Making RDBMSs Efficient on Graph Workloads Through Predefined Joins

Guodong Jin
jinguodong@ruc.edu.cn
Renmin University of China

Semih Salihoglu
semih.salihoglu@uwaterloo.ca
University of Waterloo, Canada

ABSTRACT

Joins in native graph database management systems (GDBMSs) are predefined to the system as edges, which are indexed in adjacency list indices and serve as pointers. This contrasts with and can be more performant than value-based joins in RDBMSs. Existing approaches to integrate predefined joins into RDBMSs adopt a strict separation of graph and relational data and processors, where a graph-specific processor uses left-deep and index nested loop joins (INLJ) for a subset of joins. In this paper we study and experimentally evaluate this technique's performance against an alternative technique that is based on using hash joins that use system-level row IDs (RIDs). In this alternative approach, when a join between two tables is predefined to the system, the RIDs of joining tuples are materialized in extended tables and optionally in RID indices. Instead of using the RID index to perform the join directly, we use it primarily in hash joins to generate filters that can be passed to scans using sideways information passing (sip), ensuring sequential scans. We further compare these two approaches against: (i) the default value-based joins of an RDBMS; and (ii) using materialized views that can avoid evaluating predefined joins completely and instead replace them with scans. We integrated our alternative approach to DuckDB and call the resulting system *GRainDB*. Our evaluation demonstrates that existing INJL-based approach can be very efficient when entity relations contain very selective filters. However, GRainDB's approach is more robust and is either competitive with or outperforms the INLJ-based approach across a wide range of settings. We further demonstrate that GRainDB far improves the performance of DuckDB, which uses default value-based joins, on relational and graph workloads with large many-to-many joins, making it competitive with a state-of-the-art GDBMS, and incurs no major overheads otherwise.

PVLDB Reference Format:
Guodong Jin and Semih Salihoglu. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. PVLDB, 15(5): 1011 - 1023, 2022.
doi:10.14778/3510397.3510400

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

Proceedings of the VLDB Endowment, Vol. 15, No. 5 ISSN 2150-8097.
doi:10.14778/3510397.3510400

1 INTRODUCTION

Perhaps the two most commonly used data structures to model data in enterprise database applications are tables, which are the core structures of relational database management systems (RDBMSs), and graphs, which are the core structures of several classes of systems, most recently of property graph database management systems (GDBMSs for short), such as Neo4j [3], TigerGraph [4], DGraph [1], and GraphflowDB [15, 21, 28–30]. GDBMSs target what are colloquially referred to as *graph workloads*. This is a colloquial term as the distinction of application data and queries as graph vs relational is not very descriptive. Data of many applications can equivalently be modeled as a set of relations and queried in SQL or a graph and queried in the query languages of GDBMSs, which are generally similar to SQL. In the context of our paper, we use this term to refer to workloads that contain large many-to-many joins. For example, these workloads appear in social networking applications for finding long paths between two people over many-to-many friendship relationships or in financial fraud detection applications for finding fraudulent patterns across many-to-many money transfers across bank accounts.

Several economic and technical factors have lead researchers to investigate techniques to support efficient graph querying natively inside RDBMSs. For example, it is recognized that the data stored in many specialized GDBMSs are extracted and replicated from RDBMSs [8, 38, 40, 43] because users require the fast join capabilities of GDBMSs for specialized applications. In addition, many applications query their graphs by running predicates on node and edge properties or grouping and aggregations, for which RDBMSs already employ efficient techniques. Therefore leveraging mature RDBMS technology to support graph workloads natively is highly appealing to both users and vendors: users can avoid the challenges of duplicating data and keeping multiple systems in sync, while vendors can avoid the efforts to develop a new system from scratch. We revisit this goal of extending an RDBMS natively with the fast join capabilities of GDBMSs.

We begin by analyzing the primary differences between the join evaluation techniques in RDBMSs and GDBMSs. We observe that contemporary GDBMSs perform joins between node records along *predefined edges* using dense record IDs of nodes, which serve as pointers to directly look up nodes. We refer to such joins as *predefined/pointer-based joins*, using a term used by Ted Codd to describe the difference between GDBMS of his era and the value-based joins of RDBMSs [11]. In some settings, these pointer-based joins can be more efficient than using non-integer keys.

The common approach of implementing predefined joins in GDBMS has two main components: (i) using adjacency list join indexes that index for each node ID, the joining edge and node

records; and (ii) using index nested loop join-style (INLJ) join operators, that directly use adjacency list indexes to perform the join. Often a sequence of INLJ operators are used in left-deep join plans to evaluate subgraph queries. This overall approach has also been adopted to integrate predefined joins to RDBMSs, most recently by the GR-Fusion system [17, 18] and also the GQ-Fast system [25] .

In GR-Fusion, SQL is extended to contain graph-specific constructs, using which users create graphs. The topologies of these graphs, i.e., the vertices and edges without properties, are stored in native adjacency list indexes, which are used during query processing for graph traversals/many-to-many joins, using new graph-specific operators, such as `EdgeScan` and `PathScan`. Parts of queries that refer to graph-specific constructs compile to these specialized graph operators, while the non-graph parts of queries compile to existing operators of the RDBMS. Similar to GDBMSs, graph traversals in GR-Fusion are evaluated with left deep join plans that use INLJ operators, so avoid using efficient bushy plans. As we demonstrate, this approach can also lead to inefficient accesses when a query accesses properties of nodes and edges because scans of properties are done after joins are performed, instead of when scanning base relations. GQ-Fast [25] follows a very similar approach that is also based on using INLJ operators and adjacency list indexes to directly perform joins.

In this paper, we propose and experimentally evaluate an alternative approach to integrate predefined/pointer-based joins into RDBMSs. Our approach integrates predefined joins into a columnar RDBMS by extending two components of the system:

• **Physical Storage and Query Processor:** When a user predefines a primary-foreign key join from table F to table P, where a column of F has a foreign key to a column of P, this performs an ALTER TABLE command that inserts an additional RID_p column to F that contains for each row r_f in F the row ID (RID) of row r_p in P that r_f points to. RIDs are dense integer-based system-level IDs in columnar RDBMS that are used to identify the physical locations of the column values of each row. They are therefore system-level pointers, similar to node IDs in GDBMSs. Our approach relies on this property of RIDs.

In order to use these pointers to perform the primary-foreign key joins, we rewrite queries to replace primary-foreign key equalities with RID equalities. Equality predicates in many columnar RDBMSs are primarily evaluated with hash-joins. To exploit the pointer-nature of predefined joins, we employ *sideways information passing* (sip) from hash join operators to base tables scans to select only the RIDs that successfully join in these hash joins.

• **Indexing Sub-system:** A common way to represent many-to-many relationships between two sets of entities in relational databases is to have a table F that contains two foreign keys on two other (not necessarily different) tables P_1 and P_2. For simplicity of terminology, we refer to such F as a *relationship* table and P_i as *entity* tables. If the joins with both entity tables have been predefined to the system, users can additionally build an index on table F on the two extended RID columns RID_{p1} and RID_{p2}. This index is stored in the adjacency list format and serves two purposes. First, it is used to generate further information to pass when a query joins P_1, F, and P_2 and when a hash join operator builds a table of P_1 or P_2. Second, when a query refers to F only to facilitate the join of tuples in P_1 and P_2, namely, the query contains no predicates on F and projects out F's columns, the index allows us to reduce the number of joins in query plans.

We integrated this alternative approach into the DuckDB columnar RDBMS [35, 36] and call the extended system *GRainDB*. We perform extensive experimental evaluation of our approach to compare its performance against: (i) value-based joins by comparing it against vanilla DuckDB plans; (ii) left-deep INLJ plans of GDBMSs; and (iii) using materialized views to evaluate predefined joins, which can replace actual joins with scans of views. We demonstrate that GRainDB improves the median query execution time of DuckDB by 3.6x on the relational JOB benchmark which contains many-to-many joins, and by 22.5x on the LDBC SNB graph benchmark, making a columnar RDBMS competitive with the state-of-the-art GraphflowDB GDBMS [15]. We further show that our alternative apporach can be more efficient than both left-deep INLJ-based plans of GDBMSs (and GR-Fusion and GQ-Fast) on many queries, such as those with selective predicates on tables that represent edges/relationships, and against materialized views. In our detailed experimental analysis, we also show that using sip makes the optimizer of a system more robust because its semi-join computations can mitigate a poor join order selection of the optimizer. Our code, queries, and data are available here [2].

2 RELATED WORK

We next review prior work that leverage RDBMSs for supporting graph applications and the literature on sip and join indexes. GR-Fusion [17, 18] performs graph querying natively inside an RDMBS. Users define graphs as views over tables, The topology of graph views is stored natively in an adjacency list index. In contrast, the node and edge properties are stored as pointers to the underlying tables. Users refer to the paths in a graph view as if they are a separate table using a new `Paths` construct in the FROM clause of SQL. Then, part of the query that enumerates paths and their constraints are evaluated with special operators, such as VertexScan or PathScan, whose results are tuples that can be input to further relational operators. Therefore this approach creates dual query processing pipelines inside the system. One advantage of this approach is that the original relational operators remain unchanged because outputs from the graph pipeline are regular tuples. However, PathScan enumerates only paths, so some other patterns, such as stars, need to be evaluated by the vanilla relational query processor. Instead, our approach is purely relational and can improve equality joins on arbitrary queries. Second, paths are enumerated through DFS or BFS algorithms, which are akin to left-deep plans that use index nested loop join operators. These plans can be suboptimal compared to bushy plans, which can be generated in GRainDB. This is further exacerbated if vertex and edge properties need to be scanned during DFS of BFS by following pointers to the tables, which can lead to many random accesses. In contrast, the alternative hash join- and sip-based approach we integrated in GRainDB uses adjacency list indices to generate information to pass to scan operators but performs scans always sequentially. We intended but could not compare our solution against GR-Fusion. Since GR-Fusion's join evaluation approach is based on GDBMSs approach of implementing predefined joins, we will instead use existing GDBMSs to compare the pros and cons of GR-Fusion's approach.

