Foundations and Trends® in Databases

Modern Techniques For Querying Graph-structured Databases

Suggested Citation: Amine Mhedhbi, Amol Deshpande and Semih Salihoğlu (2024), "Modern Techniques For Querying Graph-structured Databases", Foundations and Trends® in Databases: Vol. 14, No. 2, pp 72–185. DOI: 10.1561/1900000090.

Amine Mhedhbi
Polytechnique Montreal
amine.mhedhbi@polymtl.ca

Amol Deshpande
University of Maryland
amol@umd.edu

Semih Salihoğlu
University of Waterloo
semih.salihoglu@uwaterloo.ca

now
the essence of knowledge
Boston — Delft

Contents

Modern Techniques For Querying Graph-structured Databases

Amine Mhedhbi[1], Amol Deshpande[2] and Semih Salihoğlu[3]

[1] *Polytechnique Montreal, Canada; amine.mhedhbi@polymtl.ca*
[2] *University of Maryland, USA; amol@cs.umd.edu*
[3] *University of Waterloo, Canada; semih.salihoglu@uwaterloo.ca*

ABSTRACT

In an era of increasingly interconnected information, graph-structured data has become pervasive across numerous domains from social media platforms and telecommunication networks to biological systems and knowledge graphs. However, traditional database management systems often struggle when confronted with the unique challenges posed by graph-structured data, in large part due to the explosion of intermediate results, the complexity of join-heavy queries, and the use of regular path queries.

This survey provides a comprehensive overview of *modern* query processing techniques designed to address these challenges. We focus on four key components that have emerged as pivotal in optimizing queries on graph-structured databases: (1) Predefined joins, which leverage precomputed data structures to accelerate joins; (2) Worst-case optimal join algorithms, that avoid redundant computations for queries with cycles; (3) Factorized representations, which compress intermediate and final query results; and (4) Advanced techniques for processing recursive queries, essential for traversing graph structures. For each component,

Amine Mhedhbi, Amol Deshpande and Semih Salihoğlu (2024), "Modern Techniques For Querying Graph-structured Databases", Foundations and Trends® in Databases: Vol. 14, No. 2, pp 72–185. DOI: 10.1561/1900000090.

we delve into its theoretical underpinnings, explore design considerations, and discuss the implementation challenges associated with integrating these techniques into existing database management systems. This survey aims to serve as a comprehensive resource for both researchers pushing the boundaries of query processing and practitioners seeking to implement state-of-the-art techniques, in addition to offering insights into future research directions in this rapidly evolving field.

1

Introduction

Graph-structured data is ubiquitous in many real-world domains and applications. Social networks, financial transactions, telecommunication networks, and knowledge graphs are just a few examples where data is naturally represented as a graph with entities as *nodes* and relationships as *edges*. Querying and analyzing these graph-structured datasets is integral to a wide range of analytical applications such as recommendations in social networks, fraud detection in financial transaction networks, threat detection in call networks, and inference over knowledge bases (Sahu *et al.*, 2020).

As an example application domain, consider financial transaction networks. These networks model entities like individuals, businesses, and financial institutions as nodes, with transactions between them as edges. Analyzing these networks is crucial for tasks like detecting money laundering, tax evasion, and other financial crimes. Typical queries involve finding long chains of transactions (acyclic paths) that may indicate suspicious activity, or identifying tightly connected clusters (cliques/near-cliques) of entities engaging in coordinated illicit behavior.

Figure 1.1 shows an example graph containing 6 entities (nodes) and 7 relationships (edges) between them. We assume that the input

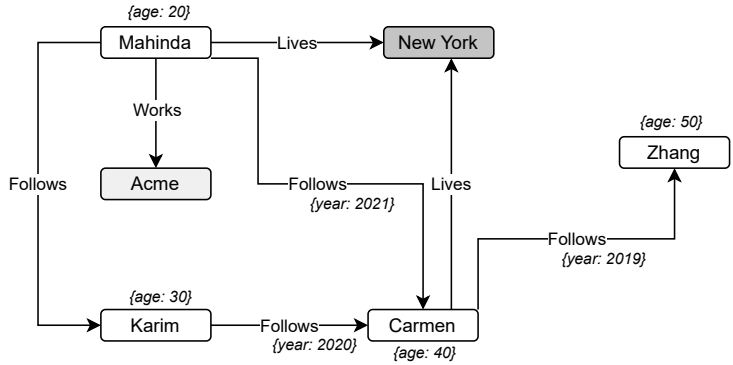


Figure 1.1: An example of a *property graph* capturing social network data. *Name* properties of nodes are written directly inside the rectangles representing nodes.

Person

name	age
Mahinda	20
Karim	30
Carmen	40
Zhang	50

Follows

pFrom	pTo	year
Mahinda	Karim	2021
Carmen	Zhang	2019
Mahinda	Carmen	2021
Karim	Carmen	2020
Mahinda	Zhang	2021

Lives

pName	lName
Mahinda	New York
Carmen	New York

Works

pName	cName
Mahinda	Acme

Locations

name
New York

Companies

name
Acme

Figure 1.2: Representation of the example graph from Figure 1.1 as a collection of relations.

graphs are directed in this monograph, however, most of the discussion applies to undirected graphs as well. Further, we logically model these graphs as a collection of relations as shown in Figure 1.2, where there is a separate relation for every entity type (label) and every relationship type (label). We note that this is a logical representation that we adopt for presentation purposes, and does not necessarily dictate the physical storage layouts, the data structures and indexes that may be built on top, or the query processing algorithms. In other words, we use the relational representation to express the queries to be executed against this graph

so that we can more easily contrast and compare with relational query processing techniques; however, following the principle of *physical data independence*, the physical storage representation is not required to match that logical representation.

Given the tabular representation of graphs, a large fraction of graph analysis and querying tasks (collectively referred to as *graph workloads*) can be seen as select-project-join-aggregate queries on these tables, potentially with recursion. However, unlike typically relational workloads, graph-structured datasets and workloads have two primary defining features:

(i) **Prevalence of many-to-many relations across entities:** Unlike typical relational workloads which primarily feature key-foreign key connections across tables, graph workloads primarily contain many-to-many relationships (e.g., all the relationships in the example graph above, i.e., *Follows*, *Works*, and *Lives*, are many-to-many relationships).

(ii) **Prevalence of complex join-heavy queries over these relations:** The prevalent tasks in graph workloads translate to queries with many joins across these many-to-many relationships. The joins in these queries can have several different structures:

  (i) cyclic, such as when finding cliques of phone calls;

  (ii) acyclic, such as when finding long chains of financial transactions; or

  (iii) recursive, such as when finding shortest connections between users in social networks.

This contrasts with traditional relational workloads, such as those found in the popular TPC benchmarks, that contain many primary-foreign key (`PK-FK`) joins. The combination of complex join structures in the graph workloads and the many-to-many cardinality of relations in these datasets pose serious challenges for traditional query processors. For example, queries can generate large intermediate relations that often cannot be handled by traditional techniques.

The last decade has seen the emergence of numerous prototype and commercial DBMSs that are optimized for graph workloads. These include specialized systems such as graph DBMSs (GDBMSs) that adopt the property graph data model, e.g., Neo4j (Neo4j, Inc, 2023a), TigerGraph (Tigergraph, 2023), GraphflowDB (Kankanamge *et al.*, 2017), Kùzu (Feng *et al.*, 2023), Avantgraph (Leeuwen *et al.*, 2022); earlier RDF systems, e.g., RDF-3x (Neumann and Weikum, 2010); and graph-optimized extensions of RDBMSs, e.g., GR-Fusion (Hassan *et al.*, 2018), GRainDB (Jin and Salihoglu, 2022), and GQ-Fast (Lin *et al.*, 2016). The goal of this monograph is to survey a set of *modern* query processing techniques that have been integrated into the query processors of these systems. These include:

(i) *Pointer-based joins* (Section 2) that rely on system-level dense IDs in contrast to traditional value-based joins in RDBMSs.

(ii) *Worst-case optimal join (WCOJ) algorithms* (Section 3), which are a new class of join algorithms that address the problem of large intermediate size generation for cyclic many-to-many join queries. Compared to traditional plans that use a tree of binary join algorithms and perform the joins in a query pairs of table at a time, WCOJs algorithms perform the joins one column at a time.

(iii) *Factorization* (Section 4), a class of techniques to compress intermediate relations that exhibit *multi-valued dependencies* generated when performing many-to-many joins, specifically in the acyclic parts of queries. The theory of factorization represents such intermediate relations in different *factorized representations* as unions of Cartesian products instead of flat tuples. The theory of factorization explains when query processors can exploit such factorized representations by analyzing the join conditions between variables during query compilation time.

(iv) *Techniques for regular path queries* (Section 5), a popular class of recursive graph queries. Amongst recursive queries, most prior work focuses on regular path queries. We cover the traditional

automata-based plans of Mendelzon and Wood (1989) and α-join plans based on α relational algebra introduced by Agrawal (1988), and the more recent WaveGuide plans of Yakovets *et al.* (2016a) that mix both of these approaches.

We overview: (i) the foundations of these techniques when appropriate; (ii) the current design choices different DBMSs have made to integrate these techniques; and (iii) the challenges for existing implementation approaches, which provide promising avenues for further research. Our goal is to bring a structure to this vast theory and systems-oriented literature. We focus on the application of these techniques for join processing over static databases though we briefly mention works that apply these techniques to the problem of incrementally maintaining query results when the underlying databases are dynamic.

We primarily cover these techniques in the context of a centralized, sequential computation model, which has been the focus of most of the work on these techniques so far, including WCOJ algorithms, factorization, and RPQs. There is a vast body of work on building on parallel and distributed graph analytics platforms (Yan *et al.*, 2017), primarily based on the so-called *vertex-centric programming framework* (Malewicz *et al.*, 2010). This framework has also been incorporated into several relational database systems (Fan *et al.*, 2015; Jindal *et al.*, 2014). However, the target workload for those systems is very different from the types of queries that we focus on in this monograph. Some of the example graph analysis tasks include: finding most central or influential nodes in the graph (e.g., by calculating metrics such *page rank* or *betweenness centrality*), identifying communities in the graph, understanding influence propagation, etc. Although there is some overlap, the algorithmic techniques discussed in this monograph are not applicable to executing those types of tasks; instead, specialized parallel systems (Shun and Blelloch, 2013; Wang *et al.*, 2016) are typically used given the scale of the graphs involved in the application domains like social media, finance, disease transmission, etc. On the flip side, it has been shown how to implement multi-way joins efficiently on top of the vertex-centric framework (Smagulova and Deutsch, 2021) and other distributed programming paradigms (Afrati and Ullman, 2011); we discuss some of that work briefly where appropriate.

The techniques we discuss in this monograph are generally applicable to any database system where the workload maps to multi-way join queries over relations. This includes RDF databases that store the RDF data as a collection of tables and map queries over the data (typically in SPARQL) to multi-way join queries over those tables (Neumann and Weikum, 2010; Abadi *et al.*, 2007; Erling and Mikhailov, 2009). It also includes document databases that adopt XML or JSON data model, but "shred" the data into a collection of tables and translate the queries to join queries (Tatarinov *et al.*, 2002). However, they do not apply directly to systems that use specialized storage schemes or index data structures to execute queries (e.g., gStore, Zou *et al.*, 2014a). We note that there are a number of similarities between WCOJ algorithms and traversal-based techniques for subgraph pattern matching (Sun *et al.*, 2020), but more work is needed to unify these somewhat disparate lines of work.

Finally, we focus exclusively on read queries in this monograph. In most cases, updates to the graph can be directly mapped to updates to the underlying tables, and can benefit from the mature and efficient support for transactions and ACID properties in relational databases. This is, in fact, a key motivation behind using a relational database as the backend storage for a graph database. However, the specific mix of queries and updates may influence the decisions about how many and which tables to use. We don't discuss these issues further in this monograph.

1.1 Target Audience

This monograph is intended for readers who are familiar with basic concepts of internals of databases, such as the general paradigm of compiling high-level queries into executable query plans and core query processing operators, such as scans and joins. Beyond that we cover the necessary background in each section. We are particularly interested in making the monograph accessible to readers with graph analytics or graph processing systems backgrounds. However, we adopt a relational view of query evaluation even if logically the datasets in our examples are often modeled as graphs. This is because the foundations of many of the techniques we cover were developed in the context of join processing

over sets of records. At parts, we cover some advanced material on database theory. We accompany these parts with suggestions to skip for readers who may be more interested in understanding the core query processing techniques and how they are integrated into systems.

1.2 Brief Background

We end this introductory section with a brief overview of some background on the formal notation we use for describing join queries and databases. The necessary notation for regular path queries is covered in Section 5. Background for each of the techniques we cover is provided in detail in each section.

Unless otherwise stated, we assume that input graphs are directed and modeled as binary relations, that are denoted with capital letters R and E or variants such as R_1 or E_2. The attributes of relations are generally denoted with lowercase letters that start with a, such as a_1 or a_2. In some figures, we use attributes 'from' and 'to', 'src' or 'dst', or a similar variant of these words to denote the sources and destinations of the edges.

We consider natural join queries over these binary relations that we denote by Q (or a variant Q_i). We generally assume full join queries, i.e., where no projections occur. We denote queries in Datalog syntax, where we generally omit the attributes in the head of the rules. The following is an example showing how we denote the "triangle" query:

$$Q_\Delta \coloneqq R_1(a_1, a_2), R_2(a_2, a_3) R_3(a_3, a_1)$$

Sometimes, we write a predicate next to a variable in the head or body of these rules to indicate filters. For example, $Q(a_1{=}1, a_2, a_3) \coloneqq R_1(a_1 = 1, a_2), R_2(a_2, a_3), R_3(a_3, a_1 = 1)$ represents the query that finds all triangles where a_1 has value 1.

In a few parts of the monograph, we also use example queries from SQL and the Cypher query language (Francis *et al.*, 2018). Cypher is the query language of the Neo4j system that is also adopted by several other GDBMSs, such as MemGraph (Memgraph Ltd, 2023) and Kùzu (Feng *et al.*, 2023). The meanings of Cypher queries is explained in the parts of text when they are used.

2

Predefined Joins

In this section, we describe a common difference in the implementations of the core relational join operators that are used in RDBMSs and GDBMSs. Specifically, joins in RDBMSs are value-based while those in GDBMSs are based on using internal integer record IDs (RIDs) of the nodes and a join index. We refer to the latter style of joins as *predefined* or *pointer-based* joins. These terms will be described in detail momentarily. For now, we note only that the distinction between predefined and value-based join processing is orthogonal to the actual join algorithms preferred in these systems. In other words, there can be predefined and value-based implementations of any join algorithm, such as binary hash join or index nested loop joins as well as worst-case optimal join algorithms. In this section, we focus on the implementations of standard binary join operators; in Section 3, we discuss examples of worst-case optimal join operators that perform predefined joins as well as those that perform value-based joins. We first begin by clarifying the meaning of joins in the context of query languages of GDBMSs.

2.1 Overview of Joins in SQL and Graph Query Languages

In the context of RDBMSs, joins are expressed in the FROM and WHERE clauses of SQL. Consider the Person and Follows relations from our running example from Section 1, shown partially also in Figure 2.1. Consider the SQL query below that finds the ages of people that Carmen follows:

```
SELECT b.age
FROM Person a, Follows e, Person b
WHERE a.name = 'Carmen'
      AND a.name = e.pFrom AND e.pTo = b.name
```

In the query languages of many GDBMSs, joins are expressed explicitly in the graph patterns that are drawn in the queries. Below is the Cypher query that is equivalent to the above SQL query.

```
MATCH (a:Person)-[e:Follows]→(b:Person)
WHERE a.name = 'Carmen'
RETURN b.age
```

The '( )' notation is used to specify the node records and '-[ ]→' notation is used to specify the relationship records. Together, the pattern in the MATCH clause above is equivalent to joining two Person "node" tables, 'a' and 'b', with the Follows "relationship" table 'e'.[1] The rest of the query is very similar to SQL: the WHERE clause expresses additional predicates and RETURN, similar to SQL's SELECT, returns the projected expressions.

2.2 Value-based Joins

Although the above two queries are equivalent, their typical execution in existing RDBMSs and GDBMSs are quite different. Specifically, joins in RDBMSs are value-based, i.e., the system will run the join predicates

[1] In many publications on graph literature, this computation is referred to as "traversing" the neighborhoods of node records. However, from the perspective of the DBMS that is evaluating the query, this computation is exactly joining the Person and Follows relations as expressed in the SQL query.

Person

name	age
Mahinda	20
Karim	30
Carmen	40
Zhang	50

(a) Person records.

Follows

pFrom	pTo	year
Mahinda	Karim	2021
Carmen	Zhang	2019
Mahinda	Carmen	2021
Karim	Carmen	2020
Mahinda	Zhang	2021

(b) Follows records.

Figure 2.1: Person node records and Follows edge records.

expressed in the WHERE clause using the actual data types of the columns in the predicate. For example, in the above query, the condition 'a.name = e.pFrom' is on string columns. To demonstrate this, let us review the execution of the HashJoin operator, which is one of the standard join operators for equality joins.[2] Figure 2.2 shows an example HashJoin plan for the query. HashJoin has two children operators, each one giving the operator one of the two relations to join. Each HashJoin operator's execution consists of two phases:

- Build phase: The "build relation" (the right child in Figure 2.2) is scanned completely and a hash table is built on the scanned tuples. The key of the hash table is the attribute value in the join predicate. For example, the HashJoin-1 operator in the figure builds a hash table over the Person tuples that satisfy the condition; in this case, it would contain just one entry with the name property Carmen as the key (no payload is needed for this hash table).

- Probe phase: For each tuple t of the "probe relation" (the left child in Figure 2.2), the hash table is probed to find matching tuples from the build relation.

The plan in Figure 2.2 contains a second HashJoin operator to join the result of ($\sigma_{name='Carmen'}$ (Person) $\bowtie_{\text{name=pFrom}}$ Follows) with the second Person table. Importantly, the evaluation of the join condition

[2]Equality joins refers to the fact that the join predicates between Person and Follows tables in our example query are equality conditions.

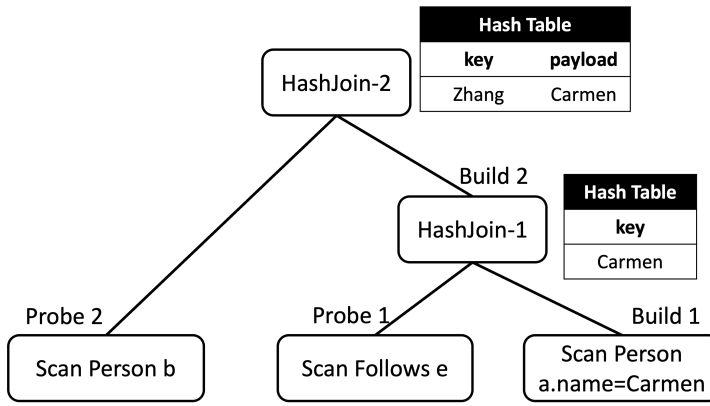


Figure 2.2: An example query plan, using value-based HashJoin operators, for finding the ages of Person nodes that follow the node with name Carmen. The plan is simplified and omits showing the projections done in the plan.

is value-based. For example, in the HashJoin-1 operator, when the '(pFrom=Carmen, pTo=Zhang, year=2019)' tuple from Follows is used to probe into the hash table, a string equality check is performed between the pFrom value "Carmen" and the key "Carmen" stored in the table. Similar value-based equality checks would be performed if the system had instead used other join operators, such as merge-join or index nested-loop join operators. In short, we call a join operator value-based if the operator performs comparison operations on the actual values stored in the records that are being joined. As we momentarily discuss, GDBMSs typically do not perform the equality checks on actual values of the records that are stored.

2.3 Predefined Joins and Join Indices

In contrast to value-based joins, joins expressed in the graph patterns of graph query languages are often evaluated using internal integer record IDs (RIDs) of the nodes. We refer to these type of joins as *predefined* or *pointer-based* joins. We begin by clarifying the term "predefined". When using GDBMSs, users need to explicitly define which of their records are nodes and which are edges. Since edges are used to join

node records with each other, users effectively *predefine*[3] to the system the common joins that they will perform.

This information is almost universally exploited by GDBMSs by building a join index (a.k.a. an adjacency list index). Join indices were introduced by Valduriez (1987) as an index to speed up arbitrary join queries, say between two relations R and S, by keeping the RIDs of successfully joining R and S tuples in an index. We omit reviewing their initial designs here except to note that join indices have not seen any adoption in RDBMSs. However, throughout history, many DBMSs that adopt a graph model have adopted forms of join indices to link node records with each other. We describe an overview of the design and usage of join indices in modern native GDBMS, such as Neo4j (Neo4j, Inc, 2023a), Kùzu (Feng *et al.*, 2023), GraphflowDB (Kankanamge *et al.*, 2017), Memgraph (Memgraph Ltd, 2023), as well as several prototype GDBMSs that have been developed over RDBMSs such as GR-Fusion (Hassan *et al.*, 2018), GQ-Fast (Lin *et al.*, 2016), or GRainDB (Jin and Salihoglu, 2022).

2.3.1 Example Join Index (aka Adjacency List Index)

When a user defines the Follows table as an edge/relationship table from Person node records to Person node records, GDBMSs will physically construct "links" between the joining Person records. These links are physical pointers between the node records that store the RIDs of the Person records in a data structure. In graph terms, this can be simply thought of as a form of an adjacency list index of a graph. Different GDBMSs use different data structures to store these RIDs but in the end all of those store the following index over the Follows relation.

[3]Historically, the term "predefined" was used by Ted Codd in his Turing Award Lecture to criticize the GDBMSs of the 1960s and 1970s (Codd, 1982). Codd was criticizing these systems for requiring users to predefine their joins to the system as links/edges and arguing that in the relational model of data, any two records from two different relations can be joined on arbitrary columns. Similar to RDBMSs, modern GDBMSs implement high level query languages, using which users can join arbitrary node or edge records with each other. Therefore, at this point, this predefinition is an advantage for GDBMSs as they can use this information to build automatic join indices over the edge records to perform joins of nodes with their neighbors more efficiently than value-based joins.

Consider extending the Person relation with the RIDs of each record. Figure 2.3 shows a virtual vRID column on Person that lists the RID of each Person record. For example, Mahinda has a RID of 0, Karim has 1, etc. The figure also shows a logical version of the Follows relation if the original pFrom and pTo values were replaced with the RIDs of the referenced Person columns. A join index is an index that takes in the RID of a Person node u and returns the RIDs of all joining Person records with u through the Follows relation. For example, a join index needs to map 0 to {1, 2, 3} because there are three Follows records with Mahinda (0) in the pFrom column: Karim (1), Carmen (2), and Zhang (3). Importantly, the join index is over the Follows relation but stores the RIDs of the Person node records.[4]

Person′

vRID	name	age
0	Mahinda	20
1	Karim	30
2	Carmen	40
3	Zhang	50

(a) Extended Person records.

Follows′

RIDFrom	RIDTo	year
0	**1**	2021
2	**3**	2019
0	**2**	2021
1	**2**	2020
0	**3**	2021

(b) Indexed Follows records.

Figure 2.3: Logical representation of extended tables with RIDs. Gray values are not materialized in any data structure.

GDBMSs use different data structures to store the join indices over relationship records. Bonifati *et al.* (2018) cover the commonly used structures in more detail. For reference, we show an example physical storage of an index in a data structure called the *compressed sparse row* (CSR) storage. Logically, a CSR consists of two arrays: (i) an offsets array; and (ii) a data array. The data array stores the neighbor IDs of each node. For each node u with RID j, offsets[j] stores the beginning offset of u's adjacency list in the data array. The end of u's list is offsets[j+1] (not inclusive). Figure 2.4 shows the join index over the Follows relation stored in CSR format. In practice, GDBMSs often

[4]We note that join indices often additionally keep track of the RIDs of the relationship records as well.

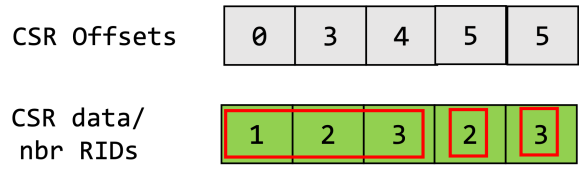


Figure 2.4: Logical view of (forward) join index implemented in the CSR format.

store disk-based versions of these indices, e.g., Kùzu's indices are stored on disk and in CSR format. Although different GDBMSs use different storage structures, the two common properties of the join indices in GDBMSs are the following:

- RIDs are system-level dense integers, i.e., they are consecutive integers that start from 0 and go until the number of nodes in the referenced nodes relation.
- GDBMSs index the relationship records in two join indices, once in the *forward* direction and once in the *backward* direction. Given the RID of a node record v, double indexing allows quickly accessing both the incoming and outgoing neighbors of v during query processing. In our example, with two join indices, one can find both the people who Mahinda follows quickly as well as the people who follow Mahinda.

The fact that the join indices store dense integers gives GDBMSs several advantages during join processing. First, RIDs in DBMSs generally serve as pointers to retrieve records. That is, given the RID of a node record, systems often can find the page of the record on disk. That is why predefined joins in GDBMSs are sometimes also referred to as *pointer-based* joins. Second, the use of RIDs can avoid value-based comparisons. Value-based comparisons can be slow operations if the join predicates are on variable-length data types, such as strings.

In the rest of this section, we cover two different operators that are used in systems to evaluate predefined joins using RIDs and discuss their pros and cons: (i) using Extend/Expand operators; and (ii) using HashJoins using RIDs. Both of these are binary join operators similar to value-based HashJoins of RDBMSs.

2.3.2 Predefined Joins using Extend/Expand Operators

One common join algorithm in GDBMSs appears under the names of `Expand` or `Extend`. We use the term `Extend` here. Example systems implementing `Extend` include Neo4j (Neo4j, Inc, 2023a), GraphflowDB (Kankanamge *et al.*, 2017), GRFusion (Hassan *et al.*, 2018) and GQFast (Lin *et al.*, 2016). Figure 2.5 shows an example plan using this operator for the Cypher query that finds the ages of people Carmen follows. The plan is drawn left-to-right. `Extend` has a single child and scans tuple from this child operator. Each scanned tuple contains the RID of a node ID bound to a variable. For example, in our example query, the tuple would contain (a=2), which is Carmen's RID. Then the operator extends (a=2) to the RIDs of the neighbors of node 2 using the join index. In our running example and query plan, `Extend` would produce (a=2, b=3), since 3 (Zhang) is the only neighbor of 2 (Carmen). Assuming that the join index is stored on disk, this style of join processing is akin to indexed nested loop join algorithms in database literature (Silberschatz *et al.*, 2005).[5] The example plan in Figure 2.5 also shows a `Lookup` operator, another common operator that can be found in GDBMSs, which looks up properties/columns of node records from the node relationships. In our example, given the RID of Zhang, which is 3, `Lookup` retrieves the age of Zhang.

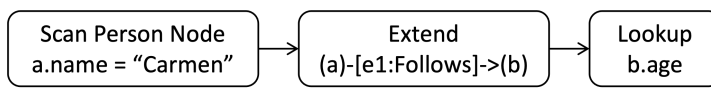


Figure 2.5: Example plan for the 3-hop query that uses Extends.

These operators are simple to implement in practice. Further, plans using these operators can be very efficient for simple queries, such as the query above, where the query can be answered simply by scanning a few node records and looking up their neighbors in the join index. At the same time, `Extend` can also lead to random I/Os, which can

[5] If the join index is completely in memory, as in in-memory GDBMSs such as Graphflow and MemGraph, then this is akin to hash join algorithms, where the index serves as a pre-computed hash table, where joining tuples can be looked up.

degrade performance in disk-based systems. Consider a 3-hop version of the query above, this time finding Karim's neighborhoods:

```
MATCH (a:P)-[e1:F]→(b:P)-[e2:F]→(c:P)-[e3:F]→(d:P)
WHERE a.name = 'Karim'
RETURN d.age
```

Person and Follows are abbreviated as P and F in the above query. Consider the plan shown in Figure 2.6 that uses only `Extend` operators followed by a `Lookup` operator. Suppose the query is executed on the graph database shown in Figure 2.7 and Karim's RID is 1. In this plan, the third `Extend` will take as input the tuples (a=1, b=12, c=1000), (a=1, b=12, c=50), (a=1, b=3, c=1000), (a=1, b=3, c=100), and (a=1, b=12, c=2). This will lead to look ups of adjacency lists of nodes 1000, 50, 1000, 100, and 2 in the join index. Note that the neighbors have no order and that neighbors can be repeated across adjacency lists. This can result in random and repeated I/Os if the join index is stored on disk, which can lead to severe performance degradation. A similar situation arises when properties of those neighbors need to be scanned, e.g., in the last `Lookup` operator of the plan in Figure 2.6.

