# Columnar Storage and List-based Processing for 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.

**PVLDB Reference Format:**
Pranjal Gupta, Amine Mhedhbi, Semih Salihoglu. Columnar Storage and List-based Processing for Graph Database Management Systems. PVLDB, 14(11): 2491-2504, 2021.
doi:10.14778/3476249.3476297

**PVLDB Artifact Availability:**
The source code, data, and/or other artifacts have been made available at https://github.com/graphflow/graphflow-columnar-techniques.

## 1 INTRODUCTION

Contemporary GDBMSs are data management software such as Neo4j [47], Neptune [5], TigerGraph [59], and GraphflowDB [32, 41] that adopt the property graph data model [48]. In this model, application data is represented as a set of vertices and edges, 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 [56] that search for patterns in a graph-structured database, which require reading large amounts of data. In the context of RDBMSs, column-oriented systems [29, 53, 57, 66] 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 [1, 2, 57, 68], and block-based query processing [3, 11]. 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 describe 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

Proceedings of the VLDB Endowment, Vol. 14, No. 11 ISSN 2150-8097.
doi:10.14778/3476249.3476297

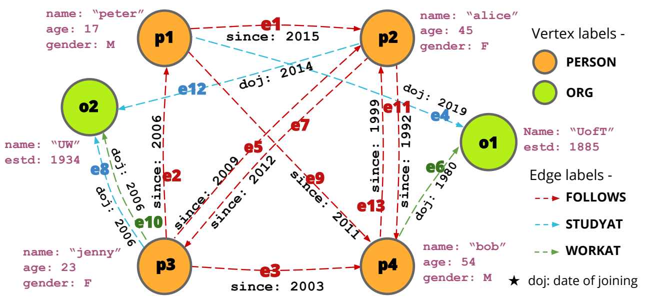


**Figure 1: Running example graph.**

next show that existing techniques for compressing NULL values in columns from references [1, 2] by Abadi et al. lead to very slow accesses to arbitrary non-NULL values. We then review Jacobson's bit vector index [30, 31] to support constant time rank queries, which has found several prior applications e.g., in a range filter structure in databases [64], in information retrieval [21, 44] and computational geometry [14, 45]. 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 compared to prior techniques.

***List-based Processing:*** In Section 6, we observe that traditional block-based processors or columnar RDBMSs [3, 67] process fixed-length blocks of data in tight loops, which achieves good CPU and cache utility but results in expensive data copies under n-n 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 n-n 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 [32]. 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 [25].

## 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* [12], 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, the adjacency list of $v$ is further clustered by edge label which enables efficient traversal of the neighbourhood of $v$, given a particular label. Vertex and edge properties can be stored in a number of ways. For example, some systems use a key-value store, such as DGraph [15] and JanusGraph [6], and some use a variant of *interpreted attribute layout* [9], where records consist of variable-sized key-value properties. Records can be located consecutively in disk or memory or have pointers to each other, as in Neo4j.

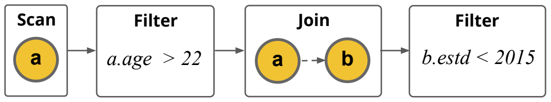


**Figure 2: Query plan for the query in Example 1.**

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 a query processor that uses variants of the following relational operators, which is the case in many GDBMSs, e.g., Neo4j [47], Memgraph [40], 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 $t$, Join 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-neighbour pair in $v_i$'s forward adjacency list[1].
Filter: Applies a predicate $\rho$ 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 [19]:*

```
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.*

[1]GraphflowDB can perform an intersection of multiple adjacency lists if the pattern is cyclic (see reference [41]).

Observe that `JOIN` accesses the edges and neighbours 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 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 n-n relationships have power-law degree distributions [37]. So, there are often many short adjacency lists in the dataset. For example, the `FLICKR`, `WIKI` graphs that 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 [33] 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 happen 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 for this is because the data in GDBMSs sometimes comes from structured data from RDBMSs as observed in a recent user survey [56]. In fact, several vendors and academics are actively working on defining a schema language for property graphs [13, 27]. 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 vertex and edge properties.* Similar to attributes of a relational table, properties on an edge or vertex and their datatypes can often 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: 1-n (single cardinality in the backward edges), n-1 (single cardinality in the forward edges), 1-1, and n-n. An example of 1-n 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.*

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

| 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. |

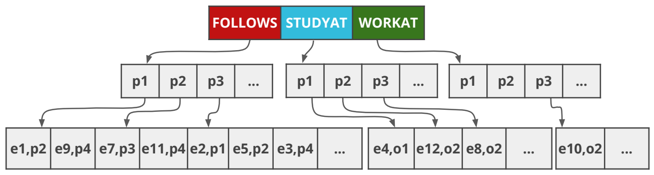


**Figure 3: Example forward adjacency lists implemented as a 2-level CSR structure for the example graph.**

