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

Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu
University of Waterloo
{pranjal.gupta, amine.mhedhbi, semih.salihoglu}@uwaterloo.ca

ABSTRACT

We revisit column-oriented storage and query processing techniques in the context of contemporary graph database management systems (GDBMSs). Similar to column-oriented RDBMSs, GDBMSs support read-heavy analytical workloads that however have fundamentally different data access patterns than traditional analytical workloads. We first derive a set of desiderata for optimizing storage and query processors of GDBMS based on the access patterns in GDBMSs. We then present the design of columnar storage, compression, and query processing techniques based on these requirements. In addition to showing direct integration of existing techniques from columnar RDBMSs, we also propose novel ones that are tailored for GDBMSs. These include a novel list-based query processor, which avoids expensive data copies of traditional block-based processors under many-to-many joins and avoids materializing adjacency lists in intermediate tuples, a new data structure we call single-indexed edge property pages and an accompanying edge ID scheme, and a new application of Jacobson's bit vector index for compressing NULL and empty lists. We integrated our techniques into the GraphflowDB in-memory GDBMS. Through extensive experiments, we demonstrate the scalability and performance benefits of our columnar storage and the query performance benefits of our list-based processor.

1. INTRODUCTION

Contemporary GDBMSs are data management software such as Neo4j [8], Neptune [1], TigerGraph [13], and Graphflow [43, 48] that adopt the property graph data model [10]. 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 key-value properties on the vertices and edges.

GDBMSs have lately gained popularity to support a wide range of analytical applications, from fraud detection and risk assessment in financial services to recommendations in e-commerce and social networks [54]. These applications have workloads that search for patterns in a graph-structured database, which often requires reading large amounts of data. In the context of RDBMSs, column-oriented systems [11, 40, 55, 60] employ a set of read-optimized storage, indexing, and query processing techniques to support traditional 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 paper, we revisit columnar storage and query processing techniques in the context of GDBMSs. Specifically, we focus on an in-memory GDBMS setting and discuss the applicability of columnar storage techniques [55], compression schemes [16, 18, 63], and vector-based query processing [19, 26] for storing and accessing different components of the system. 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. The contributions of this paper are as follows.

Guidelines and Desiderata: We begin in Section 3 by analyzing the properties of data access patterns in GDBMSs. For example, we observe that different components of data stored in GDBMSs can have some structure and the order in which operators access vertex and edge properties often follow the order of edges in adjacency lists. This analysis instructs a set of guidelines and desiderata for designing the physical data layout and query processor of a GDBMS.

Columnar Storage: Section 4 explores the application of columnar data structures for storing different components of GDBMSs. While existing columnar structures can directly be used for storing vertex properties and many-to-many (n-n) edges, we observe that using a straightforward columnar structure, which we call *edge columns*, to store properties of n-n edges is suboptimal as it does not guarantee sequential access when reading edge properties in either forward or backward directions. An alternative, which we call *double-indexed property CSRs*, can achieve sequential access in both directions but requires duplicating edge properties, which can be undesirable as graph-structured data often contain orders of magnitude more edges than vertices. We then describe an alternative design point, *single-directional property pages*, that avoids duplication and achieves good locality when reading properties of edges in one direction and still guarantees random access in the other when using a new edge ID scheme that we describe. Our new ID schemes allow for extensive compression when storing them in adjacency lists without decompression overheads. Lastly, as a new application of vertex columns, we show that single cardinality edges and edge properties, i.e. those with one-to-one (1-1), one-to-many (1-n) or many-to-one (n-1) cardinalities, are stored more efficiently with vertex columns instead of the structures we described above for properties of n-n edges.

Columnar Compression: In Section 5, we review existing

columnar compression techniques, such as dictionary encoding, that satisfy our desiderata and can be directly applied to GDBMSs when storing different components. We next show that existing techniques for compressing NULL values in columns from references [16, 18] by Abadi et al. would lead to very slow accesses to non-NULL values. We then review the bit vector indexing structure by Jacobson [41, 42] to support constant time rank queries. This structure is used in many other data structures, e.g., recently in a range filter structure in databases [58] or RRR sequences that have many applications in information retrieval [36, 49] and computational geometry [29, 50]. We show that an adaptation of Jacobson's bit vector index is more suitable than existing schemes from references [16, 18] to compress both NULL edge and vertex properties, as well as empty adjacency lists. This structure allows constant-time access to NULL or non-NULL values with a small increase in storage overhead per entry compared to the techniques.

List-based Processing: In Section 6, we observe that traditional Volcano-style [37] tuple-at-a-time processors, which are used in some GDBMSs, do not benefit from processing blocks of data in tight loops but also has the advantage of avoiding expensive data copies when performing many-to-many joins, e.g. in long path queries. On the other hand, columnar RDBMSs have block-based [19, 61] query processors that process fixed-length blocks of data in tight loops, which achieves better CPU and cache utility. However, block-based processing results in expensive data copies under many-to-many joins. To address this, we propose a new block-based processor we call *list-based processor (LBP)*, which modifies traditional block-based processors in two ways to tailor them for GDBMSs: (i) Instead of representing the intermediate tuples processed by operators as a single group of equal-sized blocks, we represent them as multiple groups of blocks. We call these *list groups*. LBP avoids expensive data copies by flattening blocks of some groups into single values when performing many-to-many joins. Flattening does not require any data copies and happens by setting an index field of a list group. (ii) Instead of fixed-length blocks, LBP uses variable length blocks that take the lengths of adjacency lists that are represented in the intermediate tuples. Because adjacency lists are already stored in memory consecutively, this allows use to avoid materializing adjacency lists during join processing, improving query performance.

We integrated our techniques into GraphflowDB [43]. We present extensive experiments that demonstrate the scalability and performance benefits (and tradeoffs) of our techniques both on micro-benchmarks and on the LDBC social networking benchmark against a row-based Volcano-style implementation of the system, the open-source version of a commercial GDBMSs, and a column-oriented RDBMS. Our code, queries, and data are available here [15].

2. BACKGROUND

We review the general storage components and primary query plan operators of GDBMSs. In the property graph model, vertices and edges have labels and arbitrary key value properties. Figure 1 shows a property graph that we will use as our running example. The graph contains vertices with `PERSON` and `ORGANIZATION (ORG)` labels, and edges with `FOLLOWS`, `STUDYAT` and `WORKAT` labels.

There are three primary storage components of GDBMSs: (i) topology, i.e., adjacencies of vertices; (ii) vertex properties; and (iii) edge properties. In every native GDBMS we

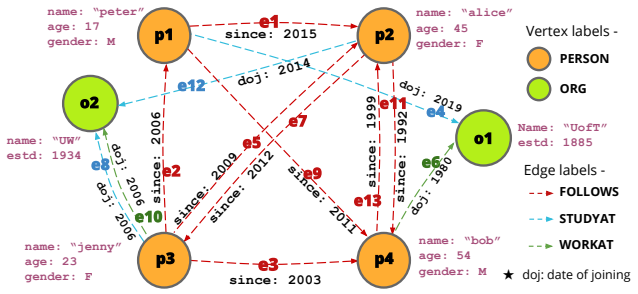

Figure 1: Running example graph.

are aware of, the topology of the graph is stored in data structures that organize data in *adjacency lists* [27], such as a column-sparse row (CSR). Typically, given the ID of a vertex v, the system can access v's adjacency list, which contains a list of (edge ID, neighbour ID) pairs, in constant time. Typically, an adjacency list of v is further clustered/-partitioned by the edge label which enables traversing the neighbourhood of v based on a particular label efficiently, which optimizes systems for queries that contain edge labels. Vertex and edge properties can be stored in a number of ways. For example, some systems use a separate key-value store, such as DGraph [2] and JanusGraph [5], and some use a variant of *interpreted attribute layout* [24], where records consist of serialized variable-sized key-value properties. These records can be located consecutively in disk or memory or have pointers to each other, as in Neo4j.

Queries GDBMSs consist of a subgraph pattern Q that describes the joins in the query (similar to SQL's FROM) and optionally predicates on these patterns with final group-by-and-aggregation operations. We assume a GDBMS with a query processor that uses variants of relational operators, e.g., Neo4j [8], Memgraph [6], Neptune [1], or GraphflowDB, with the following operators:

`SCAN`: Scans a set of vertices from the graph.

`JOIN` (e.g. `EXPAND` in Neo4j and Memgraph, `EXTEND` in GraphflowDB): Performs an index nested loop join using the adjacency list index to match an edge of Q. Takes as input a partial match t that has matched k of the query edges in Q. For each t, extends t by matching an unmatched query edge $qv_s \rightarrow qv_d$, where one of qv_s or qv_d has already been matched. Suppose qv_s has already been matched to data vertex v_i. The operator reads the forward adjacency list $\{(e_{i1}, v_{i1}), ..., (e_{i\ell}, v_{i\ell})\}$ of v_i to produce a $(k+1)$-match that matches one edge-neighbor pair (e_{ij}, v_{ij}) of v_i.[1].

`FILTER`: Applies a predicate ρ to a partial match t, reading any necessary vertex and edge properties from storage.

`GROUP BY AND AGGREGATE`: Performs a standard group by and aggregation computatio on a partial match t.