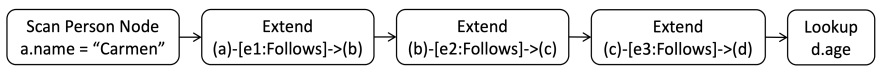


Figure 2.6: Example plan for the 3-hop query that uses Extends.

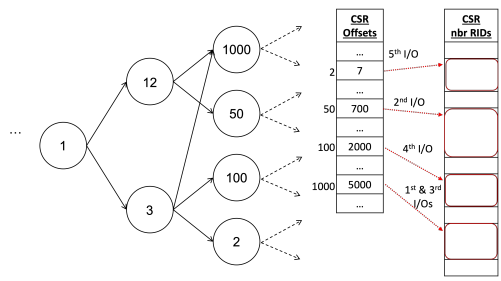


Figure 2.7: Example random I/Os in the name file when using Extends in the query plan in Figure 2.6.

2.3.3 Predefined Joins Using HashJoins

Notice that in value-based HashJoin plans of RDBMSs, the above problem does not arise at all. HashJoins by design sequentially scan and hash the tuples in one relation and then sequentially scan the other one and probe into the hash table. In a way, the random I/Os are replaced with random look ups in the hash tables, which are assumed to fit in memory. Instead, the I/Os are done when the build relation is scanned but is guaranteed to be sequential. However, `Extend`s have the advantage that they only scan the necessary parts of the files and avoid full scans of relations or join indices. For example, in the 1-hop query we used in the beginning of this section, a modern RDBMS, say Umbra (Neumann and Freitag, 2020) or DuckDB (Raasveldt and Mühleisen, 2019a), would scan the Person records and create a hash table with Carmen's tuple in it. Then it would read the entire Follows table and probe into the hash table. Recall that the GDBMS plan in Figure 2.5 that uses `Extend` scanned only Carmen's adjacency list, which is a very small fraction of the Follows relation.

Recently, Jin and Salihoglu (2022) have proposed and implemented an extension over DuckDB, called GRainDB. This work proposes to implement a version of predefined joins that use RIDs and a join index to obtain the best behavior of `Extend`s and HashJoins. The approach is based on extending HashJoins to pass the necessary RIDs that need to be scanned sideways to the probe side in a *semijoin filter*. This filter is then used by the scan operators on the probe side to avoid full scans of relations or join indices. This core apprach has also been implemented in the Kùzu GDBMS (Feng *et al.*, 2023), which is what we review here. At a high-level Kùzu's storage is similar to the setting we described here and consists of disk-based and CSR-based join indices.

Consider the 1-hop query again. Kùzu's plan for this query is shown in Figure 2.8. In the build phase, the Person node records are scanned and Carmen's record and RID 1 is found and hashed in a hash table. Once this phase is over, the operator has enough information to know that only vertex 1's Follows edges need to be scanned from the join index. This is put into a semijoin filter, which has 1 bit for each Person node. 1 indicates that the node's neighbors should be scanned and 0

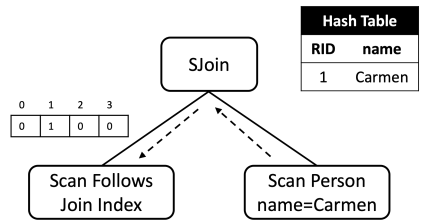


Figure 2.8: Example plan using modified HashJoin operator that uses sideways information passing. The right side is the build side and scans the tuple (RID=1, name=Carmen) from Person records and passes a semijoin filter with only v1 set to 1 and other RIDs set to 0. This enables the scan operator on the probe side to only scan v1's adjacency list.

indicates that it should not. This filter is passed to the probe-side scan operator that scans only the adjacency lists of vertices that have 1 in the semijoin filter. Therefore this approach avoids full scans of tables as well as guards these plans against random and repeated I/Os. Note that the use of a semijoin filter exploits the dense integer ID property of predefined joins as it assumes that the keys that can be joined are integers between 0 and |V|, where |V| is the number of Person nodes in the database.

Jin and Salihoglu (2022) and Feng *et al.* (2023) present performance comparisons of HashJoin vs `Extend`-based plans. These evaluations suggest that `Extend`-based plans are more performant when selectivities of the joins are very high, but degrade as they get low. In particular, if queries are more complex and `Extend` operators are used in sequence similarly to left deep plans, then the performance degradation is severe. HashJoin-based plans are more robust for such complex queries.

2.3.4 A Note on the Storage Designs of GDBMSs

Join indices are one storage structure used in GDBMSs. We end with a note about other common storage structures of GDBMSs. The design space here is not different than the design space in RDBMSs. Consider node and edge records that have structure, i.e., where each node record with a particular type, say Person, has the same set of properties. Then the two major design points to store these records are row-oriented

or column-oriented layouts. GDBMSs always index edge records in join indices. Some systems use structures that mimic the join indices to store edge properties. For example, Graphflow and Kùzu adopt columnar structures and store the properties of the edges in parallel CSR structures (the different CSRs for the join index and each rel property share the CSR offsets) (see Gupta *et al.*, 2021, and Feng *et al.*, 2023).

Another design point is the one adopted by Neo4j. Neo4j's join index (Neo4j, Inc, 2023b) is a linked list of relationship records. Within each relationship record, there are pointers to the first property record. Each property record stores a single property as (key, data type, value) triple and has pointers to other property records. A similar structure exists for node records and node property records as well. Linked list-based storage design is a very old design point in databases, e.g., it was used in very early DBMSs, such as the IDS system of the 1960s (Bachman, 2009). Unlike CSRs, which are optimized for read performance, linked list-based storage is optimized for small writes. It also makes it easy to store arbitrary properties that are outside the general structure of the records. This is a feature some systems support to provide a more flexible data model, albeit compromising on scan performance.

3

Worst-case Optimal Join Algorithms

In this section, we cover the theory and system integration of worst-case optimal join (WCOJ) algorithms, which propose a solution to the problem of large intermediate size generation for cyclic m-n join queries. Cyclicity/acyclicity of queries is an important structural property of queries that is known to have an important role in terms of how efficiently the query can be evaluated. For now we only note that, in the context of subgraph pattern queries, if the undirected version of the pattern has cycles, then a query is cyclic. Otherwise, it is acyclic.

To motivate the challenge of evaluating cyclic queries, consider the triangle query $R(a,b)$, $S(b,c)$, $T(a,c)$ on the input graph shown in Figure 3.1. R, S, and T each have m tuples, shown respectively in the figure as red, green, and blue edges (e.g., $(1_a, 1_b)$ is a tuple in R). The graph has 3m edges and contains $3m-2$ triangles. So, the number of triangles is linear in the number of edges. These triangles are in the form of: (1_a, 1_b, i_c), (1_a, i_b, 1_c), and (i_a, 1_b, 1_c) for $i = [1, ..., m]$ (-2 is for over counting (1_a, 1_b, 1_c) three times). However, any binary join plan P on the query will take $\Omega(m^2)$ time because the intermediate size of its first join will contain $\Omega(m^2)$ many open triangles. Consider as an example the plan $((R \bowtie S) \bowtie T)$, where the inner join is $R(a,b) \bowtie S(b,c)$.

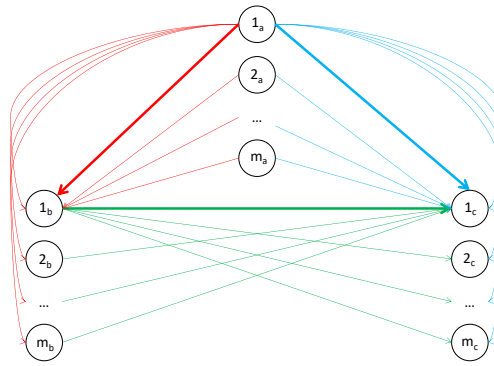


Figure 3.1: Example input database in the form of a graph on which any binary join plan is suboptimal. The figure is based on a figure in Ngo *et al.* (2013).

This computes the open triangles that consist of a red and a green edge. There are $m^2 + m - 1$ many such open triangles. The first m^2 of these are in the form of $(i_a, 1_b, j_c)$, for $i, j = [1, ..., m]$. Any other binary join plan will create the same amount of open triangles.

The core problem of binary join plans is that they can generate many intermediate tuples that do not participate in the final output. In join processing there is a technique called *semijoin reductions* that aims to address the above challenge. At a high-level, the goal of semijoin reductions is to remove "dangling" tuples from the inputs that are guaranteed to not participate in the final output. Consider the input database in Figure 3.2 and the query $R(a, b)$, $S(b, c)$, $T(c, d)$, $W(d, e)$. The tuples in R, S, T, and W are shown, respectively, as red, green, blue, and black edges in the figure. As a subgraph pattern this can be interpreted as an acyclic 4-hop query. The database has $\Theta(m)$ many tuples. Observe that the output of this query is empty as any 3-paths from left to right ends with on even d node, i.e., with 2_d, 4_d, 6_d, etc., but none of those nodes have an outgoing black edge. Consider the join plan $(((R \bowtie S) \bowtie T) \bowtie$ W). If we blindly execute this query, the first $R \bowtie S$ join would generate $\Omega(m^2)$ many tuples. However, some of the input tuples are "dangling", i.e., are guaranteed to not participate in the output. For example, none of the blue edges can participate in the output because none of them have a following black edge. In other words none of the tuples in T join successfully with W. Recursively, once blue edges are

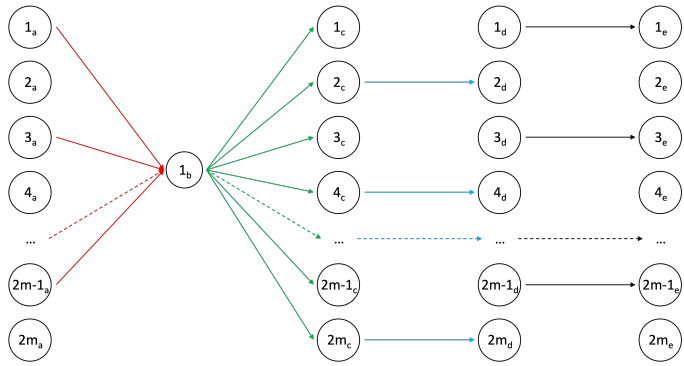


Figure 3.2: Example input database in the form of a graph. Red, green, blue, and black edges represent, respectively, the tuples in $R(a, b)$, $S(b, c)$, $T(c, d)$, and $W(d, e)$. On the 4-hop query that joins these 4 tables, the output is empty and all of the edges in this graph are dangling and can be removed.

removed, we can observe that none of the green and black edges can participate in the final output. Finally, once those are removed, we can recursively observe that none of the red edges can participate in the final output. This type of recursive removal of dangling tuples is the core idea behind semijoin reductions.

In a well celebrated result, Yannakakis (1981) has shown that on any acyclic join query without projections, one can remove all dangling tuples with a linear time semijoin reduction. Let IN and OUT denote, respectively, the number of tuples in the input and output relations of a query. Yannakakis (1981) has shown that after removing all dangling tuples in linear time, one can pick any binary join plan P and have the guarantee that the intermediate result sizes of P will not be larger than the output size. This implies that there is a join algorithm for acyclic queries that runs in time $\Theta(\text{IN} + \text{OUT})$. This is an *instance-optimal* algorithm, which is the highest notion of algorithmic optimality. No algorithm, modulo some offline preprocessing and indexing steps, can beat this runtime asymptotically.

Let us come back to the earlier example of running the triangle query on the database of Figure 3.1. In this example, both the input and output are of size $\Theta(m)$. However, even though the intermediate size of any binary join plan is $\Theta(m^2)$, no tuples are actually dangling.

Readers can check that every edge in Figure 3.1 participates in a triangle. Therefore, the technique of removing dangling tuples cannot improve the runtime of binary join plans. Nonetheless, can we still do better than generating $\Omega(m^2)$ many intermediate tuples? Even if we cannot have an algorithm that is instance optimal, i.e., runs in time $\Theta(m)$, can we have algorithms whose runtimes are asymptotically better than $\Omega(m^2)$ for the triangle query? These are the motivating questions that have led to the theory of WCOJ algorithms.

3.1 History of the AGM Bound and WCOJ Algorithms

We next cover some history of the algorithms for cyclic join queries and the sequence of progress in literature that led to the development of WCOJ algorithms. This section covers formal material and can be skipped by readers eager to learn about the workings of WCOJ algorithms.

It was known since the 1980s that Yannakakis's algorithm can evaluate acyclic join queries in $O(\text{IN} + \text{OUT})$ time. It was further known that the semijoin plans used by Yannakakis's algorithm could not be used for cyclic queries (or cyclic parts of queries). Therefore it was well understood that acyclicity was an important advantage for efficient evaluation of join queries. Therefore a standard approach for evaluating queries that had cyclic components was to perform *query decompositions*.

Briefly, query decompositions transform queries by evaluating/flattening their cyclic components into base relations, after which the transformed query becomes acyclic. In decomposition approaches, one pays the upfront cost of evaluating the cyclic parts of queries. Often this cost is a super-linear polynomial in the data size and the degree of this polynomial depends on the *width* of the query. Notions of query width, such as *treewidth* or *hypertree width*, are formal notions that describe "how cyclic" a query is. We will cover several of these later in this section. For our discussion here, we only note that query decomposition approaches achieve runtimes in the form of $O(\text{IN}^{f(w)} + \text{OUT})$, where w is some notion of query width and $f(w)$ is some function of width, e.g., $w + 1$. However, it was not well understood how well different notions of width characterized the costs of evaluating cyclic parts of queries, and

no major algorithmic progress for evaluating cyclic parts of queries was made.

In the late 2000s and early 2010s, a series of papers by Atserias *et al.* (2008) and Ngo *et al.* (2012) made progress along two directions:

(i) They tightly characterized the costs of evaluating cyclic parts of queries, known as *the AGM bound of queries*, which is characterized by a new structural property of queries known as their *fractional query number*.

(ii) They provided join algorithms that match the AGM bound of queries, known as *worst-case optimal join (WCOJ) algorithms*.

At a high-level, this literature observes that binary join plans of existing systems are provably subobtimal when evaluating cyclic join queries. The classic example is the triangle query: $E(v_1, v_2)$, $E(v_2, v_3)$, $E(v_3, v_1)$, whose AGM bound when the input relation E contains IN tuples is $\text{IN}^{3/2}$, whereas one can show input databases where any binary join plan will take $\Omega(\text{IN}^2)$ time. To address this sub-optimality, WCOJ algorithms evaluate queries attribute-at-a-time, instead of table-at-a-time approach of binary join plans. The core algorithmic operation of WCOJ algorithms is to perform multiway intersections of sets of values from different relations that bind to the same attribute.

The suboptimality of binary join plans manifests itself only if the relations in the queries depict m-n cardinality. For example, if the joins in a cyclic query are non-growing primary-foreign key joins, binary join plans are not sub-optimal. That is why both the cyclicity and the m-n nature of the joins are important for these algorithms to have an advantage over binary joins. This is also why these algorithms have found their best applications in the context of graph data management, where the input databases are generally assumed to be in the form of a network that depicts m-n relationships between nodes, and workloads of applications frequently find cyclic patterns in these networks, such as triangles, diamonds, or cliques, which correspond to cyclic joins.

In this section we first provide the background on the AGM bound and a core WCOJ algorithm known as Generic Join algorithm, which contains the core algorithmic steps needed to meet the AGM bound of

queries under arbitrary queries. Then we cover the different approaches that have implemented WCOJ algorithms in systems. We break these approaches into two parts: (1) evaluation approaches for WCOJ-only plans or sub-plans, where we primarily focus on the actual operators that implement WCOJ algorithms in several implementations and several optimizations for picking a good join attribute order (see Section 3.2); (2) approaches that generate plans that combine binary join and WCOJ operators.

3.2 AGM Bound and WCO "Generic Join" Algorithm

In this section, we provide some background on the foundations of WCOJ algorithms. We begin by introducing the AGM bound of Atserias *et al.* (2008) and then review a WCOJ algorithm known as "Generic Join" algorithm. The section on AGM bound is very formal and can be skipped by readers who are eager to learn about Generic Join.

3.2.1 AGM Bound

Let $\mathcal{R} = \{R_1, ..., R_n\}$ be the set of relations in a query Q and $\mathcal{A} = \{a_1, ..., a_m\}$ be the set of attributes in the relations in $\mathcal{R}$. Let $\mathbf{x} = (x_i)_{i \in \{1,...,n\}}$ be a vector of values between $[0, 1]$, one for each relation R_i in Q such that the following m inequalities hold for each attribute a_j in Q:

$$\sum_{R_i : a_j \in R_i} x_i \geq 1$$

Such a vector is called a fractional edge cover. In other words, in the hypergraph representation of the query, we give a value between 0 and 1 to each relation/hyperedge and ensure that every attribute is "covered". The AGM bound, proved by Atserias *et al.* (2008), is the following inequality about the output size:

$$|Q| \leq \Pi_{x_i \in \mathbf{x}} |R_i|^{x_i}$$

We refer readers to proofs of this bound by Ngo *et al.* (2013, Section 4.1) or the original proof by Atserias *et al.* (2008) for details. Therefore

each fractional edge cover, along with the statistics about the sizes of the relations R_i derives an upper bound on the maximum output size of Q. Atserias *et al.* (2008) have also proved that the fractional edge cover $\mathbf{x}^*$ that achieves the lowest upper bound is asymptotically tight, i.e., there exists a database instance $\mathcal{D}$, whose relations have size $|R_i|$ and the query output is of size $\Theta(\Pi_{x_i * \in \mathbf{x}^*} |R_i|^{x_i^*})$. When input relations have the same size, e.g., in subgraph queries which are modeled as self-joins over a single edge relation of a graph, the size of $\mathbf{x}^*$ is called the fractional edge cover number of Q and denoted by ρ^*. It has since been common usage in literature to call this tightest upper bound for Q as "the AGM bound of Q". Henceforth, the AGM bound of Q should be understood as referring to the $\rho*$ of Q.

Example 3.1. Consider the triangle query Q_Δ. Let us assume the query is running on an input graph, and let us represent the query as $Q_\Delta := E_1(a_1, a_2), E_2(a_2, a_3), E_3(a_3, a_1)$.[1] Suppose each E_i has IN tuples and represent the edges of a graph under the common scenario where the E_is are copies of the same edge relation. Several example fractional edge covers include (1, 1, 0), (1, 0, 1), (0, 1, 1) (recall the weights are given to (E_1, E_2, E_3)). For example, (1, 1, 0) is a cover because: (i) a_1 is "covered" as a_1 appears in E_1 and E_2 and their weights sum to more than 1: $1 + 0 = 1 \geq 1$; (ii) a_2 is covered because $1 + 1 = 2 \geq 1$; and (iii) a_3 is covered because $0 + 1 = 1 \geq 1$. Therefore each attribute is covered. Using each of these fractional edge covers in the AGM bound gives an upper bound of IN^2. The tightest upper bound is achieved with the fractional edge cover (1/2, 1/2, 1/2), which the reader can verify indeed covers every attribute. Therefore, the fractional edge cover number ρ^* of the triangle query is 3/2, so the maximum number of triangles that can exist in a graph with IN edges is $\Theta(\text{IN}^{3/2})$.

One further observation made by Atserias *et al.* (2008) was that for some cyclic queries, such as the triangle query, existing binary join plans used in systems can generate polynomially more intermediate results than the AGM bound of Q. Figure 3.1 shows an example database on

[1] Note that we used $R(a, b)$, $S(b, c)$, $T(a, c)$ notation in an earlier section where we wanted to give each edge a different color. We are changing the notation to represent the more common case when the query is a self-join.

which any binary join plan generates IN^2 many intermediate tuples as the reader can verify (in the example, the output size is $\Theta(\text{IN})$). This highlights a possible sub-optimality of binary join plans on cyclic joins. The algorithmic steps of how to fix this sub-optimality existed in the original reference of Atserias *et al.* (2008): *evaluate queries attribute at a time instead of relation at a time.*

The worst-case run time of the algorithm by Atserias *et al.* (2008) did not tightly meet the AGM bound of queries. That algorithm ran in time $O(\text{IN}^{\rho^*+1})$. Several algorithms were later introduced that fixed this sub-optimality, i.e., whose worst-case run times were asymptotically bounded by the AGM bound of queries. These include the Leapfrog TrieJoin by Veldhuizen (2012) and NPRR by Ngo *et al.* (2012). Perhaps the simplest such algorithm is the Generic Join algorithm by Ngo *et al.* (2013), which we cover in the next section. These algorithms are called *worst-case optimal join algorithms*, as their worst-case runtimes are optimal in the sense that the worst-case runtime of any join algorithm to evaluate a query Q on input databases with relation sizes $|R_i|$ is asymptotically at least lower bounded by the AGM bound of Q. This is true because the AGM bound is asymptotically tight. Therefore, any algorithm has to take at least the AGM bound time on at least one database instance just to write the output of query.

3.2.2 Generic Join WCOJ Algorithm

Given a query Q, we first fix a *join attribute order* (JAO), which is the order in which partial joins will be computed. Without loss of generality, assume the ordering J is $a_1, a_2, ..., a_m$. Further assume that the relations are pre-indexed so that given bindings to a prefix p of the $(a_1 = t_1, ..., a_{j-1} = t_{j-1})$, if a relation R_i contains a_j, using the index on R_i we can obtain "extensions" of p to a_j in R_i. Let us refer to these extensions of p in R_i as $Ext^i_j(p)$. For example, this can be done simply by building a B+ tree index on R_i that follows the order of attributes in J. As we will see later in this section, in some integrations of WCOJ algorithms in systems, this pre-indexing step is omitted, although algorithmically this is necessary for the runtime analyses of WCOJ algorithms to meet the AGM bound.

Given J and the necessary indices on the relations, the Generic Join algorithm is given in Figure 3.3. The algorithm iteratively computes partial matches P_1, P_2, ..., P_m, where P_j is the set of bindings found for the first j attributes $a_1, ..., a_j$ and is simply computed by taking each prefix $p \in P_{j-1}$, and finding each extension set Ext_j^i to variable a_j for p from each relation R_i that contains a_j and intersecting them. Note that P_j can equivalently be seen as the output to sub-query Q_j, where Q_j projects Q onto the first j attributes, i.e., $\prod_{a_1,...,a_j} Q$. Ngo *et al.* (2013) provide an inductive proof that proves that as long as the computation satisfies an *intersection property*, ignoring logarithmic factors, the worst-case runtime of Generic Join is asymptotically bounded by the AGM bound of Q (again ignoring some logarithmic factors). The intersection property requires that intersections of Ext_j^i take time linear in the size of the smallest set that's being intersected. This is a simple constraint and can be achieved easily by standard intersection algorithms that use binary-search like routines to skip over ranges of elements in sets that need to be skipped during intersecting.[2]

```
P_0={}
for (j = 1... m):
    P_j={}
    for (p ∈ P_{j-1}):
        // ∩ below is performed starting from smallest Ext_j^i(p)
        ext_p = ∩Ext_j^i(p)
        P_j = P_j ∪ ext_p
```

Figure 3.3: Pseudocode of Generic Join.

3.3 Worst-case Optimal Join Only Plans

In this section we give overviews of the common approaches across systems to implement WCOJ algorithms. We focus on the WCOJ plans

[2]Interestingly, many actual implementations of WCOJ algorithms that we are aware of ignore this constraint in most cases and do simple in-tandem intersections, because in-tandem intersections access memory sequentially, which performs well in practice.

or sub-plans of these systems and leave the coverage of how to mix these sub-plans with binary join operators to the next section. For several of these approaches, we will also discuss how to optimize the JAO of the join computation. Given a JAO J the system has picked for a WCOJ computation, each approach has two components:

1. JAO-consistent indices: The approach either maintains or builds on the fly a set of indices that store the relations in a sorted order consistent wit J for a query, i.e., if a relation R contains a_i and a_j and a_i comes before a_j in J, then a_i is earlier than a_j as a sort order in the index for J.

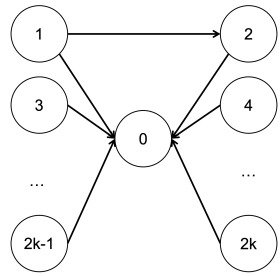
2. Operators for multiway intersections: The approach also needs to implement operators or compiled functions that perform index lookups and Generic Join-like multiway intersections.

We will cover these approaches, highlighting the pros and cons of different approaches. We note that some of the approaches we cover perform value-based joins while others are performing predefined joins. Recall from Section 2 that predefined vs value-based categorization of join algorithm implementations is orthogonal to the binary vs worst-case optimal categorization.

3.3.1 LogicBlox: Tries and Iterator-based Execution

The first published implementation of a WCOJ algorithm is the Leapfrog Triejoin (LFTJ) algorithm by Veldhuizen (2012). LFTJ is the core join algorithm of the LogicBlox system (Aref *et al.*, 2015). We will start with explaining the implementation on the acyclic triangle Q_Δ query $E_1(a_1, a_2) \bowtie E_2(a_2, a_3) \bowtie E_3(a_1, a_3)$ on the example database shown in Figure 3.4. The implementation is based on indexing the relations according to the JAO J, which we will take to be a_2, a_3, a_1. LogicBlox publications that describe LFTJ algorithm assume that indices consistent with J already exist in the system.[3] So it is not clear

[3]The implementation described by Aref *et al.* (2015) stores actual record values, so the implementation is value-based though if joins are predefined to the system, one can easily index RIDs in these indices and implement a predefined version of LogicBlox's approach.



Edges	
src	dst
1	0
1	2
2	0
3	0
...	...
2k	0

Figure 3.4: Example database of edge relations in both graph and tabular representations.

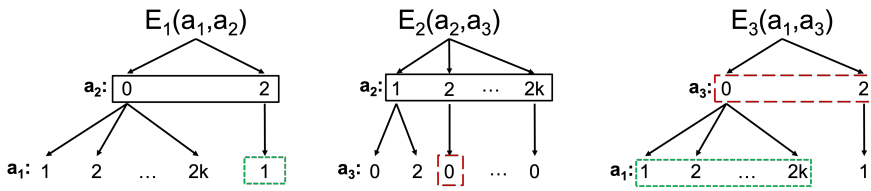


Figure 3.5: Trie indices used by LFTJ corresponding to each relation in Q_Δ. JAO is $\{a_2, a_3, a_1\}$.

if these indices incur the cost of sorting on the fly but we know from Veldhuizen (2012) that the index used is a B+ tree. For the J in our running example, the indices would be ordered as shown in Figure 3.5.

LogicBlox's implementation has an iterator interface to the indices with two functions, `next()` and `seek(key)`. Then, the algorithm will perform a sequence of "Leapfrog joins", which are equivalent to one attribute extensions of P_{j-1} to P_j. For Q_Δ, the three Leapfrog joins for our chosen J in Datalog syntax would be: (i) $E_1(\_, a_2)$, $E_2(a_2, \_)$; (ii) $E_{2_{a_2}}(a_3)$, $E_3(a_3, \_)$; and (iii) $E_{1_{a_2}}(a_1)$, $E_{3_{a_3}}(a_1)$. Here $E_1(\_, a_2)$ and $E_2(a_2, \_)$ are the set of a_2 values in these relations (similarly for $E_3(a_3, \_)$). $E_{i_{a_k}}(a_t)$, is the set of a_t values for a specific a_k value in E_i. The implementation is recursive and based on performing intersections using a sequence of `seek()` calls to the set of iterators at each level of the recursion.