GQ-Fast [25] supports a restricted subset of SQL called "relationship queries" which contain joins of tables that are similar to path queries, followed with aggregations. Similar to GRainDB and GR-Fusion, GQ-Fast stores relationship tables in CSR-like indices. However, similar to GR-Fusion, the joins are limited to paths and evaluated with left-deep index nested loop join operators that are equivalent to DFS traversals. Unlike GRainDB and GR-Fusion, GQ-Fast is implemented as a standalone system and does not integrate these techniques into an underlying RDBMS to support more general queries, which is left as future work [25]. However, even this envisioned integration is similar to GR-Fusion, where the GQ-Fast layer is a separate query processor whose outputs are given to the query processor of the RDBMS. We intended to but could not compare against GQ-Fast because the system supports a very limited set of queries (e.g., none of the LDBC queries are supported).

Another approach is to develop a translation layer between a graph data model and query language to the relational model and SQL and leverage the underlying RDBMS without modifications. IBM DB2 Graph [40], SQLGraph [39], and SAP Hana's graph database extension [37] adopt this approach. This is very attractive for commercial vendors because it is lightweight but this approach is not performance focused and is limited by the underlying RDBMS's baseline performance of value-based joins.

SIP is a term used for techniques that use information passed "sideways" from non-local parts of a query plan in an operator o, i.e., from operators that are not immediate children of o. SIP has been used in RDBMSs to avoid scanning large tables or indices or data from remote compute nodes [9, 14, 19, 31, 32, 44]. The use of sip closest to our work has been proposed by Neumann et al. [32] inside the RDF-3X system that manages RDF databases. This work has proposed using sip to avoid scans of large fractions of indices that store RDF triples. This work specifically targets queries with large joins but small outputs that contain sub-queries with non-selective filters. Evaluation of these sub-queries in regular execution requires large index scans, but by passing information from other sub-queries, the system can avoid scanning parts of the index. Zhu et al. [44] similarly used sip to avoid large table scans in in-memory star schema data warehouses when using left-deep query plans in queries. Our use of sip to integrate predefined joins is similar to the use of sip in these works with several differences. In these systems, joins are value-based so passing the values, which may be of arbitrary data types, requires compacting the keys in probabilistic data structures, specifically bloom filters. This requires running hash functions both when creating the filter as well as performing the semi-joins in scans. Since our pointer-based predefined joins are over dense integer-based ID, we can directly compact the keys in a deterministic bitmap filter and avoid any hash computations. As was done in reference [44], we also demonstrate that using sip in graph or relational workloads with large many-to-many joins make the system more robust by analyzing GRainDB's plan space.

The analogue of adjacency list indices in our solution is the *RID indices* (Section 6) that we use to index tables that are part of predefined joins. Our RID indices can be seen as a form of join index [22, 24, 33, 34, 41, 42]. Valduriez originally introduced join indices [42] to index results of arbitrary join queries, e.g., consisting of equality or inequality predicates, and index for each RID of one table, the list of matching RIDs from one or more other tables. Similar to join indices, our RID indices store RID keys and list of RID values but are over base tables instead of results of join queries.

Our use of RID indices is closest to reference [33]'s use of bitmapped join indices in select-project-join queries over databases with star schemas. This work uses one join index Ind_i for each dimension table D_i. Ind_i indexes for each RID j of D_i the list of RIDs of fact table F that j joins with. The assumption is that D_i are small, so each RID list is very large, so it is more efficient to store RID lists as bitmaps. The authors consider a specific query template, where a set of D_i that have separate selection predicates p_i are joined with F. Then by computing the AND/OR of the bitmaps b_i of each RID j in D_i that passes p_i, a bitmask filter is generated that identifies the RIDs of F that are part of the join. Finally, for each tuple t_k from F that passes this filter, the tuple from D_i that join with t_k is fetched. In contrast, our approach does not compress RID lists in RID indices nor use fast bit operations. Instead, inside modified hash-join operators, we use uncompressed RID lists to generate bitmapped filters and pass these filters to downstream scans through sip. At the same time, our approach is more general and can be applied to any query, specifically non-star joins, that contain predefined joins. Finally, our *extended RID indices* facilitate joins between 3 tables (instead of only 2 tables) and can be used to reduce the number of join operations in some special cases (Section 6.2).

Join indices are a form of simple materialized views [6, 16], except they store RID values from two or more tables instead of the actual output columns of join queries. We also compare our approach to a setup where instead of predefining a join, we create an equivalent materialized view. As such, materialized views reduce the number of joins in the query, but unlike our approach, they alone cannot avoid scanning large tables when there are other predicates on the tables that represent nodes or edges. As we demonstrate, on many queries in our workloads, materialized views indeed improves performance over vanilla DuckDB but not as much as GRainDB.

3 PRELIMINARIES

In Sections 4-7, we describe our proposed approach of using hash join- and sip to implement predefined joins in RDBMSs. Throughout these sections, we assume that we are given a select-project-join query Q and the underlying RDBMS has generated a query plan T for Q that consist of a tree of table scan, filter, projection, and join operators, where the joins are inner joins. Our techniques modify T to generate a new query plan T' that replaces some of the join operators with modified join operators called `SJoin`, `SJoinIdxR`, or `SJoinIdxM` and some of the scan operators with a modified scan operator called `ScanSJ`. We first describe our changes to the physical storage of relations when users predefine joins in Section 4. We then describe our new operators and RID indices in Sections 5 and 6 along with our rule-based plan transformation algorithm. Finally we discuss several implementation details in Section 7.

4 RID MATERIALIZATION

Users predefine their joins using a `PREDEFINE JOIN` command that we added to the SQL dialect in DuckDB. In this command users specify an equality join from a table $F(A_{f1}, ...A_{fk_f})$ to $P(A_{p1}, ..., A_{pk_p})$ on attributes $A_{ft_1} = A_{pz_1}$, ..., $A_{ft_\ell} = A_{pz_\ell}$, such that $A_{ft_1}, ..., A_{ft_\ell}$ forms a foreign key to P. In response, the system adds a new column $\texttt{RID}(A_{ft_1}, ..., A_{ft_\ell})$ to F that contains for each row $r_f \in F$, the RID

Table 1: Input tables for our running example.

(a) `Person` table.

Person

ID	name
101	Mahinda
202	Karim
303	Carmen
404	Zhang

(b) `Follows` table.

Follows

ID1	ID2	year
101	202	2021
303	404	2019
101	303	2021
202	303	2020
101	404	2021

Table 2: Extended tables. RID columns are abbreviated as `VR` and are in gray to indicate that they are not materialized. The `RID(ID`$_i$`)` columns of `Follows'` are abbreviated as `R`i.

(a) Extended `Person` table.

Person

VR	ID	name
0	101	Mahinda
1	202	Karim
2	303	Carmen
3	404	Zhang

(b) Extended `Follows` table.

Follows′

VR	R1	ID1	R2	ID2	year
0	**0**	101	**1**	202	2021
1	**2**	303	**3**	404	2019
2	**0**	101	**2**	303	2021
3	**1**	202	**2**	303	2020
4	**0**	101	**3**	404	2021

of the row $r_p \in P$ to which r_f has the foreign key. This column is visible only to the system and not to users. RIDs in columnar RDBMSs serve as system-level pointers and can be used to compute the locations of rows in storage. Therefore the `RID(`$A_{ft_1}, ..., A_{ft_t}$`)` column stores for each r_f the pointer to the matching r_p. Multiple joins on F can be predefined, which is common for relationship tables that represent many-to-many joins between two entity tables.

Example 1. *Table 1 shows a simple database with two tables, a* `Person(ID, name)` *table and a* `Follows(ID1, ID2, year)` *table, that will serve as our running example. The* `ID1` *and* `ID2` *columns in* `Follows` *are both foreign keys to the* `ID` *column of* `Person`*. Table 2 shows the extended* `Follows` *table (as* `Follows'`*) when a user predefines the* `Person.ID = Follows.ID1` *and* `Person.ID = Follows.ID2` *joins. The* `Follows` *table is extended with* `RID(ID1)` *and* `RID(ID2)` *columns (abbreviated as* `RID1` *and* `RID2`*) that contain the RIDs of the rows in Person that match the values in the* `ID1` *and* `ID2` *columns, respectively. Both* `Person` *and* `Follows` *tables also have RID columns (abbreviated as* `VR`*) that show the contiguous RIDs of the rows in these tables. These are shown in gray to indicate that unlike* `RID(ID1)` *and* `RID(ID2)` *columns, they are not materialized in storage.*

5 SJOIN: THE SIP OF RIDS

Our implementation of predefined joins consists of two steps:

Step 1: Rule-based query optimization. Algorithm 1 gives the pseudocode of our first rule-based optimization algorithm (a second one is described in Section 6.2). We recursively traverse the system's default logical plan T for a query (lines 1 and 2) and find each join operator that evaluates a predefined join from F to P. In our implementation, these are `HashJoin` operators because DuckDB evaluates equality joins with `HashJoin`. Upon finding these `HashJoin`s, we perform one of two actions. We review the first case here and will review the second case in Section 6.1:

Join Replacement Case 1 (Lines 8-10): F is on the build and P is on the probe side. In this case, we make the following changes to T:

- `HashJoin` is replaced with a new operator called `SJoin` (line 9).

Algorithm 1 ReplaceHashJoinsWithSJoins. T.op refers to the root operator in a (sub-) plan tree T.

input: DuckDB's optimized plan tree T
output: A plan tree T^* possibly with `SJoin`/`SJoinIdxR` operators.

```
1:  T*_l ← ReplaceHashJoinsWithSJoins(T_l);
2:  T*_r ← ReplaceHashJoinsWithSJoins(T_r);
3:  if ((F, P) = FindPredefinedJoin(T.op)) ≠ NULL then
4:      return RewriteJoinAndScans(T, F, P);
5:  return T;
6:
7:  function RewriteJoinAndScans(T, F, P):
8:  if (F is on the build and P is on the probe side) then
9:      (i) replace T.op with SJoin;
10:     (ii) replace the scan of P with ScanSJ of P.
11: else if P is on the build and F is on the probe side and the
    appropriate RID index on F exists then
12:     (i) replace T.op with SJoinIdxR;
13:     (ii) replace the scan of F with ScanSJ of F.
```

- `Scan`(F) operator (on the build side sub-tree) is modified to scan the materialized RID column of F (omitted in Algorithm 1).
- Every probe-side `Scan`(P) operator is replaced with a modified scan operator, which we call `ScanSJ` (for **s**emi **j**oin) (line 10).

