KÙZU* Graph Database Management System

Xiyang Feng** Guodong Jin** Ziyi Chen Chang Liu Semih Salihoğlu
{x74feng,guodong.jin,z473chen,c.liu,semih.salihoglu}@uwaterloo.ca
University of Waterloo
Canada

ABSTRACT

Datasets and workloads of popular applications that use graph database management systems (GDBMSs) require a set of storage and query processing features that RDBMSs do not traditionally optimize for. These include optimizations for: (i) many-to-many (m-n) joins; (ii) cyclic joins; (iii) recursive joins; (iv) semi-structured data storage; and (v) support for universal resource identifiers. We present Kùzu, a new GDBMS we are developing at University of Waterloo that aims to integrate state-of-art storage, indexing, and query processing techniques to highly optimize for this feature set.

This paper serves the dual role of describing our vision for Kùzu and the system's factorized query processor, which is based on two design goals: (i) achieving good factorization structures under m-n joins; and (ii) ensuring sequential scans that avoid entire scans of columns and join indices when possible. As we show these two goals can sometimes conflict and we describe our core binary and worst-case optimal (multiway) join operators that simultaneously achieve both goals. Kùzu is actively being developed to be a fully functional open-source DBMS with the goal of wide user adoption.

1 INTRODUCTION

Modern GDBMSs adopt a graph/network data model and SQL-like high-level query languages that have several graph-specific constructs, such as arrows to describe joins and Kleene star to describe reachability between records. At their cores, GDBMSs are relational in the sense that they map their query language constructs to relational operators, such as join, project, filter, or group by, that process and output sets/relations of tuples. In practical use, GDBMSs power several analytics-oriented applications that are popular in fraud detection systems, recommendation engines, master data and knowledge management, among other domains.

The workloads of these application require several storage and processing features that existing RDBMSs are generally not optimized for. These features include: (i) many-to-many (m-n) joins; (ii) cyclic join queries, such as when finding cyclic graph patterns; (iii) recursive joins, such as those used for reachability computations; (iv) ability to store semi-structured data, i.e., whose columns/property names and types are not defined to the system apriori; and (v) storage and processing of universal resource identifiers (URIs), which are strings identifying entities in knowledge graphs encoded as RDF-style triples. This paper presents *Kùzu*[1], a new GDBMS that we are developing at University of Waterloo that aims to optimize these features through the integration of existing and novel state-of-art storage, indexing, and query processing techniques.

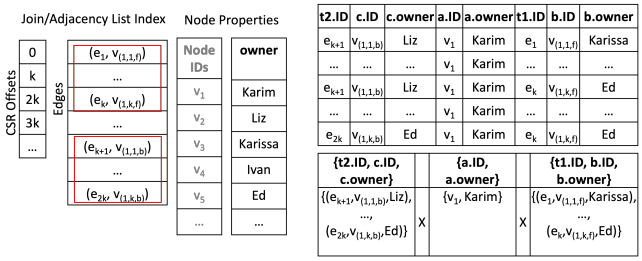

Figure 1: Left: columnar storage of Kùzu. Right: flat and factorized output of the 2-hop query in Example 1.

Kùzu's design is informed by our insights from discussions with many users of GDBMSs, which we previously published as a user survey paper [28], and developing our group's previous system GraphflowDB [12, 15, 18, 19], from which it differs in many important aspects. GraphflowDB was an in-memory non-transactional prototype system we used for our research agenda. Instead, Kùzu aims to be a fully functional, user-facing, and a *highly scalable* GDBMS, which was the most pressing challenge of users in our survey [28]. To that end, Kùzu is disk-based and scales out of memory, implements disk-based primary key and join indices, and integrates more robust and scalable join capabilities than GraphflowDB. In addition, Kùzu is transactional and aims to provide the core DBMS functionalities to be user-facing. In its current usage vision, Kùzu is an open source embeddable library suitable for quickly developing pipelines in the graph data science ecosystem. This is inspired by DuckDB [1, 26] and originally by SQLite [3] [2].

This paper focuses on Kùzu's processor, which is block-based as in modern analytical DBMSs and further aims to satisfy two goals:

(i) Intermediate relations of m-n joins should be factorized [23], i.e., represented as Cartesian products instead of flat tuples.
(ii) *Scans should always be sequential* and when possible only scan necessary blocks from base columns or join indices.

As we next demonstrate, these two design goals are often in conflict even in very simple queries:

Example 1. *Consider the query below that asks for the owners of the source and destination accounts of each 2-hop money transfer facilitated by Karim's accounts:*

```
MATCH (c:Acc)-[t2:Trnsfr]->(a:Acc)-[t1:Transfer]->(b:Acc)
WHERE a.owner = 'Karim' RETURN c.owner, b.owner
```

*Kù-zu ('bright' + 'to know') is a Sumerian word for"wisdom" [13].
**Equal contributions.

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

[1] https://github.com/kuzudb/kuzu
[2] Kùzu is also inspired by other open-source DBMS projects originally from research groups, such as PostgreSQL and MonetDB, which have also achieved wide adoption.

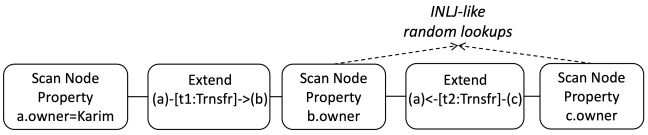

Figure 2: Standard GDBMS plan with INLJ-like operators.

