Integrating Column-Oriented Storage and Query Processing Techniques into Graph Database Management Systems

by

Pranjal Gupta

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

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

Column-oriented RDBMSs, which support traditional read-heavy analytics workloads, employ a specific set of storage and query processing techniques for scalability and performance, such as positional tuple IDs, column-specific compression, and block-oriented processing. We revisit these techniques in the context of contemporary graph database management systems (GDBMSs). GDBMSs support a new set of analytics workloads, such as fraud detection in financial transaction networks or recommendations in social networks, that are also read-heavy but have fundamentally different access patterns than traditional analytics workloads. We first review the data characteristics and query access patterns in GDBMS to identify components of GDBMSs where existing columnar techniques can and cannot directly be used. We then present the physical data layout of columnar data structures, new columnar compression, and query-processing techniques that are optimized for GDBMSs. Our techniques include a new compact vertex and edge ID scheme, a new null and empty list compression scheme based on prefix-sums, and list-based query processing. We have integrated our techniques into GraphflowDB, an in-memory GDBMS. Compared to uncompressed storage, our compression techniques has scaled the system by 3.55x with minimal performance overheads. Our null compression scheme outperforms existing columnar schemes in query performance, with minor loss in compression rate and achieves both higher compression rate and better query performance as compared to row-oriented storage techniques adopted by existing GDBMSs. Finally, our list-based query processor techniques improve query performance by 2.7x on a variety of path queries and significantly outperform their corresponding conventional versions.

Acknowledgements

First and foremost, I want to thank my supervisor, Professor Semih Salihoglu, for all the guidance and support he has provided me over the past two years. Working with Semih has been a great experience and am really grateful to be a part of his reaserch group. He has been a constant source of inspiration to whom I can look upto. Thank you for being a great teacher and guide.

I next want to thank Siddhartha, for being a friend to whom I can turn up anytime and for anything, and Amine, for helping me with the Graphflow project, collaborating with me on the A+ Indexes and also coping with my not-so-good reaction at times. All nighters were fun too with you guys. I am also grateful to Manoj, Antony and Aatish who helped me during my time here; and to my friends Yash and Garvit for just being there to listen to me. Moreover, I would also like to thank Siddhartha, Amine and Xiyang for proofreading my thesis.

I also want to express my sincerest gratitude to my thesis readers, Ken Salem and Tamer Özsu, for taking out valuable time from their busy schedules to read my thesis and provide their valuable comments.

I would also like to thank my parent for their guidence, support and encouragement in all my endeavours in life; and my siblings, Pranshri and Shrijal, for keeping up my morale.

Finally, I want to thank my friend, Snehal, for being my constant source of love, happiness and motivation. Thank you for seeing me through all my good and bad phases.

Dedication

This is dedicated to the ones I care about .

Table of Contents

List of Tables

List of Figures

Abbreviations

CSR Compressed Sparse Row viii, 2, 13, 20, 21, 25, 34, 37, 40, 41, 43, 46–49

GDBMS Graph Database Management System 1–3, 5–7, 9, 11–15, 35, 46, 50, 51

LDBC SNB Linked Data Benchmark Council Social Network Benchmark x, 26, 35, 36, 39, 41, 51

RDBMS Relational Database Management System 1–4, 11, 23, 24, 50, 51

Chapter 1

Introduction

The term Graph Database Management System (GDBMS), in its contemporary usage, refers to data management software such as Neo4j [2], JanusGraph [33], TigerGraph [52], and GraphflowDB [35, 38], that adopt the property graph data model [39]. In this model, application data is represented as a set of vertices, which represent the entities in the application, directed edges, which represent the relationships between entities, and arbitrary key-value properties on the vertices and edges, where both the relationships and key-value properties can depict different levels of structure.

GDBMSs have lately gained popularity to support a wide range of analytics applications, such as fraud detection, risk assessment in financial services and recommendations in e-commerce and social networks [50]. These applications have read-heavy workloads that search for patterns in a graph-structured database, which often requires processing large amounts of data. In the context of Relational Database Management System (RDBMS)s column-oriented storage [32, 43, 51, 54] employ a set of storage, indexing, and query processing techniques to support traditional read-heavy analytics applications, such as business intelligence and reporting, that also process large amounts of data. As such, these columnar techniques are relevant for improving the performance and scalability of GDBMSs. In this thesis, we revisit columnar storage and query processing techniques and investigate their integration in GDBMSs. Specifically, we discuss the applicability of columnar storage techniques [51], compression schemes for columns [5, 3, 56], and vector-oriented query processing [16, 6] for storing and accessing different components of GDBMSs. Even though analytical workloads that are run on GDBMSs and those on column-oriented RDBMSs exhibit many similarities, they have different fundamental data access patterns. This calls for redesigning columnar techniques in the context of GDBMSs.

We begin with an analysis of general query execution on GDBMSs to understand data access patterns at the physical layer. We also identify different types of structure that can be present in graph data, such as set of properties on a vertex or edge and the cardinality of edge labels. This analysis gives us a set of guidelines and desiderata that instruct how to integrate and adopt columnar techniques in the context of GDBMSs. We first identify the storage components of a GDBMS where columnar storage can be directly integrated. For instance, columnar storage is directly applicable for storing vertex properties. Similarly, we use the popular CSR or compressed sparse column (CSC) formats to store the topology of graphs, i.e., the edges between vertices, which are columnar data structures that store multi-value attributes, employing a form of run-length encoding. In contrast to vertex properties, we observe that using the straightforward columnar storage to store edge properties with global positional edge IDs leads to a suboptimal solution. Similarly we show that integrating existing null compression schemes from column-oriented RDBMSs and vector-oriented processing directly into GDBMSs is not appropriate and do not satisfy the set of desiderata we outline. We then describe new techniques that address the shortcomings of these techniques. We integrate all of our techniques into the GraphflowDB GDBMS and demonstrate that our techniques has increased the system's scalability significantly with performance benefits.

1.1 Contributions

The specific contributions of this thesis are as follows:

- **Guidelines and Desiderata:** Chapter 2 reviews the properties of data access patterns in GDBMSs, from which we derive a set of general guidelines and desiderata for designing the physical data layout of GDBMSs. We further explore the characteristics of real-world graph data and identify different types of structures that can exist in the graph data. The guidelines instruct the applicability of the columnar techniques we revisit in later chapters.
- **Columnar Storage:** Chapter 3 explores the application of columnar data structures for storing different components of GDBMSs. We start with components that can directly be stored in existing columnar structures, specifically vertex properties and adjacency lists for many-to-many edges. Next, we identify the requirements in using columnar data structures for edge properties and present two initial solutions that optimize storage and performance, respectively. We discuss the pros and cons of both solutions and then describe a third solution, *single-directional property pages*, that

lies in between the previously described 2 solutions and strikes a good balance between storage and performance efficacy. Lastly, we show how single cardinality edges (having one-to-one, one-to-many and many-to-one edge labels) can be stored and referenced as the property of either source or destination vertex in vertex columns. We experimentally demonstrate how storing these edges in vertex property columns can achieve huge storage and performance improvements.

- **New Vertex and Edge ID Scheme:** We introduce new ID schemes for identifying vertex and edges, which can be used as positional offsets to directly access their properties from the columnar structures. This allows for fast random access to properties in property columns. Additionally, the new scheme allows representing edges and vertices in the compressed form in adjacency lists. At a high-level, our new ID scheme identifies the vertex or edge in a system by a set of values that are small-sized and can be omitted from storing in the adjacency lists when the graph is highly structured. In fact, we can represent both an edge and a neighbour vertex together in adjacency list with a single value of just 4 bytes.

- **Columnar Compression:** In Chapter 4, we discuss the application of existing columnar compression techniques in GDBMSs based on our guidelines. For each of the columnar techniques, we review their applicability and discuss where in our columnar storage they can be applied. As property columns and adjacent lists can be sparse, we next review existing null compression techniques for columns and show the existing schemes would lead to very slow read accesses. We then describe a new null compression scheme, based on storing prefix sums of non-null values, that addresses this shortcoming. Our new null compression scheme can be used to compress both null edge and vertex properties, as well as empty adjacency while allowing constant-time access to any null or non-null property with a small increase in storage overhead per entry.

- **List-based Processing:** In Chapter 5, we review the query processing techniques used in GDBMS as well as in column-oriented RDBMSs. We show that the traditional Volcano-style [28] query processor, that processes the query one match-at-time, are not able exploit the benefits from arrangement of data in our columnar data structures. On the other hand, column-at-a-time [6] or vectorized [55] query processors employed by several column stores benefits from the arrangement of the data but do not adapt well to graph queries that have many many-to-many join operations, i.e long path queries. To overcome these shortcomings, we introduce a new list-based query processor that runs queries on entire adjacency lists at a time. Our new processor is a hybrid between volcano-style and vectorized processing, that can operate

on single values as well as entire lists at a time, and overcomes the drawback of vectorized processing on graph data. However, unlike vectorized processing, the size of lists in our case are of variable length, depending on the number of adjacent edges of a particular vertex.

In Chapter 6, we present experiments on our columnar data structures and techniques to show the benefits in terms of memory usage and query performance. Chapter 7 presents related work in storage and compression techniques in the context of column-oriented RDBMSs and graph structured data management systems. Finally, Chapter 8 concludes and outlines directions for future work.

Chapter 2

Guidelines and Desiderata for Optimizing the Physical Data Layout and Query Processor in GDBMSs

In this chapter, we review the components of storage layer, primary query plan operators, and the general data access patterns of operators when evaluating a query in a GDBMS. We then draw a basic set of guidelines and desiderata that will instruct the physical data layout of our columnar data structures and query-processing techniques introduced in later chapters.

Section 2.1 briefly describes the *property graph data model*. Section 2.2 describes the primary storage components of GDBMSs that adopt the property graph data model, while Section 2.3 reviews the query processing operators in GDBMSs. We end the chapter by stating our guidelines in Section 2.4.

2.1 Property Graph Data Model

Figure 2.1 shows a graph data represented using the property graph data model, that will serve as our running example in this thesis. A property graph consists of *vertices* that represent entities and directed *edges* between vertices that represent relationships between entities. Each vertex and edge has a particular *label*, describing the high-level categories of vertices and edges. For example, in Figure 2.1, vertices have labels: `PERSON` and `ORG`, while edges have labels: `FOLLOWS`, `WORKAT`, and `STUDYAT`.

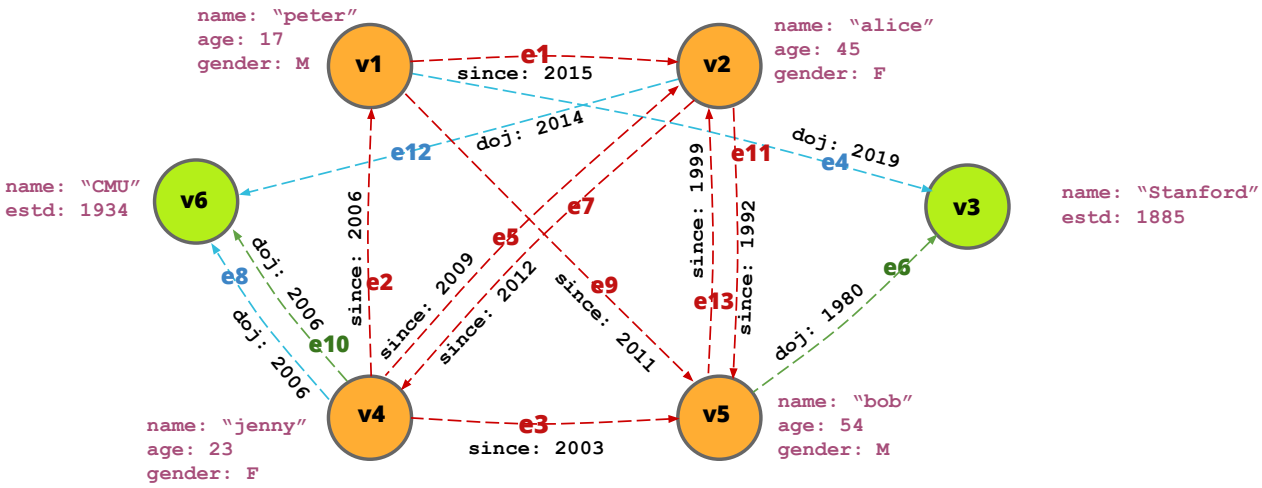


Figure 2.1: Running example graph.

Similar to columns in relational tables, vertices and edges can have key-value *properties*. Although the properties of vertices and edges do not need to adhere to a strict *schema*, in practice many of these properties are often highly structured, i.e., the same set of properties exists on all the vertices and edges of the same vertex and edge label respectively.

2.2 Primary Storage Components in GDBMSs

In every GDBMS we are aware of, the edges of a graph are stored in an data structure called *adjacency lists* [17]. An adjacency list of a vertex v is a list of *adjacent* edges v and the corresponding *neighbouring* vertices. Each vertex has 2 adjacency lists: a *forward adjacency list* containing all outward edges of that vertex, and a *backward adjacency list* that holds all inward edges of the vertex. One can think of edges in the graph as a relational table with 3 attributes: a source vertex, a destination vertex, and an 8-byte edge ID. The adjacency lists can then be thought of as an *index* on this relational table that is *clustered* by either the source or destination vertex. In practice, this indexing structure often has a depth of 1 or 2, hence, given the ID of a vertex v, a system can access v's list of adjacent edges and neighbour vertices in 1 or 2 lookups. By having adjacency lists of a vertex in either direction, the system can access the list of outward and inward edges as well as the neighbouring vertices of u in a constant-time lookup operation, which provides the core capability of fast joins on vertices to a GDBMS.

Typically, an adjacency list of vertex u is further clustered into sublists grouped by the

edge label. This enables traversing the neighbourhood of a vertex based on a *particular* label in constant time. The rationale behind the sub-clustering is that queries made by applications often have specific labels on query edges. Some systems further order the edges in the adjacency lists either by a specific property associated with the edges or neighbour vertices, such as *location* or *timestamp*, or simply by neighbour vertices' IDs. Sorting enables the system to access parts of lists in time logarithmic in the size of the adjacency list.

A GDBMS also stores properties associated with the vertices and the edges. A straightforward approach is to store properties in a *key-value store* [1] and use the attribute key and the vertex or edge ID as the key into the store. Properties can also be stored in an *interpreted attribute layout* [13], where a record of a vertex or an edge consists of the the key-value pairs of that particular vertex or edge, and is variable-sized depending on the size of keys and values data. However, searching for a property in variable-sized records involves decoding and parsing the entire record until the matching attribute is found, which can be expensive. Another way of storing properties is using doubly linked-lists, as done in Neo4j [2], where the system keeps track of a pointer to the first property record of a vertex or edge and each subsequent property record stores a pointer to the next record. This is a cache-inefficient layout since it does not organize the properties of vertices and edges in the order in which they would be accessed in query execution.

