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 their access patterns. We then present the design of columnar storage, compression, and query processing techniques based on these desiderata. In addition to showing direct integration of existing techniques from columnar RDBMSs, we also propose novel ones that are optimized 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, 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 values and empty lists. We integrated our techniques into the GraphflowDB in-memory GDBMS. Through extensive experiments, we demonstrate the scalability and query performance benefits of our techniques.

1. INTRODUCTION

Contemporary GDBMSs such as Neo4j [9], Neptune [1], TigerGraph [13], and GraphflowDB [44, 52] that adopt the property graph data model [10]. In this model, application data is represented as a set of vertices and edge, which represent the entities and their relationships, and key-value properties on the vertices and edges. GDBMSs support a wide range of analytical applications, such as fraud detection and recommendations in financial, e-commerce, or social networks [61] that search for patterns in a graph-structured database, which require reading large amounts of data. In the context of RDBMSs, column-oriented systems [11, 41, 63, 69] employ a set of read-optimized storage, indexing, and query processing techniques to support traditional analytical applications, such as business intelligence and reporting, that also process large amounts of data. As such, these 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 and compression techniques for storing different components of graphs [17, 19, 63, 71], and block-based query processing [20, 26]. Despite their similarities, workloads in GDBMSs and columnar RDBMSs also have fundamentally different access patterns. For example, workloads in GDBMSs contain large many-to-many joins, which are not frequent in column-oriented RDBMSs. 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 data components in GDBMSs. While existing columnar structures can directly be used for storing vertex properties and many-to-many (n-n) edges, we observe that using straightforward edge columns, to store properties of n-n edges 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. 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. This requires using a new edge ID scheme that is conducive to extensive compression when storing them in adjacency lists without any 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 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. We next show that existing techniques for compressing NULL values in columns from references [17, 19] by Abadi et al. lead to very slow accesses to arbitrary non-NULL values. We then review Jacobson's bit vector index [42, 43] to support constant time rank queries, which has

found several prior applications e.g., in a range filter structure in databases [67], in information retrieval [35, 54] and computational geometry [29, 55]. We show how to enhance one of Abadi's schemes with an adaptation of Jacobson's index to provide constant-time access to arbitrary non-NULL values, with a small increase in storage overhead per entry compared to prior techniques.

List-based Processing: In Section 6, we observe that traditional block-based processors or columnar RDBMSs [20, 70] process fixed-length blocks of data in tight loops, which achieves good CPU and cache utility but 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 factorized 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. (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 us to avoid materializing adjacency lists during join processing, improving query performance.

We integrated our techniques into GraphflowDB [44]. We present extensive experiments that demonstrate the scalability and performance benefits (and tradeoffs) of our techniques both on microbenchmarks and on the LDBC and JOB benchmarks against a row-based Volcano-style implementation of the system, an open-source version of a commercial GDBMSs, and two column-oriented RDBMSs. Our code, queries, and data are available here [15].

2. BACKGROUND

In the property graph model, vertices and edges have labels and arbitrary key value properties. Figure 1 shows a property graph that will serve as our running example, which contains vertices with `PERSON` and `ORGANIZATION (ORG)` labels, and edges with `FOLLOWS`, `STUDYAT` and `WORKAT` labels.

There are three storage components of GDBMSs: (i) topology, i.e., adjacencies of vertices; (ii) vertex properties; and (iii) edge properties. In every native GDBMS we are aware of, the topology is stored in data structures that organize data in *adjacency lists* [27], such as in compressed sparse row (CSR) format. Typically, given the ID of a vertex v, the system can in constant-time access v's adjacency list, which contains a list of (edge ID, neighbour ID) pairs. Typically, an adjacency list of v is further clustered by the edge label which enables traversing the neighbourhood of v based on a particular label efficiently. 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 in 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

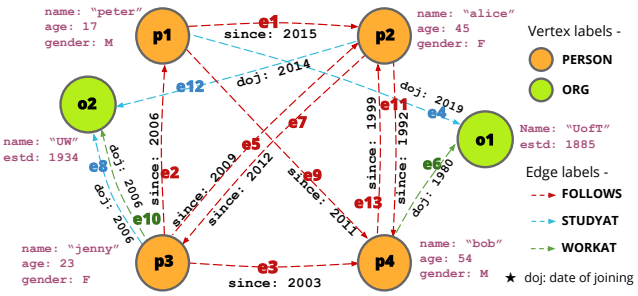

Figure 1: Running example graph.

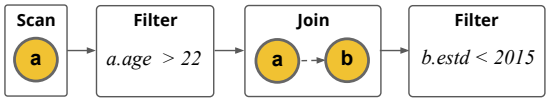

Figure 2: Query plan for the query in Example 1.

a query processor that uses variants of the following relational operators, which is the case in many GDBMSs, e.g., Neo4j [9], Memgraph [6], or GraphflowDB:

`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 qv_s or qv_d has already been matched. For example if qv_s has already been matched to data vertex v_i, then the operator produces one $(k+1)$-match for each edge-neighbor pair in v_i's forward adjacency list[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 computation on a partial match t.

