2021 IEEE 37th International Conference on Data Engineering (ICDE) | 978-1-7281-9184-3/20/$31.00 ©2021 IEEE | DOI: 10.1109/ICDE51399.2021.00130

A+ Indexes: Tunable and Space-Efficient Adjacency Lists in Graph Database Management Systems

Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, Semih Salihoglu
Cheriton School of Computer Science, University of Waterloo
{amine.mhedhbi, pranjal.gupta, shahid.khaliq, semih.salihoglu}@uwaterloo.ca

***Abstract*— Graph database management systems (GDBMSs) are highly optimized to perform fast traversals, i.e., joins of vertices with their neighbours, by indexing the neighbourhoods of vertices in adjacency lists. However, existing GDBMSs have system-specific and fixed adjacency list structures, which makes each system efficient on only a fixed set of workloads. We describe a new tunable indexing subsystem for GDBMSs, we call A+ indexes, with materialized view support. The subsystem consists of two types of indexes: (i) vertex-partitioned indexes that partition 1-hop materialized views into adjacency lists on either the source or destination vertex IDs; and (ii) edge-partitioned indexes that partition 2-hop views into adjacency lists on one of the edge IDs. As in existing GDBMSs, a system by default requires one forward and one backward vertex-partitioned index, which we call the primary A+ index. Users can tune the primary index or secondary indexes by adding nested partitioning and sorting criteria. Our secondary indexes are space-efficient and use a technique we call *offset lists*. Our indexing subsystem allows a wider range of applications to benefit from GDBMSs' fast join capabilities. We demonstrate the tunability and space efficiency of A+ indexes through extensive experiments on three workloads.**

I. Introduction

The term *graph database management system* (GDBMS) in its contemporary usage refers to data management software such as Neo4j [1], JanusGraph [2], TigerGraph [3], and GraphflowDB [4], [5] that adopt the property graph data model [6]. In this model, entities are represented by vertices, relationships are represented by edges, and attributes by arbitrary key-value properties on vertices and edges. GDBMSs have lately gained popularity among a wide range of applications from fraud detection and risk assessment in financial services to recommendations in e-commerce [7]. One reason GDBMSs appeal to users is that they are highly optimized to perform very fast joins of vertices with their neighbours. This is primarily achieved by using *adjacency list indexes* [8], which are join indexes that are used by GDBMSs' join operators.

Adjacency list indexes are often implemented using constant-depth data structures, such as the compressed sparse-row (CSR) structure, that partition the edge records into lists by source or destination vertex IDs. Some systems adopt a second level partitioning in these structures by edge labels. These partitionings provide constant time access to neighbourhoods of vertices and contrasts with tree-based indexes, such as B+ trees, which have logarithmic depth in the size of the data they index. Some systems further sort these lists according to some properties, which allows them to use fast intersection-based join algorithms, such as the novel intersection-based worst-case optimal (WCO) join algorithms [9]. However, a major shortcoming of existing GDBMSs is that systems make different but fixed choices about the partitioning and sorting criteria of their adjacency list indexes, which makes each system highly efficient on only a fixed set of workloads. This creates physical data dependence, as users have to model their data, e.g., pick their edge labels, according to the fixed partitioning and sorting criteria of their systems.

We address the following question: *How can the fast join capabilities of GDBMSs be expanded to a much wider set of workloads?* We are primarily interested in solutions designed for read-optimized GDBMSs. This is informed by a recent survey of users and applications of GDBMSs that we conducted [7], that indicated that GDBMSs are often used in practice to support read-heavy applications, instead of primary transactional stores. As our solution, we describe a tunable and space-efficient indexing subsystem for GDBMSs that we call *A+ indexes*. Our indexing subsystem consists of a *primary* index and optional *secondary* indexes that users can build. This is similar to relational systems that index relations in a primary B+ tree index on the primary key columns as well as optional secondary indexes on other columns. Primary A+ indexes are the default indexes that store all of the edge records in a database. Unlike existing GDMBSs, users can tune the primary A+ index of the system by adding arbitrary nested partitioning of lists into sublists and providing a sorting criterion per sublist. We store these lists in a nested CSR data structure, which provides constant time access to vertex neighborhoods that can benefit a variety of workloads.

We next observe that partitioning edges into adjacency lists is equivalent to creating multiple materialized views where each view is represented by a list or a sublist within a list. Similarly, the union of all adjacency lists can be seen as the coarsest view, which we refer to as the *global view*. In existing systems and primary A+ indexes, the global view is a trivial view that contains all of the edges in the graph. Therefore, one way a GDBMS can support an even wider range of workloads is by indexing other views inside adjacency lists. However storing and indexing views in secondary indexes results in data duplication and consumes extra space, which can be prohibitive for some views.

Instead of extending our system with general view functionality, our next contribution carefully identifies two sets of global views that can be stored in a highly space-efficient manner when partitioned appropriately into lists: (i) 1-hop views that satisfy arbitrary predicates that are stored in *secondary vertex-*

978-1-7281-9184-3/21/$31.00 ©2021 IEEE
DOI 10.1109/ICDE51399.2021.00130

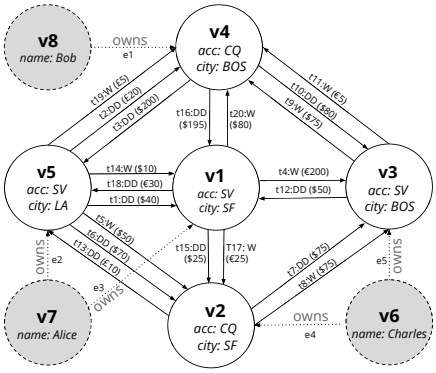


Fig. 1: Example financial graph.

partitioned A+ indexes; and (ii) 2- hop views that are stored in *secondary edge-partitioned A+ indexes*, which extend the notion of neighborhood from vertices to edges, i.e., each list stores a set of edges that are adjacent to a particular edge. These two sets of views and their accompanying partitioning methods guarantee that the final lists that are stored in secondary A+ indexes are subsets of lists in the primary A+ index. Based on this property, we implement secondary A+ indexes by a technique we call *offset lists*, which identify each indexed edge by an offset into a list in the primary A+ index. Due to the sparsity, i.e., small average degrees, of real-world graphs, each list in the primary A+ index often contains a very small number of edges. This makes offset lists highly space-efficient, taking a few bytes per indexed edge instead of the ID lists in the primary index that store globally identifiable IDs of edges and neighbor vertices, each of which are often 8 bytes in existing systems. Similar to the primary A+ index, secondary indexes are implemented in a CSR structure that support nested partitioning, where the lower level is the offset lists. To further improve the space-efficiency of secondary A+ indexes, we identify cases when the secondary A+ indexes can share the partitioning levels of the primary A+ index.

We implemented A+ indexes inside the GraphflowDB in-memory GDBMS [5]. We describe the modifications we made to the optimizer and query processor of the system to use our indexes in query plans. We present examples of highly efficient plans that our system is able to generate using our indexing subsystem that do not exist in the plan spaces of existing systems. We demonstrate the tunability and space efficiency of A+ indexes by showing how to tune GraphflowDB to be highly efficient on three different workloads using either primary index reconfigurations or building secondary indexes with very small memory overhead. GraphflowDB is a read-optimized system that does not support transactions but allows non-transactional updates. Although update performance is not our focus, for completeness of our work, we report the update performance of A+ indexes in the longer version of our paper [10].

Figure 1 shows an example financial graph that we use as a running example throughout this paper. The graph contains vertices with `Customer` and `Account` labels. `Customer` vertices have `name` properties and `Account` vertices have `city` and `accountType(acc)` properties. From customers to accounts are edges with `Owns(O)` labels and between accounts are transfer edges with `Dir-Deposit(DD)` and `Wire(W)` labels with `amount(amt)`, `currency`, and `date` properties. We omit dates in the figure and give each transfer edge an ID such that $t_i.date < t_j.date$ if $i < j$.

II. Overview of Existing Adjacency List Indexes

Adjacency lists are accessed by GDBMS's join operators e.g., Expand in Neo4j or Extend/Intersect in GraphflowDB, that join vertices with neighbours. GDBMSs employ two broad techniques to provide fast access to adjacency lists while performing these joins:

(1) Partitioning: GDBMSs often partition their edges first by their source or destination vertex IDs, respectively in *forward* and *backward* indexes; this is the primary partitioning criterion.

Example 1: Consider the following 2-hop query, written in openCypher [11], that starts from a vertex with name "Alice". Below, a_i, c_j, and r_k are variables for the `Account` and `Customer` query vertices and query edges, respectively.

MATCH $c_1{-}[r_1]{->}a_1{-}[r_2]{->}a_2$
WHERE $c_1.name$ = '*Alice*'

In every GDBMS we know of, this query is evaluated in three steps: (1) scan the vertices and find a vertex with `name` "Alice" and match a_1, possibly using an index on the `name` property. In our example graph, `v7` would match c_1; (2) access `v7`'s forward adjacency list, often with one lookup, to match $c_1 \rightarrow a_1$ edges; and (3) access the forward lists of matched a_1's to match $c_1 \rightarrow a_1 \rightarrow a_2$ paths.

