DPQP: A D-representation-based Pipelined Factorized Query Processor for Graph Database Management Systems

by

Xiyang Feng

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

Statement of Contribution

This thesis is a joint work with Amine Mhedhbi, Annie Zhou, and Semih Salihoglu.

Xiyang Feng, Amine Mhedhbi, Annie Zhou, and Semih Salihoglu all contribute to the design and discussion of the tuple structure, operators, and plans of the factorized query processor presented in this thesis. Xiyang Feng implemented the prototype of factorized query processor. Xiyang Feng designed and conducted all experiments in this thesis. Xiyang Feng and Semih Salihoglu analyzed experiments results. Xiyang Feng wrote this thesis. Amine Mhedhbi and Semih Salihoglu reviewed the thesis.

Abstract

Factorized databases utilize factorized data representations during query processing to obtain more compact final query results and faster runtimes for queries with many-to-many joins. We revisit this technique in the context of graph database management systems (GDBMSs) whose common workloads are large joins with many-to-many relationships on graph-structured data. We first review the theory of factorized databases and classic flat intermediate tuple structure in traditional pipelined GDBMSs. We then present our tuple representation which mimics factorized representations and can be easily integrated into existing query processors. We further describe how to cache sub-query results with this factorized tuple structure through a static dependency analysis of the query. We have integrated our factorized query processor into GraphflowDB, an in-memory GDBMS. Compared to the original version of GraphflowDB, whose processor is not fully factorized, query plans in our processor can be orders of magnitude faster and produce orders of magnitude smaller result sizes.

Acknowledgements

First, I want to thank my supervisor, Prof. Semih Salihoglu, for his constant support in the past two years. This thesis would not be possible without his advice and help. Working with Semih has been a pleasant experience and I am very grateful to join his research group.

Second I would like to thank my friend and colleague Amine, for collaborating with me on this project and offering help whenever I need them. Amine brought me valuable insights on factorization when I joined the group and taught me many skills in coding, research, and presentation. I am also grateful to my friends Guodong and Rex who helped me during my time here.

I also want to thank my thesis readers, Prof. Xi He and Prof. Ken Salem, for spending the time reading my thesis, attending my presentation and offering valuable feedback.

Finally, I want to thank my parents, Shihong Feng and Liping Xi, for their love and support.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Graph database management systems (GDBMSs) are systems that use the property graph data model where entities are stored as nodes and relationships between entities are represented as edges. The property graph data model is intuitive for applications such as social networks, fraud detection, and friend recommendation. The typical workloads on these applications are read-heavy graph pattern matching queries, for example, finding cycles in a money flow network which could be a potential fraud action [22]. GDBMSs are primarily designed to power applications whose workloads contain large joins over graph-structured data, which often contain many-to-many relationships. Similar to traditional relational database management systems (RDBMSs), query processors of existing GDBMSs represent and process relations as flat tuples. Such query processing architectures were not designed to efficiently handle workloads with large many-to-many joins that can lead to an explosion in the size of intermediate and final relations [5, 14, 19].

The problem of large intermediate results is very common and arises on even very simple query templates. Consider as examples the two queries in Figure 1.2. The query in Figure 1.2a is a 2-path query that asks for all money flows that an account represented by vertex v_3 is facilitating. The query in Figure 1.2b asks for all 4-hop money flow paths from an account with ID v_1. Both of these queries have output sizes of k^2 on the example relation `TRANSFER` shown as a graph in Figure 1.1. One can see however that the answer to the former query is contained in the subgraph around v_3, which only contains $2k$ edges, and the answer to the latter is contained in the subgraph between v_1 and v_5 which only contains $4k$ edges. Intuitively, if a system could represent intermediate relations as subgraphs and reuse them, it could process queries without generating large intermediate results.

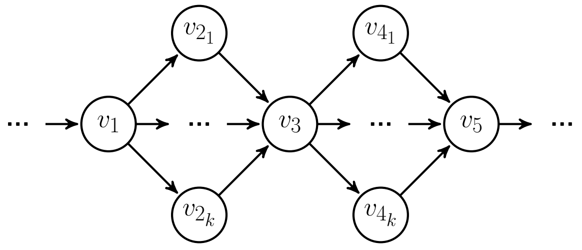
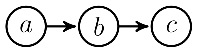
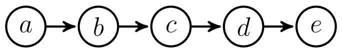


Figure 1.1: Instance of a `TRANSFER` relation where each edge represents a money transfer between two accounts (`ACC`) entities from v$_i$ to v$_j$.



$WHERE\ b.ID = v_3$

(a) A 2-path query denoted Q$_{2H}$.



