View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by University of Waterloo's Institutional Repository

Gromit
An In-Memory Graph Database

by

Yunling Cui

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2017

© Yunling Cui 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This work presents the implementation of an in-memory graph database management system called *Gromit*. This graph database represents large and complex networks using labelled property graphs, and encodes semantic information in property lists of the vertices and edges. *Gromit* uses a vertex-edge graph model and represent both vertices and edges as entities of the graph. Edges are stored in a doubly linked list manner in main memory. We implement breadth-first traversal and depth-first traversal to retrieve data for queries. This database supports concurrency and implements locking mechanisms for transaction management. We deploy two benchmark suites from social network domain to evaluate our implementation. These are GDBench and LDBC.

Acknowledgements

I would like to thank my supervisor Professor Hiren D. Patel for his guidance, support, and encouragement. This work would not be possible without his help. I would also like to thank my colleages Nivedita Sritharan, Mohamed Hassan, and Anirudh Kaushik for their invaluable suggestions on this work. I would like to thank my readers, Professor Wojciech Golab and Professor Mahesh V. Tripunitara.

Table of Contents

List of Tables viii

List of Figures ix

1 Introduction 1

2 Background 4

2.1 Database Management Systems 4
2.2 Graph Database Management Systems 5
2.2.1 Property Graph 5
2.2.2 Graph Representation 6
2.2.3 Data Storage 7
2.2.4 Graph Traversal 8
2.3 Transaction Management 8
2.3.1 Data Consistency Model 9
2.3.2 Concurrency Control Mechanism 9

3 Related Work 12

3.1 Graph Database Management System 12
3.1.1 Graph Storage 12
3.1.2 Graph Data Layout 13
3.2 Transactions 15

4 *Gromit* 17
4.1 Graph . . . 17
4.1.1 PropertyList . . . 18
4.1.2 Vertex . . . 19
4.1.3 Edge . . . 19
4.1.4 Fixalloc . . . 20
4.2 Traversal . . . 21
4.2.1 Breadth-First Traversal . . . 21
4.2.2 Depth-First Traversal . . . 23
4.2.3 Visitor . . . 24
4.3 Query . . . 26

5 Transaction Management 29
5.1 Transaction . . . 29
5.2 Lock . . . 30
5.2.1 Granularity . . . 30
5.2.2 Protocol . . . 31
5.2.3 Behavior . . . 31
5.3 Deadlock . . . 34
5.3.1 No-Wait . . . 34
5.3.2 Wait-Die . . . 34
5.3.3 Wait-With-Deadlock-Detection . . . 34
5.4 Transaction Management . . . 35

6 Benchmark 37
6.1 GDBench . . . 37
6.2 LDBC . . . 39
6.3 *Gromit* on Micro-architectural Simulators . . . 42

7 Summary and Future Work 44

References 46

List of Tables

2.1 *Person* Table . . . 5
2.2 *University* Table . . . 5
2.3 *Country* Table . . . 5

4.1 Variables and Functions for Fixalloc . . . 20
4.2 A Filter Example . . . 25
4.3 A Query Example . . . 27

5.1 *Transactions* Table . . . 30
5.2 Changes in Fields of Edges . . . 33
5.3 Changes in *NextEdge* of Vertices . . . 33
5.4 Locking Behaviors in *Gromit* . . . 33

6.1 Query Set in GDBench . . . 38
6.2 Query Set in LDBC-SNB . . . 41
6.3 Simulation Configurations and Tool Versions . . . 42
6.4 Simulation Result . . . 43

List of Figures

2.1 A Property Graph Example 6
2.2 A Deadlock 11

4.1 Graph Diagram 18
4.2 Vertex 19
4.3 PropertyList 19
4.4 Edge 19
4.5 Traversal Diagram 22
4.6 Query Diagram 27

5.1 Lock Diagram 31
5.2 A Property Graph Example 32
5.3 TransactionManagement Diagram 36

6.1 GDBench schema 38
6.2 LDBC-SNB schema 40

7.1 Architecture Diagram 45

Chapter 1

Introduction

Graph data structures are commonly used to represent complex relationships. They are used in software frameworks for developing modern applications such as in social networks, recommendation systems, and fraud detection. These applications not only involve in various entities, but also requires perspective on connections between data. Therefore, graph is a generic data structure to represent information in these applications. Graph computing, therefore, involves bringing linked data into main memory and analyzing entities and their connections. Applications such as social networks have large volume of data, frequent data accesses and updates, and various data sources and types, which resemble that of big data. There have been a lot of computing models proposed to process large scale of data, such as Bulk Synchronous Parallel [36] model for parallel computing, and MapReduce [26] model for distributed parallel computing. However, how to store and process dynamic graphs with relationships between data efficiently captured is still a problem to explore.

One approach to implement support for graphs involves using existing technology such as those offered by Relational database management systems (RDBMSs). These are conventional databases to deal with graph analytics workloads. RDBMSs are ubiquitous in enterprise systems for their robust and mature technology that has been developed over decades [29]. They hold structured data in tables with references to connected data in predetermined columns. Those references are key attributes of the referred records. To explore relationships between data, one has to do JOIN operations: retrieving related tables for each reference present in data entries. If data are strongly connected, one has to retrieve multiple tables from storage and then search and match references from each table. This operation is memory-intensive and can add exponential cost in terms of processing time. A response to this observation has been to develop native stored graph databases (GDBs). Graph databases are database systems that build connections into data structures and

represent information in the form of graphs. GDBs allow direct accesses from one piece of data to its related ones since each node of the graph stores relationships itself. This reduces performance overhead of the JOIN operation. Modern GDBs such as Neo4j [13], Titan [16], and Sparksee [14] have emerged to process complex relationships in graphs. These database systems usually support a flexible data model that can manage a variety of domains. To respond to graph analytics queries such as social network recommendation, GDBs usually need execute graph algorithms to retrieve information. These algorithms include general traversals such as breadth-first search (BFS), depth-first search (DFS) and other specific algorithms such as single-source shortest path (SSSP) and strongly connected component (SCC).

In-memory GDBs are designed to shorten response times by avoiding expensive, but frequent input/output operations during graph processing. Those GDBs use main memory to store the data, and usually support higher transaction rates [33] compared with disk-based ones. To allow large graph datasets to fit entirely in main memory, one can enlarge the size of main memory or distribute datasets into multiple computing nodes. Since memory prices continue to decline, it is practical to store data in computing systems with terabytes of memory [33]. Distributed GDBs such as Titan [16] have been implemented and incorporated in enterprises in order to process large volume of graph data.

GDBs usually execute multiple queries concurrently using multithreading to support high transaction rate. Traditional locking has been implemented as the concurrency control method in database systems such as Neo4j [13, 28]. Locks can be applied to different granularities of the graph such as graph level, component level, and vertex level. Some GDBs have locks on vertex and relationship level, such as Neo4j, and this provides the opportunity for optimization with finer granularity of locks.

It is also noticeable that most graph databases have backends implemented in Java [13,16]. These databases run on Java Virtual Machine with garbage collection, which makes it difficult to run applications on simulators. However, there has been increasing amount of research work on hardware designs to accelerate graph processing, such as hardware prefetcher design [27] and processor architecture design [18]. These works require a graph database as the application to run on top of simulators, such as Sniper [23], that allows for micro-architecture exploration. A simulator-friendly graph database is demanded to fill the role of memory-intensive application for simulation.

We design and implement a graph database backend written in C++ to serve as the purpose of simulation. We present this in-memory graph database called *Gromit* that stores social networks graphs. This GDB treats both vertices and edges as entities and stores semantic information as attributes in each entity. *Gromit* uses two-phase locking as

the concurrency control method and sets locks on a finer-grained level: fields of vertices and edges. We implement benchmarks from GDBench [20] and LDBC [9] using both storage and processing engines from *Gromit*. *Gromit* is implemented in C++ and hence allows memory manipulation and management.The key reason to build this graph database is to allow for micro-architecture exploration and hardware designs. This backend is accessible online [8] for researchers to do hardware exploration.

Chapter 2

Background

This chapter introduces a general database management system (DBMS) first and then focuses on graph DBMSs. This chapter also describes transaction management (TM) and widely used concurrency control mechanisms (CCMs) in database systems.

2.1 Database Management Systems

A database is an integrated collection of data. A DBMS is a software package that manages this data. The relational model of data is the most commonly used one today. This model stores relation in the form of table with a set of rows and columns. A DBMS employs storage engines to create, read, update or delete this collection of tables while maintaining relation between data. Although called relational DBMS (RDBMS), this model of database does not store relationships explicitly. If one wants to retrieve information involving more than one type of relation, several tables have to be searched for this inquiry, known as query in DBMS. An example is shown in Table 2.1. Table 2.1 has records of two students *Sam* and *John* who study at the *University of Waterloo*. To answer the query from *Sam* that *Who have been to UK and currently study at Univeristy of Waterloo*, we have to search Table 2.1 for records of persons and join this Table with table of *University* (Table 2.2) and table of *Country* (Table 2.3) to see if this person's country and university both satisfy the criteria. When a query involves more relationships, database has to join more tables for information. For each table, there may be thousands of records that meet one of the requirements, therefore millions of records may have to be scanned for one query.

To simplify this type of search, graph DBMS is implemented for data with complicated relationships. We focus on introducing graph DBMS in section 2.2.

ID	Name	Age	Gender	Country	University
0	Sam	19	Male	4	2
1	John	21	Male	3	2

Table 2.1: *Person* Table

ID	Name	City	Country
2	University of Waterloo	Waterloo	Canada

Table 2.2: *University* Table

ID	Name	Capital City
3	Canada	Ottawa
4	UK	London

Table 2.3: *Country* Table

2.2 Graph Database Management Systems

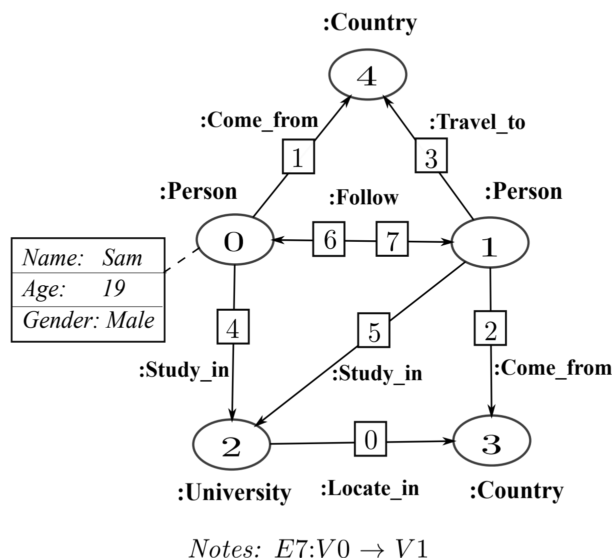
Graph processing frameworks such as *Pregel* and *Graph500* are well-known for their high performance. These frameworks process large volumes of data in parallel and therefore, have high throughput. They do analytical computation on graphs and are mostly optimized for read operation. When we have complicated queries with more than read operation, we need a management system such as TM (section 2.3) to support transactional processing. We call a graph processing framework with such a management system a graph DBMS.