## 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.

### 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, 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 by 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 have a *vertex column* for each $p_{i,j}$, that stores $p_{i,j}$ 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

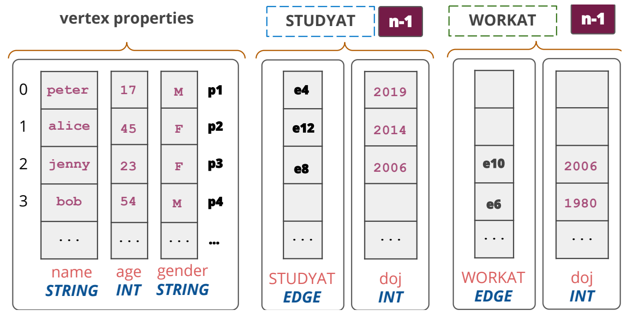


**Figure 4: Example vertex columns storing vertex properties and single-cardinality edges and their properties.**

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 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 duplicate storage of double-indexed property CSRs but still 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 in 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

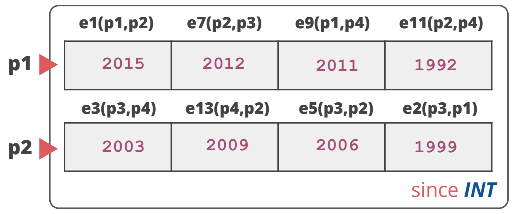


**Figure 5: Single-indexed property pages for since property of FOLLOWS edges in the example graph.** $k = 2$.

$e_1$, 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 forward adjacency list of $nbr_1$, $L_{nbr_1}$, 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. Therefore, when deletions happen, we need to leave gaps in adjacency lists and recycle them when insertions happen. This may leave many gaps in adjacency lists because to recycle a list-level offset, the system needs to wait for another insertion into the same adjacency list, which may be infrequent.

Our *single-indexed property pages* addresses these two issues (Figure 5). We store $k$ property lists (by default 128) in a property page. In a property page, properties of the same list does not have to be 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

Compression and query processing on compressed data are widely used in columnar RDBMSs. We start by reviewing techniques that apply directly to GDBMSs and are not novel. We then discuss the

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

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 [1, 2] and enhance one of them with Jacobson's bit vector index to make it 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 [2, 20, 36] 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 [2, 63, 68].This scheme maps a domain of values into compact codes using a variety of schemes [2, 23, 68], some producing variable-length codes, such as Huffmann encoding, and others fixed-length codes [2]. 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:** This scheme omits storing leading zero bits in each value of a given block of data [9]. 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 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 many small components, which can be factored out when the data depicts some structure (Desideratum 3). This allows compression without the need to decompress 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 in 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, edges of a particular label do not have any properties and only represent the relationships between vertices. For example, 10 out of 15 edge labels in LDBC SNB do not have any properties. In this case, edges need not be identifiable, as the system will not access their properties. We can therefore distinguish two edges by their neighbour 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.

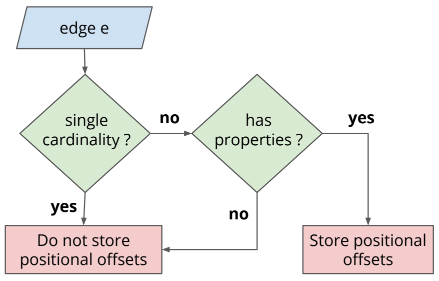


**Figure 6: Decision tree for storing page-level positional offsets of edges in adjacency lists.**

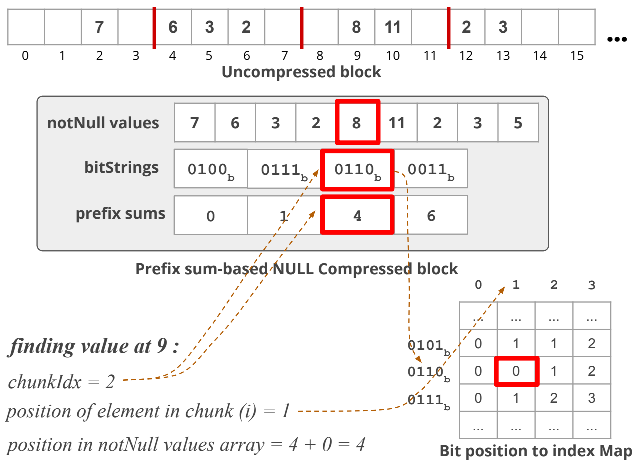
Figures 6 shows our decision tree to decide when to omit storing the page-level positional offsets in edge IDs.

## 5.3 NULL and Empty List Compression

Edge and vertex properties can often be quite sparse in real-world 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 [1] 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. The 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 as 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, compute the *rank* of $p$, i.e., the number of non-NULL values before $p$.

Abadi's third design, that uses a bit vector, already supports checking if the value at $p$ is NULL. To support rank queries, we enhance this design with a simplified version of Jacobson's bit vector index [30, 31]. 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*



Uncompressed block
notNull values
bitStrings
prefix sums
Prefix sum-based NULL Compressed block
finding value at 9 :
chunkIdx = 2
position of element in chunk (i) = 1
position in notNull values array = 4 + 0 = 4
Bit position to index Map

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

*map M* with $2^c * c$ 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 rank($p$) = $ps[p/c] + M[b, p \mod c]$.

The choice of $c$ 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 the overhead this 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 $2^c * c$ cells and needs to store a log($c$)-bit count value in each cell; (ii) we can compress a block of size $2^m$; and (iii) we store one prefix sum for each $c$ elements, so incur a cost of $m/c$ extra bits per element. By 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 [1]'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 limitations of traditional Volcano-style tuple-at-a-time processors and block-based processors of columnar RDBMSs when processing n-n 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 n-n edges and STUDYAT edges have single cardinality. Volcano-style tuple-at-a-time processing [22], which some GDBMSs adopt [41, 47], 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, $b_1 \dots b_{k_1}$, and each $b_i$ extends to $k_2$ many c's to $b_{ik_2} ..., b_{(i+1)k_2}$ (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.