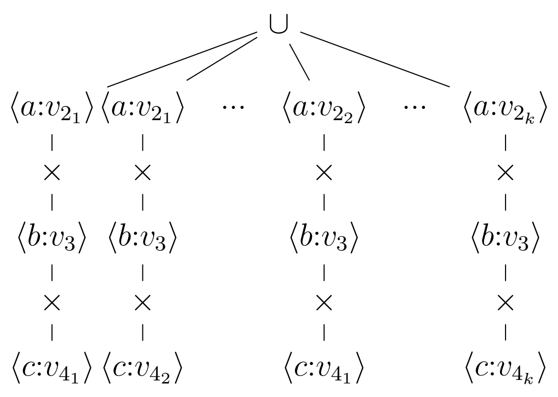
$WHERE\ a.ID = v_1$

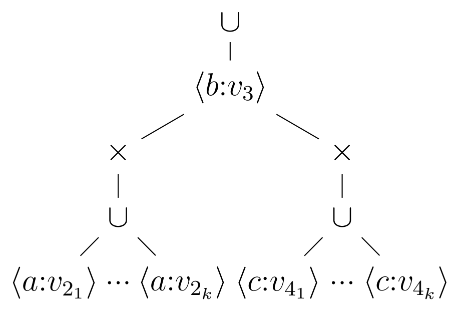
(b) A 2-path query denoted Q$_{4H}$.

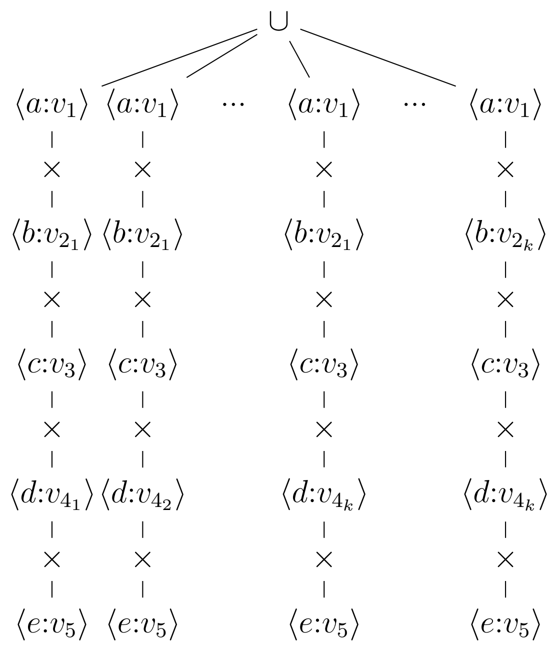
Figure 1.2: Two money flow path queries.

Recent theory of factorized databases has laid the foundation for DBMSs to address the intermediate results and final size explosion problem. Instead of representing relations as flat tuples and similar to the above intuition, factorized databases represent relations as tries that are unions of Cartesian products of sub-relations or singleton values [5]. Figure 1.3 shows factorized representations of the outputs of each query from Figure 1.2. The representation used in Figure 1.3b for the 2-hop query is called an *f-representation* [5] which represents the 2-hop path as the Cartesian product of three sets: incoming edges of v_3 (i.e. $v_{2_1}, ..., v_{2_k}$), $\{v_3\}$, and outgoing edges of v_3 (i.e. $v_{4_1}, ..., v_{4_k}$). This captures the intuition that once v_3 is assigned to variable b, the values of a and c in the output are independent. The representation used in Figure 1.3d is called a *d-representation* [19], which expands f-representations by storing repeated sub-relations once and reusing them. In Figure 1.3d, the sub-relation of the 2-hop path from v_3 to v_5 is stored once and reused k times. All v$_3$ nodes refer to the same node in a dashed box. Figures 1.3a and 1.3c show the flat representations for 2-hop and 4-hop path queries respectively where the set of tuples are represented as the union of the Cartesian product of each singleton value for all variables.

The theory of factorized databases has established that f-representations of relations can be asymptotically smaller than flat representations and d-representations can further be asymptotically smaller than f-representations [19]. In addition, when computing relational queries that have conditionally independent variables, it is possible to perform computations on succinct factorized representations to obtain asymptotically faster run


$\cup$
$\langle a{:}v_{2_1}\rangle$
$\langle a{:}v_{2_2}\rangle$
$\langle a{:}v_{2_k}\rangle$
$\times$
$\langle b{:}v_3\rangle$
$\langle c{:}v_{4_1}\rangle$
$\langle c{:}v_{4_2}\rangle$
$\langle c{:}v_{4_k}\rangle$
$\cdots$
(a) Q_{2H} results in a flat representation.