Consider a columnar storage that we assume in this paper, where both adjacency lists and node properties are stored in columnar structures as in Figure 1. Consider a k-regular database, where a node v_i has k outgoing/incoming neighbors $\{v_{(i,1,f/b)}, ..., v_{(i,k,f/b)}\}$, where $v_{(i,j,f/b)}$ stands for v_i's, j'th **f***orward/***b***ackward neighbor. Suppose, Karim has one account v_1, so the output has k^2 tuples. Figure 1 shows both the flat and the succinct factorized representation of this output.*

A standard plan in GDBMSs for this query, shown in Figure 2, performs the following: (i) scan `a.owner` sequentially and identify Karim's accounts; (ii) extend the `a` nodes forward to `b` neighbors by scanning the adjacency lists of the `a` bindings sequentially (except due to the previous filter); (iii) scan the `b.owner` values; and (iv) and (v) repeat (ii) and (iii) for the backward edges of `a` bindings. Note that in steps (ii) and (iv), which implicitly perform an m-n join, by extending `a` nodes to their forward and backward edges and `b` neighbors in factorized format, such as $T_{v_1} = \{v_{(1,1,b)}, ..., v_{(1,k,b)}\} \times (v_1, \texttt{Karim}) \times \{v_{(1,1,f)}, ..., v_{(1,k,f)}\}$ (edge IDs are omitted) one can achieve the factorization structure from Figure 1. Such plans were used in GraphflowDB [12]. However, the scans of `b` and `c`'s owner properties perform non-sequential lookups in the `owner` column, since the neighbor IDs of nodes have no locality guarantees.

In a separate experiment and analysis study [14], we showed that because GDBMSs rely on such INLJ-like operators, they are amenable to performing non-sequential scans when scanning properties, which can degrade their performances on many queries. We also showed that RDBMSs, which rely only on hash joins, never suffer from this but they scan entire tables even if only a few tuples are needed, nor do they factorize intermediate tuples.

Our key technical approach to simultaneously achieve these design goals is a novel modified hash join operator that we call `ASP-Join`, for **a**ccumulate-**s**emijoin-**p**robe, which consist of 3 pipelines instead of the 2-pipelined vanilla hash join. The first pipeline accumulates a set of factorized tuples P, such as T_{v_1}, which form the probe side. We use P to construct a semijoin filter which selects the node IDs on the build side that will successfully join with P. Using sideways information passing (sip), this filter is then passed to the 2nd pipeline. which builds the hash table, but only (sequentially) scanning the necessary properties/edges. Finally, the 3rd pipeline re-scans the probe tuples in factorized format and probes the hash table. `ASP-Join` is the core join operator in Kùzu and is also at the core of our novel multiway worst-case optimal (wco) join algorithm. As we demonstrate in our evaluations, `ASP-Join`-based plans are robust and can broadly outperform the factorized and flat plans of several state-of-the art baseline systems on a wide range of selectivity settings on both acyclic and cyclic queries.

Our contributions and the outline of this paper are as follows: (1) We describe the overall vision of Kùzu (Section 2); (2) We describe Kùzu's factorized query processor, its two design goals and our `ASP-Join`-based binary and multiway wco join algorithms that achieve these goals (Section 3). We provide details of integrating `ASP-Join` into our system, including data structures to accumulate tuples, our optimizer, as well as the system's other core operators and their optimizations. Sections 4, 5, and 6 present experiments, related work, and conclusions, respectively.

2 KÙZU OVERVIEW

Kuzu is a single node (multi-core), disk-based, transactional but read-optimized GDBMS. It adopts a classic DBMS architecture that consists of a parser, binder, planner, storage manager, buffer manager, query processor, and transaction manager. We give brief overviews of different components, leaving the details of its query processor to Section 3.

Query Language and Data Model: Kùzu adopts the property graph model (PGM) [27], where databases are modeled as a set of labeled nodes and edges with properties. While original PGM supports only semi-structured properties, properties in Kùzu can also be structured. We allow nodes and edges to have a single label, which allows us to model them as single relations. The system has a data definition language (DDL), through which node and edge relations with pre-defined property/column names and data types can be specified. Semi-structured properties, which are supported only for nodes, are not apriori defined and can be inserted into the database through update commands. At a logical data level, the system supports modeling records in node or edge relations, where node relations have an extra "semi-structured" column, where arbitrary key-data type-value triples can be stored. Nodes have a primary key property where as edges are identified through system-level global numeric identifiers. The query language of Kùzu is openCypher [25], which we extended to define structured properties.

Storage and Indices: Kùzu is a columnar system similar to modern read-optimized analytical DBMSs [1, 4]. Kùzu's storage structures are primarily disk-based versions of the columnar in-memory designs described in reference [12]. Structured node properties are stored in vanilla column files. Edges are double indexed and stored in CSR-based adjacency list indices (a very simplified version is shown in Figure 1), which are the core join indices in the system to join node records. Adjacency list indices contain the destination node IDs of the edges and their labels, which are omitted if the labels of all destination nodes are guaranteed to be the same (this can be specified in DDL statements). Edge properties are similarly stored in "parallel" but separate CSR-based structures and double-indexed (see reference [12]). This has storage and update costs yet ensures that we can scan any node's edges and properties of these edges sequentially in both forward and backward directions. Each node relation has by default a hash index to be able to look up nodes on their primary keys. Storage structures are accessed through a buffer manager, which has fixed page sizes (4KB) and adopts the GClock eviction strategy.

Query Planner: Query planner adopts a standard architecture: a parser parses the Cypher query and binds the types of the structured properties and results of expressions (using the system catalog). The planner generates an initial logical plan, which goes through a dynamic programming-based join optimizer, which also internally performs several optimizations such as pushing filters. Then this plan is mapped into physical operators.