2.3 Query Execution in GDBMSs

In this section, we review the general query execution in a GDBMS by analyzing the major operators used in query plans. Although systems differ in their architectures and implementation of operators that they support for executing queries, there are similarities in their data access patterns. We use the Cypher query language [26] to describe the queries we use in our examples. A user query typically consists of 3 parts: 1) a `MATCH` clause that describes a subgraph query pattern $Q(V_Q, E_Q)$, where V_Q and E_Q are the query vertex and edge variables, respectively, that the system will match on the input graph; 2) a `WHERE` clause the contains a predicate ρ over properties of the edges and vertices that the matched subgraph must satisfy; and 3) a `RETURN` statement that returns a projection of the variables in the match query or performs a group-by and aggregate information. The `MATCH`, `WHERE`, and `RETURN` clauses of a Cypher query effectively corresponds to the `FROM`, `WHERE`, and `SELECT` clauses of SQL. Example 1 shows a typical query written in Cypher, that queries the example graph in Figure 2.1.

Example 1. *Example Cypher query.*

```
MATCH (a:PERSON)−[e:WORKAT]→(b:ORG)
WHERE a.age > 22 AND b.estd < 2015
RETURN *
```

This query returns all `PERSON` *vertices and their workplaces, constrained by the condition that the* `'age'` *property of all* `PERSON` *vertices has a value greater than 22 and the* `'established'` *property of all* `ORG` *vertices is less than 2015. a and b are query vertex variables while e is a query edge variable.*

The following are the major operators used for matching a subgraph pattern and evaluating predicates in a query.

- `SCAN`: Scans a set of vertices and edges from the graph topology.
- `NEIGHBOURHOOD JOIN` (e.g. `EXTEND/INTERSECT` in Graphflow; `EXPAND` in Neo4j): At a high level, this join operator matches the subgraph query pattern Q, one edge at a time. Some systems, e.g Graphflow, can also match cyclic queries by matching multiple edges at a time. The input to the operator is a partial match, t, that has matched k of the query edges in Q. For each partially matched t, the operator extends t by matching an unmatched query edge $e_q(u_q, v_q)$, where one of v_q or u_q has already been matched in t, say v_q has already been matched. The *join* happens by sequentially reading adjacent edges and neighbour vertices from forward or backward adjacency list of the matched v_q, to produce a $k+1$-match. The output of the `JOIN` is the tuple t with e_q and u_q filled.
- `PROPERTY READER`: The vertex or edge property reader reads a property value of any vertex or edge that has been assigned to a variable in V_Q or E_Q of a partial match t, from the underlying property storage.
- `FILTER`: Given the predicate ρ from the `WHERE` clause and a partial match t of Q, the `FILTER` operator omits t from the result of the query if t does not pass predicate ρ.

Figure 2.2 shows one of the query plans that the system will generate to execute query in Example 1. It consists of the following sequence of operators: 1) `SCAN` operator that matches the variable a in query to vertex in the graph having label `PERSON`; 2) `PROPERTY READER` reads the property `age` of the vertex matched to a; 3) `FILTER` operator filter out the partial match based on the contraint $a.age > 22$; 4) `JOIN` operator matches b by reading

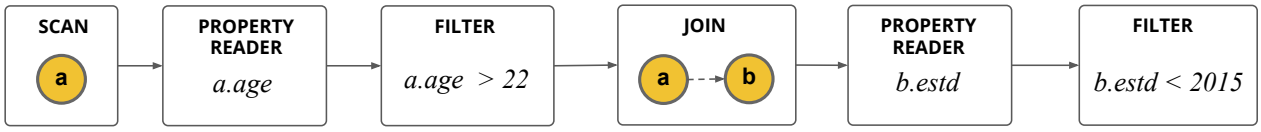


Figure 2.2: Query plan for Example 1.

the forward adjacency list of a's match; 5) `PROPERTY READER` reads the property `estd` of b's match; and finally 4) another `FILTER` operator filters out the matched query pattern that do not confirm to the constraint $b.estd < 2015$.

2.4 Guidelines and Desiderata

We next outline a set of guidelines and desiderata for designing the physical data layout and query processor of a GDBMS.

Guideline 1: Edges are doubly-indexed.

Each edge appears in the forward adjacency list of that edge's source vertex and the backward adjacency list of its destination vertex. This results in a 2x replication factor in storing the topology of a graph in the system. This replication cannot be avoided by dropping an adjacency list in any one direction without hampering the capability to perform fast neighbour joins, which is one of the core features of a GDBMS. Therefore, we will also doubly index the edges in our design.

Guideline 2: Edge properties are read in the same order as edges in an adjacency list.

During the execution of a query, the `JOIN` operator will access the edges of a vertex v in the order these edges appear in v's forward or backward adjacency lists. If the query also needs to access the properties of these edges, the access to these edge properties will also be in the same order in which edges were read from the adjacency list. Given that the edges and edge properties are read in order, we define our first desideratum for the physical data layout and query processor:

Desideratum 1: *Store the properties of edges in the order in which edges are ordered in the adjacency lists and read the edges and their properties sequentially in the operators*

Guideline 3: Vertices cannot be ordered to make access from all neighbour vertices sequential.

Contrary to how the edges and edge properties can be strictly ordered for each of the adjacency lists, in general, there cannot be ordering on the vertices that completely localizes the access to neighbour vertices of every vertex and the properties of these neighbour vertices without prohibitive data replication. In general, if a vertex v has n neighbours, then v and its properties need to be replicated n times. Hence, localizing access to neighbour vertices and their properties should not be put in the desiderata of the system's physical data layout design.

Guideline 4: Access to vertex properties are random and many adjacency lists are very small.

Guideline 3 implies that the system should take it granted that access to the vertex properties will require random accesses and will not be sequential. For example, when joining a node v with its neighbors and accessing the `age` property of each neighbor, the accesses will be to non-consecutive locations in the vertex properties' storage based on the neighbour vertex IDs. Another property of real-world graphs is that adjacency lists of many vertices are very small because of the power-law distribution of adjacency lists. Systems should expect many very short adjacency lists, with only a few edges in them. Therefore during query processing with two or more joins, reading different adjacency lists and properties of these edges will require reading a short list followed by a random access and then reading another short list, etc.

In an in-memory setting, which we focus on in this work, our aim with compression is not to achieve high compression ratios but to optimize for high decompression rates or to avoid decompression of data altogether, as compression schemes with slow decompression hurt performance. Moreover, techniques that require decompressing blocks of data, say a few KBs, to only read a single property or a single short adjacency list can be prohibitively expensive. Our second desideratum is:

Desideratum 2: *If compression is used, the system should be able to decompress arbitrary single elements in a compressed block in constant time.*

Guideline 5: Graph data often has partial structure.

Even though the property graph data model is semi-structured, in practice graph databases

stored in GDBMSs often have a structure in the data, which GDBMSs can exploit. One reason this structure exists is that, as observed in prior work [49], the data in GDBMS often comes from structured data stored in RDBMS. In fact, several of the GDBMSs from industry and some academics are actively working on defining a schema language for the property graph data model [18, 31]. We identify three commonly appearing structure in property graph data:

1. **Edge label determines the source and destination vertex labels.** Often, edge labels in the graph data have a well-defined set of source and destination vertex label(s). This restricts the vertices to having inward or outward edges of only a definite set of labels. In our example graph, edges with label FOLLOWS only exists between vertices of label `PERSON`.

2. **Edge label has fixed cardinality.** The number of edges of a particular label to which a source or destination vertex can be associated is a property of the edge label. We call this the *cardinality* of an edge label. *One-to-one* cardinality for a label l_e means that each source vertex can be connected to at most one destination vertex through an edge with label l_e and vice versa. *Many-to-one* permits a single edge of a label from a source vertex but multiple edges to a destination vertex. Similar analogy can be applied to *one-to-many* and *many-to-many* cardinality edge labels.

3. **Label determines properties on vertices and edges.** Similar to the attributes of a relational table, properties on an edge or vertex and the datatypes of these properties can *often* be determined by its edge or vertex label. In our example graph, all vertices having the label `PERSON` have 3 properties: name: `STRING`, age: `INT` and gender: `STRING`. As long as a significant fraction of vertices and edges with a particular label have a common set of properties, a system can exploit this structured to store these properties more efficiently.

Such structure in data provides an opportunity to design more efficient and simpler data structures for accessing the storage layer of GDBMS. Our third desideratum is:

Desideratum 3: *Exploit the above three commonly appearing structures in the graph data to (i) compress the data to save space; and (ii) provide faster access to the data.*

However, not all data in graph databases have structure. As a working terminology, we will use the following terms:

- **Structured/unstructured edge:** An edge of a particular label, that follows above-mentioned points , is called an structured edge. An edge that is not structured, is called an unstructured edge.
- **Structured/unstructured property:** A structured property is a property on a vertex or edge that; (i) can be determined by the vertex type or edge label of that entity; and (ii) have the same data type in all its occurrence. We will also require that the property is not too sparse, i.e., appears in a significant fraction of the vertices of edges of a particular label. Any property that is not a structured property, is considered unstructured.

We focus on optimizing the storage of structured part of the graph data in this thesis. Optimizing a system for unstructured part of graph data is an interesting research topic for the future work. A standard approach is to serialize the key, datatype and value of each property in variable-length records. Structured property storage can, however, be optimized to benefit memory footprint as well as access performance.

Guideline 6: Queries read a small subset of the vertex or edge properties

In order to understand the nature of queries users ask a GDBMS, we conducted a survey of 100 StackOverflow questions containing openCypher queries. We focused on queries of analytical nature and discarded transactional ones like insert, delete and update. We observe the following:

- Out of the 100 queries, 68 accessed at least one of the properties on a vertex or an edge. Of these, 61 accessed vertex properties and 13 accessed edge properties.
- Only 11 queries returned all the properties of a query edge or vertex, while 35 of them returned specific properties.
- The average number of properties accessed by those queries that explicitly return a set of properties was only 1.6.

We can observe that vertex properties are more popularly accessed than edge properties and most of the queries only access 1 or 2 properties. This leads to our fourth desideratum:

Desideratum 4: *Allow fast access to individual properties of multiple vertices instead of all properties of a single vertex.*

Chapter 3

Columnar Storage

In this chapter, we explore the application of columnar data structures for different storage components of GDBMSs to meet the desiderata we outlined in Chapter 2.

Section 3.1 describes the design of columns to store vertex properties, called *vertex columns*, and a new compact vertex ID scheme that accompanies the design. In Section 3.2, we start by describing two columnar storage designs to store edge properties and their pros and cons. Then, we propose a third design, *Single-directional property pages*, that is a sweet spot between the earlier two designs and the one we adopt for storing edge properties. Similar to Section 3.1, we describe a novel and compact edge ID scheme that accompanies our design from single-directional property pages. Section 3.3, we briefly describe the existing structure of adjacency lists in GraphflowDB, that is implemented as CSR, which is already a variant of columnar data structure and stores edges associated to a particular vertex in contiguous memory locations. However, in Section 3.4, we look for better ways of storing edges and hence, describe a storage optimization that involves storing edges with cardinalities 1-1, 1-n, n-1 in vertex columns instead of using heavy-weight CSR columnar structure. We address how to compress the data stored in these columnar structures in Chapter 4.

3.1 Columnar Storage for Vertex Properties

Columnar data-structures can be directly used for storing vertex properties. Let $(lv_1, lv_2, ...)$ be the vertex labels in a graph. Let $p_{i,1}, p_{i,2}, ...p_{i,n}$ be the structured vertex properties of lv_i, each with a specific datatype $d_{i,j}$. We define a *vertex column* for each $p_{i,j}$, having a

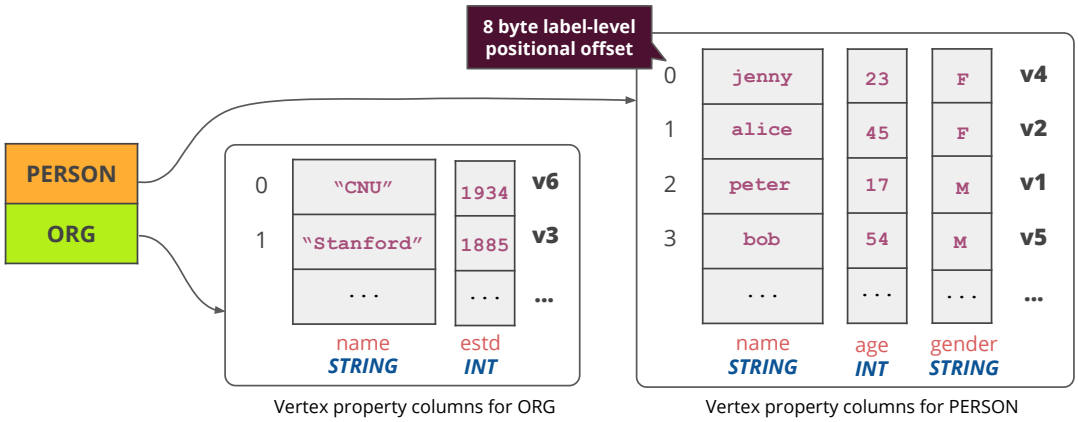


Figure 3.1: Vertex columns for the graph in Figure 2.1.

fixed data type $d_{i,j}$. Each column stores the value of a single property $p_{i,j}$ for all vertices having lv_i at consecutive locations. All property values of a particular vertex v with label lv_i is located at the same positional offset for each column for $p_{i,j}$.

Ideally, the property value of a vertex should directly be read using the ID of the vertex as the positional offset in the column. However, GDBMS typically gives globally unique 8-byte consecutive IDs to all the vertices in the system, irrespective of their labels. That means ID 0 can be given to a vertex with label `PERSON` and 1 to a vertex with label `ORG` and 2 to another vertex `PERSON` etc. We cannot use this ID scheme as the positional offset for the above design. One possible solution is to maintain a map for each label, from each vertex v's "global" ID to a "local" label-level positional offset, which is unique only among all vertices with the same label as v. This requires extra storage for maintaining the map and one level of indirection when accessing the vertex properties. Instead, we adopted an ID scheme where each vertex is identified by a ***(vertex label, label-level positional offset)*** tuple in the system in place of global vertex ID. This allows direct access to the properties by using the local positional offset which now is part of the vertex ID. However, using this new vertex ID scheme requires materializing 2 pieces of information in the adjacency lists - a vertex label and a local positional offset, compared with a single global vertex ID. This may increase the memory overhead if the local positional offsets and the global vertex ID are of the same size. However, we will show in Section 4.2 that we can often avoid storing the vertex labels with vertex IDs and even save space by using fewer bytes for local positional offsets than the bytes needed by the global vertex IDs.