In a graph DBMS, the core engine includes designs of graph representation (section 2.2.1), graph structure (section 2.2.2), graph storage (section 2.2.3), and method of operating on data called graph traversal (section 2.2.4)

2.2.1 Property Graph

A graph is a collection of vertices and edges. Vertex is an entity that has incoming and outgoing edges; edge is an object with a head and a tail vertex. We use $G(V,E)$ to denote a graph where V represents the set of vertices and $E \subseteq V \times V$ represents edges. Vertex and edge illustrate basic relation between data but not all semantic information. We can use a property graph to represent such information as well as basic relation. A property graph is a graph whose vertices and edges are associated with attributes, also known as properties.

In addition to properties, vertices and edges can be attached with labels. An example of a property graph with three vertices and four edges is shown in Figure 2.1. This figure converts information in Table 2.1, 2.2 and 2.3 into a property graph. Only properties of vertex 0 are shown in the figure as an example.



Notes: $E7{:}V0 \rightarrow V1$

Figure 2.1: A Property Graph Example

Vertex In Figure 2.1, circles with id 0, 1 and 2 are vertices. *Person* and *University* are vertex labels. *Name, Age, Gender* are vertex properties.

Edge Lines between circles are edges, e.g. edge 1 between vertex 0 and 4. They usually represent relationships between two vertices, such as the edge *Come from* between vertex 0 and 4. Usually the connection has a direction indicating which vertex is the source and which is destination. It is also noticeable that there may be multiple relationships between two objects, for example, vertex 0 and 1 follow each other.

2.2.2 Graph Representation

There are several ways to represent a graph. Here are four common representations.

Adjacency list Adjacency list is the most common graph representation. It consists of an array of vertices, each contains a list of adjacent vertices in an arbitrary order. The number of adjacent vertices for each vertex may not be fixed.

Adjacency matrix Adjacency matrix stores the adjacency information in a matrix. Each value in the matrix indicates whether two given vertices are connected to each other. Adjacency matrices require the same amount of memory space regardless of graph connectivity.

Edge list Adjacency list and adjacency matrix are both vertex-centric representations. As an alternative, we can use edge-centric model, which means edges are also entities in the graph. In this case, graph is represented with a list of all edges. Each edge in the list is a pair of vertices that are connected to each other.

Vertex-edge model Both vertices and edges can be entities in a vertex-edge model. Usually each vertex keeps a record of its incoming or outgoing edges and each edge contains references to head and tail vertices. In this way, all vertices and edges are linked to represent the whole graph.

2.2.3 Data Storage

There are two major media for graph data storage: disk and main-memory.

Disk-based storage Traditionally, databases are stored on disk drives for their large space and low price. Data is formatted in blocks to be read or modified, therefore disk input/output (I/O) time is required to move data from disk to main-memory. With workloads where data are frequently accessed, I/O latency can be a bottleneck for performance. However the benefit is that data is persistent in presence of power failure.

In-memory storage An in-memory database is stored in a computer's main memory and also managed by an in-memory management system. Since this storage avoids data transfer overhead, database performance benefits from fast data access. The problem with in-memory storage is that it suffers from data loss with machine failure or power failure. Besides, the whole dataset has to fit completely in main memory, which is a limit to main-memory storage.

2.2.4 Graph Traversal

The key idea of graph traversals is to explore the graph starting from a given vertex and visit its neighbors via edges. Based on the order of visiting vertices, traditional graph traversals can be classified as breadth-first search (BFS) or depth-first search (DFS). BFS visits all vertices of the same depth before exploring further distance while DFS always explores vertices of the next depth first till leaf nodes (vertices without outgoing edges).

In a practical database, graph traversals tend to not explore the entire graph. Traversals terminate with certain conditions, for example, the explored depth or the amount of visited vertices are satisfied by the queries or certain data objects are retrieved. Therefore, we define the terms breadth-first traversal (BFT) and depth-first traversal (DFT) to generalize graph traversals in a practical graph DBMS. Several graph analytics algorithms are based on BFT and DFT, such as the shortest path between two given vertices, the strongly connected components of a graph.

2.3 Transaction Management

A transaction is a set of operations executed on data such as reading or deleting. Transaction management (TM) includes registering transactions in a global transaction table, allocating resources to transactions, checking transaction status, and deciding if they succeed or not. If a transaction succeeds, it gives back all resources, changes its status, and retires. This set of actions is usually called *commit* in DBMS. If a transaction does not succeed, it has to revert the database into previous state as if this transaction never happens. This is called *abort*.

A database needs TM if it allows more than one transactions to read data or make changes to a database simultaneously. This is because TM can guarantee desired properties required by a database. These properties includes atomicity, consistency, isolation, and durability, which constitute the data consistency model ACID. TM employs a variety of CCMs to guarantee such properties and each method comes with its advantages and disadvantages. Section 2.3.1 introduces various properties and a data consistency model. Section 2.3.2 explains different CCMs and focus on one of them. Section 2.3.2 discusses databases implementing a specific mechanism.

2.3.1 Data Consistency Model

ACID is a known technique to provide databases with data safety. ACID transactions guarantee the following properties:

- **A**tomicity: All actions in one transaction happen, or none happen.
- **C**onsistency: If each transaction is consistent, and the database starts in a consistent state, it ends up in a consistent state.
- **I**solation: Execution of one transaction is isolated from that of other transactions.
- **D**urability: If a transaction commits, its effects persist.

Above properties can also be interpreted as rules for TM to safely execute operations. Performance is often compromised under these constraints. Besides, for some domains and use cases, ACID is more strict than a database really needs to be. Some compromises are made for performance purpose. For example, data can be less consistent and more available. A selection from data consistency models is specific to applications and usually on a case-by-case basis.

2.3.2 Concurrency Control Mechanism

CCMs manage transactions in the way that desired outcome can be generated and optimized performance can be achieved. Two typical CCMs are lock-based protocol and timestamp-based protocol. Lock-based protocol is a commonly used concurrency control method and section 2.3.2 describes the details.

Lock-based Protocol

A database system usually guards data with read and write locks (R-W locks). Read locks, or shared (S) locks, allow multiple transactions to read the same data concurrently and write locks, or exclusive (X) locks, can be acquired by only one transaction at a time. Various locking mechanisms in terms of lock granularity and lock protocol are implemented for TM.

Lock granularity In a graph DBMS, one instance of database includes a graph and a graph can contain one or more graph components. Each component consists of vertex and edge objects and both have multiple fields such as property and label. Locks can be applied to different levels of data: those to a higher level such as graph level are regarded as coarse-grained locks and those in lower level such as property level are fine-grained. With coarse-grained locks, one transaction may obtain X lock on graph, but only make changes to one vertex in a component. In this way, this transaction blocks others that intend to read other graph components, therefore compromising overall performance. If the database set locks on individual vertices, it needs a complicated locking protocol since graph objects are related to each other, and changes to one object may affect another. In brief, coarse-grained locking mechanism is beneficial for its simplicity and a finer one requires more memory space and a complicated strategy, but allows larger concurrency and potential performance gains.

Lock protocol Two-phase locking (2PL) is a widely used lock-based protocol. 2PL, as the name indicates, has two phases: expanding and shrinking phase. Locks are acquired only in expanding phase, and released one by one in shrinking phase. Once a transaction releases any lock it acquires in expanding phase, it cannot request any additional lock. This protocol guarantees serializability, which means the outcome of all transactions is equal to that of all transactions executed sequentially without overlap in time.

Many variants of 2PL exist with various lock semantics to achieve better performance without compromising serializability, including multiple granularity locking as mentioned in section 2.3.2.

Deadlock detection Deadlock happens when two transactions request locks that are held by the other. Figure 2.2 shows a possible deadlock with two transactions and two X locks: transaction i (T_i) acquires X lock 1 (X_1) and transaction j (T_j) acquires X lock 2 (X_2). Now T_i requests X_2 and T_j requests X_1 and a deadlock is incurred in this situation.

Deadlock can be detected with a wait-for graph [19]. A wait-for graph records all transactions that one lock is held by and all locks that trasactions are waiting for. In a wait-for graph, transactions are denoted as nodes. If transaction Ti waits for a lock that another transaction T_j is holding, there is an edge from the node T_i to node T_j. A deadlock exists if any cycle is found in this wait-for graph. In case of a deadlock, a database system stops one transaction from waiting for the lock it requires therefore breaks the cycle.

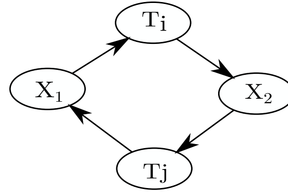


Figure 2.2: A Deadlock

Other Protocols

Lock-based protocol is a pessimistic method because a transaction is blocked until certain condition (for example, lock is free) is satisfied. A deadlock can cause larger performance regression. Timestamp is a method designed to eliminate blocking and deadlock. One classic timestamp-based protocol is timestamp ordering (TO). Each transaction obtains a timestamp that records its starting time. The idea is to allow multiple transactions to proceed until violations of timestamp are detected at the end of any transaction. These transactions have to be aborted and then restarted in case of violations, which incurs an overhead to database systems.

Since both blocking and aborting cause performance regression, other concurrency control mechanisms are explored and utilized in DBMS. Multiversion-concurrency-control (MVCC) is a method used in many databases such as mongoDB [12]. The main idea of MVCC is to create a new version of a database object once the object is modified by a transaction while other transactions can read last relevant versions of this object according to its scheduling rules. This method requires a large amount of extra memory besides those needed to build database systems initially.

A database management system usually implements one or mixed methods from above as a transaction management tool to achieve different performance goals. MongoDB, as mentioned above, is an example that utilizes both MVCC and locking mechanisms at different levels. At the global, database or collection (a group defined by mongoDB) level, mongoDB implements multiple granularity locking. Fo individual storage engines, it allows for MVCC on document-level. This mixed concurrency control mechanism helps mongoDB achieve optimal performance at each level.

Chapter 3

Related Work

3.1 Graph Database Management System

Graph database management systems have been the focus of research in recent years with the increasing popularity of social graphs and web applications. Instances of graph DBMSs including Neo4j [13], Titan [16], Giraph [6], and Sparksee [14] have been widely used for product recommendation systems, fraud detection platforms, and social network systems. To deal with the dramatically increasing workloads and provide high performance, researchers investigate on different aspects of graph DBMSs, such as data structure, memory layouts, and concurrency control mechanisms.

3.1.1 Graph Storage

In this section, I introduce various media that have been utilized to store graph data objects. Traditionally, data objects are spread over disks for the persistence, low price and large space. Neo4j [13] is a disk-based graph database and it saves vertices, edges, and properties in separate files as native stores. Neo4j keeps linked lists of fixed size records and these records are divided evenly into different files. The file system information such as the region of vertex files is stored in a table for fast access.