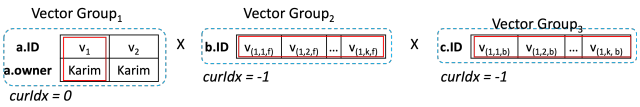

Figure 3: Example intermediate tuple representation as factorized vectors for the 2-hop query in Example 1.

Transaction Support: Kùzu uses write ahead logging to achieve atomicity and durability. Transactions are by design serializable as we currently allow one writer transaction in the system, which can run concurrently with multiple read transactions. All updates to the database, from node and edge relation creation to insertion, updates or deletions are atomic and durable if committed and can be rolled back. We have so-far designed our transaction management for correctness and simplicity. A more advanced multi-version concurrency control mechanism supporting multiple concurrent writers is in the roadmap of the next phase of the project.

User Interfaces and APIs: At the time of writing this paper, Kùzu is an embeddable library with Python and C++ APIs. Users can import the Kùzu library in applications, point to a database directory, and issue queries and update statements. In addition, the system has a shell console to issue statements.

Our immediate focus is on extending and polishing the system's user interfaces and APIs. Many applications need to store, clean, query, or extract subgraphs from their graph-structured data before performing more advanced graph analytics, such as building graph neural network models. Such applications need an embeddable easy-to-use data management system that is optimized for graph storage and querying. An important usage vision of Kùzu is to be the core DBMS for the graph data science ecosystem. To that end we plan to integrate Kùzu with popular graph data science libraries in Python, such as Python Geometric [10], Deep Graph Library [8], Mindspore Graph Learning [2], and NetworkX [20]. In addition, we plan to develop a REST API to better support user-facing applications that need a server to answer interactive queries, (e.g., return friends-of-friends of node v), which are common in the workloads of recommendation and question answering systems.

3 FACTORIZED QUERY PROCESSOR

We next present Kùzu's block-based and factorized processor. We first review the processor's intermediate factorized tuple representation design. We then cover the system's novel join operators.

3.1 Background: Factorized Vectors

Kùzu represents intermediate relations passed between operators as *factorized vectors* as described in reference [12][3]. Traditional block-based processors represent intermediate data as a set of flat tuples in a single group of vectors. When joins are m-n, this leads to data redundancy. For example, in traditional block-based processors, the output of the 2-hop query from Example 1 would be represented as a single group of vectors similar to the flat representation in Figure 1. Even if the output is produced in small batches (e.g., of size 1024), it would contain k^2 tuples with many value repetitions,

[3]The original term we used in reference [12] was *list-based processing*. The term *factorized vectors*, which we find more suitable, is due to Amine Mhedhbi.

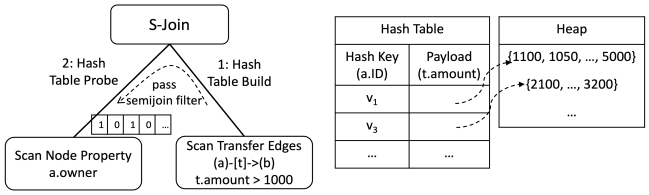

Figure 4: An S-Join plan for the query in Example 2.

In reference [12], we showed how a block-based processor can represent the same output more succinctly by using multiple *factorized vector groups*, each containing a set of independent vectors. Figure 3 shows an example of such representation, where each vector group has a `curIdx` field that can be in one of two states:

- Flat: If `curIdx` ≥ 0, the vector group (e.g., Vector Group$_1$ in the figure) is flattened and represents a single tuple that consists of the `curIdx`'th values in the vectors.
- Unflat: If `curIdx` $=-1$ (e.g., Vector Group$_2$), the vector group represents as many tuples as the size of the vectors it contains.

The intermediate tuples that factorized vector groups represent is the Cartesian product of the sets of tuples in each vector group. For example, the vector groups in Figure 3 represent k^2 tuples. In the theory of factorization [24], this is known as an *f-representation*.

3.2 Design Goals for m-n Join Operators

In reference [12], we obtained factorized structures using INLJ-like join and scan operators. Factorization was achieved by an `Extend` operator that extended a node binding from a (flattened) vector group i, e.g., `a`, to its neighbors by writing the neighbors to another vector group j in an unflat format. For example, we could achieve the factorization structure in Figure 1 for the 2-hop query with the plan in Figure 2. The use of such INLJ-like operators also guarantees reading only the adjacency lists that will successfully join with a node binding. However, INLJ-like operators cannot guarantee sequential scans of adjacency lists or properties, as the vertices in adjacency lists have no guaranteed locality. In disk-based systems this can be very expensive. Our primary research question when designing Kùzu's m-n join operators was: *how can we simultaneously benefit both from factorization and fast sequential scans?*

3.3 S-Join

In reference [14], we had designed a modified hash join operator for DuckDB, called `S-Join`, that could pass semijoin filters from its build side to probe side to avoid scanning large tables. Our implementation in DuckDB was based on flat tuple processing. We implemented a variant of `S-Join` in Kùzu that does factorized processing, which we describe here. However, as we show `S-Join` can only obtain a limited set of factorized structures, which we will address with `ASP-Join`. We describe `S-Join` through an example.