$\cup$
$\langle b{:}v_3\rangle$
$\times$
$\langle a{:}v_{2_1}\rangle$
$\langle a{:}v_{2_k}\rangle$
$\langle c{:}v_{4_1}\rangle$
$\langle c{:}v_{4_k}\rangle$
$\cdots$
(b) Q_{2H} results in an f-representation.


$\cup$
$\langle a{:}v_1\rangle$
$\times$
$\langle b{:}v_{2_1}\rangle$
$\langle b{:}v_{2_k}\rangle$
$\langle c{:}v_3\rangle$
$\langle d{:}v_{4_1}\rangle$
$\langle d{:}v_{4_2}\rangle$
$\langle d{:}v_{4_k}\rangle$
$\langle e{:}v_5\rangle$
$\cdots$
(c) Q_{4H} results in a flat representation.

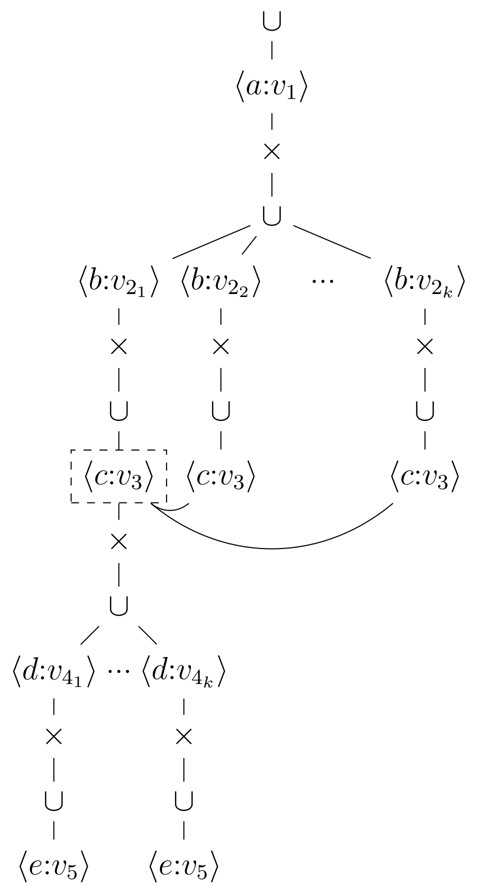

$\cup$
$\langle a{:}v_1\rangle$
$\times$
$\langle b{:}v_{2_1}\rangle$
$\langle b{:}v_{2_2}\rangle$
$\langle b{:}v_{2_k}\rangle$
$\langle c{:}v_3\rangle$
$\langle d{:}v_{4_1}\rangle$
$\langle d{:}v_{4_k}\rangle$
$\langle e{:}v_5\rangle$
$\cdots$
(d) Q_{4H} results in a d-representation.

Figure 1.3: Example flat and factorized query results.

times [5, 19]. However, the only factorized query processor implementation in prior work, which is described in the reference FDB system [5], has several shortcomings for integrating into existing GDBMSs.

First, FDB's query processor is based on operators that perform transformations over entire tries. In other words, FDB indexes all input relations as tries and its plans consist of a linear sequence of operators that take as input tries and output tries, starting with tries that represent the input relations. The sequences of trie transformations are decided according to join plans called *f-trees*, which represent the factorization structure of the output of each operator. One cannot integrate this approach by modifying the traditional pipelined query processors of existing systems, whose operators operate on a single or a block of tuples. In addition, this approach is memory intensive as tries that operators use are full materialization of intermediate results. Such materialization is avoided by pipelining in traditional processors. Second, FDB's processor is based on f- instead of d-representations and can leave important performance behind when queries can benefit from reusing result of sub-relations. Such queries are common in GDBMS workloads. Finally, FDB's processor is designed for an in-memory RDBMS, and does not exploit GDBMS-specific storage and workload characteristics, such as the list-based data storage layout of GDBMSs or the integer ID-based joins that are prevalent in graph workloads.

1.1 Contributions

This thesis describes a factorized query processor that addresses these shortcomings. First, our approach is based on reusing the pipelined operators of existing GDBMSs with minimal modifications and implements a new operator, specifically, a factorized representation construction operator named `FGroup`, that performs computation on factorized representation. Similar to existing query processors, every operator, except `FGroup`, performs computation on a block of tuples instead of tries. Second, apart from f-representations, our approach also utilizes d-representations, so caches and reuses results of sub-queries whenever possible. We add a d-representations caching layer in `FGroup` and modify our join operator to call one of two possible children operators depending on whether the remainder of the sub-query they are evaluating has been cached or not for a prefix of tuple values. Similar to the *list-based query processor* (`LBQP`) for GDBMSs described in reference [9], our processor also exploits list-based storage of GDBMSs to do late materialization of adjacency lists when possible.