Some GDBMSs employ further partitioning on each adjacency list, e.g., Neo4j [1] partitions edges on vertices and then by edge labels. Given the ID of a vertex v, this allows constant time access to: (i) all edges of v; and (ii) all edges of v with a particular label through the lower level lists e.g., all edges of v with label `Owns`.

Example 2: Consider the following query that returns all `Wire` transfers made from the accounts Alice `Owns`:

MATCH $c_1{-}[r_1{:}O]{->}a_1{-}[r_2{:}W]{->}a_2$
WHERE $c_1.name$ = '*Alice*'

The "r_1:O" is syntactic sugar in Cypher for the `r1.label=Owns` predicate. A system with lists partitioned by vertex IDs and edge labels can evaluate this query as follows. First, find `v7`, with name "Alice", and then access `v7`'s `Owns` edges, often with a constant number of lookups and without running any predicates, and match a_1's. Finally access the `Wire` edges of each a_1 to match the a_2's.

(2) Sorting: Some systems further sort their most granular lists according to an edge property [2] or the IDs of the neighbours in the lists [4], [12]. Sorting enables systems to access parts of lists in time logarithmic in the size of lists. Similar to major and minor sorts in traditional indexes, partitioning and sorting keeps the edges in a sorted order, allowing systems to use fast intersection-based join algorithms, such as WCOJs [9] or sort-merge joins.

Example 3: Consider the following query that finds all 3-edge cyclical wire transfers involving Alice's account $v1$.

MATCH $a_1{-}[r_1{:}W]{->}a_2{-}[r_2{:}W]{->}a_3,\ a_3{-}[r_3{:}W]{->}a_1$
WHERE $a_1.ID{=}v1$

In systems that implement worst-case optimal join (WCOJ) algorithms, such as EmptyHeaded [12] or GraphflowDB [4], this query is evaluated by scanning each `v1`→a_2 `Wire` edge and intersecting the pre-sorted `Wire` lists of `v1` and a_2 to match the a_3 vertices.

To provide very fast access to each list, lists are often accessed through data structures that have constant depth, such as a CSR instead of logarithmic depths of traditional tree-based indexes. This is achieved by having one level in the index for each partitioning criterion, so levels in the index are not constrained to a fixed size unlike traditional indexes, e.g., k-ary trees. Some systems choose alternative implementations. For example Neo4j has a linked list-based implementation where edges in a list are not stored consecutively but have pointers to each other, or JanusGraph uses a pure adjacency list design where there is constant time access to all edges of a vertex. In our implementation of A+ indexes (explained in Section III), we use CSR as our core data structure to store adjacency lists because it is more compact than a pure adjacency list design and achieves better locality than a linked list one. Finally, we note that the primary shortcoming of adjacency list indexes in existing systems is that GDBMSs adopt fixed system-specific partitioning and possibly sorting criteria, which limits the workloads that can benefit from their fast join capabilities.

III. A+ Indexes

There are three types of indexes in our indexing subsystem: (i) primary A+ indexes; (ii) secondary vertex-partitioned A+ indexes; and (iii) secondary edge-partitioned A+ indexes. Each index, both in our solution and existing systems, stores a set of adjacency lists, each of which stores a set of edges. We refer to the edges that are stored in the lists as *adjacent* edges, and the vertices that adjacent edges point to as *neighbour* vertices.

A. Primary A+ Indexes

The primary A+ indexes are by default the only available indexes. Similar to primary B+ tree indexes of relations in relational systems, these indexes are required to contain each edge in the graph, otherwise the system will not be able to answer some queries. Similar to the adjacency lists of existing GDBMSs, there are two primary indexes, one forward and one backward, and we use a nested CSR data structure partitioned first by the source and destination vertex IDs of the edges, respectively. In our implementation, by default we adopt a second level partitioning by edge labels and sort the most granular lists according to the IDs of the neighbours, which optimizes the system for queries with edge labels and matching cyclic subgraphs using multiway joins computed through intersections of lists. However, unlike existing systems, users can reconfigure the secondary partitioning and sorting criteria of primary A+ indexes to tailor the system to variety of workloads, with no or very minimal memory overhead.

1) Tunable Nested Partitioning

A+ indexes can contain nested secondary partitioning criteria on any categorical property of adjacent edges as well as neighbour vertices, such as edge or neighbour vertex labels, or the `currency` property on the edges in our running example. In our implementation we allow integers or enums that are mapped to small number of integers as categorical values. Edges with null property values form a special partition.

Example 4: Consider querying all wire transfers made in USD currency from Alice's account and the destination accounts of these transfers:

MATCH $c_1{-}[r_1{:}O]{->}a_1{-}[r_2{:}W]{->}a_2$
WHERE $c_1.name = \text{`}Alice\text{'},\ r_2.currency{=}USD$

Here the query plans of existing systems that partition by edge labels will read all `Wire` edges from Alice's account and, for each edge, read its `currency` property and run a predicate to verify whether or not it is in USD.

Instead, if queries with equality predicates on the `currency` property are important and frequent for an application, users can reconfigure their primary A+ indexes to provide a secondary partitioning based on `currency`.

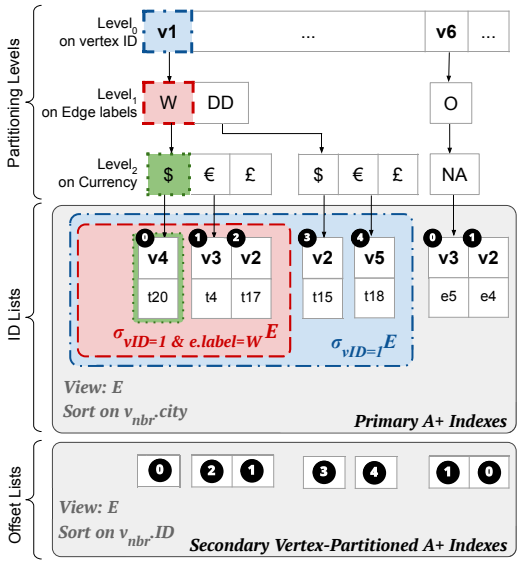
RECONFIGURE PRIMARY INDEXES
PARTITON BY $e_{adj}.label,\ e_{adj}.currency$
SORT BY $v_{nbr}.city$

In index creation and modification commands, we use reserved keywords e_{adj} and v_{nbr} to refer to adjacent edges and neighbours, respectively. The above command (ignore the sorting for now) will reconfigure the primary adjacency indexes to have two levels of partitioning after partitioning by vertex IDs: first by the edge labels and then by the `currency` property of these edges. For the query in Example 4, the system's join operator can now first directly access the lowest level partitioned lists of Alice's list, first by `Wire` and then by `USD`, without running any predicates.

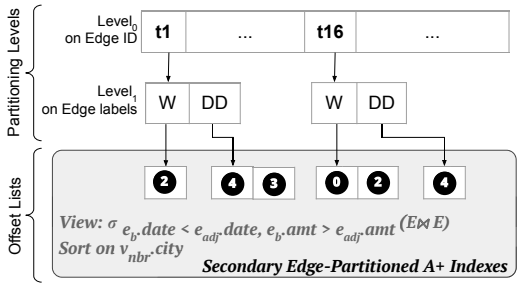
Figure 2a shows the final physical design this generates as an example on our running example. We store primary indexes in nested CSR structures. Each provided nested partitioning adds a new partitioning level to the CSR, storing offsets to a particular slice of the next layer. After the partitioning levels, at the lowest level of the index are *ID lists*, which store the IDs of the edges and neighbour vertices. The ID lists are a consecutive array in memory that contains a set of nested sublists. For example, consider the second level partitions of the primary index in Figure 2a. Let L_W, L_{DD}, and L be the list of `Wire`, `Dir-Deposit`, and all edges of a vertex v, respectively. Then within L, which is the list between indices 0-4, are sub-lists L_W (0-2) and L_{DD} (3-4), i.e., $L = L_W \cup L_{DD}$.

2) Tunable ID List Sorting

The most granular sublists can be sorted according to one or more arbitrary properties of the adjacent edges or neighbour vertices, e.g., the `date` property of `Transfer` edges and the `city` property of the `Account` vertices of our running example. Similar to partitioning, edges with null values on the



(a) Example primary adjacency lists and secondary vertex-partitioned adjacency lists.



(b) Example secondary edge-partitioned A+ Index.

Fig. 2: Example A+ indexes on our running example.

sorting property are ordered last. Secondary partitioning and sorting criteria together store the neighbourhoods of vertices in a particular sort order, allowing a system to generate WCOJ intersection-based plans for a wider set of queries.