Example 2. *Consider the query below:*

```
MATCH (a:Account)-[t:Transfer]->(b:Account)
WHERE t.amount > 1000 RETURN a.owner, t.amount
```

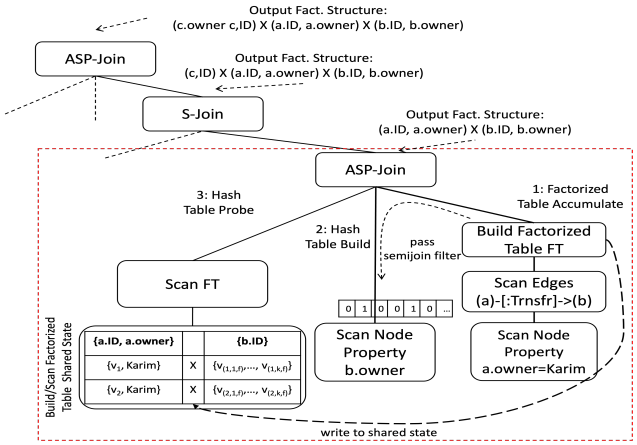

Figure 5: A plan with ASP-Join, whose sub-plan computing the (a)-[t1:Transfer]->(b) join is shown in detail.

A plan using S-Join for this query is shown in Figure 4. Recall that Kùzu stores edge properties in CSR structures that are "parallel" to the adjacency lists. So we can scan the edge properties in (a.ID)× (t.ID, t.amount) format, where a.ID would be flattened. An example tuple would be $\{v_1\} \times \{(v_{(1,1,f)}, \texttt{amount}_1) \ldots (v_{(1,k,f)}, \texttt{amount}_k)\}$. The query further needs the a.owner properties, which needs to be joined with these factorized tuples. S-Join has two pipelines to perform this join.

Pipeline 1 (Hash Table Build): Since a.ID is flattened, we can directly hash these factorized tuples on a.ID in a hash table with payloads that contain lists of t.amount values. Figure 4 shows the hash table that would be constructed. As part of building this hash table, S-Join also constructs a semijoin filter identifying the a.ID values that are hashed. 1 in location i indicates that a.ID=i should be scanned. This filter is passed to the probe side scan to only scan the a.owner values that are guaranteed to be part of the join.

Pipeline 2 (Hash Table Probe): Sequentially scans the a.owner column (only the values that pass the semijoin filter) and probes the hash table to join a.owner values with hashed factorized tuples.

S-Join achieves our two goals only if the build side contains a good factorization structure in which the join key is flat. As we next demonstrate, sometimes the good factorization structure of a sub-query may contain an unflat join keys, so joining requires flattening the join key values and losing the factorization structure.

3.4 Binary ASP-Join

Let us return to the 2-hop query in Example 1. In this query a plan should start scanning a nodes, and then scan the forward (or backward) edges of a in factorization structure (a.ID, a.owner) × (t.ID, b.ID), where (a.ID, a.owner) is flattened. The plan would next need to scan b.owner column and join with these factorized tuples. However, we cannot use S-Join because b.ID values are unflat. We would need to flatten these values, thereby losing this factorization structure. Ideally, we should use these factorized tuples on the probe and not the build side, but also pass a semijoin filter to the build side to identify the successfully joining b.ID values. To do so, our ASP-Join algorithm first accumulates the probe side of a hash join as a first pipeline before the actual building and probing. Figure 5 shows the plan Kùzu would generate for this query.

Pipeline 1 (Probe Accumulation): We first accumulate the set of probe factorized tuples and store them in a temporary relation, we call *Factorized Table* FT. This computation consists of only sequential operations, as we sequentially write to this temporary relation. Similar to hash table build pipeline of S-Join we also construct a semijoin filter to pass to the 2nd pipeline. In Example 1, the filter would be on the b.ID values of the accumulated factorized tuples.

Pipeline 2 (Hash Table Build): The algorithm next evaluates the build side sub-plan of the hash join, during which the appropriate scans at the leaves, e.g., of b.owner, use the passed semijoin filter.

Pipeline 3 (Hash Table Probe): The factorized tuples in FT are scanned and probed in the hash table to produce outputs.

We review the rest of the plan in Figure 5 for completeness. The plan contains two other join operators: (i) the S-Join joins the (a)-[t1]->(b) factorized tuples from the below ASP-Join with the (c)<-[t2]-(a) edges. Here, the join key is a.ID which is flat on both sides, so we can use S-Join; and (ii) The top ASP-Join, joins the (c)<-[t2]-(a)-[t1]->(b) tuples, with factorization structure (c.ID, t2.ID) × (a.ID, a.owner) × (t1.ID, b.ID, b.owner), with (c,ID, c.owner) tuples. Here we use ASP-Join because the *c.ID* values from the child S-Join operator are unflat.

ASP-Join has the overhead of accumulating the probe side tuples and re-scanning them yet as we show its overhead is acceptable when the benefits of keeping a good factorization structure is low and in many settings the benefits far overweighs the overheads.

3.5 Multiway WCO ASP Join

WCO join algorithms [21] have asymptotic runtime advantages over binary joins for cyclic join queries. Briefly, in graph terms, these algorithms take prefixes of tuples P that match parts of a cyclic query and extend each $p \in P$ to a common node variable by intersecting multiple adjacency lists of vertices matched in p. We review the wco *Generic Join* algorithm [21] through an example.

Example 3. *Consider the triangle query below:*

```
MATCH (c:Acc)<-[t3:DD]-(a:Acc)-[t1:Trn]->(b:Acc)-[t2:Trn]->(c)
WHERE a.owner ='Karim'  RETURN a.owner, b.ID, c.ID
```

Generic Join would find all (a)-[t1:Transfer]->(b) *"prefix" tuples and for each edge (*a=v_i, b=v_j*), would intersect the forward* Transfer *list of* v_j *and backward* DirectDeposit (DD) *list of* v_i *to compute a set of* c *node IDs. We call* c *the "extension node variable" and* a *and* b *the "bound" variables. This intersection effectively joins the prefix tuples with two edge relations in a single operation. In other cyclic queries, such as cliques, more than 2 adjacency lists can be intersected.*

The operator implementing wco joins in Kùzu is an extension of ASP-Join. An example plan for the above triangle query is shown in Figure 6. Suppose we have a sub-plan that generates the prefix tuples and that needs to be followed with an ℓ-way intersection.