Example 1. *Below is an example query written in the Cypher language [32]:*

```
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 2015. Figure 2 shows a typical plan for this query.

3. GUIDELINES AND DESIDERATA

We next outline a set of guidelines and desiderata for organizing the physical data layout and query processor of GDBMSs. We assume edges are doubly-indexed in forward and backward adjacency lists, as in every GDBMS we are aware of. We will not optimize this duplication as this is needed for fast joins from both ends of edges.

Guideline 1. *Edge and vertex properties are read in the same order as the edges appear in adjacency lists 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 $L_{v_i} = \{(e_{i1}, v_{i1}) ..., (e_{i\ell}, v_{i\ell})\}$. 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. Our first desiradata is to store the properties of e_{i1} to $e_{i\ell}$ sequentially in the same order. Ideally, a system should also store the properties v_{ij} sequentially in the same order

[1] GraphflowDB can perform an intersection of multiple adjacency lists if the pattern is cyclic (see reference [52]).

but in general this would require prohibitive data replication because while each e_{ij} appears in two adjacency lists, each v_{ij} appears in as many lists as the degree of v_{ij}.

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 operators access vertex properties. In addition, real-world graph data with many-to-many relationships have power-law degree distributions [49]. So, there are often many short adjacency lists in the dataset. For example, the `FLICKR`, `WIKI` graphs we use have single edge labels with average degrees of only 14 and 41, and the Twitter dataset used in many prior work on GDBMSs [45] has a degree of 35. 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 implies that techniques that require decompressing blocks of data, say a few KBs, to only read a single vertex property or a single short adjacency list can be prohibitively expensive.

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 model is semi-structured, data in GDBMSs often have some structure. One reason this structure exists is that the data in GDBMSs sometimes comes from structured data from RDBMSs as observed in a recent user survey [61]. In fact, several vendors and academics are actively working on defining a schema language for property graphs [28, 39]. Common structure are:

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

(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.

(iii) *Edges with single cardinality.* Edges might have 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 `organization` has one `isLocatedIn` edge.

We refer to edges that satisfy properties (i) and (ii) as *structured edges* and properties that satisfy property (ii) as *structured vertex/edge property*. Other edges and properties will be called *unstructured*. The existence of such structure in some graph data 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. We start with directly applicable structures and then describe our new single-indexed property pages structure and its accompanying edge ID scheme to store edge properties.

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.

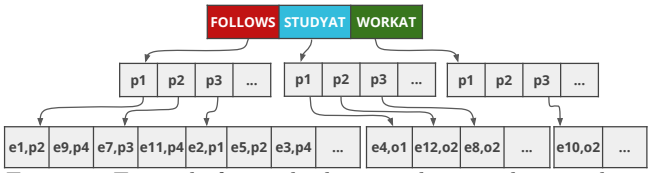

Figure 3: Example forward adjacency lists implemented as a 2-level CSR structure for the example graph.

4.1 Directly Applicable Structures

4.1.1 CSR for n-n Edges

CSR is an existing columnar structure that is widely used 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. In this work, we store the edges of each edge label with n-n cardinality in a separate CSR. As we discuss next, we can store the edges with other cardinalities more efficiently than a CSR using vertex columns.

4.1.2 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 and ensure that offsets with the same label are consecutive. As we discuss in Section 5.2, this ID scheme also can be compressed by factoring out vertex labels.

Similarly, we can store single cardinality edges, i.e., those with 1-1, 1-n, or n-1 constraints, and their properties directly as a *property* of source or destination vertex of the edges in a vertex column and directly access them using a vertex positional offset. As we momentarily discuss, this is more efficient both in terms of storage and access time than the structures we cover for storing properties of n-n edges (Desideratum 3). Figure 4 shows single cardinality STUDYAT and WORKAT edges from our example and their properties stored as vertex column of `PERSON` vertices.

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*

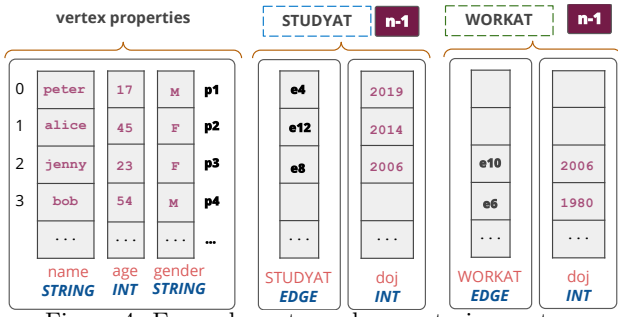

Figure 4: Example vertex columns storing vertex properties and single-cardinality edges and their properties.

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 store the properties according to any order. 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. This can often be prohibitive especially for in-memory systems, as many graphs have orders of magnitude more edges than vertices.

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. Then given say e_1, we need to be able to read e_1's $q_{i,j}$ property 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 offset o of e_1 in L_{nbr_1}, which may require scanning the entire L_{nbr_1}, which is not constant time.

Instead, 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 o directly from edge IDs, so reading edge properties in the opposite direction (backward in our example) can now be constant time. In addition, this scheme can be more space-efficient than schemes that assign consecutive IDs to all edges as its first two components can often be compressed (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 a $q_{i,j}$ property 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 of tuples, are implicit and not explicitly stored. Therefore, when deletions happen, GDBMSs initially leave gaps in their data structures, and recycle deleted IDs when insertions happen. For example, Neo4j's nodestore.db.id file keeps track of deleted IDs for later recycling [8]. Similarly, the list-level positional offsets 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.

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

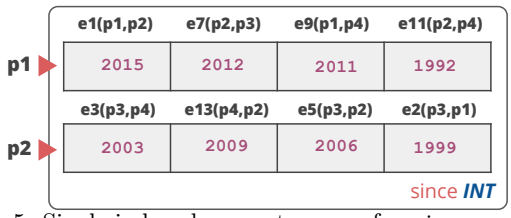

Figure 5: Single-indexed property pages for `since` property of `FOLLOWS` edges in the example graph. $k = 2$.

Our *single-indexed property pages* addresses these two issues (Figure 5). 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. For reference, Figure 5 shows the single-indexed property pages in the forward direction for `since` property of edges with label `FOLLOWS` when k=2.

5. COLUMNAR COMPRESSION

Data compression and query processing on compressed data are extensively used in columnar RDBMSs. To present our complete integration, we start by reviewing techniques that are directly applicable to GDBMSs 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 from columnar RDBMSs [17, 19] and enhance one of them with Jacobson's bit vector index to make the 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 [19, 34, 48] for details of other fixed-length schemes, such as frame of reference.

Dictionary encoding: This is perhaps the most common encoding scheme to be used in RDBMSs [19, 71, 66]. This scheme maps a domain of values into compact codes using a variety of schemes [19, 37, 71], some producing variable-length codes, such as Huffmann encoding, and others fixed-length codes [19]. 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: Given a block of data, this scheme omits storing leading zero bits in each value of the block [24]. 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 $\lceil log_2(t)/8 \rceil$ 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 can be factored out when the data depicts some structure (Desideratum 3). This enables us to compress them without needing to decompress them while scanning. 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 vertex IDs inside the edge ID can be omitted because this is the neighbour vertex ID, which is already stored in the pairs. Second edge labels can be omitted because we cluster our adjacency lists by edge label. The only components 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 of a particular label do not have any properties and only represent the existence of relationships between vertices. For example 10 out of 15 edge labels in LDBC SNB do not have any properties. In this case, edges do not need to be identifiable, as the system will not access their properties. 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, e.g., `Knows` edges in social networks are between `Person` nodes. In this case, we can omit storing the vertex label of the neighbour ID.
- *Single cardinality edges:* Recall from Section 4.1.2 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. So, the page-level positional offsets of these edges can be omitted.

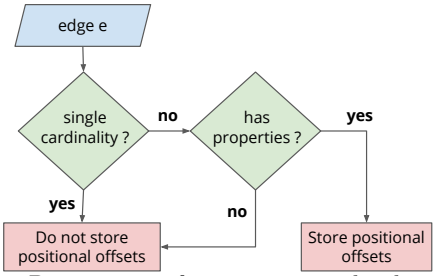

Figure 6: Decision tree for storing page-level positional offsets of edges in adjacency lists.

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 real-world structured graph data. Similarly, many vertices can have empty adjacency lists in CSRs. Both can be seen as different columnar structures containing NULL values. Abadi in reference [17] describes a design space of optimized techniques for compressing NULLs in columns. 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 values. First technique in Abadi's paper, 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 because they do not allow constant-time access to non-NULL values (Desideratum 2). To support constant-time access to a non-NULL value at position p, the secondary structure needs to support two operations in constant time: (i) check if p 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 enhance this design with a simplified version of Jacobson's bit vector index [42, 43]. Figure 7 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, and ps the array storing prefix sums in a block. Then $\text{rank}(p) = ps[p/c] + M[b, p \mod c]$.

The choice of c primarily affects how big the pre-populated map is. A second parameter in this scheme is the number of bits m used 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) $2^c * c * \lceil \log(c)/8 \rceil$ byte size map, because the map has 2^c cells and needs to store a $\log(c)$-bit count value

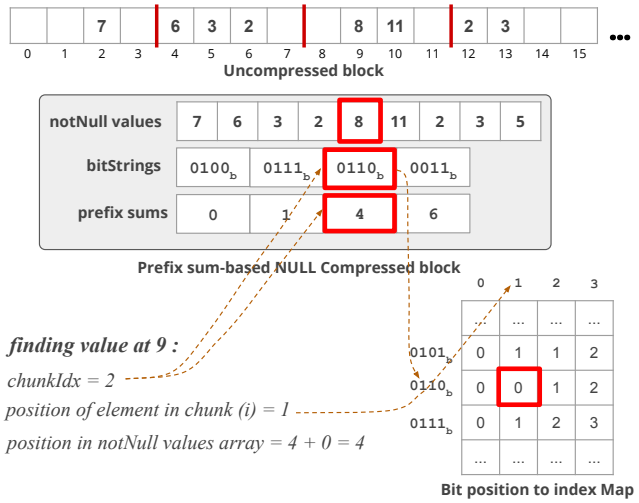

Figure 7: NULL compression using a simplified Jacobson's bit vector rank index with chunk size 4.

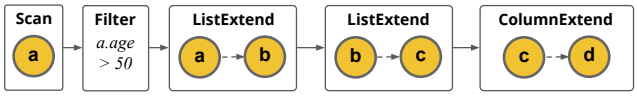

Figure 8: Query plan for the query in Example 2.

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 default we choose $m = 16, c = 16$, we require $2^c * c * 1 =$ 1MB-size map, can compress $2^m =$ 64K blocks, and incur $m/c = 1$ extra bit overhead for each element, so increase the overhead of reference [17]'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 motivate our list-based processor by discussing the shortcomings of traditional Volcano-style tuple-at-a-time processors and block-based processors of columnar RDBMSs when processing many-to-many joins.

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 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 [36], which some GDBMSs adopt [9, 52], is efficient in terms of how much data is copied to the intermediate tuple. Suppose the scan matches `a` to a_1 and a_1 extends to k_1 many `b`'s, $b_1 \ldots b_{k_1}$, and each b_i extends to k_2 many `c`'s to $b_{ik_2} \ldots, b_{(i+1)k_2}$ (let us ignore the `d` extension for now). Although this generates $k_1 \times k_2$ tuples, the value a_1 would be copied only once to the tuple. This is an important advantage for processing many-to-many joins. However, Volcano-style processors are known to achieve low CPU and cache utility as processing is intermixed with many iterator calls.

