the numer of overall disk accesses.
- 38 be able to monitor the number of disk accesses that are necessary, if only a certain limited amount of memory is available.
- 39 be able to monitor the cache hit rate.
- 310 be able to monitor the prefetch hit rate.

4 Queries
The system shall

41 implement the most basic traversal schemes — BFS and DFS.

42 implement Dijksta's algorithm for finding all shortest paths from a single source (SSSP).

43 implement the A* algorithm for the shortest path problem.

44 implement the ALT algorithm for the shortest path problem.

45 provide data structures for the results of the above algorithms.

46 implement the Louvain method for community detection.

47 implement a random walk.

5 Data Import
The system shall

51 be able to import a set of standard data sets from the SNAP data set collection.

6 Layout Tools
The system shall

61 provide a function to generate a randomized record layout.

62 provide functions to reorganize the record layout on disk, given updated record IDs.

63 provide a function to sort the incidence list structure after reorganizing the record structure.

64 provide a function to reorganize the relationships given a layout of the vertices.

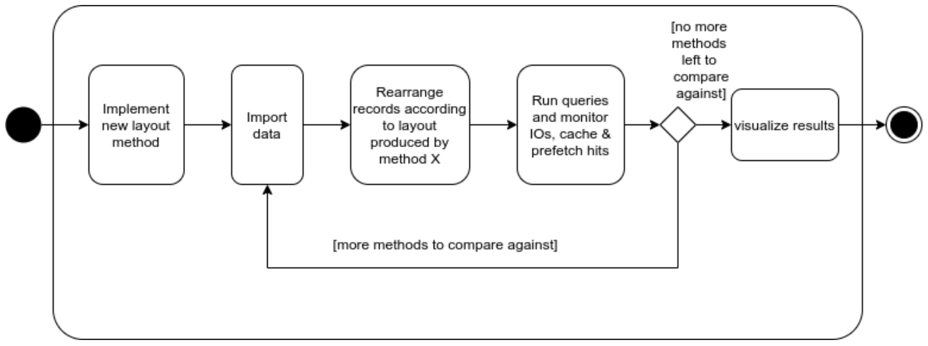
7 Layout Methods
The system shall

71 implement one static and one dynamic history-based layout method for a static database.

72 provide an interface for implementing new layout methods.

8 Visualization Tools
The system shall

81 provide functions to visualize and visually compare the data that was monitored by the caching functionality.


Implement new layout method
Import data
Rearrange records according to layout produced by method X
Run queries and monitor IOs, cache & prefetch hits
[no more methods left to compare against]
visualize results
[more methods to compare against]

2.1.5 Performance Requirements

1 The system shall work with limited RAM resources even for very large datasets.

2.1.6 Software System Attributes

1 Maintainability

2 Extensibility

3 Correctness

Bibliography

[1] *The Open Group Base Specifications Issue 7, 2018 edition, IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. June 1, 2021. URL: https://pubs.opengroup.org/onlinepubs/9699919799/functions/contents.html (visited on 06/01/2021).

2.2 Software Design Document

2.2.1 Introduction

Purpose

This document is written, to document the proposed architecture of a zoo management system. The document is inteded for the developers implementing the system, the persons deploying the final system and the other stakeholders to see the process and decide, if the developed system is according to the specifications.

Scope

The System developed is a zoo management system, that assists the zoo keepers in ordering food, managing and visualizing the feed cost of each animal, the administrators in managing budgets, preferred feed dealers and the staff. The software requirements specification is already present.

Definitions, Acronyms, Abbreviations

term	**short form**	**description**
Java	-	A portable programming language
Java Virtual Machine	JVM	The software system nessecary to run java applications.
Database Management System	DBMS	A software system used to manage, create and maintain databases.
relational DBMS	-	DBMS storing data in tables referencing each other.
Transport Control Protocol / Internet Protocol	TCP/IP	Low level Protocols to transfere data over the internet.
Secure Hypertext Transfer Protocol	HTTPS	The standard protocol to transfer data over the internet in a secure fashion.