Step 2: Sideways information passing during query evaluation: Suppose an `SJoin` operator has replaced a `HashJoin` operator in T and a `ScanSJ`(P) has replaced the standard `Scan`(P) in the Join Replacement Case 1. Then during query evaluation we use sip to pass information from `SJoin` to `ScanSJ`(P) as follows. `SJoin` is a specialized hash join operator that first reads all of the tuples from its build side. These tuples contain materialized RID values in F and point to the tuples in P. `SJoin` then constructs two bitmask filters:

- *Zone bitmask:* For each zone of P, i.e., a block of tuples on disk, indicates whether the zone has any matching tuples joining with F. This bitmask contains 1 bit for each zone and is constructed by taking the modulo of the RIDs with the zone size.
- *Row bitmask:* Indicates whether each row r_p of P joins with an F tuple. This bitmask contains $|P|$ many bits and is constructed by directly setting the positions of the seen RIDs to 1.

`SJoin` then passes both of these bitmasks to all of the `ScanSJ`(P) operators in its probe side recursively. The zone bitmask is used to skip over scanning zones of P whose bits are 0. For zones with matching tuples, The `ScanSJ` operator scans the zone into vectors as regular scan operator and adds a new RID vector to the intermediate tuples that store the RIDs of the scanned tuples. Finally, `ScanSJ` attaches the row bitmask of this zone as a selector vector, which filters out the P tuples without matching F tuples, to the intermediate tuples.

Note that passing bitmask filters sideways from `SJoin` operator down to all `ScanSJ`(P) operators is a safe operation in our setting of select-project-join queries. Observe that the bitmask filter generated by `SJoin` is the set of RIDs of P that will successfully join with F tuples in the `SJoin`. Let $P.R$ be the (virtual) RID column of P. One can therefore think of the bitmask filter as a runtime predicate on $P.R$. One can further think of applying this predicate as the last operator of the probe-side of `SJoin`. Therefore by passing the bitmask

filter to the ScanSJ(P) operators, we are pushing this predicate down the probe side sub-tree. By algebraic rules for pushing down predicates [12], this predicate can be pushed down through any selection, projection, or (inner) join operator. In general, this information cannot be safely passed to all ScanSJ operators, e.g., if the probe-side sub-tree of SJoin contains a group by and aggregation, limit, or outer join operator. We give an example.

Example 2. *Consider a query that finds two hop friends of Karim:*

```
SELECT *
FROM Pers. P1,Follows F1,Pers. P2,Follows F2,Pers. P3
WHERE P1.ID=F1.RID1 AND F1.RID2=P2.ID AND P2.ID=F2.RID1
    AND F2.RID2=P3.ID AND P1.name = Karim
```

Figure 1a shows an example plan for this query that has: (i) replaced two HashJoins *with* SJoin *operators; (ii) replaced two* Scan Person *table operators (for* P2 *and* P3*) with* ScanSJ*; and (iii) modified the* Scan Follows *operators to read the materialized RID columns.* HashJoin_1 *and* HashJoin_2 *operators are not replaced with* SJoin *because the* Scans *of F1 and F2 are on their probe sides.* SJoin *operators pass two bitmasks, shown at the* ScanSJ P2 *and* ScanSJ P3 *operators. The top one is the zone bitmask and the bottom one is the row bitmask. The figure assumes zones of size 2. In our running example,* HashJoin_1 *joins the (1, 202, Karim) and (1, 202, P2.RID=2, 303, 2020) tuples, which is given to* SJoin_1*. Because the only matching* P2 *in this tuple has RID 2, the row bitmask passed to* ScanSJ P2 *is [0, 0, 1, 0] and the zone bitmask is [0, 1].* ScanSJ P2 *only scans the second zone and puts the [1, 0] selector vector to the two tuples in this zone (filtering out the tuple with RID 3). The output of* SJoin_1 *is (1, 202, Karim, 2, 303, Carmen, 2020) and the following* HashJoin_2 *produces (1, 202, Karim, 2, 303, Carmen, 2020, P3.RID=3, 404, 2019). This is given to* SJoin_2 *(during build), which passes the [0, 1] zone bitmask and the [0,0,0,1] row bitmask to* ScanSJ P3*. The final output is (1, 202, Karim, 2, 303, Carmen, 2020, 3, 404, Zhang, 2019).*

6 THE RID INDEX AND ITS APPLICATIONS

Next, we describe two applications of indexing the materialized RIDs in table F in an index. We call this index the *RID index*.

6.1 The RID Index and Reverse Information Passing

In our approach of evaluating predefined joins so far, we can pass RID values only from F to scans of P and not vice versa. In many settings, F is a much larger table than P, and the ability to filter tuples of F by obtaining information from P can be very beneficial. For example, in LDBC benchmark with scale 30, Knows table is 41x larger than Person. However, given a row $r_p \in P$, we cannot directly find from the RID value of r_p the RIDs of rows $r_{f1}, ..., r_{fp} \in F$ that join with r_p, as this list is not materialized in P. Instead, we construct a RID index on F that for each r_p returns this list. In our implementation, users can construct RID index on any table F on which at least one join has been predefined (say to a table P). Therefore, F already has a materialized $\text{RID}(A_{it_1}, ..., A_{it_\ell})$ column and its own virtual RID column. The RID index stores for each value in the $\text{RID}(A_{it_1}, ..., A_{it_\ell})$ column the RIDs of $r_{f1}, ..., r_{fp} \in F$ that join with r_p. Similar to adjacency list indices in GDBMSs, we store the RID index using a compressed sparse row data structure [10].

When there is a RID index, we apply the following transformation rule, which is the second Join Replacement Case in Algorithm 1: *Join Replacement Case 2 (Lines 11-13): P is on the build and F is on the probe side and there is a RID index on F for this predefined join.* In this case we make the following changes to the plan tree:

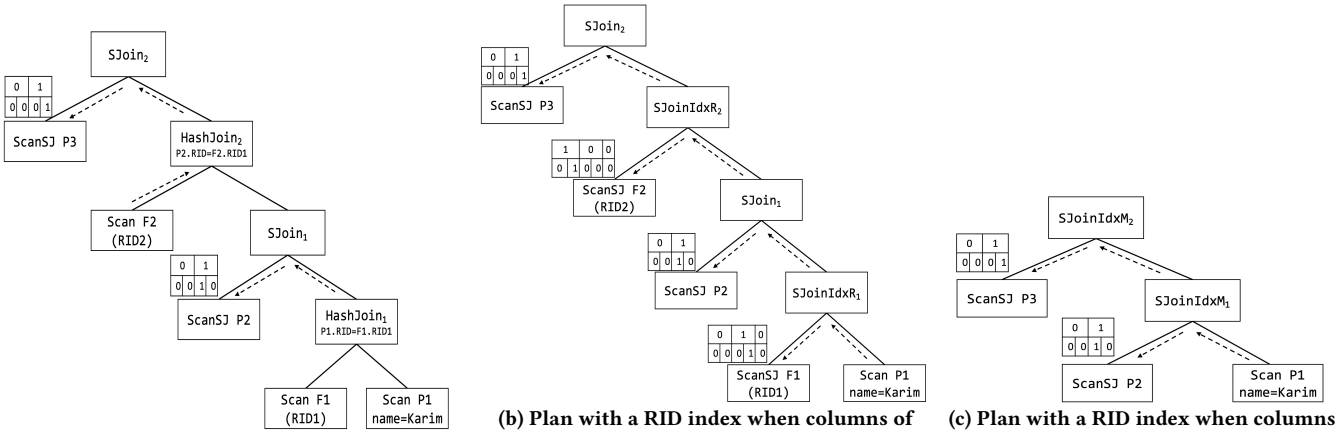
- HashJoin is replaced with an operator called SJoinIdxR (line 12).
- Scan(P) operator (on the build side sub-tree) is modified to scan the materialized RID column of P (omitted in Algorithm 1).
- Every probe-side Scan(F) operator is replaced with ScanSJ.

Similar to SJoin, SJoinIdxR builds a hash table, now of tuples from P and constructs the bitmasks for the sip as follows: For each tuple r_p from the build side, SJoinIdxR consults the RID index to find the RIDs of the F tuples that join with r_p and sets the bits corresponding to these RIDs. Then, these bitmasks are passed to the ScanSJ(F) operators, which filter out the F tuples without any matching P tuples in SJoinIdxR. The pseudocode of SJoinIdxR is shown in Algorithm 2 in the longer version of our paper [20]. Note that the RID index is a join index and generates the RIDs of all of the matching tuples F. Therefore, by the same argument we made in Section 5, passing down of these bitmasks down to ScanSJ(F) is safe in our setting, as it follows the algebraic rules for pushing down predicates [12].