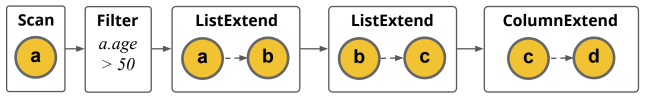


**Figure 8: Query plan for the query in Example 2.**

This is an important advantage for processing n-n 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 [10, 29], which process an entire block at a time in operators. Block sizes are fixed length, e.g. 1024 tuples [11, 17]. 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 n-n 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* [8, 51, 52] to address the data copying problem and uses block sizes set 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 n-n joins, repetitions of values are necessary. To address these repetitions we adopt a *factorized tuple set representation* scheme [52]. 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 $\geq 0$, the list group is flattened and represents a single tuple that consists of the curIdx'th values in the blocks.
- 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.

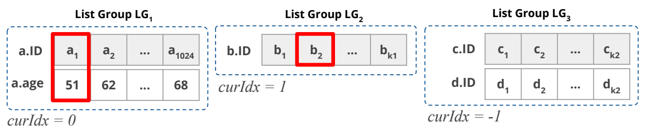


**Figure 9: Intermediate chunk for the query in Example 2. The first two list groups are flattened to single tuples, while the last one represents $k_2$ many tuples.**

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:* $(a_1, 51) \times (b_2) \times ((c_1, d_1) \cup \ldots \cup (c_2, d_2))$.

In addition, instead of using fixed-length blocks as in existing block-based processors, the blocks in each group can take different lengths, which are aligned to the lengths of adjacency lists in the database. As we shortly explain, this allows us to avoid materializing adjacency lists into the blocks.

## 6.2 Operators

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

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

**`ListExtend` and `ColumnExtend`**: In contrast to a single `Join` operator that implements index nested loop join algorithm using the adjacency list indices, such as `Expand` of Neo4j, we have two joins.

`ListExtend` is used to perform joins from a node, say, $a$ to nodes $b$ over 1-n or n-n edges $e$. The input list group $LG_a$ that holds the block of $a$ 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. It then loops through each $a$ value, say, $a_\ell$, and extends it to the set of $b$ and $e$ values using $a_\ell$'s adjacency list $Adj_{a_\ell}$. The blocks holding $b$ and $e$ values are put 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 blocks in $LG_b$, including those storing $b$ and $e$ as well as blocks that may be added later, will be equal to the length of $Adj_{a_\ell}$. This contrasts with fixed block sizes in existing block-based processors. In addition, we exploit that $Adj_{a_\ell}$ already stores $b$ and $e$ values as lists, and do not copy these to the intermediate chunk. Instead, the $b$ and $e$ blocks simply points to $Adj_{a_\ell}$.

`ColumnExtend` is used to perform 1-1 or n-1 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`:** 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 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 here and refer the reader to our code base [25]. Briefly, similar to `Filter`, `Group By And Aggregate` 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 [2, 60]. For example, count(*) 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 output by the* `ColumnExtend` *operator in the plan from Figure 8. In this, 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 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 [60]. 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. Insertion of edge properties in single-directional property pages are append only and do not require any sorting. Insertions to vertex columns are also simple as these too are unsorted structure. 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 new insertions. Note that this is also how node deletions are handled in Neo4j [46]. 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 prefixes after the location of update; and (iii) shifting the non-NULL elements array. These added complexities are an inherent trade off when integrating read-optimized techniques and can be mitigated by several existing techniques, like bulk updates or keeping a write-optimized second storage that keeps track of recent writes, which are not immediately merged. Positional delta trees [28] or C-Store's write-store are examples [57] of the latter technique.

Two of our techniques also require additional considerations when modifying the optimizer. 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 the backward direction. The former leads to sequential reads while the latter to random reads. The optimizer should assign costs based on this criterion 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, an in-memory GDBMS written in Java. We refer to this version of GraphflowDB as `GF-CL` (**C**olumnar **L**ist-based). We based our work on the publicly available version here [24], which we will refer to as `GF-RV` (**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, neighbour 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. Due to space constraints our experiment demonstrating benefits of vertex columns for single cardinality edges appears in the longer version of our paper [26].

### 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 to yield benefits under join queries over 1-n and n-n 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 [18] 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 and 34 (29 vertex, 5 edge) properties.

*JOB:* We used the IMDb movie database and the JOB benchmark [35]. Although the workload has originally been created to study optimizing join order selection, the dataset contains several n-n, 1-n, and 1-1 relationships between entities, like actors, movies, and companies, and structured properties, some of which contain NULLs. This makes it suitable to demonstrate the benefits from our storage and compression techniques. JOB also contains join queries over n-n relationships, making it suitable to demonstrate benefits of LBP. We created a 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 n-n 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 $r_\ell$ in a relationship table $T_\ell$ connects row $r_u$ and $r_v$. The final dataset can be found in our codebase [25].