In our example, this is done as follows. At depth 0, the algorithm has 2 iterators over the indices E_1 and E_2, which iterate through the a_2 values in these indices by making a sequence of `seek()` calls to find

common values across the indices. Figure 3.6 shows the pseudocode of the computation at each recursive depth implemented in a function called LFTH-Intersect(). The code assumes that there is an `allIters` array of arrays that contains the iterators of each recursive depth. At depth 0, this computation takes the intersection of $E_1(\_, a_2)$, $E_2(a_2, \_)$ through seek() calls that execute until a common value across all iterators is found or one of the iterators reaches its end. The sets of values these iterators iterate over are drawn inside black boxes in Figure 3.5. As soon as the first intersection value is found, which in our example would be (a_2=2), the recursion goes to the next depth (`LFTH-Intersect(d+1)` call in Figure 3.5).

```
Input: depth d;
Output: bind a single value to a_j, where a_j is the
        dth variable in the JAO.
LFTJ-Intersect(d):
// Assume allIters contains the iterators at each depth
iters=allIters[d]
for iter in iters:
   iter.init() // move iters to their first keys
   iter.sort() // sort iters on their keys.
max = iters[|iters|-1].key() // max key
ℓ = 0
while true:
  min = iter[ℓ].key() // min key
  if (min = max):
     a_j=min // found an intersection
     if (d < maxDepth):
       LFTJ-Intersect(d+1) // move to the next iteration
     else: output tuple // computed a successful join tuple
  else:
      iters[ℓ].seek(max)
      if iters[ℓ].atEnd(): return;
      else:
         max = iters[ℓ]
         ℓ++
```

Figure 3.6: Pseudocode of LFTJ implementation at one depth.

At the next depth, two new iterators are initialized. First is initialized to E_2's second level below a_2 = 2. This iterator iterates over a single

value, since $a_2 = 2$ contains a single child in E_2's trie. Second is initialized to E_3's first level which contains 0 and 2 values. The sets of values of these iterators are drawn inside red boxes with long dashes in Figure 3.5. The `LFTJ-Intersect()` code binds $a_3 = 0$ and moves to the next depth, where the 2nd level iterator of E_1 (under $a_2 = 2$ and 2nd level iterator of E_3 (under $a_3 = 0$) are intersected to produce outputs.

We end this subsection with a few observations. First, the `seek()` calls to the iterators described in Figure 3.6 automatically satisfy the intersection property. As we will see later, some other implementations inspect the sizes of sets of values to start intersections from the smallest set. Second, the papers on LeapfrogTrieJoin do not cover how the system picks a JAO for a query but acknowledge the importance of this choice for performance. We will discuss this more in the rest of this section.

3.3.2 Umbra: On-the-Fly Hash Tries and Iterator-based Execution

Umbra's implementation of WCOJ sub-plans by Freitag *et al.* (2020) is similar to LFTJ. Umbra has a rule-based approach, which will be discussed momentarily, that transforms the system's binary join plans into those that contains WCOJ sub-plans. We first briefly discuss the computation that is performed in Umbra's WCOJ operator. In this operator[4] a LFTJ-like computation is performed using "hash tries" instead of sorted tries. Hash trie data structure of Umbra is a set of nested hash tables that store the hashes of the values in tuples (instead of the actual values). The rationale for this is that comparison of actual values can be expensive especially if the keys are variable length in size and can be replaced by lookups in hash tables. Additionally, storing the actual values in the indices increases the memory footprint and decreases the cache performance when iterating over indices.

Figure 3.7 shows an example hash trie for E_1 and E_2 for the JAO J $\{a_2, a_3, a_1\}$. The top level of these tries contain the hashes of the first attribute in the relation according to the given JAO (sorted according to the hashes). For example, E_2's hash trie stores the hash of each a_2

[4]Umbra compiles queries to executable code, so there are no "operators" as in traditional DBMSs. The computation described here is part of the compiled code for a plan.

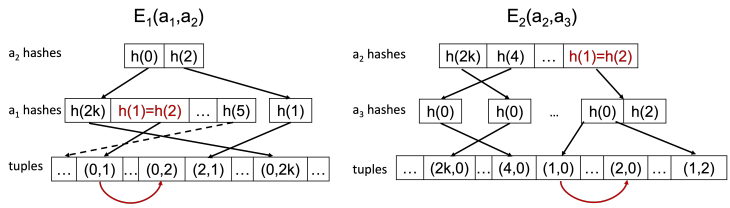


Figure 3.7: Hash trie indices used by Umbra corresponding to E_1 and E_2 in Q_Δ. JAO is $\{a_2, a_3, a_1\}$.

value in E_2 and not the actual values. Each of these hash values has a pointer to second level hash tables. The second level hash tables also store the second attribute in the JAO for the given prefix of values. For example, the leftmost hash table for a_3 hashes the a_3 value of every tuple in E_2 whose a_2 hash value equals h(4) (which is 4's hash value). In the example there is only one such tuple ($a_2 = 4$, $a_3 = 0$). Therefore this hash table contains a single hash value. In contrast, the right most hash value at that level contains hashes of a_3 value of tuples whose a_2 values hash to h(1) or h(2) (note that there is a hash clash for h(1) and h(2) at the first level). There are three such tuples: ($a_2 = 1$, $a_3 = 0$), ($a_2 = 2$, $a_3 = 0$), ($a_2 = 1$, $a_3 = 2$). Therefore this hash table will store the hashes of 0 and 2, which are two distinct a_3 values in these three tuples.

The last-level hash tables point to the actual tuples. Given that there may be some hash collisions in upper layers, a hash value from the last-level hash table may need to point to multiple tuples, which is done by chaining some tuples in this layer. For example, in our case, the tuples ($a_2 = 1$, $a_3 = 0$), ($a_2 = 2$, $a_3 = 0$) follow the same set of chains in the hash trie for E_2, so (1,0) in the last level points to (2,0) (and there are no pointers directly to (2,0) from the last-level hash tables).

The use of hash tries requires some modifications to the LFTJ implementation of LogicBlox. At each level of the recursion, the computation initializes iterators similar to LFTJ (recall Figure 3.6). Then the hash table with the fewest hash values, say H_{min} will be identified, and the computation will iterate over the values in H_{min}. Then, each hash value in H_{min} will be looked up in the rest of the hash tables that are being iterated over. If the same hash value is found in all other hash tables, then the next recursive call will be made (also Volcano style). In the

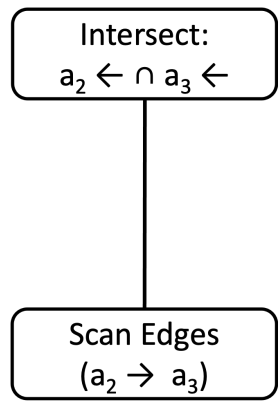
last level of the recursion, sets of tuples from each relation from the last levels of the index of each relation in Q, and the final key comparisons are made to ensure tuples that indeed match on their common attributes are kept and tuples whose values do not match (but the hashes of these values match) are removed.

For example, in our running example we assumed that there was a collision of h(1) and h(2). Let us call this hash value h*. Consider these combination of hash values: (a_1=h*, a_2=h*), (a_2=h*, a_3=h(0)), (a_1=h*, a_3=h(0)). Umbra's LFTJ computation will generate 4 possible, not necessarily actual, triangles for these hash values. When iterating over the E_1 index, only the (a_1=2, a_2=1) tuple is extracted as there is no tuple with a_2=2 in this relation, so the collision does not lead to getting additional tuples. From E_2, (a_2=1, a_3=0) and (a_2=2, a_3=0) will be extracted by following the red pointer at the last level in the index. Similarly from E_3, whose index is omitted, (a_1=1, a_3=0) and (a_1=2, a_3=0) will be extracted. Out of these 4 combinations only (a_1=2, a_2=1), (a_2=1, a_3=0), and (a_1=2, a_3=0) form a triangle and the other 3 are false positives and will be eliminated at the last verification stage of the algorithm.

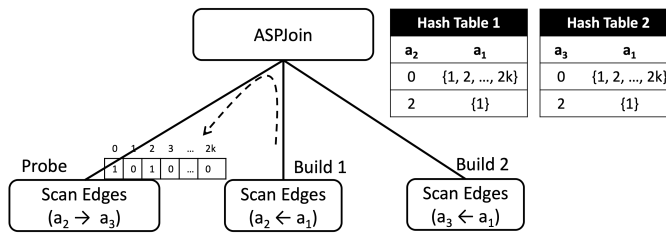
3.3.3 Graphflow: Pre-Sorted Adjacency Lists and Extend-like Join Operators

Graphflow (Mhedhbi *et al.*, 2021) is an in-memory GDBMS that can generate plans with WCOJ computations. Graphflow performs its joins over one or more `Edge` relations in the system, which contain the source and destination node record IDs (RIDs) in the system. Therefore, all joins in the system are predefined and based on RIDs. In this section we will assume that there is a single `Edge` relation in the system for simplicity. In Graphflow, this relation is indexed twice as a forward and a backward "adjacency list" index. Further, the system keeps each adjacency list in already sorted order of their neighbor node IDs. One advantage GDBMSs have is that the `Edge` relation is binary. Therefore, storing 2 pre-sorted indices is all that is needed to be able to generate any JAO for any (self-) join query over `Edge` relation (but not for arbitrary joins). Therefore, for these join queries no indices need to be built on the fly to perform a Generic Join-like computation.

Graphflow's WCOJ sub-plans are very simple and consist of: (i) a Scan operator that scans the `Edge` relation and binds each (src, dst) tuple to the first two attributes in the JAO, i.e., computes partial matches P_2 in our pseudocode of Generic Join in Figure 3.3; and (ii) a sequence of `Extend/Intersect` operators that each extend a previous set of partial matches to the next attribute in the given JAO starting from P_3, ..., P_m. Figure 3.8a shows the Graphflow plan for Q_Δ according to JAO a_2, a_3, a_1. In terms of the query processor architecture, Graphflow's query processor is Volcano-style, where parent operators pull tuples from child operators. The intermediate tuples are pulled in the form of *factorized vectors*, which will be covered in Section 4 on factorization. Here, readers can assume that single flat tuples are passed between operators.



(a) Graphflow plan.



(b) Kùzu plan.

Figure 3.8: Example Graphflow and Kùzu plans for Q_Δ for the JAO a_2, a_3, a_1. `Intersect` is short for the `Extend/Intersect` operator in the Graphflow plan.

Consider an `Extend/Intersect` operator that will extend partial matches P_{j-1} to P_j. For each tuple t that the operator receives, the operator accesses one or more adjacency lists (forward or backward) from the adjacency list index and intersects them to compute an extension set $S = \{s_1, s_2, ..., s_\ell\}$ (if there is a single adjacency list, this operation is a simple extension). Then for each s_k, the operator produces one new tuple that's passed to the next operator by appending $a_j = s_k$ to t. As an example, consider the plan in Figure 3.8a, where the Scan operator scans each edge in the input graph and binds to an (a_2, a_3) tuple. For each edge, say, $(a_2 = 2, a_3 = 0)$, given that Q_Δ contains relations $E_1(a_1, a_2)$ and $E_3(a_1, a_3)$, the operator needs to access: (i) the backward adjacency list of node 2 to read $E_1(a_1, a_2 = 2)$=$\{1\}$; and (ii) the backward adjacency list of node 0 to read $E_3(a_1, a_3 = 0)$=$\{1, 2, ..., 2k\}$. This intersection would return S=$\{1\}$ and lead to outputting the triangle (a_1=1, a_2=2, a_3=0).

We discuss several implementation details of Graphflow's plans. First, Graphflow's plans avoid any intersections when binding the first two variables in the JAO. In practice, this may be an advantage or a disadvantage. For example, in our running example and the plan in Figure 3.8a, Graphflow's `Extend/Intersect` operator will get each edge in the graph from the Scan operator (so a total 2k+1 intermediate results) and try to perform intersections on each one. Instead, both LogicBlox's and Umbra's implementations would have intersected $E_1(\_, a_2)$ and $E_2(a_2, \_)$ to bind the a_2's. This only returns 2, which would then be extended only to a single tuple (a_2=2, a_3=0). In contrast, there is also an important advantage because the first levels of the trie indices over relations can be large, so avoiding intersecting them can be beneficial. This observation will also motivate the FreeJoin work by Wang *et al.* (2023) that unifies WCOJs and binary joins. We will cover FreeJoin in Section 3.5. Second, Graphflow's intersections are performed by simple in tandem intersections. When more than two lists are being intersected, the operator performs a sequence of pairwise intersections. The first intersection intersects the smallest list with another list, the result of which is intersected with the next list, so on and so forth.

The primary focus of the papers on Graphflow's implementation of WCOJs is on how to pick a good JAO for a query and how to mix

the WCOJ-only sub-plans of the system with binary join operators to obtain bushy plans. We will discuss bushy plans later. Here we discuss the system's approach to picking a JAO for WCOJ-only sub-plans. This is based on enumerating all possible JAOs[5] and estimating the cost of each using a cost metric called *intersection cost*, which is defined as follows:

$$\sum_{P_{j-1} \in P_2 \ldots P_{m-1}} \sum_{t \in P_{j-1}} \sum_{\substack{\text{adj intersected} \\ \text{to extend } t}} |adj| \tag{3.1}$$

The inner two summations model the expected sizes of the adjacency lists that will be intersected when extending each tuple $t \in P_{j-1}$ to P_j. The outer summation loops over each set of partial matches. In summary, the intersection cost of a plan is the total sizes of the adjacency lists that will be intersected in the `Extend/Intersect` operators of the plan.

This cost-metric captures two important observations. First is that different JAOs for the same query can yield different intermediate results and therefore runtimes. This is expected and is captured in the i-cost formula which loops over each expected intermediate tuple. Second two plans that generate exactly the same number of intermediate results may still yield different runtimes for intersecting adjacency lists in different directions. For example, consider the JAO a_1, a_2, a_3. A Graphflow plan for this will scan each edge $E(u, v)$ in the input graph as before and intersect u's forward adjacency list with v's forward adjacency list. Recall that the plan in Figure 3.8a intersected the backward adjacency list of both u and v. In practice, because the distributions of backwards and forward adjacency lists can be very different, this can make an important difference (Mhedhbi and Salihoglu, 2019) reports 12.1x difference on a microbenchmark experiment on Q_Δ on a small web graph). For example on social networks, backward adjacency lists can be much larger than forward ones. A classic example is that on the Twitter network, popular users can have tens of millions of followers (e.g., Donald Trump) yet users will tend to follow at most tens of thousands of other accounts. Suppose, the Donald Trump node, DT has 60 million incoming edges. Consider

[5]When enumerating all JAOs is prohibitive, Graphflow falls back to a greedy approach (Mhedhbi and Salihoglu, 2019).

running the vanilla Q_{Δ} query on the plan from Figure 3.8a. Then, for 60 million (u, DT) edges, the plan would execute an intersection where one of the lists would be of size 60 million. This is possibly a very expensive computation even if the costs of intersections are commensurate with the size of smaller lists, considering possible data copies that would be performed in the query processor and the CPU caching effects while performing these intersections. Intersecting forward lists can avoid this phenomenon.

Finally, Graphflow estimates the sizes of sub-queries/sub-graphs using a cardinality estimation technique based on using a *subgraph catalogue*. We omit details here and refer the reader to the paper by Mhedhbi and Salihoglu (2019). Briefly, this catalogue stores estimates for the cardinalities of small size subgraphs, e.g., two-paths, asymmetric and symmetric triangles, and different possible extensions of these subgraphs to larger graphs, and each extension estimates the sizes of the adjacency lists that would be intersected. As the system enumerates different JAOs, the catalogue is consulted to compute the i-costs of different plans.

3.3.4 Kùzu: On-the-fly Sorted Adjacency Lists and `HashJoin` Operators

Kùzu is another GDBMS and the successor of Graphflow developed in the same research group at University of Waterloo. Similar to Graphflow, Kùzu also performs predefined joins and has a join index that stores the RIDs of node records. However, in contrast to Graphflow's in-memory architecture, Kùzu stores its join index on disk and in an unsorted manner. Recall from Section 2 that `Extend/Intersect`-like operators, which access the index for each tuple, can be very inefficient and lead to random I/Os if the index is managed on disk. To avoid this, Kùzu has a multi-way `HashJoin` operator called *ASPJoin* that ensures that each relation is scanned sequentially and only once. Further, the adjacency lists in the indices are not sorted in Kùzu, so before intersecting, these lists need to be sorted. In our coverage of Kùzu's implementation, we omit the planning part of queries, which is based on similar cost-based ideas from Graphflow. Instead, we focus on the core join operator that performs WCOJ-style multiway intersections.

Kùzu has several variants of `ASPJoin`. We cover one variant here and refer the readers to the paper by Feng *et al.* (2023) for more details. `ASPJoin` takes as input a relation P_{j-1} and ℓ many binary relations that will be used to extend P_{j-1} partial matches to P_j partial matches to extend by a single attribute. Similar to regular hash join algorithm the computation is broken into two phases:

- **Build phase:** The operator scans the adjacency lists corresponding to each of the ℓ "build relations" and forms ℓ many hash tables. For each hash table, the key is an attribute that exists in the tuples in P_{j-1} and values are the sorted list of a_j values. The sorting is done on the fly. After each hash table is created, the key values in the hash tables are passed down as a semijoin filter to the probe side.
- **Probe phase:** Then the tuples in P_{j-1} are scanned and for each tuple t, one sorted list from each hash table is looked up and intersected to produce a set of extensions of t.

An example Kùzu plan using an `ASPJoin` operator is shown in Figure 3.8b according to our example JAO. The execution is performed as follows:

- **Build phase:** First the backward adjacency lists index is used to scan (a_2, a_1) tuples and a Hash Table 1 is built. Note that the parent `ASPJoin` will use this index to probe the extensions of a_2 values to a_1 values. Therefore, in Hash Table 1, a_2 is the key and values are the set sorted a_1's. Next, a Hash Table 2 that hashes a_3 values to a set of a_1 values is built.[6]
- **Sideways semijoin filter passing:** Recall from Section 2 that Kùzu has a sideways information passing optimization to avoid scanning parts of adjacency list indices that are guaranteed not to successfully participate in the join. After the build phase, the system knows from Hash Table 1 that only a_2 values with 0 and 2 need to be scanned in the probe phase, which will scan (a_2, a_3) tuples. This information is passed sideways to the probe phase in the form of a semijoin filter.
- **Probe phase:** The computation first scans the forward adjacency lists of node 0 and 2 and each scanned (a_2, a_3) tuple is extended to

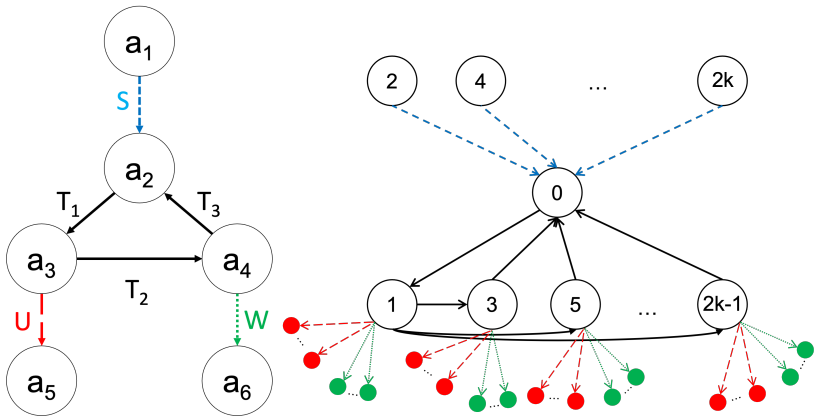
[6]As readers will observe, in our example these hash tables are identical, so in an optimized implementation this creation of Hash Table 2 can be completely avoided.

a_1 values as follows. Next, two lookups are made in Hash Tables 1 and 2, respectively with keys $a_2 = 0$ and $a_3 = 2$, and two sets of a_1 values are obtained. Finally, these two sets are intersected to produce (a_1, a_2, a_3) tuples.

3.3.5 CTJ: Caching of Common Extensions

During vanilla WCOJ plan evaluation, the same intersections can be computed multiple times. Consider the query $Q_{\Delta\star}$ shown in Figure 3.9a. The query has a triangle in the middle and three edges coming into or out of each query node in the triangle. To simplify our presentation, different query edges have different colors. We consider a database with four different types of relations, blue small-dashed edges S, black edges T_1, T_2 and T_3, red long-dashed edges U, and green dotted edges W. The query can be expressed in our syntax as $S(a_1, a_2)$, $T_1(a_2, a_3)$, $T_2(a_3, a_4)$, $T_3(a_4, a_2)$, $U(a_3, a_5)$, $W(a_4, a_6)$. Consider running Generic Join on the JAO $a_1, a_2, a_3, a_4, a_5, a_6$ on the database shown in Figure 3.9b. The database contains $k-1$ triangles (0, 1, 3), (0, 1, 5) ..., (0, 1, $2k-1$). 0 has k incoming blue small-dashed edges. Nodes 1, 3, ..., $2k-1$ each have k outgoing red and k outgoing green edges. Note that when executing Generic Join, for each of the blue edges matching (a_1=2j, a_2=0), the same set of k-1 triangles will be computed to bind to a_3 and a_4. Similarly for each triangle that is computed, say (a_2=0, a_3=1, a_4=3), and each extension of a_3=1 to the red nodes to bind to a_5, the same set of extensions to the green nodes will be made when binding a_6. Therefore, in a vanilla evaluation of Generic Join there will be many repeated computations. *In the rest of this section (covering CTJ and binary join plans) and the next section (covering factorization), the techniques we cover can be seen as different techniques that aims to avoid such repeated computations.*

CTJ's approach for avoiding repeated the computation is to put caches in different depths of the recursion of LFTJ. Specifically, after binding certain values to $a_1, ..., a_j$, CTJ caches and reuses a subset of the bindings for $a_{j+1}, a_{j+2}, ...$ for a given set of keys that are bound to $a_1, ..., a_j$. To decide the set of key and value attributes, CTJ generates tree decompositions (TDs), which are a specific type of bushy plans.



(a) $Q_{\Delta\star}$ query. **(b)** Example database.

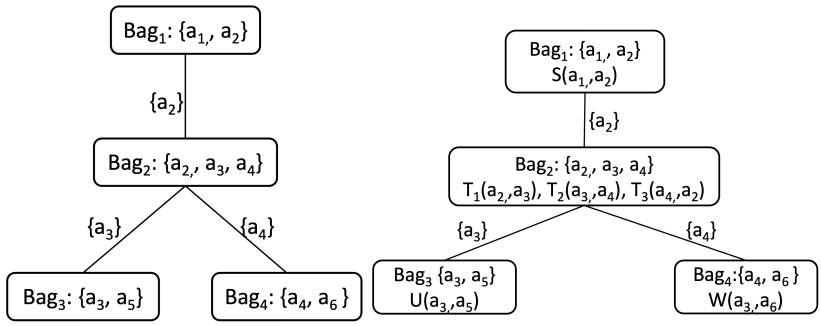
Figure 3.9: Example query and database for the two triangles query $Q_{\Delta\star}$.

Formally, a TD of a query $Q(A) = R_1(A_1), ..., R_m(A_m)$ is a rooted tree, whose nodes are called *bags*, and is assigned a set of attributes that satisfies two properties:

1. Each relation's attributes must be fully assigned to some bag.
2. For any attribute a_i, the sub-tree of bags that contain a_i must be connected (also called the *running intersection or connectedness property*).

Each bag of a TD represent the projection of Q into a set of attributes. An example TD for $Q_{\Delta\star}$ is shown in Figure 3.10a with four bags. For example Bag_1 represents just S, Bag_2, which is $\prod_{a_2,a_3,a_4} Q$, represents the join of $T_1(a_2, a_3)$, $T_2(a_3, a_4)$, and $T_3(a_4, a_2)$, Bag_3 represents U, and Bag_4 represents W.[7] Recall from Section 3.1 that in query decompositions, if one models each bag as a relation, the entire decomposition represents an acyclic join query over the "bag relations". In other words, the query $Bag_1(a_1, a_2) \bowtie Bag_2(a_2, a_3, a_4) \bowtie Bag_3(a_3, a_5) \bowtie Bag_4(a_4, a_6)$ is equivalent to $Q_{\Delta\star}$.

[7]Another type of decomposition generalized hypertree decompositions makes this connection to bags more concrete by also assigning relationships to the bags. This will be covered momentarily when we discuss the EmptyHeaded system.



(a) TD **(b)** GHD

Figure 3.10: Example TD and GHD for $Q_{\Delta\star}$.

Consider a left-deep binary join plan on the bag relations: $(((Bag_1(a_1,\ a_2) \bowtie Bag_2(a_2, a_3, a_4)) \bowtie Bag_3(a_3, a_5)) \bowtie Bag_4(a_4, a_6))$. Then consider running the inner-most join $(Bag_1(a_1, a_2) \bowtie Bag_2(a_2, a_3, a_4))$. Then one can interpret the bags as caching common computations within the binary join plan. For example, the inner-most join $(Bag_1(a_1, a_2) \bowtie Bag_2(a_2, a_3, a_4))$ is equivalent to joining $S(a_1, a_2)$, $T_1(a_2, a_3)$, $T_2(a_3, a_4)$, $T_3(a_4, a_2)$. The $Bag_1(a_1, a_2) \bowtie Bag_2(a_2, a_3, a_4)$ join can be interpreted as caching for each $a_2 = x$ the output of the $T_1(a_2 = x, a_3)$, $T_2(a_3, a_4)$, $T_3(a_4, a_2 = x)$ join, instead of computing this multiple times for each $a_2 = x$ value.[8] However to benefit from such repeated common computations using TDs, one has to fully materialize each bag of TD, which may be superlinear in size. How can we benefit from repeated computations without incurring such memory costs? CTJ addresses this question in the context of WCOJ algorithms.

CTJ assumes that a "good" TD has been picked by a system and uses the TD to pick a JAO that can benefit from caching.[9] At a high-level and simplified for ease of presentation, the JAO is computed as follows. A preorder traversal of the TD from its root is done and as each bag is visited, each attribute of the bag that has not yet been listed is listed in some order (or using some heuristic). In our example, the

[8]The same intuition applies in general to any bushy join plan.

[9]The issue of picking a "good TD" will be discussed later when we discuss the EmptyHeaded system later.

preorder traversal would list the root, and suppose we list its attributes in the order a_1, a_2. Then we visit the child of the root, and list the two "uncovered" attributes a_3, a_4 in some order. We call these the attributes *owned by the bags*. Any attribute that is not owned by a bag is an *intersection attribute* with the parent of the bag. Then we visit the left grandchild of the root and list a_5 and finally visit the right grandchild and list a_6. Then at each non-root bag B, there is cache whose key is the intersection attributes with the parent and the values are the set of attributes that were owned by B. In our example, there would be a cache in the child of the root that stores a_3, a_4 values keyed by a_2 values. The intuition is that following JAO $a_1, a_2, a_3, a_4, a_5, a_6$, at the 3rd level of GJ, i.e., after having computed a_1, a_2 values, the set of a_3, a_4 values that are computed are independent of the a_1 values and only depend on a_2. Therefore, we can cache and reuse them. Similarly there would be two other caches for the left and right grandchildren nodes of the root. For example, for the left grandchild the key would be a_3 and the value attribute would be a_5.

Given the JAO and the choice of caches, CTJ runs LFTJ except before enumerating certain sequences of variables checks some of its caches to see if parts of the computation can be skipped. In our example, CTJ's LFTJ code would start computing $a_1 = 2, a_2 = 0$ and then move on to computing the k-1 triangles in Figure 3.9b that have the form of $a_2 = 0, a_3 = 1, a_4 = 2j + 1$, for j=2,...,k-1. Then these would be put into the cache with key $a_2 = 0$. When the recursion unrolls to enumerate $a_1 = 4, a_2 = 0$, since the cache contains key $a_2 = 0$, the set of $a_3 = 1, a_4 = 2j + 1$ are directly enumerated without calling the next two levels of the recursion.[10]

Kalinsky *et al.* (2017) describes CTJ as a general approach to enhance LFTJ with caching. It takes as input different TDs and different caching policies. Therefore the caches do not need to be able to store all

[10]Readers may observe that in fact one could cache for $a_2 = 0$ the bindings of the rest of all attributes, a_3, a_4, a_5, a_6, since all of these attributes are independent of a_1 once a_2 is fixed. The theory of factorization (Section 4) builds a more complete way of exploiting such conditional independences between attributes in queries to avoid repeated computations. Using techniques of factorization, one would cache the entire sub-tree under the root.

intermediate results for the sub-queries they cache. If a system cannot keep these sub-queries entirely, an eviction policy can ensure keeping these relations partially and recomputing parts of them when needed.