Example 5: Consider the following query that searches for a three-branched money transfer tree, consisting of wire and direct deposit transfers, emanating from an account with `vID` `v5` and ending in three sink accounts in the same city.

$$\textit{MATCH}\ \ a_1{-}[{:}W]{->}a_2{-}[{:}W]{->}a_3\,,\ \ a_1{-}[{:}W]{->}a_4$$
$$a_1{-}[{:}DD]{->}a_5{-}[{:}DD]{->}a_6$$
$$\textit{WHERE}\ \ a_1.ID{=}v5,\ \ a_3.city{=}a_4.city{=}a_6.city$$

If `Wire` and `Dir-Deposit` lists are partitioned or sorted by `city`, as in the above reconfiguration command, after matching $a_1 \rightarrow a_2$ and $a_1 \rightarrow a_5$, a plan can directly intersect two `Wire` lists of a_1 and a_2 and one `Dir-Deposit` list of a_5 in a single operation to find the flows that end up in accounts in the same city. Such plans are not possible with the adjacency list indexes of existing systems.

Observe that the ability to reconfigure the system's primary A+ indexes provides more physical data independence. Users do not have to model their datasets according to the system's default physical design and changes in the workloads can be addressed simply with index reconfigurations.

B. Secondary A+ Indexes

Many indexes in DBMSs can be thought of as data structures that give fast access to *views*. In our context, each sublist in the primary indexes is effectively a *view* over edges. For example, the red dashed list in Figure 2a is the $\sigma_{\text{srcID=v1 \&e.label=Wire}}$`Edge` view while the green dotted box encloses a more selective view corresponding to $\sigma_{\text{srcID=1 \& e.label=wire \& curr=USD}}$`Edge`. Each nested sublist in the lowest-level ID lists is a view with one additional equality predicate. One can also think of the entire index as indexing a *global view*, which for primary indexes is simply the `Edge` table. Therefore the views that can be obtained through the system's primary A+ index are constrained to views over the edges that contain an equality predicate on the source or destination ID (due to vertex ID partitioning) and one equality predicate for each secondary partitioning criteria.

To provide access to even wider set of views, a system should support more general materialized views and index these in adjacency list indexes. However, supporting additional views and materializing them inside additional adjacency list indexes requires data duplication and storage. We next identify two classes of global views and ways to partition these views that are conducive to a space-efficient implementation: (i) 1-hop views that are stored in secondary vertex-partitioned A+ indexes; and (ii) 2-hop views that are stored in secondary edge-partitioned A+ indexes. These views and partitioning techniques generate lists that are subsets of the lists in the primary index, which allows us to store them in space-efficient *offset lists* that exploit the small average-degree of real-world graphs and use a few bytes per indexed edge. In Sections III-B1 and III-B2 we first describe our logical views and how these views are partitioned into lists. Similar to the primary A+ index, these lists are stored in CSR-based structures. Section III-B3 describes our offset list-based storage and how we can further increase the space efficiency of secondary A+ indexes by avoiding the partitioning levels of the CSR structure when possible.

1) Secondary Vertex-Partitioned A+ Indexes: 1-hop Views

Secondary vertex-partitioned indexes store 1-hop views, i.e., 1-hop queries, that contain arbitrary selection predicates on the edges and/or source or destination vertices of edges. These views cannot contain other operators, such as group by's, aggregations, or projections, so their outputs are a subset of the original edges. Secondary vertex-partitioned A+ indexes store these 1-hop views first by partitioning on vertex IDs (source or destination) and then by the further partitioning and sorting options provided by the primary A+ indexes. In order to use secondary vertex-partitioned A+ indexes, users need to first define the 1-hop view, and then define the partitioning structure and sorting criteria of the index.

Example 6: Consider a fraud detection application that searches money flow patterns with high amount of transfers, say over 10000 USDs. We can create a secondary vertex-partitioned index to store those edges in lists, partitioned first by vertices and then possibly by other properties and in a sorted manner as before.

CREATE 1–HOP VIEW $LargeUSDTrnx$
MATCH v_s−[e_{adj}]−>v_d
WHERE $e_{adj}.currency$=USD, $e_{adj}.amt$>10000
INDEX AS FW–BW
PARTITION BY $e_{adj}.label$ *SORT BY* $v_{nbr}.ID$

Above, v_s and v_d are keywords to refer to the source and destination vertices, whose properties can be accessed in the WHERE clause. FW and BW are keywords to build the index in the forward or backward direction, a partitioning option given to users. FW-BW indicates indexing in both directions. The inner-most (i.e., most nested) sublists of the resulting index materializes a view of the form $\sigma_{\text{srcID=* \& elabel=* \& curr=USD \& amount > 10000}}$Edge. If such views or views that correspond to other levels of the index appear as part of queries, the system can directly access these views in constant time and avoid evaluating the predicates in these views.

2) Secondary Edge-Partitioned A+ Indexes: 2-hop Views

Secondary edge-partitioned indexes store 2-hop views, i.e., results of 2-hop queries. As before, these views cannot contain other operators, such as group by's, aggregations, or projections, so their outputs are a subset of 2-paths. The view has to specify a predicate and that predicate has to access properties of both edges in 2-paths (as we momentarily explain, otherwise the index is redundant). Secondary edge-partitioned indexes store these 2-hop views first by partitioning *on edge IDs* and then, as before, by the same partitioning and sorting options provided by the primary A+ indexes. Vertex-partitioned indexes in A+ indexes and existing systems provide fast access to the adjacency of a vertex given the ID of that vertex. Instead, our edge-partitioned indexes provide fast access to the *adjacency of an edge* given the ID of that edge. This can benefit applications in which the searched patterns concern relations between two adjacent, i.e., consecutive, edges. We give an example:

Example 7: Consider the following query, which is the core of an important class of queries in financial fraud detection.

MATCH a_1−[r_1:]−>a_2−[r_2:]−>a_3−[r_3:]−>a_4
WHERE $r_1.eID$=t13,
$r_1.date$<$r_2.date$ & $r_2.amt$<$r_1.amt$<$r_2.amt$+α &
$r_2.date$<$r_3.date$ & $r_3.amt$<$r_2.amt$<$r_3.amt$+α

The query searches a three-step money flow path from a transfer edge with eID t13 where each additional transfer (Wire or Dir-Deposit) happens at a later date and for a smaller amount of at most α, simulating some money flowing through the network with intermediate hops taking cuts.

The predicates of this query compare properties of an edge on a path with the previous edge on the same path. Consider a system that matches r_1 to t13, which is from vertex v2 to v5. Existing systems have to read transfer edges from v5 and filter those that have a later date value than t13 and also have the appropriate amount value. Instead, when the next query edge to match r_2 has predicates depending on the query edge r_1, these queries can be evaluated much faster if adjacency lists are partitioned by edge IDs: a system can directly access the *destination-forward adjacency list of* t13 in constant time, i.e., edges whose srcID are v5, that satisfy the predicate on the amount and date properties that depend on t13, and perform the extension. Our edge-partitioned indexes allow the system to generate plans that perform this much faster processing. Note that in an alternative design we can partition the same set of 2-hop paths by vertices instead of edges. However, this would store the same number of edges but would be less efficient during query processing. To see this, suppose a system first matches r_1 to the edge (v_2)−[t13]−>(v_5) and consider extending this edge. The system can either extend this edge by one more edge to r_2, which would require looking up the 2-hop edges of v_2 and find those that have t13 as the first edge. This is slower than directly looking up the same edges using t13 in an edge-partitioned list. Alternatively the system can extend t13 by two more edges to $[r_2]$−>a_3−$[r_3]$−>a_4 by accessing the 2-hop edges of v_5 but would need to run additional predicates to check if the edge matching r_2 satisfy the necessary predicates with t13, so effectively processing all 2-paths of v_5 and running additional predicates, which are also avoided in an edge-partitioned list.

There are three possible 2-paths, →→, →←, and ←←. Partitioning these paths by different edges gives four unique possible ways in which an edge's adjacency can be defined:

1) Destination-FW: v_s−[e_b]→v_d−[e_{adj}]→v_{nbr}
2) Destination-BW: v_s−[e_b]→v_d←[e_{adj}]−v_{nbr}
3) Source-FW: v_{nbr}−[e_{adj}]→v_s−[e_b]→v_d
4) Source-BW: v_{nbr}←[e_{adj}]−v_s−[e_b]→v_d

e_b, for "bound", is the edge that the adjacency lists will be partitioned by, and v_s and v_d refer to the source and destination vertices of e_b, respectively. For example, the Destination-FW adjacency lists of edge $e(s,d)$ stores the forward edges of d. To facilitate the fast processing described above for the money flow queries in Example 7, we can create the following index:

CREATE 2–HOP VIEW MoneyFlow
MATCH v_s−[e_b]→v_d−[e_{adj}]→v_{nbr}
WHERE $e_b.date$<$e_{adj}.date$, $e_{adj}.amt$<$e_b.amt$
INDEX AS PARTITION BY $e_{adj}.label$ SORT BY $v_{nbr}.city$

The location of the variable e_b in the query implicitly defines the type of partitioning, which in this example is Destination-FW. This query creates an index that, for each edge t_i, stores the forward edges from t_i's destination vertex which have a later date and a smaller amount than t_i, partitioned by the labels of their adjacent edges and sorted by the city property of the neighbouring vertices, i.e., the vertex that is not shared with t_i. Figure 2b shows the lists this index stores on our running example. The inner-most lists in the index correspond to the view: $\sigma_{\text{e}_b\text{.ID=* \& e}_{adj}\text{.label=* \& e}_b\text{.date < e}_{adj}\text{.date \& e}_b\text{.amt > e}_{adj}\text{.amt}}(\rho_{e_b}(E) \bowtie \rho_{e_{adj}}(E))$. E abbreviates Edge and the omitted join predicate is e_b.dstID=e_{adj}.srcID. Readers can verify that, in presence of this index, a GDBMS can evaluate the money flow query from Example 4 (ignoring the predicate with α) by scanning only one edge. It only scans t13's list which contains a single edge t19. In contrast, even if all Transfer edges are accessible using a vertex-partitioned A+ index, a system

would access 9 edges after scanning t13.