Example 3. *Figure 2 shows the RID index that indexes the (RID1, RID) columns of the* Follows *table. Ignore the* Follows(RID2) *values in the figure for now. Figure 1b shows the plan we now generate in presence of this RID index. The two* HashJoin *operators from the plan in Figure 1a are replaced with* SJoinIdxR *operators and the previous* Scan *operators of the* Follows *table are replaced with* ScanSJ *operators. The figure also shows bitmasks that the new* ScanSJ *operators take. For example, the* ScanSJ $F1$ *operator takes a row bitmask with only the index 3 set to 1 and zone bitmask with only index 2 set to 1. This is because the RID of the (1, 202, Karim) tuple is 1 and 1's list of matching RIDs contains only the RID 3 of* Follows*, because 202 joins with (3, 1, 202, 2, 303, 2020) (see Table 2).*

6.2 The Extended RID Index and Join Merging

Many-to-many joins between two tables P_1 and P_2 that represent two (possibly same) sets of entities are often facilitated through a third relationship table F. This is, for example, the case in our running example, where the Follows table is joined with two Person tables. Therefore, it can be beneficial to predefine two joins on F. In this case, each row r_f of F would contain the virtual RID of F and two materialized RIDs, one for row $r_{p_1} \in P_1$ and the other $r_{p_2} \in P_2$ that r_f joins with. Consider building a RID index from the RIDs of P_1 to lists of the RIDs of F tuples. For each RID of P_1, say i_1, we store a list $L_{i_1} = \{r_{f_i}, ..., r_{f_k}\}$ of RIDs of F tuples that have i_1 in their materialized RID column for P_1. We can also extend L_i to store the RIDs of P_2 tuples along with the RIDs of F as follows: $\{(r_{f_i}, r_{p2_1}), ..., (r_{f_k}, r_{p2_k})\}$. This is similar to how GDBMSs store both the edge IDs and neighbor node IDs in their adjacency lists. Analogous to *forward* and *backward* adjacency list indices in GDBMSs, one can similarly build a second RID index that stores for each RID of P_2 a list of RIDs of joining F and P_1 tuples. Figure 2 is an example "forward" extended RID index for the Follows table, that stores for each "source" Person tuple r_p, the list of RIDs of the joining Follows tuples, shown as Follows(RID) values, as well



(a) Plan without a RID index. **(b) Plan with a RID index when columns of `Follows` tables are in the final projection.** **(c) Plan with a RID index when columns of `Follows` tables are projected out.**

Figure 1: Example plans for our running example queries with different system configurations.

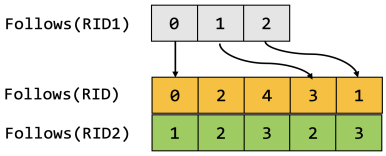


Figure 2: A RID index on Follows in the CSR format.

as the RIDs of the "destination" `Person` tuples that these `Follows` tuples point to, shown as `Follows(RID2)` values.

Consider a query that performs a join of $P_1 \bowtie F \bowtie P_2$ with the predefined conditions, but F is only used to facilitate the join. That is: (i) there are no filters or other joins on F; and (ii) the final projection does not contain any columns of F. We call the scan of F in this case an **unfiltered** scan operator. Then, we can directly join the P_1 tuples with P_2 tuples using an extended RID index, without ever scanning the F table and joining F with P_1 or P_2. This is because for each tuple $t_{p_1} \in P_1$, the extended RID index already contains the RIDs of every tuple t_f of F that joins with t_{p_1} and the RIDs of corresponding P_2 tuples that are materialized in t_f. We call this the *join merging optimization*. Specifically, in our query optimization step, we look for two consecutive join operators J_1, evaluating $P_1 \bowtie F$, and J_1's parent J_2, evaluating $P_2 \bowtie F$ such that conditions (i) and (ii) above are satisfied. We replace J_1 and J_2 with a new `SJoinIdxM` operator S. S takes as its build side J_1's build side and as its probe side J_2's probe side, and we drop the unfiltered scan of F, i.e., the probe side of J_1.

Algorithm 3 in the longer version of our paper [20] shows the pseudocode of the join merging transformation algorithm. Given DuckDB's original plan T, we apply the join merging rule after the `SJoin` replacement rule (Algorithm 1). We note that since both of our transformation rules are deterministic, for each T, our rule-based optimizer always returns a unique T^*. During evaluation, for each P_1 tuple r_p, S looks for the RIDs of joining P_2 tuples directly from the extended RID index, and passes these RIDs as bitmasks to the `ScanSJ`(P_2) operators on its probe side, without ever scanning F. The join with F happens implicitly while accessing the RID index to read the RIDs of P_2 tuples.

Example 4. *Figure 1c shows our plan in the presence of an extended RID index from RID1 to RID2 columns of* `Follows`*. Observe that compared to the plan in Figure 1b, we have merged* $\texttt{SJoinIdxR}_1$ *and* $\texttt{SJoin}_1$ *into a new* $\texttt{SJoinIdxM}_1$ *operator, and* $\texttt{SJoinIdxR}_2$ *and* $\texttt{SJoin}_2$ *into a new* $\texttt{SJoinIdxM}_2$ *operator.*

7 IMPLEMENTATION CONSIDERATIONS

We next elaborate on the optimizer and the update handling under our proposed approach. We use the default join optimizer of DuckDB to generate an initial plan T_d and then replace some of the hash joins in a rule-based approach with our `S-Join` variants to obtain T_d^*. Even if T_d is the best default join order of DuckDB, in principle modifying another plan T with predefined joins can outperform T_d^*. Therefore, one can extend our integration to develop a sip-aware optimizer to generate such plans. In Section 8, we present a plan spectrum study on a subset of the JOB benchmark queries to study the effects of predefined joins on the plan space of DuckDB. We also evaluated how much performance opportunity there was to improve T_d^*, if a sip-aware optimizer could pick the optimal plan T_s^* with predefined joins. Although we found several queries for which we could obtain improvements, broadly we found T_d^* and T_s^* competitive and chose not to modify the optimizer of the system.

Second, although we do not focus on updates, our integration requires further considerations for handling updates. Insertions of a tuple t to a table F that has a predefined join to P requires finding the RID of the tuple in P that t refers to and inserting it in the system-visible $\texttt{RID}(A_{it_1}, ..., A_{it_\ell})$ column F. If there is a RID index on F, possibly an extended one, we need to further update the index. Deletions also require additional handling. Observe that because we use RIDs as pointers, we materialize them in system-visible RID columns or a RID index. Therefore once a tuple t is assigned a RID, it needs to remain fixed. When a tuple t with the RID k is deleted, the system needs to keep track of the gap in k and assign it to the next inserted tuple. The system cannot shift tuples with RIDs $k+1$, which would change a large number of RIDs and require updating the references to these RIDs. Reusing gaps or IDs is common practice both in RDBMSs and GDMBSs, e.g., MySQL [5] reuses gaps left by deleted tuples for new insertions.

8 EVALUATION

We next evaluate our hash join- and sip-based approach of implementing predefined joins with several alternatives. Our approach is implemented inside DuckDB and we call this version of DuckDB as GRainDB.

8.1 Setup

Baseline Systems: We compare GRainDB against the following baseline systems: (i) vanilla DuckDB; (ii) DuckDB-MV, which is the DuckDB configuration where we extend our original databases with a set of materialized views corresponding to the joins predefined in GRainDB; and (iii) GraphflowDB, a state-of-art academic graph database system [15, 21, 28, 29], which uses INLJ-based implementation of predefined joins. We use the version of GraphflowDB from the reference [15]. We also performed preliminary experiments with Neo4j's community edition, but as with several prior work [15, 21, 28] did not find it competitive with GraphflowDB (or GRainDB) on many queries and omit these experiments.

As we explained in Section 2, we also intended but could not compare against GQ-Fast and GR-Fusion. One of our goals was to show that the pure left-deep and INLJ-based plans used by these approaches can be suboptimal to bushy and hash join-based plans of GRainDB. Instead, we will perform this comparison against similar plans from Neo4j and GraphflowDB.

Benchmarks: We expect predefined joins to provide performance improvements on queries with the following properties: *(i) Existence of predefined joins:* As a necessary condition, the query must contain at least one predefined join. *(ii) Existence of selective predicates on F and/or P:* This is critical because when F (or P) has a selective predicate, the bitmask filters used by sip more effectively reduces the scan of P (or F). *(iii) Existence of one/many-to-many joins:* We also expect to see performance improvements when queries contain one/many-to-many joins for two reasons. First, predefined joins primarily improves join performance (as opposed to say aggregations), and the join performance is often an important runtime factor in queries with one/many-to-many joins. Second, the application of reverse information passing from P to F and the join-merging optimizations can primarily benefit queries with one/many-to-many joins. This is because these optimizations require a RID index, which is generally built on tables that represent one/many-to-many relationships between two other tables.

In light of these, we used one relational and one graph benchmark that contain queries that satisfy these properties and for a more complete evaluation, a second relational workload that does not.

- Join order benchmark (JOB) on the IMDB dataset [23], which contains more than 2.5 M movie titles produced by 235K different companies with over 4 M actors. When using GRainDB, we predefine every one-to-many primary foreign key relationship in the database and for tables that represent many-to-many relationships, such as `movie-companies`, we build a RID index.
- LDBC Social Network Benchmark [7] (SNB) benchmark at scale factor 10 and 30, which is a commonly used graph benchmark that models a social networking application with users, forums, and posts. In relational format, LDBC10 dataset contains 8 entity (i.e, node) and 10 relationship (i.e., edge) tables, with a total number of 36.5M and 123.6M tuples, respectively. LDBC30 contains 106.8M entity and 385.2M relationship tuples. We used SNB primarily to compare against DuckDB with materialized views and GraphflowDB (and Neo4j, which was not competitive). GraphflowDB does not implement several language features, such as recursive queries. Therefore, we slightly modified the benchmark and refered to it as *SNB-M*, for **m**odified. We removed queries involving shortest paths and decomposed queries with variable-length joins into multiple queries, each of which has a fixed path join (we denote each version with a suffix -ℓ, where ℓ denotes the length). In the longer version of this paper [20], we list our full SNB-M queries. SNB is generated in both relational and property graph formats. We use the relational format in DuckDB and GRainDB. For every edge type in the graph format of SNB, e.g., `Knows` edges, we build a RID index over the corresponding table in GRainDB. For each predefined join in GRainDB, we create a materialized view that joins two tables and project all their properties in DuckDB-MV, e.g., predefined join Person.ID=Knows.P1ID, we create a materialized view on the subquery SELECT * FROM Person JOIN Knows ON Person.ID=Knows.P1ID. We also created an index on primary key columsn of these views.
- TPC-H benchmark at scale factor 10. We include TPC-H to perform a sanity check that making the primary-foreign key joins on such traditional workloads does not hurt performance. We do not expect GRainDB to provide meaningful improvements on TPC-H as it does not contain selective many-to-many joins. We predefined every one-to-many primary foreign key relationships in GRainDB, such as `customer` and `orders`. Although we did not expect performance speedups, we still found several queries on which we obtained non-negligible runtime improvements.