For reference, Figure 3.1 shows a set of vertex columns for our example graph in Figure 2.1. It has 2 set of columns one for each vertex label, with a column for each structured

property of the label. The global vertex IDs on the right of the set of columns of a vertex label indicates the positional offset at which the properties of a particular vertex are located in the columns of a family. For instance, the properties of vertex $v2$ appear at offset 1 in columns of `PERSON`'s family. By the new vertex ID scheme, $v2$ is identified as `PRESON:1`.

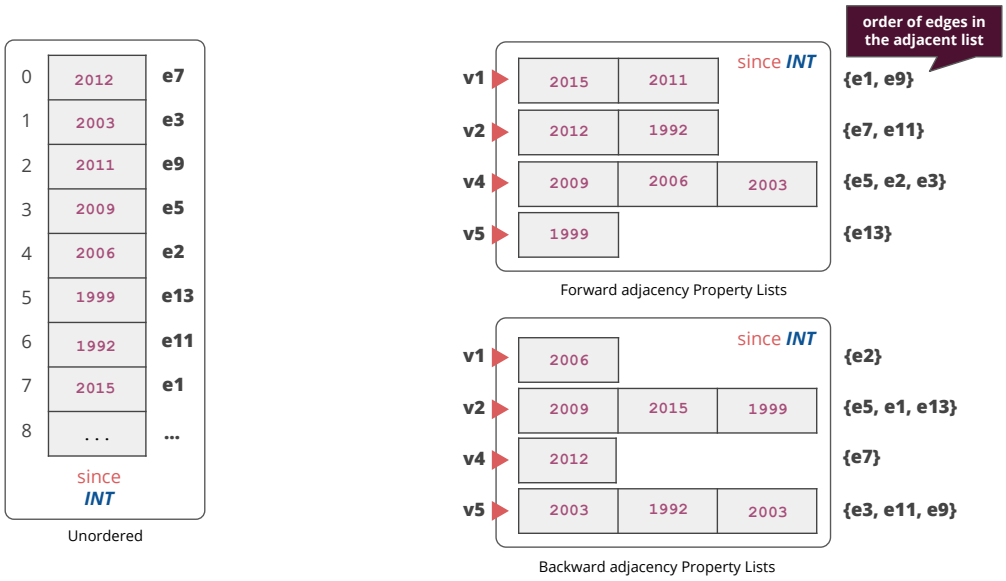
Recall from Guideline 3 that in general, reading the properties of vertices (specifically when reading the properties of neighbors of a vertex) cannot be localized without prohibitive replication. In light of this guideline, we adopt a simple and efficient additions and deletion scheme for of vertex properties and vertices. Deleting the property of vertex v is handled by setting the property at v's positional offset to NULL. Deleting v is handled by setting all of v's properties to NULL and adding v's positional offset to a list of "recycled" offsets. When a new vertex v is added, the system generates an ID by using a recycled offset if one exists or gives a new consecutive positional offset (i.e., increments the maximum positional offset by 1). Updating v's property is handled by overwriting the value in the property column for v's positional offset. All of these operations are very efficient involving only random access to different locations in columns.

3.2 Columnar Storage for Edge Properties

Recall from Guideline 2 that edges and edge properties are read by the `JOIN` operators in the order they appear in the adjacency lists. GDBMSs store the edges, i.e the (edge ID, neighbour vertex ID) pairs, consecutively in adjacency lists, through double indexing. Ideally, edge properties should also be stored in the same order. In this section, we begin by presenting two columnar storage designs for storing edge properties which can be seen as opposite ends of a design spectrum. The first design, *edge columns*, is optimized for *storage* and does not replicate the edge properties. The second design, *double-indexed property lists*, is optimized for *performance* through double indexing of the edge properties. We then describe a third design, *single-directional property lists*, that can be seen as a sweet spot between the previous two designs, which localizes the reads in one direction without any replication. However, this natural third design has several limitations, which we address in our fourth design, *single-directional property pages*, which is adopted in Graphflow.

3.2.1 Edge Columns and Double-Indexed Property Lists

Edge Columns (Nonsequential reads, no replication): One possibility is to use the columnar storage design similar to that for storing the vertex properties. That is, we have



(a) Edge Columns

(b) Double-indexed Property Lists

Figure 3.2: Storing edge properties of `FOLLOWS` edges in the Edge Columns and Double-indexed Property Lists.

one edge column for each $q_{i,j}$, where $q_{i,j}$ is a structured property of edge label le_i. Edges in the system with this solution can be identified as *(edge label, label-level positional offset)*. However, such a design would not localize the properties of the edge according to their appearance in the adjacency lists, so cannot provide sequential reads when reading the edge properties. Figure 3.2a shows how this particular design would look like. The figure shows a column storing property `since` of `FOLLOWS` edges. The property values are not ordered. For our example, the forward adjacency list of `v4` contains edges `e5`, `e2` and `e3`, whose `since` property values (at positional offsets 3, 4 and 1) are not stored consecutively in the edge column.

Double-Indexed Property Lists (Sequential reads in both directions, 2x replication): An alternative solution is to directly mimic the storage of the adjacency lists for storing the edge properties. For each vertex v that has edges with a label l_i, and each $q_{i,j}$, we store the edge properties in the *forward property lists* and *backward property lists*. This provides sequential read of properties. For example, a query that is reading the forward (backward) adjacency list of v can read the edge property values of the edges sequentially from the forward (backward) adjacency property list of v. Figure 3.2b shows the forward and backward property lists for edgel label `FOLLOWS` in our example graph in Figure 2.1. However, this design requires replicating each edge property twice. In addition, if the

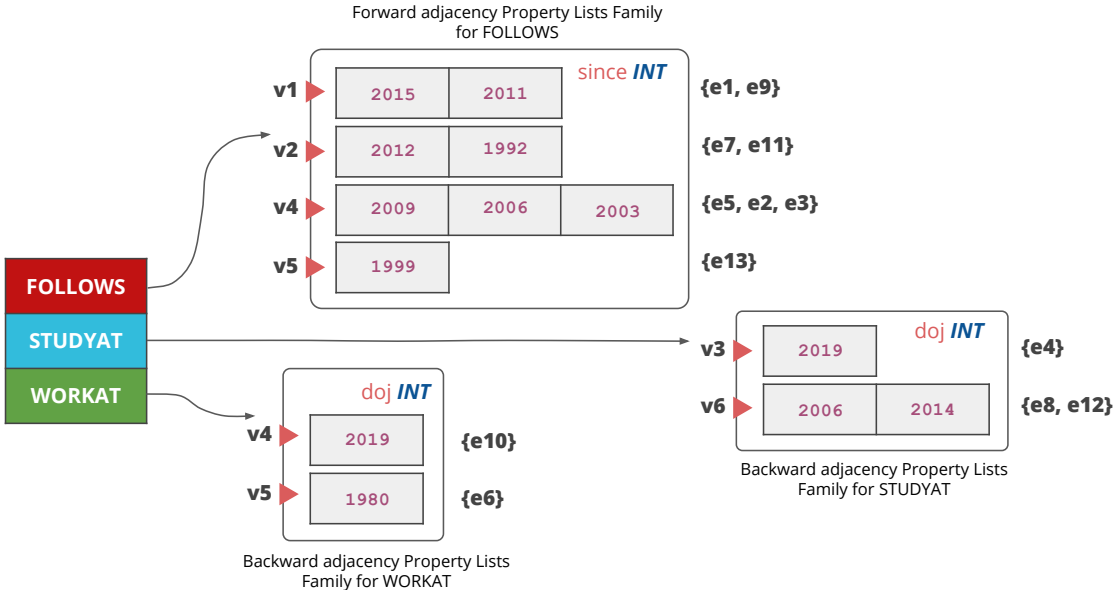


Figure 3.3: Single-directional Property Lists

original adjacency lists are sorted, then all the property lists need to be sorted too in the same way, which would make updates slower.

3.2.2 Single-directional Property Pages

A natural middle ground between edge columns and double-indexed property lists is to store only one of the forward or the backward property lists. We call this design *single-directional property lists*. Suppose the system indexes the properties of the edges with label l_i in the forward direction. Then, the edge properties can be read sequentially in the forward direction. However when reading the edges of a vertex v in the backward direction, then the edge properties will not be sequential. Figure 3.3 shows the Single-directional Property Lists for storing the properties of the example graph in Figure 2.1. In the example, edge properties in the family of `FOLLOWS` and `WORKAT` labels are stored in the forward adjacency property lists, while in `STUDYAT` the properties are in the backward adjacency property lists. In this design, reading the edges in the backward direction requires that given the ID of an edge e, the system is able to locate the positional offset of e in the forward direction. This requires a new edge ID scheme. Specifically, the conventional globally consecutive edge IDs cannot be used to access the properties quickly as they do not contain information about positional offsets. We next describe a new edge ID scheme to achieve this. Then we talk

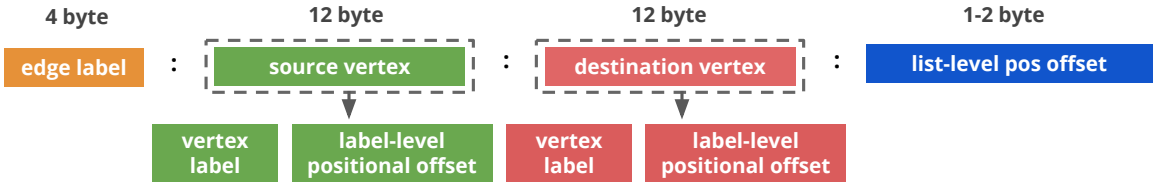


Figure 3.4: Components of the new edge identification scheme.

about some important limitations of single-directional property lists, which we address in a new design.

To access a property of an edge e having label le_i, we need 3 pieces of information; 1) $q_{i,j}$; 2) source vertex if the $q_{i,j}$'s values are stored in the forward adjacency property lists, else destination vertex; and 3) the list-level positional offset of e in that property list. For instance, in figure 3.3, `since` property of $e2$ can be accessed knowing $v4$ (source vertex of $e2$) and offset of $e2$ in $v4$'s forward adjacency property list, i.e 1. We adopt a new edge identification scheme that identifies the edge in the system by a tuple having 4 components: ***(edge label, source vertex, destination vertex, list-level positional offset)***. In our new scheme, the $e2$'s ID will be given as `FOLLOWS:v4:v1:2`, where $v4$ and $v1$ are the source and destination edges of $e2$. These edge IDs provides us with compact storage too. Most of the components need not be stored in the adjacency lists and the edge ID can be constructed during query execution by reading as few as only the neighbour vertex's local positional offset in the property list.

Limitation of single-directional property lists: Though single-directional property lists are a good middle-ground solution, it has an important limitation. Suppose that the property list for property $q_{i,j}$ of edge label le_i is indexed in the forward direction. If the original adjacency lists are sorted, any insertion or deletion can change a large number of, possibly all of the positional offsets of every edge in the adjacency list. Suppose an edge is inserted to the beginning of v's forward adjacency list L_f. This requires first calculating a map of old and new positional offsets of the edges in L_f, then searching through each backward neighbor of v and finding each edge in the backward adjacency list and updating its positional offset. A similar problem exists when an edge is deleted. If the original edges are not sorted, then insertions can be handled easily but handling deletions is challenging. The system has two options for deletions. First, the system can directly delete the edge e, which again can change the positional offsets of as many as $|L_f|$ edges. Instead, the system can thumbnail e's positional offset and recycled e's positional offset as was described for vertex deletions. However, this is also expensive. In contrast to keeping a single list of recycled IDs as in vertex columns, the system now needs to keep track of recycled positional

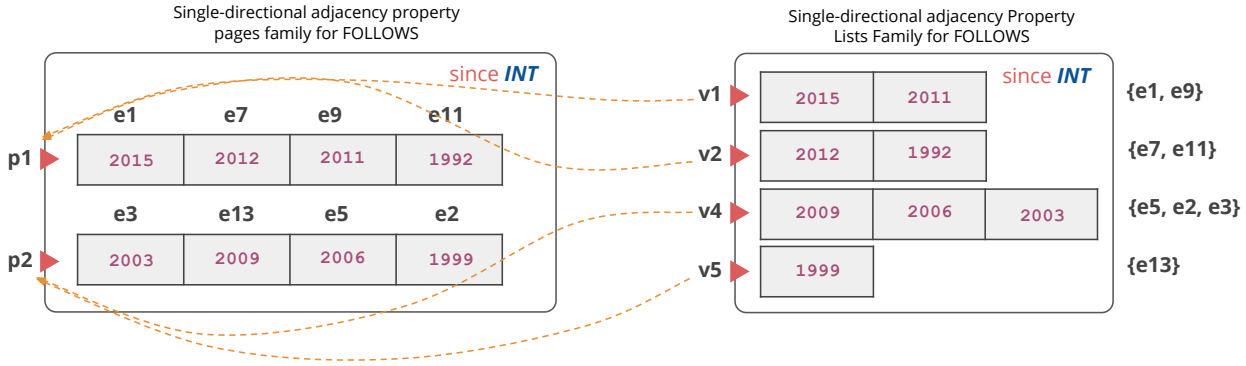


Figure 3.5: Mapping single-directional property lists to single-directional property pages for since property in Figure 2.1.$n = 2$.

offsets for each vertex, so up to $|V|$ many recycled IDs list.

Single-directional property pages: To address the above limitation, we take k property lists (by default 64) and store their properties in a property page. Properties in a property page is not necessarily stored consecutively. However, because we use a small value of k and adjacency lists of many vertices are short in many real world datasets, these properties are stored in close-by memory locations. We do not sort these pages when the original adjacency lists are sorted and keep a recycled ID list for each page, avoiding the cost of sorting and the maintenance of list-level recycled ID lists. We can use the same edge ID scheme we described above, except the positional offsets now identify the properties of edges in property pages, instead of property lists. Suppose again that the property $q_{i,j}$ is stored in the forward direction (though properties of k lists are grouped). Given an edge e in the scheme from Figure 3.4, we can take the source vertex's label-level positional offset and divide it by k to identify the page in which e's $q_{i,j}$ property is. Within this page, the property can be accessed with a single lookup using the positional offset of e.

Figure 3.5 shows the mapping of a single-directional property lists to single-directional property pages for $n = 2$. The single-directional property pages for FOLLOWS edges has two pages that stores the property values from vertex groups $(v1, v2)$ and $(v3, v4)$ respectively, assuming that the local positional offset of vi is i and pi is the ith property page for `since` property. In Section 6.3, we show that the single-directional property pages is similar, about 1.2x slower, in performance to single-directional property lists in read-heavy stress tests and about 2.5x faster than using edge columns.