EXAMPLE 1. *Below is an example query written in the Cypher language [34]:*

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

The query returns all persons a and their workplaces b, where a is older than 22 and b was established before 2014. Figure 2 shows a typical plan for this query:

3. GUIDELINES AND DESIDERATA

[1]GraphflowDB can perform an intersection of multiple adjacency lists if the pattern is cyclic (see reference [48]).

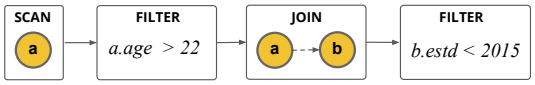

Figure 2: Query plan for the Example 1.

We next outline a set of guidelines and desiderata for organizing the physical data layout and query processor of GDBMSs. We note that we assume edges are doubly-indexed in both a forward and a backward adjacency list, which holds in all GDBMSs we are aware of. We will not optimize to avoid this duplication because this is fundamental for performing fast joins on source or destination vertices of edges.

GUIDELINE 1. *Edge and vertex properties are read in the same order as edges in an adjacency list after joins.*

Observe that `JOIN` accesses the edges and neighbors of a vertex v_i in the order these edges appear in v_i's adjacency list $L_{v_i} = \{(e_{i1}, v_{i1})..., (e_{i\ell}, v_{i\ell})\}$, where (e_{ij}, v_{ij}) are IDs of edges and neighbors of v_i. If the next operator also needs to access the properties of these edges or vertices, e.g., `FILTER` in Figure 2, these accesses will be in the same order. Ideally, a system should store the properties of e_{ij} and v_{ij} sequentially in the same order to increase CPU cache locality. Note that each $e_{ij}(v_s, v_d)$ is located in only two adjacency lists: in v_s's forward and v_d's backward lists. However, if a vertex v_{ij} has n edges, then v_{ij}'s ID is replicated in n lists. Therefore, localizing v_{ij}'s property for n different possible accesses would require prohibitive data replication. So our first desiradata is to localize the storage of edge properties.

DESIDERATUM 1. *Store and access the properties of edges sequentially in the order edges appear in adjacency lists.*

GUIDELINE 2. *Access to vertex properties will not be to sequential locations and many adjacency lists are very small.*

Guideline 1 implies that we should expect random accesses in memory when an operator, say `FILTER` in Figure 2 accesses vertex properties. Another property of real-world graph-structured data with many-to-many relationships is that they depict power-law distributions, which implies that there might be many very short adjacency lists in the dataset. Therefore when processing queries with two or more joins, reading different adjacency lists will require iteratively reading a short list followed by a random access. This has an important implication for adopting compression techniques in in-memory GDBMSs. Specifically, 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, decompressing arbitrary data elements in a compressed block should be possible in constant time.*

GUIDELINE 3. *Graph data often has partial structure.*

Although the property graph data model is semi-structured, data stored in GDBMSs often have some structure, which GDBMSs can exploit. In fact, several vendors from industry and academics are actively working on defining a schema language for the property graph data model [28, 39]. We identify three types of structure in property graph data:

(i) *Edge label determines source and destination vertex labels.* Often, edges with a particular labels are between a fixed source and destination vertex label. For example, in the popular LDBC social network benchmark (SNB), `KNOWS` edges exist only between vertices of label `PERSON`.

Data	Columnar data structure
Vertex Properties	V-Column
Edge Properties	V-Column: of src when n-1, of dst when 1-n, of either src or dst when 1-1
	Single-indexed prop. pages when n-n
Fwd Adj. lists	V-Column when 1-1 and n-1, CSR o.w.
Bwd Adj. lists	V-Column when 1-1 and 1-n, CSR o.w.

Table 1: Columnar data structures and data components they are used for. V-Column stands for vertex column.

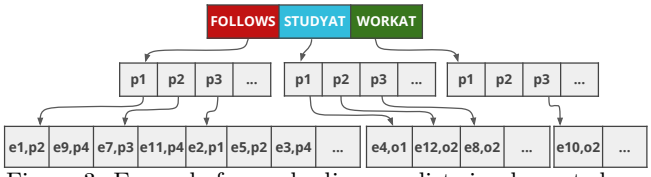

Figure 3: Example forward adjacency lists implemented as a 2-level CSR structure for the example graph.

(ii) *Label determines properties on vertices and edges.* Similar to the attributes of a relational table, properties on an edge or vertex and their datatypes can sometimes be determined by the label. For example, this is the case for every vertex and edge label in LDBC.

Throughout this paper we will call edges that satisfy the above two properties as *structured edges*. Similarly, we will consider vertex and edge properties that satisfy property (ii) as *structured vertex/edge property*. Other edges and properties will be called *unstructured*. A third structure that may exist for edges is the following.

(iii) *Edges with single cardinality.* Edges might have cardinality constraints. We will differentiate between the following cardinality constraints: one-to-many (single cardinality in the backward edges), many-to-one (single cardinality in the forward edges), one-to-one, and many-to-many. An example of one-to-many cardinality from LDBC SNB is that each company has one `isLocatedIn` edge.

Such structure can be exploited to allow faster access to data or compression. This motivates our third desideratum:

DESIDERATUM 3. *Exploit structure in the data for space-efficient storage and faster access to data.*

4. COLUMNAR STORAGE

We next explore using columnar structures for storing data in GDBMSs to meet the desiderata from Section 3. For reference, Table 1 presents the summary of the columnar structures we use and the data they store. Some techniques we review here are not novel and even widely used in systems, such as CSR for storing edges. We review them to present a complete columnar storage design for a GDBMS. Others are either new designs, such as single-indexed property pages and its accompanying edge ID scheme, or new applications of existing designs, such as using vertex columns to store single cardinality edges.

4.1 CSR for n-n Edges

CSR is an existing columnar structure that is directly applicable and in fact widely adopted by existing GDBMSs to store edges. A CSR, shown in Figure 3, is a columnar structure that effectively stores a set of (vertex ID, edge ID, neighbour ID) triples sorted by vertex ID, where the vertex IDs are compressed similar to run-length encoding.

The main difference is that instead of storing vertex ID-run length pairs, CSR stores a list of prefix-sums of the run lengths in the vertex ID column. These prefix sums are used to directly access the adjacency list of a vertex. In this work, we store the edges of each edge label with n-n cardinality in a separate CSR, i.e., similar to some other systems, such as Neo4j, we partition the edges first by their label. As we discuss in Section 4.3, we can store the edges with other cardinalities more efficiently than a CSR using vertex columns.

4.2 Single-indexed Edge Property Pages for Properties of n-n Edges

Recall Desideratum 1 that access to edge properties should be in the same order of the edges in adjacency lists. We first review two columnar structures, *edge columns* and *double-indexed property CSRs*, the former of which has low storage cost but does not satisfy Desideratum 1 and the latter has high storage cost but satisfies Desideratum 1. We then describe a new design, which we call *single-indexed property pages*, which has low storage cost as edge columns and with a new edge ID scheme can partially satisfy Desideratum 1, so dominates edge columns in this design space.

Edge Columns: We can use a separate *edge column* for each property $q_{i,j}$ of edge label le_i. Then with an appropriate edge ID scheme, such as (edge label, label-level positional offset), one can perform a random access to read the $q_{i,j}$ property of an edge e. This design has low storage cost and stores each property once but does not satisfy Desideratum 1, as it does not necessarily store the properties of the edges according to their appearance in the adjacency lists. In practice, the order would be determined by the sequence of edge insertions and deletions.

Double-Indexed Property CSRs. An alternative is to mimic the storage of adjacency lists in the CSRs in separate CSRs that store edge properties. For each vertex v we can store $q_{i,j}$ twice in *forward* and *backward property lists*. This design provides sequential read of properties in both directions, thereby satisfying Desideratum 1, but also requires double the storage of edge columns (plus the CSR offset overheads). This can often be prohibitive as many graphs have orders of magnitude edges than vertices and avoiding duplicate storage of large properties, such as strings, is desirable especially for in-memory systems, which we focus on.

A natural question is: *Can we avoid the duplicate storage of double-indexed property CSRs but at the same time achieve sequential reads?* We next show a structure that with an appropriate edge ID scheme obtains sequential reads in one direction, so partially satisfying Desideratum 1. This structure therefore dominates edge columns as a design.

Single-indexed property pages: A first natural design uses only one property CSR, say forward. We call this structure *single-indexed property CSR*. Then, properties can be read sequentially in the forward direction. However, reading a property in the other direction quickly, specifically with constant time access, requires a new edge ID scheme.

To see this suppose a system has read the backward adjacency lists of a vertex v with label le_i, $\{(e_1, nbr_1), ..., (e_k, nbr_k)\}$, and needs to read the $q_{i,j}$ property of these edges. Note that (e_i, nbr_i) are edge ID, neighbor ID pairs. Then given e_1, we need to be able to read $q_{i,j}$ from the forward property list P_{nbr_1} of nbr_1. With a standard edge ID scheme, for example one that assigns consecutive IDs to all edges with label le_i, the system would need to first find the position p of e_1 in L_{nbr_1}, which may require reading all of the edges in L_{nbr_1} and then use p to read the property. This access is not constant time.

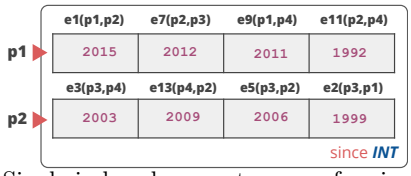

Figure 4: Single-indexed property pages for since property of FOLLOWS edges in the example graph. $k = 2$.