System Configurations and Hardware: We set DuckDB to the in-memory mode. DuckDB is still in early stage and does not integrate full cardinality estimation. We observed that this limits its ability to choose good join orders on many instances, especially in queries with large joins and selective predicates. GRainDB can improve DuckDB's plans with poor join orders as well as good join orders. We demonstrate this in Section 8.3.3 where we analyze the plan spectrums of GRainDB and DuckDB. However, using poor join orders decreases GRainDB's competitiveness with GraphflowDB. To isolate the influence of join order selection, we injected the true cardinalities of sub-queries into the system. In the longer version of this paper [20], we present a demonstrative experiment that shows that: (i) GRainDB's improvement factors over DuckDB is similar when we use DuckDB's default cardinality estimates or true cardinalities; and (ii) injecting true cardinalities improves the performance of both DuckDB and GRainDB. We note that making true cardinality estimates is not realistic in practice. To verify that GRainDB does not strictly require true cardinalities to perform well, we also injected into the system Postgres's cardinality estimates (also presented in the longer version of this paper [20]). We observed that using Postgres's cardinalities leads to very similar results to injecting true cardinalities. In light of this demonstrative experiment, we present DuckDB's performance under true cardinalities in the following sections.

GraphflowDB is already an in memory system. The GraphflowDB version we use does not contain an optimizer, so does not need to estimate cardinalities. We manually picked the systems' best join order, which for many queries was obvious.

All experiments were conducted on a machine with two Intel E5-2670 @2.6GHz CPUs and 256 GB of RAM, consisting of 16 physical cores and 32 logical cores. Because GraphflowDB runs only in serial mode, we set DuckDB to run in serial mode as well. All reported times are averages of five successive runs after a warm-up running.

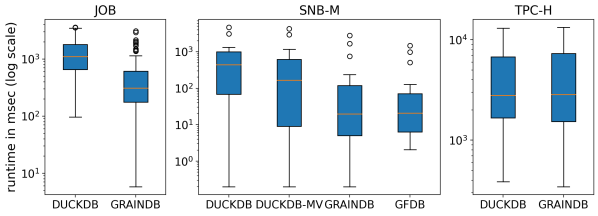


Figure 3: Runtimes (in ms) of DuckDB and GRainDB on JOB, SNB-M and TPC-H, and DuckDB-MV and GraphflowDB on SNB-M.

Table 3: Detailed percentiles of runtimes (in ms) for DuckDB and GRainDB on JOB.

	Min	5th	25th	50th	75th	95th	Max
DuckDB	96.0	203.6	652.4	1110.0	1797.0	2939.9	3584.6
GRainDB	5.8	27.4	176.4	309.0	614.2	1878.5	3104.4

Our measurements reflect the end-to-end query evaluation time, and a timeout of 10 minutes is imposed on each running.

8.2 End-To-End Benchmarks

We first present end-to-end evaluations on JOB, SNB-M, and TPC-H.

8.2.1 JOB: Relational Workload with Selective Many-to-Many Joins. The box plots of DuckDB and GRainDB on JOB are shown in Figure 3. Each boxplot shows the distribution of the runtimes of the queries in the workloads, specifying the 5th, 25th, 50th, 75th, and 95th percentiles of the distribution with marks. As we expect, we observe GRainDB outperforms DuckDB by large margins. JOB contains 113 queries. Table 3 lists detailed percentiles for query execution times of DuckDB and GRainDB of these queries. We see consistent large runtime reductions for each percentile. For example, the 25th percentile, median, and 75th percentile query execution times reduce respectively from 652.4ms to 176.4ms (3.7x), from 1110ms to 309ms (3.6x), and from 1797ms to 614.2ms (2.9x). For reference, Table 4 presents the execution times of a subset of the queries in JOB. Specifically, JOB queries contain between 2 to 6 variants and we present the first variant of each query in the table. The full table can be found in the longer version of this paper [20]. Importantly, we see consistent runtime improvements on all queries, with a few exceptions. Table 4 also presents the reduction on the amount of scanned tuples for each query in DuckDB and GRainDB. Although runtime reductions depend on many factors, the reductions in scanned tuples is a good proxy for explaining when sip and predefined joins improve performance. For example, we observe that the queries in which we observe the largest improvement factors, such as Q6a, Q21a, Q27a and Q32a, have large reductions in scanned tuples by 348.9x, 182.2x, 185.4x, and 53.8x. In contrast, queries with negligible improvements, such as Q5a and Q20a have no or small (1.3x) reductions in scanned tuples.

8.2.2 SNB-M: Graph Workload with Selective Many-to-Many Joins. The box plots of DuckDB, GRainDB, and GraphflowDB on SNB-M are shown in Figure 3 (ignore the DuckDB-MV bar chart for now). Table 5 also lists detailed percentiles for query execution times of the systems (simiarly, ignore the DuckDB-MV row for now). We see that GraphflowDB outperforms DuckDB by large margins on SNB-M. Specifically for the 25th percentile, median, and 75th percentile query execution times, GraphflowDB outperforms DuckDB respectively by 10.7x (68.4ms vs 6.4ms), 22.5x (441.8ms vs 20.8ms), and 14.1x (989.0ms vs 70.3ms). However, by predefining the joins in SNB-M, GRainDB closes this gap significantly, making DuckDB competitive with GraphflowDB. Specifically for the 25th percentile, median, and 75th percentile query execution times, GRainDB and GraphflowDB compare as follows: 5.0ms vs 6.4ms (0.78x), 19.6ms vs 20.8ms (0.94x), and 119.4ms vs 70.3ms (1.7x).

Table 6 shows the detailed execution times of each query for all systems (again ignore the DuckDB-MV row for now). We see that GRainDB outperforms DuckDB on almost all queries by up to 90x, except for IS1 and IS4, which are two small queries executed within 2ms. Similarly, GraphflowDB outperforms DuckDB in most queries by up to 426.6x. We observe that there are also 9 queries in which GRainDB outperforms GraphflowDB by large margins. We analyzed each of these queries to study GRainDB's performance advantages. First are IS01 and IS04-IS07, which are point lookup queries over large base tables with inexpensive joins. Here, GraphflowDB resorts to sequential scans of these tables/nodes while GRainDB (and DuckDB) uses a primary key index. This is not an inherent limitation of GraphflowDB plans and can be remedied if GraphflowDB also supports primary key indexes. On the remaining 4 queries, GRainDB plans have two separate advantages.

- *Bushy vs Left-deep Plans:* IC1-3, IC6-2, IC11-2 are three queries in which GRainDB outperforms DuckDB and uses a bushy plan. We describe IC6-2 as an example. IC6-2 is a complex query with 8 joins in SQL. In graph version, this is a 5-path query with selective predicates on both ends of the path. GraphflowDB does not implement bushy plans, so this query is implemented with a left-deep plan. This is less performant than the bushy plan that GRainDB uses that breaks the path into two parts. In the longer version of this paper [20], we show the plans from both systems. This is an example of when the left-deep plan-based approaches to evaluate such path queries, which are used in systems like GR-Fusion and GQ-Fast, can be suboptimal to bushy plans.
- *Hash Join vs Index Nested Loop Joins and Scanning Edges Before vs After Joins:* The IC9-2 query is a smaller query with 4 joins in SQL. This is a 3-path query that also has filters on both ends. Now both systems use left-deep plans. The majority of the time in this query is spent in the very last join, which requires joining 7681 tuples from a `Person` table with a `Comments` table. 2.7M of these tuples successfully join with the 7681 tuples and 2.4M of these also pass a filter on the `Comments` table. In graph terms, 7681 `Person` nodes have 2.7M outgoing `Comment` edges (so an average degree of 351). GraphflowDB follows these steps: (i) *joining nodes with edges*: looks up the edges of each of the 7681 keys in a large adjacency list index that point to `Comments`. This performs 7681 random lookups into a hash table of size 26.5M, and then generates 2.7M intermediate tuples. (ii) *property scan and edge filtering*: reads the necessary properties of the 2.7M `Comments` and runs the filter predicates on these edges. In contrast, GRainDB follows these steps: (i) *hash table build:* creating a hash table of size 7681; (ii) *edge scanning and filtering:* sequentially scanning a large `Comments` table with 26.5M tuples and running a predicate on them which returns 2.4M tuples; (iii) *joining nodes with edges:*

Table 4: Runtimes (in ms) of DuckDB and GRainDB on each query in JOB.

	1a	2a	3a	4a	5a	6a	7a	8a	9a	10a	11a	12a	13a	14a	15a	16a	17a
DuckDB	234.2	207	1491.4	216	96	885.4	879.8	1307.4	1933.2	1404.2	264.2	684	981.2	1644.2	987.8	652.4	1164.8
GRainDB	34.2	154.0	328.0	114.2	116.4	57.0	203.4	349.8	554.8	614.2	44.4	242.4	336.8	202.2	283.4	177.6	486.0
	6.8x	**1.3x**	**4.5x**	**1.9x**	**0.8x**	**15.5x**	**4.3x**	**3.7x**	**3.5x**	**2.3x**	**6.0x**	**2.8x**	**2.9x**	**8.1x**	**3.5x**	**3.7x**	**2.4x**
Scan Reduction	751.2x	23.9x	97.2x	58.7x	1x	348.9x	47.0x	9.2x	6.0x	13.0x	73.7x	44.8x	42.4x	142.1x	7.4x	14.4x	10.7x

	18a	19a	20a	21a	22a	23a	24a	25a	26a	27a	28a	29a	30a	31a	32a	33a
DuckDB	1797.0	2632.4	1118.4	1629.2	1471.8	866.6	2554.8	2318.8	1074.6	761.4	2068.4	2742.6	2198.0	2523.8	126.0	336.2
GRainDB	612.6	491.6	1071.6	54.2	864.0	296.8	788.8	1376.6	733.4	44.4	240.2	266.6	673.0	612.0	8.2	149.4
	2.9x	**5.4x**	**1.0x**	**30.1x**	**1.7x**	**2.9x**	**3.2x**	**1.7x**	**1.5x**	**17.1x**	**8.6x**	**10.3x**	**3.3x**	**4.1x**	**15.4x**	**2.3x**
Scan Reduction	13.8x	7.8x	1.3x	182.2x	7.6x	8.2x	8.7x	32.2x	15.7x	185.4x	156.9x	10.8x	282.2x	165.5x	53.8x	2.6x

Table 5: Detailed percentiles of runtimes (in ms) for DuckDB, GRainDB, and GraphflowDB on SNB-M.