Pipeline 1 (Probe Accumulation): This pipeline is the same as in the binary ASP-Join. The tuples can arrive in any factorized format and necessary flattenings will be performed in the last pipeline (see below). The pipeline computes ℓ semijoin filters, one for each edge relation that store the adjacency lists that will be intersected.

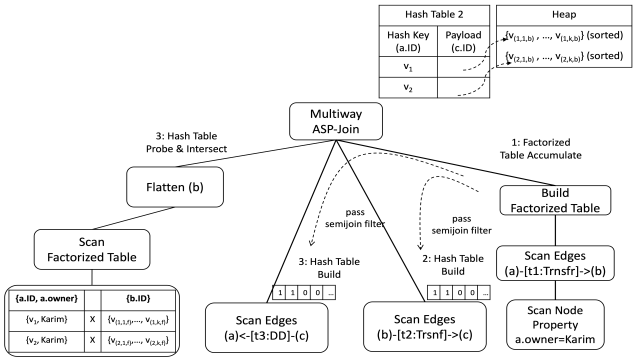

Figure 6: A plan with a multiway `ASP-Join` operator that computes the triangle query from Example 3.

Pipelines 2..ℓ+1 (Hash Table Build): For each of the ℓ edge relations, the necessary adjacency lists passing the semijoin filter are scanned, sorted, and hashed into separate hash tables using their bound variables as keys. In practice if multiple adjacency lists with the same edge labels and in the same direction are needed, we take the union of these semijoin filters and construct a single hash table. Figure 6 shows one of the hash tables, those storing `(b)-[t2:Transfer]->(c)` lists, in our example.

Pipeline ℓ+2 (Hash Table Probe): The accumulated factorized probe tuples are re-scanned and any bound variable is flattened using a sequence of `Flatten` operators. For example, in Figure 6, the b variables of probe tuples are in unflat format and need to be flattened as b's forward `Transfer` lists will be intersected. `Flatten` takes a factorized prefix tuple p with an unflat vector group VG with k values in it and produces k many tuples where VG is flat. After flattenings, the necessary adjacency lists are fetched from the hash tables and intersected. This produces a factorized output tuple where the results of the intersections, e.g., the `c` values, are stored in an unflat vector group.

Similar to the binary `ASP-Join`, the multiway `ASP-Join` operator ensures that the scans of the adjacency lists from storage are sequential.

As we show in our evaluations, under many settings, this approach is more robust than using INLJ-like operators, such as GraphflowDB's `Intersect` [19] to perform multiway intersections.

3.6 Further Details

We briefly cover several implementation details of Kùzu's processor.

Parallelism: We adopt morsel-driven parallelism [16]. Parallel tasks run copies of the same pipeline and coordinate to get morsels of node IDs/properties to scan until no morsels remain.

Optimizer: Kùzu adopts a dynamic programming-based join optimizer whose cost metric is the number of factorized tuples, which we chose for simplicity. Kùzu models the contents of the MATCH clause in Cypher as a standard equi-join graph, where each node and edge variable is modeled as a relation. For any sub-query, we keep a best plan for each possible factorization structure, which we limit as the number of factorization structures can grow. For edge relations (a_i)-$[e]$->(a_j), we start with two plans: one for scanning the edges in the forward direction, so when `a` values are flat and (`e`, `b`) values are unflat and one where `b` values are flat and (`e`, `a`) values are unflat. For each sub-plan SP, we maintain the IDs of the nodes a_i that can pass sideways information if SP were to be joined on a_i. This is based on a simple rule: if there is a predicate on a_i or on an edge that a_i is part of, then we assume that a_i can pass information. When enumerating plans for larger sub-queries, we pick between hash join, `S-Join`, or `ASP-Join` based on the factorization structures of sub-plans and whether sub-plans can do sip. Currently, except for primary key predicates, the selectivity estimations use magic constants. If a subquery is cyclic, we enumerate at least one plan where the last operator is a multiway `ASP-Join`.

Expression evaluations: In our approach to integrating factorization to a block-based processor, binary expression evaluators in the processor can take two types of operands: (i) both sides are in the same vector group, in which case the evaluation happens as in standard block-based processors and as both operands are guaranteed to have the same size; and (ii) both sides are in different vector groups. In this case, we ensure during plan enumeration that at least one side is flattened and has size 1. Each binary expression evaluation has two branches to handle these two cases.

Finally, we note that we need to maintain the multiplicity of each factorized tuple because projections can effect the multiplicities. For example, suppose an operator projects out the unflat b values in factorized tuples such as: `a.owner`=Karim)} $\times$ {(`b.ID`=$v_{(1,1,f)}$), ..., (`b.ID`=$v_{(1,k,f)}$)}. After the projection, we need to maintain the information that the tuple `a.owner`=Karim)} has multiplicity k. So some operators, such as aggregations, cannot blindly assume that each factorized tuple represents one tuple, and may need to check multiplicities to produce correct outputs.