`FLICKR` *and* `WIKI`*:* To enhance our microbenchmarks further, we use two additional datasets from the popular Konect graph sets [33] covering two application domains: a Flickr social network (`FLICKR`) [42] and a Wikipedia hyperlink graph between articles of the German Wikipedia (`WIKI`) [34]. 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 first demonstrate 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) `+0-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 using the new ID scheme that factors out components, such as edge and vertex labels, and using small size integers for positional offsets. We also see a 1.58x reduction by storing vertex properties in columns, which, unlike interpreted attribute layout, saves on storing the keys of the properties explicitly. 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 cheaper than in CSRs, as we do not need 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 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

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

**(a) LDBC100**

| | GF-RV | +COLS | +NEW-IDS | +0-SUPR | +NULL | GF-CL |
|---|---|---|---|---|---|---|
| Vertex | 31.40 | 19.87 | 19.87 | 19.87 | 19.28 | - |
| Props. | | **+1.58x** | - | - | **+1.03x** | **1.62x** |
| Edge | 7.92 | 2.07 | 2.07 | 2.07 | 2.05 | - |
| Props. | | **+3.82x** | - | - | **+1.01x** | **3.87x** |
| F. Adj. | 31.93 | 28.95 | 20.67 | 11.41 | 10.78 | - |
| Lists | | **+1.10x** | **+1.40x** | **+1.81x** | **+1.06x** | **2.96x** |
| B. Adj. | 31.29 | 31.07 | 24.93 | 13.10 | 11.41 | - |
| Lists | | **+1.01x** | **+1.25x** | **+1.90x** | **+1.15x** | **2.74x** |
| Total (GB) | 102.56 | 81.97 | 67.55 | 46.45 | 43.54 | - |
| | | **+1.25x** | **+1.21x** | **+1.45x** | **+1.07x** | **2.36x** |

**(b) IMDb**

| | GF-RV | +COLS | +NEW-IDS | +0-SUPR | +NULL | GF-CL |
|---|---|---|---|---|---|---|
| Vertex | 2.54 | 1.98 | 1.98 | 1.98 | 1.96 | - |
| Props. | | **+1.28x** | - | - | **+1.01x** | **1.29x** |
| Edge | 2.81 | 1.63 | 1.63 | 1.63 | 0.90 | - |
| Props. | | **+1.72x** | - | - | **+1.83x** | **3.14x** |
| F. Adj. | 1.13 | 1.02 | 0.65 | 0.41 | 0.36 | - |
| Lists | | **+1.10x** | **+1.57x** | **+1.57x** | **+1.15x** | **2.96x** |
| B. Adj. | 1.10 | 1.10 | 0.76 | 0.50 | 0.49 | - |
| Lists | | **+1.00x** | **+1.45x** | **+1.51x** | **+1.01x** | **2.20x** |
| Total (GB) | 7.57 | 5.74 | 5.02 | 4.53 | 3.72 | - |
| | | **+1.32x** | **+1.14x** | **+1.11x** | **+1.22x** | **2.03x** |

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 reduction factors in forward and backward lists, which is comparable to that of LDBC. However, there are two main differences. First, we save more by compressing the edge properties using NULL compression, because 7 of 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: (i) EDGE COLS: Stores edge properties in an edge column in a randomized way as edges are given random edge IDs; (ii) PROP PAGES: Edge properties are stored in forward-directional property pages with $k$=128 . In the longer version of our paper [26], we test sensitivity of $k$ that demonstrates read performance from property pages in our datasets are similar until $k$=512 and slows down for larger value of $k$ .

**Table 3: Runtime (in secs) of k-hop (H) queries when using property pages (PAGE$_P$) vs edge columns (COL$_E$).**

| | | 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 | 34.22 | 0.96 | 16.48 | 0.42 | 189.39 |
| | | **3.4x** | **1.9x** | **3.1x** | **2.6x** | **4.5x** | **4.7x** |
| P$_B$ | COL$_E$ | 1.23 | 131.01 | 6.33 | 99.28 | 2.40 | 1009.84 |
| | PAGE$_P$ | 1.29 | 134.43 | 6.10 | 91.75 | 2.25 | 1183.14 |
| | | **0.9x** | **1.0x** | **1.0x** | **1.1x** | **1.1x** | **0.9x** |

We use 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). 1-hop query compares the edge's timestamp for WIKI and FLICKR and the creationDate property for LDBC to be greater than a constant. 2-hop query compares the property of each query edge to be greater than the previous edge's property. Since WIKI contains prohibitively many 2-hops we put a predicate on the source and destination nodes to make queries finish within reasonable time. For each query and configurations, we consider two plans: (i) the *forward plan* that matches vertices from left to right in forward direction; (ii) the *backward plan* that matches in reverse order.

Forward plans perform sequential reads of properties under PROP-PAGES, achieving good CPU cache locality. Therefore, they are expected to be more performant than backward plans under PROP-PAGES as well as both the plans plans under EDGE COLS, which all 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 1.9x to 4.7x faster than the forward plans under EDGE COLS and are also faster than the backward plans under PROP-PAGES. In contrast, the performance of both backward plans are comparable. This is because neither edge columns nor forward-directional property pages provide any locality for accessing properties in order of backward adjacency lists. This confirms our claim in Section 4.2 that PROP-PAGES is a better design than using vanilla edge columns.

## 8.4 Null Compression

We demonstrate the memory/performance trade-off of our NULL compression scheme on sparse vertex property columns. We create multiple versions of the LDBC100, with the creationDate property of Comment vertices containing different percentage of NULL values. LDBC100 contains 220M Comment vertices, so our column has 220M entries. We use the following 1-hop query: MATCH (a:Person)−[e:Likes]→(b:Comment) RETURN b.creationDate. This query is evaluated 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 default configuration (m=16, c=16); (ii) Vanilla-NULL is the vanilla bit string-based scheme from reference [1]; and (ii) Uncompressed stores the column in an uncompressed format. In the longer version of our paper [26], we demonstrate a sensitivity analysis for

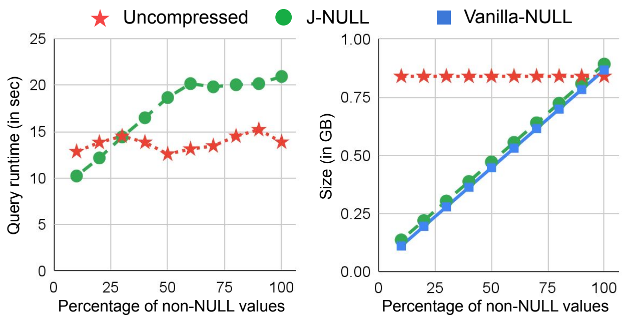


**Figure 10: Query performance and memory consumption when storing a vertex property column as uncompressed, compressed with Jacobson's scheme, and the vanilla bit string scheme from Abadi, under different density levels.**

J-NULL running under different m and c values. This experiment shows that read performance is insensitive to these 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 query performance under three different configurations. Recall that with default configuration J-NULL requires slightly more memory than Vanilla-NULL, 2 bits per element instead of 1 bit. As expected the performance of J-NULL is slightly 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 (with >70% 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 at element's cell, which has a higher chance of a CPU cache miss.