Similar to Neo4j, System G [15] also stores the vertices and edges separately into two sets of native stores and each store is saved in several disk files. Each vertex is assigned with a unique internal sequential identity (ID) and stored in a table as the key. The value for the corresponding key is the timestamp for the latest version of properties associated with

the vertex. When the database is created, the table is loaded into memory for accessing and querying.

Titan [16] also has a disk-based storage backend. It leverages adjacency list format representation on disk: edges are co-located with the adjacent vertex and can be sorted and maintained in the order customized by applications. Incident edges can be retrieved from the same page on the disk or consecutive pages, which reduces random read from disk.

Researchers observe that workloads of graph DBMSs are data intensive and DBMSs spend a significant amount of time on reading from or writing to file system. Thus disk I/O cost can be a bottleneck for graph DBMS performance and optimization has to be done to reduce data access latency. One simple solution without altering data structures is to cache the most frequently used data objects in main memory.

Neo4j caches file system information and part of graph objects in main-memory. Every disk file is split into several regions and a table recording region addresses is cached. Besides, simplified version of vertex and edge objects are also cached for accelerating graph traversals.

Titan also creates and maintains in-memory caching. Main memory stores hotspots of global adjacency lists and locally traversed subgraphs for reuse in further traversals. Most likely, reusing graph objects can also increases cache hit rate, therefore improving the overall performance.

3.1.2 Graph Data Layout

Data layout is another important factor that can be optimized for high performance. This is because data layout affects the distance of two sequenctial data accesses in memory and shorter distance results in smaller access latency. "Distance" is defined as number of blocks between two data locations. Graph traversals explore the relationship between linked data and decide the order of graph object accesses. Databases attempt to arrange data in the way to shorten distance of two sequential accesses during traversals.

In order to follow graph traversals, Neo4j implements the most generic solution of constructing edge-centric graphs with pointer-chasing pattern on vertices and edges, which are known as nodes and relationships in Neo4j. Each node or relationship record has a fixed length and pre-defined fields. Each node record in the graph model contains a list of "pointers" to the relationship records and properties. The "pointers" are actually the offsets of the address for that data object. Each relationship record also has "pointers"

indicating the source and destination nodes, properties, the next relationship that shares the same endpoint. For example, an relationship with label "follow" has an ID field, one pointer to the source vertex and one to the destination vertex, one pointer to the next relationship that share the source vertex and one for the destination vertex, one pointer to the previous relationship that shares the source vertex and one for the destination vertex, respectively. In addition, these is one field for the property ID, which stores the index to the file to retrieve associated semantic information such as "creation date". In this way, whenever users use traversals, Neo4j uses the addresses encoded in the record to retrieve other connected entities directly instead of looking for data in another data structure. In RDBMS, the traversal is equivalent to JOIN operation, which involves retrieving records from other data storage with operations such as sequential scan. In brief, the direct access in Neo4j eliminates the need of joining two or more tables for RDBMS, which involves long disk I/O latency and expensive search-and-match computations. Therefore, compared with the pure index-based approach in RDBMS, Neo4j gains better performance.

Titan has a different data layout from Neo4j. It deploys the Bigtable [25] data model on a disk-based storage backend. This table is a collection of rows and each row contains the adjacency list and properties of current vertex in the form of cell array. Each cell has a column and value field. For properties, columns record key ID of properties and the value fields store the real value corresponding to the key. For each incident edge, each column stores label ID, direction, a sort key, edge ID and the value stores the real value and other properties. This layout of clustered edges is for efficient retrievals of a subset of edges with a specific type or range in the list. The cell array is identified and sorted by the key, which is the unique ID of each vertex assigned by Titan. The sorted order of vertices helps partition graphs in distributed systems. However, it is noticeable that the number of cells in each row is limited, which means Titan only deals with limited outedges for each vertex.

GraphChi [7] implements a method called *parallel sliding windows* (PSW) [32] to do parallel computations. Inspired by *compressed sparse row* (CSR) storage format, which stores the outedges consecutively as an adjacency list in the file for each vertex, GraphChi implemented a similar storage format called *compressed sparse column* (CSC). CSC stores all incoming and outgoing edges for each vertex in different files. In other word, each edge is stored twice with the source and destination node, respectively. Vertices in the graph are split into disjoint intervals, and edges that share the same destination are sorted by the source vertex and stored in a shard (a partition of data) associated with each interval. The outgoing edges are store in consecutive chunks. Each shard is fully loaded into memory so that the in-memory subgraph can be created. In this way, when PSW moves from one interval to the next one, it slides a window over each of the shard. There are only limited accesses made to these intervals on disk, and data accesses to the same interval are

sequential since the edges are in a sorted order of the source node. There is performance gain with this data layout, however, the limit of PSW is that it only supports asynchronous model computing.

System G [37] also evaluates the effect of different graph data layouts on performance. In order to maximize graph data locality, System G proposes and implements a compact graph representation with *compressed vertex storage* (CVS) format. The CVS uses a few long arrays to represent the vertex index, neighbor index, and edge weight information. This format can be partitioned into segments, therefore, when updates are applied to the graph, only some segments encounter changes and the others remain the same. Data locality for these unchanged segments will remain in cache. System G also exploits data locality with vector-like and list-like layouts, respectively. Results from experiments show that list-like layout has better cache hits and better performance.

3.2 Transactions

A transaction symbolizes a unit of work performed against a database and is treated as independent of other transactions. Multiple transactions can be performed at the same time to achieve concurrency, which helps improve throughput and achieve better performance. Concurrency control strategies have to be implemented to ensure DBMS safety. Locking is the predominate method among traditional RDBMS. Neo4j borrows locking semantics from RDBMS and applies locks to graph data object to ensure atomic update operations. As demonstrated in section 3.1.1, Neo4j mainly has two types of graph data: vertex and relationship. Locks are acquired at vertex and relationship level. Neo4j supports READ_COMMITTED as the default isolation level, which means updates to data are not visible to other transactions until current transction succeeds.

Cong Yan et al. [38] proposes a method to leverage lock contention and improve performance of online transaction processing (OLTP) application. In this paper, authors propose *QURO* as a tool to compile and reorder transaction codes. Operations in a transaction are reordered in the way that least lock conflicts are encountered while the isolation level can also be guaranteed. They also use other techniques to reduce reordering time and overheads of reordering can be minimized. Experiments show up to 6.53x improvement in throughput. Although not specific to main-memory graph databases, this paper demonstrates that reordering queries shows benefits to database performance.

Based on Shore-MT [31] system, authors of the paper [30] propose an optimized locking method called *Speculative Lock Inheritance*. According to their observation, locking

manager has the most contention when processing transactions for a locking-based DBMS. Authors focus on reducing this manager bottleneck and implement the "lock carry-over" optimization: instead of releasing the locks when the transaction is finished, keep the locks until other transactions request them. The successive transaction requests locks from the predecessor transaction, instead of the centralized lock manager. The method is based on the assumption that there exist hotspots in data, which is one of the characteristics of graph DBMS workloads.

Researchers investigate and evaluate various concurrency control algorithms on a main-memory DBMS. Paper [39] claims that concurrency control problem is a bottleneck to scalability of multicore on the same chip for main-memory databases. Authors implements seven concurrency control algorithms including two-phase locking and MVCC and run OLTP workloads with 1024 cores. The results show that all seven algorithms fail to scale to this magnitude for different reasons. Authors propose mixed concurrency control algorithms and software-hardware co-design solutions as future work to this problem.

There have been lots of research activities on optimizing concurrency control mechanisms for databases [33] [38] [30]. Authors of the paper [33] propose multiversioning as a concurrency control mechanism for main-memory databases. They design and implement two MVCC methods, one is optimistic using validation and the other is pessimistic using locking. They also compare the performance of these two methods with that of single version with locking. Experiments show that MVCC schemes achieve better throughput than the single version in the following two scenarios: 1. There exist contentious accesses to data in a small area of storage (hotspot). 2. Transactions only have read operation on data and one single transaction involves a large number of reads (long read-only transaction). However, usually MVCC encounters larger overheads due to assignment of timestamp and existence of multiple versions. Comparison between the optimistic and pessimistic MVCC mechanisms show that the optimistic MVCC method retains higher throughput than the pessimistic one in the same cases.

Chapter 4

Gromit

Gromit is a database that employs a graph as both data structure and computation model. We use Figure 2.1 from the background chapter as an example to illustrate how data are stored in *Gromit*. Section 4.2 describes how information is retrieved through traversals and visitors.

4.1 Graph

Graph is the most generic data structure to represent social networks. *Gromit* uses vertex-edge model to store property graphs of social networks. According to the model, both vertices and edges are entities of a graph. We use individual objects to represent a vertex and an edge and refer to other entities with a pointer instead of index in relational databases. Vertices and edges are created and linked in a doubly linked list manner, which means each object keeps references to its previous and next object that are directly connected. Each vertex and edge contain a property list that encodes semantic information of the graph.

The vertex-edge graph model enables flexible representation of social graphs. Since there is no constraint on either type or number of connections of each entity, or predetermined field in property lists, *Gromit* is able to store dynamic graphs. one can make changes to graph entities such as changing vertex labels, removing relationships, and updating vertex properties. Those operations are mapped to searching and updating data or pointers in a linked list in underlying storage, without consulting or upgrading schema as in relational databases.

In this section, we describe the classes implemented in *Gromit* to storage graphs and the diagram is shown in Figure 4.1. We also introduce the memory allocation method implemented for graph storage called Fixalloc in section 4.1.4. We represent vertex and edge in Figure 4.2 and 4.4 using Figure 2.1 and information from table 2.1, 2.2 and 2.3.

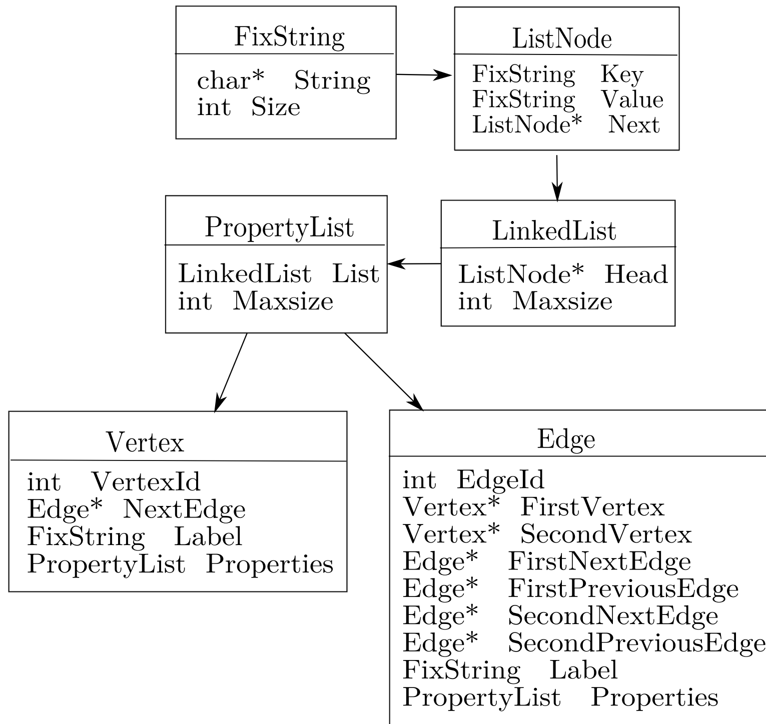


Figure 4.1: Graph Diagram

4.1.1 PropertyList

PropertyList encodes properties of a graph in key-value pairs. Data type of key and value can be dynamically allocated string or the customized type called FixString. FixString is a limited size of char array designed for *Gromit*. Each key-value pair is stored in a node called ListNode and nodes are chained in linked list manner. This linked list is also limited by Maxsize, which is the number of nodes allowed for a property list. One can update

properties by adding, deleting, and changing values of a node within the size limit. Each property list can keep customized key-value pairs. If a value is absent from a pair, the key can be deleted from the list to save space and shorten searching latency.