To overcome this, we can adopt a new edge ID scheme that stores the following: (edge label, source vertex ID, list-level positional offset)[2]. With this scheme a system can: (i) identify each edge, e.g., perform equality checks between two edges; and (ii) read the offset p directly from edge IDs, so reading edge properties in the opposite direction (backward in our example) can now be constant time. In addition, the first two components of this ID scheme can often be compressed and as a result it can more space-efficient than schemes that assign consecutive IDs to all edges (see Section 5.2). However, single-indexed property CSR and this edge ID scheme has two limitations. First access to properties in the 'opposite direction' requires two random accesses, e.g., first access obtains the P_{nbr_1} list using nbr_1's ID and the second access reads $q_{i,j}$ from P_{nbr_1}. Second, although we do not focus on updates in this paper, using edge IDs that contain positional offsets has an important consequence for GDBMSs. Observe that positional offsets that are used by GDBMSs are explicitly stored in data structures. For example, in every native GDBMS we are aware of vertex IDs are used as positional offsets to read vertex properties and they are also explicitly stored in adjacency lists. This is unlike traditional columnar RDBMSs, where positional offsets, specifically row IDs (RIDs) of tuples, are implicit and not explicitly stored. Therefore, when deletions happen, GDBMSs initially leave gaps in their data structures, and keep track of the deleted IDs to recycle them when new insertions happen. For example, Neo4j's nodestore.db.id file keeps track of deleted IDs for later recycling [9]. Similarly, the list-level positional offsets in edge IDs we described above are explicitly stored in adjacency lists and need to be recycled. This may leave many gaps in adjacency lists because to recycle a list-level offset, the system needs to wait for another insertion into the same adjacency list, which may be infrequent.

Our *single-indexed property pages* addresses these two issues (Figure 4). We store k property lists (by default 128) in a property page. Within a property page, properties of the same list are not necessarily stored consecutively. However, because we use a small value of k these properties are stored in close-by memory locations. We modify the edge ID scheme above to use page-level positional offsets. This has two advantages. First, given a positional offset, the system can directly read an edge property (so we avoid the access to read P_{nbr_1}). Second, the system can recycle a page-level offset whenever any one of the k lists get a new insertion.

Figure 4 shows the single-indexed property pages in the forward direction for `since` property of edges with label `FOLLOWS` when k=2. In Section 7.3, we show that the read performance of single-indexed property pages on a microbench-

[2]If we use the backward property CSR, the second component would instead be the destination vertex ID.

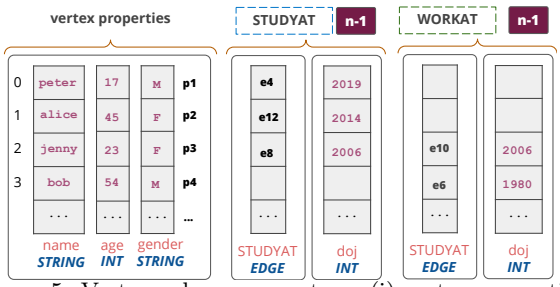

Figure 5: Vertex columns can store: (i) vertex properties; and (ii) single-cardinality edges and their properties.

mark is 2.5x faster than edge columns in one direction and similar in the other.

4.3 Vertex Columns for Vertex Properties, Single Cardinality Edges and Edge Properties

With an appropriate vertex ID scheme, columns can be directly used for storing structured vertex properties in a compact manner. Let $p_{i,1}, p_{i,2}, ...p_{i,n}$ be the structured vertex properties of vertices with label lv_i. We can store each $p_{i,j}$ in a *vertex column*, that store $p_{i,j}$ properties of vertices in consecutive locations. Then we can adopt a (vertex label, label-level positional offset) ID scheme that ensures that offsets of vertices with the same label are consecutive. Similar to the new edge ID scheme we described in Section 4.2, this ID scheme also can be compressed and requires less space than storing global consecutive IDs, which is the typical ID scheme in GDBMSs (see Section 5.2).

We next observe that we can exploit single cardinality constraints, i.e., 1-1, 1-n, or n-1 constraints, on the edges to store them and their properties more efficiently than CSR or property pages (Desideratum 3). Specifically, instead of a CSR or a property page, we can use vertex columns also for storing single cardinality edges and their properties. Specifically, we can store these edges and their properties directly as a *property* of source or destination vertex of the edges and directly access them using a vertex positional offset. Figure 5 shows single cardinality STUDYAT and WORKAT edges from our example and their properties stored as vertex column of `PERSON` vertices. Note that vertex columns are more space-efficient and provides faster access to edges than a 2-level CSR or property pages (e.g., they do not require storing the CSR offsets). As we discuss in the next section, by storing the properties of single cardinality edges in vertex columns, we can also compress their edge IDs further because we no longer need page-level positional offsets.

5. COLUMNAR COMPRESSION

Data compression and query processing directly on compressed data are extensively used in columnar RDBMSs. We next revisit the integration of some of these techniques into in-memory GDBMSs. To present our complete integration, we start by reviewing techniques that are directly applicable and not novel. We then discuss the cases when we can compress the new vertex and edge ID schemes from Section 4. Finally, we review existing null compression schemes in columnar RDBMSs from references [16, 18] and show their shortcomings for compressing null properties and empty lists in GDBMSs. We then review Jacobson's bit vector indexing technique [41] which addresses these shortcomings and describe an adaptation of it that is suitable for GDBMSs.

5.1 Directly Applicable Techniques

Recall our Desideratum 2 that because access to vertex properties cannot be localized and because many adjacency lists are very short, the compression schemes that are suitable for in-memory GDBMSs need to either avoid decompression completely or support decompressing arbitrary elements in a block in constant time. This is only possible if the elements are encoded in *fixed-length codes* instead of variable-length codes. We review dictionary encoding and leading 0 suppression, which we integrated in our implementation and refer readers to references [18, 35, 45] for details of other fixed-length schemes, such as frame of reference.

Dictionary encoding: The dictionary encoding is perhaps the most common encoding scheme to be used in RDBMSs [18, 63]. At a high-level, this scheme maps a domain of values into compact representation codes using a variety of schemes [18, 38, 63]. 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 [18]. These techniques are often used to encode `STRING`s into 64-bit integer values or any categorical property into a small number of fixed-length bits. We use dictionary encoding to map a categorical edge or vertex property p, e.g., gender property of `PERSON` vertices in LDBC SNB dataset, that takes on z different values to $\lceil log_2(z)/8 \rceil$ bytes (we pad $log_2(z)$ bits with 0s to have a fixed number of bytes).

Leading 0 Suppression [3]**:** Given a block of data, this scheme omits storing leading zero bits in each value of the block [24]. In the fixed-length variant of this scheme the minimum number leading 0s from all values is encoded in fixed number of bits. We adopt a fixed-length variant of this for storing components of edge and vertex IDs, e.g., if the maximum size of a property page of an edge label is t, we use $\lceil log_2(k)/8 \rceil$ many bytes for the page-level positional offset of edge IDs.

5.2 Factoring Out Edge/Vertex ID Components

Our vertex and edge ID schemes from Sections 4 decompose the IDs into multiple small components, which enables us to factor out most of these components, when the data depicts some structure (Desideratum 3). This enables us to compress the adjacency lists without needing to decompress while scanning them. Recall that the ID of an edge e is a triple (edge label, source/destination vertex ID, page-level positional offset) and the ID of a vertex v is a pair (vertex label, label-level positional offset). Recall also that GDBMSs store (edge ID, neighbour ID) pairs inside adjacency lists. First, the source (or the destination) vertex IDs inside the edge ID can be omitted because the the source (or thed destination) vertex ID is the neighbour vertex ID in the backward (or forward) lists, which is already stored in the pair. Second, we do not have to store the edge labels because we cluster our adjacency lists by edge label. The only components of (edge ID, neighbour ID) pairs that need to be stored are: (i) positional offset of the edge ID; and (ii) vertex label and positional offset of neighbour vertex ID. When the data depicts some structure, we can further factor out some of these components as follows:

- *Edges do not have properties:* Often, the edges (having a particular label) do not have any structured or unstructured properties. They are in the data to represent the

[3]In reference [18], Abadi et al. refer to this as NULL suppression, not to be confused with NULL compression to compresses actual NULL values in columns (see Section 5.3).

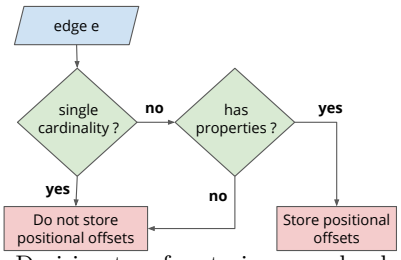

Figure 6: Decision tree for storing page-level positional offsets of edges in adjacency lists.

existence of a relationship and facilitate joins between vertices. For example, in the LDBC SNB dataset, 10 out of 15 edge labels do not have any edge properties. Note that if edges do not have properties, edges do not need to be identifiable, as the system will not access properties of these edges. Therefore, we can distinguish two edges by their neighbour vertex ID and edges with the same IDs are simply replicas of each other. Hence, we can completely omit storing the positional offsets of edge IDs.