	Min	5th	25th	50th	75th	95th	Max
DuckDB	0.2	1.5	68.4	441.8	989.0	2762.5	4647.0
DuckDB-MV	0.2	0.48	9.0	166.0	612.0	2580.7	4258.0
GRainDB	0.2	0.7	5.0	19.6	119.4	1482.4	2768.0
GraphflowDB	2.1	2.5	6.4	20.8	70.3	888.2	1473.6

and finally doing 2.4M lookups into this very small hash table and performing the join. Now the joins happen after a sequential scan and filter of the "edge" table, leveraging columnar RDBMS techniques highly optimized for sequential scans and filters of large columns. In addition, in the final join, now the lookups are performed on a very small hash-table instead of a large adjacency list index. This is more performant than performing the joins by lookups into a large index and non-sequentially scanning and filtering the joined edges. We will present a more controlled experiment to demonstrate this difference in Section 8.3.2.

8.2.3 Comparison against Materialized Views. Materialized views is a commonly used technique to improve the performance of DBMSs. The goal of our next experiment is to compare the performance of our predefined joins to a DuckDB configuration that has one corresponding materialized view for each of the joins we predefine on SNB-M. We call this configuration DuckDB-MV. Specifically, for each predefined join between $F \bowtie_\rho P$ that we have in our GRainDB setup for SNB-M, we define a materialized view $MV_{F \bowtie_\rho P} = \Pi_*(F \bowtie_\rho P)$. ρ here is an abbreviation for the the equality condition between F and P and Π_* represents selecting all columns from F and P. Note that the primary benefit we expect from a materialized view is to avoid performing a binary join and instead replace it with a scan of the view. To obtain this benefit, we select all columns of F and P in $MV_{F \bowtie_\rho P}$. If non-join columns are projected out, we cannot replace the predefined join with a scan of $MV_{F \bowtie_\rho P}$ in our queries because all of SNB-M queries contain projections or predicates on non-join columns of F or P for each of our predefined join. To run these projections or predicates, we would need to join $MV_{F \bowtie_\rho P}$ with P or F, defeating the primary purpose of using $MV_{F \bowtie_\rho P}$. We could alternative define more granular views with additional predicates, but this would limit the use these views to only queries with those exact predicates. Instead in our approach, we can benefit from our implementation of predefined joins under arbitrary additional predicates on P and F.

DuckDB does not have native support for materialized views, therefore, we cannot let the system decide which views to use. Instead, we created each $MV_{F \bowtie_\rho P}$ as separate table in DuckDB and manually rewrote each query Q in SQL, in all possible ways. Our algorithm can be seen as a simplified and brute-forced version of the transformational query rewrite algorithm described in reference [13]. The algorithm first rewrites Q in all possible ways with one materialized view by enumerating every pair of joins in the FROM clause and checks if it can be replaced by a view. This gives us rewrites $Q_{1,1}, ..., Q_{1,k}$. Then we iteratively take each query rewrite $Q_{k,i}$ and repeat the above step by only considering the remaining base tables in the query. We report the best performance number of DucDB-MV across all of its rewritings.

Figure 3 shows the boxplot for DuckDB-MV, Table 6 shows the performance of DuckDB-MV on each query, and Table 5 shows DuckDB-MV's detailed runtime percentiles. Similar to GRainDB, DuckDB-MV consistently outperforms vanilla DuckDB. However, with a few exceptions, the performance improvements are less than GrainDB. We analyzed the GRainDB and DuckDB-MV plans and observed that in many cases GRainDB outperforms DuckDB-MV, because it can avoid scans of largest tables by sip that DuckDB-MV plans cannot. We take IC03-2 as an example. This query contains a join between `Place` and `Comments` with a selective filter `pl_name='China'` on `Place`. `Comments` is a much larger table than `Place` with 26.5M tuples vs 1460 tuples of `Place`. In DuckDB-MV, this query runs the filter predicate directly on the large $MV_{Place \bowtie Comment}$ view of size 26.5M. Instead GRainDB runs this predicate on the small `Place` table, and through sip reduces the scan of of `Comment` table from 26.5M to 3.8M, leading to the join of two small tables. Although DuckDB-MV avoids this join completely, this benefits is not larger than the cost of scanning the large view. In the longer version of this paper [20], we show the plans for DuckDB-MV using $MV_{Place \bowtie Comment}$ and GRainDB on this query. Similar behavior exists on several other queries (e.g., IC06-1, IC07, and IC08). The only queries where DuckDB-MV outperforms GRainDB are IS1 and IS7. As we explained in Section 8.2.2 both of these are point look up queries with inexpensive joins and vanilla DuckDB already outperforms GRainDB on these queries.

In our detailed evaluations in Section 8.3.4, we analyze the storage overheads of our RID columns and compare it against materialized views (and adjacency lists of GraphflowDB).

8.2.4 TPC-H: Traditional OLAP Workloads. For completeness of our work, we also compared the performances of DuckDB and GRainDB on TPC-H, which does not contain many queries with selective many-to-many joins. The box plots of DuckDB and GRainDB are shown in Figure 3. The longer version of this paper [20] shows detailed execution time of each query. As expected we do not see

Table 6: Runtimes (in ms) of DuckDB, DcukDB-MV, GRainDB, and GraphflowDB on each query in SNB-M.

	IS1	IS2	IS3	IS4	IS5	IS6	IS7	IC1-1	IC1-2	IC1-3	IC2	IC3-1	IC3-2
DuckDB	0.8	524.8	36.6	0.2	4.5	148.0	989.0	38.0	72.0	110.5	926.0	1177.8	4647.0
DcukDB-MV	0.8	133	3.8	0.2	0.4	42.0	3.0	9.0	38.0	94.0	906.0	1159.6	4258.0
	1.0x	**3.9x**	**9.6x**	**1.0x**	**11.3x**	**3.5x**	**329.7x**	**4.2x**	**1.9x**	**1.2x**	**1.0x**	**1.0x**	**1.1x**
GRainDB	1.2	19.6	3.4	0.2	0.6	5.0	11.0	4.0	6.4	38.2	134.8	119.4	1665.0
	0.7x	**26.8x**	**10.8x**	**1.0x**	**7.5x**	**29.6x**	**90.0x**	**9.5x**	**11.2x**	**2.9x**	**6.9x**	**9.9x**	**2.8x**
GraphflowDB	6.8	3.0	2.5	42.7	82.7	66.4	72.0	2.1	6.4	70.3	47.8	11.5	505.4
	0.1x	**175.8x**	**14.5x**	**0.005x**	**0.05x**	**2.2x**	**13.7x**	**18.3x**	**11.3x**	**1.6x**	**19.4x**	**102.6x**	**9.2x**

	IC4	IC5-1	IC5-2	IC6-1	IC6-2	IC7	IC8	IC9-1	IC9-2	IC11-1	IC11-2	IC12
DuckDB	402.0	636.0	3125.0	244.6	471.2	1186.8	1017.0	441.8	1312.6	35.8	68.4	788.4
DuckDB-MV	166.0	612.0	2936.0	223.0	442.0	336.0	967.0	414.0	827.0	6.0	42.0	597.0
	2.4x	**1.0x**	**1.1x**	**1.1x**	**1.1x**	**3.5x**	**1.1x**	**1.1x**	**1.6x**	**6.0x**	**1.6x**	**1.3x**
GRainDB	54.0	174.0	2768.0	13.0	22.0	33.2	14.0	113.6	752.0	2.8	9.0	234.8
	7.4x	**3.7x**	**1.1x**	**18.8x**	**21.4x**	**35.7x**	**72.6x**	**3.9x**	**1.8x**	**12.8x**	**7.6x**	**3.4x**
GraphflowDB	12.3	20.8	984.0	8.1	127.2	2.8	2.8	55.6	1473.7	2.6	14.4	28.8
	32.6x	**30.6x**	**3.2x**	**30.2x**	**3.7x**	**426.6x**	**359.5x**	**7.9x**	**0.9x**	**13.8x**	**4.8x**	**27.3x**

significant speedups or slowdowns on this benchmark. GRainDB replaces value-based hash joins with predefined joins in 13 of the 22 queries in TPC-H. The median runtime improvement out of these queries is 1.1x, with the maximum slow-down and speedup of 0.8x (so 1.2x slowdown) and 2.6x, respectively. Interestingly, even on a benchmark of traditional analytical queries, we found two queries with one/many-to-many joins on which replacing value-based joins with predefined joins lead to visible speedups (2.6x for Q2 and 1.8x for Q3) and no queries visibly slowed down, indicating the low performance overheads of our implementation when queries are not suitable to benefitting from predefined joins.

We further analyzed the 13 queries in which GRainDB replaced DuckDB's original plans to understand why these replacements sometimes lead to large and sometimes small or no improvements. Although the reasons that determine the exact performance factors are naturally query-specific, we made two main observations. First, in the queries with very small improvements, sometimes the DuckDB plan has a large relationship table F (e.g., the `lineitem` table with 60M tuples) on the build side of the hash join and the scan of F is dominant runtime factor. In this case, because the information is passed from F to the already small probe side table P, the benefits of reducing the scans of P is small. Q14 is an example for this case. Second, for queries whose performance is dominated by the join of F and P, when the large F table is on the probe side, then, the improvement factor depends on the selectivity of the information passed from P. For example in Q3, where we obtain a 1.8x improvement, the bitmask passed from the `orders` to `lineitem` reduces the scan of 60M tuples to 5.8M. In contrast, Q5 has a sub-plan with a very similar join, yet the information passed is less selective (reduces 60M to 9.1M) and we obtain a 1.3x improvement.