4.1.2 Vertex

A vertex object has four fields: *ID*, *NextEdge*, *Label* and *PropertyList*. *ID* is an integer value assigned by the database. *NextEdge* stores reference to the first edge that is connected to current vertex. *Label* is initialized when the vertex is created. It keeps information in string format and can be used for filtering vertices from a certain domain. *PropertyList* is a list of key-value pairs, which expressively encode semantic information such as *Name* and *Age*. Vertex with ID 0 and 2 in Figure 2.1 is shown in Figure 4.2. Pointers in figures are represented with $(ID) and prefix V represents vertex, E for pointer to edge and P for reference to property list, e.g.$\$(V1)$ represents the pointer to vertex with ID 1, $\$(E0)$ represents the pointer to edge with ID 0 and $\$(P_v0)$ is the reference to property list (Figure 4.3) of vertex 0.

ID	NextEdge	Label	PropertyList
0	$\$(E1)$	Person	$\$(P_v0)$
2	$\$(E0)$	Person	$\$(P_v2)$

Figure 4.2: Vertex

Key	Value
Name	Sam
Age	19
Gender	Male

Figure 4.3: PropertyList

ID	FirstVertex	SecondVertex	FirstNextEdge	FirstPreviousEdge	SecondNextEdge	SecondPreviousEdge	Label	PropertyList
0	$\$(V2)$	$\$(V3)$	$\$(E4)$	NULL	$\$(E2)$	NULL	Locate in	$\$(P_e0)$

Figure 4.4: Edge

4.1.3 Edge

In *Gromit*, each edge has nine fields and these fields are shown in Figure 4.4. *ID* is an integer value. *FirstVertex* is a pointer to the head vertex of this edge and *SecondVertex* is a

pointer to the tail vertex. If there is no direction associated with this edge, these two fields just indicate which vertex shows up first when this edge is created. *FirstPreviousEdge* and *FirstNextEdge* point to the previous and the next edge that are connected to this *FirstVertex*, and *SecondPreviousEdge* and *SecondNextEdge* point to the previous and the next edge that are connected to *SecondVertex*. *Label* field stores edge type in a string format and is used in the same way as that of a vertex. *PropertyList* may keep information such as creation date of this edge. We show an example of edge 0 in Figure 4.4. *NULL* in the figure represents no reference.

Edges are linked with pointers to allow direct accesses to the next entities. Once can refer to the *SecondVertex* pointer for the destination vertex of an edge without checking tables with an index. Information is available once the object is accessed by referring to a pointer. Edges are chained in doubly linked list manner to facilitate deletion in *Gromit*. After deleting an edge, one can rechain edges without traversing the whole list by referring to *FirstPreviousEdge* and *SecondPreviousEdge* pointers.

4.1.4 Fixalloc

In a program, constructed objects with operator *new* are placed in unused memory blocks and usually these blocks are not contiguous in memory. Fixalloc is the method that allocate sequential memory blocks for vertices and edges before constructing graphs. The Table 4.1 shows the main variables and functions used in Fixalloc technique.

Variables
char* VertexStartAddr char* EdgeStartAddr char* VertexNextAddr char* EdgeNextAddr
Functions
bool allocVertexMemory(int numberOfVertices) bool allocEdgeMemory(int numberOfEdges) Vertex* new(VertexNextAddr) Vertex() Edge* new(EdgeNextAddr) Edge()

Table 4.1: Variables and Functions for Fixalloc

Before building a graph, Fixalloc asks for the total number of vertices and edges from the input readers. To allow for insertions of new vertices and edges, the parameters of total

numbers can be enlarged with ratio range from 1.1 to 1.3. Fixalloc allocates two memory regions and then place newly created objects sequentially within the regions. In this way, all vertices and edges are next to each other in underlying storage. When one cache block is replaced with memory from vertex region, more vertex objects are brought into cache.Processors encounter cache hits when asking for accesses to those vertex objects. The same logics applies to edge objects as well. Therefore, Fixalloc can improve cache performance and accelerate graph processing. Besides, Fixalloc is pluggable to graph storage and it can be turned off if not required.

4.2 Traversal

Algorithms such as *breadth-first search* and *depth-first search* terminate when every vertex in a connected graph component is explored. However, in real applications, it is not necessary to traverse each vertex in a graph. For example, social network recommandation may terminate visiting once it retrieves friends of your friends and only recommands you those "close" friends. In *Gromit*, traversals realize this "termination-on-demand" by checking termination conditions at different points of traversals, which are called event points. These termination conditions are set up in one or a set of filters in traversals. Those filters have functions that check properties, labels or directions of vertices and edges and return boolean values to indicate whether current object satisfy conditions. To allow traversals to terminate at various points, we have an instance of visitor class to check termination flags during traversals. The diagram for traversals are shown in Figure 4.5. Section 4.2.1 and 4.2.2 introduce these two traversal methods separately and section 4.2.3 describe how termination check works during traversals.

4.2.1 Breadth-First Traversal

The main idea of Breadth-First traversal (BFT) is to visit vertices of the same depth before it moves to the next level and store the next vertex to be visited in the queue. BFT uses two data structures for traversing: a queue and a map. The queue in BFT is called *VertexQueue* and it stores vertices to be visited. The map is called *VisitedMap* and it records if a vertex is visited or not. BFT always picks the first vertex in this queue and starts visiting from this vertex till the queue is empty or other conditions are satisfied. In default, BFT traverses each vertex only once. Since each vertex can have more than one vertices connecting to it, it is possible for BFT to enter a loop and visit vertices more than

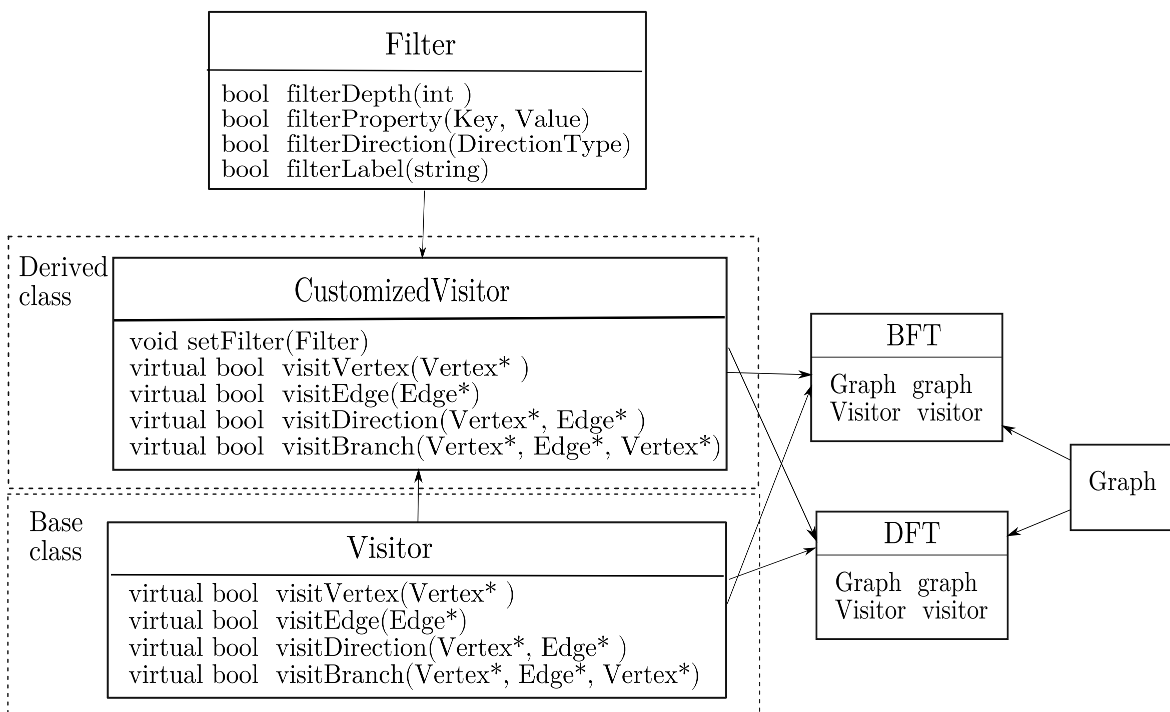


Figure 4.5: Traversal Diagram

once. BFT checks against this map whether the vertex has been visited before it is pushed into the queue. List 4.1 shows part of BFT function in *Gromit*.

```
1   ... ... ...
2   /// Variables declarations are not shown
3   VertexQueue.push(StartVertex);
4   VisitedMap.insert(StartVertex, false);
5   while (!VertexQueue.empty()) {
6     CurrentVertex = VertexQueue.front();
7     VertexQueue.pop();
8     /// Set to visited.
9     VisitedMap[CurrentVertex] = true;
10    auto NextEdge = CurrentVertex->getNextEdge();
11    while (NextEdge != nullptr) {
12      /// Get the target node.
13      NextVertex = NextEdge->getTarget(CurrentVertex);
14      if (VisitedMap.find(NextVertex) == VisitedMap.end()) {
15       /// Queue up the target for visitation
16       VertexQueue.push(NextVertex);
17       VisitedMap.insert(NextVertex,false);
18      }
19      NextEdge = NextEdge->getNextEdge(CurrentVertex);
20    }
21  }
22  ... ... ...
```

Listing 4.1: Function BFT

4.2.2 Depth-First Traversal

Depth-First traversal (DFT) implements a recursive function to visit graphs. It starts with a given vertex and visits the first adjacent vertex. DFT keeps visiting the undiscovered neighbor of current vertex till a leaf node or till other requirements are met. These requirements may include that current vertex has been visited once or path depth is satisfied. Then, DFT backtracks to the last vertex it visited and traverse other undiscovered paths via that vertex. Since DFT goes back to the last vertex once the recursive functions returns, there is no need to record the to-be-visited vertices in a queue. It is necessary to implement *visitedMap* the same way as that of BFT to avoid infinite loops. In DFT function, the start vertex is visited first and then a recursive DFT function is called on adjacent vertices of this vertex. DFT returns when all reachable vertices are visited in the graph. List 4.2 shows the recursiveDFT used in DFT function.

```
void recursiveDFS(GraphType & Graph
                , VertexType & Vertex
                , map VisitedMap) {
  ... ... ...
  /// Variables declarations are not shown.
  VisitedMap.insert(Vertex, true);
  while (NextEdge != nullptr) {
    NextVertex = NextEdge->getTarget(Vertex);
    /// Check if this vertex is visited.
    if (VisitedMap.find(NextVertex) == VisitedMap.end()) {
      recursiveDFS(Graph, NextVertex, VisitedMap);
    }
    NextEdge = NextEdge->getNextEdge(Vertex);
  }
}
```

Listing 4.2: Function recursiveDFS

4.2.3 Visitor

Gromit implement a class called *Visitor* to walk into the graphs, collect graph information, and terminate visiting at some point of traversals. Each visitor object contains a list of functions that can be inserted into traversers such as *BFT* and *DFT*. Traversers check return values of those functions to decide whether to finish visiting. List 4.3 shows how a visitor inserts termination functions at different points of BFT.