- *Edge label determines neighbour vertex label.* Often, edge labels in the graph are between a single source and destination vertex label. In this case, we can omit storing the vertex label of the neighbour ID. For example, this property exists for 10 edge labels in LDBC SNB.
- *Single cardinality edges:* Recall from Section 4.3 that the properties for single cardinality edges can be stored in vertex columns. So we can directly read these properties by using the source or destination vertex ID. Equivalently the page-level positional offset of a single cardinality edge is always 0, so can be omitted. For example, this property exists in 8 of the edge labels in LDBC SNB.

Figures 6 shows our decision tree to decide when to omit storing the page-level positional offsets in edge IDs.

5.3 NULL and Empty List Compression

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

One way to compress NULLs in column-oriented RDBMSs is to treat NULL values as a separate data value and use run-length encoding [16]. Depending on the sparsity level of the column, Abadi in reference [16] describes three other and more optimized NULL compression techniques. All of these techniques list non-NULL elements consecutively in a 'non-NULL values column' and use a secondary structure to indicate the positions of these non-NULL elements. First technique lists positions of each non-NULL value consecutively, which is suitable for very sparse columns, e.g., with $> 90\%$ NULLs. Second, for dense columns, lists non-NULL values as a sequence of pairs, each indicating a range of positions with non-NULL values. Third, for columns with intermediate sparsity, uses a bit vector to indicate if each location is NULL or not. The last technique is quite compact and requires only 1 extra bit per each element in a column.

However, none of these techniques are directly applicable to GDBMSs, including run-length encoding. This is because they do not allow constant-time access to non-NULL values, so do not satisfy our Desideratum 2. Instead, these schemes are designed assuming an operator will read the entire block

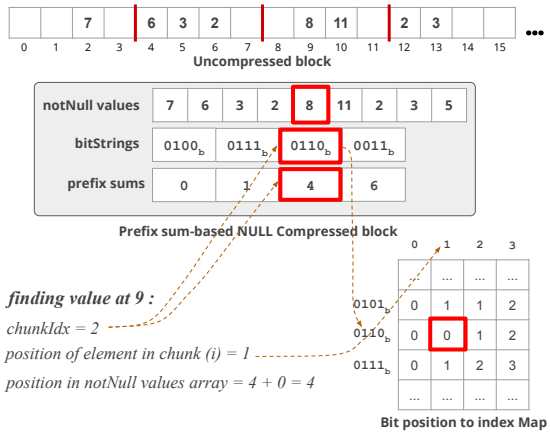

Figure 7: NULL compression using a simplified Jacobson's bit vector rank index with chunk size 4.

sequentially, so the non-NULL values column and secondary data structure in tandem. To support constant-time access to the non-NULL values, which is needed for GDBMSs, the secondary structure needs to support two operations in constant time: (i) check if it is NULL or not; and (ii) if it is non-NULL, we need to compute the *rank* of p, i.e., compute the number of non-NULL values before p.

Abadi's third design that uses a bit vector already supports checking if an arbitrary position p is NULL. To support rank queries, we extend this design with a simplified version of Jacobson's bit vector index [41, 42]. Figure **??** shows this design. In addition to the array of non-NULL values and the bit-string, we store prefix sums for each c (16 by default) elements in a block of a column, i.e., we divide the block into chunks of size c. The prefix sum holds the number of non-NULL elements before the current chunk. We also maintain a pre-populated static 2D *bit-string-position-count map* M with 2^c cells. $M[b, i]$ is the number of 1s before the i'th bit of a c-length bit string b. Let p be the offset which is non-NULL and b the c-length bit string chunk in the bit vector that p belongs to. Then $\text{rank}(p) = ps[p/c] + M[b, p \bmod c]$. This is a simplified version of Jacobson's original index, which divides the bit vector into two sets of prefix sums. This slightly reduces the space for storing prefix sums but also requires twice as many operations to compute ranks.

The choice of c primarily affects how big the pre-populated map is. A second important parameter in this scheme is the number of bits m that we use for each prefix sum value, which determines how large a block we are compressing and how much overhead the scheme has for each element. For an arbitrary m, c, we require: (i) $2^c * \log(c)$ size map, because the map has 2^c cells and needs to store a $\log(c)$-bit count value in each cell; (ii) we can compress a block of size 2^m; and (iii) we store one prefix sum for each c elements, so incur a cost of m/c extra bits per element. By choosing $m = 16, c = 16$ (so we use short integers), we require $2^c * c = 1$MB-size map, can compress $2^m = 64$K blocks, and incur $m/c = 1$ extra bit overhead for each element, so increase the overhead of reference [16]'s scheme from 1 to only 2 bits per element (but provide constant time access to non-NULL values).

6. LIST-BASED PROCESSING

We next describe our list-based query processor (LBP). We first motivate LBP by discussing the shortcomings of traditional Volcano-style tuple-at-a-time query processors that

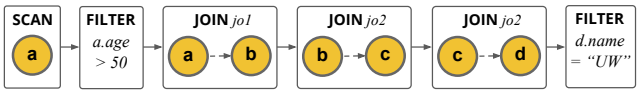

Figure 8: Query plan for Example 2.

some GDBMSs adopt [8, 48] and the traditional block-based processors of columnar RDBMSs when processing many-to-many joins. We give an example.

Example 2. *Consider the following query. P, F, S, and O abbreviate* ***PERSON****,* ***FOLLOWS****,* ***STUDYAT****, and* ***ORGANISATION****.*

```
MATCH (a:P)−[:F]→(b:P)−[:F]→(c:P)−[:S]→(d:O)
WHERE a.age > 50 and d.name = "UW"  RETURN *
```

Consider evaluating a simple plan for this query shown in Figure 8, which is akin to a left-deep plan in an RDBMS, on a graph where `FOLLOWS` are many-to-many edges and `STUDYAT` edges have single cardinality. Volcano-style tuple-at-a-time processing [37], which some GDBMSs adopt [8, 48], is efficient in terms of how much data is copied to the intermediate tuple. For example, even if a scan matches `a` to, say a_1 which extends to say k_1 many `b`'s, say $b_1 \ldots b_{k_1}$, and suppose each b_i extends to k_2 many `c`'s (let us ignore the `d` extension for now), the value a_1 would be copied only once to the tuple instead of $k_1 \times k_2$ times. This is an important advantage for GDBMSs which frequently process many-to-many joins. However, it is well known that Volcano-style processors do not achieve high CPU and cache utility as processing is intermixed with many iterator calls.

Column-oriented RDBMSs instead adopt block-based processors [25, 40], which processes an entire block at a time in operators[4]. Block sizes are fixed length, e.g. 1024 tuples [3, 26]. While processing blocks of tuples, operators read consecutive memory locations, achieving good cache locality, and perform computations inside loops over arrays which is efficient on modern CPUs. However, traditional block-based processors have two shortcomings for GDBMSs. (1) For many-to-many joins, block-based processing requires more data copying into intermediate data structures than tuple-at-a-time processing. Suppose for simplicity a block size of $k_1 \times k_2$. In our example a_1 has k_1 `FOLLOWS` edges to vertices $b_1, ..., b_{k_1}$ and each b_i has k_2 `FOLLOWS` edges to vertices $b_{ik_2} ..., b_{(i+1)k_2}$. Then at a high-level, the scan would output an array $a : [a_1]$, jo_1 would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_{k_1}]$ vectors, and jo_2 would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_1, b_2, ..., b_2, ..., b_{k_1}, ..., b_{k_1}]$, $c : [c_1, ..., c_{k_2}, ..., c_{(k_1-1)k_2}, ..., c_{k_1 k_2}]$, where for example the value a_1 gets copied $k_1 \times k_2$ times into intermediate arrays. Instead of copying data values, some column-oriented systems, such as DuckDB [3], keep a small 'data array' with a single a_1 value, and use additional 'selection arrays' that contain copies of offsets that point to the data values (so instead offset 0 is now copied many times). (2) Traditional block-based processors do not exploit the list-based data organization of GDBMSs. Specifically, the adjacency lists that are used by join operators are already stored consecutively in memory, which can be exploited to avoid materializing these lists into blocks.

[4]Block-based processing has been called vectorized processing in the original work on MonetDB/X100 from CWI [62]. We use the term block-based not to confuse it with SIMD vectorized instructions. For example, a new column store DuckDB [3] also from CWI uses block-based processing but without SIMD instructions. Note however that operators in block-based processors can use SIMD instructions as they process multiple tuples at a time inside loops.

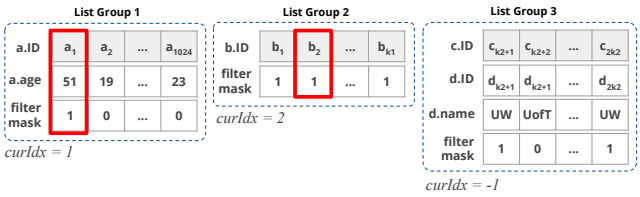

Figure 9: Intermediate chunk for the query in Example 2. The first two list groups are flattened to single tuples, while the last one represents k_2 many tuples.

We developed a new block-based processor, which we call *list-based processor* (LBP) that addresses these shortcomings. LBP has two differences from traditional block-based processors, each of which addresses one of the two shortcomings we identified above. First, traditional block-based processors represent a set of tuples as a single group of blocks/arrays, each corresponding to one variable. In our example we had three variables `a`, `b`, and `c` corresponding to three arrays. The values at position i of all arrays form a single tuple. Therefore to represent the tuples that are produced by many-to-many joins, repetitions of values (or selector offsets) are necessary. One approach to address this can be to compress the arrays that correspond to the variables that will be repeated with run-length encoding. These are the variables in the join that are extended such as the a, and b values in our example. This approach however incurs the overhead of updating the run length values of each of these variables each time a join extends partial matches. Instead LBP breaks the arrays into multiple groups of blocks, which we call *list groups*, which we explain in detail momentarily.