## 8.5 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 in existing systems, like Neo4j [47] or MemGraph [40]. 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-neighbour 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 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 4. We see that GF-CL outperforms GF-CV by large margins, between 2.7x and 15.2x.

**Table 4: Runtime (ms) of GF-RV and GF-CL (LBP) plans.**

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

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 for aggregation are shown in the COUNT(*) rows of Table 4. 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.

## 8.6 Baseline System Comparisons

In our final experiment, we compare the query performance of GF-CL against GF-RV, Neo4j, which is another row-oriented and Volcano style GDBMSs, and two columnar RDBMSs, MonetDB and Vertica, which are not tailored for n-n 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, on which we base our work, is already competitive with or outperforms other baseline systems on workloads containing n-n joins. We used the SNB on LDBC10 and JOB, both of which contain n-n join queries.

We used the community version v4.2 of Neo4j GDBMS [47], the community version 10.0 of Vertica [61] and MonetDB 5 server 11.37.11 [43]. We note that our experiments should not be interpreted as one system being more efficient than another. It is difficult to meaningfully compare completely separate systems, e.g., all baseline systems have many tunable parameters, and some have more

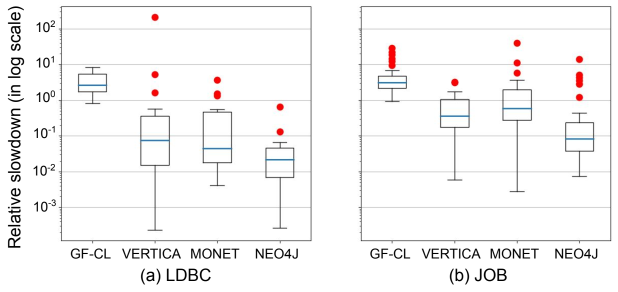


**Figure 11: Relative speedup/slowdown of the different systems in comparison to GF-RV on LDBC10. The boxplots show the 5th, 25th, 50th, 75th, and 95th percentiles.**

efficient enterprise versions. For all baseline systems, we map their storage to an in-memory filesystem, set number of CPUs to 1 and disable spilling intermediate files to disk. We maintain 2 copies of edge tables for Vertica and MonetDB, sorted by the source and destination vertexIDs, respectively. For GF-RV and GF-CL, we use the best left-deep plan we could manually pick, which was obvious in most cases. For example, 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.

*8.6.1 LDBC.* We use the LDBC10 dataset. GraphflowDB is a prototype system that implements parts of the Cypher language 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 [18] 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 can be found in the longer version of our paper [26].

Figure 11a shows the relative speedup/slowdown of the different systems in comparison to GF-RV. We report individual runtime numbers of all the queries in the longer version of our paper [26]. As expected, GF-CL is broadly more performant than GF-RV on LDBC with a median query improvement factor of 2.6x. With the exception of one query, which slows down a bit, the performance of *every query* improves between 1.3x to 8.3x. The improvements come from several optimizations but primarily from LBP and our columnar storage. In GF-RV, scanning properties requires checking equality on property keys, which are avoided in columnar storage, so we observed large improvements on queries that produce large intermediate results and perform filters, such as IC05. IC05 has 4 n-n 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 neighbour IDs to intermediate tuples. More importantly, LBP performs filters inside loops and GF-CL's single-indexed property pages provides faster access to the edge properties that are used in the filter than GF-RV's row-oriented format. On this query, GF-RV takes 8.9s while GF-CL takes 1.6s.

As we expected, we also found 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 performed slightly worse than other baselines, we also observed that there were some 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, had several n-n joins, but did not generate large intermediate results, like IS02 or IC06. On such queries, GDBMSs, 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 scanning both Person and Knows tables. Instead, Neo4j and GraphflowDB only scan the Person table to find $p_i$ and then extend $p_i$ to its neighbours, without scanning all Knows edges. For this, 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 n-n joins that generate large intermediate results. For example, on IC05 that we reviewed before, Vertica take 1 minute, MonetDB 3.25 minutes, while Neo4j took over 10 minutes.

*8.6.2 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 JOB queries returned aggregations on strings, such as min(name), where name is a string column. Since Graphflow supports aggregations only on numeric types, we removed these aggregations. Our final queries can be found in the longer version of our paper [26].

Figure 11b shows the relative performance of different systems in comparison to GF-RV. The individual runtime numbers of each query can be found in the longer version of our paper [26]. Similar to our LDBC results, we see GF-CL to improve the performance, now by 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, i.e, >20x. In LDBC, the largest improvement factor was 8.3x. This is expected as 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. Whereas on star queries, multiple extensions from the center node can remain unflattened. Therefore GF-CL's plans can benefit more from LBP as they can compress their intermediate tuples more. We also see that similar to LDBC, GF-RV is more performant than the columnar RDBMSs.