4 EVALUATIONS

We present experiments demonstrating the robustness and performance benefits of using `ASP-Join` based plans on acyclic and cyclic micro-benchmark queries. We also perform end-to-end system-to-system comparisons against DuckDB (v0.4.0)[4], and Umbra on the LDBC business intelligence benchmark [9] at scale factor 100 (LDBC 100). We also compared against Neo4j but did not find its community edition competitive with the other systems on the majority of our queries. We do not compare against GraphflowDB, which is an in-memory system (others are disk-based) but use a Kùzu configuration (Kùzu-INLJ), which behaves similarly to GraphflowDB plans.

We used a machine with two Intel E5-2670 @2.6GHz CPUs and 256 GB of RAM with 16 physical and 32 logical cores. We used 8 threads in each system and gave 64GB to their buffer managers. We mark experiments that take more than 1000 seconds as timeout (TO). We repeated each query 3 times and report the fastest runtime.

4.1 End-to-end Comparisons.

We used the LDBC interactive short reads (IS), which contain a total of 7 queries. These are 1- to 4-hop "point" queries, i.e., they contain an equality predicate on the primary key of one node variable. These queries also scan properties of other nodes in the query. We changed the outer joins (Optional Match clause in openCypher) with full joins because our current outer join implementation is not

[4] https://github.com/duckdb/duckdb/releases/tag/v0.4.0

	IS01	IS02	IS03	IS04	IS05	IS06	IS07
Kùzu	1.0	**16.69**	**1.77**	1.0	**1.49**	**5.66**	**4.74**
DuckDB	**0.10**	5264.00	43.76	**0.10**	24.10	5496.00	481.80
Umbra	0.94	189.22	22.37	0.83	1.89	49.92	166.66

Table 1: Runtime (ms) of Kuzu, DuckDB, and Umbra on LDBC-SNB IS queries.

optimized. Table 1 shows our results. IS01 and IS04 are 1-hop queries. These take sub-millisecond times across all systems and DuckDB performs the best on these. Other queries contain at least 2-hop queries over m-n edges. On these Kùzu consistently outperforms DuckDB and Umbra. These queries still take sub-second time and the primary benefit of Kùzu is its ability to do sip and avoid scanning large chunks of the base relations. This is because the primary key predicate is very selective and the queries contain relatively short paths. Kùzu can do chained sip here to pass semijoin filters to all other node variables, which gives it an advantage on these queries. Kùzu also factorizes the outputs. However, on these queries, this benefit is not very visible because the joins are not the dominant factor. Finally, we investigated why DuckDB performs very poorly on IS06 and we found that its optimizer picked a poor plan that joined two relations without any predicates, which Umbra and Kùzu avoided.

4.2 Acyclic Query Microbenchmarks

W evaluate different Kùzu configurations, DuckDB and Umbra, under three simple acyclic path queries. We use LDBC 100. We use path queries of the form (a_1)-$[e_1]$->(a_2) .. ->(a_i) and put a predicate <p on a manually inserted ID column/property on one of the a_i and we return the min(a_j.prop) of two properties every a_j (including a_i). We show our 1-hop query as an example momentarily. We modify the selectivity p on each query to test the effects of sip in plans with ASP-Join and S-Join. In all of these queries Kùzu picks a left-deep plan that scans the node with the predicate and extends to the rest of the nodes using solely ASP-Join. This is our "Kùzu" configuration. Our Kùzu-S-Join and Kùzu-INLJ use equivalent left-deep plans that use solely S-Join and INLJ operators. For DuckDB and Umbra we write the equivalent SQL query and let the systems pick their plans through their optimizers. Our 1-hop query is:

```
MATCH (a1:Person)-[e1:Knows]->(a2:Person) WHERE a1.ID < p
RETURN min(a1.ID), min(a1.bday), min(a2.ID), min(a2.bday)
```

On low selectivities, the dominant cost is the scan of the adjacency lists and the properties and join is not a major cost. Therefore, we expect all Kùzu configurations to perform well as they all scan small amounts of Knows adjacency list index and other properties and benefits from factorization should be minor. DuckDB and Umbra should perform relatively worse as they need to scan these relations. Gradually as we increase the selectivity, we expect DuckDB and Umbra's performances to become more competitive and Kùzu-S-Join and Kùzu-INLJ to perform worse. For Kùzu-INLJ this is because INLJ will lead to random and repeated I/Os of a2.ID/bday properties; for Kùzu-S-Join because it loses its factorization structure when reading a2.ID/bday properties. In contrast we expect ASP-Join's performance to degrade slower than other Kùzu configurations.

Our results are shown in Table 2 and consistent with our expectations. At low selectivities, we see Kùzu configurations outperforming DuckDB and Umbra (with > 10x factors). At high selectivities of 10% and 100%, Umbra and DuckDB outperform Kùzu configurations but Kùzu-S-Join and Kùzu-INLJ degrade much faster than Kùzu. Importantly Kùzu either outperforms other systems/configuration or is within 3.3x of the best performing one, which happens at 100% selectivity, i.e., when ASP-Join only has overheads.

Our 2-hop query is (a_1:Comment)<-[:Likes]-(a_2:Person)-[:Likes]->(a_3:Comment) where the predicate is on the middle a_2 node. This ensures that left deep plans that start from a_2 can obtain a good factorization structure, so we can evaluate a case when factorization can have major benefits. At low selectivities, we expect systems to behave similarly to the 1-hop query for exactly the same reasons. As selectivity increases, we again expect DuckDB/Umbra to start closing the gap and outperforming Kùzu-S-Join and Kùzu-INLJ which will suffer for the same reasons above. However, now Kùzu-INLJ should remain competitive until higher selectivity levels because we expect it to win more significantly from factorization. Kùzu now should not only stay competitive but outperform other systems as it does not degrade due to random I/O and still maintains a very good factorization structure. Our results are shown in Table 2 and confirm this expectation. We omit DuckDB which was not competitive in these queries because it picked a poor join plan that joined two large tables. As shown in the table, Kùzu performs at sub-millisecond levels except at the highest selectivity levels, where it outperforms other configurations and Umbra. Kùzu-INLJ is also competitive at every level due to benefits from factorization.