Column-oriented RDBMSs instead adopt block-based processors [25, 41], which process an entire block at a time in

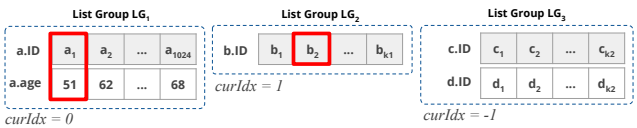

Figure 9: Intermediate chunk for the query in Example 2. The first two list groups are flattened to single tuples, while the last one represents k_2 many tuples.

operators. [3] 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_2 and $k_1 < k_2$. In our example, the scan would output an array $a : [a_1]$, the first join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_{k_1}]$ blocks, and the second join would output $a : [a_1, ..., a_1]$, $b : [b_1, ..., b_1]$, $c : [c_1, ..., c_{k_2}]$, where for example the value a_1 gets copied k_2 times into intermediate arrays. (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.

We developed a new block-based processor called *list-based processor* (LBP), which we next describe. LBP uses *factorized representation of intermediate tuples* [23, 58, 59] to address the data copying problem and uses block sizes that are aligned to the lengths of adjacency lists in the database, to exploit list-based data storage in GDBMSs.

6.1 Intermediate Tuple Set Representation

Traditional block-based processors represent intermediate data as a set of *flat* tuples in a single group of blocks/arrays. 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 are necessary. To address these repetitions we adopt a *factorized tuple set representation* scheme [59]. Instead of flat tuples, factorized representation systems represent tuples as unions of Cartesian products. For example, the k_2 flat tuples $[(a_1, b_1, c_1) \cup (a_1, b_1, c_2) \cup ... \cup (a_1, b_1, c_{k_2})]$ from above can be represented more succinctly in a factorized form as: $[(a_1) \times (b_1) \times (c_1 \cup ... \cup c_{k_2})]$.

To adopt factorization in block-based processing, we instead use multiple groups of blocks, which we call *list groups*, to represent intermediate data. Each list group has a `curIdx` field and can be in one of two states:

- Flat: If `curIdx` ≥ 0, the list group is flattened and represents a single tuple that consists of the `curIdx`'th values in the blocks.

[3] Block-based processing has been called vectorized processing in the original work on MonetDB/X100 from CWI [**?**]. 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.

- Unflat list of tuples: If `curIdx` $=-1$, the list groups represent as many tuples as the size of the blocks it contains.

We call the union of list groups *intermediate chunk*, which represents a set of intermediate tuples as the Cartesian Product of each tuple that each list group represents.

EXAMPLE 3. *Figure 9 shows an intermediate chunk, that consists of three list groups. The first two groups are flattened and the last is unflat. In its current state, the intermediate chunk represents k_2 intermediate tuples as:* $(a_1, 51) \times (b_1) \times ((c_1, d_1) \cup ... \cup (c_2, d_2))$.

In addition, instead of using fixed-length blocks as in existing block-based procewssors, 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.

6.2 Operators

We next give a description of the main relational operators we implemented to process intermediate chunks in LBP.

`Scan`: Scans are as before and read a fixed size (1024 by default) nodeIDs into a block in a list group.

`ListExtend` and `ColumnExtend`: In contrast to the single `Join` operator that implements index nested loop join algorithm using the adjacency list indices, such as `Expand` of Neo4j, we have two join operators. `ListExtend` is used to perform joins from a node with variable, say, a to nodes b over one-to-many or many-to-many edges e. The input list group LG_a that holds the block containing a values can be flat or unflat. If LG_a is not flat, `ListExtend` first flattens it, i.e., sets the `curIdx` field of the list group to 0. Then in a for loop, it loops through each a value, say, a_ℓ, and extends it to a set of b and e values using a_ℓ's adjacency list Adj_{a_ℓ}. The blocks holding b and e values are written to a new list group, LG_b. This allows factoring out a list of b and e values for a single a value. The lengths of all of the blocks in LG_b, including those storing b and e as well as other blocks that may be added to LG_b later, will be equal to the length of Adj_{a_ℓ}. This contrasts with fixed block sizes in existing block-based processors. In addition, we exploit that Adj_{a_ℓ} already stores the extended b and e values as lists, and do not copy these values to the intermediate chunk. Instead, the b and e blocks are simply pointers to Adj_{a_ℓ}.

`ColumnExtend` is used to perform one-to-one or many-to-one joins. We call the operator `ColumnExtend` because recall from Section 4.1.2 that we store such edges in vanilla vertex *columns*. Suppose now that each a can extend to at most one b node. `ColumnExtend` expects a block of unflat a values. That is, it expects LG_a to be unflat and adds two new blocks into LG_a for storing b and e that are of the same length as a's block (so unlike `ListExtend` does not create a new list group). Inside a for loop, `ColumnExtend` copies the matching e and b of each a from the vertex column to these two blocks. Note that because each a value has a single b and e value, these values do not need to be factored out.

`Filter`: Our implementation of LBP requires a more complex filter operator than those in existing block-based processors. In particular, in traditional block based processors, binary expressions, such as a numeric comparison expression, can always assume that their inputs are two blocks of values. Instead, now binary expressions need to operate on three possible value combinations: two flat, two lists or one list and one flat, because any of the two blocks can now be in a flattened list group.

`Group By And Aggregate`: We omit a detailed description of this operator and refer the reader to our code base [15]. Briefly, similar to `Filter`, `Group By And Aggregate` operator needs to consider whether the values it should group by or aggregate are flat or not, and performs a group by and aggregation on possibly multiple factorized tuples. Factorization allows LBP to sometimes perform fast group by and aggregations, similar to prior techniques that compute aggregations on compressed data [18, 19]. For example, simple count(*) computation simply multiplies the sizes of each list group to compute the number of tuples represented by each intermediate chunk it receives.

EXAMPLE 4. *Continuing our example, the three list groups in Figure 9 are an example intermediate chunk that is output by the `ColumnExtend` operator in the plan from Figure 8. In the example, the initial `Scan` and `Filter` have filled the 1024-size a and a.age blocks in LG_1. The first `ListExtend` has: (i) flattened LG_1 to the first tuple $(a_1, 51)$; and (ii) filled a block of k_1 b values in a new list group LG_2. The second `ListExtend` has (i) flattened LG_2 and iterated over it once, so its `curIdx` field is 1, and LG_2 now represents the tuple (b_2); and (ii) has filled a block of k_2 c values in a new list group LG_3. Finally, the last `ColumnExtend` fills a block of k_2 d values also in LG_3 by extending each c_j value to one d_j value through the single cardinality `STUDY_AT` edges.*

7. UPDATES AND QUERY OPTIMIZATION

Although we do not focus on handling updates and query optimization within the scope of this paper, these components require further considerations in a complete integration of our techniques. As in columnar RDBMSs, the columnar storage techniques we covered are read-optimized and necessarily add several complexities to updates [18]. First recall from Section 4.1.1 that CSR data structure for storing adjacency list indexes are effectively sorted structures that are compressed by run-length encoding. So handling deletions or insertions requires resorting the CSRs and recalculating the CSR offsets. Insertions into edge properties stored in single-directional property pages are append only and do not require any sorting. Insertions into vertex columns also do not add any complexities as these are pure unsorted columns. However, deletions of nodes or edges, require leaving gaps in vertex columns and single-directional property pages. This requires keeping track of these gaps and reusing them for newly added vertices and edges. Note that this is also how node deletions are handled in Neo4j [8]. Finally, the null compression scheme we adopted requires three updates upon insertion and deletions: (i) changing the bit values in the bitstrings; (ii) re-calculating prefix sum values for the prefixes after the location of the update; and (iii) shifting the non-NULL elements array. These additional complexities are an inherent trade off when integrating read-optimized techniques and can be mitigated by several existing techniques, such as bulk updates or keeping a small write-optimized second storage that keeps track of the recent writes, which are not immediately merged. Positional delta trees [40] or C-Store's write-store are examples [63] of the latter technique.

Two of our techniques also require additional considerations when modifying the optimizer of GDBMSs. First, our use of factorized list groups changes the size of tuples that

are passed between operators, as the intermediate tuples are now compressed. When assigning costs to plans, the compressed sizes, instead of the flattened sizes of these tuples should be considered. In addition, scans of properties that are stored in, say forward single-directional property pages behave differently when the properties are scanned in the forward direction (e.g., after a join that has used the forward adjacency lists) vs backward direction. The former leads to sequential reads while the latter to random reads. The optimizer can consider this criterion when assigning costs as well. We leave a detailed study of how to handle updates and optimize queries under our techniques to future work.

8. 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 present both microbenchmark experiments comparing `GF-RV` and `GF-CL`, and baseline experiments against Neo4j, MonetDB, and Vertica using LDBC and JOB benchmarks.

8.1 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. 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.

Datasets: Our LBP is designed primarily to yield benefits under join queries over one-to-many and many-to-many relationships. Our storage compression techniques exploit some structure in the dataset and NULLs. These techniques are not designed for datasets that do not depict structure, e.g., a highly heterogenous knowledge graph, such as DBPedia. We choose the following datasets and queries that satisfy these requirements:

LDBC: We generated the LDBC social network data [31] 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.