8.3 Detailed Evaluation

We next provide a more detailed evaluation consisting of (i) an ablation study to verify that each of the optimizations in our proposed hash- and sip-based approach leads to additional performance benefits; (ii) a controlled experiment comparing the performances of INLJ-based plans and hash-join-based plans when joining relationship tables with entity tables under varying selectivities; and (iii) an analysis of the effects of our sip-based predefined joins in the plan space of DuckDB on a suite of queries.

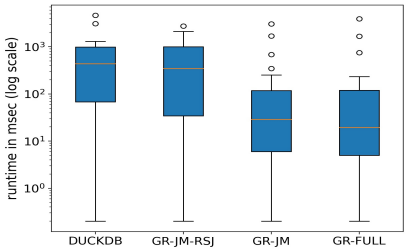
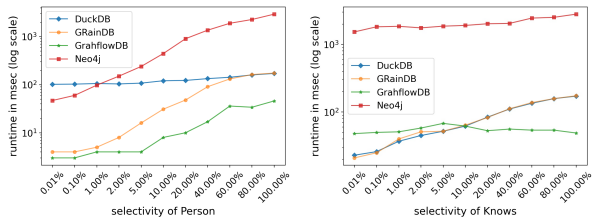


Figure 4: Ablation tests for different optimization levels in GRainDB.

8.3.1 Ablation Study. We performed an ablation study, to show the positive performance benefits of each of the optimizations we integrated into DuckDB: (i) RID materialization (Section 4); (ii) reverse information passing (Section 6.1); and (iii) extended RID index and join merging (Section 6.2). We turned our optimizations off in a specific order and in growing sets. We first turned off extended RID index and join merging (-JM), then, we turned off reverse information passing (-JM-RIP), and finally we turned off all optimizations, which gives us vanilla DuckDB. Then we ran each version of the system on the SNB-M benchmark. Figure 4 shows the box plot charts of each version of the system. GR-FULL in the figure is the configuration with all optimizations on. We see that each optimization has a positive effect on performance, which can be seen by inspecting the median and 25 percentile lines, which consistently shift down as we add more optimizations. We see most impact from the reverse information passing optimization, which is expected as it allows passing information from smaller entity tables (P in our notation) to much larger relationship tables (F in our notation). In the longer version of this paper [20], we show the runtime numbers of each query on each system configuration.

8.3.2 Performance of Predefined Joins Under Varying Entity vs Relationship Table Selectivity. We next do a controlled experiment to demonstrate the behavior of our sip- and hash-join based implementation under various selectivities on the P and F tables. Our goals are twofold: (i) to show the cases when sip yields performance



(a) MICRO-P. (b) MICRO-K.

Figure 5: Runtimes (in ms) of DuckDB, GRainDB, GraphflowDB and Neo4j on MICRO-P and MICRO-K benchmarks.

improvements; and (ii) to demonstrate the different performance behaviors of INLJs, which GDBMSs use, vs hash joins, which many RDBMSs use for equality joins. We take the LDBC30 dataset and the 1-hop $(p_1{:}Person) \xrightarrow{e:Knows} (p_2{:}Person)$ query, where the Person and Knows tables have 18.4K and 7.5M tuples, respectively. We then run two sets of micro-benchmark queries: (1) MICRO-P: we fix a predicate with 99.9% selectivity on the creationDate property of Knows and vary the selectivity of a predicate on the id property of Person between 0.01% to 100%. (2) MICRO-K: we now fix a predicate with 99.9% selectivity on the id property of Person and vary the creationDate property of Knows between 0.01% to 100%. We run each set of queries on DuckDB, GRainDB, GraphflowDB and Neo4j. Our goal in including Neo4j in these experiments, which was omitted in our baselines, is to show that the two GDBMSs behave very similarly albeit in different performance levels.

In both sets of queries GraphflowDB and Neo4j's executions are as follows: (i) scan the Person nodes and their id property and run the filter on id; (ii) join these nodes with their Knows edges by INLJ using the Knows adjacency list index; (iii) read the creationDate property of the joined edges and run a filter. We will momentarily show that this execution is too rigid and can be suboptimal. This is also the execution in systems such as GR-Fusion and GQ-Fast.

Figure 5a shows the results for MICRO-P. First, we note that on all MICRO-P queries, DuckDB makes Person the build side as it is already much smaller than Knows and gets even smaller as we decrease the selectivity on the predicate on Person. Therefore, in GRainDB, as we decrease the selectivity, we can pass selective information to Knows table and decrease the amount of scanned Knows tuples. We see that GRainDB outperforms DuckDB significantly and closes the gap with GraphflowDB's performance at these lower selectivities. Second, observe that both GDBMSs have consistent upward curves, indicating that their runtimes decrease as selectivity on the Person nodes decreases. This happens because the amount of join work that GDBMSs perform decreases proportionately as fewer Person nodes pass the filter. We cannot observe this desirable behavior with DuckDB because although its cost of hash table build decreases, its probe cost, which is the dominant cost here, does not. In fact, although broadly Neo4j is not competitive with other systems, it can still outperform DuckDB at lower selectivities on MICRO-P, because of its performance gains from decreasing selectivity on Person. Unlike DuckDB, GRainDB however also behaves similarly to GDBMSs and obtains this desirable behavior because it can also decrease the amount of probes through sip.

We next analyze the results of MICRO-K, shown in Figure 5b. First observe that now the GDBMSs do not react as positively to the decreasing selectivity on Knows. This is because now selectivity on Person is fixed, so the amount of probes GDBMSs perform is fixed. So both Neo4j and GraphflowDB curves are relatively straight (similar to the DuckDB curve in Figure 5a). Now note that DuckDB has a downward curve. This is because at all selectivity levels except 0.01% and 0.01%, DuckDB chooses Person as the build side. Therefore, decreasing the selectivity proportionately decreases the probe work. DuckDB can even outperform GraphflowDB when the selectivity is low enough. Note also that as expected GRainDB does not improve the performance of DuckDB now because although it passes information from Person to Knows, this information is not useful since Person does not have a selective predicate (it is fixed at 99.9%). We see minor benefits at the lowest two selectivity levels, when DuckDB starts to choose Knows as the build side, and can pass selective information to scans of Person. Although we do not observe major improvements, this shows the flexibility of join processing in RDBMS, where there is no notion of node vs edge tables and for hash joins, systems can make any table the probe or build side. In contrast, GraphflowDB and Neo4j first scan node records and then probe the adjacency list indices with the IDs of these nodes to perform the join (and not vice versa). As the MICRO-K benchmark demonstrates, this can prevent them from benefiting from selective predicates on the edge records.

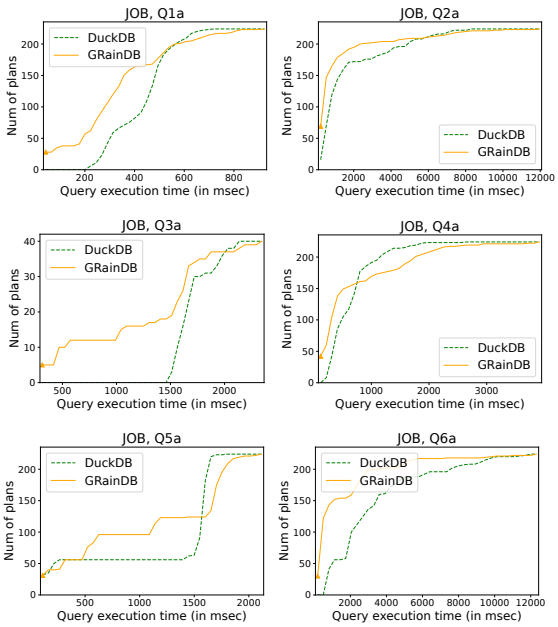


Figure 6: Cumulative distributions of the number of DuckDB and GRainDB plans (y-axis) that have runtimes below different thresholds (x-axis).

8.3.3 Plan Spectrum Analyses. Prior work [44] has observed that sip-based query processing makes systems broadly more robust to join order selection by decreasing the performance differences between different join orders. To demonstrate that our proposed solution also has similar effects, we picked the first six query groups in JOB, and for the first two variants of each query (so a total of 12 queries) performed a plan spectrum analysis as follows. We take each plan T for each Q, corresponding to one join order, and execute

Table 7: Runtimes (in ms) of T^*_{Duck} and T^*_{opt} on JOB queries.

	Q1a	Q1b	Q2a	Q2b	Q3a	Q3b	Q4a	Q4b	Q5a	Q5b	Q6a	Q6b
T^*_{Duck}	34	3	154	143	328	502	114	73	116	146	57	97
T^*_{opt}	31	3	77	67	287	135	72	47	110	112	32	82

both the default version of T (call it T_d) and the version where we use sip-based predefined joins (call it T^*_d). We then plotted two cumulative distribution lines for Q, one for the set of T_d and one for T^*_d plans, which show the number of plans (on y-axis) for different runtime value cutoffs (on x-axis). Figure 6 shows the distributions we obtained for the first variants of these queries. The remaining 6 charts, which are similar to the ones in Figure 6, can be found in the longer version of our paper [20]. Dashed and straight lines are the distributions of T_d and T^*_d plans, respectively. We observe on left sides of the curves, which summarize the best performing plans, the line showing T^*_d is consistently above the line for T_d. This shows that by predefining joins, we obtain larger sets of good plans. In many queries we also observe runtimes that were not achievable by any default plan. For example, on Q1a, while there are 60 plans with a runtime of $\leq$ 200ms under predefined joins, there is no such plan with default plans. We also observe that on the right ends of many curves, which plot the set of worst-performing plans, the curve for T^*_d plans is now below the curve for T_d plans. This is also expected because we expect there to be some plans that do not benefit from predefined joins and instead incur minor overheads that `S-Join` operators incur, e.g., to prepare bitmasks.