3.4 Mixing With Binary Joins

In this section we review approaches from literature that generate plans that mix traditional binary join operators with WCOJ sub-plans. As mentioned above, both CTJ's enhancement of LFTJ and bushy plans are different approaches that achieve the similar goals of avoiding re-computation of certain sub-queries.[11] For example, in the $Q_{\Delta\star}$ example, the benefit obtained by CTJ for caching and reusing the computation of triangles would be achieved by a bushy plan that has a top `HashJoin` operator with two branches (ignoring the rest of the query): (i) left sub-tree that joins $BD(a_1, a_2)$; and (ii) right sub-tree that joins $B_1(a_2, a_3), B_2(a_3, a_4), B_3(a_4, a_2)$. Suppose the right sub-tree is the build side. In this case, the triangles would be computed once in the right sub-tree and cached in a hash table and for each scanned BD tuple ($a_1 = 2j$, 0) for j=1,...,k, from the left sub-tree, this computation would be reused.

3.4.1 EmptyHeaded: Tree decompositions

EmptyHeaded by Aberger *et al.* (2017) was the first system implementation that described an approach of generating plans that mix both binary join operators and WCOJ computations. EmptyHeaded is a prototype system that stores and evaluates join queries on arbitrary relations. EmptyHeaded's plan generation is solely based on a specific form of TDs called *generalized hypertree decompositions* (GHDs). GHDs are extended TDs, where each bag is also assigned a set of relations and has an additional constraint that each variable assigned to each bag must also appear in some relation assigned to that bag. For our goals, the formal details of GHDs will not be important. The GHD

[11] In fact, as we will see, the technique of factorization that we will cover in Section 4 also achieves the same goal yet it also is a compression technique so offers additional benefits.

corresponding to the TD that we used for $Q_{\triangle\star}$ is shown in Figure 3.10b. Instead, EmptyHeaded has a specific interpretation of a GHD D as bushy plans as follows:

(i) First, the relations assigned to each bag are joined and materialized into an intermediate relation using Generic Join. EmptyHeaded does not propose a way to order the JAO within each bag. So, the choice of JAO is arbitrary. For each child bag Bag_c, EH materializes a relation. Let us call this relation also as Bag_c. Bag_c is indexed by the intersection attributes that Bag_c has with its parent Bag_p (if any). For example, Bag_3 in Figure 3.10b would be indexed by a_3.

(ii) Second phase is to run Yannakakis's algorithm. Each child bag Bag_c passes to its parent Bag_c the projection of itself onto the intersection attributes of Bag_c with Bag_p. Then in a top-to-bottom phase, the GHD D is traversed in pre-order fashion and each parent's tuples are joined with its children relations, using the indexes created over the materialized relations of its children.

EmptyHeaded's approach to picking a GHD is based on picking one of the GHDs for Q with the minimum *generalized hypertree width* (ghw). Ghw is a number that characterizes the largest AGM bound of an intermediate relation, i.e. bag, in a GHD D. In other words, it tries to capture how large will be the largest intermediate result if a query was decomposed using D. If we assume for simplicity that each relation has IN many tuples, the total runtime of using D in EmptyHeaded's approach can be upper bounded by $O(\text{IN}^{D_{ghw}} + \text{OUT})$, where D_{ghw} is the ghw of D. This is a query-only approach cost-metric that only uses the number of tuples in relations as a statistic when picking a plan. GHDs with the minimum width guarantee that the worst-case runtime over the plan is asymptotically minimized (given that Yannankakis's algorithm is used in the binary join phase).

However, this style of approach also has several shortcomings. First, finding the GHD with the minimum width is NP-hard. However, the complexity parameters here is on the number of attributes and relations in the query and independent of the database. So if we assume that in practice many queries are small, this may not be a major performance bottleneck. Yet, it still deviates from the standard join optimizers in

systems which are based on dynamic programming. This makes it challenging to integrate into existing systems. Second, EmptyHeaded leaves the picking of JAO in the first phase of the algorithm undefined. Finally, these plans are limited to performing a sequence of binary joins after performing WCOJ-like intersections. That is these plans cannot generate a plan that uses WCOJ-like intersections after performing binary joins, which may be beneficial on some complex queries. Some of these shortcomings have motivated the next approach of mixing binary and WCOJ operators that was developed in the Graphflow system.

3.4.2 Graphflow/Kùzu: Cost-based Mixed Dynamic Programming Join Plan Enumeration

Graphflow and its successor Kùzu adopt the common approach of using dynamic programming (DP) cost-based optimization to pick a join plan. At each iteration j of the DP optimization, DP-based optimizers compute for each connected sub-query Q_j that contains j relations the best plan P_j using best plans for two sub-queries that respectively contain i and $j-i$ many relations. In contrast, Graphflow's optimizer at iteration j enumerates plans for sub-queries that contain j attributes/query vertices. For queries with 2 attributes, the optimizer only enumerates plans that scan base `Edge` relations. When iterating a plan for a sub-query Q_j with $j > 2$ attributes, the optimizer considers two different sets of options:

- **Appending an `Extend/Intersect`:** The optimizer picks the best found plan P^*_{j-1} for a sub-query Q_{j-1} and append an `Extend/Intersect` operator to P^*_{j-1} that extends it by one query vertex a_j. Suppose there are t many query edges that contain a_j and whose other query vertex is in Q_{j-1}. Then, the `Extend/Intersect` operator performs a multiway join of t relations.
- **Appending a `HashJoin`:** Pick the best plans P^*_{c1} and P^*_{c2} for two sub-queries Q_{c1} and Q_{c2} whose binary join would compute Q_j. Append a `HashJoin` operator as the root of a new plan that has P_{c1} on one side and P_{c2} on the other side.

Therefore, for each Q_j, the optimizer enumerates both the best plan whose last operator is a WCOJ-like operator as well as the best plan

whose last operator is binary join operator. The optimizer then picks the better of these two. The system adopts the i-cost metric from Section 3.3.3 for `Extend/Intersect` operators and a separate metric that estimates the cost of `HashJoin`. Importantly, both i-cost and `HashJoin`'s cost estimate number of tuples that operators process. Therefore, unlike EmptyHeaded's GHD-based optimization, Graphflow's optimizer is not agnostic to the statistics about the underlying database.

Further, this approach can generate a larger set of plans that seamlessly mix binary joins and WCOJ plans. That is, there is no restrictions that first some WCOJ-only sub-plans will execute and then a set of binary joins. For example, consider the plan in Figure 3.11, which has an `Extend/Intersect` as the top operator and comes after a `HashJoin`. Such plans cannot exist in GHD-based plans, where binary join operators are performed completely after each bag is computed, which is where a WCOJ algorithm is used. As shown by Mhedhbi and Salihoglu (2019), in some queries, such plans can be more performant than GHD-based ones. On that other hand, Graphflow's simple extension of existing DP-based optimizers assumes that best plans for smaller sub-queries can be re-used to generate best plans for larger sub-queries. Mhedhbi and Salihoglu (2019) argues that this assumption is not necessarily true when enumerating WCOJ-only plans.

3.4.3 Umbra: Rule-based Binary Join Plan Modification

Unlike EmptyHeaded and Graphflow, Umbra has a rule-based approach for generating plans that mix binary joins and LFTJ computation. The rule is very simple. The system first optimizes a binary join-only plan P. Then it traverses the plan to detect a "growing" binary join operator o. o is growing if its expected output cardinality is larger than the larger of its two inputs o_ℓ, o_r. Such operators are removed and its two children are merged into o's parent, which becomes a multiway join operator. A multiway join operator is then compiled into hash trie-based LFTJ implementation that we described in Section 3.3.2.

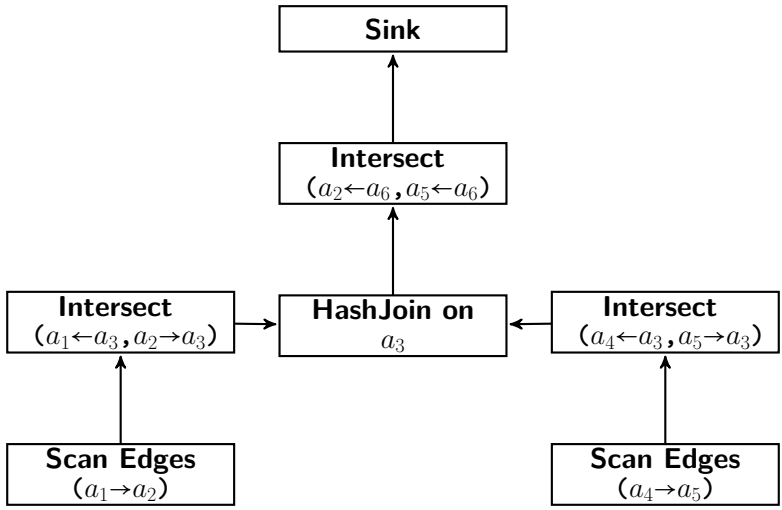


Figure 3.11: An example Graphflow plan that seamlessly mixes binary and WCOJ-like multiway intersection computations. This figure is based on a figure from Mhedhbi and Salihoglu (2019).

3.5 FreeJoin: Rule-based Binary Join Plan Modification

We next briefly cover FreeJoin by Wang *et al.* (2023), which is an approach to unify left-deep binary join operations that perform joins one relation at a time, with WCOJ algorithms that perform joins one attribute at a time. FreeJoin is motivated by the observation that WCOJ algorithms do not have a major difference from left-deep binary join (LDBJ) plans on acyclic queries. This is because on acyclic queries, WCOJ algorithms extend tuples by probing into one trie index after another, akin to using a sequence of index nested loop binary join operators. Further, a primary performance bottleneck in using WCOJ algorithms is the requirement to build indices over every relation. Therefore, one can also argue that WCOJ algorithms have a disadvantage over LDBJ plans. This is because in LDBJ plans that use hash joins, hash indices are built only on the smaller relations. The largest relation is left as the probe relation and is not indexed. In contrast, WCOJ algorithms also have an advantage over LDBJ plans. Specifically, WCOJ algorithms bind values to an attribute a_j if that value exists in all relations that

contain a_j. In contrast, LDBJ plans can assign bindings to an attribute a_j using one relation only to detect later that those binding values do not exist in another relation that contains a_j. For example, recall that in the motivating example Q_δ we used for WCOJ algorithms, any binary join plan first computes open triangles and then try to close them. As we discussed, this can result in binary join plans generating large intermediate results.

We present a running example from Wang *et al.* (2023) to demonstrate the pros and cons of both approaches on the 3-star query $Q_{3\star}(x, a, b, c) : R(x.a)$, $S(x, b)$, $T(x, c)$. Figures 3.12a and 3.12b show, respectively, the computation of a left-deep binary join plan and a WCOJ plan as in pseudocode form. In the code R[x]? indicate a lookup into relation R for a specific x value and if the value is not found, continues to the next iteration of the outer for loop. Assuming that indices over S and T are built, a LDBJ plan would read an (x, a) tuple from R and join it with tuples in S on x to generate (x, a, b)'s and then join those again on x with T tuples to produce outputs. However if a particular value v that binds to x does not appear in T, then this computation would still unnecessarily join ($x = v$, a) tuples with tuples in S. WCOJ algorithms avoid such unnecessary computations. Figure 3.12b shows the WCOJ code for this query. Instead of directly enumerating (x, a) tuples from R, the code first enumerates an x value in R, then does initial lookups in S and T for that x value, and then outputs the a's, b's, and c's from each relation. However, to do this, the algorithm needs to incur the cost of indexing the largest relation R.

```
for (x,a) in R:
  s = S[x]?
  for b in s:
    t = T[x]?
    for c in t:
      output(x, a, b, c)
```

(a) FJ binary join plan.

```
for x in R:
  r=R[x]?, s = S[x]?, t = T[x]?
  for a in r:
    for b in s:
      for c in t:
        output(x, a, b, c)
```

(b) FJ WCOJ plan.

```
for (x,a) in R:
  s = S[x]?
  t = T[x]?
  for b in s:
    for c in t:
      output(x, a, b, c)
```

(c) FJ alternative plan.

Figure 3.12: Three different FreeJoin plans for $Q_{3\star}$.

Wang *et al.* (2023) observe that other computations are possible between these two alternatives, such as the algorithm in Figure 3.12c.

In this case, R is not indexed and (x, a) tuples in R are directly enumerated. Then the x values are checked in indices over S and T and successful (x, a) tuples are further extended to full outputs. We omit the details of FreeJoin's implementation. Briefly, given a query Q, the approach generates bushy binary join-only plans P optimized by the DuckDB system (Raasveldt and Mühleisen, 2019a). Then P is broken into a sequence of left-deep plans LP_1, ..., LP_k. Each LP_i is compiled by default to a FreeJoin binary plan as shown in Figure 3.12a. Then, the code in Figure 3.12a is optimized pulling any of the index lookups to upper levels to do some early filtering of tuples that will not successfully join. In this example, this optimization would yield the plan in Figure 3.12c.

3.6 Other Work and Open Problems

In this section, we described approaches that use WCO plans for evaluating queries over static databases in shared memory single node setting. Several other works have studied applying these techniques in two other settings. Ammar *et al.* (2018) implement distributed versions of the Generic Join algorithm. These algorithms assume that there is a distributed index that is consistent with a given JAO for a query Q across a cluster of machines. In graph terms, one can think of this setting as the adjacency lists of an input graph being distributed across machines. Then these algorithms use Generic Join's multiway intersections as the core algorithmic primitive. The key optimization in this work is to avoid sending large adjacency lists that will be intersected across machines. This work demonstrate that one can achieve good performance on actual evaluations. In distributed join algorithms literature, the cost metrics that algorithms optimize include *load*, i.e., the amount of memory required per machine; *total communication*; and the *number of synchronizations* between machines. In terms of theoretical guarantees, the Generic Join-based algorithms by Ammar *et al.* (2018) have communication levels that are bounded by the AGM bound while maintaining a guaranteed load balance across machines. However, for many parallelism levels, much lower communication levels than the AGM bound can be achieved. For example, the HyperCube distributed join

algorithm (Afrati and Ullman, 2011; Beame *et al.*, 2017) can achieve $O(p^{1/3}\text{IN})$ communication with asymptotically perfect load balance, where p is the number of machines. This is asymptotically much better than the $O(\text{IN}^{3/2})$ communication for any realistic parallelism level. See the survey on distributed join algorithms by Koutris *et al.* (2018) for an extensive coverage of this literature.

Veldhuizen (2013) has described an incremental version of the Leapfrog TrieJoin algorithm that we covered in Section 3.3.1. This approach is based on keeping a *trace* of the computation, specifically low-level descriptions of the iterator positions throughout the computation. Then, upon updates to the underlying relations, this trace is updated to reflect the accurate trace of Leapfrog TrieJoin on the updated relation, in addition to producing the changes in the output. Veldhuizen (2013) proves that this approach takes time proportional to the *trace edit difference* between the traces of Leapfrog TrieJoin before and after each update.

Ammar *et al.* (2018) have proposed an incremental version of Generic Join called *Delta-GJ* and distributed versions of this incremental join maintenance algorithm. Delta-GJ is based on the idea of delta decompositions of join queries (Blakeley *et al.*, 1986), where a join query Q is decomposed into multiple delta queries, $\delta Q_1, ..., \delta Q_k$, where each δQ_i has a delta relation, which only contains the set of updates that need to be processed. This makes each δQ_i easy to process because at least one relation is very small. The Delta-GJ algorithm is based on evaluating each delta query using Generic Join. Ammar *et al.* (2018) show that under insertion-only workloads, Delta-GJ maintains any query Q in time that is asymptotically bounded by the AGM bound of Q. In other words, the total work that is done by Delta-GJ after t batch of insertions to the input relations of Q, is asymptotically bounded by the AGM bound of Q on the final, i.e., largest, versions of the input relations. In fact, this is a property also shared by the incremental Leapfrog TrieJoin algorithm by Veldhuizen (2013). However, Delta-GJ is guaranteed to require memory that is linear in the size of the inputs and deltas, while the traces maintained by Leapfrog TrieJoin can be super-linear. Finally, Mhedhbi *et al.* (2021) studied the problem of maintaining multiple queries using Delta-GJ. This work proposes optimizations to share work across multiple delta queries to minimize total computation.

Aside from giving rise to WCOJ algorithms, a separate interesting application of the AGM bound in database systems has been on cardinality estimation. Along with AGM bound, there have been several other upper bounds on query sizes that use different statistics about the database, such MOLP by Joglekar and Ré (2018) or CLLP by Abo Khamis *et al.* (2016). Since these are upper bounds on the size of queries, Cai *et al.* (2019) have shown that they can be used as a cardinality estimation technique. It is well known that systems suffer from extreme underestimation when estimating the cardinalities of sub-queries (Leis *et al.*, 2018). The use of upper bounds ensure that the estimates of sub-queries can only be over-estimates. Therefore, by design, using *pessimistic* query size upper bounds as estimation solves this underestimation problem. However, Cai *et al.* (2019) and Chen *et al.* (2022) observe that simply using these bounds is very pessimistic and leads to very inaccurate estimates. Cai *et al.* (2019) present techniques to improve pessimistic estimates that rely on AGM-like bounds and implement their technique in Postgres. Chen *et al.* (2022) have shown that the pessimistic estimator by Cai *et al.* (2019) and Graphflow's estimator, which was based on keeping statistic of small-size subgraphs/joins are in fact related. Specifically, Chen *et al.* (2022) show that these estimators are special instances of a more generic cardinality estimation technique they call *cardinality estimation graphs*. The primary difference between these estimators is that pessimistic estimators use maximum degrees of attribute values in relations while Graphflow's estimator uses average degrees in a cardinality estimation graph.

In this monograph, we model graph-structured data using the relational model, to focus on the adoption of query processing techniques within DBMSs. We note however that evaluating join queries on many-to-many relations relates to the subgraph matching problem (Sun *et al.*, 2020). Subgraph matching takes as input an edge- and vertex-labeled graph $G = (V, E)$ where V and E are the set of vertices and edges, respectively, and a query $Q(V_Q, E_Q)$ that is typically much smaller than G. It then requires enumerating the instances of Q in G, called matches, which are subgraphs of G. If we assume that each edge label maps to an Edge table, then each pair of query edges in Q sharing a query vertex represent an equi-join (possibly a self-join) between Edge

tables dependent on the query edge labels. As such, subgraph queries can be seen as the same problem of multi-way joins.[12] State-of-the-art subgraph matching techniques rely on vertex-at-a-time evaluation using adjacency list intersections, which is similar to the computation performed by WCOJ algorithms. Earlier methods used edge-at-a-time evaluation, which are similar to binary joins. The equivalence between these computations has been highlighted in several publications. Perhaps Sun *et al.* (2020) have demonstrated this equivalence best. At the same time, the techniques developed for these two separate lines of work target different workloads. Subgraph matching focuses on finding large but rarely appearing patterns on dense graphs, such as in protein interaction or chemical networks (Bhattarai *et al.*, 2019; Bi *et al.*, 2016; Han *et al.*, 2019). These patterns correspond to queries with a large number of query edges. Multiway join queries instead focus on queries with a small number of vertices on relatively sparse graphs found in social and transactional networks. In term of query processing techniques, subgraph matching algorithms first generate a candidate vertex set for each query vertex using specialized pruning methods before enumeration, often by building an index at query evaluation time. In contrast, join querying techniques rely on direct enumeration similar to the plans mentioned in this section. RapidMatch (Sun *et al.*, 2020) takes a relational approach with a simple pruning technique to bridge this gap and support both workloads.

We end this section by discussing several open problem for the field:

- Optimizing WCO plans: First, besides Graphflow's approach that estimates the size of adjacency lists when picking JAOs, there is very little work on optimizing WCOJ algorithms. This approach assumes that the relations are binary and cannot be applied for general relations. So far, techniques for picking good JAOs have been based on very simple heuristics.

[12] Some versions of subgraph matching can have special conditions, such as the nodes or edges in the matches to be unique. In vanilla equi-join queries over Edge tables, there are no such restrictions. So the equivalence we mention holds in this unrestricted version of subgraph matching.

- Index sorting bottleneck: Sorting indices is acknowledged in multiple works as a main performance bottleneck when using WCOJs in systems (Freitag *et al.*, 2020; Wang *et al.*, 2023). Therefore database cracking-like approaches of Idreos *et al.* (2007), which build parts of indices on demand can be explored to lower or amortize the cost of building indices.

- Beyond WCOJ algorithms: Finally, after the development of WCOJ algorithms, a class of *beyond WCOJ* algorithms have been developed by Ngo *et al.* (2014) and Khamis *et al.* (2016). These algorithms' complexities are measured by the size of the "proofs" they construct to ensure that the output of a join is correct. As a simple example, consider the intersection of two relations R(A) and S(A). Further suppose that A is a numeric column and R's A values end with n while S's start with $n + 1$. Therefore the output is empty and if the relations are sorted there is a very simple proof of this: n < n+1. This proof does only 1 comparison. Each correct join algorithm can be thought of as producing such implicit proofs during its execution. The work on beyond WCOJ algorithms aims to develop algorithms that are optimal in terms of the number of comparison operations they perform for specific database instances. Very interestingly, the algorithms developed so far, such as Minesweeper (Ngo *et al.*, 2014) and Tetris (Khamis *et al.*, 2016), do not operate on the input values. Instead, they operate *on the gaps between the values*. Instead of joining input tuples as traditional join algorithms do, they find large output spaces that cannot contain any tuples. Needless to say, this style of processing deviates significantly from how existing query processors work. Understanding these algorithms better and making them practical is an important area of research.

4

Factorization

Section 3 introduced the theory of worst-case optimal joins (WCOJs) and their adoption by DBMSs. WCOJs reduce the size of large intermediate results for cyclic m-n join queries by, intuitively speaking, avoiding generating intermediate results that do not satisfy cyclic join conditions. However, the standard table-at-a-time query processing may generate large intermediate results also for acyclic join queries. In some cases, this is in fact unavoidable because the final join result itself may be large, but in many cases, such intermediate results may get pruned through further joins. However, in either case, using more compact or compressed representations of the results may bring significant benefits in terms of storage costs as well as processing costs during the query evaluation.

Consider the m-n input relation $R(src, dst)$ in Figure 4.1. Consider the following queries:

i. $Q_1(a_1{=}1, a_2, a_3) \coloneqq R(a_1{=}1, a_2),\ R(a_1{=}1, a_3)$; and

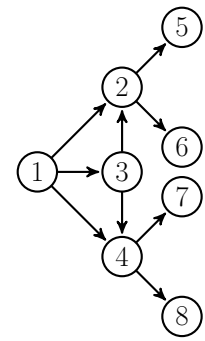
ii. $Q_2(a_1, a_2, a_3) \coloneqq R(a_1, a_2), R(a_2, a_3)$.

Note that Q_1 and Q_2 assign different variables to *src* and *dst* columns of R. We will use these two queries and their variants throughout this

$R(src, dst)$

src	*dst*
1	2
1	3
1	4
2	5
2	6
3	2
3	4
4	7
4	8

(a) Tabular format.



(b) Graph format.

Figure 4.1: Input relation $R(src, dst)$ in tabular and graph format and the output of $Q_1(a_1{=}1, a_2, a_3) := R(a_1{=}1, a_2), R(a_1{=}1, a_3)$ as a factorized representation.

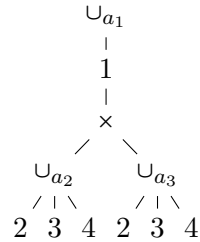
section. Figure 4.2 shows the output relations of Q_1 and Q_2 following a flat tuple representation when evaluated over $R(src, dst)$. Q_1's output relation has 11 tuples for a total of 33 "data values", where each column value is counted as one atomic unit, while Q_2's has 12 tuples for a total of 36 data values. Both relations however have many value repetitions.

A possible and more compact representation, called *factorized* representation, of Q_1's output is as follows: $a_1 : \{1\} \times a_2 : \{2, 3, 4\} \times a_3 : \{2, 3, 4\}$, which contains only 7 data values. Figure 4.2 shows this representation titled "Q_1's factorized output". This is an example of avoiding Cartesian products as the a_2 and a_3 values are *conditionally independent* given a_1. In other words, once the a_1 variable is fixed to a value, the sets of a_2 and a_3 values in the output (corresponding to that a_1 value) are independent of each other. Equivalently, the *multi-valued dependency* (MVD) (Fagin, 1977) $a_1 \twoheadrightarrow a_2$ holds in the output relation (equivalently $a_1 \twoheadrightarrow a_3$ also holds). Similar to how 4th normal form decomposition uses MVDs to decompose and compress base database relations in a lossless way, factorization uses them to compress intermediate or output query relations. In fact, this is an important distinction of factorization from standard applications of compression techniques in literature. Compression in DBMSs is generally a technique that is primarily integrated into storage

$Q_1(a_1{=}1, a_2, a_3)$

a_1	a_2	a_3
1	2	2
1	2	3
1	2	4
1	3	2
1	3	3
1	3	4
1	4	2
1	4	3
1	4	4

Q_1's factorized output



$Q_2(a_1, a_2, a_3)$

a_1	a_2	a_3
1	2	5
1	2	6
1	3	2
1	3	4
1	4	7
1	4	8
3	2	5
3	2	6
3	4	7
3	4	8

Figure 4.2: Output of queries $Q_1(a_1{=}1, a_2, a_3) \coloneqq R(a_1{=}1, a_2), R(a_1{=}1, a_3)$ and $Q_2(a_1, a_2, a_3) \coloneqq R(a_1, a_2), R(a_1, a_3)$ evaluated on $R(src, dst)$ in Figure 4.1.

managers to compress base relations. Instead factorization is a technique for the query processors to compress intermediate relations that are generated during query execution. This is the core of the representation system called *f-representations* (Section 4.2).

The above is an example of a **data-independent** compact representation; given the schemas of the relations and the joins in the query, the above factorization would apply to any relation instances. We devote the bulk of this section to discussing such representations. These representations are proposed in the theory of *factorization* (Olteanu and Zavodny, 2015; Olteanu and Schleich, 2016). This theory shows that intermediate results of acyclic m-n join queries can be highly compressible as they typically contain MVDs between the attributes. Further, the theory shows how to determine these dependencies statically during query compilation time (see the "path constraint" in Section 4.2.1).

We also briefly discuss a **data-dependent** approach, that analyzes the specific relation instances to identify compression opportunities (which in turn is closely related to the work on *graph compression*), and can be applied even if there are no MVDs in the result to exploit, which is often the case when there are projections in queries. However, the computational cost of the compression is significant in such approaches, and can only be justified if the resulting compressed result is repeatedly used or stored.

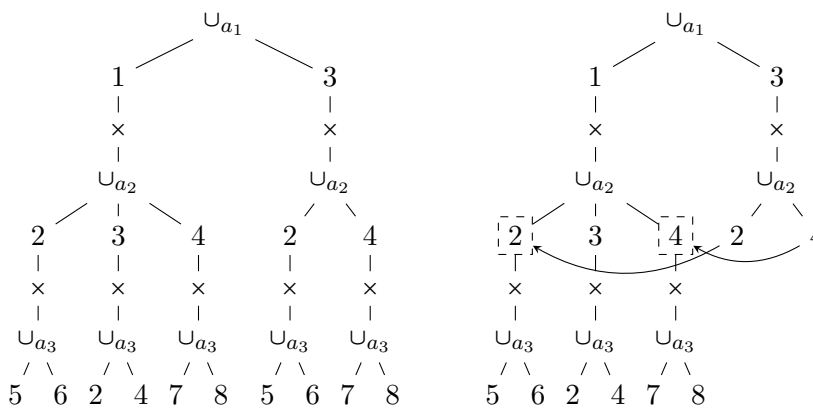
A natural question here is how this relates to the classical Yannakakis algorithm (Yannakakis, 1981) that provides guarantees on the overall query processing time in terms of the total input and output sizes (thus avoiding the generation of intermediate results larger than the final output size). First, we note that when the final output size itself is very large, the Yannakakis algorithm does not avoid the generation of large intermediate results. In contrast, factorized representations will work even in that case, and in many cases, it is possible to use the compressed representation itself for the next processing step. Second, the key to the Yannakakis algorithm is its use of two semijoin passes to remove "dangling" tuples, which enables it to avoid generation of redundant intermediate tuples. However, in practice, the overheads of those two passes are significant and unlikely to pay off; in fact, we are not aware of any practical implementation of the algorithm in a commercially used system. In contrast, factorized representations can be used in a single pass, and although they do not provide the same guarantee, in practice, they can be used to handle many of the worst-case scenarios that the Yannakakis algorithm is designed to handle. Lookahead-based approaches (e.g., Zhu *et al.*, 2017) provide a promising and practical in-between alternative, that in essence approximates one of the passes of the Yannakakis algorithm through use of efficient semi-join implementations (e.g., using bloom filters).