Second, instead of using fixed-length blocks as in traditional processors, the blocks in each group can take different lengths, which are aligned to the lengths of adjacency lists in the database. As we shortly explain, this allows us to avoid materializing adjacency lists into the blocks. Specifically, the size of a list group is either fixed, by default to 1024, if the group has a block that contains vertices (and their properties) that are output by an initial scan operator. Otherwise, the output vertices (and edges and their properties) of one-to-many or many-to-many join operators are put into new list groups whose sizes are dynamic and take different adjacency list sizes during runtime. If a one-to-one or many-to-one join operator extends say vertices u to v, then the blocks that contain vertices v (and $u \rightarrow v$ edges and their properties) are put into an existing list group. We call the union of list groups *intermediate chunk*, which represents a set of intermediate tuples.

Example 3. *Continuing our example, suppose that each c_j has a (single)* ***STUDYAT*** *at edge to d_j. Figure 9 shows the example list groups when LBP evaluates our running example query using the plan in Figure 10. The example contains three list groups. The first group has blocks of size 1024, which contains the variables* ***a.ID*** *and* ***a.age****, output by the* ***Scan*** *operator. The second and third groups have blocks of size k_1 and k_2. The sizes of these groups are determined during run time by the join operators jo_1 and jo_2 that respectively extend vertices a_1 and b_2 to their k_1 and k_2 many* ***FOLLOWS*** *edges. Note that the third join operator in the plan, which extends partial matches with a single car-*

dinality ***STUDYAT*** *edges to* **d** *vertices, does not create a new group. Instead the blocks* **d.ID** *and* **d.city** *are also part of the third list group.*

Each list group has another field `curIdx`, which can either be: (i) a non positive integer that is less than the block sizes in the group; or (ii) -1. If `curIdx` ≥ 0, the list group is flattened and represents a single tuple that consists of the `curIdx`'th values in the blocks. For example, as shown with vertical rectangles in Figure 9, the first and second list groups are flattened to their first and second values, respectively. Otherwise, the list group is not flattened and represents as many tuples as the size of its blocks. Each list group also has an optional `filter mask`, that identifies the tuples that are actually in the block and have not been filtered out. The set of tuples that the entire data chunk represents are the Cartesian product of the tuples from each list group. Therefore, this is a form of factorization [52], and is more lightweight than run-length encoding of blocks to compress intermediate results.

Importantly, similar to traditional block-based processors, every primitive data operator in LBP performs computations on blocks inside loops. For example, the first filter operator in Figure 10 runs on the 1024 tuples in the first list group to filter those with age > 50 and write the necessary bits to the `filter mask` field. Join operators that perform one-to-many or many-to-many joins are the only operators that flatten list groups. For example consider jo_1 in Figure 10 that extends a vertices to their possibly many b neighbors. This operator loops through the 1024 values in the a block, by iteratively flattening the first list group (by setting its `curIdx` field) and then for each a value extends it to a list of b values, which are stored in the second list group. So subsequent operators get a flattened list group 1 and an unflattened list group 2. One-to-one or many-to-one join operators, e.g., jo_3 in Figure 10, does not need to flatten list groups because they take a block of values, e.g. c, and produce new blocks with the same length, e.g., d. Finally similar to columnar RDBMSs and factorized query processors [22, 23] that perform computations directly on compressed data [18, 63] representing intermediate tuples as a set of list groups allows for fast group by and aggregation computations. For example, in a count(*) computation, our group by and aggregation operator simply multiplies the sizes of each list group to compute the number of tuples represented by each intermediate chunk it receives.

Finally, LBP avoids materializing two types of blocks to improve performance. These are shown in gray shade in Figure 9. First are vertex IDs from initial vertex scans, in our example `a.ID`, which are simply consecutive integers. We have a special block implementation, called `nodeIDSequence`, that supports our block interface and avoids any data copies. Second, and more importantly, results of one-to-many and many-to-many joins, which are lists of edge IDs and neighbor vertex IDs are also not materialized. These ID lists are already stored consecutively in CSR structures, so we can exploit this design and implement these blocks simply as pointers to parts of CSRs. This can allow us to even perform some computations without any or very little data copying into intermediate blocks. For example, the query to count k-degree neighbors of a vertex, e.g., MATCH (a)→(b)→(c) WHERE a.ID = 0 RETURN count(*), is evaluated in a plan where operators use blocks that uses a `nodeIDSequence` block and blocks that point to existing adjacency lists.

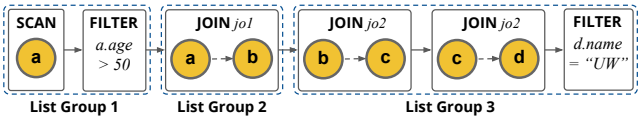

Figure 10: Query plan in LBP for Example 2.

7. EVALUATIONS

We integrated our columnar techniques into GraphflowDB, which is an in-memory GDBMS written in Java. We refer to this version of GraphflowDB as `GF-CL` for **C**olumnar **L**ist-based. We based our work on the publicly available version here [4], which we will refer to as `GF-RV`, for **R**ow-oriented **V**olcano. `GF-RV` uses 8 byte vertex and edge IDs and adopts the interpreted attribute layout to store edge and vertex properties. `GF-RV` also partitions adjacency lists by edge labels and stores the edge ID-neighbor ID pairs inside a CSR. We first present an experiment evaluating the memory improvements of `GF-CL` over `GF-RV` on a large social network dataset generated by the popular LDBC benchmark. We then perform a detailed evaluation of each of our optimizations using microbenchmark queries on several datasets. We end with an end-to-end baseline performance comparisons of `GF-CL` against `GF-RV`, the commercial Neo4j GDBMS system, and the Vertica and MonetDB columnar RDBMS systems on the LDBC social network benchmark (SNB). Unless otherwise specified, we run queries on `GF-CL`.

7.1 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 generated LDBC [32] benchmark's social network data using scale factors 10 and 100, which we refer to as LDBC10 and LDBC100, respectively. In LDBC, all of the edges and edge and vertex properties are structured but several properties and edges are very sparse. LDBC10 contains 30M vertices and 176.6M edges while LDBC100 contains 1.77B edges and 300M vertices. Both datasets contain 8 vertex labels, 15 edge labels, 29 vertex properties, and 5 edge properties. We use LDBC datasets in both our end-to-end and and micro benchmark experiments. To enhance our microbenchmarks further, we use two datasets from the popular Konect graph sets [?] covering two application domains: a Flickr social network [?] and a Wikipedia hyperlink graph [?]. Flickr graph has 2.3M nodes and 33.1M edges while Wikipedia graph has 2.1M nodes and 86.3M edges. Both datasets have timestamps as edge properties.

We describe our queries within each sub-section. The exact queries used in each experiment can be found in our repo [15]. In each experiment, we ran our queries 5 times consecutively and report the average of the last 3 runs.

7.2 Memory Reduction

We demonstrate the memory reduction we get from the columnar storage and compression techniques we studied using LDBC100. We start with `GF-RV` and integrate one additional storage optimization step-by-step ending with `GF-CL`:

(i) `+COLS`: Vertex properties are stored in vertex columns, edge properties in single-directional property pages, and single cardinality edges in vertex columns (instead of CSR).

	GF-RV	+COLS	+NEW-IDS	+O-SUPR	+NULL	GF-CL
Vertex Props.	31.40	19.87 **+1.58x**	19.87 -	19.87 -	19.28 **+1.03x**	- **1.62x**
Edge Props.	7.92	2.07 **+3.82x**	2.07 -	2.07 -	2.05 **+1.01x**	- **3.87**
F. Adj. Lists	31.93	28.95 **+1.10x**	20.67 **+1.40x**	11.41 **+1.81x**	10.78 **+1.06x**	- **2.96**
B. Adj. Lists	31.29	31.07 **+1.01x**	24.93 **+1.25x**	13.10 **+1.90x**	11.41 **+1.15x**	- **2.74**
Total (GB)	102.56	81.97 **+1.25x**	67.55 **+1.21x**	46.45 **+1.45x**	43.54 **+1.07x**	- **2.36**

Table 2: Memory consumption improvements on each data component and the total storage (last row) after applying each optimization on LDBC100. Each column applies a new optimization on top of the configuration on the left. GF-CL column reports the total improvements of GF-CL over GF-RV.

(ii) +NEW-IDS: Introduces our new vertex and edge ID schemes and factors out possible ID components (recall Section 5.2).

(iii) +O-SUPR: Implements leading 0 suppression in the components of vertex and edge IDs in adjacency lists.

(iv) +NULL: Implements NULL compression of empty lists and vertex properties based on Jacobson's index.