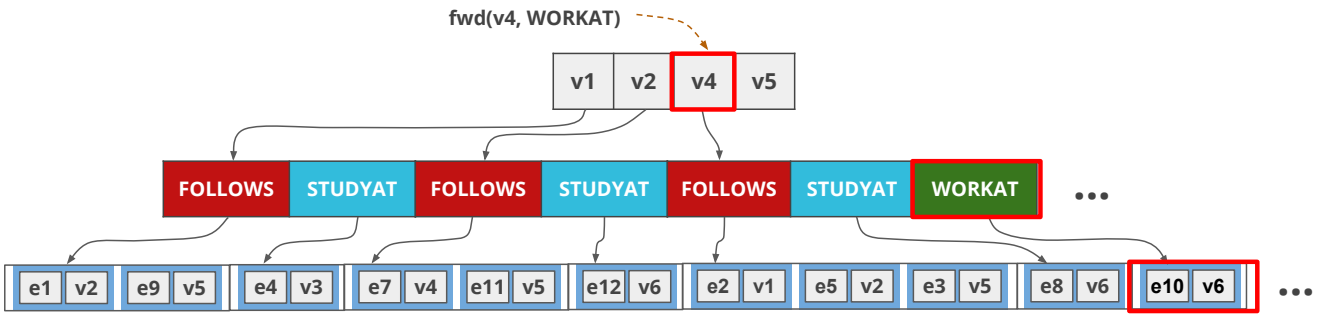


Figure 3.6: Forward adjacency lists implemented as a 2-level CSR structure for the example graph.

3.3 CSR Adjacency Lists

In GraphflowDB, the adjacency lists of a vertex are further partitioned by edge labels and sorted by neighbour vertex ID to support matching cyclic queries that need to join multiple edges at a time in the `JOIN` operation. We implement the adjacency lists in a 2-level CSR structure. CSR can be thought of as a columnar data structure that stored multi-value attributes. Figure 3.6 shows the physical data layout of forward adjacency lists in our example graph. The adjacency list of each vertex is indexed by its vertex ID, which is the list of sequential positional offsets for each label. This is implemented as an array of offsets to the first level of CSR that contains one entry for each edge label that a particular vertex has edges of. Similarly, this first level of CSR holds offsets to the next level of CSR that consists of a list of (edge ID, neighbour vertex ID) pairs that are sorted by the neighbour vertex ID. Each sublist in the final CSR is the set of edges of a particular vertex and having a particular label. This sublist sits at the depth of 2 in the hierarchy and hence, can be accessed by 2 indirections.

GraphflowDB represents edge and vertex ID as 8 byte values which means that each edge in the adjacency lists has the payload of 16 bytes. Our new vertex and edge ID scheme compacts this representation. As show in Section 4.2, the payload for each edge in the adjacency lists can be significantly reduced by exploiting the structures in graph that we describe in Guideline 5. With our set of optimizations, we end up reducing the size of this payload to only 4 bytes at times.

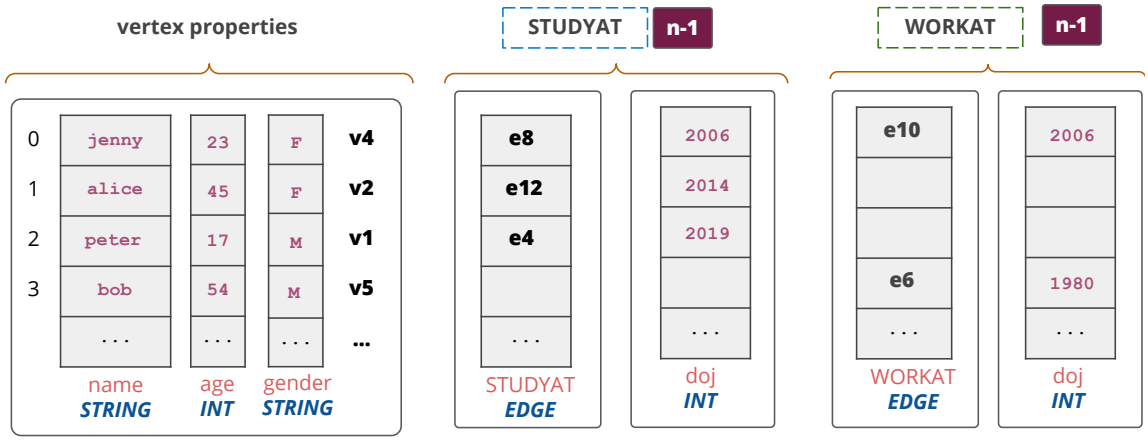


Figure 3.7: Storage of edges having single cardinality edge label `STUDYAT` and `WORKAT` as special property of `PERSON`.
Properties of these edges are also stored in vertex columns of `PERSON`.

3.4 Vertex Columns for Single Cardinality Edges

When an edge label has 1-1, 1-n, n-1 cardinality then vertices can have at most one edge in at least one direction. We refer to these edges as *single cardinality edges*. For instance, in our Figure 2.1 example graph, edge label `STUDYAT` and `WORKAT` have the cardinality `n-1`, i.e., a `PERSON` vertex can have at most one `STUDYAT`'s and `WORKAT`'s edge (in the forward direction). Therefore, instead of storing these single edge in the adjacency lists in last level of the 2-level CSR structure, we can store them as a *property* of source or(and) destination vertex and directly access them using the positional offsets of the source or(and) destination vertex of the edge. This both saves space, as we do not need to store the CSR offsets, and we can directly access these edges with 1 instead of 2 random lookups in a CSR. Similarly, the properties of these edges can also be stored in vertex columns as a property of either the source or destination vertex. Thus, we do not need page-level postional offsets for these single edges since the edge property can be accessed using the source or desitination vertex's positional offset, similar to how the edge is accessed. Figure 3.7 shows the edges of edge label STUDYAT and WORKAT, from our example graph, and their properties being stored as the special property of vertex type `PERSON`.

Data		Columnar data structure
Vertex Properties		Vertex Columns
Edge Properties	1-1	Vertex Column of either source or destination vertex label
	1-n	Vertex Column of destination vertex label
	n-1	Vertex Column of source vertex label
	n-n	Single-directional Property Pages
Edge	1-1	Vertex Columns of source vertex for forward edge Vertex Column of destination vertex label for backward edge
	1-n	CSR Adjacency lists for forward edge Vertex Columns of destination vertex label for backward edge
	n-1	Vertex Column of source vertex label for forward edge CSR Adjacency lists for backward edge
	n-n	CSR Adjacency lists for forward and backward edges

Table 3.1: Our Columnar data structures and the component of the property graph for which they are used.

3.5 Summary

Table 3.1 presents the summary of our columnar data structures. Each of our structure support prefixSum-based null compression, defined in Section 4.3, to compress null values or empty adjacency lists.

Chapter 4

Columnar Compression

Compressing data stored in columns and query processing directly on compressed data have been extensively studied in the context of column-oriented RDBMSs. These techniques are directly relevant to our work. However, not all columnar compression techniques can be directly applied to the data-structures that we introduced in Chapter 3. Recall our Desideratum 2 that in the context of in-memory GDBMSs, we are interested in compression schemes that either avoid decompression at all or decompress arbitrary single elements in a compressed block in constant time. For example, schemes such as run-length encoding are not suitable for in-memory GDBMSs because it is not possible to decompose the value of an arbitrary element in constant time. We begin in Section 4.1 by reviewing a number of existing compression techniques that satisfy this constraint and where they can be applied in GDBMSs. In Section 4.2 we discuss opportunities where we can compress our adjacency lists, i.e., the storage of (edge ID, neighbour ID) pairs, given our new ID schemes. Finally, in Section 4.3, we discuss the shortcomings of directly applying existing null compression schemes from columnar RDBMSs to compress null values or empty lists in GDBMSs and propose a new *prefixSum-based null compression* scheme that is suitable for GDBMSs.

4.1 Directly Applicable Compression Techniques

Random access to a compressed column is possible only if the elements of a column are encoded in *fixed-length codes*, i.e. using a fixed number of bits, instead of variable-length codes. Several existing schemes, such as dictionary encoding, leading 0 suppression, bit vectors, and frame of reference produces fixed length codes. We review dictionary encoding and leading 0 suppression below, which we have integrated in our implementation. We refer

the readers to references [5, 27, 37] for details of other schemes. We next review several of existing techniques that produce fixed length codes.

Dictionary encoding: The dictionary encoding is perhaps the most common encoding scheme to be used in RDBMSs [5, 56]. At a high-level, this scheme maps a domain of values into more compact and shorter representations using a variety of schemes [56, 29, 5]. Some of these schemes produce variable-length codes, such as Huffmann encoding, and others use fixed-length codes, such as the one described in reference[5]. These techniques are often used to encode `STRING`s into 64-bit integer values or any categorial property into a small number of fixed-length bits. Different schemes can further pack these bits into bytes. In our implementation, we use dictionary encoding to map any categorial property p that takes on z different many values to $\lceil log_2(z)/8 \rceil$ bytes (so we pad $log_2(z)$ bits with 0s to have a fixed number of bytes).

Leading 0 Suppression: Given a block of data, this scheme omits storing leading zero bits or in each value in the block [13]. Each value is, hence, encoded in a variable number of bits along with the count of bits used for storing the value. For an integer value 12, the number of bits in which it can be stored is 4, instead of default 32. There are both variable-length and fixed-length variants of this scheme. We adopt a fixed length version where for storing labels and positional offsets in both edge and vertex IDs (though it is trivial to apply to vertex and edge properties as well). Specifically, if there max_v many vertices of a particular label, we store the positional offsets in $\lceil log_2(max_v)/8 \rceil$ many bytes instead of 8 bytes. Similarly, if the maximum size of a property page of an edge label is k, for all for all the associated edges, we use $\lceil log_2(k)/8 \rceil$ many bytes for the page-level positional offset of the edge ID. Often the forward adjacency lists of many edge labels are quite small (even the maximum length one), and we store the positional offsets with a few bytes, instead of 4 bytes. Similarly if there are max_ℓ many different labels, we store labels as in $\lceil log_2(max_\ell)/8 \rceil$ many bytes. That said, during query processing and accessing properties, the labels, positional offsets of vertices, and positional offset of edges are stored as 4, 8 and 4 bytes, respectively.

4.2 Compressed Storage of Edge and Vertex ID Pairs in Adjacency Lists

We next discuss how to compress the edge ID and vertex ID pairs in the adjacency lists. Our new ID schemes from Sections 3.1 and 3.2 decompose the IDs into multiple small components, which enables us to factor out most of these components, when the data

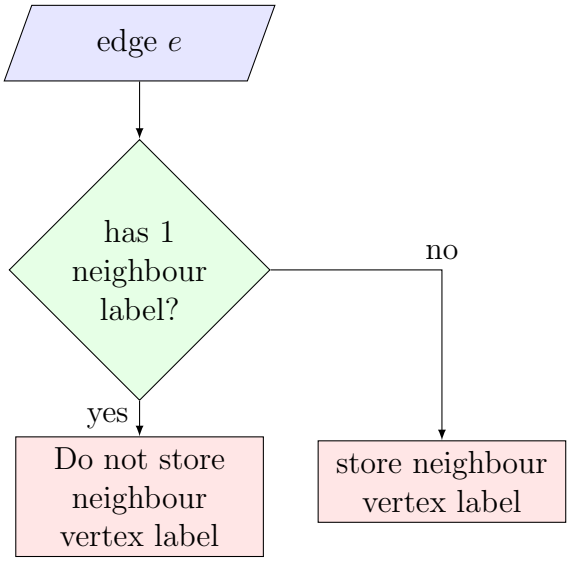


Figure 4.1: Decision tree to store neighbour vertex's label in the Adjacency lists.

depicts some structure.

Recall that the ID of an edge e in our new edge ID scheme contains 4 components: (i) edge label; (ii) source vertex ID; (iii) destination vertex ID; and (iv) positional offset of the properties of e in property pages. Recall also that then ID of a vertex v contains 2 components: (i) vertex label; and (ii) (label-level) positional offset. First, both the source and the destination vertex IDs inside the edge ID of (edge ID, neighbor ID) pairs can be omitted. This is because the source (destination) vertex ID is implied by the offset of the pair in the forward (backward) adjacency list and the destination (source) vertex ID is the neighbor ID, which is already stored in the pair. Second, we do not have to store the labels of the edges as in our storage of the adjacency lists because recall that we store a different CSR-like structure for each edge label. Therefore the only parts that need to be stored are positional offsets for edge IDs, and vertex label and positional offsets of neighbor IDs. We next discuss further cases when the structure and multiplicities of edges allows us to factor some of these components out:

- **Edge label determines a single neighbour vertex label.** In this very frequent case, an edge label is between pairs of nodes when the sources or destinations (or both) can have a single label. For instance, in our example graph, `FOLLOWS` edges are between vertices having the label `PERSON`. In this case, we can factor out the vertex label component of neighbor ID.

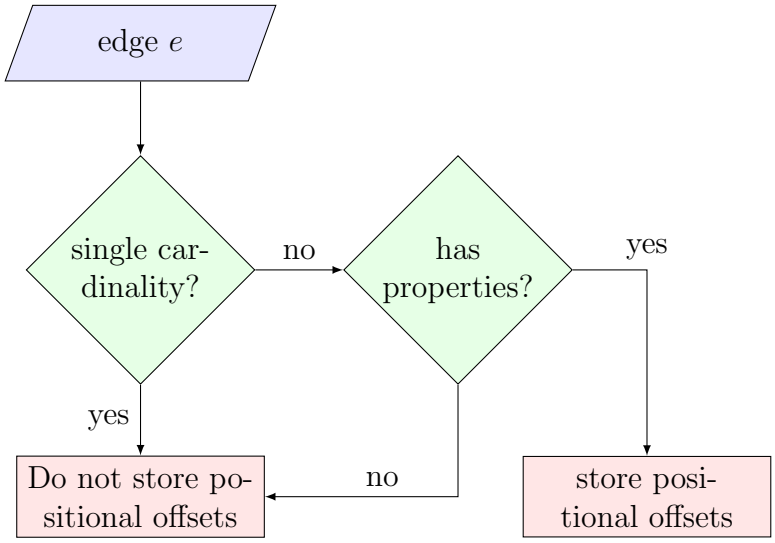


Figure 4.2: Decision tree to store edge's positional offset in the Adjacency lists.

- **Edges do not have properties:** This is also a frequently appearing case. That is the edges with a particular label do not have any structured or unstructured properties. In this case, notice that the edges do not need to be identifiable at all as the system will never access properties of these edges. Therefore, what distinguishes two edges are their neighbor IDs and edges with the same IDs are simply replicas of each other and are stored twice. Therefore, we can omit storing the positional offsets of edge IDs.
- **Single cardinality edges:** Recall from Section 3.4 that the properties for single cardinality edges can be stored in vertex property columns. So by using the source or destination vertex IDs, we can directly read properties. Therefore, we can omit storing any positional offsets.