In this section, we cover the f- and d-representations proposed by Olteanu and Schleich (2016), and system implementations of those representations in `FDB`, `Graphflow`, and Kùzu. We use the term "factorization" to specifically refer to this line of work, and the term "compression" to refer to general techniques that reduce the size of intermediate results. We begin with the foundations of factorization theory in Section 4.1. In Section 4.3, we cover two approaches that adopt f-representations: (i) `FDB` system's approach based on materialized tries; and (ii) `Graphflow` and Kùzu's approach of factorized vectors. We also cover how `Graphflow` adopts d-representations in Section 4.5.

4.1 Overview of Factorization

Theory of factorization proposes two relation representation schemes called *f- and d-representations*. Both of these schemes represent relations as tries and can lead to significant space savings over representing relations as flat tuples, which is the de facto way to represent and store relations in systems. This is done through two techniques: 1) avoiding Cartesian products across sets of values; and 2) caching and reusing subquery results.

We already saw an example of the former. Next, we demonstrate the reuse of subquery results for further compression. A possible factorization of Q_2's output that avoids Cartesian products is shown as a trie in Figure 4.3a. This trie has 17 data values instead of 36 in the flat representation. The trie in the figure can be read as follows. At a node $\cup_y$, we take the union of the tuples of the subtrees under the union symbol. y in $\cup_y$ is the variable of the values of the immediate children of $\cup_y$. At a node $\times$, we take the Cartesian product of each branch of $\times$ with the parent value of $\times$ (if exists).



(a) F_1. **(b)** F_2.

Figure 4.3: Output of $Q_2(a_1, a_2, a_3) := R(a_1, a_2), R(a_1, a_3)$ evaluated on $R(src, dst)$ in Figure 4.2 with two different factorizations F_1 and F_2.

For example, the left subtree under the root of Figure 4.3a corresponds to the following expression:

$$(a_2 = \{2\} \bigtimes a_3 = \{5, 6\}) \cup (a_2 = \{3\} \bigtimes a_3 = \{2, 4\}) \cup (a_2 = \{4\} \bigtimes a_3 = \{7, 8\})$$

Combined with the $a_1 = \{1\}$ at the root, in flat representation, this corresponds to six (a_1, a_2, a_3) tuples:

$$\{(1,2,5),(1,2,6),(1,3,2),(1,3,4),(1,4,7),(1,4,8)\}$$

The use of $\cup$ and $\times$ as in this example is the core of f-representations.

An even more compact representation is possible if we reuse the subquery results as shown in Figure 4.3b. These same reusable subquery results are highlighted in blue and red boxes in Figure 4.2. In the figure, the two inner nodes $(a_2 : 2)$ and $(a_2 : 4)$ that are under $(a_1 : 3)$ reuse the subtrees of $(a_2 : 2)$ and $(a_2 : 4)$ under $(a_1 : 1)$ by pointing to them. In this case, the factorized representation has instead 13 data values. This is the core of the representation system called *d-representations* (Section 4.4), which extends f-representations with reused nested relations.

In addition to the two representations above, the theory of factorization offers two core insights to developers of DBMSs to efficiently evaluate acyclic m-n join queries:

1. Process relations using factorized representations at the physical layer during query evaluation.
2. Use tries as the backing data structure for factorized representations.

4.2 F-Representations Background

Next, we introduce the formalism of f-representations following (Olteanu and Zavodny, 2015), and introduce factorized trees (f-trees), used to describe the algebraic factorization over the query variables, describing in turn the structure of an f-representation. We also discuss the size bounds of f-representations on query results.

Consider the input relation R shown in Figure 4.4. Consider further the 4-hop query $Q_{4H} := R(a_1, a_2),\ R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$ as a running example. To have a consistent formalism across flat and factorized representations,we will represent tuples as Cartesian products. DBMSs use flat representations and as such Q_{4H}'s output evaluated on R contains k^3 tuples of the form $(v_{1_i} \times v_2 \times v_{3_j} \times v_4 \times v_{5_\ell})$, where i, j, and ℓ are $\in [1, k]$. If we count the size of relations as the number of data values, since each tuple contains 5 values, the size of this representation is $5k^3$.

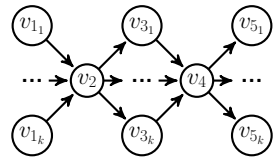


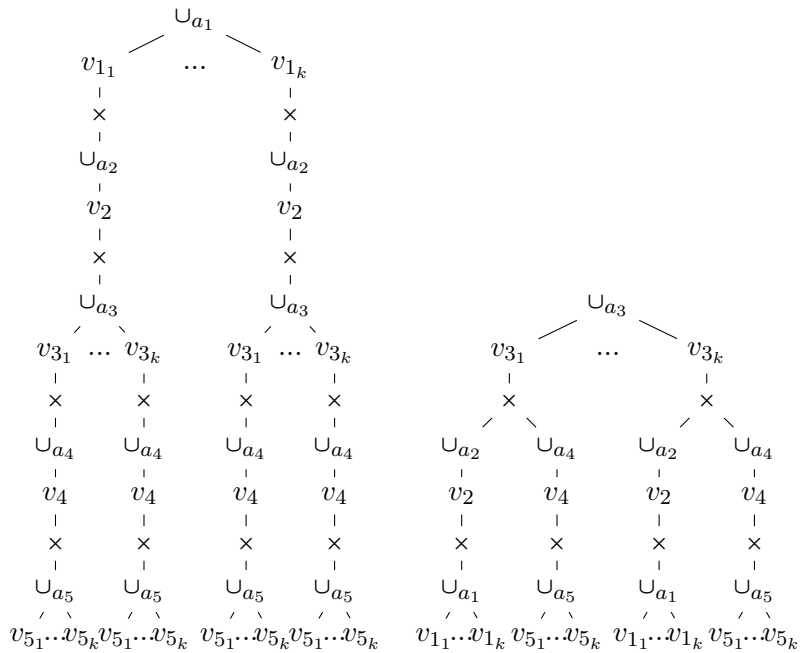
Figure 4.4: Another instance of relation $R(src, dst)$.

Factorized representations employ algebraic factorization to nest Cartesian products and unions and compress results. Consider the possible factorized representations (f–representations) in Figures 4.5a and 4.5b. Factorizations can be thought of as groupings. F_1 in Figure 4.5a has in its root a_1 values grouping a_2's, which group a_3's, which group a_4's, which group a_5's. F_2 in Figure 4.5b, however, has a different structure. Each a_3 value groups separately a_2 and a_4 values. a_2 values group a_1's and a_4 values separately group a_5's.

Given an arbitrary relation $R(a, b, ...)$, finding the most compact f–representation of R is NP-hard (Olteanu and Zavodny, 2015). However, if R is the result of a query, then the structure of the query can give rise to MVDs between the attributes in the result. In other words, the structure of the query can lead to attribute-level conditional independence relationships that can be exploited to obtain compact f–representations. For example, in Q_{4H} for any fixed value of a_3, we can infer from the query that the sets of $\{a_1, a_2\}$ and $\{a_4, a_5\}$ values are independent. Equivalently the $a_3 \twoheadrightarrow \{a_1, a_2\}$ holds in R. Therefore in the query results, $\{a_1, a_2\}$ and $\{a_4, a_5\}$ are independent, conditioned on a_3. This is how the f–representation in Figure 4.5b above was obtained. This conditional independence is described using *factorized trees*, which we introduce next.

4.2.1 Factorized Trees

An f-tree $\mathcal{T}$ is a formalism to describe the structure of an f-representation F. Formally, an f–tree $\mathcal{T}$ is a rooted tree such that each node n_i of $\mathcal{T}$ is labelled by a query attribute. The union of all query attributes labelling the nodes in $\mathcal{T}$ is the set of all query attributes. The shape of $\mathcal{T}$ provides a hierarchy of attributes by which we group the tuples of the represented relation. Let R be the equivalent flat representation of

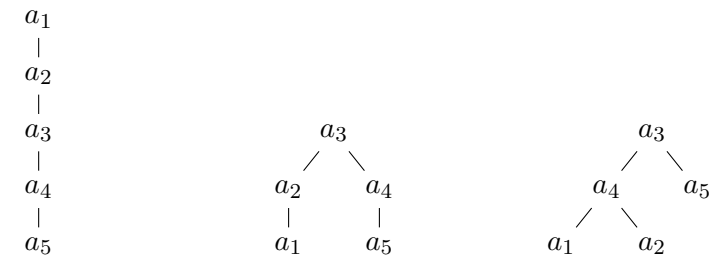


(a) F_1. **(b)** F_2.

Figure 4.5: Output of $Q_{4H} := R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$ as f-representations F_1 and F_2 following $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. T_3 is an example f-tree that cannot be used to factorize the output of Q_{4H}.

the relation that F represents. This grouping process can be thought of as follows. We group the tuples in R by the values of the attributes labelling the $\mathcal{T}$'s root, say a. This forms a set of groups, say $G_{a=1}$, $G_{a=2}$, ..., $G_{a=g}$. Suppose that for $\mathcal{T}$, a has k children labeled $c_1, ..., c_k$. Then for each c_i, child of a, and for each group, say $G_{a=j}$, we project the tuples in $G_{a=j}$ to the attributes in the subtree rooted at c_i. When constructing the original relation, we take Cartesian product of each projection to re-construct each $G_{a=j}$ and unions across all groups $G_{a=1}$, ..., $G_{a=g}$ to re-construct R. This process iteratively continues in each child of the root.

As an example, the f–trees $\mathcal{T}_1$ and $\mathcal{T}_2$ in Figures 4.6a and 4.6b describe the structure of the f-representations in Figures 4.5a and 4.5b, respectively. Not every f-tree can be used to factorize the output of a query Q. We call an f-tree that can factorize Q a *valid f-tree for* Q.



(a) $\mathcal{T}_1$. **(b)** $\mathcal{T}_2$. **(c)** $\mathcal{T}_3$.

Figure 4.6: Output of $Q_{4H} := R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$ as f-representations F_1 and F_2 following $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. T_3 is an example f-tree that cannot be used to factorize the output of Q_{4H}.

The structure of an f-tree represents a set of conditional independence relationships between the attributes and these relationships need to hold in the output of Q. Consider a node a_i of an f-tree with ℓ children $c_{i1}, ..., c_{i\ell}$. In a corresponding f-representation, conditioned on a_i, we take Cartesian products of ℓ+1 sets: (i) the tuples represented by each children of a_i, which form ℓ many sets; and (ii) one fixed set of bindings to the ancestors of a_i, which forms a set with a single tuple.

For example, the f-tree $\mathcal{T}_3$ in Figure 4.6c is not a valid f-tree to factorize the output of Q_{4H} under all database instances. This is because, given a_3 and a_4, a_1 and a_2 are not independent. To see this, consider the modified input graph shown in Figure 4.7. In this graph, one can see that the output contains $(v_{1_2}, v_{2_1}, v_{3_1}, v_4, v_{5_1})$ and $(v_{1_1}, v_{2_2}, v_{3_1}, v_4, v_{5_1})$. These two tuples have the same a_3 and a_4 values. However, the output does not contain a tuple where a_1=v_{1_2} and a_2=v_{2_2} for the same a_3 and a_4 values, i.e., the tuple $(v_{1_2}, v_{2_2}, v_{3_1}, v_4, v_{5_1})$, which would have to be included if a_1 and a_2 were conditionally independent given a_3 and a_4.

There is a simple condition that can be used to decide whether or not an f-tree can be used to factorize the outputs of Q. To explain the condition we first make the notion of dependence formal:[1]

[1] Olteanu and Zavodny (2015) have a different definition defined not for queries but relations first. From this initial definition, the two conditions we used to define dependence are deduced as properties of their definition. We simplify the formalism here to focus on the use of the notion of dependence for relations that are outputs of queries.

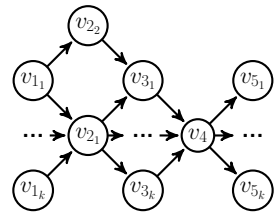


Figure 4.7: Modified Relation $R(src, dst)$ in Figure 4.4 with an extra node v_{2_2} added to the relation in Figure 4.4.

Definition 4.1. (Dependence)[2] Two attributes a and b are dependent if they are in the same relation, or if there is a chain of relations $R_1, ..., R_k$ in Q such that a is in the schema of R_1 and b is in the schema of R_k, and for all pairs of relations R_i and R_{i+1} in the chain, R_i and R_{i+1} are joined on an attribute that is projected out in Q.

We can also define a notion of conditional independence between sets of attributes as follows: two disjoint sets of attributes $\mathcal{A}$ and $\mathcal{B}$ are conditionally independent on the set of attributes $\mathcal{C}$ in a relation R where $\mathcal{C}$ denotes the rest of the attributes in R (in this case, the output relation of Q), if $R = \pi_{\mathcal{A},\mathcal{C}}(R) \bowtie_{\mathcal{C}} \pi_{\mathcal{B},\mathcal{C}}(R)$.

Note that if $\mathcal{A}$ and $\mathcal{B}$ are conditionally independent, then $C \twoheadrightarrow \mathcal{A}|\mathcal{B}$, therefore we can factor out $\mathcal{A}$ and $\mathcal{B}$ conditioned on $\mathcal{C}$ in an f-representation. We will not need to use this direct definition of conditional independence to define valid f-trees. Instead, we will use a property of f-trees called the "path constraint", which a simple and practical condition that can be used to check if an f-tree is valid for Q, i.e., it captures a set of correct conditional dependencies or MVDs in the final output.

Theorem 4.1. (Path constraint) Given a query Q and an f–tree $\mathcal{T}$, $\mathcal{T}$ is a valid factorization tree of Q if any two dependent attributes are on the same root-to-leaf path.

Proof of this theorem is provided by Olteanu and Zavodny (2015). Readers can verify that the f-trees in Figures 4.6a and 4.6b satisfy the

[2]The notion of dependence we introduce here follows the definition of Q-dependence by Olteanu and Zavodny (2015).

path constraint. In contrast, the f-tree in Figure 4.6c does not satisfy the path constraint because a_1 and a_2 are dependent (they appear in R_1) and appear on different root-to-leaf paths.

4.2.2 F-Trees and Multi-valued Dependencies

We next discuss the connection between f-trees and MVDs in more detail. We first begin by reviewing the definition of an MVD (Fagin, 1977). An MVD $X \twoheadrightarrow Y$ holds in a relation R, where X and Y are attributes of R, if the following condition holds. Let Z be $attr(R) - X - Y$, i.e., Z are the "rest" of the attributes in R. Given two tuples t_1 and t_2, if $t_1[X] = t_2[X]$ and $t_1[Y] \neq t_2[Y]$, then there are 2 further tuples t_3 and t_4 that take the two other combinations of t_1 and t_2's Y and Z values. That is t_3 is of the form: $t_1[X]{\cdot}t_1[Y]{\cdot}t_2[Z]$, and t_4 is of the form: $t_1[X]{\cdot}t_2[Y]{\cdot}t_1[Z]$, where we are using $\cdot$ as the concatenation operator. In other words, once a particular X values are fixed, Y and Z can be "factored out" and are independent of each other. 4NF decompositions are indeed obtained by using such factorizations.

Our first observation is that each f-tree represents a set of MVDs in the output of Q. Specifically, at each node a_h in an f-tree $\mathcal{T}$, there is an MVD of the following form.

Lemma 4.2. Let an f-tree $\mathcal{T}$ be a valid f-tree for a relation R. Let $\text{anc}(a_h)$ be the ancestors of a_h including a_h and let c be a child of a_h. Let $\text{des}(c)$ be the set of attributes in the subtree rooted in c, including c. Then, the MVD $\text{anc}(a_h) \twoheadrightarrow \text{des}(c)$ holds in R.

Proof. Let $a_1, ..., a_m$ be the attributes in $\mathcal{T}$. Let a_h be an arbitrary node in $\mathcal{T}$. Without loss of generality, assume that a_1 is the root of $\mathcal{T}$ and the ancestors of a_h are a_1, a_2, ..., a_h. Consider again without loss of generality that $\mathcal{T}$ drawn so that this path is the left most path as shown in Figure 4.8. That is, each a_i is the left most child of its parent. Let $\text{des}'(a_{i+1})$ be the union of all subtrees that are rooted in the siblings of a_{i+1}. Figure 4.8 shows $\text{des}(a_{i+1})$ and $\text{des}'(a_{i+1})$ pictorially. We can prove the lemma using the definition of MVDs that we reviewed above.

To prove that $\text{anc}(a_h) \twoheadrightarrow \text{des}(c)$, observe X=$\text{anc}(a_h)$, Y=$\text{des}(c)$, and $Z = \cup_{i=1...,h}\text{des}'(a_i)$. Let's represent any tuple t as follows: $t[a_1...a_h] \cdot$

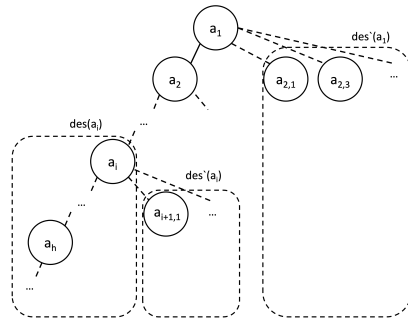


Figure 4.8: Example f-tree demonstrating the definition of $desc(a_i)$.

$t[\text{des}'(a_1)] \cdot \ldots \cdot t[\text{des}'(a_{h-1})] \cdot t[\text{des}'(a_h)] \cdot t[\text{des}(a_h)]$. Now consider two tuples:

$$t_1[a_1...,a_{h-1}] \cdot t_1[\text{des}'(a_1)] \cdot \ldots \cdot t_1[\text{des}'(a_{h-1})] \cdot t_1[\text{des}'(a_h)] \cdot t_1[\text{des}(a_h)]$$

$$t_2[a_1...,a_{h-1}] \cdot t_2[\text{des}'(a_1)] \cdot \ldots \cdot t_2[\text{des}'(a_{h-1})] \cdot t_2[\text{des}'(a_h)] \cdot t_2[\text{des}(a_h)]$$

Assume that $t_1[a_1...,a_h]$=$t_2[a_1...,a_h]$ but $t_1[\text{des}(a_h)]\neq t_2[\text{des}(a_h)]$. Then notice that, by the structure of the f-tree $\mathcal{T}$, $\text{des}(a_h)$ is independent of $\text{des}'(a_h)$ conditioned on $a_1,\ldots a_{h-1}$. Therefore, we can infer that two tuples, say t'_3, and t'_4, of the following form exist:

$$t_1[a_1...,a_{h-1}] \cdot t_1[\text{des}'(a_1)] \cdot \ldots \cdot t_1[\text{des}'(a_{h-1})] \cdot t_2[\text{des}'(a_h)] \cdot t_1[\text{des}(a_h)]$$

$$t_2[a_1...,a_{h-1}] \cdot t_2[\text{des}'(a_1)] \cdot \ldots \cdot t_2[\text{des}'(a_{h-1})] \cdot t_1[\text{des}'(a_h)] \cdot t_2[\text{des}(a_h)]$$

Note that $\text{des}(a_{h-1})$=$\text{des}(a_h) \cup \text{des}'(a_h)$. Therefore, we now have two tuples that agree on $a_1,...,a_{h-1}$ and differ in $\text{des}(a_{h-1})$ attributes (because we assumed $t_1[\text{des}(a_h)]\neq t_2[\text{des}(a_h)]$). We can therefore infer by the structure of $\mathcal{T}$ that the following t''_3 and t''_4 tuples exist:

$$t_1[a_1...,a_h] \cdot t_1[\text{des}'(a_1)] \cdot \ldots \cdot t_2[\text{des}'(a_{h-1})] \cdot t_2[\text{des}'(a_h)] \cdot t_1[\text{des}(a_h)]$$

$$t_2[a_1...,a_h] \cdot t_2[\text{des}'(a_1)] \cdot \ldots \cdot t_1[\text{des}'(a_{h-1})] \cdot t_1[\text{des}'(a_h)] \cdot t_2[\text{des}(a_h)]$$

Repeating the same argument iteratively, we can infer that the following two tuples t_3 and t_4 exist, completing the proof:

$$t_1[a_1...,a_h] \cdot t_2[\text{des}'(a_1)] \cdot \ldots \cdot t_2[\text{des}'(a_{h-1})] \cdot t_2[\text{des}'(a_h)] \cdot t_1[\text{des}(a_h)]$$

$$t_2[a_1...,a_h] \cdot t_1[\text{des}'(a_1)] \cdot \ldots \cdot t_1[\text{des}'(a_{h-1})] \cdot t_1[\text{des}'(a_h)] \cdot t_2[\text{des}(a_h)]$$

□

For example, consider the root of $\mathcal{T}_2$ in Figure 4.6b. By Lemma 4.2, we can write $a_3 \twoheadrightarrow \{a_2,a_1\}$ as well as $a_3 \twoheadrightarrow \{a_4, a_5\}$. Alternatively, we can use the more compact notation for MVDs and write these as $a_3 \twoheadrightarrow \{a_2, a_1\}|\{a_4, a_5\}$. Corresponding to the two children of a_1, we also have MVDs: $a_1, a_2 \twoheadrightarrow a_3$ and $a_1, a_4 \twoheadrightarrow a_5$.

Similarly, consider a query for which $\mathcal{T}_3$ in Figure 4.6c is valid. Recall that $\mathcal{T}_3$ is not valid for Q_{4H} but it would be valid in a query Q_{T_3} that looks exactly in the shape of $\mathcal{T}_3$, i.e., $Q_{T_3} \coloneqq R(a_3, a_4)$, $R(a_3, a_5)$, $R(a_4, a_1)$, $R(a_4, a_2)$. Then by Lemma 4.2, the following MVDs holds in the output $a_3 \twoheadrightarrow \{a_4, a_1, a_2\}$; $\{a_3, a_4\} \twoheadrightarrow a_1$; or $\{a_3, a_4\} \twoheadrightarrow a_2$. Note however that each f-tree captures only a subset of the MVDs that exist in the output of the query. For example, $\mathcal{T}$ does not capture $a_3 \twoheadrightarrow \{a_2,a_1\}$, which we showed is captured by $\mathcal{T}_2$.

In fact, as a corollary, an f-tree as a whole represents *a full first order hierarchical decomposition (FOHD)* (Delobel, 1978) of the form:

$$anc(a_h) : des(c_1)|des(c_2)|...|des(c_k)$$

FOHDs were introduced as a generalization of MVDs to more properly represent *embedded* MVDs and to capture a hierarchical structure akin to f-trees.

A note on projected attributes: If the query contains projections, the standard rule for projections and MVDs can be used to find the set of MVDs that hold on the result. Specifically, if $X \twoheadrightarrow Y|Z$ holds on a relation such that $X \cap Y = X \cap Z = Y \cap Z = \phi$, and if we project out attributes from Y and Z to get Y' and Z' respectively; then the MVD $X \twoheadrightarrow Y'|Z'$ holds on the result relation. Note that, if any attribute from X is projected out, then this MVD must be thrown away. As an example, in the join of $R(A, B)$ and $S(B, C)$, we have that the MVD $B \twoheadrightarrow A|C$ holds, but if B is projected out, then there is no non-trivial MVD on the result relation. The path constraint captures this case for f-representation. That is, If B is projected out from the output of $R(A, B) \bowtie S(B, C)$, then according to the path constraint A and B become dependent, so there is no f-tree that can factor out the set of A's from set of B's.

4.2.3 Worst-case Size Bounds for F-representations

In general, a query Q may be factorized with different f-trees, which will typically result in different output sizes. For example, $\mathcal{T}_1$ and $\mathcal{T}_2$ (Figures 4.6a and 4.6b) are two different f-trees for Q_{4H}, and $\mathcal{T}_2$ leads to a more compact f-representation than $\mathcal{T}_1$ on Q_{4H} on the $R(src, dst)$ relation from Figure 4.1. In fact the groupings done by $\mathcal{T}_1$ are effectively the same as the flat representation. $\mathcal{T}_1$'s corresponding representation F_1 is shown in Figure 4.5a. Readers can verify that F_1 contains $\Theta(k^3)$ many values (k^3+2k+2 to be exact). The f-representation corresponding to $\mathcal{T}_2$ is shown Figure 4.5b as F_2. As readers can verify, F_2 has a more succinct representation with an asymptotic size of $\Theta(k^2)$ ($2k^2+3k$ to be exact). We next discuss the worst-case bounds for f-representations of queries akin to the AGM bound.

For simplicity, we assume all queries contain self-joins and are hence on the same relation R with N tuples. Recall from Section 3 that the AGM bound of a query Q, denoted by $N^{\rho^*(Q)}$ (ignoring asymptotic notations), is the worst-case output size for a given query on database instances with N tuples. $N^{\rho^*(Q)}$ is effectively the worst-case size for flat representations. The theory of factorization introduces a similar notation, $N^{s(Q)}$, to refer to the minimal worst-case output sizes for f–representations across all possible valid f-trees for Q.

Definition 4.2. (Minimal worst-case output sizes of f–representations) Consider Q and an arbitrary valid f–tree $\mathcal{T}$. Let $N^{s(Q)_\mathcal{T}}$ be the worst-case size of the output of Q as an f-representation in the structure of $\mathcal{T}$ across all possible database instances where each relation in Q contains N tuples. Then $N^{s(Q)}$ is the minimum such worst-case size under the "best" valid f-tree for Q, i.e., $s(Q) = \min_\mathcal{T} s(Q)_\mathcal{T}$ where the minimum is taken over all valid f–trees for Q.

At a high-level, f-trees with the smallest $s(T)$ value lead to producing the most compressed worst-case f-representations. Such "optimal" f-trees are those that have the shortest root-to-leaf paths and branch out the most.

Since $N^{\rho^*(Q)}$ is the size of flat representations, it has the same asymptoic size of f–representations corresponding to f–trees that are

paths, i.e., each node has at most a single child (e.g., $\mathcal{T}_1$ from Figure 4.6a). As such, by definition, the worst-case f–representation of the output of Q is at most the AGM bound of Q. Further, for some queries it is strictly smaller. Therefore, $s(Q) \leq \rho^*(Q)$ for any query Q. For example, consider the 4-hop query Q_{4H} on an input edge relation R of size N. The AGM bound of Q_{4H} is $\rho^*(Q_{4H}) = \Theta(N^4)$. Meanwhile the worst-case output size of the f–representation that uses $\mathcal{T}_2$ in Figure 4.6b is $\Theta(N^2)$. In fact $\mathcal{T}_2$ is the optimal f-tree, so for Q_{4H}, $\rho^*(Q) = 4$ while $s(Q) = 2$.

4.3 Approaches to Adopting F-Representations

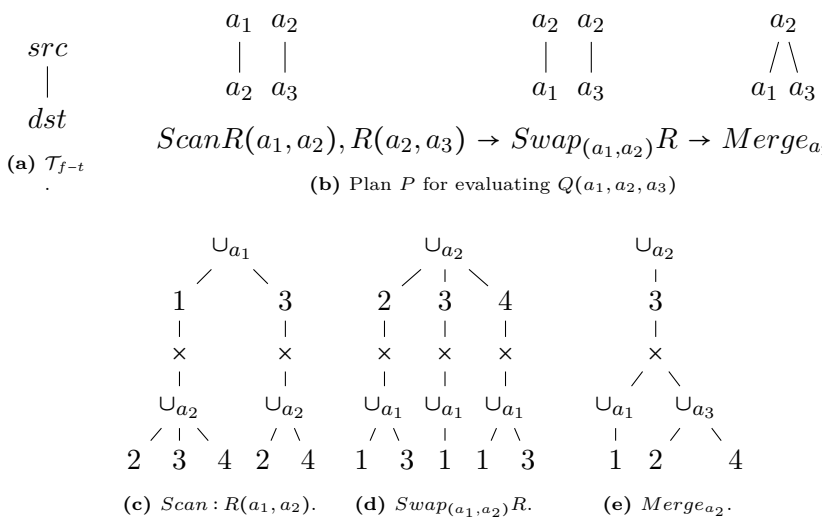
The two approaches that adopt f-representations in literature are based either on materialization or pipelined vectorized execution. Specifically, in the `FDB` system by Bakibayev *et al.* (2012), operators produce intermediate f-representations of relations as fully materialized tries. The `Graphflow` (Gupta *et al.*, 2021) and Kùzu (Feng *et al.*, 2023) systems pipeline f-representations in chunks of *factorized vectors*.