Table 2 shows how much memory each storage component of the system takes as well as the total memory consumption of the system after each additional optimization. Overall, we see significant reduction in the memory consumption of each component. We see 2.96x and 2.74x reduction for storing forward and backward adjacency lists, respectively. We reduce memory significantly by adopting our new ID scheme and factoring out components, such as edge and vertex labels, and using small size integers for positional offsets. We also see significant reduction, by 1.58x, in the memory cost of storing vertex properties in columns, which avoids the overhead of storing the keys of the properties explicitly, as done in interpreted attribute layout design. The modest memory gains in +COLS for forward and backward adjacency lists is due to the fact that 8 out of 15 edge labels in LDBC SNB are single cardinality and storing in vertex columns is more lightweight than in CSRs, as it avoids the storage of CSR offsets. We see a reduction of 3.82x in the cost of storing edge properties by adopting single-directional property pages. This is primarily because GF-RV stores a pointer for each edge, even if the edges with a particular label have no properties. GF-CL stores no columns for these edges, so incur no overheads. In addition, similar to vertex properties, we avoid storing the keys of the properties explicitly. We see modest benefits in NULL compression since empty adjacency lists are infrequent in LDBC100 and 26 of 29 vertex properties and all of the edge properties contain no NULL values. Overall, we obtained a reduction of 2.36x on the LDBC100 dataset, reducing the memory cost from 102.5GB to 43.5GB, which demonstrate the potential benefits of applying our techniques to scale in-memory GDBMSs.

7.3 Single-Directional Property Pages

We next demonstrate the query performance benefits of storing edge properties in single-directional property pages compared to the baseline edge column design. We use our new ID scheme and configure GraphflowDB in two ways:

EDGE COLS: Stores edge properties in an edge column, where we assume properties of edges in an adjacency list can appear anywhere within this large column and simulate this by using a random order.

PROP PAGES: Edge properties are stored in forward-directional pages by combining k=128 adjacency lists's properties.

		1-hop	2-hop	3-hop
Forward Plan	EDGE COLS	0.59	71.93	-
	PROP PAGES	0.24 **2.46x**	38.29 **1.88x**	- **-x**
Backward Plan	EDGE COLS	1.25	130.72	-
	PROP PAGES	1.30 **0.96x**	136.49 **0.96x**	- **-x**

Table 3: Runtime (in secs) of k-hop queries when storing edge properties in forward-directional property pages vs edge columns on LDBC100.

We use the LDBC100, Wikipedia, and Flickr datasets. As our workload, we use 1- 2- and 3-hop queries, i.e., queries that enumerate all knows edges, 2-paths, and 3-paths, with predicates on the edges. For LDBC the paths enumerate Knows edges (Wikipedia and Flickr contain only one edge label). Our 1-hop query compare the edge's timestamp for Wikipedia and Flickr and the creationDate property for LDBC to be greater than a constant. Our 2- and 3-hop queries compares the property of each query edge to be greater than the previous edge's property. For Wikipedia and Flickr, which contain a prohibitively many 2- and 3-hops we further put a predicate on the source or destination nodes to make the queries finish within several minutes. When running queries in both configurations, for each query, we consider two plans: (i) the *forward plan* that matches vertices from left to right in the forward direction and applies filters at the earliest position; (ii) the *backward plans* that matches vertices from right to left.

Because forward plans when running under PROP-PAGES perform sequential reads of properties, which achieves good CPU cache locality, we expect them to be more performant than backward plans. We expect backward plans to behave similarly under both configurations. Table 3 shows our results. Observe that forward plans that access properties through forward-directional property pages is between -x to -x faster than those that access through edge columns. In contrast, the differences between the performances of the backward plans are very minor. This is because neither edge columns nor forward-directional property pages provide any locality when accessing properties in the order of backward adjacency lists. This verifies our claim in Section 4.2 that PROP-PAGES is a strictly better columnar design than using vanilla edge columns.

7.4 Vertex Columns for Single Cardinality Edges

In Section 7.2, we showed the memory gains of storing single cardinality edges in vertex columns. Storing single cardinality edges in vertex columns also improves performance because the system can direct access the edge without an indirection through a CSR. We next demonstrate this benefits 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:

(i) 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 7.2.

(ii) CSR-UNC: Single cardinality edge label edges are stored in CSR format and are not compressed.

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

(iii) `V-COL-C`: Null compressed version of `V-COL-UNC`. This is equivalent to `+NULL` configuration in Section 7.2.

(iv) `CSR-C`: Null compressed version of `CSR-UNC`.

We use the LDBC datasets only because the other datasets do not contain single cardinality edges. We use LDBC100. The workload consists of simple 1-, 2-, and 3-hop queries on the `replyOf` edge between `Comment` vertices. To ensure that the joins are the dominant operation in these queries, teh queries do not contain any predicates and return as output the count star aggregation. We evaluate each query with a plan that performs the joins in the forward direction.

Tables 4a and 4b 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`). 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*. Recall that in Section 7.2, we had reported that NULL compression leads to modest memory reduction when we look at the size reduction of the entire database (by 1.07x). This was because majority of the edges and properties are not sparse in LDBC. However, our current experiment demonstrates that compressing empty lists can lead to major memory reduction when an edge label is sparse and contains many empty lists. Specifically, in LDBC100, 50.5% of the `replyOf` forward adjacency lists are empty. The last columns of Tables 4a and 4b report the size of storing `replyOf` edges with vertex columns or CSR. Observe that NULL compressing these lists lead to 1.75x memory reduction when using vertex columns (from 839.93MB vs 478.86MB). Also note that the reduction is still significant but lower, by 1.4x, if we use CSRs because CSRs incur the cost of storing extra offsets which cannot be compressed if we want to maintain constant time access to adjacency lists.

7.5 Null Compression

We demonstrate the memory/performance trade-off of our NULL compression scheme using Jacobson's index. We now focus on compressing sparse property columns. We take the following query: "MATCH (a:`Person`)−[e:`Likes`]→ (b:`Comment`) RETURN b.`creationDate`". Then we create multiple versions of the LDBC100 dataset, in which the `creationDate` property of `Comment` vertices contains different percentage of non-NULL values. LDBC100 contains 220M `Comment` vertices, so our column contains 220M entries. We then evaluate this query with a simple plan that scans a, extends to b, and then a sink operator that reads b.`creationDate`. We compare the query performance and the memory cost of storing the `creationDate` column, when it is stored in three different ways: (i) `J-NULL` compresses the column using Jacobson's bit index with our default configuration (m=16, c=16); (ii) `Vanilla-NULL` is the vanilla bit string-based scheme from reference [16]; and (ii) `Uncompressed` stores the column in an uncompressed format.

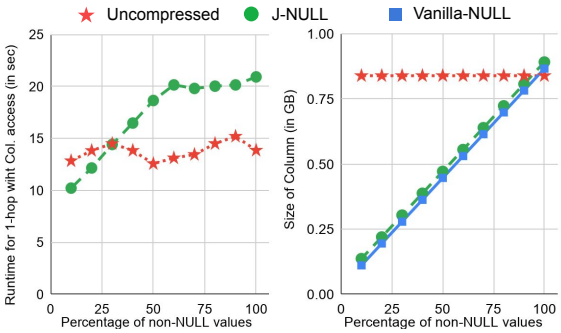

Figure 11: Query performance and memory consumption when storing a vertex property column as uncompressed, compressed with Jacobson's scheme, and the vanilla bit string scheme from Abadi, under different density levels.

Figure 11 shows the memory usage and performance of our query under three different system configurations. Recall that under our default configuration `J-NULL` requires slightly more memory than the vanilla scheme, 2 bits for each element instead of the 1 bit overhead of `Vanilla-NULL`. The memory reduction due to compression is high enough to compensate this overhead until 93.7% of the values are non-NULL. As expected the query performance of `J-NULL` configuration is generally slower than `Uncompressed`, between 1.19x and 1.51x, but much faster than `Vanilla-NULL`, which was >20x slower than `J-NULL` and is therefore omitted in Figure 11. Interestingly, when the column is sparse enough (containing less than 30% non-NULL values), `J-NULL` configuration can even outperform the performance `Uncompressed`. This is because when the column is very sparse, accesses are often to NULL elements, which takes one access for reading the bit value of the element. When the bit value is 0, so the value is NULL, iterators return a global value which is likely to be in the CPU's cache. Instead, `Uncompressed` always returns the value stored in the element's cell, which has a higher chance of a CPU cache miss.

7.6 List-based Processor

We next present experiments demonstrating the performance benefits of LBP against a traditional Volcano-style tuple-at-a-time processor, which are adopted by some existing systems, such as Neo4j [8] or MemGraph [6]. Recall that similar tuple-at-a-time Volcano-style processors, LBP does not duplicate data values under many-to-many joins but has three advantages over traditional tuple-at-a-time processor: (1) all primitive computations over data happen inside loops as in block-based operators; (2) the join operator can avoid copies of edge ID-neighbor ID pairs into intermediate tuples, exploiting the list-based storage; and (3) we can perform group-by and aggregation operations directly on compressed data. We present two separate sets of experiments that

		1-hop	2-hop	3-hop
FILTER	VOLCANO	-	-	-
	LIST-BASED	- **-x**	- **-x**	- **-x**
COUNT(*)	VOLCANO	-	-	-
	LIST-BASED	- **-x**	- **-x**	- **-x**

Table 5: Comparison of Volcano-style query processor and our hybrid list-based query processor's plans.

demonstrate the benefits from these three factors. To ensure that our experiments only test differences due to query processing techniques, we integrated our columnar storage and compression techniques into GF-RV (recall that this is GraphflowDB with row-based storage and Volcano-style processor). We call this version GF-CV, for **c**olumnar **V**olcano, and compare GF-CL against GF-CV.