Observe that unlike vertex-partitioned A+ indexes, an edge e in the graph can appear in multiple adjacency lists in an edge-partitioned index. For example, in Figure 2b, edge t17 (having offset 2) appears both in the adjacency list for t1 as well as t16. As a consequence, when defining edge-partitioned indexes, users have to specify a predicate that accesses properties of both edges in the 2-hop query. This is because if all the predicates are only applied to a single query edge, say $v_s{-}[e_b]{\rightarrow}v_d$, then we would redundantly generate duplicate adjacency lists. Instead, defining a secondary vertex-partitioned A+ index would give the same access path to the same lists without this redundancy. The longer version of our paper [10] demonstrates this with an example.

3) Offset List-based Storage of Secondary A+ Indexes

The predominant memory cost of primary indexes is the storage of the IDs of the adjacent edges and neighbour vertices. Because the IDs in these lists globally identify vertices and edges, their sizes need to be logarithmic in the number of edges and vertices in the graph, and are often stored as 4 to 8 byte integers in systems. For example, in our implementation, edge IDs take 8 and neighbour IDs take 4 bytes.

In contrast, the lists in both secondary vertex- and edge-partitioned indexes have an important property, which can be exploited to reduce their memory overheads: they are subsets of some ID list in the primary indexes. Specifically, a list L_v that is bound to v_i in a secondary vertex-partitioned index is a subset of one of v_i's ID lists. A list L_e that is bound to $e = (v_s, v_d)$ in a secondary edge-partitioned index is a subset of either v_s's or v_d's primary list, depending on the direction of the index, e.g., v_d's list for a Destination-FW list. Recall that in our CSR-based implementation, the ID lists of each vertex are contiguous. Therefore, instead of storing an (edge ID, neighbour ID) pair for each edge, we can store a single offset to an appropriate ID list. We call these lists *offset lists*. The average size of the ID lists is proportional to the average degree in the graph, which is often very small, in the order of tens or hundreds, in many real world graph data sets. This important property of real world graphs has two advantages:

1) Offsets only need to be list-level identifiable and can take a small number of bytes which is much smaller than a globally identifiable (edge ID, neighbour ID) pair.
2) Reading the original (edge ID, neighbour ID) pairs through offset lists require an indirection and lead to reading not-necessarily consecutive locations in memory. However, because the ID list sizes are small, we still get very good CPU cache locality.

An alternative implementation design here is to use a bitmap instead of offset lists. A bitmap can identify whether each edge in the lists of the primary A+ index is a secondary A+ index. This design has the shortcoming that it cannot support the cases when the sorting criterion of secondary A+ indexes is different than the primary index. However when the sorting criteria are the same, this is also a reasonable design point. This has the advantage that when the predicates in the lists are not very selective, bitmaps can be even more compact than offset lists, as they require a single bit for each edge. However reading the edges would now require additional bitmask operations. In particular, irrespective of the actual number of edges stored in a secondary index, the system would need to perform as many bitmask operations as the number of edges in the lists of the primary index. Therefore as predicates in secondary indexes get more selective, bitmaps would progressively lose their storage advantage over offset lists and at the same time progressively perform worse in terms of access time.

We implement each secondary index in one of two possible ways, depending on whether the index contains any predicates and whether its partitioning structure matches the secondary structure of the primary A+ indexes.

- *With no predicates and same partitioning structure*: In this case, the only difference between the primary and the secondary index is the final sorting of the edges. Specifically, both indexes have identical partitioning levels, with identical CSR offsets, and the same set of edges in each inner-most ID/offset sublists, but they sort these sublists in a different order. Therefore we can use the partitioning levels of the primary index also to access the lists of the secondary index and save space. Figure 2a gives an example. The bottom offset lists are for a secondary vertex-partitioned index that has the same partitioning structure as the primary index but sorts on neighbors' IDs instead of neighbors' city properties. Recall that since edge-partitioned indexes need to contain predicates between adjacent edges, this storage can only be used for vertex-partitioned indexes.
- *With predicates or different partitioning structure*: In this case, the inner-most sublists of the indexes may contain different sets of edges, so the CSR offsets in the partitioning levels of the primary index cannot be reused and we store new partitioning levels as shown in Figure 2b.

We give the details of the memory page structures that store ID and offset lists in Section IV.

IV. Implementation Details

We implemented our indexing subsystem in GraphflowDB [4], [5] and describe our changes to the system to enable the use of A+ indexes for fast join processing.

A. Query Processor, Optimizer and Index Store

A+ indexes are used in evaluating subgraph pattern component of queries, which is where the queries' joins are described. We give an overview of the join operators that use A+ indexes and the optimizer of the system. Reference [4] describes the details of the EXTEND/INTERSECT operator and the DP join optimizer of the system in absence of A+ indexes.

JOIN OPERATORS: EXTEND/INTERSECT (E/I) is the primary join operator of the system. Given a query $Q(V_Q, E_Q)$ and an input graph $G(V, E)$, let a *partial k-match* of Q be a set of vertices of V assigned to the projection of Q onto a set of k query vertices. We denote a sub-query with k query vertices as Q_k. E/I is configured to intersect $z{\geq}1$ adjacency

lists that are sorted on neighbour IDs. The operator takes as input (k-1)-matches of Q, performs a z-way intersection, and extends them by a single query vertex to k-matches. For each (k-1)-match t, the operator intersects z adjacency lists of the matched vertices in t and extends t with each vertex in the result of this intersection to produce k-matches. If z is one, no intersection is performed, and the operator simply extends t to each vertex in the adjacency list. The system uses E/I to generate plans that contain WCOJ multi-way intersections.

To generate plans that use A+ indexes, we first extended E/I to take adjacency lists that can be partitioned by edges as well as vertices. We also added a variant of E/I that we call MULTI-EXTEND, that performs intersections of adjacency lists that are sorted by properties other than neighbour IDs and extends partial matches to more than one query vertex.

Dynamic Programming (DP) Optimizer and INDEX STORE: GraphflowDB has a DP-based join optimizer that enumerates queries one query vertex at a time [4]. We extended the system's optimizer to use A+ indexes as follows. For each $k{=}1, ..., m{=}|V_Q|$, in order, the optimizer finds the lowest-cost plan for each sub-query Q_k in two ways: (i) by considering extending every possible sub-query Q_{k-1}'s (lowest-cost) plan by an E/I operator; and (ii) if Q has an equality predicate involving $z{\geq}2$ query edges, by considering extending smaller sub-queries Q_{k-z} by a MULTI-EXTEND operator. At each step, the optimizer considers the edge and vertex labels and other predicates together, since secondary A+ indexes may be indexing views that contain predicates other than edge label equality. When considering possible Q_{k-z} to Q_k extensions, the optimizer queries the INDEX STORE to find both vertex- and edge-partitioned indexes, $I_1, ..., I_t$, that can be used. INDEX STORE maintains the metadata of each A+ index in the system such as their type, partitioning structure, and sorting criterion, as well as additional predicates for secondary indexes. An index I_ℓ can potentially be used in the extension if the edges in the lists in a level j of I_ℓ satisfy two conditions: (i) extend partial matches of Q_{k-z} to Q_k, i.e., can be bound to a vertex or edge in Q_{k-z} and match a subset of the query edges in Q_z; and (ii) the predicates $p_{\ell,j}$ satisfied in these lists subsume the predicate p_Q (if any) that is part of this extension. We search for two types of predicate subsumption. First is conjunctive predicate subsumption. If both $p_{\ell,j}$ and p_Q are conjunctive predicates, we check if each component of $p_{\ell,j}$ matches a component of p_Q. Second is range subsumption. If p_Q and $p_{\ell,j}$ or one of their components are range predicates comparing a property against a constant, e.g., $e_{adj}.amt{>}15000$ and $e_{adj}.amt{>}10000$, respectively, we check if the range in $p_{\ell,j}$ is less selective than p_Q.

Then for each possible index combination retrieved, the optimizer enumerates a plan for Q_k with: (i) an E/I or MULTI-EXTEND operator; and (ii) possibly a FILTER operator if there are any predicates that are not fully satisfied during the extension (e.g., if $p_{\ell,j}$ and p_Q are conjunctive but $p_{\ell,j}$ does not satisfy all components of p_Q). If the Q_{k-z} to Q_k extension requires using multiple indices, so requires performing an intersection, then the optimizer also checks that the sorting criterion on the indices that are returned are the same. Otherwise, it discards this combination. The systems' cost metric is *intersection cost* (i-cost), which is the total estimated sizes of the adjacency lists that will be accessed by the E/I and MULTI-EXTEND operators in a plan.