As shown in List 4.3, most of these functions in visitor class return a boolean value, which indicates if traversals should be terminated or if current object (vertex or edge) should be visited again. For example, function *visitDirection()* checks if the next edge is incident to current vertex and decides to traverse in this direction or not. Some functions may just collect information about vertex or edge of this point. For example, function *visitStartVertex()* is inserted when the starting vertex is given to traversals. Visitors can mark this vertex e.g. as the source node for shortest path. Other functions employ similar ideas as these two function examples.

```
1  ... ... ...
2  VertexQueue.push(StartVertex);
3  Visitor.visitStartVertex(StartVertex);
4  VisitedMap.insert(StartVertex, false);
5  while (!VertexQueue.empty()) {
6    CurrentVertex = VertexQueue.front();  VertexQueue.pop();
7    bool VertexMatch = Visitor.visitVertex(CurrentVertex);
8    if (VertexMatch) return;
9    VisitedMap[CurrentVertex] = true;
10   auto NextEdge = CurrentVertex->getNextEdge();
11   while (NextEdge != nullptr) {
12     NextVertex = NextEdge->getTarget(CurrentVertex);
13     bool RevisitFlag = Visitor.discoverVertex(NextVertex);
14     bool BranchMatch = Visitor.scheduleBranch(CurrentVertex,
15       NextEdge, NextVertex);
16     bool TypeMatch =  Visitor.scheduleEdge(NextEdge);
17     bool DirecMatch= Visitor.visitDirection(NextVertex, NextEdge);
18     if (BranchMatch) return;
19     if (VisitedMap.find(NextVertex) == VisitedMap.end()
20         || RevisitFlag) {
21       GraphVisitor.scheduleTree(CurrentVertex, NextEdge, NextVertex);
22       if (TypeMatch && DirectionMatch) {
23         VertexQueue.push(NextVertex);
24       }
25 ... ... ...
```

Listing 4.3: BFT with Visitor functions

Gromit implements *Filter* class to accept different termination conditions and provide feedback to the visitor that employs the filter. Part of filter functions is listed in diagram 4.5. As shown in the diagram, one can configure directions, properties, labels, and other limits to instruct the visitor where to terminate. For example, for the query asking for webpages that friends of person P like, one can set the parameters as in Table 4.2 as below.

Filter 1	filterLabel("Friend"); filterDirection(OUT);
Filter 2	filterLabel("Like"); filterDirection(OUT);
Filter 3	filterDepth(2)

Table 4.2: A Filter Example

Once the visitor set all filters, it uses the information from each filter to check depth of each vertex, label and direction of each edge that BFT or DFT traverses. In this way, traversers avoid unnecessary visits to other graph objects and return as soon as the depth

reaches the limit. Since visitor class is extendable, one can have customized visitor class descended from the base and override functions for different types of work.

4.3 Query

A query is an inquiry into the database that asks for specific information about the database. Usually queries are requests from users that retrieve data such as "the most popular games that my friends like". A query may involved in operations such as creation, reading, updating, and deletion (CRUD). Those operations are operated on data or in certain positions of a database. In *Gromit*, data can be retrieved through traversals. As introduced in 4.2.3, one can provide different parameters to filters and customize visitors for traversals to reach specific goals. *Gromit* also has utility functions that processing information during traversals. These functions include checking creation date of an edge, checking label of a vertex, checking direction of an edge and other functions.

Besides traversals, another way to retrieve data in *Gromit* is to use index. *Gromit* provides a pluggable and extendable class called *Index* that collects, builds, and manages indices on graph objects such as vertices and edges. Index class implements functions to group vertices by their labels and build index on semantic identification of vertices. For example, to answer the query that asks for deletion of webpage with URL, this index checks vertex map with "Label=Webpage; Key=URL" and return the vertex that has the URL satisfying the query.

To answer queries with different goals and limits, one can also customize query class to utilize traversals or indices. Each query can decide on traversal methods, keys to build index, and other details. We show the diagram of query classes in Figure 4.6. We show how to customize queries with an example in Table 4.3.

First we can choose to use breadth-first traversal and set person P as the start vertex for BFT. Then we set three filters with the same configurations shown in Table 4.2 and pass them to a visitor named WebVisitor. The next step is to customize this WebVisitor. Function *visitVertex()*[1] checks current depth and returns true once BFT passes depth limit, which is three in this case. In function *discoverVertex()*, we set the return value as false in base class of visitor. Therefore, vertices can only be visited once in BFT with help of *VisitedMap* and there is no need to override this function. Function *scheduleEdge()* checks edge labels and return true if current edge has label "Friend" at depth 1 and "Like" at

[1]Parameters in functions are ignored for simplicity. This applies to all functions metioned in this paragraph.

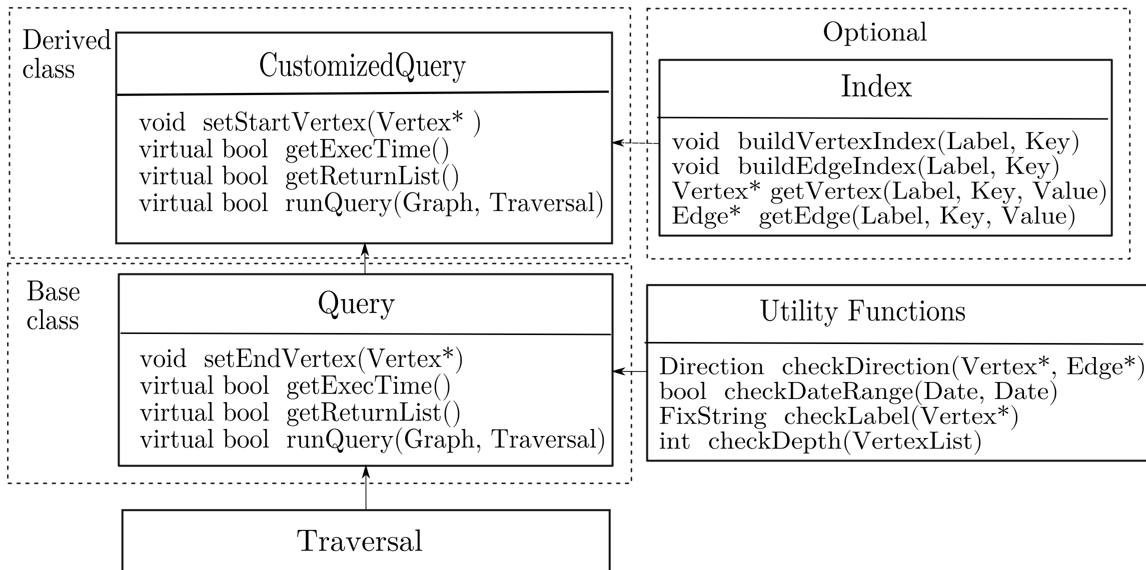


Figure 4.6: Query Diagram

Goal	Retriev all webpages liked by friends of a given person P
Input	Vertex of Person P , Graph G
Output	List of Webpage URLs
Variable	*Function*
Traverser	BFT(G, P, WebVisitor)
Filters	Table 4.2

WebVisitor	void setFilter(Filters);
	bool visitVertex(Vertex*);
	bool scheduleEdge(Edge*);
	bool visitDirection(Vertex*, Edge*);

Table 4.3: A Query Example

depth 2. Functions *visitDirection()* works in a similar way but checks directions instead of edge labels. Those return values from above two functions are used to decide if we should visit the adjacent vertices or not. If both label and direction of current edge satisfy the conditions, the *NextVertex* is pushed to the queue to be visited later. Otherwise, we skip

this vertex and visit the next neighbor. This statement can be seen from line 22 in List 4.3. For other functions such as *visitStartVertex()*, we can use the default settings.

With the help of filters, visitors and traversals, query class can be extended to accommodate different inquiries to *Gromit*. We describe the queries that have been implemented in *Gromit* in chapter 6.

Chapter 5

Transaction Management

In this chapter, we will introduce the concept of a transaction in section 5.1 and the locking mechanism implemented in *Gromit* for transaction management. We will focus on details of locking storage in section 5.2.1, locking protocol in section 5.2.2, locking behavior in section 5.2.3, and how to solve deadlock problem associated with locking in section 5.3. In section 5.4, we describe the transaction management implementation in *Gromit*.

5.1 Transaction

A transaction is a logical unit of operations executed on a database. An example is shown in Table 5.1. t_i is a transaction that updates data X_1 to 20 first, and then reads data X_2. Each transaction in *Gromit* has a unique identifier *TID*. Once a transaction begins, it may have status of either *Processing* or *Rollback*. If a transaction succeeds, it *Commit*s; otherwise, it *Abort*s.

The effect of a transaction is either all or none, which means a transaction finishes all its operations or changes nothing. In other words, a transaction has to be atomic in a database. Table 5.1 shows an example of two transactions and each executes two operations on the same database. The initial value of X_1 is 0 and X_2 is 10. If t_i has to abort, its effect of updating X_1 into 20 has to be removed, which means X_1 has to rollback to initial value of 0.

A modern design of databases support multiple transactions at a time. When two transactions overlap in both data set and execution time, they may operate on the same data without notifying each other. One transaction may update the data based on its own

$X_1 = 0$, $X_2 = 0$

t_i	t_j
$\mathrm{W}(X_1) \leftarrow 20$	
	$\mathrm{W}(X_2) \leftarrow 0$
$\mathrm{R}(X_2)$	
	$\mathrm{R}(X_1)$

Table 5.1: *Transactions* Table

local state while the other transaction changes the data into another state. In the above example, t_j will read X_1 as 0 instead of 20, even though t_j reads the data later than the update according to timeline. Therefore, transaction management is required to prevent a database from the above situation.

5.2 Lock

Locking is a conventional method used in transaction management. This section introduces lock storage and lock behaviors implemented in *Gromit*.

5.2.1 Granularity

In *Gromit*, lock storage is optional for a graph. Once locking is enabled by setting the flags in configurations, lock pointers are embedded in vertex and edge storage. Locks can be constructed and placed in memory either during or after building graphs. Those lock pointers can be directed to where locks are stored.