JOB: We used the IMDb movie database and the JOB benchmark [47] Although this workload has been originally created to study optimizing join order selection, the dataset contains several many-to-many, one-to-many, and one-to-one relationships between entities like actors, movies, and companies and contains structured properties, some of which contain NULLs. This makes it suitable to demonstrate the benefits from our storage and compression techniques. In addition JOB contains join queries over the many-to-many relationships, making it suitable to demonstrate benefits of LBP. We created property graph version of this database and workload as follows. IMDb contains three groups of tables: (i) *entity tables* representing entities, such as actors (e.g., `name` table), movies, and companies; (ii) *relationship tables* representing many-to-many relationships between the entities (e.g., the `movie_companies` table represents relationships between movies and companies); and (iii) *type tables*, which denormalize the entity or relationship tables to indicate the types of entities or relationships. We converted each row of an entity table to a vertex. Let u and v be vertices representing, respectively, rows r_u and r_v from tables T_u an T_v. We added two sets of edges between u and v: (i) a *foreign key edge* from u to v if the primary key of row r_u is a foreign key in row r_v; (ii) a *relationship edge* between u to v if a row r_ℓ in a relationship table T_ℓ connects row r_u and r_v. Our final dataset can be found in our codebase [15].

`FLICKR` and `WIKI`: To enhance our microbenchmarks further, we use two additional datasets from the popular Konect graph sets [45] covering two application domains: a Flickr social network (`FLICKR`) [53] and a Wikipedia hyperlink between articles of the German Wikipedia graph (`WIKI`) [16]. 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.

In each experiment, we ran our queries 5 times consecutively and report the average of the last 3 runs. We did not observe large variances in these experiments. Across all of the LDBC and JOB benchmark queries we report, the median difference between the minimum and maximum of the 3 runs we report was 1.02% and the largest was 25%, which was a query in which the maximum run was 24ms while the minimum was 19ms.

8.2 Memory Reduction

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

(i) `+COLS`: Stores vertex properties in vertex columns, edge properties in single-directional property pages, and single cardinality edges in vertex columns (instead of CSR).
(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 2a shows how much memory each component of the system as well as the entire system take after each optimization. On LDBC, 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 1.58x reduction by storing vertex properties in columns, which avoids the overhead of storing the keys of the properties explicitly, which the interpreted attribute layout design stores. The modest memory gains in `+COLS` for storing adjacency lists is due to the fact that 8 out of 15 edge labels in LDBC SNB are single cardinality and storing them in vertex columns is more lightweight than in CSRs, as it does not store the CSR offsets. We see a reduction of 3.82x when storing edge properties in 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

	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**	- **3.87x**
F. Adj. Lists	31.93	28.95 +**1.10x**	20.67 +**1.40x**	11.41 +**1.81x**	10.78 +**1.06x**	- **2.96x**
B. Adj. Lists	31.29	31.07 +**1.01x**	24.93 +**1.25x**	13.10 +**1.90x**	11.41 +**1.15x**	- **2.74x**
Total (GB)	102.56	81.97 +**1.25x**	67.55 +**1.21x**	46.45 +**1.45x**	43.54 +**1.07x**	- **2.36x**

(a) LDBC100

	GF-RV	+COLS	+NEW-IDS	+O-SUPR	+NULL	GF-CL
Vertex Props.	2.54	1.98 +**1.28x**	1.98 -	1.98 -	1.96 +**1.01x**	- **1.29x**
Edge Props.	2.81	1.63 +**1.72x**	1.63 -	1.63 -	0.90 +**1.83x**	- **3.14x**
F. Adj. Lists	1.13	1.02 +**1.10x**	0.65 +**1.57x**	0.41 +**1.57x**	0.36 +**1.15x**	- **2.96x**
B. Adj. Lists	1.10	1.10 +**1.00x**	0.76 +**1.45x**	0.50 +**1.51x**	0.49 +**1.01x**	- **2.20x**
Total (GB)	7.57	5.74 +**1.32x**	5.02 +**1.14x**	4.53 +**1.11x**	3.72 +**1.22x**	- **2.03x**

(b) IMDb

Table 2: Memory reductions after applying one more optimization on top of the configuration on the left.

edges, so incurs no overheads and avoids 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 LDBC100, reducing the memory cost from 102.5GB to 43.5GB.

The reductions on IMDb are shown in Table 2b and are broadly similar to LDBC. For example, we see 2.96x and 2.2x reductions in the storage of forward and backward lists, which is comparable to the 2.96x and 2.74x factors of LDBC. There are two main differences. First, we see more visible reduction in compressing the edge properties using NULL compression, because 7 of the 12 edge properties in IMDb have more than 50% null values. Second, instead of a 3.82x reduction by storing edge properties using single directional property columns and single cardinality edges in vertex columns (`+COLS` column of `Edge Props` row), the factor is now 1.72x. This is because all of the edge properties in LDBC are 4-byte integers. Instead, IMDb has primarily string edge properties (8 out of 12 of the edge properties), so these take more space compared to integers. Hence, the storage savings per byte of actual data is higher in case of LDBC. Overall, the total reduction factor is 2.03x reducing the memory overheads from 7.57G to 3.72G.

8.3 Single-Directional Property Pages

We next demonstrate the query performance benefits of storing edge properties in single-directional property pages. We configure GraphflowDB in two ways:

`EDGE COLS`: Stores edge properties in an edge column. We assume edges get random global IDs, so their properties can appear anywhere within this column.

`PROP PAGES`: Edge properties are stored in forward-directional pages by combining k=128 adjacency lists's properties. In Appendix A, we describe a sensitivity analysis to justify this default value of k that we choose in our experiments and actual system implementation.

		LDBC100		WIKI		FLICKR	
		1H	**2H**	**1H**	**2H**	**1H**	**2H**
P_F	COL_E	0.55	65.22	2.97	42.92	1.88	888.30
	$PAGE_P$	0.16 **3.4x**	34.22 **1.9x**	0.96 **3.1x**	16.48 **2.6x**	0.42 **4.5x**	189.39 **4.7x**
P_B	COL_E	1.23	131.01	6.33	99.28	2.40	1009.84
	$PAGE_P$	1.29 **0.9x**	134.43 **1.0x**	6.10 **1.0x**	91.75 **1.1x**	2.25 **1.1x**	1183.14 **0.9x**

Table 3: Runtime (in secs) of k-hop (H) queries when using property pages ($PAGE_P$) vs edge columns (COL_E).

We use the `LDBC100`, `WIKI`, and `FLICKR` datasets. As our workload, we use 1- and 2-hop queries, i.e., queries that enumerate all edges and 2-paths, with predicates on the edges. For LDBC the paths enumerate `Knows` edges (`WIKI` and `FLICKR` contain only one edge label). Our 1-hop query compares the edge's timestamp for `WIKI` and `FLICKR` and the `creationDate` property for LDBC to be greater than a constant. Our 2-hop query compares the property of each query edge to be greater than the previous edge's property. For `WIKI`, which contain prohibitively many 2-hops we further put a predicate on the source and destination nodes to make the queries finish within a reasonable time. 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; (ii) the *backward plans* that matches vertices from right to left.

Forward plans perform sequential reads of properties under `PROP-PAGES`, which achieves good CPU cache locality. We therefore expect them to be more performant than backward plans under `PROP-PAGES` as well as the forward plans under `EDGE COLS`, which both lead to random reads. We also expect backward plans to behave similarly under both configurations. Table 3 shows our results. Observe that forward plans under `PROP-PAGES` is between 1.8x to 4.8x faster than the forward plans under `EDGE COLS` and are also faster than the backward plans under `PROP-PAGES`. In contrast, the differences between the performances of the backward plans are comparable. 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.

8.4 Vertex Columns for Single Cardinality Edges

In Section 8.2, we showed the memory gains of storing single cardinality edges in vertex columns. This storage also improves performance because the system can directly access the edge without an indirection through a CSR. We next demonstrate this benefit 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:

(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 8.2.

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

(iii) `V-COL-C`: Null compressed version of `V-COL-UNC`. This is equivalent to `+NULL` configuration in Section 8.2.

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

	1-hop	2-hop	3-hop	Mem (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**
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**

Table 4: Vertex property columns vs. 2-level CSR adjacency lists for storing single cardinality edges: Query runtime is in seconds and memory usage in MBs.

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, the 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.

Table 4 shows the result of queries under each system configuration. 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 8.2, 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, for the storage cost of specific edges, we observe major reductions. The last column of Table 4 reports the size of storing `replyOf` edges under each system configuration. In LDBC100, 50.5% of the `replyOf` forward adjacency lists are empty. Observe that NULL compressing these lists lead to 1.75x memory reduction when using vertex columns (from 839.93MB vs 478.86MB). We note that the reduction is 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.

8.5 Null Compression

We next demonstrate the memory/performance trade-off of our NULL compression scheme on sparse property columns. We use 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 [17]; and (ii) `Uncompressed` stores the column in an uncompressed format. In Appendix A, we demonstrate a sensitivity analysis for `J-NULL` running under different m and c values. This experiment demonstrates that read performance is insensitive to these

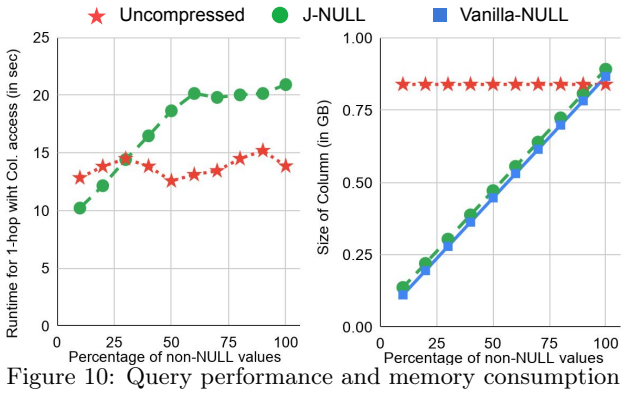

Figure 10: Query performance and memory consumption when storing a vertex property column as uncompressed, compressed with Jacobson's scheme, and the vanilla bit string scheme from Abadi, under different density levels.

parameters. The memory overhead increases as m increases, albeit marginally. So a reasonable choice is picking $m = c = 16$, which incurs 1 bit extra overhead per element for storing prefix sums.

Figure 10 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 `Vanilla-NULL`, 2 bits for each element instead of 1 bit. As expected the query performance of `J-NULL` 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 10. Interestingly, when the column is sparse enough (containing less than 30% non-NULL values), `J-NULL` can even outperform `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 iterators return a global NULL value which is likely to be in the CPU cache. Instead, `Uncompressed` always returns the value in the element's cell, which has a higher chance of a CPU cache miss.

8.6 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 [9] or MemGraph [6]. LBP 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 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.

We use LDBC100, Wikipedia, and Flickr datasets. In our first experiment, we take 1-, 2-, and 3-hop queries (as in Section 8.3, 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

			1-hop	2-hop	3-hop
LDBC100	FILTER	GF-CV	24.6	1470.5	40252.4
		GF-CL	7.7	116.2	2647.3
			3.2x	**12.7x**	**15.2x**
	COUNT(*)	GF-CV	13.4	241.9	6947.3
		GF-CL	4.2	18.9	357.9
			3.2x	**12.8x**	**19.4x**
FLICKR	FILTER	GF-CV	32.6	1300.0	14864.0
		GF-CL	12.2	95.3	1194.7
			2.7x	**13.7x**	**12.4x**
	COUNT(*)	GF-CV	35.3	519.2	4162.5
		GF-CL	16.9	23.4	51.7
			2.1x	**21.4x**	**80.6x**
WIKI	FILTER	GF-CV	35.8	4500.2	236930.2
		GF-CL	11.9	1192.5	20329.3
			2.9x	**3.8x**	**11.7x**
	COUNT(*)	GF-CV	32.7	1745.2	109000.2
		GF-CL	19.0	27.6	120.4
			1.7x	**63.2x**	**905.1x**

Table 5: Runtime (ms) of GF-RV and GF-CL (LBP) plans.