All of the above cases arise frequently in real world graph data. For example, in the LDBC SNB dataset 6.1, 10 out of 15 many edge labels determine a single source and destination neighbour label, 10 many do not have any properties, 8 many have single cardinality edges. In all, 10 many of them are not required to store any neighbour vertex's labels and positional offset. This implies that we only store neighbour vertex's positional offset for such edges in the adjacency lists, which can be stored with as few as 4 bytes.

Figures 4.1 and 4.2 summarizes these cases in a decision tree that instruct when to omit storing certain components.

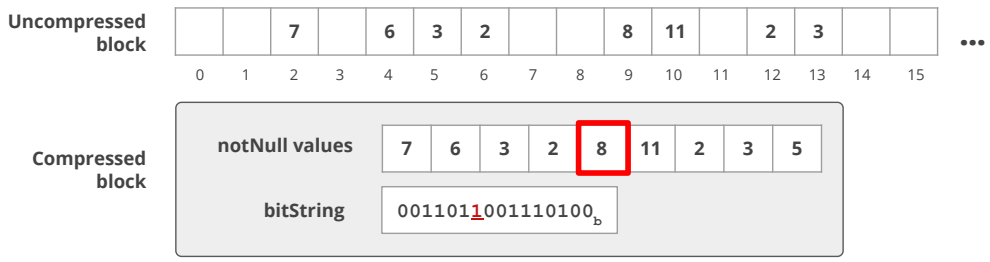


Figure 4.3: NULL compression using bit-Strings.

4.3 NULL Compression

Edge and vertex properties can often be quite sparse in real-world graph structured data. Similarly in many adjacency lists, due to the power-law nature of the degree distributions, a significant fraction of vertices can have empty forward or backward adjacency lists. Both the property sparsity and empty lists can be seen as different columnar structures containing null values, which can be compressed.

A general null compression technique is to treat the NULLs in the column as another potential value in the domain of column's datatype which could then be compressed by any of the columnar compression schemes. For example, a column that is very sparse ($> 95\%$ NULL values) can be effectively encoded using run-length encoding. This is not directly applicable in GDBMSs as it does not allow accessing an arbitrary location in a column and finding the value (or null if the value does not exist).

Abadi in reference [3] has described three specific NULL compression techniques. All of these techniques list all non-NULL elements consecutively in a block of data. Then to indicate the positions of these non-NULL elements, they use different techniques. The simplest is to list along with each value the position. Borrowing the terminology from reference [3], this technique is designed for sparse columns, i.e., whose significant fraction of values, say $> 90\%$ are NULL. A more compact way, designed for columns that have low sparsity, is to list in a separate array the beginning and end indices of consecutive non-NULL values. The third technique, designed for columns with intermediate sparsity, is based on using bit-string to indicate if each location is NULL or not. This is quite compact and requires only 1 bit extra storage per each cell in the block. Figure 4.3 shows an example column and its null compressed version using the bit-string scheme.

However, these techniques are not directly suitable for GDBMSs as they do not satisfy our Desideratum 2. In the bit-string method, we can in constant time learn whether or not if the value at a position i is NULL or not (e.g., i would be the positional offset of a vertex

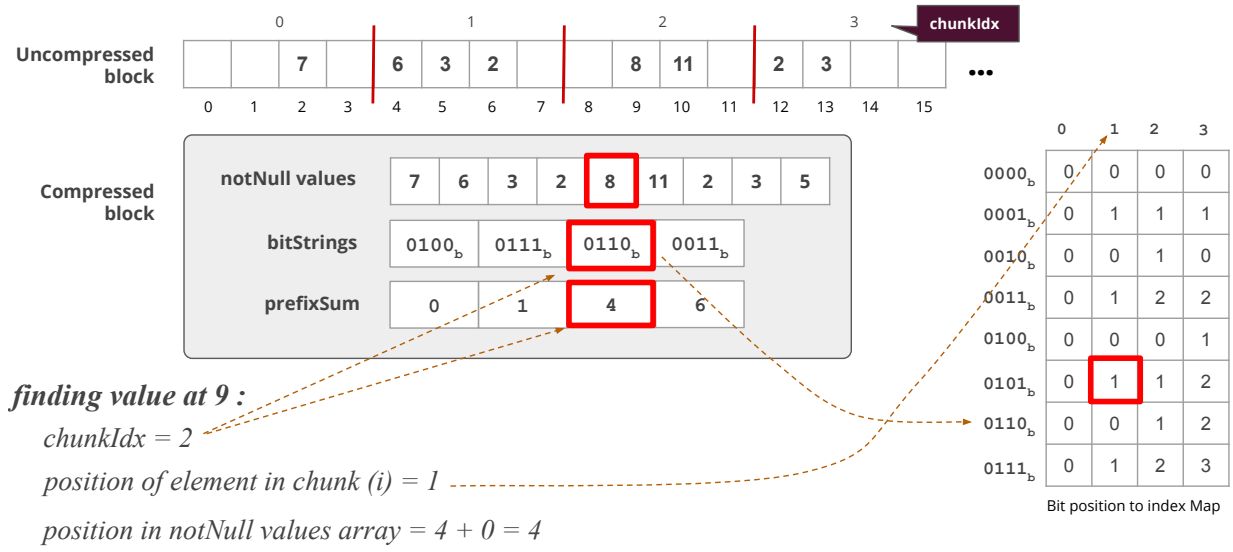


Figure 4.4: PrefixSum-based NULL compression scheme. Chunk Size (n) = 4

in a column storing a vertex property). However, if the value is not NULL, the system needs to iterate over the bits until location i and count the number of 1's to find out the location of the value. For instance, in Figure 4.3, accessing the element at index 9 of uncompressed block involves counting the number of 1's till before index 9 in the bit-string, which is 4. Thus, the value is then read from index 4 of the non-NULL values array.

We next present a modification to the bit-string-based compression scheme to satisfy our requirement of constant time access to arbitrary elements. In addition to the array of non-NULL values and the bit-string, we store a prefixSum for each c (16 by default) elements in a block of the column, i.e., we divide the block into chunks of size c. While the bit-strings indicate positions in the block with non NULL elements, the prefixSum holds the number of non NULL elements in the uncompressed block before a particular chunk. We also maintain a pre-populated static 2D map in the memory having size $(2^c, c)$. We call this *bit-position-to-index map*. Let b and p respectively be the bit-string of length c and prefixSum of a chunk j. Given b and the position i in the bit-string, the map returns the number of 1's in b until position i. The map allows us to avoid iterating over the bit-string to count the number of 1s and perform this count with a single lookup. Then, by adding the value returned from the map to p, we get the exact location of the value of location i in j (so the location of $j * c + i$ in the entire block). In total, after checking that the value is non-NULL, which needs to be done in any bit-string based scheme, we perform 1 lookup of prefix, 1 look up in the map, and 1 arithmetic, before the final look up of the actual

value in the non-NULL values array.

Figure 4.4 depicts compression of a sparse column using our modified scheme. As an example, suppose we need to find the element at index 9 of the uncompressed block. Given $c = 4$, this element will appear in the 2nd chunk, say c_2, of the compressed block. The position i of index 9 in c_2 is 1. Also, c_2's bit-string and prefixSum are 0110_b and 4 respectively. The entry for (0110_b, 1) in the bit-position-to-index map is 0. Thus, the index of element at 9 in non-NULL values array is $4 + 0 = 4$.

The choice of the value of c affects how big the bit index to position map is. For $c = 16$, the size of the map is 1MB. The overhead of bit-string and prefixSum can also be optimized. Since the bit-string takes a bit for each element in an uncompressed block, the overhead depends on the size and number of prefixSums we have for an uncompressed block. By default we set c to 16 and maintain blocks of size $N = 2^c$ (so about 64K cells) and store our prefixSums as 16-bit unsigned integers. This ensures that the prefixes we keep increase the overhead from 1 to only 2 bits per element in the compressed block. If the size of prefixSum is w and $c = 16$, the overhead from prefixSums per element will be $w/16$ bytes. w itself depends on the number of elements N in the uncompressed block. For $N = 2^{16} = 64K$, $w = 16$ and hence, the overhead of prefixSum is 1-bit per each cell in the block.

In our implementation, we use the prefix-sum based NULL compression schemes to compress vertex and edge properties (so vertex property columns and single-directional property pages) as well as empty lists in adjacency lists.

Chapter 5

List-based Query Processing

In this chapter, we briefly review the traditional Volcano-style query processing technique, which is common across a wide range of different database management systems, and the column- or vector-oriented processing technique that is particularly suited for column-oriented relational database management systems. We then describe the limitations of these techniques for GDBMSs and propose a technique which we call list-based processing, that is a hybrid between the two.

5.1 Existing Techniques

In the Volcano-style query processors [28] execution happens by passing a single tuple between operators. Specifically, each operator operates on a single intermediate tuple, e.g., a partial match of the subgraph query, extends or modifies one value in this tuple and passes it to the next operator in the query plan. Consider the following query and a plan for this query shown in Figure 5.1.

Example 2. *Consider the following query.*

```
MATCH (a:PERSON)−[ex:FOLLOWS]→(b:PERSON),
      (b:PERSON)−[ey:FOLLOWS]→(c:PERSON)
WHERE a = v4
RETURN ey.since
```

In a GDBMS that adopts the Volcano-style processing, a scan operator could match the a variable to v_4 then give the partial match $[a = v_4]$ to the next join operator jo_1.

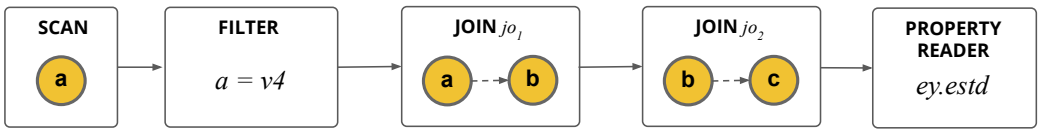


Figure 5.1: Query plan for Example 2.

Suppose v_4 has k many outgoing edges, say $(e_{41}, v_{41}), ..., (e_{4k}, v_{4k})$. jo_1 would read the first edge (e_{41}, v_{41}) and give the tuple $[a = v_4, e_x = e_{41}, b = v_{41}]$ to the next join operator jo_2. jo_2 would then read the first outgoing edge of v_{41}, say (e_{411}, v_{411}), and give the tuple $[a = v_4, e_x = e_{41}, b = v_{41}, e_y = e_{411}, c = v_{411}]$ to the next property reader operator, so and so forth. One advantage of Volcano-style processing is that even if there are many intermediate tuples with a particular variable value, this value is copied only once to the tuple that is passed between operators. For example, even if v_{41} has z_1 many outgoing edges in the input graph, so there will be z_1 many intermediate tuples with $b = v_{41}$, this value is copied to the tuple only once. This is an important advantage for GDBMSs where 1-to-many joins, specifically one vertex joining to many neighbor vertices, are common. At the same time, it is well known that Volcano-style processors do not achieve high CPU cache locality. For example although $(e_{41}, v_{41}), ..., (e_{4k}, v_{4k})$ are stored consecutively in memory, reading and processing these values are intermixed with function calls to the other operators which might read values from other adjacency lists. The second shortcoming of Volcano-style processors is that they perform many function calls between operators.

Column-oriented relational systems have addressed the shortcomings of Volcano-style processors by introducing column- and vector-oriented processing [15, 32]. Instead of a tuple-at-a-time processing, these techniques pass an entire column or vector, e.g., 1024 tuples, at a time between operators. Each primitive operator in the system takes a vector of tuples and outputs a vector of tuples. While processing a vector, each operator can read consecutive memory locations, achieving good cache locality, and employ efficient block algorithms and tight loop over arrays that draw benefits from advanced compiler optimizations and SIMD instructions that are common in modern CPUs. Vector-at-a-time processing also reduces the number of function calls that are made during query processing. One shortcoming of vector-oriented processing is that, for 1-to-many joins, it requires copying intermediate values. For example in a vector-oriented processor, the scan operator would output $a : [v_4]$ vector, the first join operator jo_1 would output $a : [v_4, v_4, ..., v_4]$, $e_x : [e_{41}, ..., e_{4k}]$, $b : [v_{41}, ..., v_{4k}]$ vectors, and the second join operator jo_2 would output $a : [v_4, v_4, ..., v_4]$, $e_x : [e_{41}, ..., e_{41}, e_{42}, ..., e_{42}, ..., e_{4k}, e_{4k}]$, $b : [v_{41}, ..., v_{41}, v_{42}, ..., v_{42}, ..., v_{4k}, v_{4k}]$, $e_y : [e_{411}, ..., e_{41z_1}, e_{421}, ..., e_{42z_2}, ..., e_{4k1}, e_{4kz_k}]$, $c : [v_{411}, ..., v_{41z_1}, v_{421}, ..., v_{42z_2}, ..., v_{4k1}, v_{4kz_k}]$,

assuming vertices v_{41}, v_{42}, ..., v_{4k} have z_1, ..., z_k neighbors, respectively (and assuming $z_1 + ..., + z_k$ is less than the fixed vector size). While vector style processing is good for aggregation heavy workloads that require reading columns of data consecutively, it is not particularly suited for workloads that contain many 1-to-many joins due to the data duplication it requires.

5.2 List-based Processing

We have developed a new query processing technique, which we call *list-based processing*, that is a hybrid between Volcano-style and vector-oriented processing. In particular, we have two versions of each operator, a Volcano-style and a vector-oriented version. For example, we have a `JOIN` operator that takes an intermediate tuple and copies one value of one adjacency list and produces one tuple. We also have a `LIST JOIN` operator that takes a tuple and produces an output tuple that contains a copy of an adjacency list that is used in extending. We do lazy evaluation, so until it is necessary, the list is simply a pointer to the adjacency list that should be extended and is not materialized. Similarly we have a `PROPERTY READER` and `LIST PROPERTY READER` operators, where the latter takes a tuple where one value is an adjacency list containing a list of edges and neighbors, and copies over a property of the edges or neighbors. For example, Figure 5.2 shows the query plan that operates on lists rather than single values. The last join operator is `LIST JOIN` followed by the `LIST PROPERTY READER` for reading property of e_y edge matches. The execution is performed Volcano-style until the `LIST-JOIN` operator, which for example would get the tuple $[a = v_4, b = v_{41}]$, and outputs $[a = v_4, b = v_{41}, c = \{(e_{411}, v_{411}), ..., (e_{41z_1}, v_{41z_1})\}]$ as output. The `LIST PROPERTY READER` could then output $[a = v_4, b = v_{41}, c = \{(e_{411}, v_{411}), ..., (e_{41z_1}, v_{41z_1})\}, e_y = \{1980, 1981, ..., 1990\}]$.