We have integrated our query processor into the GraphflowDB in-memory GDBMS [11, 13] and focus on select-join queries. In graph terminology, this corresponds to subgraph pattern matching followed by arbitrary predicates on the nodes, edges, or node and edge properties of the matched pattern. We focus on query processing and rely on GraphflowDB's default optimizer to pick a join order and adopt FDB's optimization algorithm to generate an f-tree that describes the factorization structure of the given join order. We demonstrate that our approach outperforms the previous GraphflowDB with `LBQP` by orders of magnitude on many queries in the LDBC benchmark as well as a suite of micro-benchmark queries. To refer to our processor with a single term throughout the paper, we call it `DPQP`, for **d**-*representation-based* **p***ipelined* **q***uery* **p***rocessor*.

1.2 Outline

The outline for the rest of this thesis is as follows.

- Chapter 2 reviews the definition of f-/d-representations as well as f-/d-trees which describe the structure of factorized representations. This chapter also reviews `LBQP` upon which `DPQP` is built.
- Chapter 3 describes a trie-like intermediate tuple structure used in `DPQP`. The tuple structure mimics f-representations and, if certain constraints are satisfied, could also mimic d-representations.
- Chapter 4 describes operators and query plans in `DPQP`. Specifically, we describe major operators inherited from `LBQP` and introduce a new operator named `FGroup` that constructs factorized representations. We also present how to use `FGroup` to obtain a `DPQP` plan and how to integrate caching into it.
- Chapter 5 presents experiments evaluating `DPQP`, including the effectiveness of utilizing d-representations in terms of both runtimes and size of final results.
- Chapters 6 and 7 cover related work and conclude, respectively.

Chapter 2

Background

We provide background on the foundations of factorized databases, covering factorized representations of relations and select-join query processing. We also give an overview of GraphflowDB's LBQP, which we modified to develop DPQP for select-join queries. We use the Cypher language [8] developed for the property graph data model. In this model, entities are represented by vertices, relationships are represented by edges, and attributes are represented by arbitrary key-value properties on vertices and edges. Both entities and relationships have unique ID attributes. Queries 2.1 and 2.2 are written in Cypher as shown below. a and b are aliases for the ACCOUNT relation and represent query vertices. Each t_i is an alias for the TRANSFER relation and represents a query edge. Query vertices and query edges are referred to as query variables or simply variables for short.

Query 2.1: 2-hop money transfer path query.

```
MATCH (a:ACC)–[t1:TRANSFER]→(b:ACC)–[t2:TRANSFER]→(c:ACC)
WHERE b.ID = v3
RETURN a.ID, b.ID, c.ID
```

Query 2.2: 4-hop money transfer path query.

```
MATCH (a:ACC)–[t1:TRANSFER]→(b:ACC)–[t2:TRANSFER]→(c:ACC),
      (c:ACC)–[t3:TRANSFER]→(d:ACC)–[t4:TRANSFER]→(e:ACC)
WHERE a.ID = v1
RETURN a.ID, b.ID, c.ID, d.ID, e.ID
```

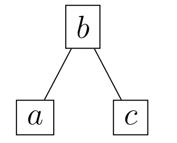
2.1 Factorized Representations of Relations

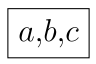
Definition 1 (F-representation [19]) *An f-representation of a relation is a relational algebra expression over a set of attributes $\mathcal{S}$ that consists of singleton values, unions, and Cartesian products. Formally the expression can be one of the following: (i) $\emptyset$, representing the empty relation; (ii) $\langle A : a \rangle$, a singleton tuple with value a for variable A; (iii) (E), where E is an f-representation; (iv) $E_1 \cup ... \cup E_n$; or (v) $E_1 \times ... \times E_n$, where E_i are f-representations.*

F-representations are a complete representation system, i.e., any relation R has at least one f-representation. Specifically, the flat representation is a straightforward f-representation that represents each row as a Cartesian product of each singleton value in each of its columns and unions each row. However, in some cases relations can have more compact f-representations. The structure of an f-representation can be represented at variable level by a trie structure called *factorization trees* (f-trees).