However, 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 benefit 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, Neo4j is again least competitive of these baselines.

## 9 RELATED WORK

Column stores [29, 57, 66, 67] are designed primarily for OLAP queries that perform aggregations over large amounts of data. Work on them introduced a set of storage and query processing techniques which include use of positional offsets, schemes for compression, block-based query processing, late materialization and operations on compressed data, among others. A detailed survey of these techniques can be found in reference [60]. This paper aims to integrate some of these techniques into in-memory GDBMSs.

Existing GDBMSs and RDF systems usually store the graph topology in a columnar structure. This is done either by using a variant of adjacency list or CSR. Instead, systems often use row-oriented structures to store properties, such as an interpreted attribute layout [9]. For example, Neo4j [47] represents the graph topology in adjacency lists that are partitioned by edge labels and stored in linked-lists, where each edge record points to the next. Properties of each vertex/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. JanusGraph too [6] stores edges in adjacency lists partitioned by edge labels and properties as consecutive key-value pairs (a row-oriented format). These 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 compress edge and vertex IDs and NULLs.

There are also several GDBMSs that are developed directly on top of an RDBMS or another database system [54], such as IBM Db2 Graph [58], Oracle Spatial and Graph [54] and SAP's graph database [55]. 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 [38] implements a limited SQL called relationship queries that support joins of tables similar to path queries, followed with aggregations. The system stores n-n relationship in tables with CSR-like indices and 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 done in 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).

Several RDF systems also use columnar structures to store RDF data. Reference [4] uses a set of columns, where each column store is a set of (subject, object) pairs for a unique predicate. However, this storage is not as optimized as the storage in GDBMSs, e.g., the edges between entities are not stored in native CSR or adjacency list format. Hexastore [62] improves on the idea of predicate partitioning by having a column for each RDF element (subject, predicate or object) and sorting it in 2 possible ways in B+ trees. This is similar but not as efficient as double indexing of adjacency lists in GDBMSs. RDF-3X [50] 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. Similar to our Guideline 3, reference [49] also observes that graphs have 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 novel storage techniques for storing graphs are optimized for write-heavy workloads, such as streaming. These works propose data structures that try to achieve the sequential read capabilities of CSR while being write-optimized. Examples of this include LiveGraph [65], Aspen [16], and LLAMA [39]. 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 factorized representations in RDBMSs, specifically FDB [7, 8], represents intermediate data 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 and does not studies how to integrate more advanced factorized processing techniques inside GDBMS.

## 10 CONCLUSIONS

Columnar 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 such techniques into GDBMSs, which are also read-optimized analytical systems. While some techniques can be directly applied to GDBMSs, adaptation of others can be significantly sub-optimal in terms of space and performance. In this paper, we first outlined a set of guidelines and desiderata for designing the 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.

## REFERENCES

[1] Daniel J. Abadi. 2007. Column Stores for Wide and Sparse Data. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007*. 292–297. http://cidrdb.org/cidr2007/papers/cidr07p33.pdf

[2] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating Compression and Execution in Column-Oriented Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2006*. 671–682. https://doi.org/10.1145/1142473.1142548

[3] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-Stores vs. Row-Stores: How Different Are They Really?. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2008*. 967–980. https://doi.org/10.1145/1376616.1376712

[4] Daniel J. Abadi, Adam Marcus, Samuel Madden, and Katherine J. Hollenbach. 2007. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data*. 411–422. http://www.vldb.org/conf/2007/papers/research/p411-abadi.pdf

[5] Amazon. 2020. Amazon Neptune. https://aws.amazon.com/neptune/. Last Accessed July 25, 2021.

[6] JanusGraph Authors. 2020. JanusGraph. https://janusgraph.org. Last Accessed July 25, 2021.

[7] Nurzhan Bakibayev, Tomás Kociský, Dan Olteanu, and Jakub Zavodny. 2013. Aggregation and Ordering in Factorised Databases. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1990–2001. http://www.vldb.org/pvldb/vol6/p1990-zavodny.pdf

[8] Nurzhan Bakibayev, Dan Olteanu, and Jakub Zavodny. 2012. FDB: A Query Engine for Factorised Relational Databases. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1232–1243. http://vldb.org/pvldb/vol5/p1232_nurzhanbakibayev_vldb2012.pdf

[9] Jennifer L. Beckmann, Alan Halverson, Rajasekar Krishnamurthy, and Jeffrey F. Naughton. 2006. Extending RDBMSs to Support Sparse Datasets using an Interpreted Attribute Storage Format. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*. 58. https://doi.org/10.1109/ICDE.2006.67

[10] Peter Boncz. 2002. *Monet: A Next-Generation Database Kernel for Query-Intensive Applications*. Ph.D. Dissertation. Universiteit van Amsterdam. https://ir.cwi.nl/pub/14832/14832A.pdf

[11] Peter A. Boncz, Marcin Zukowski, and Niels Nes. 2005. MonetDB/X100: Hyper-Pipelining Query Execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005*. 225–237. http://cidrdb.org/cidr2005/papers/P19.pdf

[12] Angela Bonifati, George H. L. Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs*. https://doi.org/10.2200/S00873ED1V01Y201808DTM051

[13] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. 2019. Schema Validation and Evolution for Graph Databases. In *Conceptual Modeling - 38th International Conference, ER 2019 (Lecture Notes in Computer Science)*, Vol. 11788. 448–456. https://doi.org/10.1007/978-3-030-33223-5_37

[14] Prosenjit Bose, Meng He, Anil Maheshwari, and Pat Morin. 2009. Succinct Orthogonal Range Search Structures on a Grid with Applications to Text Indexing. In *Algorithms and Data Structures, 11th International Symposium, WADS 2009 (Lecture Notes in Computer Science)*, Vol. 5664. 98–109. https://doi.org/10.1007/978-3-642-03367-4_9

[15] DGraph. 2020. DGraph Github Repository. https://github.com/dgraph-io/dgraph. Last Accessed July 25, 2021.