4.3.1 FDB: Fully Materialized Relations as Tries

`FDB` is an in-memory query engine capable of evaluating select-project-join and aggregation queries. It is the first engine to rely on factorized representations to improve query performance. It is capable of storing factorized base relations. The core query processing approach of FDB is based on query plans that consist of operators that take an input f-representation following an f-tree $\mathcal{T}$ and produce another f-representation following $\mathcal{T}'$. Each primitive operator in the system performs $\mathcal{T} \rightarrow \mathcal{T}'$ mapping. In the remainder of this section, we describe FDB's approach through an example plan and cover several of its core operators: *swap* and *merge*, along with standard operators, such as *scan*. Several other variants of the operators we cover, e.g., *absorb*, can be found in the original FDB publication (Bakibayev *et al.*, 2012). Bakibayev *et al.* (2013) also describe further optimizations to perform aggregations and ordering.

Example FDB Plan and FDB Operators

FDB stores base relations as f-representations. Consider the base relation $R(src, dst)$ and let us suppose that $R(src, dst)$'s raw storage follows the f-tree $\mathcal{T}_{f-t}$ shown in Figure 4.9a. We can think of this as equivalent to a forward adjacency list index that holds for each source node v in the graph a list of destination nodes. Consider further the FDB plan in Figure 4.10a evaluating $Q := R(a_1, a_2), R(a_2, a_3)$. The plan is made of three operators:



(a) $\mathcal{T}_{f-t}$.

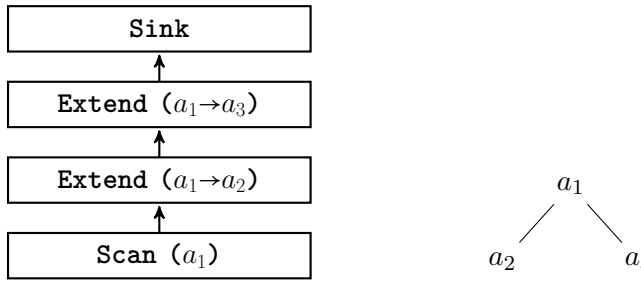
(b) Plan P for evaluating $Q(a_1, a_2, a_3)$

(c) $Scan : R(a_1, a_2)$. **(d)** $Swap_{(a_1,a_2)}R$. **(e)** $Merge_{a_2}$.

Figure 4.9: An Example evaluation of $Q_2(a_1, a_2, a_3) := R(a_1, a_2), R(a_1, a_3)$ on $R(src, dst)$ from Figure 4.2. $R(src, dst)$ is stored following the f–tree $\mathcal{T}_{a_1-a_2}$; (c) shows that representation as the output of the Scan operator. (d) shows the result of the Swap operation, and (e) shows the joined result for $a_2 = 3$.

1. Scan scans base relations as f-representations. Q contains 2 relations so for simplicity we show one scan operator scanning both relations and producing two f-representations. Specifically, scan scans $R(a_1, a_2)$ and $R(a_2, a_3)$ as f-representations following the f-tree $\mathcal{T}_{f-t}$.

2. Swap is an operator that restructures an f-representation. Here it is used to align the two f-representations from scan so that they can be joined. As we will see next, the join of $R(a_1, a_2)$ and $R(a_2, a_3)$ on



(a) Plan evaluating Q_1. **(b)** $\mathcal{T}$.

Figure 4.10: Plan evaluating $Q_2(a_1, a_2, a_3) := R_1(a_1, a_2)$, $R_2(a_1, a_3)$ on $R(src, dst)$ in Figure 4.2 with a possible factorization for the output per f-tree $\mathcal{T}$.

a_2 is performed by "merging" the f-representations on the a_2 values. This is only possible by making one of the f-representations the child of a_2 in the other (assuming this operation does not violate the path constraint). For example, in their current form after scan, one can put the $R(a_2, a_3)$'s f-representation directly under the a_2 nodes in the $R(a_1, a_2)$ f-representation. This would produce an intermediate relation $I(a_1, a_2, a_3)$ that follows the linear f-tree of $a_1 \rightarrow a_2 \rightarrow a_3$ (drawn left-to-right), which is equivalent to a flat representation. The plan in Figure 4.10a instead produces another f-tree with a_2 as the root. To produce this f-tree, both of the relations need to be in f-representations with a_2 as root. To achieve this, we need to restructure $R(a_1, a_2)$'s f-representation and pull-up the a_2 values over their a_1 values, i.e., instead of grouping the tuples by a_1, we need to first group them by a_2. Swap performs this operation.

3. Merge is similar to the standard merge-join operator, and joins $R(a_1, a_2)$ and $R(a_2, a_3)$ on their a_2 values. Now that both relations are in f-representations grouped by a_2, this involves performing a merge join on the a_2 values and for each joining a_{2_i} value, the operator puts the a_1 values corresponding to a_{2_i} from $R(a_1, a_2)$ and a_3 values corresponding to a_{2_i} from $R(a_2, a_3)$ as two children of a_{2_i} (with appropriate U nodes) in a new f-representation.

One property of the FDB operators is that any operator that restructures an existing f-representations, such as the swap operator, takes quasilinear time (see Bakibayev *et al.*, 2012). Merge operations, which

perform joins, can still increase the size of the relations depending on whether the join factorizes or can repeat certain branches.

FDB Query Optimizer

The plan introduced above was merely one possible plan. Many other plans are possible and each can be identified by the f-trees that the f-representations follow. FDB's optimizer aims to find a plan such that the maximal cost of the sequence of transformations minimizes the size of the largest produced f-representation. More specifically, the cost measure picked is the maximum integer value $s(\mathcal{T}_i)$ across all intermediate f-trees $\mathcal{T}_i$. Note that the cost measure in this case does not take into account the size of the output f-representation. FDB uses the asymptotic bounds for the size and not on actual cardinality estimates. FDB uses two approaches when enumerating plans, exhaustive search or greedy search heuristics to minimize the size of the explored space. We refer readers to Bakibayev *et al.* (2012) for details of FDB's optimizer.

4.3.2 Factorized Vector Execution by Graphflow and Kùzu

One shortcoming of FDB's approach is that it deviates from several common wisdom principles for developing performant query processors. Specifically, a common wisdom in analytical DBMSs, which are optimized for performing large reads and joins (though not m-n joins), is to use a vectorized and pipelined query execution model. MonetDB (Boncz *et al.*, 2005), VectorWise (Zukowski *et al.*, 2012), and DuckDB (Raasveldt and Mühleisen, 2019b) are examples of systems that adopt this query processing model. In this model, operators operate on vectors of tuples, e.g., vectors of 1024 tuples each, that are pipelined across operators. This contrasts with prior models where operators process and pass one tuple at-a-time in what is often called Volcano-style procesing (Graefe, 1994). In vectorized execution, all primitive operations in the system are written as for loops over vectors, which is well understood to have important performance advantages on modern CPUs.

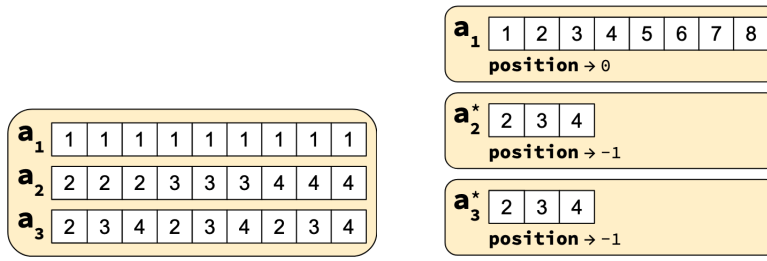
Traditional vectorized execution assumes that the operators are processing a set of flat vectors. The motivation for *factorized vector execution* model of Graphflow and Kùzu is to extend the vectorized

execution model to also benefit from factorization. As will be explained momentarily, this is done by adopting a limited form of f–representations. We review the core of the idea here, which is to represent intermediate relations as multiple factorized groups of vectors (vector groups). We refer readers to the original publications on these systems describing the details of their factorized vector executions (Gupta *et al.*, 2021; Feng *et al.*, 2023). For simplicity, we use `Graphflow`'s query processor to present this core idea using its EXTEND/INTERSECT operator (see Section 3.3.3 for a review of EXTEND/INTERSECT).

Let us continue our example, this time using Q_2:=$R(a_1, a_2)$,$R(a_1, a_3)$ from Section 4.2 on the relation $R(src, dst)$ from Figure 4.1. Consider the plan in Figure 4.10a for Q_2. The plan scans a_1 values and then does an `Extend` with $R(a_1, a_2)$ to produce (a_1, a_2) tuples and another `Extend` with $R(a_1, a_3)$ to produce (a_1, a_2, a_3) tuples. In vanilla vectorized execution, the first `Extend` operator will take in a vector of $a_1 : [1, 2, ..., 8]$ values and extend these to two vectors of $a_1 : [...]$ and $a_2 : [...]$ values. Recall that there are 8 nodes in the input graph in Figure 4.1. Since `Extend` of a_1 with $R(a_1, a_2)$, extends each a_1 value to its outgoing neighbors and since node 1 has 3 outgoing neighbors, the first 3 values in these vectors would be $a_1 : [1, 1, 1, ...]$ and $a_2 : [2, 3, 4...]$. The second `Extend` operator with $R(a_1, a_3)$ extends each a_1 value in the two a_1 and a_2 vectors to its outgoing neighbors. Figure 4.11a shows the first 9 tuples of the output a_1, a_2, and a_3 vectors (rest of the vectors are omitted for simplicity). As shown in the figure, vanilla vectorized execution generates many value repetitions in its vectors when performing m-n joins. For example $a_1 = 1$ value is repeated 9 times in these 9 tuples. Further, each of the a_2 values are repeated 3 times.[3]

Factorized vectors address these repetitions by representing intermediate relations as multiple *vector groups*. Consider the vector representation in Figure 4.11b. In this representation, there are three vector groups each drawn in separate rectangles. Each vector group has a field called position, that can take one of two values. -1 indicates that the vector represents all of the values in the vector. When it is set to ≥ 0,

[3]In some systems, instead of actual variable values, some offsets of the values would be repeated.



(a) Vanilla vector execution. **(b)** Factorized vector execution.

Figure 4.11: Vectors received by the Sink operator of the plan in Figure 4.10a using different vector execution approaches.

the vector represents a single value.[4] A set of vector groups represent the Cartesian product of the values each vector group represents. For example, the three vector groups in Figure 4.11b represents exactly the same 9 tuples represented in Figure 4.11a. Note that the figure shows 8 a_1 values, 1, ..., 8, in the a_1 vector although only the one at position 0, which is $a_1 = 1$, is part of the current state. This will become clear momentarily as we describe the actual execution.

The plan in Figure 4.10a generates these factorized vectors as follows. The scan generates the vector group VG_1 containing a_1 values by scanning a vector, in this example 8 tuples. At this stage, the position value of VG_1 would be -1 (omitted from the figure). The first `Extend` operator, which extends a_1 values to a_2 first "flattens" VG_1 by setting the position value to 0. Then, it would write the set of "neighbors" a_2 of $a_1 = 1$ to the second vector group VG_2 with position set to -1. Finally, the second `Extend` operator, which extends a_1 values to a_3, simply writes the set of neighbors a_3 of $a_1 = 1$ to the third vector group VG_3 with position set to -1. In short, the `Extend` operators, flatten the vector group of their bound variable (if they have to) and write a set of neighbors to a separate vector group in an "unflat" way, i.e., by setting position to -1. In the Kùzu system, instead of `Extend` operators `HashJoin` operators similar to those presented in Section 3 are used but intermediate relations are still represented using vector groups.

[4]We note that each vector group can contain multiple vectors, in which case their state is represented by the same position value and take part as a single unit in the Cartesian product.

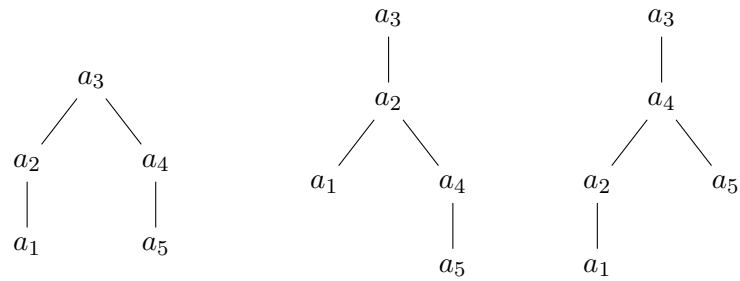
At a high-level the factorized vector execution is based on delaying any value repetitions until it is necessary according to a query plans. This allows passing tuples in f-representations between operators with some vector groups having positions set to -1. The plans in `Graphflow` and Kùzu are generated through a traditional dynamic programming-based join order optimizer. The main change to this standard optimization approach is that these systems use as cost metric the expected number of factorized tuples that would be passed between operators. To estimate this, during the dynamic programming optimization, the optimizer keeps the factorization structure of the vectors that would be generated for each sub-plan, and uses this information to estimate how much a new join would increase the number of tuples generated.

4.3.3 Benefits and Limitations of Factorized Vector Execution

Keeping the intermediate relations in factorized vectors avoids value repetitions, and leads to less data being written to intermediate data structures such as hash tables. For example Kùzu has a data structure called "FactorizedTable" in the blocking operators of the system that need to accumulate sets of tuples (Feng *et al.*, 2022). Another benefit of using factorized vectors is that the computation is pipelined similarly to standard vectorized execution and in most cases, operators can operate on vectors. Another major advantage is that some aggregation computations can be done very efficiently. For example counting operator simply multiplies the sizes of each vector group to compute the number of tuples represented by each intermediate chunk it receives. The benefit of fast aggregations is an artifact of factorized query processing, and also applies to FDB-style processing.

At the same time, there are several limitations of using factorized vectors. Specifically, there may be plans in which an expression has to run on two flattened vectors, in which case the processing happens on one tuple at a time. Further, factorized vector execution can only produce f-representations that follow a limited set of f-trees. The allowed representations are ones following f-trees where each node has at most 1 non-leaf child node, which for many queries may not be the ideal factorization. The reason behind this constraint is that unlike tries, factorized

vectors cannot produce nested groupings of variables. Consider, for example, the 4-hop query $Q_{4H} : R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$. The best possible representation would follow the f-tree in Figure 4.12a with a worst-case size N^2. However factorized vector executors can only obtain factorizations of the form in Figure 4.12b with a worst-case size N^3. This means that certain compression benefits from factorization are not attainable.



(a) Most compact f-tree for 4-hop query.

(b) Examples of most compact f-trees that factorized vector execution can use for 4-hop query.

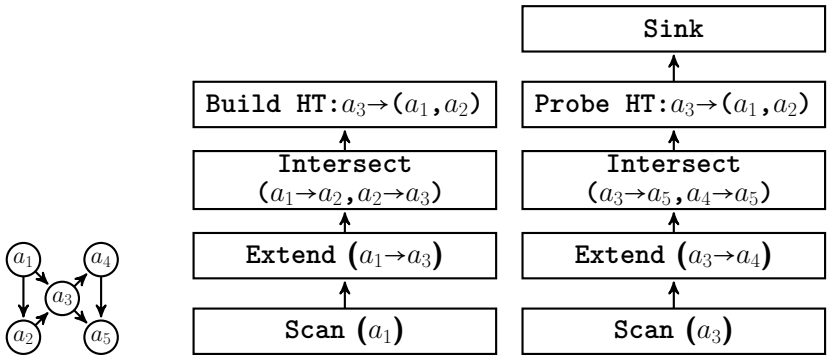
Figure 4.12: F-trees supported by factorized vector execution.

4.3.4 Note on Factorization and Worst-case Optimal Joins

Both `FDB` and factorized vector execution can work seamlessly in plans that also employ worst-case optimal join algorithms. For example, Figure 4.13b shows a `Graphflow` plan for the two-triangles query $Q_{2\Delta} \coloneqq R(a_1, a_2), R(a_1, a_3), R(a_1, a_3), R(a_3, a_4), R(a_3, a_5), R(a_4, a_5)$ in Figure 4.13a. In the plan both triangles are evaluated with `Graphflow`-style worst-case optimal join operator. Further the intermediate results of left and right triangles are factorized. An example set of vector groups that the Sink operator would receive in this plan are shown in Figure 4.13c. There are 5 vector groups. Three of them are flattened with position values 0 while two of them are unflat with values -1.

4.4 Background on D-Representations

In this section, we give an overview of *d-representations*, i.e., factorized representations *with definitions*. D-representations are extensions



(a) Query. **(b)** Plan.

(c) Vector groups.

Figure 4.13: Example of a plan and of vector groups received by the SINK when evaluating: $Q_{2\Delta} := R(a_1, a_2), R(a_1, a_3), R(a_1, a_3), R(a_3, a_4), R(a_3, a_5), R(a_4, a_5)$.

of f-representations and can be more succinct than f-representations (Olteanu and Zavodny, 2015). Consider the input relation R in Figure 4.4 and the four-hop query Q_{4H}=$R(a_1, a_2)$, $R(a_2, a_3)$,$R(a_3, a_4)$, $R(a_4, a_5)$. A copy of R is shown in Figure 4.14 for convenience. F_1, in Figure 4.5a, is an f-representation for Q_{4H}'s output following $\mathcal{T}_1$ in Figure 4.5b. F_1 uses two subquery results multiple times. In Figure 4.5a, all subquery results rooted at v_2 and v_4 are exactly the same. These subquery results can be re-used to further compress F_1. A reusable subquery result is called a *definition*. We show this in Figure 4.15a, where we use the same nodes in dashed boxes to indicate a single shared subquery result. The figure shows a d-representation Fd_1 with two definitions:

- $D_1 = \cup_{i=1,a_5}^{i=k} (v_{5_i})$
- $D_2 = \cup_{m=1,a_3}^{m=k} (v_{3_m} \times v_4 \times D_1)$

Given R, F_1 has $2k \times 2k \times k = 4k^3$ (v_i) values. After reuse of the subquery results D_1 and D_2, the size of the d-representation Fd_1 is $2k+2k+k = 5k$. Figure 4.15b shows another d-representation Fd_2. This one compresses F_2 re-using two subquery results in it.

Similar to f-representations, one can obtain possible d-representations of queries by statically analyzing queries. In other words, the

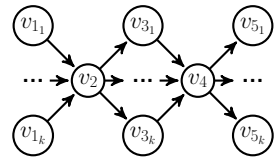
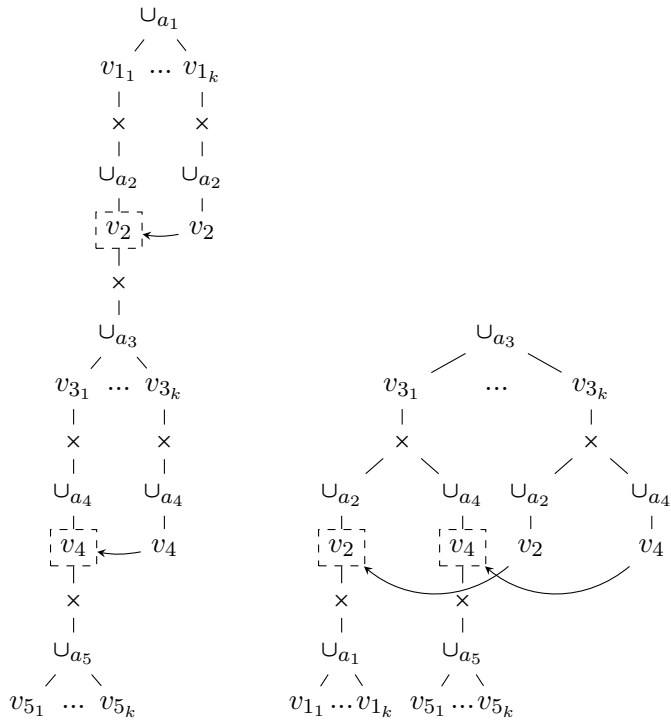


Figure 4.14: Copy of relation $R(src, dst)$ from Figure 4.4.



(a) Fd_1. **(b)** Fd_2.

Figure 4.15: Output of $R(a_1, a_2), R(a_2, a_3), R(a_3, a_4), R(a_4, a_5)$ as d-representations Fd_1 and Fd_2 following $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. $\mathcal{T}_1$ is in Figure 4.6a and $\mathcal{T}_2$ is in Figure 4.6a.

common subquery results that can be defined and reused can be identified by static analysis during query compilation. The rule is this. Consider an f-tree $\mathcal{T}$ and a subtree $\mathcal{T}$ rooted at a child of c_{ij} of node a_i. If the nodes in $\mathcal{T}$ depend only on a_i but none of of a_i's ancestors, then the result of values that will bind to the variables in $\mathcal{T}$ is identical

for a given value of a_i, say $a_i = x$. Therefore, it is possible to re-use the whole subquery results for $\mathcal{T}$ for the same value of a_i. To make these dependencies explicit, we annotate f-tree nodes with dependency information following the notation of Olteanu and Schleich (2016). Figure 4.16 shows an example denoted by $\mathcal{T}^{\uparrow}$. Specifically, each node a_i is annotated by a *key* property that is used to keep track of the subset of ancestors that the node a_i depends on. The above rule can now be restated as follows. Consider a child c_{ij} of a_i. If the key associated with c_{ij} only contains a_i, then then the result of values that will bind to the variables in $\mathcal{T}$ is identical for a given value of a_i, say $a_i = x$, irrespective of the rest of the values that were bound to the ancestors of a_i. The more general rule is that a subquery result can be re-used for each value of the key of c_{ij}, but sharing is simplest and maximized if the key of c_{ij} contains only a single variable (a_i is our case).

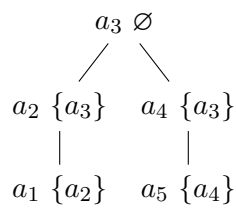


Figure 4.16: Extended f-tree example $\mathcal{T}^{\uparrow}$.

For example, in Figure 4.16, the subtree rooted in a_1 only contains a_2. Therefore when $a_2 = v_2$, the sub-query results for a_1 will be identical. Therefore, in the d-representation Fd_2 in Figure 4.15b, the right $a_2 = v_2$ branch can simply point to the left branch. Similarly, the subtree rooted in a_5 only contains a_4. Therefore when $a_4 = v_4$, the sub-query results for a_5 will be identical and the right $a_4 = v_4$ branch in Fd_2 points to the left branch.

4.4.1 Worst-case Size Bounds for D-representations

Similar to the definitions of $N^{\rho^*(Q)}$ and $N^{s(Q)}$, one can define the worst-case size of d-representations of query results. For simplicity, we assume all queries contain self-joins and are hence on the same relation R with N tuples.

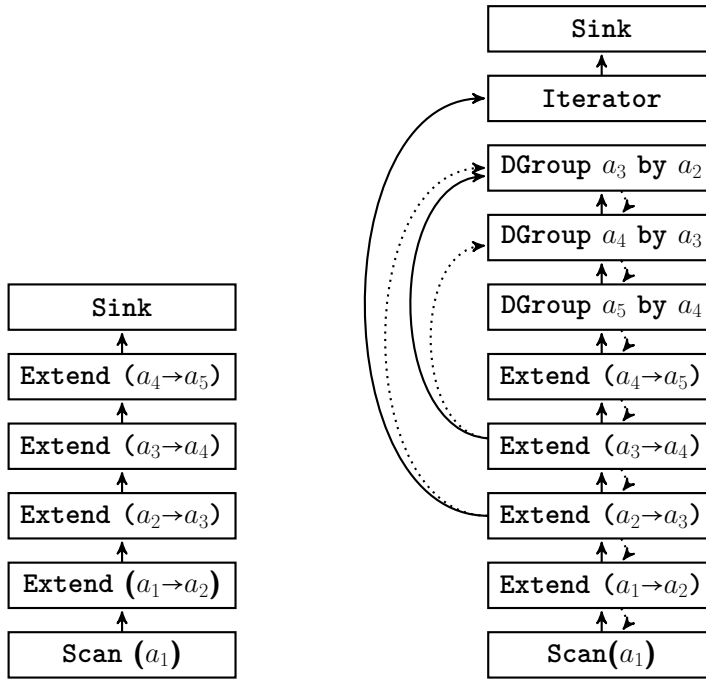
Definition 4.3. (Minimal worst-case output sizes of d–representations) Consider Q and an arbitrary valid f–tree $\mathcal{T}$. Let $N^{s(Q)_\mathcal{T}}$ be the worst-case size of the output of Q as a d-representation in the structure of $\mathcal{T}$ (re-using subquery results when possible) across all possible database instances where each relation in Q contains N tuples. Then $N^{s^\uparrow(Q)}$ is the minimum such worst-case size under the "best" valid f-tree for Q, i.e., $s^\uparrow(Q) = \min_\mathcal{T} s(Q)_\mathcal{T}$ where the minimum is taken over all valid f–trees for Q.

In general, one can establish that $s^\uparrow(Q) \leq s(Q) \leq \rho^*(Q)$. Further, there exists queries such that each representation system is strictly more succinct than the next. For example, the AGM bound of Q_{4H} is $\rho^*(Q_{4H}) = N^4$. The worst-case output size of the best f-representation of Q_{4H} is N^2, which can be obtained using the f-tree $\mathcal{T}_2$ in Figure 4.6b. In contrast, the worst-case output size of the best d-representation of Q_{4H} is N, which can be obtained using either of the f-trees $\mathcal{T}_1$ or $\mathcal{T}_2$.

4.5 Approach to Adopting D-Representations by `Graphflow`

There is little work on adopting d-representations in query processors. We briefly cover the only work that has proposed to adopt d-representations, done in the context of the `Graphflow` system (Mhedhbi, 2023). The core approach in this work is to have DAG-style query plans where some join operators have caches that store re-usable subquery results. If a subquery result has been cached before, a sequence of operators are skipped. The approach is built on top of the factorized vector execution model. We cover the approach that works for linear plans, which captures the key technique. Mhedhbi (2023) proposes further extensions to plans with `HashJoin` operators that form plans with branches.

Let us return back to the query Q_{4H}, this time on the $R(src, dst)$ relation from Figure 4.14, which contains 4 paths. Consider using the a_1 to a_5 linear f-tree $\mathcal{T}_1$ from Figure 4.15. Figure 4.17a shows a `Graphflow` plan for this query that only uses f-representations as in Section 4.3.2. Figure 4.17b shows an extension of this plan into a DAG-style plan that caches some common subquery computations in the `Extend` operators.



(a) P. **(b)** DF.

Figure 4.17: `Graphflow` WCOJ and DAG-version for Q_{4H}.

The sequence of joins that are performed across the two plans is exactly the same. The primary difference is that the extended plan contains a sequence of `DGroup` operators that are appended after some of the `Extend` operators. The `DGroup` operator takes in a set of vector groups as other operators and constructs one level of a trie by grouping by a single value of a_i a set of values of a_j. We say that a `DGroup` operator groups a_j by a_i. For example, the top `DGroup` operator groups a_3 values by a_2. The previous `DGroup` operator groups a_4 values by a_3. A sequence of `DGroup` operators form a trie. This trie is based on a hash index (see Mhedhbi, 2023, for the implementation details). Tries are caches that previous `Extend` operators use to check if a particular subquery has been previously computed or not.