We note that our optimizer extension to use A+ indexes is similar to the classic *System R-style* approach to enumerate plans that use views composed of select-project-join queries directly in a DP-based join optimizer [13], [14]. This approach also performs a bottom up DP enumeration of join orders of a SQL query Q and for a sub-query Q' of Q, considers evaluating Q' by joining a smaller Q'' with a view V. The primary difference is that GraphflowDB's join optimizer enumerates plans for progressively larger queries that contain, in relational terms, one more column instead of one more table (see reference [4] for details). Other GDBMSs that use bottom up join optimizers can be extended in a similar way if they implement A+ indexes. For example, Neo4j also uses a mix of DP and greedy bottom up enumerator [1] called *iterative DP*, which is based on reference [15]. However, extending the optimizers of GDBMSs that use other techniques might require other approaches, e.g., RedisGraph, which converts Cypher queries into GraphBLAS linear algebra expression [16] and optimizes this expression.

We also note that we implemented a limited form of predicate subsumption checking. The literature on query optimization using views contains more general techniques for logical implication of predicates between queries and views [14], [17], [18], e.g., detecting that $A > B$ and $B > C$ imply $A > C$. These techniques can enhance our implementation and we have not integrated such techniques within the scope of our paper.

B. Details of Physical Storage

Primary and secondary vertex-partitioned A+ indexes are implemented using a CSR for groups of 64 vertices and allocates one data page for each group. Vertex IDs are assigned consecutively starting from 0, so given the ID of v, with a division and mod operation we can access the second partitioning level of the index storing CSR offsets of v. The CSR offsets in the final partitioning level point to either ID lists in the case of the primary A+ indexes or offset lists in the case of secondary A+ indexes. The neighbour vertex and edge ID lists are stored as 4 byte integer and 8 byte long arrays, respectively. In contrast, the offset lists in both cases are stored as byte arrays by default. Offsets are fixed-length and use the maximum number of bytes needed for any offset across the lists of the 64 vertices, i.e. it is the logarithm of the length of the longest of the 64 lists rounded to the next byte.

C. Index Maintenance

Each vertex-partitioned data page, storing ID lists or offset lists, is accompanied with an update buffer. Each edge addition $e{=}(u, v)$ is first applied to the update buffers for u's and v's pages in the primary indexes. Then we go over each secondary vertex-partitioned A+ index I_V in the INDEX STORE. If I_V indexes a view that contains a predicate p, we first apply p to see if e passes the predicate. If so, or if I_V does not contain a

predicate, we update the necessary update buffers for the offset list pages of u and/or v. The update buffers are merged into the actual data pages when the buffer is full. Edge deletions are handled by adding a "tombstone" for the location of the deletion until a merge is triggered.

Maintenance of an edge-partitioned A+ index I_E is more involved. For an edge insertion $e{=}(u, v)$, we perform two separate operations. First, we check to see if e should be inserted into the adjacency list of any adjacent edge e_b by running the predicate p of I_E on e and e_b. For example, if I_E is defined as Destination-FW, we loop through all the backward adjacent edges of u using the system's primary index. This is equivalent to running two *delta-queries* as described in references [5], [19] for a continuous 2-hop query. Second, we create a new list for e and loop through another set of adjacency lists (in our example v's forward adjacency list in D) and insert edges into e's list.

D. Index Selection

Our work focuses on the design and implementation of a tunable indexing subsystem so that users can tailor a GDBMS to be highly efficient on a wide range of workloads. However, an important aspect of any DBMS is to help users pick indexes from a space of indexes that can benefit their workloads. Given a workload W, the space of A+ indexes that can benefit W can be enumerated by enumerating each 1-hop and 2-hop sub-query Q' of each query Q in W and identifying the equality predicates on categorical properties of these sub-queries, which are candidates for partitioning levels, and non-equality predicates on other properties, which are candidates for sorting criterion (any predicate is also a candidate predicate of a global view). Given a workload W and possibly a space budget B, one approach from prior literature to automatically select a subset of these candidate indices that are within the space budget B is to perform a "what if" index simulation to see the effects of this candidate indices on the estimated costs of plans. For example, this general approach is used in Microsoft SQL Server's AutoAdmin [20] tool. We do not focus on the problem of recommending a subset of these indexes to users. There are several prior work on index and materialized view recommendation [20], [21], [22], [23], [24], which are complementary to our work. We leave the rigorous study of this problem to future work.

V. Evaluation

The goal of our experiments is two-fold. First, we demonstrate the tunability and space-efficiency of A+ indexes on three very different popular applications that GDBMSs support: (i) labelled subgraph queries; (ii) recommendations; and (iii) financial fraud detection. By either tuning the system's primary A+ index or adding secondary A+ indexes, we improve the performance of the system significantly, with minimal memory overheads. Second, we evaluate the performance and memory overhead tradeoffs of different A+ indexes on these workloads. Finally, as a baseline comparison, we benchmark our performance against Neo4j [1] and TigerGraph [3], two commercial GDBMSs that have fixed adjacency list structures.

Name	#Vertices	#Edges	Avg. degree
Orkut (Ork)	3.0M	117.1M	39.03
LiveJournal (LJ)	4.8M	68.5M	14.27
Wiki-topcats (WT)	1.8M	28.5M	15.83
BerkStan (Brk)	685K	7.6M	11.09

TABLE I: Datasets used.

For completeness of our work, we also evaluate the maintenance performance of our indexes in the longer version of our paper [10].

A. Experimental Setup

We use a single machine with two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical ones. Table I shows the datasets used. We ran our experiments on all datasets and report numbers on a subset of datasets due to limited space. Our datasets include social, web, and Wikipedia knowledge graphs, which have a variety of graph topologies and sizes ranging from several million edges to over a hundred-million edges. A dataset G, denoted as $G_{i,j}$, has i and j randomly generated vertex and edge labels, respectively. We omit i and j when both are set to 1. We use query workloads drawn from real-world applications: (i) edge- and vertex-labelled subgraph queries; (ii) Twitter MagicRecs recommendation engine [25]; and (iii) fraud detection in financial networks. For all index configurations (Configs), we report either the index reconfiguration (IR) or the index creation (IC) time of the newly added secondary indexes. All experiments use a single thread except the creation of edge-partitioned indexes, which uses 16 threads.

B. Primary A+ Index Reconfiguration

We first demonstrate the benefit and overhead tradeoff of tuning the primary A+ index in two different ways: (i) by only changing the sorting criteria; and (ii) by adding a new secondary partitioning. We used a popular subgraph query workload in graph processing that consists of labelled subgraph queries where both edges and vertices have labels. We followed the data and subgraph query generation methodology from several prior work [4], [26]. We took the 14 queries from reference [4] (omitted due to space reasons), which contain acyclic and cyclic queries with dense and sparse connectivity with up to 7 vertices and 21 edges. This query workload had only fixed edge labels in these queries, for which GraphflowDB's default indexes are optimized. We modify this workload by also fixing vertex labels in queries. We picked the number of labels for each dataset to ensure that queries would take time in the order of seconds to several minutes. Then we ran GraphflowDB on our workload on each of our datasets under three Configs:

1) D: system's default configuration, where edges are partitioned by edge labels and sorted by neighbour IDs.
2) D_s: keeps D's secondary partitioning but sorts edges first by neighbour vertex labels and then on neighbour IDs.
3) D_p: keeps D's sorting criteria and edge label partitioning but adds a new secondary partitioning on neighbour vertex labels.

Table II shows our results. We omit Q14, which had very few or no output tuples on our datasets. First observe that D_s