We use LDBC100, Wikipedia, and Flickr datasets. In our first experiment, we take 1-, 2-, 3-hop queries (as in Section 7.3, we use the Knows edges in LDBC100), where the last edge in the path has a predicate to be greater than a constant (e.g., e.date $> c$). For both GF-CV and GF-CL, we consider the standard plan that scans the left most node, extends right to match the entire path, and a final Filter on the date property of the last extended edge. Our results are shown in the FILTER row of Table 5. We see that GF-CL outperforms GF-CV by around 2x (between 1.92x and 2.23x). This demonstrates the significant performance benefits of performing computations, specifically filters inside loops, and avoiding copies of adjacency lists into intermediate tuples.

In our second experiment, we demonstrate the benefits of performing fast aggregations over compressed intermediate results. We modify the previous queries by removing the predicate and instead add a return value of count(*), so these queries simply count the number of 1-, 2-, and 3-edge paths.We use the same plans as before except we change the last Filter operator with a GroupBy operator. Our results are shown in the COUNT(*) row of Table 5. Observe that the improvements are much more significant now, up to more than two orders of magnitude (565x). The primary advantage of GF-CL is now that the counting happens on compressed intermediate results. In addition it is interesting to note that GF-CL does not copy *any data* into intermediate tuples for these queries. For example, consider the plan for the 2-hop count query. The scan here is implemented with a nodeIDSequence implementation of blocks, which recall does not perform any data copies. Similarly the join operator is performed using pointers to adjacency lists and finally the group-by operator only counts the sizes of these lists, without any data copies.

7.7 Baseline System Comparisons

Our final set of experiments compares the query performance of GF-CL against GF-RV, Neo4j, which is another row-oriented and Volcano style GDBMSs but forms a baseline from an independent code-base, and two column-oriented RDBMSs, MonetDB and Vertica, which are not tailored for many-to-many joins. Our primary goal is to verify that GF-CL's is faster than GF-RV also on an independent end-to-end benchmark. We also aim to verify that the GF-RV system into which we integrated our techniques is already competitive with or outperforms other baseline systems on a workload that contains many-to-many joins. We use the LDBC10 dataset. GraphflowDB is a prototype system and

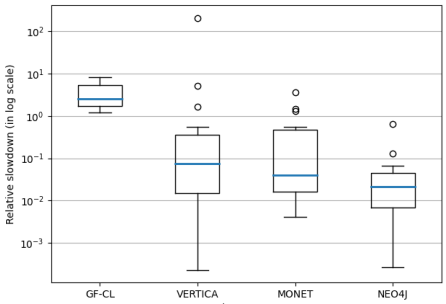

Figure 12: Relative speedup/slowdown of the different systems in comparison to GF-RV on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

currently implements parts of the Cypher query language that are relevant to our research, so lack several features of Cypher that LDBC queries exercise. The system currently has support for select-project-join queries and limited form of aggregations, where joins are expressed as fixed-length subgraph patterns in the MATCH clause. We modified the Interactive Complex Reads (IC) and Interactive Short Reads (IS) queries from LDBC [?] in order to be able to run them. Specifically GraphflowDB does not support variable length queries that search for joins between a minimum and maximum length, which we set to the maximum length to make them fixed-length instead, and shortest path queries, which we removed from the benchmark. We also removed predicates that check the existence or non-existence of edges between nodes and the ORDER BY clauses. Our final workload contains variants of 18 of the 21 queries in the IS and IC benchmarks and can be found in the appendix of the longer version of our paper [?].

We used the the community version v4.2 of the Neo4j GDBMS [8], the community version 10.0 of Vertica [14] and MonetDB 5 server 11.37.11 [7]. We note that our following experiments should not be interpreted as one system being generally more efficient than another, because it is very difficult to meaningfully compare completely separate systems because of many differences across the systems, e.g., all three baseline systems are disk-based, have many tunable parameters, and have more efficient enterprise versions (in the case of Neo4j and Vertica). For all baseline systems, we map their storage to an in-memory file system, set each system to use a single CPU and disable spilling intermediate files to disk. We maintain 2 copies of edge tables for Vertica and MonetDB, sorted by the source and destination vertexIDs, respectively. This gives the systems the option to perform fast merge joins (without sorting) when possible instead of hash joins which require creating hash tables.

Because the joins in LDBC queries start from a particular vertex ID, the join order is obvious and we use plans that are left-deep join trees. We noticed that Vertica and MonetDB sometimes did not pick left-deep plans, in which case we report the better of the system's default plan and the left deep plan, which we can force the systems to pick.

Figure 12 shows the relative speedup/slowdown of the different systems in comparison to GF-RV. We include individual runtime numbers of all the IS and IC queries in the longer version of paper [?]. As we expect, GF-CL is broadly more performant than GF-RV on this benchmark as well with a median query improvement factor of Xx. Importantly, with the exception of one query, which slows down slightly,

the performance of *every query* improves between 1.3x to 8.3x. The improvements come from a combination of optimizations but primarily from LBP and columnar storage of properties. In `GF-RV`, scanning properties require performing equality checks on property keys, which are avoided in our columnar storages, so we observed that on queries that perform large intermediate results and perform filters, such as IC05, we see larger improvements. For this query, there are 4 many-to-many joins starting from a node and extending in the forward direction and a predicate on the edges of the third join. `GF-CL` has several advantages that become visible here. First, `GF-CL`'s LBP unlike `GF-RV` does not copy any edge and neighbor IDs to intermediate tuples. More importantly, LBP performs filters inside loops and `GF-CL`'s single-indexed property pages provides faster access to the edge properties that are used in the filter than `GF-RV`'s interpreted attribute layout format. On this query `GF-RV` takes 8.9s while `GF-CL` takes 1.6s.

As we expected, we also found the other baseline systems to not be as performant as `GF-CL` or `GF-RV`. In particular Vertica, MonetDB, and Neo4j have median slowdown factors of Xx, Yy, and Zz compared to `GF-RV`. Although Neo4j broadly performed slightly worse than other baselines, we also observed that there were several queries in which it outperformed Vertica and MonetDB (but not `GF-RV` or `GF-CL`) by a large margin. These were queries that started from a single node as usual, had several many-to-many joins, but did not generate large intermediate results, such as IS02 or IC06. Interestingly on such queries, GDBMSs, so both GraphflowDB and Neo4j, have the advantage of using join operators that use the adjacency list indices to extend a set of partial matches. This is an implementation of index nested loop joins that can be highly efficient if the partial matches that are extended are small in number. For example the first join of IC06 extends a single `Person` node, say p_i, to its two-degree friends. In SQL, this is implemented as joining a `Person` table with a `Knows` table with a predicate on the Person table to select p_i. In Vertica or MonetDB this join is performed using merge or hash joins, which requires the scan of both `Person` and `Knows` tables. Instead, Neo4j and GraphflowDB only scan the `Person` table to find p_i and then directly extend p_i to its neighbors, avoiding the scan of all `Knows` edges. In this query, `GF-RV`, `GF-CL`, and Neo4j take 333ms, 113ms, and 515ms, while Vertica and MonetDB take 4.7s and 2.7s, respectively. We also found that all baseline systems, including Neo4j, degrade in performance on queries with many many-to-many joins that generate large intermediate results. For example, on IC05 that we reviewed above, Vertica take 1 minute, MonetDB 3.25 minutes, while Neo4j took over 10 minutes.

8. RELATED WORK

Column stores [40, 55, 60, 61] are designed primarily for OLAP applications that perform aggregations over large amounts of data. Work on column stores introduced a set of columnar storage, compression, and query processing techniques which include use of positional offsets, columnar compression schemes, block-based query processing, late materialization, and direct operation on compressed data, among others. A detailed survey of these techniques can be found in reference [17]. This paper studies how to integrate some of these techniques into in-memory GDBMSs. As we argued in this paper, although some of these techniques can be directly integrated, some others, such as traditional block-based processors, NULL compression schemes, or direct columnar storage of edge properties require GDBMS-specific adaptations to optimize them for the access patterns of GDBMSs.

Existing GDBMSs and RDF 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. Systems often use other row-oriented structures, such as property stores that store a sequence of key-value properties, or a key-value store to store properties. For example, Neo4j [8] 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 properties of each vertex and edge are stored in a linked-list in an unstructured manner, where each property record points to the next 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 [5] stores its edges in adjacency lists partitioned by edge labels and properties as 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 [2] uses a key-value store to hold adjacency lists as well as properties. All of the above native GDBMSs adopt Volcano-style processors. In contrast, our design adopts columnar structures for vertex and edge properties and a block-based processor. 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 [12], such as IBM Db2 Graph [56], Oracle Spatial and Graph [12] SAP's graph database [53]. These systems can benefit from the columnar techniques provided by the underlying RDBMS, which are not optimized for graph storage and queries as our columnar structures. For example, SAP's graph engine uses SAP HANA's columnar-storage for edge tables but these tables do not have CSR-like structures for storing edges.

A work that is closely related to ours is GQ-Fast [46]. GQ-Fast implements a restricted set of SQL called relationship queries that contain joins of tables, similar to path queries followed with aggregations. The system stores relationship tables, i.e., those that represent many-to-many relationships, in CSR-like indices with heavy-weight compression of lists and has a fully pipelined query processor that uses query compilation. Therefore, GQ-Fast studies how some techniques in GDBMSs, specifically joins using adjacency lists can be integrated into an RDBMS. In contrast, we focus on studying how some techniques from columnar RDBMSs can be integrated into GDBMSs. We intended to but could not compare against GQ-Fast because the system supports a very limited set of queries (e.g., none of the LDBC queries are supported). In addition the publicly available version had several compilation or runtime errors on sample queries and the system is no longer maintained.