the date property of the last extended edge. A major part of the work in these plans happen at the final join and filter operation, therefore these plans allow us to measure the performance benefits of performing computations inside loops and avoiding data copying in joins. Our results are shown in the FILTER rows of Table 5. We see that GF-CL outperforms GF-CV by large margins, between 2.7x and 15.2x.

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(*). 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(*) rows of Table 5. Observe that the improvements are much more significant now, up to close to three orders of magnitude on Wiki (by 905.1x). The primary advantage of GF-CL is now that the counting happens on compressed intermediate results.

8.7 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, 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 is faster than GF-RV also on an independent end-to-end benchmark. We also aim to verify that GF-RV into which we integrated our techniques is already competitive with or outperforms other baseline systems on workloads that contain many-to-many joins. We used two dataset LDBC10's social network benchmark and JOB, both of which contain queries with many-to-many joins.

We used the the community version v4.2 of the Neo4j GDBMS [9], the community version 10.0 of Vertica [14] and MonetDB 5 server 11.37.11 [7]. We note that our experiments should not be interpreted as one system being generally more efficient than another. It is very difficult to meaningfully compare completely separate systems, e.g., all baseline systems have many tunable parameters, and some have more efficient enterprise versions. 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

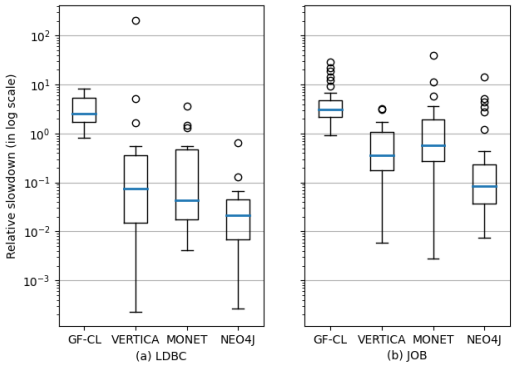

Figure 11: Relative speedup/slowdown of the different systems in comparison to GF-RV on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.

for Vertica and MonetDB, sorted by the source and destination vertexIDs, respectively. For GF-RV and GF-CL, we use the best left-deep plan we could manually pick. This plan was obvious in most cases. For example, the joins in LDBC path queries start from a particular vertex ID, so the best join orders start from that vertex and iteratively extend in the same direction. For Vertica, MonetDB, and Neo4j, we use the better of the systems' default plans and the left-deep that is equivalent to the one we use in GF-RV and GF-CL.

8.7.1 LDBC

We use the LDBC10 dataset. GraphflowDB is a prototype system and currently implements parts of the Cypher language that are relevant to our research, so lack several features that LDBC queries exercise. The system currently has support for select-project-join queries and a 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 [31] 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 exact queries are given in Appendix B.

Figure 11a shows the relative speedup/slowdown of the different systems in comparison to GF-RV. Tables 6a and 6b, show the individual runtime numbers of each IS and IC query, respectively. As we expect, GF-CL is broadly more performant than GF-RV on this benchmark with a median query improvement factor of 2.6x. 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 produce 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,

	IS01	IS02	IS03	IS04	IS05	IS06	IS07
GF-CL	2.7	3.0	2.2	36.9	40.6	69.3	38.3
GF-RV	2.2	3.9	3.9	307.3	236.6	423.0	307.9
	0.8x	**1.3x**	**1.8x**	**8.3**	**5.8x**	**6.1x**	**8.0x**
VERTICA	6.1	16728.2	7.0	1.5	45.2	259.2	24818.9
	2.2x	**5574.2x**	**3.2x**	**0.04x**	**1.1x**	**3.7x**	**647.7x**
MONET	112.3	282.2	8.3	84.1	516.4	323.0	206.3
	40.9x	**94.0x**	**3.8x**	**2.3x**	**12.7x**	**4.7x**	**5.4x**
NEO4J	103.1	117.4	86.1	12418.9	11665.9	67390.3	12095.2
	37.5x	**39.1x**	**39.1**	**336.6**	**287.4**	**972.3**	**315.6**

(a) LDBC IS Queries

	IC01	IC02	IC03	IC04	IC05	IC06	IC07	IC08	IC09	IC011	IC012
GF-CL	36.7	32.4	409.4	13.1	1565.2	113.0	3.0	2.6	1519.8	11.1	34.2
GF-RV	88.4	45.2	1521.8	57.3	8925.0	333.1	6.3	7.0	2098.1	19.2	84.9
	2.4x	**1.4x**	**3.7**	**4.4x**	**5.7x**	**3.0x**	**2.1x**	**2.7x**	**1.4x**	**1.7x**	**2.5x**
VERTICA	257.2	3063.8	18610.3	1711.6	59351.0	4715.7	4092.2	2837.2	17276.2	672.9	5028.1
	7.0x	**94.5x**	**45.5x**	**130.5x**	**37.9x**	**41.7x**	**1348.8x**	**1094.2x**	**11.4x**	**60.9x**	**147.1x**
MONET	160.3	323.2	187330.9	13955.1	165273.0	2783.1	206.3	920.5	121943.2	572.0	3251.9
	4.4x	**10.0x**	**457.6x**	**1064.3x**	**105.6x**	**24.6x**	**68.0x**	**354.8x**	**80.0x**	**51.7x**	**95.1x**
NEO4J	669.3	170722.9	86231.9	75254.9	*TLE*	515.4	95.6	108.3	219425.5	2804.1	34043.0
	18.3x	**5264.2x**	**210.6x**	**5739.4x**	**-**	**4.6x**	**31.5x**	**41.8x**	**144.4x**	**253.6x**	**996.0x**

(b) LDBC IC Queries

	1.a	2.a	3.a	4.a	5.a	6.a	7.a	8.a	9.a	10.a	11.a
GF-CL	13.5	59.7	41.4	-	124.8	9.8	53.4	298.3	209.7	51.3	23.6
GF-RV	50.3	120.4	38.0	-	460.9	91.4	77.5	1245.6	560.2	110.4	487.3
	3.7x	**2.0x**	**0.9x**	**-**	**3.7x**	**9.3x**	**1.5x**	**4.2x**	**2.7x**	**2.2x**	**2.1x**
VERTICA	214.8	329.8	3928.3	798.1	2591.4	495.2	72.8	1166.2	442.5	395.4	136.9
	15.9x	**5.5x**	**94.9x**	**-**	**20.8x**	**50.6x**	**1.4x**	**3.9x**	**2.1x**	**7.7x**	**5.8x**
MONET	36.2	33.0	36.2	120.2	232.5	1428.1	133.6	112.5	282.5	304.1	92.8
	2.7x	**0.6x**	**0.9x**	**-**	**1.9x**	**145.9x**	**2.5x**	**0.4x**	**1.4x**	**5.9x**	**3.9x**
NEO4J	3077.7	895.3	774.3	203.4	10727.2	206.7	6497.6	3451.7	14946.4	1480.7	2332.6
	227.5x	**15.0x**	**18.7x**	**-**	**85.9x**	**21.1x**	**121.8x**	**11.6x**	**71.3x**	**28.9x**	**98.8x**

	12.a	13.a	14.a	15.a	16.a	17.a	18.a	19.a	20.a	21.a	22.a
GF-CL	-	70.0	14.6	362.5	15.0	268.5	548.1	207.8	12.8	13.2	-
GF-RV	-	406.9	33.3	6772.3	34.0	594.6	1700.9	983.0	208.5	22.6	-
	-	**5.8x**	**2.3x**	**18.7x**	**2.3x**	**2.2x**	**3.1x**	**4.7x**	**1.6x**	**1.7x**	**-**
VERTICA	870.2	286.3	28.2	2100.5	1028.8	2538.5	1686.0	4777.2	982.5	34.0	99.0
	-	**4.1x**	**1.9x**	**5.8x**	**68.5x**	**9.5x**	**3.1x**	**23.0x**	**76.7x**	**2.6x**	**-**
MONET	56.7	1148.2	83.4	172.0	224.5	1304.3	868.2	644.0	7552.3	60.7	140.2
	-	**16.4x**	**5.7x**	**0.5x**	**14.9x**	**4.9x**	**1.6x**	**3.1x**	**590.0x**	**4.6x**	**-**
NEO4J	5079.1	93.8	291.9	2437.4	4526.6	167.6	1414.8	12047.2	1849.0	272.4	317.8
	-	**1.3x**	**20.1x**	**6.7x**	**301.2x**	**0.6x**	**2.6x**	**58.0x**	**144.5x**	**20.7x**	**-**

	23.a	24.a	25.a	26.a	27.a	28.a	29.a	30.a	31.a	32.a	33.a
GF-CL	14.5	10.8	107.7	10.5	10.3	-	5.6	18.3	112.5	10.0	52.3
GF-RV	407.8	47.9	1527.8	19.9	125.3	-	18.5	52.7	775.9	24.1	201.6
	28.8x	**4.4x**	**14.2x**	**1.9x**	**12.2x**	**-**	**3.1**	**2.9x**	**6.9x**	**2.4x**	**3.9x**
VERTICA	698.5	518.1	496.2	1239.9	231.0	197.3	3153.1	152.6	2696.3	193.6	125.3
	49.4x	**47.8x**	**4.6x**	**118.7x**	**22.5x**	**-**	**529.9x**	**8.5x**	**23.9x**	**19.3x**	**2.4x**
MONET	124.2	993.9	784.8	1736.1	75.9	323.8	1012.3	1940.2	848.1	87.7	88.1
	8.8x	**91.8x**	**7.3x**	**166.1x**	**7.4x**	**-**	**170.1x**	**107.5x**	**7.5x**	**8.8x**	**1.7x**
NEO4J	2497.1	3505.4	108.6	694.1	1276.7	1573.7	648.2	326.1	152.7	364.1	2723.8
	176.5x	**323.7x**	**1.0x**	**66.3x**	**124.4x**	**-**	**108.9x**	**18.1x**	**1.4x**	**36.4x**	**52.0x**

(c) JOB Benchmark

Table 6: Runtime in ms for running the LDBC Interactive Complex Reads (IC) and Interactive Short Reads (IS) queries and JOB Benchmark on 5 systems: (i) GF-CL (ii) GF-RV (iii) VERTICA (iv) MONET and (v) NEO4J.

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 13.1x, 22.8x, and 46.1x 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 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.

8.7.2 JOB

JOB queries come in four variants and we used their first variant. We converted the JOB queries to their Cypher equivalent following our conversion of the dataset. Many of the projection parts of JOB queries perform aggregations on strings, such as min(name), where name is a string column. Graphflow supports aggregations only on numeric types, so we removed these aggregations. Our final queries are in Appendix C.

Figure 11b shows the relative speedup/slowdown of the different systems in comparison to GF-RV. Table 6b shows individual runtime numbers of each query. Similar to our LDBC results, we see GF-CL to improve the performance, now by a median factor of 3.1x. Again similar to LDBC, with the exception of one query, we see consistent speed ups across all queries between 1.5x and 28.8x. Different from LDBC, we also see queries on which the improvement factors are much larger, such as 21.1x and 28.8x. In LDBC, the largest improvement factor was 8.3x. This is to be expected because most of the queries in JOB perform star joins while LDBC queries contained path queries that start from a node with a selective filter. On path queries, our plans start from a single node and extend in one direction, in which case only the last extension can truly be factorized, so be in unflat form. This is because each ListExtend that we use first flattens the previously extended node. In contrast on star queries, multiple extensions from the center node can all remain in unflat format. Therefore GF-CL's plans can benefit more from LBP because they can compress their intermediate tuples more. We also see that similar to LDBC, GF-RV is more performant than the columnar RDBMSs. However, compared to LDBC, these systems are now more competitive. We noticed that one reason for this is that on star queries, these systems's default plans are often bushy plans (27 out of 33 for MonetDB and 26 out of 33 for Vertica), which produce fewer intermediate tuples than GF-RV, which does not benefit from factorization and uses left-deep plans. So these systems now benefiting from bushy plans which they did not in LDBC. In contrast, on LDBC these systems would also primarily use left-deep plans (only 2 out of 18 for MonetDB and 4 out of 18 for Vertica were bushy) because on these path queries, it is better to start from a single highly filtered node table and join iteratively in a left-deep plan to match the entire path. Finally, similar to LDBC, we found Neo4j to be the least competitive of these baselines.

9. RELATED WORK

Column stores [41, 63, 69, 70] 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 [18]. This paper studies how to integrate some of these techniques into in-memory GDBMSs.

Existing GDBMSs and RDF systems adopt a columnar structure often for storing the topologies of graphs. This is done either by using a variant of vanilla adjacency list format or CSR. Instead, systems often use row-oriented structures to store properties, such as an interpreted attribute layout [24]. For example, Neo4j [9] 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. Similarly, the properties of each vertex and edge are stored in a linked-list, where each property record points to the next and encodes the key, data type, and value of the property. Similarly, JanusGraph [5] stores edges in adjacency lists partitioned by edge labels and properties as consecutive key-value pairs (so in row-oriented format). 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 compres edge and vertex IDs and NULLs.

There are also several GDBMSs that are developed directly on top of an RDBMS or another database system [12], such as IBM Db2 Graph [64], Oracle Spatial and Graph [12] SAP's graph database [60]. These systems can benefit from the columnar techniques in the underlying RDBMS, which are however not optimized for graph storage and queries. 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.

GQ-Fast [50] 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 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).