For example, the second `Extend` operator ($a_2 \rightarrow a_3$), which extends a_2 values to a_3 values, checks if the cache contains an a_2 value for

each a_2 value it processes. If not, then the computation continues as in Section 4.3.2. For example, consider the beginning of the execution of the pipeline. Scan and the first `Extend` operator pass ($a_1 = v_{1_1}$, $a_2 = v_2$) tuple to `Extend` ($a_2 \rightarrow a_3$) (in factorized vector representation). Since the cache is currently empty, `Extend` ($a_2 \rightarrow a_3$) will continue regular processing. It will write $v_{3_1}, ..., v_{3_k}$ to a new vector group storing a_3 bindings. The next `Extend` a_3 will also continue regular processing, flatten this vector group to position = 0, so effectively pass ($a_1 = v_{1_1}$, $a_2 = v_2$, $a_3 = v_{3_1}$) to the next `Extend` operator. Later on during this pipeline, the sequence of `DGroup` operators start accumulating the tuples that are passed in and construct a trie. The final trie that is formed is shown in Figure 4.18 pictorially.

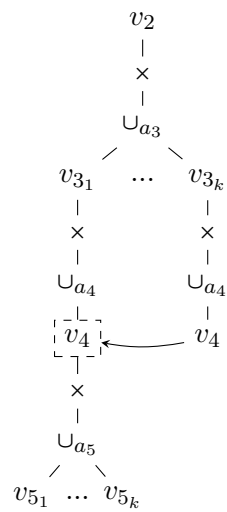


Figure 4.18: Output of the d-representation cached as part of evaluating Q_{4H} on R in Figure 4.14.

Consider the next time `Extend` ($a_2 \rightarrow a_3$) receives a tuple with $a_2 = v_2$, e.g., ($a_1 = v_{1_2}$, $a_2 = v_2$). Since the cache now contains the set of a_3, a_4, and a_5 values for $a_2 = v_2$, `Extend` ($a_2 \rightarrow a_3$) directly passes the tuple ($a_1 = v_{1_2}$, $a_2 = v_2$) to the `Iterator` operator after the sequence of `DGroup` operators, which copies the a_3, a_4, and a_5 values to their vector groups from the cache and continues pipelining these factorized vectors

to the next operator, which is `Sink` in this case. Although omitted from our description of the execution in the previous paragraph, the same logic of checking the cache and skipping a sequence of operators happens at each `Extend` operator. For example, during the construction of the trie for $a_2 = v_2$ in the cache, other `Extend` operators, such as the one extending a_4 to a_5 would skip computing the extension of $a_4 = v_4$ values to a_5 values if this computation was done and cached previously.

We next discuss several implementation details. First, although our example described this approach for join-only queries, there can be other operators in the plans such as filters. For example, if the query was computing 4-paths where the a_3 nodes have some property, a filter operator after the `Extend` operator could be in the plan without affecting the caching and placement of the `DGroup` operators. The important criteria is that the filter does not violate the dependency relationships in the f-tree. This point is related to plan generation. `Graphflow` generates these plans in a rule-based manner. First the system generates a plan with no consideration for caching opportunities, using a dynamic programming-based join optimizer that aims to minimize the number of factorized tuples passed between operators (recall from Section 4.3.2). Then the system analyzes the dependencies between the operators in each linear pipeline to identify if there are any sequence of operators whose results can be cached and reused. If so, the system puts a `DGroup` operator to the end of such pipeline with the necessary directed edge from a previous `Extend` operator. Finally, similar to our note on f-representations, the key approach presented above can be adopted as is in plans that have worst-case optimal join operators. We refer the readers to Mhedhbi (2023) for examples of such queries and plans.

4.6 Data-dependent Compression

The approaches that we have seen so far are what may be called data-independent or schema-driven approaches. For both f- and d-representations, we only need to analyze the query and the base relations to choose an f-tree and use it for factorization. Although the data does impact the sizes of the f- or d-representations and would influence any optimization decisions, the compression that they enable is based on the

relationships between the attributes in the relations and the query itself. This makes the approach very attractive and easy to use. However, it does limit the applicability somewhat. For instance, consider the query $Q(a_1, a_3) := R(a_1, a_2), R(a_2, a_3)$. In this case, we are projecting out the join attribute and the approach we have discussed cannot help in compressing the final result in a duplicate-free manner. Next, we briefly discuss a data-dependent approach, called *GraphGen* (Xirogiannopoulos and Deshpande, 2017; Xirogiannopoulos *et al.*, 2017), that was designed for this type of a query.

GraphGen was motivated by the observation that, although analyzing interconnection structures between entities in a dataset (i.e., the graph structure over them) can provide important insights, graphs are not the primary representation choice for storing most data; instead graphs must first be explicitly constructed by fetching the relevant data from the underlying database, and appropriately creating the nodes and the edges in the graph. Because these joins are also typically *m-n* joins, the resulting "hidden" graphs can often be much larger than the initial input, making it a challenge to construct them fully (i.e., in the flat representation).

We use an example from Xirogiannopoulos and Deshpande (2017) to further motivate this problem, and the solution. Here the base table, *AuthorPub*, contains information about authors and their publications. A typical graph that one might want to analyze here (e.g., using a *community detection* algorithm) is the *co-authors* graph, where there is a node for each author, and an edge between them indicates at least one co-authored paper. Figure 4.19 shows an instance of such a table, as well as the co-authors graph on that instance where the edges are created using a self-join followed by a projection. To avoid clutter and to match the query, we show all of these graphs with two copies of each author node (corresponding to $ID1$ and $ID2$); however, those two would be merged in the graph data structure that is finally built.

Figure 4.19 shows a possible compact representation, called **C-DUP**, where nodes corresponding to the publications are **not** projected out. The graph traversal algorithms can be easily modified to *pass through* such nodes (see Xirogiannopoulos and Deshpande, 2017, for more details as well as for details on how to support a *vertex-centric* API on such

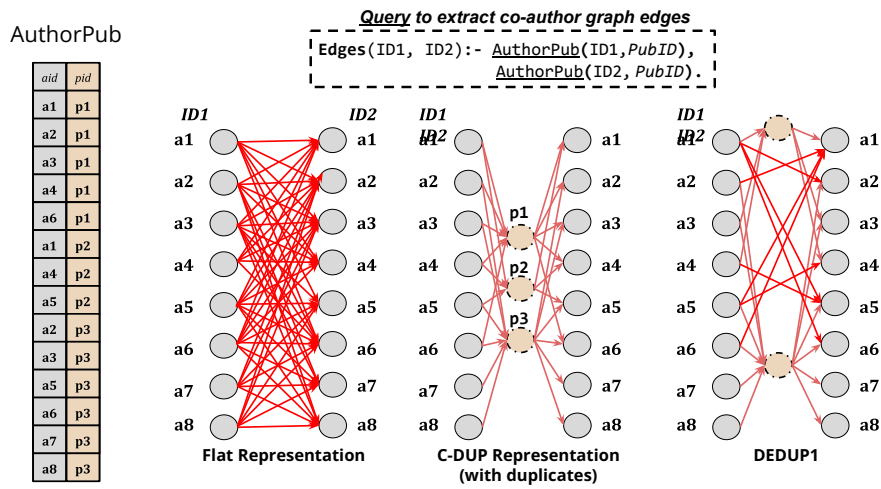


Figure 4.19: Example reproduced from GraphGen (Xirogiannopoulos and Deshpande, 2017)

representations). However, the problem with this representation is that, there are duplicate paths between the same two author nodes (e.g., $a1$ and $a4$). This makes it a challenge to run further processing because most operators we may wish to execute after this join (e.g., **sum** or **count** aggregates) would require eliminating possible duplicates in the result. A similar situation arises if a system used this representation to run batch graph algorithms, e.g., to compute PageRank or find the connected components of the graph.

Figure 4.19 also shows a duplicate-free compact representation, called **DEDUP1**; this representation is obtained, in essence, by analyzing the C-DUP representation and removing the duplicate paths between nodes while adding as few direct edges as possible. Minimizing the number of edges added is, however, NP-Hard, and the algorithms presented in that work are all heuristics. In fact, most general variations of graph compression are known to be NP-Hard. Xirogiannopoulos and Deshpande (2017) also discuss alternate bitmap-based representations that are easier to construct, but require more work during execution of graph algorithms, and also show several orders-of-magnitude improvements in memory consumption compared to generating the flat representation.

The deduplication cost, however, limits the applicability of this approach to scenarios where the resulting graph is materialized and analyzed repeatadly, or there is a need to run complex multi-pass graph algorithms on that graph. At the same time, this line of work highlights how data-dependent compression may be exploited to reduce the in-memory representations of intermediate results where the data-independent approach is not applicable.

4.7 Other Work and Open Problems

We end this section by providing pointers to several related works that use factorized representations, and then describe several open problems related to the adoption of f- and d-representations.

Answer Graph (Abul-Basher *et al.*, 2021) is a recent system that extends PostgreSQL's query processor for a join-only subset of SPARQL (i.e., without projections) that performs a two-stage query evaluation for acyclic queries. The first stage is a full semi-join reduction, similar to Yannakakis's algorithm, that identifies only and all of the edges that participate in the final output. This is done by performing a sequence of "forward extensions" according to a join order that is picked by a traditional cost-based optimizer. After this step, a second stage generates a set of flat tuples by executing a left-deep join plan. The result of the first phase of Answer Graph is similar to d-representations and the following enumeration phase flattens all results. The authors also describe an envisioned but not implemented version of semi-join reduction for cyclic queries, which is based on a more complex cascading logic which pipelined approaches do not need to handle.

At a high level, d-representations are based on caching and re-using common subquery results in queries. CTJ, which was introduced in Section 3 (Kalinsky *et al.*, 2017) is an alternate technique that can speed up WCOJ plans by reusing partial results. The difference is that its partial results are stored as flat tuples. However, it is based on a similar idea as expression reuse based on attribute dependency in f-trees, *i.e.*, extensions/joins to additional attributes can be repeated for future tuples.

Factorized representations can also be used in machine learning workloads and data science applications. The LMFAO engine (Schleich and Olteanu, 2020) is optimized to execute batches of group-by aggregates over joins within data science pipelines. F-IVM by Nikolic *et al.* (2020) and Kara *et al.* (2023) is a state-of-the-art incremental maintenance technique that uses factorization. F-IVM is based on high order delta decomposition technique, which was introduced by Ahmad *et al.* (2012) in the DBToaster system. In constrast to DBToaster, F-IVM keeps results of delta queries in factorized form. Similarly, factorized representations were used for result materialization for dashboards (Huang and Wu, 2023).

While the seminal work of factorization representation has been introduced a decade ago, there has been very few adoption approaches within systems and many open problems still remain:

- The most obvious open problem is to propose alternative approaches to integrate factorized representations in query plans in existing systems. The two approaches for f-representations and one approach for d-representations have many limitations and more systems work is needed to make factorization more practical and easier to integrate into systems.
- The optimization of plans that benefit from f- and d-representations is not well understood. The proposed approaches so far are either cost-based using very coarse metrics, such as minimizing worst-case sizes, or using rule-based optimizers, such as the one we described from `Graphflow` in the last section. These are not robust approaches, since worst-case sizes ignore the statistics about the actual database and the latter is not conscious of any caching opportunity during optimization. Novel approaches that take into account the size of intermediate results based on the f-tree used might lead to picking different plans and better performance.
- In this survey, we primarily covered approaches that aim to optimize m-n joins. Many analytical queries contain aggregations. Factorization can improve aggregations over queries with m-n joins significantly (Bakibayev *et al.*, 2013, Gupta *et al.*, 2021, Kara *et al.*, 2023). In fact, aggregation queries are probably the most promising

class of queries where factorization can improve existing systems. However, some of these benefits can also be achieved by pushing down aggregates beyond some of the joins in plans. It is however not well understood whether factorization provides new benefits to query processors on aggregations beyond pushing-down aggregations. Understanding whether the benefits are the same theoretically and empirically is an important future direction of study.

- While we covered some connections between MVDs and factorized representation, the connection between MVDs and d-representations is less clear. Unifying these different notions of conditional independences with possibly some new formalism and theories could help our understanding of these techniques.

5

Execution of Regular Path Queries

In this section, we expand our focus beyond the previously discussed subgraph pattern-based queries, which translate into multi-way join queries without recursion. We delve into the execution of regular path queries (RPQs) in graph databases, which is a significant class of queries in graph databases that has gained prominence in recent years. RPQs form a subset of recursive queries that has been studied extensively in prior research, whose semantics are well covered in modern query languages of GDBMSs, and that have been integrated into several DBMSs. We also briefly discuss several works on shortest path queries that have been integrated into DBMSs.

RQPs involve identifying paths and/or source-destination pairs in a graph, where the paths adhere to specific properties defined by a regular expression. Conceptually, a regular path query is structured as $\langle ?source,\ regex,\ ?dest \rangle$, where $?source$ and $?dest$ are vertices within the graph (that may be constants or unbound). The objective is to determine the $\langle ?source, ?dest \rangle$ pairs that are connected by a path that conforms to the regular expression applied to the path labels, and potentially to retrieve such paths.

For instance, consider a graph as depicted in Figure 5.1. One example of a regular path query could be $\langle `Mahinda', \ Follows^* \cdot Lives, \ `NewYork' \rangle$, which seeks to establish if there is a path from 'Mahinda' to 'New York' through other vertices where the labels of the edges on the path satisfy the regular expression $Follows^* \cdot Lives$, i.e., a path with any number of 'Follows' edges, followed by a single 'Lives' edge. Another query, $\langle ?src, \ Follows^* \cdot Lives, \ `NewYork' \rangle$, aims to identify all nodes connected to someone living in 'New York' through one of more 'Follows' edges. While these regular expressions could include labels on both nodes and edges, for simplicity, we focus on expressions pertaining solely to edge labels, assuming each edge bears a single label. A key challenge with regular path queries lies in their semantics, particularly in defining paths within graphs, a topic we explore in Section 6.1. Furthermore, the intricacies of identifying and selecting paths, as opposed to merely finding source-destination pairs that satisfy the query, add layers of complexity. This complexity is evident even in the simplest formulations, where the problem can escalate to being NP-hard.

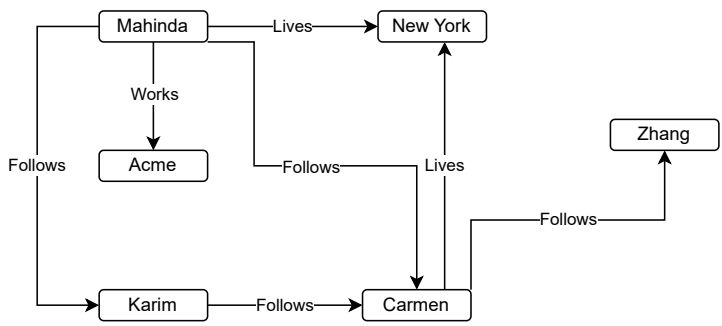


Figure 5.1: An example directed edge-labeled graph

The rising importance of regular path queries is underscored by their inclusion in various query languages, such as SPARQL (W3C, 2024) (through the notion of *property path queries*) and the newly standardized Graph Query Language (GQL) (JCC Consulting, Inc., 2024), as well as SQL extensions for graph querying like SQL/PGQ (ISO/IEC JTC 1/SC 32, 2024). Given this burgeoning interest and

the ongoing development in this field, this section will provide an in-depth examination of regular path queries, highlighting the substantial opportunities for further research in this area.

5.1 Background

Regular path queries have been a subject of study in the database literature for decades, tracing back to the seminal work by Mendelzon and Wood (1995). For simplicity, in this section, we focus on edge-labeled directed graphs, where each node and edge is associated with a unique identifier, and further each edge is labeled with a single label from a predefined set of symbols, Σ. Formally, we define a database graph to be a tuple $G = (V, E, \ell)$, where V is a set of vertices, E is a set of (directed) edges, and $\ell : E \to \Sigma$ is a labeling function that maps each edge to a label from Σ. Note that, there may be more than one edge between a pair of vertices, and self-loops are allowed.

A *path* P in G is a sequence of nodes and edges $v_1, e_1, v_2, e_2, \ldots, e_{k-1}, v_k$ such that for all $i \in [1, k-1]$, the source and destination vertices of e_i are v_i and v_{i+1} respectively. The *label* associated with a path P is the concatenation of the labels of the edges in P, i.e., $\ell(P) = \ell(e_1) \cdot \ell(e_2) \cdots \ell(e_{k-1})$.

A *regular expression* R over Σ is defined inductively as follows in a standard manner:

- $\varnothing$ is a regular expression.
- ϵ is a regular expression.
- For every $a \in \Sigma$, a is a regular expression.
- If R_1 and R_2 are regular expressions, then so are $R_1 \cdot R_2$ (*concatenation*), $R_1|R_2$ (*alteration*), R_1^* (*Kleene or transitive closure*), R_1^+ (equivalent to $R_1 \cdot R_1^*$), and $R1?$ (*0-or-1*).

SPARQL also supports a more general form of Kleene closure that allows the user to specify a range of repetitions, e.g., $R_1^{m,n}$, which denotes m to n repetitions of R_1. For simplicity, we omit this and other extensions to regular expressions in our discussion.

Given the above definition, the most general form of a regular path query requires the user to specify:

1. The start and end points of the path, which may be constants or variables.

2. A regular expression formulated over Σ.

3. **Restrictor**: There are four distinct types of restrictions that can be imposed on the path, adopted by different query languages, with GQL necessitating an explicit choice by the user. These restrictions include:

   - *walk*, where the path P is not subject to any restrictions.
   - *trail*, in which P does not repeat any edge.
   - *acyclic* if P does not revisit any node.
   - *simple* if it does not revisit any node with the exception of the start and end points (i.e., we allow $v_1 = v_k$).

4. **Selector:** In addition, some of the query languages also allow returning the matching path(s) in addition to the start and end points. In that case, the options are:

   - *any*, where any path that satisfies the query is returned.
   - *any shortest*, where any shortest path that satisfies the query is returned.
   - *all shortest*, where all shortest paths that satisfy the query are returned.

   In addition to these, GQL also supports a form of *top-k* queries, where the user can specify a positive integer k and the query returns the k shortest paths that satisfy the query.

The possible combinations of the above options lead to a large number of query types, which makes the problem of evaluating regular path queries challenging. For many of the combinations, the problem is known to be NP-hard, especially when the query is restricted to simple paths or trails. Even a simple regular expression like $(aa)^*$ is NP-hard

to evaluate on simple paths (Mendelzon and Wood, 1995) (since it is equivalent to finding a simple path between two nodes in a graph of even length; Lapaugh and Papadimitriou, 1984). We refer the readers to Farias *et al.* (2023) for a more detailed discussion of the complexity of regular path queries under different restrictions.

In the rest of this section, we discuss the basics of the two main approaches that have been proposed to evaluate regular path queries. The first approach is based on use of finite state automata, whereas the second approach uses an extended relational algebra. We also briefly discuss some of the recent work that has been proposed to improve the scalability of these approaches.

5.2 Automata-based Techniques

The first approach to evaluate regular path queries is based on the use of finite state automata (Mendelzon and Wood, 1995). This method, often called *product construction*, "multiplies" the database graph G and a non-deterministic finite state automaton (NFA) A that recognizes the regular expression R to create a new graph $G \times A = G_A$. The nodes of G_A are pairs (v, q), where $v \in V$ and q is a state of A. There is an edge from (v_1, q_1) to (v_2, q_2) in G_A if there is an edge from v_1 to v_2 with label l in G and there is a transition from q_1 to q_2 labeled l in A. We assign the label l to this edge in G_A.

Figure 5.2 shows a portion of this construction for the graph in Figure 5.1 and the regular expression $(Follows \cdot Follows)^* Lives$, for a graph traversal starting at node *Mahinda*. For example, we have an edge from $(Mahinda, q_0)$ to $(Karim, q_1)$ because $Mahinda$ follows $Karim$, and there is a transition from q_0 to q_1 in the NFA with label $Follows$. Only the labels in the regular expression appear on the edges of G_A. We also note that $(Carmen, q_0)$ has an edge to $(NewYork, q_2)$, however, there is no edge in this graph from $(Carmen, q_1)$ to $(NewYork, q_2)$, since the NFA does not have a transition from q_1 to q_2 labeled $Lives$.

Now, consider a path from (v_1, q_0) to (v_2, q_2) in G_A, and further, let q_2 be a *final* (accepting) state for the NFA. Then, the label of the path from v_1 to v_2 in G is in $L(R)$, i.e., the path satisfies the regular expression R (intuitively, we are traversing both the graph and the NFA

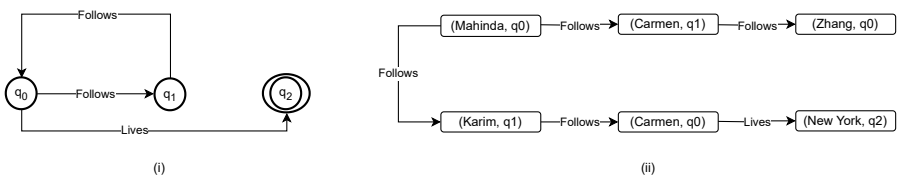


Figure 5.2: (i) An NFA for the regular expression $(Follows \cdot Follows)^* \cdot Lives$; (ii) Portion of the product graph relevant to exploration from node *Mahinda*

at the same time). Thus, the problem of evaluating a regular path query can be reduced to the problem of finding a path in G_A from (v_1, q_0) to (v_2, q_2), where q_2 is a final state, which can be done using standard graph traversal algorithms like breadth-first search.

This algorithm runs in polynomial time in the size of the graph and the NFA (specifically, $O(|G||A|)$, where $|G|$ and $|A|$ denote total number of nodes and edges in G and A respectively), and is thus practical. Furthermore, we can also construct the product graph *on demand*, i.e., as it is being traversed. In fact, it is not necessary to explicitly construct the product graph at all, since we can simply keep track of the current state of the NFA as we traverse the graph.

However, one important caveat here is that the path that we find here is without any restrictions (i.e., the "walk" semantics from above). The algorithm can be extended to handle the other restrictions, however, it does not retain its polytime complexity in that case. For example, if we want to find a simple path, then we need to ensure that the path in G_A does not revisit any node in G. The only known way to do it is by finding all paths from (v_1, q_1) to (v_2, q_2) (where $q_2 \in F$) in G_A and then checking if any of them are simple. We refer the reader to Mendelzon and Wood (1995) for a more detailed discussion of the cases where the algorithm retains its polytime complexity.

This algorithm can also be extended to return a path, a shortest path, or all shortest paths, instead of just the start and end points. In that case, we need to keep track of the path traversed in G and the states traversed in A at each step. We refer the reader to Farias *et al.* (2023) for more details.

5.2.1 Similarities to Relational Query Execution

Although the graph-traversal based algorithm might seem very different from how relational queries are executed, we can draw parallels between the two under certain assumptions about the specifics of how the traversal is done. In particular, we can view the traversal as a form of a series of joins with an *edges* relation (in the form of a recursive query). Specifically, let $R_G(src, dest, label)$ denote a table representation of the graph G, where each tuple represents an edge from v_1 to v_2 with label $label$. Further, let's assume we are traversing the graph in a breadth-first manner and neither the source nor the destination vertex is fixed. Let's say there are two transitions from q_0, to q_1 and q_2, with labels l_1 and l_2, respectively. Then, the first step of the breadth-first traversal can be seen as (with some liberties taken with the syntax):

```
\small
R_1 = select distinct src,dest,(label=l_1) ? q_1 : q_2 as q
from R_G
where label in [l_1, l_2]
```

Say now there are two transitions from q_1, to q_3 and q_4, with labels l_3 and l_4, respectively, and two transitions from q_2, to q_5 and q_6, with labels l_5 and l_6, respectively. Then, the second step of the breadth-first traversal can be seen as the join:

```
\small
R_2 = select distinct R_1.src,R_G.dest,new q
                (e.g., q_3 if R_q = q_1 and R_G.label = l_3)
from R_1, R_G
where R_C.dest = R_G.src and
             ((R_1.q = q_1 and R_G.label in [l_3, l_4]) or
              (R_1.q = q_2 and R_G.label in [l_5, l_6]))
minus
select * from R_1
```

In general, except for the initialization step, the same join will be evaluated at each step of the traversal, with the join condition being

a disjunction of conjunctions of the form ($R.q = q_i$ and $R_G.label = l_j$), where q_i is a state in the NFA and l_j is a label in the regular expression. The set minus operation ensures that we do not revisit any node in the product graph (starting with the same node). The traversal terminates when there are no more tuples in the result, i.e., the *fixpoint* is reached.

5.2.2 Performance Considerations

Although simple and elegant, the graph-traversal based algorithm as described above can have significant performance issues for large graphs. In particular, the "query plan" is fixed and dictated by the NFA, and thus, the algorithm cannot take advantage of any order optimizations or memoization opportunities. For example, consider a simple regular expression $\alpha \cdot \beta \cdot \gamma \cdot \lambda$. The algorithm will first find all 4-paths matching $\alpha \cdot \beta \cdot \gamma$ (i.e., paths with 3 edges with labels α, β, γ in that order), and then filter out the ones that cannot be extended with λ. However, if the first three labels are very common and the last label is very rare, then the algorithm will have to traverse a large portion of the graph before finding any paths that satisfy the query (or finding that there are no paths that satisfy). Koschmieder and Leser (2012), among others, have looked at ways to optimize the traversal-based algorithms, e.g., through identifying *rare* labels and traversing the graph backwards and forwards from the edges that satisfy those labels to identify the paths that satisfy the query.

Second, consider the regular expression from above, $(Follows \cdot Follows)^* \cdot Lives$. Starting with all the nodes in the graph N, the algorithm will traverse the graph to find the set of nodes reachable through two $Follows$ edges, say N_1, and then again look for nodes reachable through two $Follows$ edges from N_1, and so on (to find the closure). It may be advantageous to first find all the pairs of nodes connected by two $Follows$ edges, and do a Kleene closure on those. However, the algorithm does not have the flexibility to do that. The relational algebra-based techniques we discuss next naturally support such optimizations.

5.3 Relational Algebra-based Techniques

The second approach to evaluate regular path queries is based on the use of an extended relational algebra, with *transitive closure* as a primary new operator. This methodology, pioneered by Losemann and Martens (2013) for evaluating path expressions in SPARQL, has underpinned substantial subsequent research, especially in the domain of SPARQL query processing. The approach systematically evaluates the regular expression by progressively identifying all node pairs satisfying segments of the expression, commencing with the most nested regular expressions.

For instance, consider our ongoing example with the regular expression $(Follows \cdot Follows)^* \cdot Lives$. The process initiates by identifying all node pairs (u, v) where there is an edge from u to v labeled *Follows*. Subsequently, it assesses the sub-expression $Follows \cdot Follows$, effectuating a self-join on the result, thereby yielding node pairs (u, v) connected via an intermediary node w, with both edges labeled *Follows*.

Following this, the approach incorporates the transitive closure operator to identify node pairs (u, v) connected by an even-length path, exclusively comprising edges labeled *Follows*. It's important to note that this path need not be simple; repetition of edges and nodes is possible. The transitive closure evaluation can be done, for example, through use of a standard semi-naive fixpoint evaluation technique.

After determining node pairs satisfying $(Follows \cdot Follows)^*$, the method concurrently identifies pairs conforming to the regular expression *Lives*, and subsequently joins these two intermediate results to finalize node pairs fulfilling the entire path expression. Figure 5.3 depicts this plan, where α denotes the transitive closure operator.

As elucidated in the referenced work, this approach is adaptable for more intricate regular expressions, including those imposing constraints on path lengths. For example, an expression like $(Follows \cdot Follows)^{3,5}$ that seeks paths of lengths 6, 8, or 10, can also be evaluated through use of matrix exponentiation. This method operates in polynomial time relative to the graph's size. However, because it does not restrict reuse of edges in any way, this approach is applicable solely for the "walk" semantics among the path types discussed earlier.

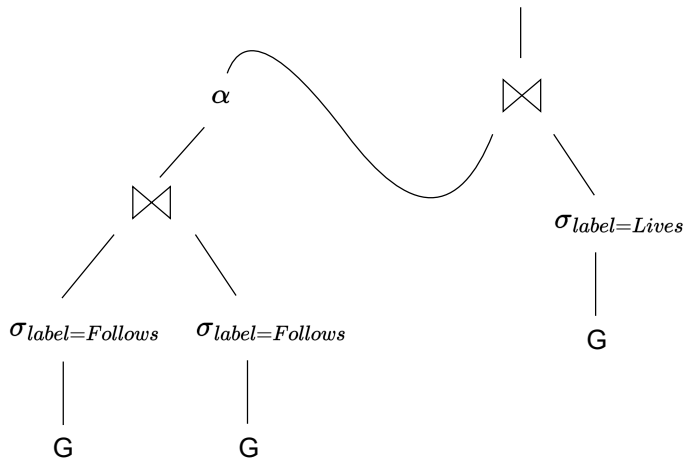


Figure 5.3: A relational algebra-based query plan for the regular path query $(Follows \cdot Follows)^* \cdot Lives$. We omit explicit "distinct" operators here – ideally, duplicates are eliminated as early as possible in the plan.