References

1. SRS Document
2. Enumerated Requirements Document

Overview

The proposed architecture will be shown from a static point of view, eg. the decomposition into subsystems and their structure. Then the final deployment will be shown. This includes the distribution of the software across the different hardware used. Furthermore the data management is visible. Therefore the database design is modeled and the used DBMS is mentioned. The next part contains a description of the security features of the system and how the users are managed. The dynamic aspects of the system are following. This contains the description of interfaces between components, the behavior of the system and information flow. Finally, used design patterns are mentioned.

2.2.2 Software Architecture

Overview

The Zoo management system is a client - server system. It is composed out of the following components.

Client The client module is the complete system running on the client computer. It bundles the UI, navigation and can obtain an access token for the inner components.

Server The server module is the system running on the server computer. It hosts the database and provides user authentification capabilities.

FeedManager (FM) The FM system coordinates all requests concerning the supply, storage and management of feeds and the according feed dealers. It is mainly used by zoo keepers for buying feeds and checking the current quantity and expiration date of stock feeds.

EmployeeManager (EM) The EM system is mainly used by the secretary to create employee schedules, organize temporary workers, check the vacation requests of employees and manage the associated budget.

BudgetManager (BM) The BM system organizes the budget of the zoo. It is used to generate reports, get an overview of the current budget and provide information to prevent negative balances. It is mainly used by the zoo director to organize the subbudgets for feeds and employees as well as granting or dening requests for additional funding created by other employees.

UserManager (UM) The UM system is used to create and manage user accounts of the system. It is exclusively used by the zoo director.

Messaging subsystem (MSS) The MSS is used by the system to deliver messages to special user accounts. It is an external system, which is used by the zoo management system.

MessageManager (MM) The Message manager is a client component, which is used to display messages that are received from the MSS.

AuthUtility (AU) The AU is a compontent running on the server. It is the only way to access the system internals. This component checks, if provided login credentials are valid, and if the access level of the user is high enough.

Database subsystem (DBMS) The DBMS is the subsystem used for persistent data storage. It is used as a transaction based system.

Subsystem decomposition

Client The client component contains the main initialisation code. It provides a host window for the other components to draw their UI and navigational elements to switch the used component. In addition the component is used to obtain an access token from the server, which allows other components to comunicate with the server.

Server The server component contains the initialisation code to start the DBMS and routes incoming connections to the AuthUtility.

Manager Components The different manager components are implemented in a similar fashion: By using the *AbstractFactory* Pattern to get instances. The following two models figure and emphasis this pattern and the implementaion style.

The *BudgetManager* is instantiated by the *BudgetManagerFactory*. The budgets are modeled as *CompositeBudget*s and *SubBudget*s, both supporting a specific set of operations. In addition animal implements the *Budget* interface, because each *Animal* has an associated budget.

MessagingSubSystem The messaging susbsystem allows the zoo management system to send messages to specific user accounts and devices. The system has to guarantee the delivery of the messages. The architecture isn't specified any further, because the MSS is an external system and therefore not part of this document.

AuthUtility The AuthUtility is used to access the database. For every request the AuthUtility checks, if the login credentials are valid and not expired. Then the privilege level for the requested action is checkt. If the user has enough permissions the request is passed to the DBMS. The answer is returned to the client.

DatabaseManagementSystem The DBMS used is MariaDB. It is a high performance relational database management system. It is used, because it is a open source software and therefore free of costs. In addition it is widely used and tested. Therefore it is reliable and it may be assumed, that it is supported for the entire lifetime of the zoo management system.

Hardware/Software mapping

Figure shows the planned hardware software mapping of the zoo management system. The System is divided into a server and a client part. The server part will run on a server computer, availiable 24/7. The server is responsible for user authentication and the storage of data. The client side consists of tablets and personal computers. These will run the client application, which is used to contact the server via a HTTPS (on top of TCP/IP). The client application requires the Java Runtime Envirement to be installed on personal computers and tablet clients. The client provides the user interface and uses the messaging subsystem to deliver and receive messages to/from users of the system. The hardware allocation of the MSS is not part of this document.