ZipG [44] 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 [21]. Succinct stores semi-structured data that is encoded as a set of key and list of values. For example, a vertex 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. 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, which is slower than our structures.

Several RDF systems also use columnar structures to store RDF databases. Reference [20] stores data in a set of columns, where each column store is a set of (subject, object) pairs for each unique predicate. 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 [57] 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 [51] 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 storages found in GDBMSs.

Several prior work has introduced novel storage techniques for storing graphs that are optimized for write-heavy, such as streaming, workloads. These works propose data structures that try to achieve the sequential read capabilities of CSR while being more optimized for writes. Example systems and data structures include LiveGraph [59], Aspen [30], LLAMA [47], STINGER [31], and DISTINGER [33]. In this paper we focused on a read-optimized system setting and implemented our techniques inside GraphflowDB, which we are designing as a read-optimized system. We therefore directly used CSR to store the graph topology but these techniques are complementary to our work.

Our list groups represent intermediate results in a factorized form. Prior work on query processing in RDBMSs on factorized representations, specifically FDB [22, 23], represents intermediate relations as tries, and have operators that transform tries into other tries. Unlike traditional query processor architectures, processing is not pipelined and all intermediate results are materialized. Instead, operators in LBP are variants of traditional block-based operators and perform computations in a pipelined fashion on batches of lists/arrays of data. This paper focuses on integration of columnar storage and query processing techniques into GDBMSs. and we have not studied how to integrate more advanced factorized processing techniques inside GDBMS within the scope of this paper. This is an interesting direction that is in our immediate future work directions.

9. CONCLUSIONS

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. While some of these techniques can be directly applied to GDBMSs, direct adaptation of others can be significantly suboptimal in terms of space and performance. In this paper, we first outlined a set of guidelines and desiderata for designing the physical storage layer and query processor of GDBMSs, based on the typical access patterns in GDBMSs which are significantly different than the typical workloads of columnar RDBMSs. We then presented our design of columnar storage, compression, and query processing techniques that are optimized for in-memory GDBMSs. Specifically, we introduced a novel list-based query processor, which avoids expensive data copies of traditional block-based processors and avoids materialization of adjacency lists in blocks, a new data structure we call single-indexed property pages and an accompanying edge ID scheme, and a new application of Jacobson's bit vector index for compressing NULL and empty lists. Through extensive experiments, we demonstrated the scalability and performance benefits of our techniques in an actual system implementation.

10. REFERENCES

[1] Amazon Neptune, 2020.
[2] Dgraph, 2020.
[3] DuckDB, 2020.
[4] Graphflowdb Source Code, 2020.
[5] Janus Graph, 2020.
[6] Memgraph, 2020.
[7] MonetDB source code, (Jun2020-SP1), 2020.
[8] Neo4j, 2020.
[9] Neo4j deletions, 2020.
[10] Neo4j Property Graph Model, 2020.
[11] Oracle In-Memory Column Store Architecture, 2020.
[12] Oracle Spatial and Graph, 2020.
[13] TigerGraph, 2020.
[14] Vertica 10.0.x Documentation, 2020.
[15] GraphflowDB Columnar Techniques, 2021.
[16] D. Abadi. Column Stores for Wide and Sparse Data. *CIDR*, 2007.
[17] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 2013.
[18] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. *SIGMOD*, 2006.
[19] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores Vs. Row-Stores: How Different Are They Really? *SIGMOD*, 2008.
[20] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. *PVLDB*, 2007.
[21] R. Agarwal, A. Khandelwal, and I. Stoica. Succinct: Enabling Queries on Compressed Data. *NSDI*, 2015.
[22] N. Bakibayev, T. Kočiský, D. Olteanu, and J. Závodný. Aggregation and ordering in factorised databases. *PVLDB*, 2013.
[23] N. Bakibayev, D. Olteanu, and J. Závodný. Fdb: A query engine for factorised relational databases. *PVLDB*, 2012.
[24] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to Support Sparse Datasets Using an Interpreted Attribute Storage Format. *ICDE*, 2006.
[25] P. Boncz. *Monet: A Next-Generation Database Kernel for Query-Intensive Applications*. PhD thesis, Universiteit van Amsterdam, 2002.
[26] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-Pipelining Query Execution. *CIDR*, 2005.
[27] A. Bonifati, G. Fletcher, H. Voigt, and N. Yakovets. *Querying Graphs*. 2018.
[28] A. Bonifati, P. Furniss, A. Green, R. Harmer, E. Oshurko, and H. Voigt. Schema Validation and Evolution for Graph Databases. *ER*, 2019.

[29] P. Bose, M. He, A. Maheshwari, and P. Morin. Succinct orthogonal range search structures on a grid with applications to text indexing. *Workshop on Algorithms and Data Structures*, 2009.

[30] L. Dhulipala, G. E. Blelloch, and J. Shun. Low-latency graph streaming using compressed purely-functional trees. *SIGPLAN*, 2019.

[31] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. STINGER: High Performance Data Structure for Streaming Graphs. *HPEC*, 2012.

[32] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz. The LDBC Social Network Benchmark: Interactive Workload. *SIGMOD*, 2015.

[33] G. Feng, X. Meng, and K. Ammar. DISTINGER: A Distributed Graph Data Structure for Massive Dynamic Graph Processing. *IEEE Big Data*, 2015.

[34] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. Cypher: An Evolving Query Language for Property Graphs. *SIGMOD*, 2018.

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

[36] G. Gonnet, R. Baeza-Yates, and T. Snider. *New Indices for Text: Pat Trees and Pat Arrays*. 1992.

[37] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. *TKDE*, 1994.

[38] G. Graefe and L. Shapiro. Data Compression and Database Performance. *High-Performance Web Databases*, 2001.

[39] O. Hartig and J. Hidders. Defining Schemas for Property Graphs by Using the GraphQL Schema Definition Language. *GRADES-NDA*, 2019.

[40] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.*, 2012.

[41] G. Jacobson. Space-efficient static trees and graphs. *Symposium on Foundations of Computer Science*, 1989.

[42] G. Jacobson. *Succinct static data sttlctates.* PhD thesis, Carnegie Mellon University, 1989.

[43] C. Kankanamge, S. Sahu, A. Mhedbhi, J. Chen, and S. Salihoglu. Graphflow: An Active Graph Database. *SIGMOD*, 2017.

[44] A. Khandelwal, Z. Yang, E. Ye, R. Agarwal, and I. Stoica. ZipG: A Memory-Efficient Graph Store for Interactive Queries. *SIGMOD*, 2017.

[45] D. Lemire and L. Boytsov. Decoding Billions of Integers Per Second Through Vectorization. *Softw. Pract. Exp.*, 2015.

[46] C. Lin, B. Mandel, Y. Papakonstantinou, and M. Springer. Fast in-memory sql analytics on typed graphs. *ICDE*, 2016.

[47] P. Macko, V. J. Marathe, D. W. Margo, and M. I. Seltzer. Llama: Efficient graph analytics using large multiversioned arrays. *ICDE*, 2015.

[48] A. Mhedhbi and S. Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *PVLDB*, 2019.

[49] G. Navarro and V. Mäkinen. Compressed Full-Text Indexes. *CSUR*, 2007.

[50] G. Navarro and L. M. S. Russo. Space-efficient data-analysis queries on grids. *Algorithms and Computation*, 2011.

[51] T. Neumann and G. Weikum. The RDF3X Engine for Scalable Management of Rdf Data. *VLDBJ*, 2010.

[52] D. Olteanu and M. Schleich. Factorized Databases. *SIGMOD Rec.*, 2016.

[53] M. Rudolf, M. Paradies, C. Bornhövd, and W. Lehner. The Graph Story of the SAP HANA Database. *BTW*, 2013.

[54] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDBJ*.

[55] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran, and S. Zdonik. C-store: A Column-oriented DBMS. *VLDB*, 2005.

[56] Y. Tian, E. L. Xu, W. Zhao, M. H. Pirahesh, S. J. Tong, W. Sun, T. Kolanko, M. S. H. Apu, and H. Peng. Ibm db2 graph: Supporting synergistic and retrofittable graph queries inside ibm db2. 2020.

[57] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple Indexing for Semantic Web Data Management. *PVLDB*, 2008.

[58] H. Zhang, H. Lim, V. Leis, D. G. Andersen, M. Kaminsky, K. Keeton, and A. Pavlo. Succinct range filters. *TODS*, 2020.

[59] X. Zhu, G. Feng, M. Serafini, X. Ma, J. Yu, L. Xie, A. Aboulnaga, and W. Chen. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *PVLDB*, 2020.

[60] M. Zukowski and P. Boncz. From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About. *SIGMOD*, 2012.

[61] M. Zukowski and P. A. Boncz. Vectorwise: Beyond Column Stores. *IEEE Data Eng. Bull.*, 2012.

[62] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100-A DBMS in the CPU Cache. *IEEE Data Eng. Bull.*, 2005.

[63] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-Scalar RAM-CPU Cache Compression. *ICDE*, 2006.