5.3.1 Performance Considerations

The relational algebra-based approach, as described above, can be seen as "materialization"-based approach, where every subexpression is computed in its entirety and materialized as a relation, and then the next subexpression is computed on top of that. Overall the approach is more amenable to optimizations than the graph-traversal-based approaches. For instance, given a simple query like $\alpha \cdot \beta \cdot \gamma \cdot \lambda$, the relational algebra-based approach can execute the joins in different orders (e.g., doing the join between pairs of nodes that satisfy γ and λ subexpressions first if there are fewer tuples in the result of the λ subexpression).

As described above, the nesting of the regular expression does impose some constraints on the order of evaluation. For instance, the expression $(Follows \cdot Follows)^* \cdot Lives$ must first evaluate the subexpression $(Follows \cdot Follows)^*$ before evaluating the join with the subexpression $Lives$. If $Lives$ is a rare label, then the transitive closure computation may be redundant for many node pairs. A computationally more efficient plan might be to first enumerate the node pairs (u, v) that satisfy $Lives$, and then traverse "backwards" from each of the u nodes to find

the node pairs (w, u) that satisfy $(Follows \cdot Follows)^*$. This can be seen as a form of *sideways information passing*, but is not possible in the bottom-up approach described above.

5.4 WaveGuide: Combining the Two Approaches

The above discussion, and the differences between the automata-based and the relational algebra-based approaches, naturally raises the question of whether it is possible to design a query execution algorithm and a corresponding plan space that subsumes both of them. This is precisely the motivation behind the work by Yakovets *et al.* (2016b), that we briefly discuss next. In addition to identifying and differentiating between the plan spaces explored by the above two approaches, the proposed system, called WaveGuide, explores a plan space that subsumes both of these plan spaces and does a cost-based optimization to choose an appropriate plan for a given query.

Specifically, for a given regular path query, WaveGuide uses a "waveplan" that consists of multiple automata, called Wavefronts, that may be independent and can be executed concurrently, or may depend on each other. We illustrate this using our running example. Figure 5.4 shows several different waveplans for executing the query $(Follows \cdot Follows)^* \cdot Lives$. The first waveplan, which emulates the automata-based approach, has a single wavefront that conceptually starts from all nodes in the graph and traverses the graph to find all the node pairs that satisfy the regular expression. The traversal follows a guided semi-naive evaluation strategy, where we start with a set of initial nodes ("seeds"), which in this case comprises of all nodes that have an outgoing $Follows$ edge. Then we take one step in the breadth-first traversal as discussed earlier for the automata-based approaches, identifying all node pairs (u, v) where v is reachable from u through a single $Follows$ edge. A cache is maintained to avoid redundant computations; all node pairs that are generated at a step are compared against the cache to identify the new node pairs to process in the next step (and to add to the cache). The process continues until a fixpoint is reached.

The second waveplan uses two wavefronts, one for computing the result of the query $Follows \cdot Follows$ (W_1), followed by a second wave-

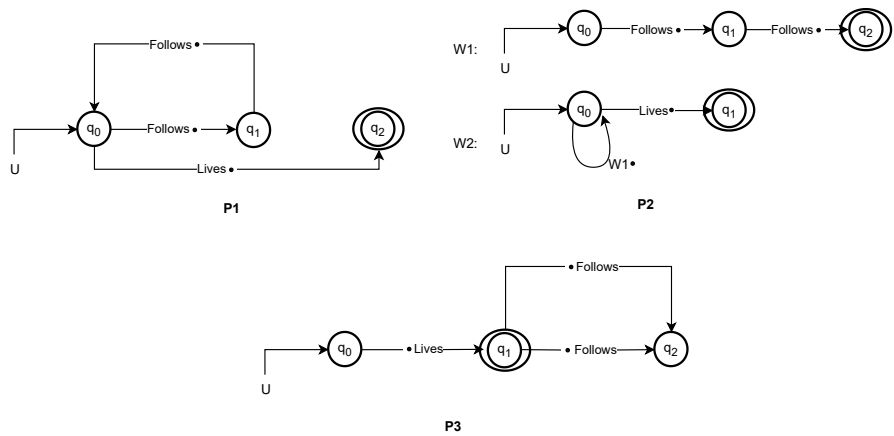


Figure 5.4: Three WaveGuide Plans: P_1 emulates a standard automata-based approach, P_2 emulates a relational algebra-based approach, and P_3 shows a plan that traverses the graph in the opposite direction.

front (W_2) that, in effect, computes the transitive closure of the result of the first wavefront, and then filters it by checking which of the resulting node pairs can be extended with a *Lives* edge. This is done by treating the result of the first wavefront as a *path view* that gets populated as the first wavefront start producing results. The ability to do this implies that the space of plans used by WaveGuide subsumes that of the relational algebra-based approach.

Another important aspect of a wavefront is the ability to contol the direction in which the graph is traversed. Figure 5.4 shows a third waveplan ($P3$) for the same query, which uses the same NFA as the first plan, but traverses the graph in the opposite direction, starting from the *Lives* edges. The traversal direction is indicated by the location of the '$\cdot$'. Specifically, $Lives\cdot$ denotes a forward traversal of the edge, where $\cdot Lives$ denotes a backward traversal. The third waveplan, thus, begins with first identifying all node pairs (u, v) where there is a *Lives* edge from u to v. For each such (u, v), it then looks for all *Follows* edges that end at u, traversing the graph in the backward direction for those edges, and so on. This ability to flexibly traverse the graph forwards or backwards enables Waveguide to address one of the major performance concerns of the automata-based approaches noted above.

The overall flexibility and significantly richer plan space of WaveGuide, makes it a promising approach for executing RPQs. However, the authors leave open the questions of how to enumerate valid waveplans for a given query, and how to choose the best waveplan for a given query. Another natural question is whether there is any "plan" that is not expressible in the waveplan space, i.e., whether the waveplan space is "complete".

5.5 Other Work

We briefly summarize some of the other works on RPQs for reference. We also briefly discuss shortest path queries, which is another class of popular recursive queries supported by modern GDBMSs.

As noted earlier, Koschmieder and Leser (2012) propose using rare labels to optimize the graph traversal-based approach. In a recent work, Arroyuelo *et al.* (2022) develop an improved automata-based approach that uses a novel compressed representation of the graph called a *ring* along with a Glushkov automaton to efficiently execute RPQs; the use of Glushkov automata enables a more space-efficient bit-parallel simulation of the NFA resulting in a significant speedup over the standard automata-based approach. Yakovets *et al.* (2013) present an SQL implementation of the relational algebra-based approach from Losemann and Martens (2013) by translating a SPARQL property path query to a *recursive* SQL query. Nguyen and Kim (2017) propose a cost-based approach to deciding how to split a query using rare labels. Dey *et al.* (2013) investigate several different variations of standard RPQ queries, including variations that return the edges along the paths between the node pairs. They use the term "provenance" to refer to the paths, also called "witnesses" in a later work by Farias *et al.* (2023).

There has also been a lot of work on a restricted class of RPQs called *label-constrained reachability queries*. The query is to find whether two nodes are reachable through a path, where each edge has a label from a set of labels ℓ_1, ℓ_2, ..., ℓ_k. Equivalently the RPQ is of the form $(\ell_1|\ell_2|$ ... $|\ell_k)^*$. Valstar *et al.* (2017) and Peng *et al.* (2020) study developing indices to process such RPQs faster and at scale. The standard algorithm to evaluate label-constrained reachability queries are breadth-first

search (BFS) traversal based algorithms. Zou *et al.* (2014b) develop a more advanced algorithm based on decomposing an input graph into strongly connected components and computing the label sets within each component. There has also been extensive work on standard reachability queries, which ignore labels. We refer readers to Yu and Cheng (2010) for a survey of this literature.

Finally, modern GDBMSs have special clauses or functions for shortest path queries, which form another class of popular recursive queries. Many existing systems such as Neo4j, Kùzu or Memgraph compile those clauses to specialized operators. These operators implement specialized shortest paths algorithms, such as Dijkstra's or Bellman Ford's shortest paths algorithms or their variants, such as bidirectional BFS algorithms. There is little technical writing on system integration approaches and implementation details of these algorithms but these systems have open-source repos, which contain their implementations. Wolde *et al.* (2023) is a recent project to implement a GDBMS layer over the DuckDB RDBMS (Raasveldt and Mühleisen, 2019a). The overarching goal of this project is to implement SQL/PGQ, which is a new extension to SQL to support property graph modeling and querying. DuckPGQ implements Multi-source BFS algorithm by Then *et al.* (2014) and its variants to find shortest path queries. However many aspects of integrating and optimizing shortest path algorithms in the context of DBMSs query processing is not well studied. For example, how to parallelize these queries is not well understood. Further the graph algorithms literature is full of techniques for fast shortest path computations, such direction optimized search algorithms. It is not clear whether systems can integrate such algorithms based on a set of primitive relational operators. This would simplify both integrating these optimizations into systems as well as enabling systems to automatically compose new optimizations. These are good directions for future systems-oriented research direction.

6

Conclusions

In this monograph, we covered a suite of modern query processing techniques that are particularly optimized for modern DBMSs that aim to support graph workloads. These are workloads on datasets that are naturally modeled as graphs and share the common characteristic of containing complex many-to-many or recursive join queries. Our techniques included pointer-based joins, WCOJ algorithms, factorization, and automata- or α-join based querying.

The first key takeaway we would like to emphasize to readers is that all of these techniques are based on relational principles. They are based on creating well-understood join indices, joins on multiple columns of relations, compressing intermediate relations, or extending relational algebra with an α operator (or using a specialized automata-based joins). This observation is important for two reasons. First, it solidifies our community's common understanding that scalable and performant data management, even if designed for datasets that are naturally modeled as graphs, should be based on relational principles. In fact, existing GDBMSs are indeed relational at their core since, aside from several specialized operators such as a shortest path algorithm, they process the records they manage using standard relational operators. Second, many

of these techniques have been developed in the context of relational systems, so we can expect that over time some of these techniques will be adopted by traditional relational systems as well. Our conviction is that over the next decade, systems that aim to optimize for graph workloads need to adopt these techniques or their variants to be competitive in their performances.

A second key takeaway is that although there is good understanding of the foundations of these techniques, the topic of how to integrate into systems is still in its experimental stage. It is difficult to highlight a common wisdom approach the community has agreed on about how to integrate these techniques into systems. For example, except for one work, there is no work on how to integrate d-representation-based query processing, and no work on how to develop optimizers to generate plans that use d-representations during query processing. Similarly, there are no clear principles about how the optimizers of systems should be developed if WCOJ algorithms were part of the suite of join operators in a DBMS. Many questions also remain on recursive queries, where the field has seen less work than the rest of the topics we covered. These include very core DBMS topics, such as whether one can develop practical join indices for these queries or how to parallelize these queries efficiently with existing parallelization techniques, such as morsel-driven parallelism. We hope these questions can inspire future work.

References

Abadi, D. J., A. Marcus, S. R. Madden, and K. Hollenbach. (2007). "Scalable semantic web data management using vertical partitioning". In: *Proceedings of the 33rd international conference on Very large data bases*. 411–422.

Aberger, C. R., A. Lamb, S. Tu, A. Nötzli, K. Olukotun, and C. Ré. (2017). "EmptyHeaded: A Relational Engine for Graph Processing". *TODS*. 42(4).

Abo Khamis, M., H. Q. Ngo, and D. Suciu. (2016). "Computing Join Queries with Functional Dependencies". In: *ACM PODS*.

Abul-Basher, Z., N. Yakovets, P. Godfrey, S. Clark, and M. H. Chignell. (2021). "Answer Graph: Factorization Matters in Large Graphs". In: *EDBT*. Ed. by Y. Velegrakis, D. Zeinalipour-Yazti, P. K. Chrysanthis, and F. Guerra. OpenProceedings.org.

Afrati, F. N. and J. D. Ullman. (2011). "Optimizing Multiway Joins in a Map-Reduce Environment". *IEEE Transactions on Knowledge and Data Engineering*. 23(9).

Agrawal, R. (1988). "Alpha: an extension of relational algebra to express a class of recursive queries". *IEEE Transactions on Software Engineering*. 14(7).

Ahmad, Y., O. Kennedy, C. Koch, and M. Nikolic. (2012). "DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views". *PVLDB*. 5(10).

Ammar, K., F. McSherry, S. Salihoglu, and M. Joglekar. (2018). "Distributed Evaluation of Subgraph Queries Using Worst-case Optimal Low-memory Dataflows". *PVLDB*. 11(6).

Aref, M., B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn. (2015). "Design and Implementation of the LogicBlox System". In: *ACM SIGMOD*.

Arroyuelo, D., A. Hogan, G. Navarro, and J. Rojas-Ledesma. (2022). "Time-and space-efficient regular path queries". In: *ICDE*.

Atserias, A., M. Grohe, and D. Marx. (2008). "Size Bounds and Query Plans for Relational Joins". In: *FOCS*.

Bachman, C. W. (2009). "The Origin of the Integrated Data Store (IDS): The First Direct-Access DBMS". *IEEE Annals of the History of Computing*. 31(4).

Bakibayev, N., T. Kociský, D. Olteanu, and J. Zavodny. (2013). "Aggregation and Ordering in Factorised Databases". *PVLDB*. 6(14).

Bakibayev, N., D. Olteanu, and J. Zavodny. (2012). "FDB: A Query Engine for Factorised Relational Databases". *PVLDB*. 5(11).

Beame, P., P. Koutris, and D. Suciu. (2017). "Communication Steps for Parallel Query Processing". *Journal of the ACM*. 64(6). DOI: 10.1145/3125644.

Bhattarai, B., H. Liu, and H. H. Huang. (2019). "CECI: Compact Embedding Cluster Index for Scalable Subgraph Matching". In: *SIGMOD*.

Bi, F., L. Chang, X. Lin, L. Qin, and W. Zhang. (2016). "Efficient Subgraph Matching by Postponing Cartesian Products". In: *SIGMOD*.

Blakeley, J. A., P.-A. Larson, and F. W. Tompa. (1986). "Efficiently Updating Materialized Views". *SIGMOD Record*. 15(2).

Boncz, P. A., M. Zukowski, and N. Nes. (2005). "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*.

Bonifati, A., G. Fletcher, H. Voigt, N. Yakovets, and H. V. Jagadish. (2018). *Querying Graphs*. Morgan & Claypool Publishers.

Cai, W., M. Balazinska, and D. Suciu. (2019). "Pessimistic Cardinality Estimation: Tighter Upper Bounds for Intermediate Join Cardinalities". In: *SIGMOD*.

Chen, J., Y. Huang, M. Wang, S. Salihoglu, and K. Salem. (2022). "Accurate Summary-Based Cardinality Estimation through the Lens of Cardinality Estimation Graphs". *PVLDB*. 15(8).

Codd, E. F. (1982). "Relational Database: A Practical Foundation for Productivity". *CACM*. 25(2).

Delobel, C. (1978). "Normalization and hierarchical dependencies in the relational data model". *TODS*. 3(3).

Dey, S., V. Cuevas-Vicenttín, S. Köhler, E. Gribkoff, M. Wang, and B. Ludäscher. (2013). "On implementing provenance-aware regular path queries with relational query engines". In: *EDBT/ICDT Workshops*.

Erling, O. and I. Mikhailov. (2009). "Virtuoso: RDF support in a native RDBMS". In: *Semantic web information management: a model-based perspective*. Springer. 501–519.

Fagin, R. (1977). "Multivalued dependencies and a new normal form for relational databases". *TODS*. 2(3).

Fan, J., A. G. S. Raj, and J. M. Patel. (2015). "The Case Against Specialized Graph Analytics Engines." In: *CIDR*.

Farias, B., C. Rojas, and D. Vrgoc. (2023). "Evaluating Regular Path Queries in GQL and SQL/PGQ: How Far Can The Classical Algorithms Take Us?" *CoRR*. abs/2306.02194.

Feng, X., G. Jin, Z. Chen, C. Liu, and S. Salihoğlu. (2022). "Kùzu Database Management System Source Code". URL: https://github.com/kuzudb/kuzu.

Feng, X., G. Jin, Z. Chen, C. Liu, and S. Salihoğlu. (2023). "Kùzu Graph Database Management System". In: *CIDR*.

Francis, N., A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor. (2018). "Cypher: An Evolving Query Language for Property Graphs". In: *ACM SIGMOD*.

Freitag, M., M. Bandle, T. Schmidt, A. Kemper, and T. Neumann. (2020). "Adopting Worst-Case Optimal Joins in Relational Database Systems". *PVLDB*. 13(12).

Graefe, G. (1994). "Volcano - An Extensible and Parallel Query Evaluation System". *TKDE*. 6(1).

Gupta, P., A. Mhedhbi, and S. Salihoglu. (2021). "Columnar Storage and List-based Processing for Graph Database Management Systems". *PVLDB*. 14(11).

Han, M., H. Kim, G. Gu, K. Park, and W. Han. (2019). "Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together". In: *SIGMOD*.

Hassan, M. S., T. Kuznetsova, H. C. Jeong, W. G. Aref, and M. Sadoghi. (2018). "Extending In-Memory Relational Database Engines with Native Graph Support". In: *EDBT*.

Huang, Z. and E. Wu. (2023). "Lightweight Materialization for Fast Dashboards Over Joins". *SIGMOD*. 1(4).

Idreos, S., M. L. Kersten, and S. Manegold. (2007). "Database Cracking". In: *Conference on Innovative Data Systems Research*.

ISO/IEC JTC 1/SC 32. (2024). "SQL/PGQ Standard". URL: https://www.iso.org/standard/79473.html.

JCC Consulting, Inc. (2024). "GQL Standard". URL: https://www.gqlstandards.org/.

Jin, G. and S. Salihoglu. (2022). "Making RDBMSs Efficient on Graph Workloads Through Predefined Joins". *PVLDB*. 15(5).

Jindal, A., P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. (2014). "Vertexica: your relational friend for graph analytics!" *Proceedings of the VLDB Endowment*. 7(13): 1669–1672.

Joglekar, M. and C. Ré. (2018). "It's All a Matter of Degree - Using Degree Information to Optimize Multiway Joins". *Theory of Computing Systems*. 62(4).

Kalinsky, O., Y. Etsion, and B. Kimelfeld. (2017). "Flexible Caching in Trie Joins". In: *EDBT*. Ed. by V. Markl, S. Orlando, B. Mitschang, P. Andritsos, K. Sattler, and S. Breß.

Kankanamge, C., S. Sahu, A. Mhedhbi, J. Chen, and S. Salihoglu. (2017). "Graphflow: An Active Graph Database". In: *SIGMOD*.

Kara, A., M. Nikolic, D. Olteanu, and H. Zhang. (2023). "F-IVM: Analytics over Relational Databases under Updates". *CoRR*. abs/2303.08583.

Khamis, M. A., H. Q. Ngo, C. Ré, and A. Rudra. (2016). "Joins via Geometric Resolutions: Worst Case and Beyond". *TODS*. 41(4).

Koschmieder, A. and U. Leser. (2012). “Regular path queries on large graphs”. In: *SSDBM*. Springer.

Koutris, P., S. Salihoglu, and D. Suciu. (2018). “Algorithmic Aspects of Parallel Data Processing”. *Foundations and Trends® in Databases*. 8(4).

Lapaugh, A. and C. Papadimitriou. (1984). “The even-path problem for graphs and digraphs”. *Networks*. 14.

Leeuwen, W. v., T. Mulder, B. van de Wall, G. Fletcher, and N. Yakovets. (2022). “AvantGraph Query Processing Engine”. *PVLDB*. 15(12).

Leis, V., B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. (2018). “Query Optimization through the Looking Glass, and What We Found Running the Join Order Benchmark”. *VLDBJ*. 27(5).

Lin, C., B. Mandel, Y. Papakonstantinou, and M. Springer. (2016). “Fast In-Memory SQL Analytics on Typed Graphs”. In: *ICDE*.

Losemann, K. and W. Martens. (2013). “The complexity of regular expressions and property paths in SPARQL”. *TODS*. 38(4).

Malewicz, G., M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. (2010). “Pregel: a system for large-scale graph processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. 135–146.

Memgraph Ltd. (2023). “MemGraph”. URL: https://memgraph.com/.

Mendelzon, A. O. and P. T. Wood. (1995). “Finding regular simple paths in graph databases”. *SIAM J. Comput.* 24(6).

Mendelzon, A. O. and P. T. Wood. (1989). “Finding Regular Simple Paths in Graph Databases”. *SIAM J. Comput.* 24: 1235–1258.

Mhedhbi, A. (2023). “GraphflowDB: Scalable Query Processing on Graph-Structured Relations”. *PhD thesis*. URL: http://hdl.handle.net/10012/19981.

Mhedhbi, A., C. Kankanamge, and S. Salihoglu. (2021). “Optimizing One-time and Continuous Subgraph Queries using Worst-case Optimal Joins”. *TODS*. 46(2).

Mhedhbi, A. and S. Salihoglu. (2019). “Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins”. *PVLDB*. 12(11).

Neo4j, Inc. (2023a). “Neo4j”. URL: https://neo4j.com/.

Neo4j, Inc. (2023b). "Neo4j Record Design". URL: https://neo4j.com/developer/kb/understanding-data-on-disk/.

Neumann, T. and M. J. Freitag. (2020). "Umbra: A Disk-Based System with In-Memory Performance". In: *CIDR*.

Neumann, T. and G. Weikum. (2010). "The RDF-3X engine for scalable management of RDF data". In: *VLDBJ*.

Ngo, H. Q., D. T. Nguyen, C. Re, and A. Rudra. (2014). "Beyond Worst-Case Analysis for Joins with Minesweeper". In: *PODS*.

Ngo, H. Q., E. Porat, C. Ré, and A. Rudra. (2012). "Worst-case Optimal Join Algorithms: [Extended Abstract]". In: *PODS*.

Ngo, H. Q., C. Ré, and A. Rudra. (2013). "Skew strikes back: new developments in the theory of join algorithms". In: *SIGMOD Rec.*

Nguyen, V.-Q. and K. Kim. (2017). "Efficient regular path query evaluation by splitting with unit-subquery cost matrix". *IEICE Transactions on Information and Systems*. 100(10).

Nikolic, M., H. Zhang, A. Kara, and D. Olteanu. (2020). "F-IVM: Learning over Fast-Evolving Relational Data". In: *ACM SIGMOD*.

Olteanu, D. and M. Schleich. (2016). "Factorized Databases". *SIGMOD Rec.* 45(2).

Olteanu, D. and J. Zavodny. (2015). "Size Bounds for Factorised Representations of Query Results". *TODS*. 40(1).

Peng, Y., Y. Zhang, X. Lin, L. Qin, and W. Zhang. (2020). "Answering billion-scale label-constrained reachability queries within microsecond". *PVLDB*. 13(6).

Raasveldt, M. and H. Mühleisen. (2019a). "DuckDB: An Embeddable Analytical Database". In: *SIGMOD*.

Raasveldt, M. and H. Mühleisen. (2019b). "DuckDB: an Embeddable Analytical Database". In: *SIGMOD*.

Sahu, S., A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu. (2020). "The Ubiquity of Large Graphs and Surprising Challenges of Graph Processing: Extended Survey". 13(12).

Schleich, M. and D. Olteanu. (2020). "LMFAO: An Engine for Batches of Group-by Aggregates: Layered Multiple Functional Aggregate Optimization". 13(12).

Shun, J. and G. E. Blelloch. (2013). “Ligra: a lightweight graph processing framework for shared memory”. In: *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. 135–146.

Silberschatz, A., H. Korth, and S. Sudarshan. (2005). *Database Systems Concepts*. 5th ed. McGraw-Hill, Inc.

Smagulova, A. and A. Deutsch. (2021). “Vertex-centric Parallel Computation of SQL Queries”. In: *Proceedings of the 2021 International Conference on Management of Data*. 1664–1677.

Sun, S., X. Sun, Y. Che, Q. Luo, and B. He. (2020). “Rapidmatch: A holistic approach to subgraph query processing”. *Proceedings of the VLDB Endowment*. 14(2): 176–188.

Tatarinov, I., S. D. Viglas, K. Beyer, J. Shanmugasundaram, E. Shekita, and C. Zhang. (2002). “Storing and querying ordered XML using a relational database system”. In: *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*. 204–215.

Then, M., M. Kaufmann, F. Chirigati, T.-A. Hoang-Vu, K. Pham, A. Kemper, T. Neumann, and H. T. Vo. (2014). “The More the Merrier: Efficient Multi-source Graph Traversal”. *PVLDB*. 8(4).

Tigergraph. (2023). “TigerGraph”. URL: https://www.tigergraph.com/.

Valduriez, P. (1987). “Join Indices”. *ACM TODS*. 12(2).

Valstar, L. D., G. H. Fletcher, and Y. Yoshida. (2017). “Landmark Indexing for Evaluation of Label-Constrained Reachability Queries”. In: *SIGMOD*.

Veldhuizen, T. L. (2012). “Leapfrog Triejoin: a worst-case optimal join algorithm”. *CoRR*. abs/1210.0481.

Veldhuizen, T. L. (2013). “Incremental Maintenance for Leapfrog Triejoin”. *CoRR*. abs/1303.5313.

W3C. (2024). “SPARQL Standard”. URL: https://www.w3.org/TR/sparql11-query/.

Wang, Y., A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens. (2016). “Gunrock: A high-performance graph processing library on the GPU”. In: *Proceedings of the 21st ACM SIGPLAN symposium on principles and practice of parallel programming*. 1–12.

Wang, Y. R., M. Willsey, and D. Suciu. (2023). “Free Join: Unifying Worst-Case Optimal and Traditional Joins”. In: *SIGMOD*.

Wolde, D. ten, T. Singh, G. Szarnyas, and P. Boncz. (2023). "DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS". In: *CIDR*.

Xirogiannopoulos, K. and A. Deshpande. (2017). "Extracting and analyzing hidden graphs from relational databases". In: *SIGMOD*.

Xirogiannopoulos, K., V. Srinivas, and A. Deshpande. (2017). "Graphgen: Adaptive graph processing using relational databases". In: *GRADES-NDA*.

Yakovets, N., P. Godfrey, and J. Gryz. (2013). "Evaluation of SPARQL Property Paths via Recursive SQL". In: *Alberto Mendelzon Workshop on Foundations of Data Management*.

Yakovets, N., P. Godfrey, and J. Gryz. (2016a). "Query Planning for Evaluating SPARQL Property Paths". In: *SIGMOD*.

Yakovets, N., P. Godfrey, and J. Gryz. (2016b). "Query planning for evaluating SPARQL property paths". In: *SIGMOD*.

Yan, D., Y. Bu, Y. Tian, A. Deshpande, *et al.* (2017). "Big graph analytics platforms". *Foundations and Trends® in Databases*. 7(1-2): 1–195.

Yannakakis, M. (1981). "Algorithms for Acyclic Database Schemes". In: *PVLDB*.

Yu, J. X. and J. Cheng. (2010). "Graph Reachability Queries: A Survey". In: *Managing and Mining Graph Data*. Springer US.

Zhu, J., N. Potti, S. Saurabh, and J. M. Patel. (2017). "Looking ahead makes query plans robust: Making the initial case with in-memory star schema data warehouse workloads". *PVLDB*. 10(8).

Zou, L., M. T. Özsu, L. Chen, X. Shen, R. Huang, and D. Zhao. (2014a). "gStore: a graph-based SPARQL query engine". *The VLDB journal*. 23: 565–590.

Zou, L., K. Xu, J. X. Yu, L. Chen, Y. Xiao, and D. Zhao. (2014b). "Efficient Processing of Label-constraint Reachability Queries in Large Graphs". *Information Systems*. 40.

Zukowski, M., M. van de Wiel, and P. A. Boncz. (2012). "Vectorwise: A Vectorized Analytical DBMS". In: *ICDE*.