[16] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2019. Low-latency Graph Streaming using Compressed Purely-functional Trees. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*. 918–934. https://doi.org/10.1145/3314221.3314598

[17] DuckDB. 2020. DuckDB. https://duckdb.org/. Last Accessed July 25, 2021.

[18] Orri Erling, Alex Averbuch, Josep Lluís Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat-Pérez, Minh-Duc Pham, and Peter A. Boncz. 2015. The LDBC Social Network Benchmark: Interactive Workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, SIGMOD 2015*. 619–630. https://doi.org/10.1145/2723372.2742786

[19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 ACM SIGMOD International Conference on Management of Data, SIGMOD 2018*. 1433–1445. https://doi.org/10.1145/3183713.3190657

[20] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proceedings of the Fourteenth International Conference on Data Engineering, ICDE 1998*. 370–379. https://doi.org/10.1109/ICDE.1998.655800

[21] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. 1992. New Indices for Text: Pat Trees and Pat Arrays. In *Information Retrieval: Data Structures & Algorithms*. 66–82.

[22] Goetz Graefe. 1994. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering, TKDE* 6, 1 (1994), 120–135. https://doi.org/10.1109/69.273032

[23] G. Graefe and L.D. Shapiro. 1991. Data Compression and Database Performance. In *Proceedings of the 1991 Symposium on Applied Computing*. 22–27. https://doi.org/10.1109/SOAC.1991.143840

[24] Graphflow. 2020. GraphflowDB Source Code. https://github.com/queryproc/optimizing-subgraph-queries-combining-binary-and-worst-case-optimal-joins/. Last Accessed July 25, 2021.

[25] Graphflow. 2021. GraphflowDB Columnar Techniques. https://github.com/graphflow/graphflow-columnar-techniques. Last Accessed July 25, 2021.

[26] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. *Columnar Storage and List-based Processing for Graph Database Management Systems*. Technical Report. https://github.com/graphflow/graphflow-columnar-techniques/blob/master/paper.pdf

[27] Olaf Hartig and Jan Hidders. 2019. Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), GRADES-NDA 2019*. 6:1–6:11. https://doi.org/10.1145/3327964.3328495

[28] Sándor Héman, Marcin Zukowski, Niels J. Nes, Lefteris Sidirourgos, and Peter A. Boncz. 2010. Positional Update Handling in Column Stores. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2010*. 543–554. https://doi.org/10.1145/1807167.1807227

[29] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45. http://sites.computer.org/debull/A12mar/monetdb.pdf

[30] Guy Jacobson. 1989. Space-efficient Static Trees and Graphs. In *30th Annual Symposium on Foundations of Computer Science, FOCS 1989*. 549–554. https://doi.org/10.1109/SFCS.1989.63533

[31] Guy Jacobson. 1989. *Succinct Static Data Structures*. Ph.D. Dissertation. Carnegie Mellon University.

[32] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An Active Graph Database. In *Proceedings of the 2017 ACM SIGMOD International Conference on Management of Data, SIGMOD 2017*. 1695–1698. https://doi.org/10.1145/3035918.3056445

[33] Jérôme Kunegis. 2013. KONECT: The Koblenz Network Collection. In *22nd International World Wide Web Conference, WWW 2013*. 1343–1350. https://doi.org/10.1145/2487788.2488173

[34] Jérôme Kunegis. 2021. Wikipedia Dynamic (de), (Konect). http://konect.cc/networks/link-dynamic-dewiki/. Last Accessed July 25, 2021.

[35] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215. http://www.vldb.org/pvldb/vol9/p204-leis.pdf

[36] Daniel Lemire and Leonid Boytsov. 2015. Decoding Billions of Integers per Second through Vectorization. *Software: Practice and Experience* 45, 1 (2015), 1–29. https://doi.org/10.1002/spe.2203

[37] Jure Leskovec, Jon M. Kleinberg, and Christos Faloutsos. 2005. Graphs Over Time: Densification Laws, Shrinking Diameters and Possible Explanations. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, SIGKDD 2005*. 177–187. https://doi.org/10.1145/1081870.1081893

[38] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. *Proceedings of the VLDB Endowment* 10, 3 (2016), 265–276. http://www.vldb.org/pvldb/vol10/p265-lin.pdf

[39] Peter Macko, Virendra J. Marathe, Daniel W. Margo, and Margo I. Seltzer. 2015. LLAMA: Efficient graph analytics using Large Multiversioned Arrays. In *31st IEEE International Conference on Data Engineering, ICDE 2015*. 363–374. https://doi.org/10.1109/ICDE.2015.7113298

[40] Memgraph. 2020. Memgraph. https://memgraph.com/. Last Accessed July 25, 2021.