Persistent data management

For persistent data management on the server side the open source relational DBMS MariaDB is used. Data is organized in tables according to the scheme in figure. Users have multiple activity logs associated. Each employee has exactly one user account for the system. For each employee, the associated vacation requests and a schedule is stored. Employees are workers of the zoo, so some data is shared with temporary workers (which are workers, too), for example name and wage/salary. For each worker there is possibly a WorkerFundRequest, if the budget was too small to hire the worker. This, in turn, is a specialization of a general FundRequest, which is associated with a given budget. Each budget consists of multiple subbudgets and may contain FundRequests. Another possible FundRequest is the FeedFundRequest, which is issued, if an order exceeds the feed budget. These requests and the associated orders are stored and linked. An order can be delayed and the delivery may be updated, so the possible DeliveryUpdates are stored. The order itself contains one or more Feed offerings. These offerings are stored as an association between FeedDealer and Feed, because the FeedDealer can make an offer for a feed. If a FeedDealer is removed from the system, all the associated feed offerings not contained in an order are deleted too. In addition the system stores all animals and the feeds they are eating. If an animal is removed from the system, the feeds consumed only by this one animal are deleted. If a feed is deleted, the associated offerings are removed, if they aren't contained in any order.

On the client side no DBMS is neseccary, because all used data is loaded at runtime. The only presistent data is the username, which is stored for the next usage of the system.

Access control and security

Network location restrictions The first layer of security is, to grant access to the system only out of the local network in the zoo. If this is too restrictive, it may be dropped.

Network protocol restrictions The client may only connect to the server using a secure HTTPS connection. This require the server to have a valid SSL certificate. The server will drop every connection attempt using insecure protocols.

User account restrictions The next layer is the AuthUtility. Each user of the system has a user account with a unique user ID and a password. These accounts are managed by the zoo director. When a user tries to contact the application server, he has to provide a valid access token and a valid user account id. The access token is obtained by sending the login credentials to the authentification server. If they are valid an access token is returned. This token has only limited validitiy and has to be obtained again, if timed out. To restrict access for zoo keepers and the secretary, every user account has an integer value associated, which represents the users access level. Each module can check the current access level of the user to determine which functionality is provided.

Storage policies The user passwords aren't stored in plain text. They are hashed using a state of the art hash funktion (PBKDF2 with iteration count larger than 20000) and stored together with the used hash function, salt, and function parameters (iteration count). Because a lot of people use the same password for different services, this prevents potential intruders from taking the password list and using the passwords on other web services or obtaining passwords from other services and using them for the zoo management system.

Logging As an additional security feature the system logs the account accessing the system for every access. These logs are visible for the zoo director.

Global software control

Startup behaviour **Server:** When the server machine is booted, the operating system automatically invokes a startup handler of the system. This handler will boot up the database, the AuthUtility and then it will enable the request handling.
Client: The client application starts on user demand. The client will show a login form for the user. When the user entered his login data the client contacts the server to obtain an access token. The client creates different instances of *UIComponentFactory* for each UI component. The components are initialized with the access token and these factory objects.

Interfaces The `MessagingService` is provided by the MSS. It provides a possibility to notify users with messages. If the notifyUser function is invoked, it is ensured, that the user is notified and the message is stored.

```
Interface MessagingService {
  public void sendMessage(string sender,
      string accesstoken, string recipient, Message message) {
    require accessValid(username, accessToken);
    require messageHandle != null;
    require message.length > 0;
    ensure getMessages(recipient, database.getUser(recipient)
        .getAccessToken()).contains(message);
  }

  public Message[] messages getMessages(string username, string accessToken) {
    require accessValid(username, accessToken);
    require messageHandle != null;
    ensure messages != null;
  }

  public void messageRead(String username, string accessToken Message message) {
    require accessValid(username, accessToken);
    require message != null;
    require getMessages(username, accessToken).contains(message);
    ensure !getMessages(username, accessToken).contains(message);
  }
}
```