Locks are acquired on field level in *Gromit*. Figure 4.2 and 4.4 show that each vertex and edge has several fields. Lock of one field may not cause conflict of another. For example, if the *NextEdge* lock in a vertex is held by one transaction, other transactions can acquire lock on *PropertyList* in the same vertex if they are not violating any rules introduced in section 5.2.3.

Since vertices and edges have both common and identical fields, *Gromit* has a base class of *Lock* to process shared fields and descended classes *VertexLock* and *EdgeLock* for different fields. The diagram of lock is shown in Figure 5.1.

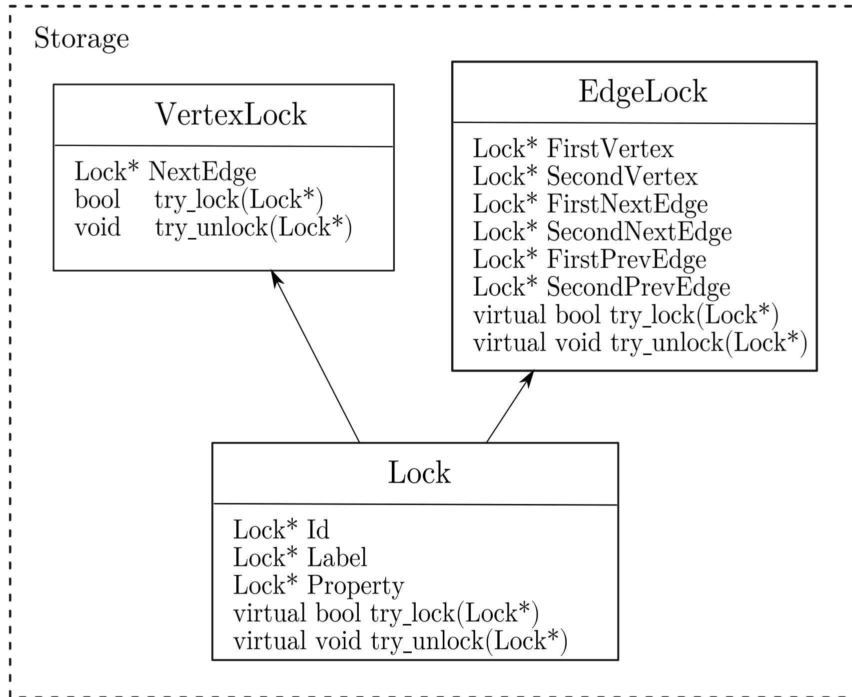


Figure 5.1: Lock Diagram

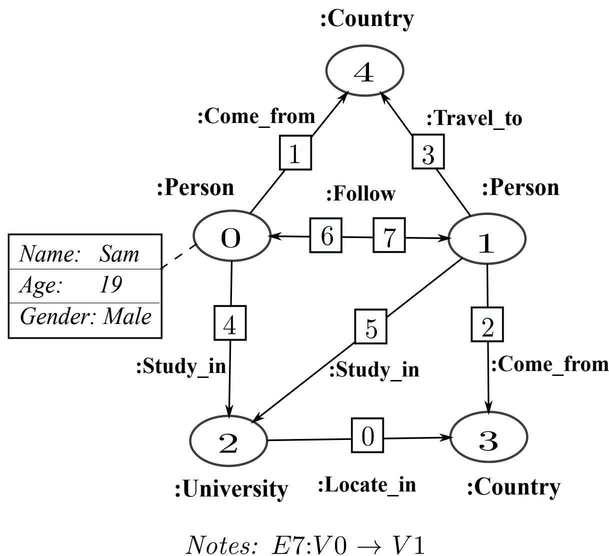
5.2.2 Protocol

Two-phase locking is used as the protocol in *Gromit*. In *expanding* phase, a transaction has to acquire a read lock on an object before reading it and a write lock before modifying it. Once a transaction releases a lock, it enters the *shrinking* phase and no more locks can be acquired. In *Gromit*, a transaction modifies data only after *expanding* phase and this transaction is not allowed to abort after it finishes requesting locks.

5.2.3 Behavior

There are four types of possible operations on a graph database, *Create*, *Read*, *Update*, and *Delete*. Each of these operations can be executed on a vertex or an edge. Specifically,

Read and *Update* can be executed on any field of a vertex and an edge; *Create* and *Delete* can be executed only on vertex and edge level. This is because all fields are reserved when *Gromit* creates a vertex or an edge. If there is no information associated with that field, *Gromit* keeps the field and stores a *NULL* pointer in it. We use Figure 5.2 to show an example of *Create* and *Delete*. Note that Figure 5.2 is the same as the one from section 2.



Notes: $E7{:}V0 \rightarrow V1$

Figure 5.2: A Property Graph Example

In this example, we create vertex V_4 and add its edges E_1 and E_3. Next we create edge E_7 and add it to graph. Then we delete edge E_7 and vertex V_4 in reverse order of creating. Table 5.2 and 5.3 show storage of edges and vertices in the graph example before add V_4. In Table 5.2, "Step" in the first column indicates the order these vertices/edges are created. In Table 5.3, cells store changes in *NextEdge* field of corresponding vertex after each step. An empty cell simply means no change in that field. We do not show *Label* and *PropertyList* fields in either vertices or edges in this example. For simplicity, we use NE for NextEdge, FNE for FirstNextEdge, FPE for FirstPreviousEdge, SNE for SecondNextEdge, and SPE for SecondPreviousEdge in following tables.

In order to *Create* and *Delete*, a transaction has to acquire certain write locks on that object. Table 5.4 shows what locks to request when operating on vertex V_4 and edge E_7. Notice that edge E_7 has been deleted from the graph when we delete Vertex V_4.

If a transaction is unable to acquire a lock for an object, it is forced to wait until this lock becomes available. However, there is no guarantee that this transaction can get the

Step	Edge ID	FNE	FPE	SNE	SPE
1	6	NULL	E_5	NULL	E_4
2	4	E_6	NULL	NULL	E_5
3	5	E_6	E_2	E_4	E_0
4	2	E_5	NULL	NULL	E_0
5	0	E_5	NULL	E_2	NULL

Table 5.2: Changes in Fields of Edges

Vertex ID	1	2	3	4	5
0	E_6	E_4			
1	E_6		E_5	E_2	
2	NULL	E_4	E_5		E_0
3	NULL			E_2	E_0

Table 5.3: Changes in *NextEdge* of Vertices

	Create	Delete
Vertex (V_4)	V_4 :*ID*	V_4: *ID* E_3: *ID* E_1: *ID* E_2, E_1: *FPE* *SPE*
Edge (E_7)	E_7: *ID* V_0: *NE* V_1: *NE* E_1, E_3: *FPE* *SPE*	E_7: *ID* V_0: *NE* V_1: *NE* E_1, E_3: *FPE* *SPE*

Table 5.4: Locking Behaviors in *Gromit*

lock eventually. This is because this database system may incur a deadlock, as shown in Figure 2.2. In section 5.3, we will introduce the methods implemented in *Gromit* to solve deadlock problems.

5.3 Deadlock

In *Gromit*, we have three methods [21] to resolve a deadlock, and they are *No-Wait*, *Wait-Die*, and *Wait-With-Deadlock-Detection*.

5.3.1 No-Wait

As the name suggests, a transaction never waits for a lock. Once a transaction fails to acquire a lock, it aborts right away and then restarts, even if waiting may not result in a deadlock. Therefore, this is a deadlock prevention technique. This "cautious" scheme is easy to understand and also simple to implement in *Gromit*. *No-Wait* method is expensive in the following scenario: If multiple transactions request lock on the same object, it may cause data contention in this database system. One transaction may not be able to obtain the lock if there is conflict in locking behaviors. Transactions abort whenever they come to this "hot spot" of data. It is worse if this transaction involves a large volume of data and has to restart several times.

Gromit implements *NoWait* with *shared_time_mutex* from C++ standard library. There is no guarantee that one transaction can get a lock even if this lock is available. Therefore, in implementation of *No-Wait*, we allow more than one locking attempt for each lock to compensate for this locking uncertainty.

5.3.2 Wait-Die

This method assigns priority to each transaction. Younger transactions have larger *TID* values but lower priority. If a younger transaction is waiting for a lock that is held by an older one, this younger transaction has to abort. Otherwise, it waits until this lock becomes available. In *Gromit*, an aborted transaction restarts with the original *TID* to ensure that new transactions have the chance to get all locks [33]. Similar to *No-Wait*, this method may abort transactions that do not form a deadlock, therefore causing performance degression. We also have limited locking attempt for the same reason stated in 5.3.1.

5.3.3 Wait-With-Deadlock-Detection

Both *No-Wait* and *Wait-Die* are deadlock prevention schemes. They "assume" deadlocks exist without a confirmation whenever a transaction fails in first few attempts to acquire a

lock. Deadlock-detection is the scheme to check and confirm a deadlock. *Gromit* constructs a wait-for graph [19] of transactions to check for cycles (i.e. deadlocks). Each node in the graph represents a transaction and each edge represents "wait-for" relationship between two transactions. If transaction t_i is waiting for a lock held by transaction t_j, we draw an edge from t_i to t_j. In this way, if transactions form a circular wait, which is a cause of deadlocks, a cycle can be detected from this wait-for graph. When a deadlock is found, the system chooses one transaction to abort and then restart it. In *Gromit*, the deadlock detector chooses to abort the last transaction whose request results in the deadlock. In this way, the system never miss-predicts a deadlock, therefore avoids unnecessary aborts.

Deadlock detection also comes with disadvantages. First, this method is timeconsuming since it has to periodically check the wait-for graph for cycles. Second, since deadlock detector is processing a dynamic wait-for graph in a database, it may restart a transaction after the circle has been broken by any transaction in this previously formed loop.

All three methods described above have their own advantages and disadvantages. One can choose the best method by analyzing length of queries and the amount of resources needed by queries.

5.4 Transaction Management

In *Gromit*, each transaction records Id, status of current transaction, list of locks acquired, and other information needed to execute a transaction. TransactionManager class stores all transactions in a table for deadlock detection. LockManager is the class that implements deadlock detection techniques. LockManager has access to transaction table from TransactionManager and executes detection functions such as checking priorities of transactions and detecting deadlocks. Figure 5.3 shows the diagram of classes implemented for transaction management.