Finally for our 3-hop query we used (a_1:P)-[:Knows]->(a_2:P)-[:Knows]->(a_3:P)-[:Knows]->(a_4:P) with the predicate on a_2. Our expectations for this query is the same for our 2-hop query (as explained momentarily) but this query demonstrates Kùzu's ability to do chained sip, where information is passed from a_2 to not only a_1 and a_3, which are adjacent to a_2 but also to a_4. Our expectations are similar to our 2-hop query because putting the predicate on a_2 still allows left deep plans used by Kùzu configurations to achieve a very good factorization where both a_1-$[e_1]$-> and -$[e_1]$->(a_4) adjacency lists and the age properties on a_1 and a_4 can be factored out. We again omit DuckDB for the same reaons. We now also omit Kùzu-S-Join because it was not competitive at low selectivities and ran out of memory for storing a large factorized table at higher ones (as in the 2-hop query). As shown in Table 2, the behavior of the systems is generally similar to our 2-hop query. The primary difference is that, Kùzu can now outperform Kùzu-INLJ at lower selectivities too, because even at the lowest selectivity level Kùzu-INLJ needs to scan a_2's neighbors of neighbors, which makes Kùzu-INLJ suffer more from random reads compared to the 2-hop query.

4.3 Cyclic Query Microbenchmarks

We next evaluate Kùzu's plans with multiway ASP-Join operator on cyclic queries. We picked three queries: (i) a triangle query; (ii) a 4-cycle; and (iii) a 4-clique query. Similar to our acyclic queries, we put a predicate on one of the node variables and compute the min of two properties for each variable. We compare Kùzu's performance against the Kùzu-INLJ configuration, which uses GraphflowDB style Intersect operator to perform multiway intersections. We expect Kùzu's performance to be competitive with Kùzu-INLJ on

	1-hop query					2-hop query				3-hop query		
Selectivity	**Kùzu**	**SJ**	**INLJ**	**DuckDB**	**Umbra**	**Kùzu**	**SJ**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**
0.01%	0.003	0.003	**0.002**	0.181	0.023	0.33	12.56	**0.01**	1.90	**0.03**	0.09	0.15
0.1%	**0.005**	0.006	0.008	0.181	0.024	0.41	54.47	**0.11**	4.05	**0.10**	0.63	0.45
1%	**0.016**	0.038	0.090	0.186	0.029	**0.96**	OOM	1.04	12.30	**0.73**	6.71	6.96
10%	0.097	0.295	0.900	0.224	**0.082**	**3.89**	OOM	10.39	230.35	**7.85**	66.60	17.80
100%	0.637	3.381	9.010	0.240	**0.191**	**31.98**	OOM	92.35	TO	**80.29**	TO	236.42

Table 2: Runtimes (sec) of different Kùzu configurations, DuckDB, and Umbra on acyclic microbenchmark queries on LDBC 100. `SJ` and `INLJ` are the Kùzu-`S-Join` and Kùzu-`INLJ` configurations. OOM and TO stand for out of memory and timeout.

	web-BerkStan									LiveJournal								
	Triangle			4-Cycle			4-Clique			Triangle			4-Cycle			4-Clique		
Selectivity	**Kùzu**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**	**Kùzu**	**INLJ**	**Umbra**
0.01%	**0.01**	0.03	0.05	**0.02**	0.08	0.11	**0.02**	0.11	0.50	**0.04**	0.72	0.39	**0.97**	13.60	4.06	**0.10**	2.19	21.87
0.1%	**0.02**	0.28	0.07	**0.21**	2.35	0.64	**0.12**	1.64	2.59	**0.11**	5.58	0.65	**3.91**	74.91	19.68	**0.89**	21.66	Crash
1%	**0.08**	2.88	0.18	**2.01**	21.82	2.76	**1.28**	17.37	8.35	**0.78**	59.93	2.23	59.70	867.21	**25.72**	**12.67**	257.90	26.09
10%	**0.59**	29.09	1.01	20.87	206.80	**8.53**	**13.69**	160.26	17.59	**6.43**	585.67	14.55	**531.88**	TO	TO	149.79	TO	**128.86**
100%	6.55	291.2	**3.27**	230.74	TO	**77.39**	151.84	TO	**87.78**	59.39	TO	**39.26**	TO	TO	TO	TO	TO	TO

Table 3: Runtimes (sec) of Kùzu, Kùzu-`INLJ` (`INLJ`), and Umbra on cyclic queries on `web-BerkStan` and `LiveJournal`.

low selectivities and outperform it across higher selectivities, when Kùzu-`INLJ` should degrade due to random and repeated scans of adjacency lists from storage. We also used the Umbra system, which has a wco join implementation. The main differences are that Kùzu scans adjacency lists, sorts them on the fly and hashes them, and probes these sorted tables and intersects them. Instead Umbra avoids any sorting of sets of values, which is an advantage, and instead creates nested hash tables and intersects hash sets through hash lookups, which is a disadvantage as this is a slower way to intersect sets. In addition, Kùzu plans can perform sip and semijoins Here forming an expectation is harder at higher levels but for lower levels we expect Kùzu to outperform Umbra due to its semijoins.