Let us call the T^*_d of a plan T_d the *predefined version* of T_d. Recall that for a query Q, GRainDB's plan, T^*_{Duck} is the predefined version of the plan T_{Duck} that DuckDB's optimizer picks for Q. Next, we analyzed the potential room for improvement on our rule-based approach if a system implements a sip-aware optimizer. We do a thought process and assume that an oracle sip-aware optimizer could pick the best GRainDB plan T^*_{opt}, i.e., the best performing T^*_d, and compare it against T^*_{Duck}. Table 7 shows this comparison. Although we did not find large rooms of improvement on most of these queries, we still found several queries, Q2a, Q2b, and Q3b with >2x improvements. The largest improvement is on Q3b, from 502ms to 135ms (3.7x). Q3b is a join query with 4 tables, and contains a selective predicate on a table called `keyword` that returns only 30 of the 134K tuples in this table. T^*_{Duck} uses a bushy plan. Instead, T^*_{opt} is a left-deep plan where the last-join has `keyword` on its build side. Normally putting this table as the last join on a left-deep plan is not efficient because joining the smaller tables first and creating smaller intermediate results is more efficient. However, under sip, this is a good plan because this can lead to iterative information passing to reduce the amount of scans in other tables.

8.3.4 Storage Costs. We next measure the storage costs of RID columns and indices and compare it against the costs of adjacency lists in GraphflowDB and materialized views in DuckDB-MV. Same as our ablation study, we turned optimizations off in a specific order and in growing sets. GR-FULL contains all optimizations, thus its storage overheads include the RID materialization, the RID index, and the extended RID index over vanilla DuckDB. GR-JM turns off the extended RID index. GR-JM-RSJ disables the RID index on top of GR-JM. Table 8 shows our results and additional memory consumption incurred by each optimization.

Table 8: Storage costs (in GB) of different optimizations.

	DuckDB	GR-JM-RSJ	GR-JM	GR-FULL
JOB	4.7	6.3(+1.6)	9.7(+3.4)	11.7(+2.0)
SNB-N	6.9	9.8(+2.9)	13.9(+4.1)	15.7(+1.8)
TPC-H	13.4	15(+1.6)	19.2(+4.2)	21.6(+2.4)

As our RID indices are similar to adjacency list indices in GDBMSs, we also profiled the memory consumption of GraphflowDB. For each RID index we have on SNB-M, there is a corresponding adjacency list in GraphflowDB. In total, GRainDB's RID indices take 5.9GB while GraphflowDB's indices take 2.8GB. This is expected because GraphflowDB implements several compression techniques [15], such as compressing trailing 0s in IDs, which in SNB-M reduces 8 byte IDs to 4 bytes. Instead, we store each RID in 8 bytes.

Finally, we measured the space overheads of DuckDB-MV on SNB-M. For DuckDB-MV we measured the additional space the system takes for storing one materialized view for each predefined join. Note that the performance of GRainDB we reported in Section 8.2.3 used the GR-FULL configuration. Using our materialized views takes 59.5GB space compared to 8.8GB (2.9+4.1+1.8) overheads of GR-FULL. This is expected because the materialized views in DuckDB-MV include all columns of the projections while RID indices only contain the integer RID columns. This comparison should not be interpreted as an accurate comparison of the space requirements of these two approaches because in practice a more careful design of views based on a workload can include only a subset of the columns or even remove some of the views (though some RID indices can also be removed). The selection of views and indices is a well studied problem [6, 27] but a detailed study of selecting views and RID indices is beyond the scope of this paper.

9 CONCLUSIONS

We described a novel approach to integrate predefined and pointer-based joins, which are prevalent in GDBMSs, into columnar RDBMSs. Our approach is based on materializing and optionally indexing RIDs similar to how edges are indexed in adjacency lists. In contrast to native GDBMSs and prior implementations of predefined joins in RDBMSs [17, 18, 25, 26] that use such indices in INLJs, we use them primarily to generate filters that are passed from hash join operators to scans. We also described an optimization that can use an extended RID index to avoid scans of a relationship table entirely in some settings. We presented extensive experiments demonstrating the performance benefits and overheads of this approach against vanilla value-based joins of DuckDB, INLJ-based implementation of predefined joins in GDBMSs, and the use of materialized views that can replace joins with scans.

ACKNOWLEDGMENTS

Guodong Jin's work was supported by National Key Research and Development Program of China(2018YFB1004401), NSFC grant No. U1711261, and China Scholarship Council (CSC) Grant #201906360139. This research was also partially funded by a grant from Waterloo-Huawei Joint Innovation Laboratory. We thank Xiyang Feng, Pranjal Gupta and Amine Mhedhbi for helping with experiments. We also thank the anonymous reviewers for their valuable comments.

REFERENCES

[1] 2021. DGraph. https://dgraph.io
[2] 2021. GRainDB. https://github.com/graindb/graindb
[3] 2021. Neo4j. http://neo4j.com
[4] 2021. TigerGraph. http://tigergraph.com
[5] 2021. Value reuse for auto-increment columns in MySQL. https://dev.mysql.com/doc/refman/8.0/en/delete.html
[6] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. 2000. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*.
[7] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, et al. 2020. The LDBC social network benchmark. *CoRR* (2020).
[8] Nafisa Anzum, Semih Salihoglu, and Daniel Vogel. 2019. GraphWrangler: An Interactive Graph View on Relational Data. In *ICDE*.
[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D Ullman. 1985. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). In *PODS*.
[10] Angela Bonifati, George Fletcher, Hannes Voigt, and Nikolay Yakovets. 2018. *Querying Graphs.* Morgan & Claypool.
[11] Edgar F Codd. 1982. Relational database: a practical foundation for productivity. *Commun. ACM* 25, 2 (1982).
[12] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. 2000. *Database system implementation.* Prentice Hall Upper Saddle River, NJ:.
[13] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing Queries Using Materialized Views: A Practical, Scalable Solution. *SIGMOD Record* 30, 2 (2001).
[14] Goetz Graefe. 1993. Query evaluation techniques for large databases. *ACM Computing Surveys (CSUR)* 25, 2 (1993).
[15] Pranjal Gupta, Amine Mhedhbi, , and Semih Salihoglu. 2021. Integrating Column-Oriented Storage and Query Processing Techniques Into Graph Database Management Systems. *CoRR* (2021).
[16] Alon Y. Halevy. 2001. Answering Queries Using Views: A Survey. *The VLDB Journal* 10, 4 (2001).
[17] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. Extending In-Memory Relational Database Engines with Native Graph Support. In *EDBT*.
[18] Mohamed S Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G Aref, and Mohammad Sadoghi. 2018. Grfusion: Graphs as first-class citizens in main-memory relational database systems. In *SIGMOD*.
[19] Zachary G Ives and Nicholas E Taylor. 2008. Sideways information passing for push-style query processing. In *ICDE*.
[20] Guodong Jin and Semih Salihoglu. 2021. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. https://cs.uwaterloo.ca/~ssalihog/papers/predefined-tr.pdf
[21] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. 2017. Graphflow: An active graph database. In *SIGMOD*.
[22] Hui Lei and Kenneth A. Ross. 1999. Faster joins, self-joins and multi-way joins using join indices. *Data & Knowledge Engineering* 29, 2 (1999).
[23] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *PVLDB* 9, 3 (2015).
[24] Zhe Li and Kenneth A. Ross. 1999. Fast Joins Using Join Indices. *VLDBJ* 8, 1 (1999).
[25] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. In *ICDE*.
[26] Chunbin Lin, Jianguo Wang, and Yannis Papakonstantinou. 2017. GQFast: Fast graph exploration with context-aware autocompletion. In *ICDE*.
[27] Imene Mami and Zohra Bellahsene. 2012. A Survey of View Selection Methods. *SIGMOD Record* 41, 1 (2012).
[28] 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*.
[29] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. 2021. Optimizing One-time and Continuous Subgraph Queries using Worst-Case Optimal Joins. *TODS* (2021).
[30] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB* 12, 11 (2019).
[31] Inderpal Singh Mumick and Hamid Pirahesh. 1994. Implementation of Magic-Sets in a Relational Database System. In *SIGMOD*.
[32] Thomas Neumann and Gerhard Weikum. 2009. Scalable join processing on very large RDF graphs. In *SIGMOD*.
[33] Patrick O'Neil and Goetz Graefe. 1995. Multi-Table Joins through Bitmapped Join Indices. *SIGMOD Rec.* 24, 3 (1995).
[34] Patrick O'Neil and Dallan Quass. 1997. Improved Query Performance with Variant Indexes. *SIGMOD Record* 26, 2 (1997).
[35] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an embeddable analytical database. In *SIGMOD*.
[36] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *CIDR*.
[37] Michael Rudolf, Marcus Paradies, Christof Bornhövd, and Wolfgang Lehner. 2013. The graph story of the SAP HANA database. *BTW*.
[38] 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).
[39] Wen Sun, Achille Fokoue, Kavitha Srinivas, Anastasios Kementsietsidis, Gang Hu, and Guotong Xie. 2015. Sqlgraph: An efficient relational-based property graph store. In *SIGMOD*.
[40] Yuanyuan Tian, En Liang Xu, Wei Zhao, Mir Hamid Pirahesh, Sui Jun Tong, Wen Sun, Thomas Kolanko, Md Shahidul Haque Apu, and Huijuan Peng. 2020. IBM Db2 Graph: Supporting Synergistic and Retrofittable Graph Queries Inside IBM Db2. In *SIGMOD*.
[41] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and Divesh Srivastava. 2003. Ranked Join Indices. In *ICDE*.
[42] Patrick Valduriez. 1987. Join indices. *TODS* 12, 2 (1987).
[43] Konstantinos Xirogiannopoulos, Virinchi Srinivas, and Amol Deshpande. 2017. GraphGen: Adaptive graph processing using relational databases. In *GRADES*.
[44] Jianqiao Zhu, Navneet Potti, Saket Saurabh, and Jignesh M Patel. 2017. Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads. *PVLDB* 10, 8 (2017).