The list operators perform vector-oriented processing with tight loops, but the vector size is not fixed and is the size of an adjacency list. For example, the `LIST PROPERTY READER` would directly access a property page and read all of the properties of $e_{411}, ..., e_{41z_1}$ inside a single loop. The `LIST JOIN` operator and the following list operators are only used when no other `JOIN` operator needs to extend the values in the list produced by `LIST JOIN`. For example `LIST JOIN` is used in the final join operators in a plan, e.g., jo_2 in our example, but it can also be used in earlier join operators, as long as the output list of `LIST JOIN` will not be extended in another join operator. The final plans therefore consists of a mix of Volcano- and vector-oriented operators. Similar to Volcano-style processing, list-based processing does not copy the same value into the tuple multiple times and also benefits from vector-oriented processing, specifically for the operators that are at the leaves

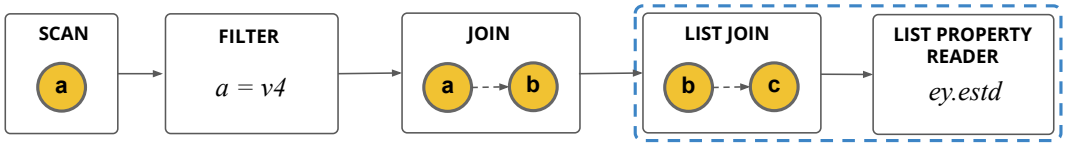


Figure 5.2: Query plan with List-based Processing for Example 2.

of the query plans. We note that in many queries, most of the work is done by the join operators in the leaf, so an important part of the processing is often still performed in a vector-oriented style. As we will demonstrate in Chapter 6, list-oriented processing can outperform both Volcano-style and vector-oriented processing. This is especially the case when queries contain multiple 1-to-many joins, corresponding to multi-hop traversals of graphs.

We note that our list-oriented processing is a simple form of *factorized query processing* [41], that evaluates queries in a compressed format. For example, the tuple $[a = v_4, b = v_{41}, c = \{(e_{411}, v_{411}), ..., (e_{41z_1}, v_{41z_1})\}, e_y = \{1980, 1981, ..., 1990\}]$ is a compressed representation of z_1 many tuples. For some queries, e.g., a star query, this type of processing can significantly decrease the amount of intermediate data that is processed. A rigorous study of factorized query processing in GDBMSs is not in the scope of this thesis and is left for future work.

Chapter 6

Evaluation

We integrated our columnar storage and techniques into the in-memory GraphflowDB GDBMS. The version of the system we modified stored the edge and vertex properties in a row-oriented fashion as a sequence of variable-sized records indexed by their IDs and partitions the edges by labels and stores them as 8-byte vertex and edge IDs inside CSR. The goal of our experiments is two-fold. First, we show that using columnar storage and compression techniques reduces the memory consumption of the system significantly, by 3.5x, when storing a graph dataset generated by the popular LDBC social network benchmark. Second, we evaluate the query performance benefits and trade-offs that these techniques provide. In particular, we organize our experiments as follows:

1. **Compression in Adjacency Lists:** In Section 6.2 we show the reduction in the size of our adjacency lists when applying the storage optimizations from Sections 3 and 4. For each optimization, we state the cause of the reduction in size and query performance effect of the optimization.

2. **Single-directional Property Pages:** In Section 6.3, we compare the performance of storing edge properties in single-directional property pages, as compared to strictly-ordered single-directional property lists and unordered edge columns.

3. **Vertex Columns for Single Cardinality Edges vs. CSR Adjacency Lists:** In Section 6.4, we show the effectiveness of storing single cardinality edges in vertex columns as compared to in CSR format.

4. **Prefix Sum-based Null Compression:** In Section 6.5, we evaluate the size and performance trade-off of compressing our columns with our prefix sum-based null

compression scheme. We also compare our technique with the vanilla null compression technique mentioned in [3].

5. **List-based Processing vs Volcano-styled Query Execution:** In Section 6.6, we show the performance benefits of using list-based processor over Volcano-styled processor for executing queries.

Noticeably missing in our evaluations is comparisons against the other GDBMS systems and conventional column stores, which we plan to perform as part of future work.

6.1 Experimental Setup

Hardware Setup: For all our experiments, we use a single machine that has two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical cores. We only use one logical core. We set the maximum size of the JVM heap to 500 GB and keep JVM's default minimum size.

Dataset: We use the LDBC SNB [23] dataset generator to generate synthetic graph datasets on which we evaluate all our experiments. LDBC SNB is a popular benchmark to generate large social graph data with multiple vertex and edge labels, and properties on vertices and edges. The generated social graph is highly structured which can be seen from the schema in Figure 6.1. In fact all of the edges and edge and vertex properties are structured according to our definition from Chapter 2 but several properties and edges are very sparse. For example, property `language` and `content` on vertices with type `STRING` appears in less than 15% of the nodes with label `post`. We generate the data at the scale factor of 100 that consists of over 1.7 billion edges and 0.3 billion vertices. We refer to this dataset as LDBC100.

Query Workload: We use a micro benchmark we generate that consists of 1-, 2-, and 3-hop path queries that optionally contain predicates on vertex and edge properties and aggregations. These queries serve as a stress test for evaluating access to the underlying storage, which our techniques optimize.

6.2 Compression in Adjacency Lists

In this experiment, we demonstrate the memory reduction we get from the columnar storage and compression techniques we described in this thesis that reduce the cost of storing

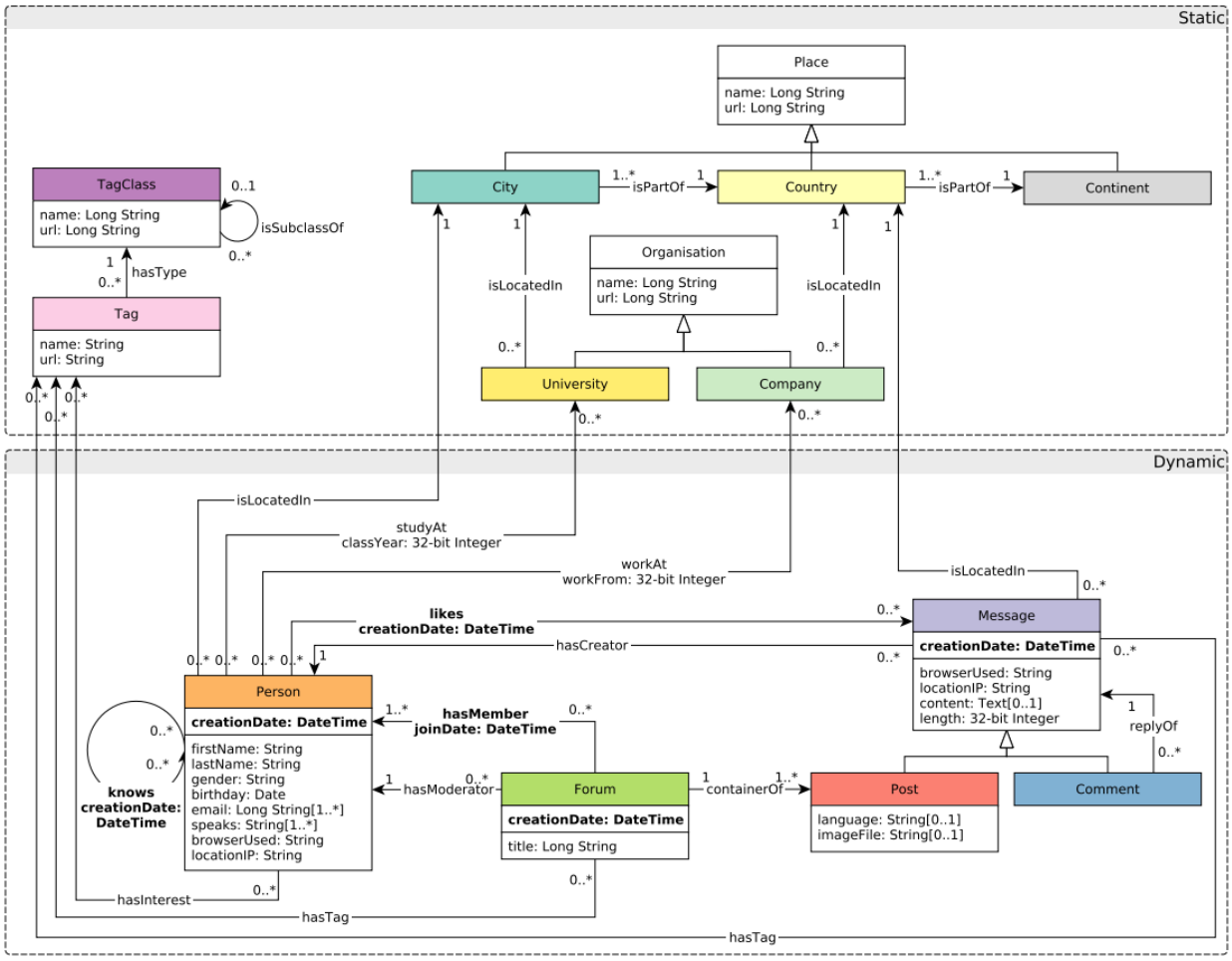


Figure 6.1: LDBC SNB Graph schema. (Obtained from LDBC SNB specification document v0.3.2)

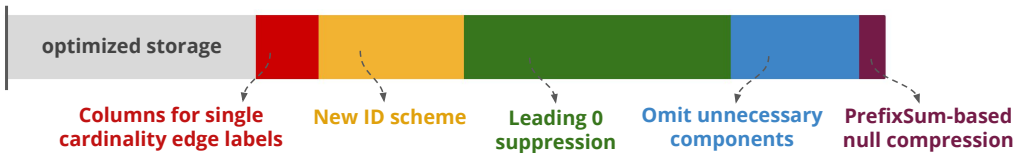


Figure 6.2: Breakup of memory gains by applying different optimizations on Adjacency Lists of LDBC100 dataset.

(edge ID, neighbour vertex ID) pairs in adjacency lists.We create multiple configurations on GraphflowDB, each with a different set of optimizations. Below are the descriptions of the configurations we evaluate the memory usage on. Each configuration builds on top of previous in the list.

1. `GF-OLD`: This is our baseline configuration that represents edges and vertices in the adjacency list as 8-byte identifiers. All the edges are stored in the 2-level CSR structure and are not compressed.
2. `+COLS`: Uses vertex columns to store edges with single cardinality edge labels instead of CSR adjacency lists.
3. `+NEW-IDS`: Introduces our new vertex and edge identification schemes that stores vertex and edge ID as a set of multiple small components.
4. `+0-SUPR`: Implements leading 0 suppression in the components of vertex and edge IDs in adjacency lists.
5. `+OMIT`: Omits neighbour vertex label and positional offsets of edges in the adjacency list or vertex column storage, when they can be inferred from the structure.
6. `+NULL`: Implements prefix sum-based null compression on adjacency lists (or vertex columns for single cardinality edges).

Table 6.1 shows how much memory (in GB) the adjacency lists take when storing the edges of LDBC100 across different configurations. Figure 6.2 gives the breakup of memory gain per optimization. Memory gain in `+COLS` is attributed to the fact that 8 out of 15 edge labels are single cardinality and stored in vertex columns in at least one direction. `+NEW-IDS` gains (~4 bytes per edge) by storing edges in smaller than 8 bytes, while `+OMIT` gains (~3 bytes per edge) from not storing edge's positional offsets and neighbour's vertex

	GF-OLD	+COLS	+NEW-IDS	+0-CMPRS	+OMIT	+NULL
Fwd. Adjacency Lists	38.25	33.25 **+1.15x**	27.22 **+1.22x**	16.35 **+1.66x**	11.14 **+1.47x**	10.53 **+1.06x**
Bwd. Adjacency Lists	37.93	37.50 **+1.01x**	30.93 **+1.21x**	18.75 **+1.65x**	12.79 **+1.47x**	11.15 **+1.15x**
Total (GB) Bytes Per Edge	76.18 23.04	70.75 21.39 **1.08x**	58.15 17.58 **1.31x**	35.10 10.61 **2.17x**	23.93 7.24 **3.18x**	21.68 6.50 **3.55x**

Table 6.1: Memory utilization (in GB) by Adjacency lists of LDBC100 when adding our optimizations one at a time. Each column i indicates an optimization i and in the rows for forward and backward lists indicate the additional reduction factor of applying optimization i on top of the previous optimizations to the left of i. In contrast, in the row on total memory consumption/bytes per edge, each column i indicates the cumulative reduction factor (compared to GF-OLD) of applying all optimizations from left until i.

label for 10 out of 15 edge labels. We see modest benefits in +NULL (~0.75 bytes per edge) since empty adjacency lists are infrequent in LDBC100.

All of these optimizations also improve query performance, with the exception of +NULL, which incurs a modest query slow down. We will evaluate the performance gains and trade-offs of +COLS and +NULL in Sections 6.4 and 6.5 respectively. We will not evaluate the benefits of leading 0 suppression but we note that this also improves performance because we do not have to decode a particular value and hence, and have modest gains because we copy less data into tuples from the adjacency lists. Similarly for the cases of +NEW-IDS and +OMIT, we gain by less copying of data into the tuple.

6.3 Effectiveness of Single-Directional Property Pages

In this experiment, we show the benefits of keeping the edge properties grouped and loosely-ordered in single-directional property pages compared to using edge columns that store the edge properties unordered. Recall that one alternative design we described was using single-directional property lists to store edge properties. This design had the advantage that edge properties would be kept in exactly the same order as they appear in adjacency lists, but it can make updates significantly slower, specifically for systems that keep edges

	1-hop	2-hop	3-hop
EDGE COLS	3.42	308.18	966.04
PROP PAGES	1.22 **2.80x**	152.89 **2.03x**	512.24 **1.87**
PROP LISTS	1.03 **3.32x**	125.65 **2.42x**	372.44 **2.59**

Table 6.2: Runtime (in sec) of k-hop queries for different configurations of edge property storage on LDBC100.

in sorted order. However, this design may still be a viable design for some GDBMSs and as a point of reference we also evaluate its benefits. To test the performance benefits of single-directional property pages and lists, we configure GraphflowDB in 3 different ways (all using our new ID schemes):

1. `EDGE COLS`: Stores edge properties in edge column in the order they were inserted into the database, so properties of edges in a particular adjacency list (forward or backward) can appear anywhere in this column. We ensure that the insertion order is random and not adhering to the order of edges in the forward or backward adjacency lists.

2. `PROP LISTS`: Edge properties are stored in the single-directional property lists. We pick the forward list for $n - n$ multiplicity edges. Therefore these property lists mimic the forward adjacency lists and gives sequential reads when edge properties of a forward adjacency list are read.

3. `PROP PAGES`: Edge properties are stored in pages by combining $n = 128$ property lists of the previous solution and appear in insertion order. This solution provides close-by reads when edge properties of a forward adjacency list are read.