		SQ_1	SQ_2	SQ_3	SQ_4	SQ_5	SQ_6	SQ_7	SQ_8	SQ_9	SQ_{10}	SQ_{11}	SQ_{12}	SQ_{13}	Mm	IR
$Ork_{8,2}$	D	1.68	5.47	3.66	1.30	1.58	1.45	1.73	2.49	0.95	17.74	7536.9	54.86	131.5	2778	-
	D_s	0.91	3.12	2.04	1.19	1.05	1.22	1.33	1.51	0.77	4.89	725.9	41.92	55.62	2778	38.90
		(1.85x)	**(1.75x)**	**(1.79x)**	**(1.09x)**	**(1.50x)**	**(1.19x)**	**(1.30x)**	**(1.65x)**	**(1.23x)**	**(3.63x)**	**(10.38x)**	**(1.31x)**	**(2.36x)**	**(1.0x)**	-
	D_p	0.68	2.61	1.35	0.97	0.77	0.60	1.30	1.46	0.60	3.89	704.9	28.32	34.22	3106	27.71
		(2.48x)	**(2.10x)**	**(2.71x)**	**(1.34x)**	**(2.05x)**	**(2.44x)**	**(1.33x)**	**(1.71x)**	**(1.25x)**	**(4.56x)**	**(10.69x)**	**(1.94x)**	**(3.84x)**	**(1.12x)**	-
$LJ_{2,4}$	D	1.47	7.87	6.46	1.69	1.59	1.60	1.91	3.35	4.07	41.54	807.8	397.1	468.8	1016	-
	D_s	1.45	6.22	5.42	1.49	1.51	1.52	1.40	2.39	2.82	28.07	241.2	268.6	259.2	1016	20.83
		(1.01x)	**(1.27x)**	**(1.19x)**	**(1.13x)**	**(1.05x)**	**(1.05x)**	**(1.36x)**	**(1.40x)**	**(1.44x)**	**(1.48x)**	**(3.35x)**	**(1.48x)**	**(1.81x)**	**(1.0x)**	-
	D_p	1.04	5.18	4.64	1.09	0.98	1.08	1.07	1.85	2.26	25.86	235.63	235.85	161.82	1164	19.92
		(1.41x)	**(1.52x)**	**(1.39x)**	**(1.55x)**	**(1.62x)**	**(1.48x)**	**(1.79x)**	**(1.81x)**	**(1.80x)**	**(1.61x)**	**(3.43x)**	**(1.68x)**	**(2.90x)**	**(1.15x)**	-
$WT_{4,2}$	D	0.61	4.59	5.48	0.84	1.17	0.90	0.73	11.25	2.85	1116.2	340.0	487.8	767.5	713	-
	D_s	0.37	2.43	3.50	0.69	0.71	0.65	0.61	3.93	1.36	697.9	77.11	319.0	386.8	713	8.70
		(1.65x)	**(1.89x)**	**(1.56x)**	**(1.22x)**	**(1.65x)**	**(1.38x)**	**(1.20x)**	**(2.87x)**	**(2.09x)**	**(1.60x)**	**(4.41x)**	**(1.53x)**	**(1.98x)**	**(1.0x)**	-
	D_p	0.32	2.09	3.05	0.55	0.59	0.54	0.61	2.86	1.09	639.7	76.32	259.1	235.7	795	6.25
		(1.91x)	**(2.20x)**	**(1.80x)**	**(1.53x)**	**(1.99x)**	**(1.66x)**	**(1.21x)**	**(3.94x)**	**(2.62x)**	**(1.74x)**	**(4.45x)**	**(1.88x)**	**(3.26x)**	**(1.12x)**	-

TABLE II: Runtime (secs) and memory usage in MBs (Mm) evaluating subgraph queries using three different index configurations: D, D_s, and D_p introduced in Section V-B. We report index reconfiguration (IR) time (secs).

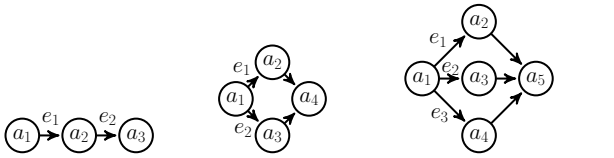
outperforms D on all of the 52 settings and by up to 10.38x and without any memory overheads as D_s simply changes the sorting criteria of the indexes. Next observe that by adding an additional partitioning level on D, the joins get even faster consistently across all queries, e.g., SQ_{13} improves from 2.36x to 3.84x on $Ork_{8,2}$, as the system can directly access edges with a particular edge label and neighbour label using D_p. In contrast, under D_s, the system performs binary searches inside lists to access the same set of edges. Even though D_p is a reconfiguration, so does not index new edges, it still has minor memory overhead ranging from 1.05x to 1.15x because of the cost of storing the new partitioning layer. This demonstrates the effectiveness of tuning A+ indexes to optimize the system to be much more efficient on a different workload without any data remodelling, and with no (or minimal) memory overhead.

C. Secondary Vertex-Partitioned A+ Indexes

We next study the tradeoffs offered by secondary vertex-partitioned A+ indexes. We use two sets of workloads drawn from real-world applications that benefit from using both the system's primary A+ index as well as a secondary vertex-partitioned A+ index. Our two applications highlight two separate benefits users get from vertex-partitioned A+ indexes: (i) decreasing the amount of predicate evaluation; and (ii) allowing the system to generate new WCOJ plans that are not possible with only the primary A+ index.

1) Decreasing Predicate Evaluations

In this experiment, we take a set of the queries drawn from the MagicRecs workload described in reference [25]. MagicRecs was a recommendation engine that was developed at Twitter that looks for the following patterns: for a given user a_1, it searches for users $a_2...a_k$ that a_1 has started following recently, and finds their common followers. These common followers are then recommended to a_1. We set k=2,3 and 4. Our queries, MR_1, MR_2, and MR_3 are shown in Figure 3. These queries have a time predicate on the edges starting from a_1 which can benefit from indexes that sort on time. MR_2, and MR_3 are also cyclic can benefit from the default sorting order of the primary A+ index on neighbour IDs. We evaluate our queries on all of our datasets on two Configs. The first



$P_\alpha(e_1)$,$P_\alpha(e_2)$ $P_\alpha(e_1)$,$P_\alpha(e_2)$ $P_\alpha(e_1)$,$P_\alpha(e_2)$,$P_\alpha(e_3)$

(a) MR_1. (b) MR_2. (c) MR_3.

Fig. 3: MagicRec (MR) queries. $P_\alpha(e_i)$ = e_i.time $< \alpha$

Config consists of the system's primary A+ index denoted by D as before. The second Config, denoted by $D+VP_t$ adds on top of D a new secondary vertex-partitioned index VP_t in the forward direction that: (i) has the same partitioning structure as primary forward A+ index so shares the same partitioning levels as the primary A+ index; and (ii) sorts the inner-most lists on the time property of edges. In our queries we set the value of α in the time predicate to have a 5% selectivity. For MR_3, on datasets LJ and Ork, we fix a_1 to 10000 and 7000 vertices, respectively, to run the query in a reasonable time.

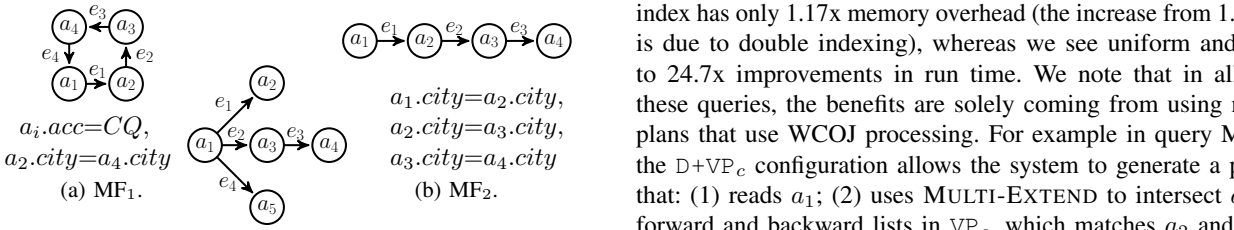
Table III shows our results. First observe that despite indexing all of the edges again, our secondary index has only 1.08x memory overhead because: (i) the secondary index can share the partitioning levels of the primary index; and (ii) the secondary index stores offset lists which has a low-memory footprint. In return, we see up to 10.6x performance benefits. We note that GraphflowDB uses exactly the same plans under both Configs that start reading a_1, extends to its neighbours and finally performs a multiway intersection (except for MR_1, which is followed by a simple extension). The only difference is that under $D+VP_t$ the first set of extensions require fewer predicate evaluation because of accessing a_1's adjacency list in VP_t, which is sorted on time. Overall this memory performance tradeoff demonstrates that with minimal overheads of an additional index, users obtain significant performance benefits on applications like MagicRecs that require performing complex joins for which the system's primary indexes are not tuned.

2) WCOJ Plans

We next evaluate the benefit and overhead tradeoff of secondary vertex-partitioned indexes when the secondary index

		MR_1	MR_2	MR_3	Mm	IC
Ork	D	29.37	255.4	22.65	2755	-
	$D+VP_t$	14.36(**2.0x**)	166.3(**1.5x**)	3.33(**6.8x**)	2982(**1.1x**)	42.10
LJ	D	18.19	38.17	842.8	1689	-
	$D+VP_t$	8.83(**2.1x**)	27.26(**1.4x**)	79.72(**10.6x**)	1820(**1.1x**)	21.79
WT	D	6.87	9.67	136.5	700	-
	$D+VP_t$	2.69(**2.6x**)	5.36(**1.8x**)	22.74(**6.0x**)	755(**1.1x**)	9.14

TABLE III: Runtime (secs) and memory usage in MBs (Mm) evaluating MagicRec queries using Configs: D and $D+VP_t$ introduced in Section V-C1. We report index creation (IC) time (secs) for secondary indexes.



(a) MF_1. (b) MF_2.

$a_2.city=a_4.city,\ a_4.city=a_5.city,\ a_3.ID<10000,$
$a_i.acc=CQ,\ a_5.acc=SV,\ P_f(e_2,e_3)$

(c) MF_3.

$a_1.city=\beta,\ a_2.city=a_4.city,\ a_2.acc=CQ,\ a_3.acc=CQ,$
$a_4.acc=SV,\ a_5.acc=SV,\ P_f(e_1,e_2),\ P_f(e_3,e_4)$

(d) MF_4.

$a_1.ID<50000,\ a_i.acc=CQ,\ P_f(e_1,e_2),P_f(e_2,e_3),P_f(e_3,e_4)$

(e) MF_5.