ZipG [**?**] 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 [**?**]. 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. Succinct compresses these files using suffix arrays and several secondary indices. Although the authors report good compression rates, 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 [21] 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 [65] 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 [57] 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. We note that similar to our Guideline 3, reference [56] has also observed that graph data depicts structure, and certain predicates in RDF databases co-exist together in a node. This is similar to the property co-occurrence structure we exploit, and is exploited in the RDF 3-X system for better cardinality estimation.

Several 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 [68], Aspen [30], and LLAMA [51]. We focus on a read-optimized system setting and use 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 processors, 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. We have not studied how to integrate more advanced factorized processing techniques inside GDBMS within the scope of this paper.

10. 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.

11. ACKNOWLEDGEMENTS

This work was supported in part by an NSERC Discovery grant. We thank Lori Paniak for promptly assisting with many system issues. We also thank Xiyang Feng and Guodong Jin for helping with baseline experiments and Snehal Mishra for helping with illustrations.

12. REFERENCES

[1] Amazon Neptune. https://aws.amazon.com/neptune/, 2020.
[2] DGraph Github Repository. https://github.com/dgraph-io/dgraph, 2020.
[3] DuckDB. https://duckdb.org/, 2020.
[4] Graphflowdb Source Code. https://tinyurl.com/pktcna6j, 2020.
[5] Janus Graph. https://janusgraph.org, 2020.
[6] Memgraph. https://memgraph.com/, 2020.
[7] MonetDB source code, (Jun2020-SP1). https://github.com/MonetDB/MonetDB/releases/tag/Jun2020_SP1_release, 2020.
[8] Neo4j Blog on Deletions. https://tinyurl.com/4wf9593y, 2020.
[9] Neo4j Community Edition. https://neo4j.com/download-center/#community, 2020.
[10] Neo4j Property Graph Model. https://neo4j.com/developer/graph-database, 2020.
[11] Oracle In-Memory Column Store Architecture. https://tinyurl.com/vkvb6p6, 2020.
[12] Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html, 2020.
[13] TigerGraph. https://www.tigergraph.com, 2020.
[14] Vertica 10.0.x Documentation. https://www.vertica.com/docs/10.0.x/HTML/Content/Home.html, 2020.
[15] GraphflowDB Columnar Techniques. https://github.com/g31pranjal/graphflow-columnar-techniques, 2021.
[16] Wikipedia Dynamic (de), (Konect). http://konect.cc/networks/link-dynamic-dewiki/, 2021.
[17] D. Abadi. Column Stores for Wide and Sparse Data. *CIDR*, 2007.
[18] 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.

[19] D. Abadi, S. Madden, and M. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. *SIGMOD*, 2006.

[20] D. J. Abadi, S. R. Madden, and N. Hachem. Column-Stores Vs. Row-Stores: How Different Are They Really? *SIGMOD*, 2008.

[21] D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. *PVLDB*, 2007.

[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] 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.

[32] 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.

[33] M. Freitag, M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. Adopting worst-case optimal joins in relational database systems. In *VLDB*, 2021.

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

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

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

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

[38] P. Gupta, A. Mhedhbi, and S. Salihoglu. Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems. Technical report, March 2021 https://github.com/g31pranjal/graphflow-columnar-techniques/blob/master/paper.pdf.

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

[40] S. Héman, M. Zukowski, N. J. Nes, L. Sidirourgos, and P. Boncz. Positional update handling in column stores. In *SIGMOD*, 2010.

[41] 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.

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

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

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

[45] J. Kunegis. Konect: The koblenz network collection. *WWW*, 2013.

[46] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[47] V. Leis, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. How good are query optimizers, really? 9(3), 2015.

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

[49] J. Leskovec, J. Kleinberg, and C. Faloutsos. Graphs over time: Densification laws, shrinking diameters and possible explanations. In *KDD*, 2005.

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

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

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

[53] A. Mislove, H. S. Koppula, K. P. Gummadi, P. Druschel, and B. Bhattacharjee. Growth of the flickr social network. *WOSN*, 2008.

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

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

[56] T. Neumann and G. Moerkotte. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *ICDE*, 2011.

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

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

[59] D. Olteanu and J. Závodný. Size bounds for factorised representations of query results. *TODS*, 2015.

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

[61] 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*.

[62] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. The ubiquity of large graphs and surprising challenges of graph processing. In *VLDB*, 2017.

[63] 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.

[64] 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.

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

[66] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Rec.*, 2000.

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

[68] 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.

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

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

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

APPENDIX

A. SENSITIVITY ANALYSES

Two of our data structures have parameters that can be modified. First is the parameter k, which groups the properties of k many adjacency lists into a single property page. The other are the m and c parameters in our Jacobson's index-based NULL compression schemes. In this section, we do a sensitivity analysis on these parameters to demonstrate their effects.

A.1 Parameter k

In this experiment, we extend our experiment from Section 8.3, where we ran 1- and 2-hop queries with a predicate on the edges on `LDBC100`, `WIKI`, and `FLICKR`, comparing property pages with $k = 128$ parameter and pure edge columns. Note the pure edge columns store all of the edges in a single column, so is equivalent to setting $k = \infty$. So we repeat the same experiment in Table 3 with $k = 2^i$ for $i = 1, ..., 17$. As before, we measure the runtime performance of the query, as a proxy for the efficiency of accessing the properties, which is the main computation done in these queries. Our results are shown in Figure 12. The last x-axis value "*" uses pure edge columns (so corresponds to the COL_E values in Table 3). Note that for most settings (except LDBC 2H figure) using up to $k = 2^9$ yields relatively stable results, after which the performance degrades. This threshold value is a bit larger, up to $k = 2^{11}$ on `Flickr`, which has a lower average degree than `LDBC` and `Wiki` (14 vs 44 and 41, respectively). This is expected because in this experiment, we read all of the edge properties in the order of the adjacency lists. We expect there to be some threshold property page block size B, under which we expect to get good cache locality. If the sizes of the property pages get larger than B we expect to lose locality and the performance to degrade. Therefore the smaller the average adjacency list sizes, the larger number of adjacency lists we can pack together into B to still get good cache locality. In light of this analysis, our choice of $k = 2^7 = 128$ is in the safe region in all of these settings. Recall also that, although we do not focus on updates in this paper, making k very small, say 2 or 4, is not a good choice, as it would make recycling deleted positional offsets, which necessarily leave gaps, difficult.

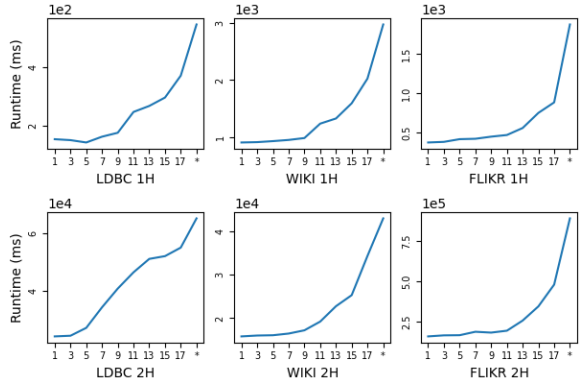

Figure 12: Sensitivity analysis of k. Values of X-axis are exponents of 2 which denotes parameter k of the property pages. '*' denotes Edge Column which is essentially a property page with k = ∞

A.2 Parameters m and c

We next analyze the effects of m and c in our NULL compression scheme that extends Abadi's bitstring-based scheme with Jacobson's index. Recall that c determines the size of the prepopulated map to compute ranks. However in practice this can be 8 or 16 because while at $c = 8$ and $c = 16$ the map sizes are at 2KB and 1MB, at $c = 24$ and $c = 32$ the size quickly becomes very large: 1.6GB and 546GB. Therefore, we consider only the MB level sizes of $c = 8$ and $c = 16$. Recall that m is the number of bytes we use to store the prefix sums. This parameter can take on a larger range of values, such as 8, 16, 24, 32 or even larger. We expect m to primarily affect the memory overhead of our scheme and not the decompression time. Specifically our memory overhead per element compressed is exactly m/c bits. Therefore, for any c value, there is value in picking a small m.

To see if there is any visible decompression time differences between $c = 8$ and $c = 16$ and to verify that different m values do not have any effect on performance, we reran the experiment from Section 8.5 with different (c,m) combinations, where we varied m to be 8, 16, 24, or 32. As before we measure the runtime of the MATCH (a:`Person`)−[e:`Likes`]→ (b:`Comment`) RETURN b.`creationDate` query where `creationDate` property contains different percentage of NULL values. Our runtime results are shown in Table 7. In the table, ρ is the percentage of non-NULL values in the `creationDate` property. As we expect, we do not see any visible sensitivity to m, at any non-NULL percentage value ρ. Similarly we also do not see any visible sensitivity between $c = 8$ vs $c = 16$.

Our memory numbers are in Table 8. We report only our overheads due to storing the prefix sums and the bitmaps and not the non-NULL values. Since these overheads are fixed for any ρ value, this table has a single row (measured at ρ=50%). As we expect the measured overheads are determined by the m/c value. For example, (8, 8) and (16, 16) have almost identical measured overheads, 52.7MB and 52.8MB, respectively. We think (8, 8), (16, 16), or (16, 8) are reasonable choices in practice for (c, m) as they achieve the lowest overheads.

ρ	8,8	8,16	8,24	8,32	16,8	16,16	16,24	16,32
100	21.0	20.9	20.2	20.3	20.1	20.6	20.1	20.8
90	20.9	20.2	20.5	20.0	20.0	20.5	20.7	20.9
80	20.3	20.1	20.2	20.1	20.1	21.0	20.9	20.4
70	19.0	19.8	19.8	20.0	19.8	19.7	20.4	20.3
60	19.0	19.0	20.2	19.2	19.0	19.1	19.6	19.2
50	18.2	18.7	18.6	18.4	18.8	18.1	18.5	18.8
40	17.2	16.5	16.3	16.1	16.7	17.7	17.9	16.9
30	16.9	14.4	14.6	14.4	14.8	15.9	15.6	15.6
20	13.7	12.2	12.1	12.2	12.2	13.6	13.9	13.0
10	10.9	10.2	10.2	10.3	10.3	11.0	10.4	10.8

Table 7: Sensitivity analysis of (c,m): runtime in ms. ρ is the percetage of non-NULL values in the `creationDate` property.

8,8	8,16	8,24	8,32	16,8	16,16	16,24	16,32
52.8	78.8	105.2	132.6	39.6	52.7	65.2	80.1

Table 8: Sensitivity analysis of (c,m): overhead (in MB) of storing null bits and prefixSums for different configurations of c and m values.

B. MODIFIED LDBC SNB QUERIES

1. *IS01*