As our workload, we use 1- 2- and 3-hop queries that use the `knows` edge label of the LDBC SNB schema (Figure 6.1) that compare each query edge's `creationDate` property to be greater than the previous edge's. `knows` edge label has `n-n` cardinality with an average of 44 outward `known` edges from each `person` vertex. This gives us significant scope for reading edge properties from close-by memory locations for each adjacency lists. 1-hop and 2-hop queries run for all vertices of `person` while we run 3-hop for only 25000 vertices

to make the query complete faster. For each query, we only consider the plan that matches vertices from left to right in the forward direction. This way, we are ensure cache locality in PROP-LISTS and PROP-PAGES configurations. Table 6.2 shows the performance of queries on 3 configurations. We observe that the queries benefit significantly from localizing edge properties in the storage. Compared to EDGE COLS, we see up to 2.8x improvement for PROP PAGES and up to 3.32x improvements for PROP LISTS. Recall also that using PROP PAGES or PROP LISTS also reduces our storage because the amount of bytes required to identify edges within a list or a property page is smaller than the bytes needed to identify an edge within an entire edge column. In a system that uses fixed size IDs, which in-memory systems optimized for performance should do, 8 bytes is needed to identify edges in billion-scale graphs while 4 bytes is sufficient to identify edges within a list or property page. Finally, note that the benefits we get from PROP PAGES will depend on n, in particular as n approaches 1, PROP PAGES design reduces to PROP LISTS. We have not performed this experiment but as n gets smaller we expect the performance differences between PROP LISTS and PROP PAGES to reduce.

6.4 Vertex Columns for Single Cardinality Edges vs CSR Adjacency Lists

Storing single cardinality edges in vertex columns ensure two benefits: (i) direct access into the edge without indirection into CSR, and 2) does not need to store offsets of the CSR. We showed the memory gains of storing edges in vertex columns for LDBC100, in Section 6.2, on the whole. We next compare the performance benefits of using vertex column vs adjacency lists in CSR format under two settings: (i) when empty lists (or edges because of single cardinality) are not null compressed; and (ii) when they are null compressed. We create 4 configurations of GraphflowDB to run our queries on:

1. V-COL-UNC: Single cardinality edge label edges are stored in vertex columns and are not compressed. This is equivalent to +OMIT configuration in Section 6.2.
2. CSR-UNC: Single cardinality edge label edges are stored in CSR format and are not compressed.
3. V-COL-C: Null compressed version of V-COL-UNC. This is equivalent to +NULL configuration in Section 6.2.
4. CSR-C: Null compressed version of CSR-UNC.

	1-hop	2-hop	3-hop	Memory (in MB)
CSR-UNC	7.03	9.13	9.60	1266.56
V-COL-UNC	4.34 **1.62x**	5.80 **1.57x**	5.85 **1.64x**	839.93 **1.51x**

(a) Uncompressed

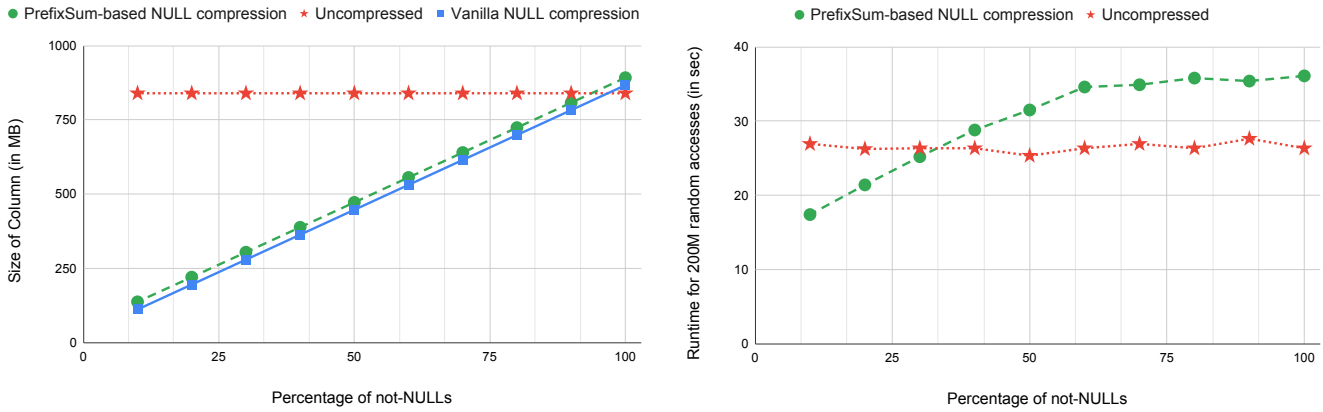
	1-hop	2-hop	3-hop	Memory (in MB)
CSR-C	7.78	10.40	11.23	905.23
V-COL-C	5.23 **1.49x**	8.28 **1.26x**	8.41 **1.34x**	478.86 **1.89x**

(b) Null Compressed

Table 6.3: Vertex property columns vs. 2-level CSR adjacency lists for storing single cardinality edges: Query runtime (in sec) and Memory usage (in MB)

The workload consists of simple 1-, 2-, and 3-hop queries on the `replyOf` edge between `comment` vertices in the LDBC SNB schema. The `replyOf` edge label has `n-1` cardinality, hence, we keep the forward edges as a special property of `comment` vertex label. Moreover, our workload queries do not do any predicate evaluation and the final output of the query is an aggregated count. This assures that `JOIN` operation in the query plan is the only dominant operation. Again, for each query, we evaluate only on the plan that matches the vertices sequentially and joins in the forward direction.

Tables 6.3a and 6.3b shows the result of queries on uncompressed and null compressed configurations respectively. We observe up to 1.62x performance gains between uncompressed variants of vertex columns and CSR (i.e., `V-COL-UNC` vs `CSR-UNC`) and up to 1.49x gains between null compressed variants (i.e., `V-COL-C` vs `CSR-C`). The last column of the tables report the size of the adjacency lists or vertex column storing `replyOf` edges. Here, vertex column uses half as much space as adjacency lists, when the data is kept compressed. In LDBC100, out of ~220M `comment` vertices 50.5% have empty forward adjacency list, i.e, do not have an outward `replyOf` edge. This is reflected in vertex columns between `V-COL-UNC` and `V-COL-C`, as the memory reduces by 1.75x (839.93MB vs 478.86MB), unlike their CSR counterparts that stores offsets as extra and thus, compression reduces memory by only 1.4x (1266.56 vs 905.23). These results verify that using vertex columns for single-cardinality edges not only saves space, but also improves query performance (irrespective of whether or not the edges/lists are null compressed or not).



(a) Memory usage

(b) Performance on random accesses

Figure 6.3: Memory and performance on random accesses for Uncompressed, prefix sum-based NULL compressed and vanilla NULL compressed columns.

6.5 Effectiveness of Prefix Sum-based Null Compression

We demonstrate the memory performance trade-off of prefix sum-based null compression when compressing both sparse property columns as well as empty adjacency lists. We evaluate two aspects of our technique: (i) storage and random access efficacy against uncompressed and vanilla null compressed columns; and (ii) query performance on compressed and uncompressed vertex columns and adjacency lists.

The goal of our first experiment is to compare uncompressed, prefix sum-based null compressed and vanilla null compressed (as implemented in [3]) columnar data-structure for memory usage and performance on random reads. We design a micro-benchmark to stress test the access performance when performing random accesses to a null compressed column. We use the `creationDate` property of 220M `comment` vertices in LDBC100 and create multiple versions of it, each with different percentage of non-null values. On each version, we do the necessary compression and measure the time taken to do 200M access to random locations in the column.

Figures 6.3a and 6.3b show memory usage and performance on 200M random read queries respectively, for uncompressed, prefix sum-based null compressed and vanilla null compressed column of `creationDate` property of `comment` vertex label. We omit the performance number of vanilla null compression as they were significantly higher than the

	1-hop	2-hop	3-hop
V-COL-UNC	12.78	14.47	14.95
V-COL-C	14.65 **1.15x**	16.00 **1.11x**	16.98 **1.14x**

Table 6.4: Runtime (in sec) of k-hop queries on compressed and uncompressed vertex column on LDBC100.

other two configurations (>20x). Our prefix sum-based compression technique requires slightly more memory than vanilla null compression technique (2-bit for each element vs. 1-bit overhead in vanilla compression). However, introducing prefix sums and map lookups in stead of iterations over bit-strings of each block provide significant performance benefits. We observe at most 1.38x slow down the performance compared to the uncompressed column, which has no decompression cost. Surprisingly, accesses in prefix-sum based compression can even be faster than accesses to an uncompressed column when the column is very sparse (< 30% non-null values). This is because testing for null at a location is a constant time operation and when most of the accesses return null, in a null compressed column, the iterators return a single global variable that keeps the null value for the data type. Instead, in an uncompressed column, the null value from the column is copied, which has a higher chance of a CPU cache miss.

Note that Tables 6.3a and 6.3b already compares reading edges from compressed and uncompressed adjacency lists and vertex columns. Reading edges from CSR adjacency lists and vertex columns that are null compressed by our technique are on average 1.14x and 1.3x slower than their uncompressed variant respectively. However, for 50% null values, they gain 50% and 25% in storage respectively.

In our second experiment, we evaluate accessing vertex properties from compressed and uncompressed vertex columns using k-hop queries. We use the `V-COL-C` and `V-COL-UNC` configurations and keep the edge storage uncompressed. We run the workload from Section 6.4 consisting of k-hop queries over `replyOf` edges. These queries are extended to include a condition that a `comment` b is made on another `comment` a such that b has a `creationDate` that is within δ timespan of a's. Table 6.4 shows our results. Using compressed storage slows down our queries by at most 1.14x. Therefore compared to the vanilla null compression schemes, which are not practical, our prefix sum-based null compression scheme is a practical compression scheme that offers a good performance and memory trade-off.

	1-hop	2-hop	3-hop
VOLCANO	3.12	259.34	734.72
LIST-BASED	2.35 **1.33x**	95.24 **2.72x**	313.67 **2.34x**

Table 6.5: Without reading vertex property

Table 6.6: Runtime (in sec) for two set of k-hop queries using volcano-styled query processing and our new list-based query processing.

6.6 List-based Processing vs. Volcano-styled Query Execution

We next provide a preliminary set of experiments that compare the performance of our list-based processing to Volcano-style processing. We use the GraphflowDB with our new columnar storage under two configurations; (i) `VOLCANO`, that uses the conventional Volcano-styled processor, which we achieve by ensuring plans do not use our vector-oriented operators (e.g., `LIST-JOIN`); and (ii) `LIST-BASED` that uses our new list-based operators. Our workload consist of simple 1-, 2-, and 3-hop queries that read the `replyOf` edges of the `person` vertex in LDBC100 and returns the count of final matches. We evaluate the queries on similar plans that perform the same sequence of joins.

Table 6.5 shows the query performance of the 2 query processors. As we expect, list-based processor is more performant than Volcano-style processor. Specifically, we obtain between 1.33x and 2.72x runtime speed up. Our evaluation of list-based processing is currently very preliminary and at the time of this writing we are in the process of implementing and optimizing our list-based operators. We leave a more extensive evaluation of the benefits and limitations of list-based processing to future work.

Chapter 7

Related Work

This thesis studied the integration of columnar storage and query processing techniques to GDBMSs. We review related work on column-oriented relational systems and other storage and compression techniques that have been designed for GDBMSs and RDF engines. There is an extensive literature on different storage and compression techniques designed for specific graphs, which we do not cover here. We refer the interested readers to reference [50, 14] for a survey on the lossless compression techniques from literature. These techniques focus on compressing the topology of the graph and broadly can achieve high compression rates for special types of graphs, e.g., Erdős Rényi graphs or web graphs, but require decompression while accessing adjacency lists. Therefore, they are less applicable for an in-memory GDBMS than the techniques we considered in this thesis.

7.1 Storage and Compression Techniques in Column-Oriented Relational Systems

Column-oriented RDBMSs are designed primarily for OLAP applications that are analytics heavy that perform aggregations over entire tables or subsets of tuples. At a high-level, these systems store the relations in disk pages as a set of separate columns, instead of a set of rows. The column c value of a particular row r can be found using a positional offset, using the row ID of r, in the page that stores column c. The row IDs are referred to as surrogate keys. Prior to the emergence of column stores, such as C-Store [51], MonetDB [32], and VectorWise [54, 55], prior work, such as PAX [9], had used columnar storage inside in the

context of row-oriented systems. PAX was a storage design that stores a set of rows in each disk page, but organizes the rows inside each page in a columnar format.

Many prior work on column stores introduced a set of columnar storage, compression, and query processing techniques. These techniques include: positional offsets (also called virtual IDs) to directly access values in columns for different rows, columnar compression schemes, vector-oriented query processing, late materialization, and direct operation on compressed data, among others. A detailed survey of these techniques can be found in reference [4], which reviews the techniques introduced in C-Store, MonetDB, and VectorWise. This thesis studied how to directly integrate these techniques to in-memory GDBMSs or modify and adapt them so that they can be integrated into in-memory GDBMSs. The most relevant techniques that we identified and integrated are: (i) a set of columnar storage structures to store different components of property graphs (shown in Table 3.1); (ii) vertex and edge ID schemes that allow direct positional offsets into these columns; (iii) compression schemes that include dictionary encoding, zero suppression, compressed edge and vertex IDs, and a null and empty-list compression scheme; and (iv) list-based processing which is a hybrid between Volcano-style and vector-oriented processing.

For disk-based GDBMSs, other techniques, in particular other compression schemes, such as run-length encoding to compress vertex and edge properties, or integer compression schemes from reference [37] to compress adjacency lists might also be beneficial. For in-memory systems, these techniques can increase the systems scalability but they will also decrease performance as data needs to be decompressed before accessing. Except for null-compression, our other compression schemes do not decrease an in-memory GDBMS's performance as they do not require decompression. Null compression slows down query execution but not significantly, so offers a good performance and space trade-off. Whether or not GDBMS-specific versions of other compression schemes can be developed for in-memory GDBMSs is an interesting research direction.