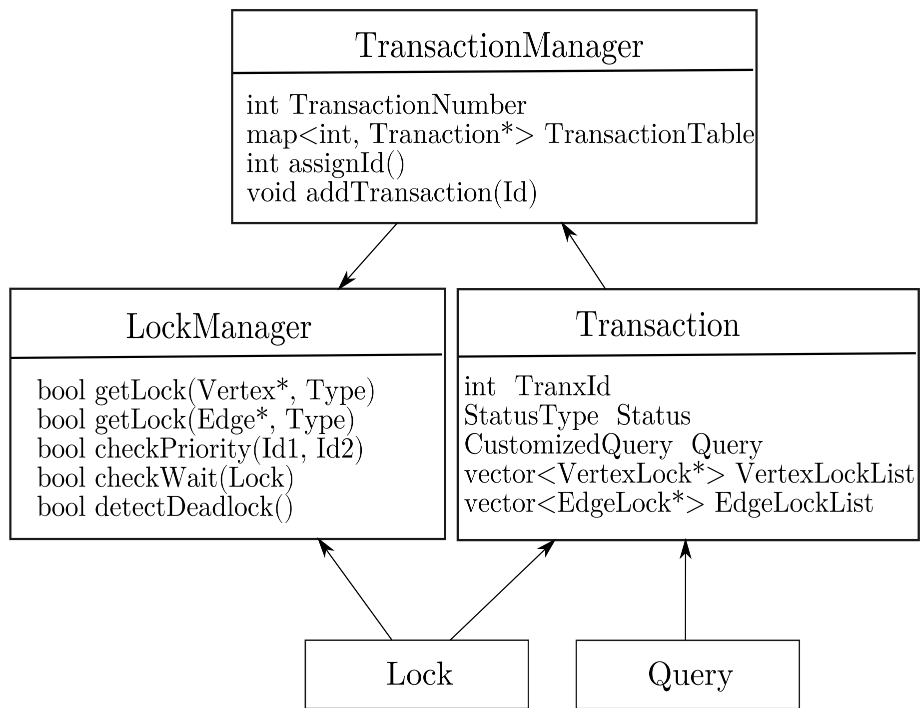


Figure 5.3: TransactionManagement Diagram

Chapter 6

Benchmark

This chapter describes two widely used benchmark suites from social network domain: GDBench and LDBC. They both are representative of social network in datasets and query workloads.

6.1 GDBench

GDBench is a micro-benchmark [20] that synthetically generates graphs. It contains a set of low-level atomic queries that model part of social network users. This section presents details on graph data schema, query set, and data generator.

Data schema There are two types of vertices defined in GDBench: person and webpage. persons are connected to each other via relationship of friend and they are connected to webpages via like. Each person has the attributes (i.e. property) of a person identifier (pid), name and two optional fields: age and location. A webpage has the attributes of a webpage identifier (wpid), URL, and optionally creation date. Figure 6.1 shows the data schema [4]. Overall, there are two types of vertices (person, webpage) and edges (person $\rightarrow$ person, person $\rightarrow$ webpage) in a graph defined by GDBench.

Query set Query selection in GDBench is based on the user interaction with Facebook to identify atomic actions. GDBench evaluates individual performance of these operations, such as join and aggregation in relational DBMSs, rather than more complex queries.

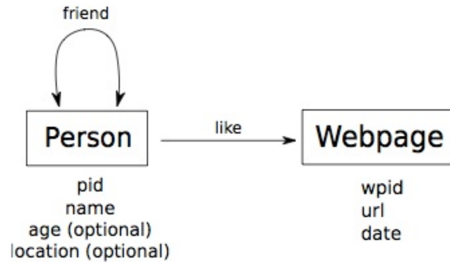


Figure 6.1: GDBench schema

These operations on social network graphs are mapped to five groups of queries: selection, adjacency, reachability, pattern matching and summarization. Query mix is shown in Table 6.1.

Q	Description	Type
1	Get all persons who have a name N.	Selection
2	Get all persons who like a given webpage W.	Adjacency
3	Get all webpages that person P likes.	Adjacency
4	Get the name of the person with a given pid.	Selection
5	Get the friends of the friends of a given person P.	Reachability
6	Get the webpages liked by the friends of a given person P.	Reachability
7	Get the persons that like a webpage which a person P likes.	Reachability
8	Is there a connection (path) between persons P1 and P2?	Reachability
9	Get the shortest path between persons P1 and P2.	Reachability
10	Get the common friends between persons P1 and P2.	Pattern Matching
11	Get the common webpages that persons P1 and P2 both like.	Pattern Matching
12	Get the number of friends of a person P.	Summarization
13	Get the friends of the friends of the friends of a given person P.	Reachability

Table 6.1: Query Set in GDBench

Table 6.1 shows five types of operations in a social network. Query 1 and 4 check the attributes of a person or a webpage. Query 2 and 3 test the efficiency of getting adjacency nodes for a database. Query 5, 6, 7 and 13 are supposed to evaluate the support of simple path expression with limited and fixed path length. This evaluation is also influenced by direction of edges the degree of vertices. Query 8 and 9 are recursive queries that involve

multiple join operations, which are both time-consuming and memory intensive. Query 10 and 11 are looking for simple graph patterns and query 12 involves a common aggregation operation.

Data generator This micro-benchmark employs the basic idea of Recursive Matrix (R-Mat) [24] and develop a more general-purpose method. R-Mat recursively add edges to the adjacency matrix of a graph until the given number of edges have been added. Nodes in a graph are partitioned into four groups and selection of a group follows a probability. Based on the strategy of R-Mat, GDBench data generator stores the distribution of edges in an array and construct graphs using such distribution. GDBench models social network graphs by following power-law distribution and using information published by current social network applications such as Facebook. Thus, data generator produces synthetic data with characteristics present in real-life social network. GDBench provides configurable parameters to generate graphs: the number of nodes in the graph (N) and the statistical distribution of edges friend/like (power law or normal). Users can specify these parameters to create graphs with different connectivity.

6.2 LDBC

We use LDBC-SNB (Linked Data Benchmark Council-Social Network Benchmark) [9] as a benchmark suite for *Gromit*. SNB is the interactive workload set based on social network from LDBC. LDBC-SNB also models a real-life social network, which includes people and their activities during a period of time. This section introduces graph data schema, query set and data generator of LDBC-SNB.

Data Schema LDBC-SNB generates graphs with thirteen types of vertices, such as person, university and comment, and relationships between vertices, such as (person) likes (post) and (university) isLocatedIn (city). Figure 6.2 shows the data schema [10]. This schema specifies attributes of both vertices and edges such as joinDate of a person to a forum. Note that schema from LDBC-SNB contains hierarchy in vertices. For example, university is a sub-class of organization and city is a sub-class of place. Besides, one class can be sub-class of itself, such as tagClass. This makes LDBC a more realistic and complex benchmark suite to execute. This is because, for example, for query that is searching post with a tag in a given tagClass or its descendent tagClass, we have limited but not fixed hops to explore, which makes both querying and executing more difficult.

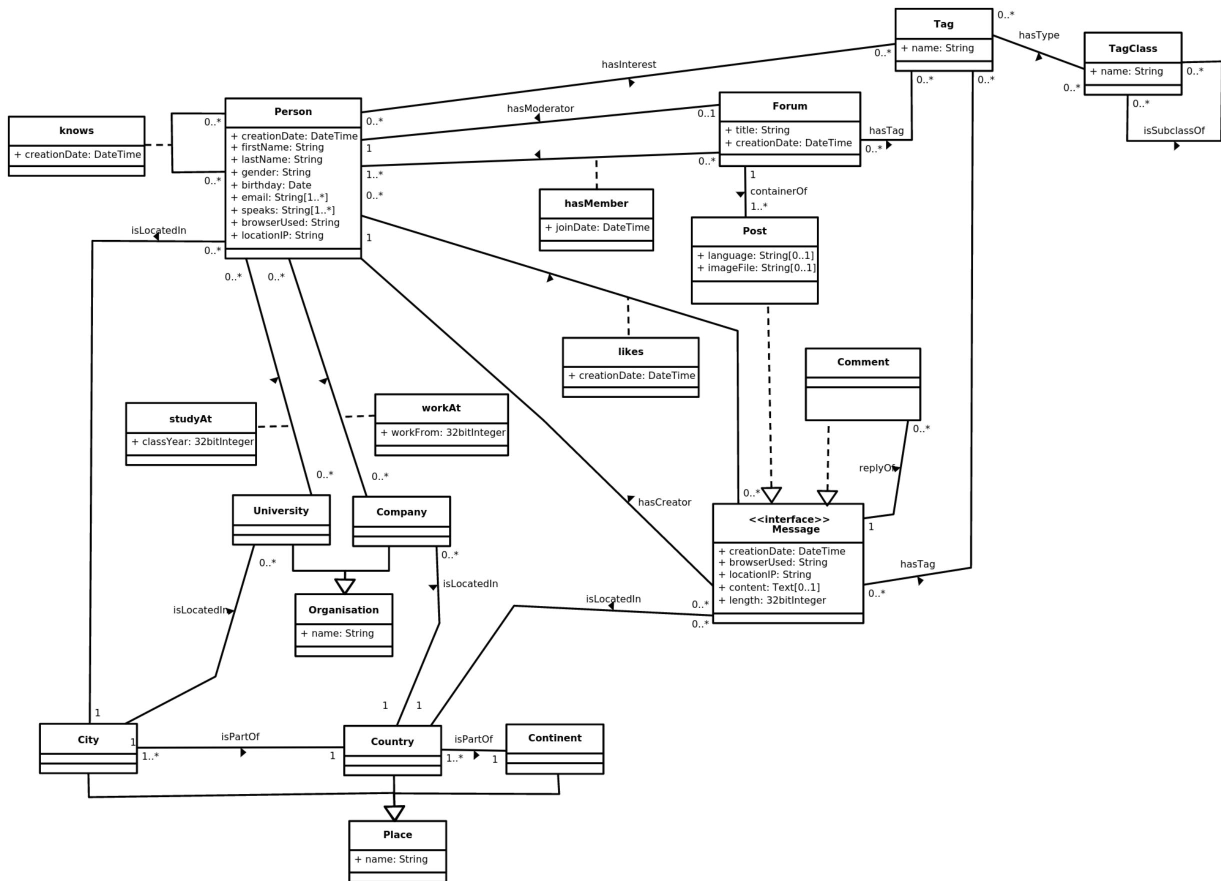


Figure 6.2: LDBC-SNB schema

Query Set Query set from LDBC-SNB includes both complex read queries and update queries. Essentially these queries capture critical operations that people use for interactions in a social network. These operations include aggregation, depth-oriented search, computation of conditional expressions, and indexing. We have a query example for each of these operations and they are listed in Table 6.2.

Query 1 from Table 6.2 involves aggregate operation that counts the number of replies of comments to each other's posts or comments between every two persons in a path of friends. Query 2 explores multi-hop of the graph with a criteria on tag co-occurrence, which can map to join operation in a RDBMS. Query 3 is more complex for it not only searches the graph with a specific path pattern ({person}[1] workAt {company} isLocatedIn {country}),

[1]Obejct in braces represents a vertex.

Q	Description	Type
1	Given two persons, find all (unweighted) shortest paths between these two persons in the sub-graph induced by Knows relationship. Then for each path calculate a weight based on the sum of comments replying a post or comment.	Aggregation
2	Given a start person and some tag, find the other tags that occur together with this tag on posts that were created by start person's friends and friends of friends (excluding start person).	Search
3	Given a start person, find that person's friends and friends of friends (excluding start person) who started working in some company in some company in a given country before a given date (year).	Expression Calculation
4	Add a like between a given person and post.	Indexing

Table 6.2: Query Set in LDBC-SNB

but also does selection on paths after calculating the property of workFrom (year). This conditional expression calculation is very common and representative in LDBC-SNB. Query 4 is a query that updates the graph. It basically asks for indices of two existing vertices and modifies some of the fields, *NextEdge* for example. There are other types of queries, for example adjacency, which overlap with that from GDBench and are not discussed again.