[41] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *Proceedings of the VLDB Endowment* 12, 11 (2019), 1692–1704. http://www.vldb.org/pvldb/vol12/p1692-mhedhbi.pdf

[42] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. 2008. Growth of the Flickr Social Network. In *Proceedings of the first Workshop on Online Social Networks, WOSN 2008*. 25–30. https://doi.org/10.1145/1397735.1397742

[43] MonetDB. 2020. MonetDB source code, (Jun2020-SP1). https://github.com/MonetDB/MonetDB/releases/tag/Jun2020_SP1_release. Last Accessed July 25, 2021.

[44] Gonzalo Navarro and Veli Mäkinen. 2007. Compressed Full-Text Indexes. *Comput. Surveys* 39, 1 (2007), 2. https://doi.org/10.1145/1216370.1216372

[45] Gonzalo Navarro, Yakov Nekrich, and Luís M. S. Russo. 2013. Space-efficient Data-Analysis Queries on Grids. *Theoretical Computer Science* 482 (2013), 60–72. https://doi.org/10.1016/j.tcs.2012.11.031

[46] Neo4j. 2020. Neo4j Blog on Deletions. https://neo4j.com/developer/kb/how-deletes-workin-neo4j/. Last Accessed July 25, 2021.

[47] Neo4j. 2020. Neo4j Community Edition. https://neo4j.com/download-center/#community. Last Accessed July 25, 2021.

[48] Neo4j. 2020. Neo4j Property Graph Model. https://neo4j.com/developer/graph-database. Last Accessed July 25, 2021.

[49] Thomas Neumann and Guido Moerkotte. 2011. Characteristic Sets: Accurate Cardinality Estimation for RDF Queries with Multiple Joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011*. 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[50] Thomas Neumann and Gerhard Weikum. 2010. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal* 19, 1 (2010), 91–113. https://doi.org/10.1007/s00778-009-0165-y

[51] Dan Olteanu and Maximilian Schleich. 2016. Factorized Databases. *SIGMOD Record* 45, 2 (2016), 5–16. https://doi.org/10.1145/3003665.3003667

[52] Dan Olteanu and Jakub Závodný. 2015. Size Bounds for Factorised Representations of Query Results. *ACM Transactions on Database Systems (TODS)* 40, 1 (2015), 2:1–2:44. https://doi.org/10.1145/2656335

[53] Oracle. 2020. Oracle In-Memory Column Store Architecture. https://tinyurl.com/vkvb6p6. Last Accessed July 25, 2021.

[54] Oracle. 2020. Oracle Spatial and Graph. https://www.oracle.com/database/technologies/spatialandgraph.html. Last Accessed July 25, 2021.

[55] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The Graph Story of the SAP HANA Database. In *Datenbanksysteme für Business, Technologie und Web (BTW), 15. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS) (LNI)*, Vol. P-214. 403–420. https://dl.gi.de/20.500.12116/17334

[56] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2020. The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey. *VLDB Journal* 29, 2-3 (2020), 595–618. https://doi.org/10.1007/s00778-019-00548-x

[57] Mike Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Sam Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stan Zdonik. 2019. C-store: A Column-Oriented DBMS. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 491–518. https://doi.org/10.1145/3226595.3226638

[58] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Suijun Tong, Wen Sun, Thomas Kolanko, Md. Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data, SIGMOD 2020*. 345–359. https://doi.org/10.1145/3318464.3386138

[59] TigerGraph. 2020. TigerGraphDB. https://www.tigergraph.com. Last Accessed July 25, 2021.

[60] Syunsuke Uemura, Toshitsugu Yuba, Akio Kokubu, Ryoichi Ooomote, and Yasuo Sugawara. 1980. The Design and Implementaion of a Magnetic-Bubble Database Machine. In *Information Processing, Proceedings of the 8th IFIP Congress 1980*. 433–438.

[61] Vertica. 2020. Vertica 10.0.x Documentation. https://www.vertica.com/docs/10.0.x/HTML/Content/Home.html. Last Accessed July 25, 2021.

[62] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. 2008. Hexastore: Sextuple Indexing for Semantic Web Data Management. *Proceedings of the VLDB Endowment* 1, 1 (2008), 1008–1019. http://www.vldb.org/pvldb/vol1/1453965.pdf

[63] Till Westmann, Donald Kossmann, Sven Helmer, and Guido Moerkotte. 2000. The Implementation and Performance of Compressed Databases. *SIGMOD Record* 29, 3 (2000), 55–67. https://doi.org/10.1145/362084.362137

[64] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2020. Succinct Range Filters. *ACM Transactions on Database Systems (TODS)* 45, 2 (2020), 5:1–5:31. https://doi.org/10.1145/3375660

[65] Xiaowei Zhu, Marco Serafini, Xiaosong Ma, Ashraf Aboulnaga, Wenguang Chen, and Guanyu Feng. 2020. LiveGraph: A Transactional Graph Storage System with Purely Sequential Adjacency List Scans. *Proceedings of the VLDB Endowment* 13, 7 (2020), 1020–1034. http://www.vldb.org/pvldb/vol13/p1020-zhu.pdf

[66] Marcin Zukowski and Peter A. Boncz. 2012. From X100 to Vectorwise: Opportunities, Challenges and Things Most Researchers Do Not Think About. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012*. 861–862. https://doi.org/10.1145/2213836.2213967

[67] Marcin Zukowski and Peter A. Boncz. 2012. Vectorwise: Beyond Column Stores. *IEEE Data Engineering Bulletin* 35, 1 (2012), 21–27. http://sites.computer.org/debull/A12mar/vectorwise.pdf

[68] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz. 2006. Super-Scalar RAM-CPU Cache Compression. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006*. 59. https://doi.org/10.1109/ICDE.2006.150