The `AccessWebService` interface provides three functions: *getAccessToken*, *accessValid* and *access*. *getAccessToken* is used to obtain access tokens. *accessValid* checks, if the provided access token is valid. This function is used by the MSS to authenticate users. The function *access* is used to execute operations on the database. In before a authentification check is made and it is tested if the user is allowed to execute the given operation.

```
Interface AccessWebService {
  public string accesstoken getAccessToken(string username, string password) {
    require username.length > 0 && password.length > 0;
    ensure database.getUser(username).accesstoken == accesstoken &&
    database.getUser(username).expirationDate;
    ensure containsLogEntry(user, system.getCurrentDate());
  }

  public boolean valid accessValid(string username, string accesstoken) {
  require database.existsEntry(username);
  ensure valid == database.getUser(username).isIsValidToken(accesstoken);
  }

  public Response r access(string username,
                              string accesstoken, Operation op) {
    require database.entryExists(username);
    ensure iff accessValid(username, accesstoken) then
      iff database.getUser(username).isLocked() then
        operation == new PasswordChangeRequiredResponse();
      else
        iff database.getUser().getPrivilegeLvl() > getPrivilegeLvl(operation) then
          operation is executed and r is the response of the operation;
        else
          r == new AccessDeniedResponse();
    ensure if accesstoken is not valid operation is not executed and r is null;
  }
}
```

Sequences table

Sequence Name	**accessValid**
Preconditions	• username is contained in the database • accesstoken is not null
Event order	1.0 initial call 1.1 AuthUtility requests the given User from the DB 1.2 The DB allocates and populates a User Object 1.3 The BD returns the User Object 1.4 AuthUtility ask the User object, if the token is valid 1.5 The User Object returns a boolean 1.6 This boolean is passed to the client
Postconditions	• the database is not modified
Quality requirements	• The accessValid call shall finish within at most 3 seconds.

Sequence Name	**getAccessToken**
Preconditions	• username is contained in the database • password is not null
Event order	1.0 initial request 1.1 AuthUtility requests the given User from the DB 1.2 The DB allocates and populates a User Object 1.3 The BD returns the User Object 1.4 The AuthUtility generates the password hash using HashImpl 1.6 The AuthUtility gets the original password hash from the User object 1.8 The hashes are compared 1.9 The hashes were equal so an access token is requested from the user object 1.10 The user object checks, if the currently active access token is valid 1.11 The token wasn't valid, so a new token is generated 1.12 The new Token is saved in the Database 1.13 Now the access token is valid and therefore returned to the AuthUtility 1.14 From there on it is passed to the callee 1.15 The password was false so null is returned
Postconditions	• the returned access token is valid • the returned access token is saved persistently in the DB
Quality requirements	• The getAccessToken call shall finish within at most 3 seconds.

Sequence Name	**access**
Preconditions	• username is contained in the database • access token is not null • the operation is not null
Event order	1.0 initial request 1.1 AuthUtility requests the given User from the DB 1.2 The DB allocates and populates a User Object 1.3 The BD returns the User Object 1.4 it is checked, if the access token is valid (using the User object) 1.6 the token was valid, so the access level of the user is checked 1.8 the access level of the operation and the user are compared 1.9 the user is allowed to execute the operation, so the operation is send to the DB 1.10 The DB Response is passed to the AuthUtility 1.11 The Response is passed to the callee 1.12 the user had no sufficient access permissions so null is returned to the callee 1.13 the access token was not valid, so null is returned
Postconditions	• the database is not modified
Quality requirements	• The accessValid call shall finish within at most 3 seconds.

Boundary conditions

1. MSS exists already.
2. The zoo has a sufficiently fast network connection availible to use the system without maintaining full disk caches.

Patterns

AbstractFactory The abstract factory pattern is used, to allow the client component to initialize the UI without knowing the exact implementation. The client component provides different interfaces (figure) which are implemented by UI components (for example figure and figure).

Composition The composition pattern is used by the BudgetManager (figure). The interface *Budget* specifies common operations possible on composed budgets and atomic budgets. It is implemented in different subclasses for example *SubBudget* and *CompositeBudget*. This is useful, because the controller doesn't have to know if a budget is composed out of different budgets or if it is an atomic budget. This simplifies the implementation of the overview screen and other parts using the budget.

2.2.3 Changelog

1. enumerated requirements (Enumerated Requirements Document)
2. language change to Java (C is impractical for client applications, due to the fact, that personal computers and tablet clients propably won't run the same operating system and therefore additional effort would be required to port the software. In addition, Java is preferable due to it's UI features (e.g. JFC framework) and the UI will have the same look, no matter which client is used)

2.3 Benchmarks & Visualizations

2.3.1 Benchmarks

Traversal-based Queries

2.3.2 Visualizations

Total Accesses per Query per Method

Access sequence