We use the `web-BerkStan` and `LiveJournal` datasets [17], which respectively contain 7.6M and 69M edges. These datasets are unlabeled, which we modeled with as a single node and single edge relations. Our results are shown in Table 3. As we expected, throughout the experiments, Kùzu outperforms Kùzu-`INLJ` due to sequential scans at higher selectivities and is competitive or outperforms Umbra. This indicates that at high selectivities when sip has overheads, pre-sorting and intersecting lists is competitive with or outperforms Umbra's hash-based approach which avoids explicit sorting.

5 RELATED WORK

Several of Kùzu's core design choices are based on our prior GraphflowDB project [12, 15, 18, 19]. Our work on GraphflowDB focused on integrating factorization and wco join algorithms into traditional pipelined and block-based processors. GraphflowDB relied heavily on INLJ algorithms and pre-sorted adjacency lists, which Kùzu completely avoids. Other approaches for integrating WCO join algorithms [21, 22, 29] into DBMSs have been proposed in prior work [5, 7, 11, 19]. LogicBlox integrates the Leapfrog TrieJoin (LFTJ) wco join algorithm [29] that processes sorted tries of input relations. The implementation description in reference [5] describes the system relying solely on LFTJ for equi-joins and not generating plans that mix wco and binary joins. Several work have described richer join plans that can mix both types of algorithms depending on the structures of the join queries [11, 19]. Umbra integrates WCO join-style multiway joins by building nested hash indices on the fly as we discussed in Section 4.

Another core goal of Kùzu's query processor is to adopt factorized processing. The primary goal of factorization is to factor out repeated values when multi-valued dependencies emerge in intermediate results under m-n joins. The theory of factorized databases [23, 24] has developed both factorization representation systems for relations as well as a set of algorithms for core relational operators that show that factorized query processing can have asymptotic time improvements over flat processing. Olteanu et al. have also developed a factorized query processor that implements these core algorithms in the FDB [6] system. These algorithms however rely on representing input relations as sorted tries and process and output tries, which does not seem suitable to integrate into traditional pipelined DBMS processors.

6 CONCLUSIONS

We presented the Kùzu system that is being developed to be a highly-scalable and efficient GDBMS that integrates state of the art techniques for graph data management and query processing. We described the system's query processor, which is designed to both use efficient factorization structures under m-n joins as well as perform only sequential scans. We described how we achieve these goals using hash join-based join operators that can perform sideways information both from the build side to the probe side and vice versa. We demonstrated the robustness and high performance of plans that use these join algorithms. Kùzu is being developed actively and integrating new features and aims to be a widely adopted fully functional DBMS. We hope Kùzu can also be a productive research infrastructure for researchers to design novel techniques for managing databases modeled as graphs.

7 ACKNOWLEDGEMENTS

We are grateful to Pranjal Gupta, who designed the core storage layer of the system, and Amine Mhedhbi, who helped in the initial implementation of the query processor. We also thank both for numerous useful discussions. We are also grateful to Lori Paniak for his help in solving numerous technical issues during our development and experiments.

REFERENCES

[1] 2021. DuckDB. https://duckdb.org
[2] 2022. Mindspore Graph Learning https://www.mindspore.cn/graphlearning/docs/en/master/index.html.
[3] 2022. SQLite. https://www.sqlite.org/index.html
[4] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* (2013).
[5] Molham Aref et al. 2015. Design and Implementation of the LogicBlox System. In *SIGMOD*.
[6] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. 2012. FDB: A Query Engine for Factorised Relational Databases. *PVLDB* 5, 11 (2012).
[7] Christopher R. Aberger et al. 2017. EmptyHeaded: A Relational Engine for Graph Processing. *TODS* 42, 4 (2017).
[8] Minjie Wang et al. 2019. Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks. *arXiv: Learning* (2019).
[9] Renzo Angles et. al. 2020. The LDBC Social Network Benchmark. *CoRR* abs/2001.02299 (2020).
[10] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. *CoRR* abs/1903.02428 (2019).
[11] Michael Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *PVLDB* 13, 12 (2020).
[12] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. 2021. Columnar Storage and List-based Processing for Graph Database Management Systems. *PVLDB* 14, 11 (2021).
[13] John A. Halloran. 2006. *Sumerian Lexicon: A Dictionary Guide to the Ancient Sumerian Language.* Logogram Publishing.
[14] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads through Predefined Joins. *PVLDB* 15, 5 (2022).
[15] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD*.
[16] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*.
[17] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[18] Amine Mhedhbi, Pranjal Gupta, Shahid Khaliq, and Semih Salihoglu. 2021. A+ Indexes: Lightweight and Highly Flexible Adjacency Lists for Graph Database Management Systems. In *ICDE*.
[19] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-Case Optimal Joins. *TODS* (2021).
[20] NetworkX 2022. NetworkX https://networkx.org/.
[21] H. Ngo, C. Ré, and A. Rudra. 2014. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record* 42, 4 (2014).
[22] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2012. Worst-case Optimal Join Algorithms. In *PODS*.
[23] Dan Olteanu and Maximilian Schleich. 2016. Factorized databases. *ACM SIGMOD Record* 45 (2016).
[24] Dan Olteanu and Jakub Závodnỳ. 2015. Size bounds for factorised representations of query results. *TODS* 40, 1 (2015).
[25] openCypher 2022. openCypher. https://www.opencypher.org.
[26] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *SIGMOD*.
[27] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph Databases: New Opportunities for Connected Data* (2nd ed.). O'Reilly Media, Inc.
[28] 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. *VLDBJ* 29, 2 (2020).
[29] Todd L. Veldhuizen. 2012. Leapfrog Triejoin: a Worst-Case Optimal Join Algorithm. *CoRR* abs/1210.0481 (2012).