Fig. 4: Fraud detection queries. $P_f(e_i,e_j)$ defined as $e_i.date<e_j.date$, $e_i.amt>e_j.amt$, $e_i.amt<e_j.amt+\alpha$.

allows the system to generate new WCOJ plans that are not in the plan space with primary indexes only. We take a set of queries drawn from cyclic fraudulent money flows reported in prior literature [27], as well as acyclic patterns that contain the money flow paths from our running examples. Figure 4 shows our queries MF_1, ..., MF_5. We focus on MF_1 to MF_4 here and use MF_5 in the next section. These four queries have equality conditions on the `city` property of the vertices, so can benefit from multiway joins computed by intersecting lists that are presorted on `city`. We evaluate these queries on two Configs. The first Config consists of the system's primary A+ index denoted by D as before. The second Config, denoted by $D+VP_c$ adds on top of D a new secondary vertex-partitioned index VP_c in both forward and backward directions that: (i) has the same partitioning structure as primary A+ indexes; and (ii) sorts the inner-most lists on neighbour's `city` property. For each dataset, we randomly added each vertex an account type property from [CQ, SV], a city from 4417 cities, and to each edge an amount in the range of [1, 1000] and a date within a 5 year range.

Table IV shows our results (ignore the MF_5 column and the $D+VP_c+EP_c$ rows for now). Similar to our previous experiment, despite indexing all of the edges (this time twice), our secondary index has only 1.17x memory overhead (the increase from 1.08x is due to double indexing), whereas we see uniform and up to 24.7x improvements in run time. We note that in all of these queries, the benefits are solely coming from using new plans that use WCOJ processing. For example in query MF_1, the $D+VP_c$ configuration allows the system to generate a plan that: (1) reads a_1; (2) uses MULTI-EXTEND to intersect a_1's forward and backward lists in VP_c, which matches a_2 and a_4; and (3) uses E/I that intersects a_2's forward and a_4's backward lists in the primary A+ index to match the a_3's. Such plans are not possible in absence of the VP_c index. Instead for MF_1, under the default configuration D, the system extends a_1 to a_2, then to a_3 separately, runs a FILTER operator to match the cities, and then uses E/I to match the a_3's.

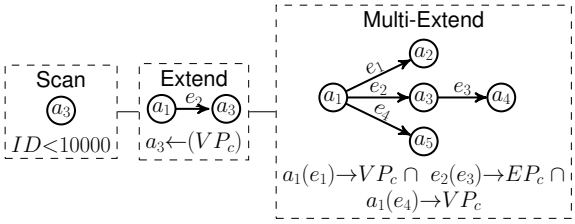


Fig. 5: WCOJ Plan for MF_3 from Figure 4c using two VP_c indexes and one EP_c index from Sections V-C2 and V-D.

D. Secondary Edge-Partitioned A+ Indexes

Finally, we evaluate the tradeoffs of our secondary edge-partitioned A+ indexes on our financial fraud application. We add a third Config to our experiment denoted by $D+VP_c+EP_c$. The configuration adds the edge-partitioned index from Example 7 in Section III-B2. We change the second-level partitioning to be on $v._{adj}$.acc instead of edge labels and add the predicate $e._b$.amt < $e._{nbr}$.amt + α. We pick the "intermediate cut" value α in our examples to have 5% selectivity.

Table IV shows our results. First we observe that the addition of EP_c only allows new plans to be generated for MF_3, MF_4 and MF_5, so we report numbers only for these queries. The improvements in run time range from 6.14x to up to 72.2x for a 2.22x memory overhead. Naturally the memory and performance tradeoff will change with the selectivity of α. What is more important to note is that the speedups are primarily due to the system producing significantly more efficient plans in the presence of the EP_c index. For example, the system now generates a highly complex plan for MF_3, shown in Figure 5, that uses a mix of vertex and edge-partitioned indexes and performs a 3-way intersection.

E. Neo4j and TigerGraph Comparisons

We next compare GraphflowDB to Neo4j and TigerGraph. These experiments are provided for completeness only. These are full-fledged commercial systems that support transactions. However Neo4j is perhaps the most popular existing GDBMS and TigerGraph, to the best of our knowledge, is the most performant one in terms of read performance.Our goal is to and show that the benefits of A+ indexes reported are on top of a system that is already competitive with existing GDBMSs.

		MF_1	MF_2	MF_3	MF_4	MF_5	Mem(MB)	$\|E_{indexed}\|$	IC
Ork	D	73.35	5.53	32.85	71.46	890.8	2730	117.1M	-
	D+VP$_c$	8.99 (**8.16x**)	2.75 (**2.01x**)	1.33 (**24.7x**)	19.03 (**3.76x**)	—	3183 (**1.17x**)	117.1M	85.83
	D+VP$_c$+EP$_c$	—	—	0.56 (**58.7x**)	0.99 (**72.2x**)	60.59 (**14.7x**)	6000 (**2.20x**)	513.2M	288.4
LJ	D	47.09	4.24	84.78	7.60	52.04	1649	68.5M	-
	D+VP$_c$	11.45 (**4.11x**)	2.86 (**1.48x**)	5.12 (**16.6x**)	3.66 (**2.08x**)	—	1910 (**1.16x**)	68.5M	46.43
	D+VP$_c$+EP$_c$	—	—	2.16 (**39.3x**)	0.39 (**19.5x**)	5.79 (**8.99x**)	3585 (**2.17x**)	276.2M	279.8
WT	D	20.27	1.47	9.02	0.86	9.02	685	28.5M	-
	D+VP$_c$	2.29 (**8.85x**)	1.12 (**1.31x**)	1.55 (**5.82x**)	0.53 (**1.62x**)	—	796 (**1.16x**)	28.5M	21.26
	D+VP$_c$+EP$_c$	—	—	0.50 (**18.0x**)	0.14 (**6.14x**)	0.79 (**11.4x**)	1521 (**2.22x**)	125.4M	843.5

TABLE IV: Runtime (secs) of GraphflowDB plans and memory usage (Mem) in MB evaluating fraud detection queries using different Configs: D, D+VP$_c$, and D+VP$_c$+EP$_c$ introduced in Section V-C2. The run time speedups and memory usage increase shown in parenthesis are in comparison to D. We report index creation time (IC) in secs for secondary indexes.

	$LJ_{12,2}$				$WT_{4,2}$			
	SQ_1	SQ_2	SQ_3	SQ_{13}	SQ_1	SQ_2	SQ_3	SQ_{13}
D	**0.4**	1.4	1.1	31.3	0.6	4.6	5.5	767.5
D_p	**0.4**	**0.7**	**0.6**	**6.0**	**0.3**	**2.1**	**3.1**	235.7
TG	2.5	11.8	15.2	30.5	1.6	7.1	10.2	**29.5**
N4	29.3	35.3	36.8	TL	1.65k	876	82.9	TL

TABLE V: Runtime (secs) of GraphflowDB on Configs D and D_p introduced in Section V-B, runtime of TigerGraph (TG), and runtime of Neo4j (N4). TL indicates >30 mins.

We report numbers for four of our labelled subgraph queries SQ_1, SQ_2, SQ_3, and SQ_{13} on $LJ_{12,2}$ and $WT_{4,2}$ on Neo4j and TigerGraph, using their default Configs and using the D and D_p configurations from Section V-B for GraphflowDB. Table V shows our results. We found GraphflowDB to be faster on all queries on the D configuration except for SQ_{13} on $WT_{4,2}$. In addition, similar to our experiments from Table II, the D_p configuration makes GraphflowDB even more performant. TigerGraph was the fastest system on SQ_{13}, which is a long 5-edge path. We cannot inspect the source code but we suspect for paths TigerGraph extends each distinct intermediate node only once and they only report pairs of reachable nodes. However, note that using the reconfigured index D_p, GraphflowDB outperforms TigerGraph on $LJ_{12,2}$ and closes the gap on $WT_{4,2}$. It is important to note that neither of these systems has a mechanism for tuning through index reconfiguration or construction to close their performance gaps.

VI. Related Work

View-based Query Processing: Answering queries using views has been well studied in the context of relational, XML, or RDF data management. We refer the reader to several surveys and references on the topic [13], [28], [29]. This extensive literature studies numerous topics, such as rewriting queries using a set of views [30], selecting a set of views for a workload e.g., web databases [31], or the computational complexities of deciding whether a query can be answered with a given set of views [32]. In this work, we observed that the lists that are stored in the adjacency list indexes can be seen as views and systems provide fast access to these lists/views through CSR-like data structures. In contrast to prior work, we explored how to extend the views that can be accessed through adjacency list indexes in a space-efficient manner. Specifically, we identified a restricted but still much larger set of views than existing indexes, that can be stored by either merely tuning the partitioning schemes of a multi-level CSR data structure or lightweight offset lists.