7.2 Storage and Compression for GDBMSs and RDF Systems

There are several existing native GDBMSs, whose internals have been described in technical papers or documents. These systems adopt a columnar structure only for storing the topology of the graph. This is done either by using a variant of vanilla adjacency list format or CSR. These systems use other row-oriented structures, such as property stores that store a sequence of key-value properties, or a separate key-value store to store vertex and edge

properties. For example, Neo4j [2] represents the topology of the graph in adjacency lists that are partitioned by edge labels and stored in linked-lists, where each edge record points to the next that is not necessarily stored consecutively in disk. Similarly, the property of each vertex and edge are stored in a linked-list in an unstructured manner, where each property record points to the next property record and encodes the key, data type, and value of the property. This can be seen as adopting a row-oriented storage for properties. Similarly, JanusGraph [33] stores its edges in adjacency lists partitioned by edge labels and properties as a consecutive key-value pairs (so in row-oriented format). JanusGraph uses variable-length encoding when storing edges in the adjacency lists. Instead, we use fixed-length encodings in our compression schemes. DGraph [1] uses a key-value store to hold adjacency lists as well as properties. We are unaware of any compression schemes adopted by Neo4j and DGraph. All of the above native GDBMSs adopt Volcano-style processors. In contrast, our design adopts columnar structures for vertex and edge properties and a list-based processor. Our storage techniques and list-based processing allows our system to benefit more from data locality. In addition, we improve on these designs by more compressed edge and vertex ID representation (these systems use 8 bytes for each ID) and null compression.

There are also several GDBMSs that are developed directly on top of an RDBMS or another database system. For example, Oracle Spatial and Graph [42] supports a property graph model using the PGQL language [45] that is built on top of Oracle database. SAP's graph database [48] is developed on top of HANA. These systems can benefit from the columnar techniques provided by the underlying RDBMS but the underlying RDBMS techniques are not optimized for graph storage and queries. For example, SAP's graph engine uses SAP HANA's columnar-storage to store vertex and edge tables but these columns do not have CSR-like structures for indexing edges of each vertex. Similarly, existing RDBMSs do not implement null compression schemes similar to our prefix-sum-based scheme that allow constant-time access to arbitrary column values.

RedisGraph [47] is a system that stores its adjacency lists and properties inside the Redis [46], a distributed in-memory key-value database, although RedisGraph's latest release we are aware of runs only on single nodes. RedisGraph has a query processor that is based on performing linear algebra operations. The system converts Cypher queries into a sequence of linear-algebra based operators, which are executed using the GraphBLAS [30] linear algebra library. This can be seen as performing column-oriented processing, as entire columns, which are stored in, are processed during query evaluation. We have not benchmarked our system against RedisGraph to evaluate the efficiency of this approach. This is an interesting direction we have left for future work.

ZipG [36] is a distributed compressed storage engine for property graphs that can answer

queries to retrieve adjacencies as well as vertex and edge properties. ZipG is based on a compressed data structure called Succinct [8]. Succinct stores semi-structured data that is encoded as a set of key and list of values. For example, a node v's properties can be stored with the v's ID as the key and a list of values, corresponding to each property. The properties are distinguished through special delimiter characters ZipG maintains. Edge properties and adjacency lists can be encoded in a similar fashion. All of this data is encoded in flat files in a sorted manner by keys. Succinct then compresses these files using suffix arrays and several secondary level indices based on taking samples of key-value properties to access different records. Although the authors report achieving a good compression rate, unlike our structures, access to a particular record is not constant time and requires accessing secondary indexes followed by a binary search. Therefore, this type of compression provides slower access than our structures.

Reference [10] describes a k2-tree-based adjacency lists storage, while properties are represented either as lists or in k2-trees depending on the number of unique values. Our columnar structures differ from that in [10] in two ways: (i) we use positional offsets to access property values; and (ii) we arrange edge properties into pages to get sequential access.

Several RDF systems also use columnar structures to store RDF databases, which consist of a set of (subject, predicate, object) triples. Reference [7] stores data in a set of columns, where each column store is a set of (subject, object) pairs for each unique predicate. This is similar to partitioning the edges based on their labels in property graphs. However, this storage is not as optimized as the standard storage in GDBMSs, e.g., the edges of a particular object are not stored in native CSR or adjacency list format. Hexastore [53] improves on the idea of predicate partitioning by defining a column for each RDF element (subject, predicate or object) and sorting the column in 2 possible ways in B+ trees. This is similar but not as efficient as double indexing of adjacency lists in GDBMSs. RDF-3X [40] is an RDF system that stores a large triple table that is indexed in 6 B+ tree indexes over each column. Similarly, this storage is not as optimized as the native graph storage found in GDBMSs.

Another set of approaches [44, 34, 25] represent RDF data compactly by making use of the underlying structural patterns to formulate a set of rules that can encode multiple entities together. While, [11, 12] aims at optimizing the topology of RDF data by representing it in data-structures similar to k2-trees [19], that represents adjacency matrices as trees that are essentially NULL pruned. Our work too emphasizes on using structure in graph data, but at the same time aims at improving query performance with a more succinct representation of data in memory.

Finally, several prior work has introduced novel storage techniques for storing graphs in GDBMSs or analytics systems. These works primarily focus on storing the topology of the graph and adopt variants of CSR format [22]. LLAMA is a data structure based on CSR for storing adjacency lists for a write-heavy system. The data structure is designed to provide multi-version support when applications require accessing different versions of adjacency lists. Our focus in this thesis are read-heavy queries and our data structures are not optimized for write-heavy workloads. Similarly, STINGER [21] is a system that describes a data structure to store the topology of a graph also under write-heavy workloads. At a high-level, the data structure adopts a vanilla adjacency list format that is divided by different edge labels, where each list is accessible using vertex IDs. Lists are stored in a linked list of blocks that allow very fast updates in a shared memory system. DISTINGER [24] adopts STINGER's structure to a distributed setting. These techniques are complementary to our work and our edgeID and vertex ID schemes and columnar structures for storing the graph topology can be integrated into these structures to benefit from their functionalities, such as multi-version support.

Chapter 8

Conclusion and Future Work

Column-oriented RDBMSs are read-optimized analytical systems that have introduced several storage and query processing techniques to improve the scalability and performances of RDBMSs. We studied the integration of these techniques into GDBMSs, which are also read-optimized analytical systems. Although some of these techniques can directly be applied to GDBMSs, there are significant differences between the data and query workloads that column-oriented RDBMSs and GDBMSs support, which require adapting some of these techniques. We first outlined a set of guidelines for designing the physical storage layer and query processor of GDBMSs, from which we derived a set of desiderata. At the core of these guidelines is the observation that graph data is not completely unstructured and there is a pattern to how this data is accessed during query processing. In fact, there exists different types of structures in a graph, that constitutes a soft schema, that every graph data adheres to. Specifically edges are read in the order they appear adjacency lists, access to neighbour properties or node properties cannot be localized, and any compression technique that is used should require constant-time operations to access and decompress arbitrary locations in columns. These guidelines and the different types of structures we observed in graph-structured data instructed our design of columnar structures, null compression scheme, and list-based processor. We demonstrated that our storage techniques has increased the scalability of GraphflowDB by 3.55x and often with non-trivial performance benefits (even for null compression when the columns are sparse enough) and our list-based processor has increased the performance by up to 1.6x in our micro-benchmarks.

We outline three immediate directions of future work:

- **Optimizing unstructured graph data:** Noticeably left out from our work are techniques to optimize the storage of unstructured data that appears in some graph-

structured datasets in real-world. A common approach to storing such data is to keep them in variable-width records and structures like linked-lists, which can be seen as a type of row-oriented storage. However, such a storage requires decoding or is distributed randomly in memory, respectively. An important line of future work is to research efficient data structures that makes accessing and compressing unstructured data more efficient. Compression schemes often exploit structure, so the lack of structure for this data makes this line of work very challenging.

- **Factorized processing:** Another interesting extension is to study adopting factorized processing in the context of GDBMS. Our current list-based processing is a simple and limited form of factorized processing which is limited in capability, i.e, it can only generate plans that has `LIST JOIN` only if no further `JOIN` operator depends on it. A complete factorized query processor could perform `LIST JOIN`s at all levels and also produce more succinct outputs of the query.

- **Comparison against other systems:** An interesting evaluation that we left out is to compare GraphflowDB, with our columnar data structures and techniques, to existing GDBMSs on a common workload from a popular benchmark, such as LDBC SNB. This will test the overall efficiency in storage and performance of our changes on more practical queries. Comparisons against column-oriented RDBMSs, such as MonetDB [32], DuckDB [20], would also provide another point of comparison for our list-based processing technique against the vector-based processing that these systems use.

References

[1] Dgraph, 2020.

[2] neo4j, 2020.

[3] Daniel Abadi. Column stores for wide and sparse data. pages 292–297, 01 2007.

[4] Daniel Abadi, Peter Boncz, and Stavros Harizopoulos. *The Design and Implementation of Modern Column-Oriented Database Systems*. Now Publishers Inc., Hanover, MA, USA, 2013.

[5] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 671–682, New York, NY, USA, 2006. Association for Computing Machinery.

[6] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, page 967–980, New York, NY, USA, 2008. Association for Computing Machinery.

[7] Daniel J Abadi, Adam Marcus, Samuel R Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422, 2007.

[8] Rachit Agarwal, Anurag Khandelwal, and Ion Stoica. Succinct: Enabling queries on compressed data. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 337–350, Oakland, CA, May 2015. USENIX Association.

[9] Anastassia Ailamaki, David J DeWitt, Mark D Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001.

[10] Sandra Álvarez, Nieves R. Brisaboa, Susana Ladra, and Óscar Pedreira. A compact representation of graph databases. In *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, MLG '10, page 18–25, New York, NY, USA, 2010. Association for Computing Machinery.

[11] Sandra Álvarez-García, Nieves R. Brisaboa, Javier D. Fernández, and Miguel A. Martínez-Prieto. Compressed k2-triples for full-in-memory RDF engines. *CoRR*, abs/1105.4004, 2011.

[12] Sandra Álvarez-García, Guillermo de Bernardo, Nieves R. Brisaboa, and Gonzalo Navarro. A succinct data structure for self-indexing ternary relations. *J. Discrete Algorithms*, 43:38–53, 2017.

[13] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending rdbmss to support sparse datasets using an interpreted attribute storage format. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 58–58, 2006.

[14] Maciej Besta and Torsten Hoefler. Survey and taxonomy of lossless graph compression and space-efficient graph representations, 2018.

[15] Peter Boncz. Monet; a next-generation dbms kernel for query-intensive applications. 01 2002.

[16] Peter Boncz, M. Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. *2nd Biennial Conference on Innovative Data Systems Research, CIDR 2005*, 01 2005.

[17] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. *Querying Graphs*, volume 10 of *Synthesis Lectures on Data Management*. Morgan & Claypool Publishers, October 2018.

[18] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. Schema validation and evolution for graph databases. *CoRR*, abs/1902.06427, 2019.

[19] Nieves R. Brisaboa, Susana Ladra, and Gonzalo Navarro. k2-trees for compact web graph representation. In *SPIRE*, 2009.

[20] DuckDB. https://duckdb.org/.

[21] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5, 2012.

[22] S. C Eisenstat, Howard Elman, M. H Schultz, and A. H Sherman. The (new) yale sparse matrix package. *Elliptic problem solvers II*, pages 45 – 52, 1983/// 1983.

[23] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. The ldbc social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 619–630, New York, NY, USA, 2015. Association for Computing Machinery.

[24] G. Feng, X. Meng, and K. Ammar. Distinger: A distributed graph data structure for massive dynamic graph processing. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 1814–1822, 2015.

[25] Javier D. Fernández, Miguel A. Martínez-Prieto, and Claudio Gutierrez. Compact representation of large rdf data sets for publishing and exchange. In Peter F. Patel-Schneider, Yue Pan, Pascal Hitzler, Peter Mika, Lei Zhang, Jeff Z. Pan, Ian Horrocks, and Birte Glimm, editors, *The Semantic Web – ISWC 2010*, pages 193–208, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[26] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An evolving query language for property graphs. 2018.

[27] J. Goldstein, R. Ramakrishnan, and U. Shaft. Compressing Relations and Indexes. In *ICDE*, 1998.

[28] G. Graefe. Volcano— an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[29] Goetz Graefe and Leonard Shapiro. Data compression and database performance. 12 2001.

[30] Graphblas. https://http://graphblas.org/, 2020.

[31] Olaf Hartig and Jan Hidders. Defining schemas for property graphs by using the graphql schema definition language. In *Proceedings of the 2nd Joint International*

Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), GRADES-NDA'19, New York, NY, USA, 2019. Association for Computing Machinery.

[32] Stratos Idreos, F. Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, and Martin Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35, 01 2012.

[33] Janus Graph. https://janusgraph.org.

[34] Amit Krishna Joshi and Pascal Hitzler. 1-1-2013 logical linked data compression. 2017.

[35] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An Active Graph Database. In *ACM SIGMOD*, 2017.

[36] Anurag Khandelwal, Zongheng Yang, Evan Ye, Rachit Agarwal, and Ion Stoica. Zipg: A memory-efficient graph store for interactive queries. *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[37] Daniel Lemire and Leonid Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45, 01 2015.

[38] Amine Mhedhbi and Semih Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB*, 12(11), 2019.

[39] Neo4j Property Graph Model. https://neo4j.com/developer/graph-database, 2020.

[40] Thomas Neumann and Gerhard Weikum. The rdf3x engine for scalable management of rdf data. *The Vldb Journal - VLDB*, 19:91–113, 02 2010.

[41] Dan Olteanu and Maximilian Schleich. Factorized databases. *SIGMOD Rec.*, 45(2):5–16, September 2016.

[42] Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html, 2020.

[43] Oracle Spatial and Graph. cs.oracle.com/en/database/oracle/oracle-database/12.2/inmem/in-memory-column-store-architecture.html, 2020.

[44] Jeff Z. Pan, José Manuel Gómez Pérez, Yuan Ren, Honghan Wu, Haofen Wang, and Man Zhu. Graph pattern based rdf data compression. In Thepchai Supnithi, Takahira Yamaguchi, Jeff Z. Pan, Vilas Wuwongse, and Marut Buranarach, editors, *Semantic Technology*, pages 239–256, Cham, 2015. Springer International Publishing.

[45] Property graph query language. https://pgql-lang.org/, 2020.

[46] Redis. https://redis.io/, 2020.

[47] Redisgraph. https://oss.redislabs.com/redisgraph/, 2020.

[48] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. The graph story of the sap hana database. pages 403–420, 01 2013.

[49] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing. *Proc. VLDB Endow.*, 11(4):420–431, December 2017.

[50] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *The VLDB Journal*, 2019.

[51] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth O'Neil, Pat O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. C-store: A column-oriented dbms. In *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, page 553–564. VLDB Endowment, 2005.

[52] TigerGraph. https://www.tigergraph.com.

[53] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.

[54] Marcin Zukowski and Peter Boncz. From x100 to vectorwise: Opportunities, challenges and things most researchers do not think about. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, page 861–862, New York, NY, USA, 2012. Association for Computing Machinery.

[55] Marcin Zukowski and Peter A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35:21–27, 2012.

[56] Marcin Zukowski, Sandor Heman, Niels Nes, and Peter Boncz. Super-scalar ram-cpu cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, page 59, USA, 2006. IEEE Computer Society.