```
MATCH   (p:Person)−[:IsLocatedIn]→(pl:Place)
WHERE   p.id = 22468883
RETURN p.fName, p.lName, p.birthday, p.locationIP,
p.browserUsed, p.gender, p.creationDate, pl.id;
```

2. *IS02*

```
MATCH  (p:Person)←[:hasCreator]−(c:Comment)
       (c:Comment)−[:replyOf]→(post:Post)
       (post:Post)−[:hasCreator]→(op:Person)
WHERE  p.id = 22468883
RETURN c.id, c.content, c.creationDate, op.id,
       op.fName, op.lName;
```

3. *IS03*

```
MATCH  (p:Person)−[k:knows]→(friend:Person)
WHERE  p.id = 22468883
RETURN friend.id, friend.fName, friend.lName,
       e.date;
```

4. *IS04*

```
MATCH  (comment:Comment)
WHERE  comment.id = 0
RETURN comment.creationDate, comment.content;
```

5. *IS05*

```
MATCH  (comment:Comment)−[:hasCreator]→(p:Person)
WHERE  comment.id = 0
RETURN p.id, p.fName, p.lName;
```

6. *IS06*

```
MATCH  (comment:Comment)−[:replyOf]→(pst:Post)
       (pst:Post)←[:containerOf]−(f:Forum)
       (f:Forum)−[:hasModerator]→(p:Person)
WHERE  comment.id = 0
RETURN f.id, f.title, p.id, p.fName, p.lName;
```

7. *IS07*

```
MATCH  (mAuth:Person)←[:hasCreator]−(cmt0:Comment)
       (cmt0:Comment)←[:replyOf]−(cmt1:Comment)
       (cmt1:Comment)−[:hasCreator]→(rAuth:Person)
WHERE  comment.id = 6
RETURN cmt1.id, cmt1.content, cmt1.creationDate,
       rAuth.id, rAuth.fName, rAuth.lName;
```

8. *IC01*

```
MATCH  (person:Person)−[:knows]→(p1:Person)
       (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)−[:knows]→(op:Person)
       (op:Person)−[:isLocatedIn]→(pl:Place)
WHERE  person.id = 22468883
RETURN op.id, op.lName, op.birthday, op.creationDate,
       op.gender, op.locationIP, city.name;
```

9. *IC02*

```
MATCH  (p:Person)−[:knows]→(frnd:Person)
       (frnd:Person)←[:hasCreator]−(msg:Comment)
WHERE  p.id = 22468883 AND
       msg.creationDate < 1342805711
RETURN frnd.id, frnd.fName, frnd.lName, msg.id,
       msg.content,msg.creationDate;
```

10. *IC03*

```
MATCH  (person:Person)−[:knows]→(p1:Person)
       (p1:Person)−[:knows]→(op:Person)
       (op:Person)−[:isLocatedIn]→(pl:Place)
       (op:Person)←[:hasCreator]−(mx:Comment)
       (mx:Comment)−[:isLocatedIn]→(px:Place)
       (op:Person)←[:hasCreator]−(my:Comment)
       (my:Comment)−[:isLocatedIn]→(py:Place)
WHERE  person.id = 22468883 AND
       mx.creationDate >= 1313591219 AND
       mx.creationDate <= 1513591219 AND
       my.creationDate >= 1313591219 AND
       my.creationDate <= 1513591219 AND
       cx.name = 'India' AND cy.name = 'China'
RETURN cmt1.id, cmt1.content, cmt1.creationDate,
       rAuth.id, rAuth.fName, rAuth.lName;
```

11. *IC04*

```
MATCH  (:Person)←[:knows]−(p:Person)
       (p:Person)−[:knows]→(frnd:Person)
       (frnd:Person)←[:hasCreator]−(pst:Post)
       (pst:Post)−[:hasTag]→(t:Tag)
WHERE  p.id = 22468883 AND
       post.creationDate >= 1313591219 AND
       post.creationDate <= 1513591219
RETURN cmt1.id, cmt1.content, cmt1.creationDate,
       rAuth.id, rAuth.fName, rAuth.lName;
```

12. *IC05*

```
MATCH  (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)−[:knows]→(p3:Person)
       (p3:Person)←[hm:hasMember]−(f:Forum)
       (f:Forum)−[:containerOf]→(pst:Post)
WHERE  p1.id = 22468883 AND hm.date > 1267302820
RETURN f.title;
```

13. *IC06*

```
MATCH  (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)−[:knows]→(p3:Person)
       (p3:Person)←[:hasCreator]−(pst:Post)
       (pst:Post)−[:hasTag]→(t1:Tag)
       (pst:Post)−[:hasTag]→(t2:Tag)
WHERE  p1.id = 22468883 AND
       t1.name = 'Rumi' AND t2.name <> 'Rumi'
RETURN t2.name;
```

14. *IC07*

```
MATCH  (p:Person)←[:hasCreator]−(cmt:Comment)
       (cmt:Comment)←[l:likes]−(frnd:Person)
WHERE  p.id = 22468883
RETURN frnd.id, frnd.fName, frnd.lName, l.date,
       cmt.content;
```

15. *IC08*

```
MATCH  (p:Person)←[:hasCreator]−(pst:Post)
       (pst:Post)←[:replyOf]−(cmt:Comment)
       (cmt:Comment)−[:hasCreator]→(cmtAuth:Person)
WHERE  p1.id = 22468883
RETURN cmtAuth.id, cmtAuth.fName, cmtAuth.lName,
       cmt.creationDate, cmt.id, cmt.content;
```

16. *IC09*

```
MATCH  (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)−[:knows]→(p3:Person)
       (p3:Person)←[:hasCreator]−(cmt:Comment)
WHERE  person.id = 22468883 AND
       cmt.creationDate < 1342840042
RETURN p3.id, p3.fName, p3.lName, cmt.id, cmt.content,
       cmt.creationDate;
```

17. *IC11*

```
MATCH  (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)−[:knows]→(p3:Person)
       (p3:Person)−[w:workAt]→(org:Organization)
       (org:Organization)−[:isLocatedIn]→(pl:Place)
WHERE  p1.id = 22468883 AND w.year < 2016 AND
       pl.name = 'China'
RETURN p3.id, p3.fName, p3.lName, org.name;
```

18. *IC12*

```
MATCH  (p1:Person)−[:knows]→(p2:Person)
       (p2:Person)←[:hasCreator]−(cmt:Comment)
       (cmt:Comment)−[:replyOf]−(pst:Post)
       (pst:Post)−[:hasTag]→(t:Tag)
       (t:Tag)−[:hasType]→(tc:TagClass)
       (tc:TagClass)−[:isSubclassOf]→(:TagClass)
WHERE  p1.id = 22468883 AND tc.name='Person'
RETURN p2.id, p2.fName, p2.lName;
```

C. MODIFIED JOB QUERIES

1. *1A*

```
MATCH  (t:title)−[mc:movie_companies]→(:company_name),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2)
WHERE  mc.company_type = 'production company' AND
       mii.info_type = 'top 250 rank' AND
       mc.note CONTAINS '(co-production)'
RETURN COUNT(*);
```

2. *2A*

```
MATCH  (t:title)−[:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  k.keyword = 'character-name-in-title' AND
       cn.country_code = '[de]'
RETURN COUNT(*);
```

3. *3A*

```
MATCH  (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:has_movie_info]→(mi:movie_info)
WHERE  k.keyword CONTAINS 'sequel' AND
       mi.info = 'Sweden' AND t.production_year > 2005
RETURN COUNT(*);
```

4. *4A*

```
MATCH  (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2)
WHERE  k.keyword CONTAINS 'sequel' AND
       mii.info_type = 'rating' AND mii.info > '5.0' AND
       t.production_year > 2005
RETURN COUNT(*);
```

5. *5A*

```
MATCH  (t:title)−[mc:movie_companies]→(:company_name),
       (t:title)−[:has_movie_info]→(:movie_info)
WHERE  mc.company_type = 'production company' AND
       mc.note CONTAINS '(theatrical)' AND
       mc.note CONTAINS '(France)' AND
       t.production_year > 2005
RETURN COUNT(*);
```

6. *6A*

```
MATCH  (t:title)−[:cast_info]→(n:name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  k.keyword = 'marvel-cinematic-universe' AND
       n.name CONTAINS 'Downey' AND
       t.production_year > 2010
RETURN COUNT(*);
```

7. *7A*

```
MATCH  (t:title)−[ml:movie_link]→(:title),
       (t:title)−[:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(an:aka_name),
       (n:name)−[:has_person_info]→(pi:person_info)
WHERE  pi.info_type = 'mini biography' AND
       an.name CONTAINS 'a' AND n.name_pcode_cf >= 'A' AND
       n.name_pcode_cf <= 'F' AND n.gender = 'm' AND
       pi.note = 'Volker Boehm' AND
       t.production_year >= 1980 AND
       t.production_year <= 1995 AND
       ml.link_type = 'features'
RETURN COUNT(*);
```

8. *8A*

```
MATCH  (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[ci:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(:aka_name)
WHERE  ci.note ='(voice: English version)' AND
       cn.country_code ='[jp]' AND mcs.role = 'actress' AND
       mc.note CONTAINS '(Japan)' AND n.name CONTAINS 'Yo'
RETURN COUNT(*);
```

9. *9A*

```
MATCH  (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[ci:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(:aka_name)
WHERE  cn.country_code = '[us]' AND ci.role = 'actress' AND
       n.gender = 'f' AND mc.note CONTAINS '(USA)' AND
       ci.note STARTS WITH '(voice' AND
       n.name CONTAINS 'Ang' AND
       t.production_year >= 2005 AND
       t.production_year <= 2015
RETURN COUNT(*);
```

10. *10A*

```
MATCH  (t:title)−[:movie_companies]→(cn:company_name),
       (t:title)−[ci:cast_info]→(:name)
WHERE  ci.note CONTAINS '(voice)' AND ci.role = 'actor' AND
       ci.note CONTAINS '(uncredited)' AND
       cn.country_code = '[ru]' AND
       t.production_year > 2005
RETURN COUNT(*);
```

11. *11A*