Data Generator LDBC-SNB mimics the characteristics of social networks by following edge distribution from Facebook and using real data from DBpedia [3], which extracts information from Wikipedia. Data generator first generates all person nodes in the graph and their properties. And then, it determines the number of friends of each person based on a degree distribution similar to that from Facebook. The materialized edges are determined and generated by calculating correlations between two persons with similarity functions. Other activities are created and linked based on the number of friends. Data generator in LDBC is able to generate graphs of difference sizes and the size is measured by scale factors. One can choose to configure the number of persons, the number of years and start year of all activities for a graph.

6.3 *Gromit* on Micro-architectural Simulators

Micro-architectural simulations can be conducted on a variety of simulators such as sniper [23] and graphite [34]. We evaluate whether Gromit executes on a subset of the commonly used simulators. For our evaluation, we select sniper, gem5 [22], and zsim [35] as simulators since they are known to support multithreaded applications. Table 6.3 shows information about the system configurations, versions of each simulator, and required tools.

System Configurations	Linux 3.13.0, Ubuntu 14.04 Thread model posix, GCC-4.9.4
sniper	sniper-6.1, PIN-2.14
gem5	SE: gem5 [5], m5thread [11] FS: Linux kernel 2.6.22.9
zsim	zsim [17], libconfig-1.5-0.3 hdf5-1.10.0 , PIN-2.14

Table 6.3: Simulation Configurations and Tool Versions

gem5 provides both system call emulation (SE) and full system (FS) simulations. Gem5-SE supports simulations with statically compiled binary files; therefore, it requires static linking to a given thread library called m5thread [11]. The m5thread library is a specific emulation library for a subset of threading primitives. On the other hand, Gem5-FS provides an operating system based environment for simulations, and one has to provide a Linux kernel to simulate the complete system. *Gromit* supports single threaded execution (*Gromit*), and multithreaded (*Gromit-MT*) execution versions for simulations. The results are shown in Table 6.4. In this table, Yes indicates that *Gromit* can run and yield correct results and No indicates the simulator or application crashes.

In the above evaluation experiments, sniper works with both versions of *Gromit*. Sniper can simulate multithreaded applications in a trace-driven manner, and it works with C++ standard thread library, which is used by *Gromit*. The thread library, m5thread, from gem5 supports limited number of functions and features about thread and *Gromit* is unable to link with this library. Therefore, *Gromit-MT* is not supported by gem5-SE or zsim, which expect static linking to this thread library. Gem5-FS supports simulations under different computer architectures, such as X86 and ALPHA. However, gem5-FS has problem in simulating *Gromit* with both architectures. The X86 one complains about Linux kernels and the cross-compilation fails in compiling *Gromit* for this ALPHA architecture. We

Simulator	*Gromit*	*Gromit-MT*
sniper	Yes	Yes
gem5-SE	Yes	No
gem5-FS	No	No
zsim	Yes	No

Table 6.4: Simulation Result

believe the thread library support from m5thread need to be improved and the tool set from gem5 needs to be updated to support full system simulations.

Chapter 7

Summary and Future Work

In this thesis, we introduced *Gromit* as an in-memory graph database management system specially designed for social network applications. *Gromit* used vertex-edge graph model and represented both vertices and edges as entities in labeled property graphs. Vertices and edges were stored in a doubly linked list manner in main memory for direct accesses. We implemented breadth-first traversal and depth-first traversal as graph traversal methods to retrieve information encoded in graphs. This graph database employed locking mechanisms for concurrency controls. To solve the deadlock problem associated with two-phase locking, we implemented three different methods and they are *No-Wait*, *Wait-Die*, and *Deadlock-detection*. The diagram of *Gromit* architecture is shown in Figure 7.1. It is noticeable that *Gromit* is implemented in C++, which allows for low-level memory management. This direct manipulation of memory provides the opportunity to explore hardware facility designs, such as hardware prefetcher and branch predictor.

To explore the full potential of an in-memory graph database management system, we believe there are some future work to do. First, to thoroughly evaluate the performance of locking mechanisms, we need deletion operation extended to the query range. Although benchmark suite LDBC tends to evaluate a database in lots of different aspects, it lacks in deletion of graph objects, which is common and frequent in social networks. To our beliefs, deletion is important in evaluating the efficiency of graph structure design and locking behaviour. Second, some optimization can be done in terms of vertex or edge grouping. We observed that queries from social networks are usually limited in specific domains for each hop of vertices, thus, it is feasible to group vertices with labels and organize edges accordingly. This can accelerate traversals by avoiding unnecessary enquiring of vertices or edges and alleviate locking contention by unlocking irrelevant connections. Third, we need to explore different concurrency control method such as multi-version concurrency control

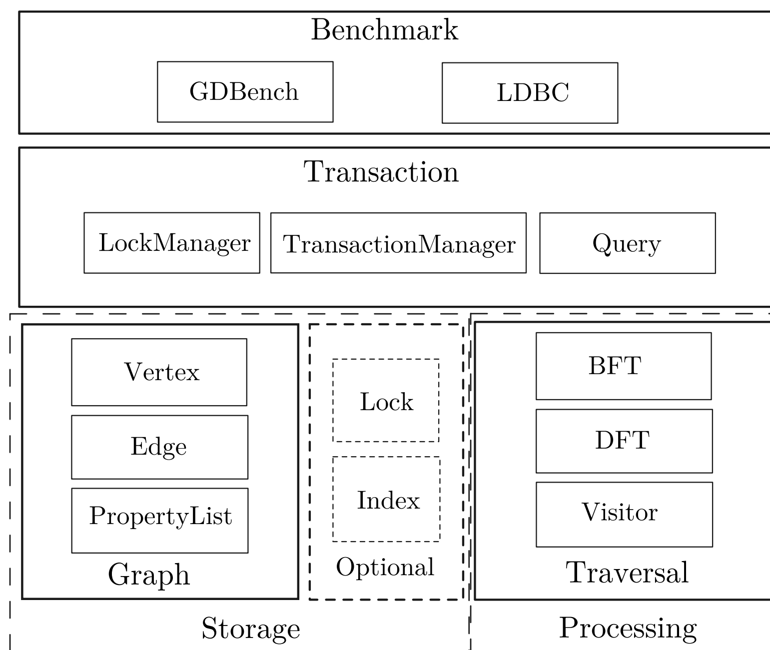


Figure 7.1: Architecture Diagram

(MVCC). It is noticeable that there have been lots of research on optimizing MVCC in databases other than RDBMSs and they achieve excellent performance. We believe *Gromit* can achieve comparable performance with MVCC.

References

[1] Boost graph visitor. http://www.boost.org/doc/libs/1_61_0/libs/graph/doc/visitor_concepts.html. Accessed September 17, 2016.

[2] Cassandra. http://cassandra.apache.org/. Accessd November 13, 2016.

[3] Dbpedia. http://wiki.dbpedia.org/. Accessed October 18, 2016.

[4] Gdbench schema. https://github.com/renzoar/GDBench/blob/master/schema.jpg. Accessed October 17, 2016.

[5] gem5. https://github.com/gem5/gem5/. Accessed February 14, 2017.

[6] Giraph. http://giraph.apache.org/. Accessd December 31, 2016.

[7] Graphchi. https://github.com/GraphChi/. Accessed March 27, 2016.

[8] Gromit. https://git.uwaterloo.ca/caesr-pub/gromit. Accessd Feburary 13, 2017.

[9] LDBC-SNB. http://ldbcouncil.org/developer/snb. Accessed September 17, 2016.

[10] LDBC-SNB Schema. https://github.com/ldbc/ldbc_snb_docs/blob/master/tex/figures/schema/schema.pdf. Accessed October 17, 2016.

[11] m5thread. https://github.com/tiwarianoop2/m5threads/. Accessed February 25, 2017.

[12] mongoDB. https://docs.mongodb.com/manual/. Accessd August 23, 2016.

[13] Neo4j. http://neo4j.com/. Accessd March 23, 2016.

[14] Sparksee. http://www.sparsity-technologies.com/. Accessd March 24, 2016.

[15] System G. http://systemg.research.ibm.com/. Accessd March 26, 2016.

[16] Titan. http://s3.thinkaurelius.com/docs/titan/1.0.0/. Accessd March 23, 2016.

[17] zsim. https://github.com/s5z/zsim/. Accessed February 15, 2017.

[18] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 105–117, New York, NY, USA, 2015. ACM.

[19] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1974.

[20] Renzo Angles, Arnau Prat-Pérez, David Dominguez-Sal, and Josep-Lluis Larriba-Pey. Benchmarking database systems for social network applications. In *First International Workshop on Graph Data Management Experiences and Systems*, GRADES '13, pages 15:1–15:7, New York, NY, USA, 2013. ACM.

[21] Philip A. Bernstein and Nathan Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[22] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[23] Trevor E. Carlson, Wim Heirman, Stijn Eyerman, Ibrahim Hur, and Lieven Eeckhout. An evaluation of high-level mechanistic core models. *ACM Trans. Archit. Code Optim.*, 11(3):28:1–28:25, August 2014.

[24] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *SDM*, volume 4, pages 442–446. SIAM, 2004.

[25] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A

Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[26] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[27] E. Ebrahimi, O. Mutlu, and Y. N. Patt. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, pages 7–17, Feb 2009.

[28] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, November 1976.

[29] Jing Fan, Adalbert Gerald Soosai Raj, and Jignesh M Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[30] Ryan Johnson, Ippokratis Pandis, and Anastasia Ailamaki. Improving OLTP Scalability Using Speculative Lock Inheritance. *Proc. VLDB Endow.*, 2(1):479–489, August 2009.

[31] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35, New York, NY, USA, 2009. ACM.

[32] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. Graphchi: Large-scale Graph Computation on Just a PC. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, pages 31–46, Berkeley, CA, USA, 2012. USENIX Association.

[33] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. High-performance Concurrency Control Mechanisms for Main-memory Databases. *Proc. VLDB Endow.*, 5(4):298–309, December 2011.

[34] Jason E Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, pages 1–12. IEEE, 2010.

[35] Daniel Sanchez and Christos Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, pages 475–486, New York, NY, USA, 2013. ACM.

[36] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990.

[37] Y. Xia, I. G. Tanase, L. Nai, W. Tan, Y. Liu, J. Crawford, and C. Y. Lin. Graph Analytics and Storage. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 942–951, Oct 2014.

[38] Cong Yan and Alvin Cheung. Leveraging Lock Contention to Improve OLTP Application Performance. *Proc. VLDB Endow.*, 9(5):444–455, January 2016.

[39] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.*, 8(3):209–220, November 2014.