Kaskade [33] (KSK) is a graph query optimization framework that uses *materialized graph views* to speed up query evaluation. Specifically, KSK takes as input a query workload $\mathcal{Q}$ and an input graph G. Then, KSK enumerates possible *views* for $\mathcal{Q}$, which are other graphs G' that contain a subset of the vertices in G and other edges that can represent multi-hop connections in G. For example, if G is a data provenance graph with job and file vertices, and there are "consumes" and "produces" relationships between jobs and files, an example graph view G' could store the job vertices and their 2-hop dependencies through files. KSK materializes its selected views in Neo4j, and then translates queries over G to appropriate graphs (views) that are stored in Neo4j, which is used to answer queries. Therefore, the framework is limited by Neo4j's adjacency lists.

There are several differences between the views provided by KSK and A+ indexes. First, KSK's views are based on "constraints" that are mined from G's schema based only on vertex/edge labels and not properties. For example, KSK can mine "job vertices connect to jobs in 2-hops but not to file vertices" constraints but not "accounts connect to accounts in 2-hops with later dates and lower amounts", which can be a predicate in an A+ index. Second, because KSK stores its views in Neo4j, KSK views are only vertex ID and edge label partitioned, unlike our views which are stored in a CSR data structure that support tunable partitioning, including by edge IDs, as well as sorting. Similarly, because KSK uses Neo4j's query processor, its plans do not use WCOJs.

Adjacency List Indexes in Graph Analytics Systems: There are numerous graph analytics systems [34], [35], [36] that are designed to do batch analytics, such as decomposing a graph into connected components. These systems use native graph storage formats, such as adjacency lists or sparse matrices. Work in this space generally focuses on optimizing the physical layout of the edges in memory. For systems storing the edges in adjacency list structures, a common technique is to store them in CSR format [8]. To implement A+ indexes we used a variant of CSR that can have multiple partitioning levels. Reference [36] studies CSR-like partitioning techniques for large lists and reference [37] proposes segmenting a graph stored in a CSR-like format for better cache locality. This line

of work is complementary to ours.

Indexes in RDF Systems: RDF systems support the RDF data model, in which data is represented as a set of (subject, predicate, object) triples. Prior work has introduced different architectures, such as storing and then indexing one large triple table [38], [39] or adopting a native-graph storage [40]. These systems have different designs to further index these tables or their adjacency lists. For example, RDF-3X [38] indexes an RDF dataset in multiple B+ tree indexes. As another example, the gStore system encodes several vertices in fixed length bit strings that captures information about the neighborhoods of vertices. Similar to the GDBMSs we reviewed, these work also define fixed indexes for RDF triples. A+ indexes instead gives users a tunable mechanism to tailor a GDBMS to the requirements of their workloads.

Indexes for XML Data: There is prior work focusing on indexes for XML and the precursor tree or rooted graph data models. Many of this work provides complete indexes, such as DataGuides [41] or IndexFabric [42], or approximate indexes [43], [44] that index the paths from the roots of a graph to individual nodes in the data. These indexes are effectively summaries of the graph that are used during query evaluation to prune the search of path expressions in the data. These indexes are not directly suitable for contemporary GDBMS which store non-rooted property graphs, where the paths that users search in queries can start from arbitrary nodes.

Other path and complex subgraph indexes: Many prior algorithmic work on evaluating subgraph queries [45], [46], [47] have also proposed auxiliary indexes that index subgraphs more complex than edges, such as paths, stars, or cliques. This line of work effectively demonstrates that indexing such subgraphs can speed subgraph query evaluation. Unlike our work, these subgraphs can be more complex but their storage is not optimized for space efficiency.

VII. Conclusions

Our work was motivated by the shortcoming that existing GDBMSs have fixed adjacency list indexes that limit the workloads that can benefit from their fast join capabilities. As a solution, we described the design and implementation of a new indexing subsystem with restricted materialized view support that can be stored using a space-efficient technique. We demonstrated the flexibility of A+ indexes, and evaluated the performance and memory tradeoffs they offer on a variety of applications drawn from popular real-world applications.

References

[1] "Neo4j," https://neo4j.com.
[2] "JanusGraph," https://janusgraph.org.
[3] "TigerGraph," https://www.tigergraph.com.
[4] A. Mhedhbi and S. Salihoglu, "Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins," *PVLDB*, 2019.
[5] C. Kankanamge *et al.*, "Graphflow: An Active Graph Database," *SIGMOD*, 2017.
[6] "Neo4j Property Graph Model," https://neo4j.com/developer/graph-database, 2019.
[7] S. Sahu *et al.*, "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey," *VLDBJ*, 2019.
[8] A. Bonifati *et al.*, *Querying Graphs*. Morgan & Claypool, 2018.
[9] H. Q. Ngo *et al.*, "Skew strikes back: New developments in the theory of join algorithms," *SIGMOD Rec.*, 2014.
[10] A. Mhedhbi *et al.*, "A+ indexes: Tunable and space-efficient adjacency lists in graph database management systems," *CoRR*, 2021.
[11] "openCypher," https://www.opencypher.org.
[12] C. R. Aberger *et al.*, "EmptyHeaded: A Relational Engine for Graph Processing," *TODS*, 2017.
[13] A. Y. Halevy, "Answering queries using views: A survey," *VLDBJ*, 2001.
[14] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing queries with materialized views," in *ICDE*, 1995.
[15] D. Kossmann and K. Stocker, "Iterative dynamic programming: A new class of query optimization algorithms," *TODS*, vol. 25, no. 1, 2000.
[16] A. Buluc, T. Mattson, S. McMillan, J. Moreira, and C. Yang, "Design of the graphblas api for c," in *International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017.
[17] J. Goldstein and P.-r. Larson, "Optimizing queries using materialized views: A practical, scalable solution," in *SIGMOD*, 2001.
[18] J. D. Ullman, *Principles of Database and Knowledge-Base Systems, Vol. I.* Computer Science Press, Inc., 1989.
[19] K. Ammar *et al.*, "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal and Low-Memory Dataflows," *PVLDB*, 2018.
[20] S. Chaudhuri and V. Narasayya, "Autoadmin what-if index analysis utility," *SIGMOD Rec.*, 1998.
[21] S. Chaudhuri and V. R. Narasayya, "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server," in *VLDB*, 1997.
[22] B. Ding *et al.*, "Ai meets ai: Leveraging query executions to improve index recommendations," *SIGMOD*, 2019.
[23] S. Agrawal *et al.*, "Automated selection of materialized views and indexes in sql databases," *PVLDB*, 2000.
[24] H. Gupta and I. S. Mumick, "Selection of views to materialize in a data warehouse," *TKDE*, 2005.
[25] P. Gupta *et al.*, "Real-time Twitter Recommendation: Online Motif Detection in Large Dynamic Graphs," *PVLDB*, 2014.
[26] F. Bi *et al.*, "Efficient Subgraph Matching by Postponing Cartesian Products," *SIGMOD*, 2016.
[27] X. Qiu *et al.*, "Real-Time Constrained Cycle Detection in Large Dynamic Graphs," *PVLDB*, 2018.
[28] S. Abiteboul, "On views and xml," *PODS*, 1999.
[29] R. Castillo *et al.*, "Selecting materialized views for rdf data," 2010.
[30] F. Goasdoué *et al.*, "Materialized View-Based Processing of RDF Queries," *Bases de Données Avancées*, 2010.
[31] F. Goasdoué *et al.*, "View selection in semantic web databases," *PVLDB*, 2011.
[32] W. Fan *et al.*, "Answering pattern queries using views," *TKDE*, 2016.
[33] J. M. F. da Trindade *et al.*, "Kaskade: Graph views for efficient graph analytics," *ICDE*, 2020.
[34] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, Implementation, and Applications," *IJHPCA*, 2011.
[35] G. Malewicz *et al.*, "Pregel: A System for Large-Scale Graph Processing," *SIGMOD*, 2010.
[36] J. Shun *et al.*, "Ligra: A Lightweight Graph Processing Framework for Shared Memory," *ACM SIGPLAN Notices*, 2013.
[37] Y. Zhang *et al.*, "Making Caches Work for Graph Analytics," *IEEE Big Data*, 2017.
[38] T. Neumann *et al.*, "RDF-3X: A RISC-style Engine for RDF," *PVLDB*, 2008.
[39] C. Weiss *et al.*, "Hexastore: Sextuple Indexing for Semantic Web Data Management," *PVLDB*, 2008.
[40] L. Zou *et al.*, "gStore: A Graph-Based SPARQL Query Engine," *VLDBJ*, 2014.
[41] R. Goldman and J. Widom, "Dataguides: Enabling query formulation and optimization in semistructured databases," *PVLDB*, 1997.
[42] B. Cooper *et al.*, "A fast index for semistructured data," *PVLDB*, 2001.
[43] T. Milo *et al.*, "Index structures for path expressions," *ICDT*, 1999.
[44] R. Kaushik *et al.*, "Exploiting local similarity for indexing paths in graph-structured data," *ICDE*, 2002.
[45] J. M. Sumrall *et al.*, "Investigations on path indexing for graph databases," *Euro-Par*, 2016.
[46] J. Cheng *et al.*, "Fast Graph Pattern Matching," *ICDE*, 2008.
[47] L. Lai, L. Qin, X. Lin, Y. Zhang, L. Chang, and S. Yang, "Scalable distributed subgraph enumeration," *PVLDB*, 2016.