```
MATCH  (t:title)−[ml:movie_link]→(:title),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  cn.country_code <> '[pl]' AND
       cn.name CONTAINS 'Film' AND k.keyword ='sequel' AND
       mc.company_type = 'production company' AND
       ml.link_type IN ('follows', 'followedBy') AND
       ml.link_type LIKE 'follow' AND
       t.production_year > 1950 AND
       t.production_year < 2000
RETURN COUNT(*);
```

12. *12A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2)
WHERE  cn.country_code = '[us]' AND
       mi.info_type = 'genres' AND
       mii.info_type = 'rating' AND
       mc.company_type = 'production company' AND
       mi.info = 'Drama' AND t.production_year >= 2005 AND
       t.production_year <= 2008 mii.info > '8.0' AND
RETURN COUNT(*);
```

13. *13A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2)
WHERE  cn.country_code = '[de]' AND t.kind = 'movie' AND
       mc.company_type = 'production company' AND
       mi.info_type = 'release dates' AND
       mii.info_type = 'rating'
RETURN COUNT(*);
```

14. *14A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2)
WHERE  mi.info_type = 'countries' AND
       k.keyword = 'murder' AND t.kind = 'movie' AND
       mi.info = 'USA' AND mii.info_type = 'rating' AND
       mii.info < '8.5' AND t.production_year > 2010
RETURN COUNT(*);
```

15. *15A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(:keyword)
WHERE  cn.country_code = '[us]' AND
       mc.note CONTAINS '(worldwide)' AND
       mi.info STARTS WITH 'USA:' AND
       mi.note CONTAINS 'internet' AND
       mi.info_type = 'release dates' AND
       mc.note CONTAINS '(200' AND t.production_year > 2000
RETURN COUNT(*);
```

16. *16A*

```
MATCH  (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:movie_companies]→(cn:company_name),
       (t:title)−[:cast_info]→(:name),
       (n:name)−[:has_aka_name]→(:aka_name)
WHERE  cn.country_code = '[us]' AND
       k.keyword = 'character-name-in-title' AND
       t.episode_nr >= 50 AND t.episode_nr < 100
RETURN COUNT(*);
```

17. *17A*

```
MATCH  (t:title)−[:cast_info]→(n:name),
       (t:title)−[:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  cn.country_code ='[us]' AND
       n.name STARTS WITH 'B' AND
       k.keyword ='character-name-in-title'
RETURN COUNT(*);
```

18. *18A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:cast_info]→(n:name)
WHERE  mi.info_type = 'budget' AND
       n.name CONTAINS 'Tim' AND
       mii.info_type = 'votes' AND n.gender = 'm'
RETURN COUNT(*);
```

19. *19A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[ci:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(:aka_name)
WHERE  cn.country_code = '[us]' AND n.gender = 'f' AND
       mi.info_type = 'release dates' AND
       ci.role = 'actress' AND
       t.production_year <= 2009 AND
       ci.note STARTS WITH '(voice' AND
       mc.note CONTAINS '(USA)' AND
       n.name CONTAINS 'Ang' AND
       mi.info STARTS WITH 'Japan:' AND
       t.production_year >= 2005 AND
RETURN COUNT(*);
```

20. *20A*

```
MATCH  (t:title)−[mk:movie_keyword]→(k:keyword),
       (t:title)−[:has_complete_cast]→(cc:complete_cast),
       (t:title)−[ci:cast_info]→(n:name)
WHERE  t.kind = 'movie' AND cc.subject = 'cast' AND
       cc.status IN ('complete', 'complete+verified') AND
       k.keyword = 'superhero' AND
       t.production_year > 1950 AND
       ci.name CONTAINS 'Tony' AND ci.name CONTAINS 'Stark'
RETURN COUNT(*);
```

21. *21A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[mk:movie_keyword]→(k:keyword),
       (t:title)−[ml:movie_link]→(:title)
WHERE  cn.country_code <> '[pl]' AND
       mi.info = 'Germany' AND
       c.name CONTAINS 'Film' AND
       mc.company_type = 'production company' AND
       k.keyword CONTAINS 'sequel' AND
       ml.link_type IN ('follows', 'followedBy') AND
       t.production_year >= 1950 AND
       t.production_year <= 2000
RETURN COUNT(*);
```

22. *22A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  cn.country_code <> '[us]' AND
       mi.info_type = 'countries' AND
       mii.info_type = 'rating' AND
       k.keyword = 'murder' AND mi.info = 'USA' AND
       t.kind = 'movie' AND mc.note CONTAINS '(200' AND
       mii.info < '7.0' AND t.production_year > 2008
RETURN COUNT(*);
```

23. *23A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:has_complete_cast]→(cc:complete_cast)
WHERE  cn.country_code = '[us]' AND
       cc.status = 'complete+verified' AND
       mi.info_type = 'release dates' AND
       mi.note CONTAINS 'internet' AND
       t.kind = 'movie' AND mi.info STARTS WITH 'USA:' AND
       t.production_year > 2000
RETURN COUNT(*);
```

24. *24A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[ci:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(:aka_name),
       (t:title)−[:movie_keyword]→(k:keyword)
WHERE  ci.note STARTS WITH '(voice:' AND
       cn.country_code = '[us]' AND k.keyword = 'hero' AND
       mi.info_type = 'release dates' AND
       mi.info STARTS WITH 'USA:' AND n.gender = 'f'  AND
       ci.role = 'actress' AND t.production_year > 2010
RETURN COUNT(*);
```

25. *25A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:cast_info]→(n:name)
WHERE  mi.info_type = 'genres' AND n.gender = 'm' AND
       mii.info_type = 'votes' AND
       k.keyword = 'murder' AND mi.info = 'Horror' AND
RETURN COUNT(*);
```

26. *26A*

```
MATCH  (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[ci:cast_info]→(:name),
       (t:title)−[:has_complete_cast]→(cc:complete_cast)
WHERE  cc.subject = 'cast' AND ci.name CONTAINS 'man' AND
       cc.status IN ('complete', 'complete+verified') AND
       mii.info_type = 'rating' AND mii.info > '7.0' AND
       k.keyword = 'superhero' AND
       t.kind = 'movie' AND t.production_year > 2000
RETURN COUNT(*);
```

27. *27A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[ml:movie_link]→(:title),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:has_complete_cast]→(cc:complete_cast)
WHERE  cc.subject IN ('cast', 'crew') AND
       cc.status = 'complete' AND k.keyword ='sequel' AND
       cn.name CONTAINS 'Film' AND
       cn.country_code <> '[pl]' AND
       mc.company_type = 'production company' AND
       t.production_year >= 1950 AND
       ml.link_type IN ('follows', 'followedBy') AND
       mi.info = 'Sweden' AND t.production_year <= 2000
RETURN COUNT(*);;
```

28. *28A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[mc:movie_companies]→(cn:company_name),
       (t:title)−[:has_complete_cast]→(cc:complete_cast)
WHERE  cc.subject = 'crew' AND mc.note CONTAINS '(200' AND
       cc.status <> 'complete+verified' AND
       cn.country_code <> '[us]' AND t.kind = 'movie' AND
       mi.info_type = 'countries' AND mii.info < '8.5' AND
       mii.info_type = 'rating' AND
       k.keyword = 'murder' AND mi.info = 'Germany' AND
       t.production_year > 2000
RETURN COUNT(*);
```

29. *29A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:has_complete_cast]→(cc:complete_cast),
       (t:title)−[ci:cast_info]→(n:name),
       (n:name)−[:has_aka_name]→(:aka_name),
       (n:name)−[:has_person_info]→(pi:person_info),
       (t:title)−[mc:movie_companies]→(cn:company_name)
WHERE  cc.subject = 'crew' AND cn.country_code = '[us]' AND
       cc.status = 'complete+verified' AND
       ci.name = 'Queen' AND ci.note CONTAINS '(voice' AND
       mi.info_type = 'release dates' AND
       pi.info_type = 'trivia' AND
       k.keyword = 'computer-animation' AND
       mi.info STARTS WITH 'Japan:' AND
       n.gender = 'f' AND t.production_year >= 2000 AND
       n.name CONTAINS 'An' AND
       ci.role = 'actress' AND
       t.title = 'Shrek 2' AND
       t.production_year <= 2010
RETURN COUNT(*);
```

30. *30A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[ci:cast_info]→(n:name),
       (t:title)−[:has_complete_cast]→(cc:complete_cast)
WHERE  cc.subject IN ('cast', 'crew') AND
       cc.status = 'complete+verified' AND
       mi.info_type = 'genres' AND
       mii.info_type = 'votes' AND
       k.keyword = 'murder' AND mi.info = 'Horror' AND
       n.gender = 'm' AND t.production_year > 2000
RETURN COUNT(*);;
```

31. *31A*

```
MATCH  (t:title)−[:has_movie_info]→(mi:movie_info),
       (t:title)−[:has_mov_info_2]→(mii:mov_info_2),
       (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[ci:cast_info]→(n:name),
       (t:title)−[mc:movie_companies]→(cn:company_name)
WHERE  mi.info_type = 'genres' AND n.gender = 'm' AND
       mii.info_type = 'votes' AND
       k.keyword = 'murder' AND mi.info = 'Horror'
RETURN COUNT(*);
```

32. *32A*

```
MATCH  (t:title)−[:movie_keyword]→(k:keyword),
       (t:title)−[:movie_link]→(:title)
WHERE  k.keyword ='character-name-in-title'
RETURN COUNT(*);
```

33. *33A*

```
MATCH  (t1:title)−[ml:movie_link]→(t2:title),
       (t1:title)−[:has_mov_info_2]→(mii1:mov_info_2),
       (t2:title)−[:has_mov_info_2]→(mii2:mov_info_2),
       (t1:title)−[mc1:movie_companies]→(cn1:company_name),
       (t2:title)−[mc2:movie_companies]→(cn2:company_name)
WHERE  cn1.country_code = '[us]' AND
       mii1.info_type = 'rating' AND
       mii2.info_type = 'rating' AND
       t1.kind = 'tv series' AND
       t2.kind = 'tv series' AND
       ml.link_type IN ('follows', 'followedBy') AND
       mii2.info < '3.0' AND t2.production_year >= 2005 AND
       t2.production_year <= 2008
RETURN COUNT(*);
```