Definition 2 (F-tree [19]) *An f-tree over a set of attributes $\mathcal{S}$ is a rooted forest with each node labeled by a non-empty subset of $\mathcal{S}$ such that each attribute of $\mathcal{S}$ occurs in exactly one node. The structure of f-representations E adheres to an f-tree $\mathcal{T}$ if it satisfies the following: (i) If $\mathcal{T}$ is empty, then $E = \emptyset$; (ii) If $\mathcal{T}$ is a single node labelled by $\mathcal{S} = \{A_1, A_2, ..., A_k\}$, then $E = \cup_i(\langle A_1 : a_i \rangle \times \cdots \times \langle A_k : a_i \rangle)$; (iii) if $\mathcal{T}$ consists of a root with label A_i and a non-empty forest $\mathcal{U}$ of children, then $E = \cup_{a_i} \langle A_i : a_i \rangle \times E_{a_i}$, where E_{a_i} is an f-representation over $\mathcal{U}$; and (iv) If $\mathcal{T}$ is a forest of trees $\mathcal{T}_1, ..., \mathcal{T}_k$, then $E = E_1 \times ...E_k$, where E_i is an f-representation over $\mathcal{T}_i$.*

Intuitively, an f-representation adheres to an f-tree $\mathcal{T}$ if its structure mimics the hierarchical structure of $\mathcal{T}$. As a shorthand, we label a node with the query variables i.e., query vertices and edges, to indicate that the node is labelled by the variable's attributes/properties mentioned in any of the query clauses. For instance, a node labelled by `a.p`$_1$ and `a.p`$_2$ where a is a query vertex and p_1 and p_2 are properties can be replaced with a if p_1 and p_2 are the only attributes of a as part of the query For example, the f-representation for the output relation of our 2-hop path query adheres to the f-tree in Figure 2.1a. Similarly, the flat representation of any relation R over $\mathcal{S}$ adheres to any f-tree $\mathcal{T}$ is a single node containing all the query attributes. One possible f-tree for flat-representation is shown in Figure 2.1b. However, an arbitrary relation R over the schema $\mathcal{S}$ does not necessarily have an f-representation that adheres to an arbitrary f-tree $\mathcal{T}$ over $\mathcal{S}$. For example, the output


(a) f-tree $\mathcal{T}_1$


(b) f-tree $\mathcal{T}_2$

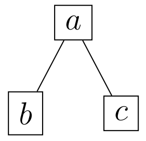

(c) f-tree $\mathcal{T}_3$

Figure 2.1: Different f-trees for query Q_{2H} in Figure 1.2a.

relation of our 2-path query does not adhere to the f-tree in Figure 2.1c. We will explain how to determine whether a relation R adheres to an f-tree $\mathcal{T}$ momentarily.

When a relation R is the output of a select-join query Q, i.e., $R = Q(G)$, one can find f-trees that R adheres to by analyzing the conditional independence relationships between the variables used in Q. We first define the notion of dependence directly quoting the definition from reference [19].

Definition 3 (Dependence [19]) *Two disjoint groups of attributes $\mathcal{A}$ and $\mathcal{B}$ of a relation R are called* independent conditioned *on another group of attributes $\mathcal{C}$, disjoint with $\mathcal{A}$ and $\mathcal{B}$, if R is a natural join $R_{\mathcal{A}} \bowtie_{\mathcal{C}} R_{\mathcal{B}}$ of two relations $R_{\mathcal{A}}$ and $R_{\mathcal{B}}$ with attributes including $\mathcal{A}$ and $\mathcal{B}$, respectively. $\mathcal{A}$ and $\mathcal{B}$ are independent if they are independent conditioned on the empty set. If two attributes are not conditionally independent, they are dependent.*

In the queries of GDBMSs, the joins are represented by subgraph queries, e.g., expressed in the MATCH clause of the Cypher query language [8]) with possibly additional join predicates across the query vertices and edges or their properties, e.g., expressed in the WHERE clause of Cypher. It is well known that these queries can equivalently be expressed as joins over binary relations that contain the edges with a particular edge label as source and destination ID pairs. Ignoring the additional join predicates, if we focus on a subgraph query Q, the above definition indicates that *in the output relation of Q*, a set of query vertices $\mathcal{A}$ and $\mathcal{B}$ are independent conditioned on $\mathcal{C}$ (disjoint from $\mathcal{A}$ and $\mathcal{B}$), if the sub-queries $Q_{\mathcal{A}}$ and $Q_{\mathcal{B}}$ of Q projected onto $\mathcal{A}$ and $\mathcal{B}$ are sub-queries that do not share a direct query edge, and $\mathcal{C}$ is the remaining set of query vertices in Q. Figure 1.2a gives an example subgraph query, where a and c are independent conditioned on b. Instead the same set of variables are not conditionally independent in Figure 2.2, as they share direct edges.

As also described in reference [19], the dependency information can be inferred statically from Q. In particular: (1) Any pair of variables in a query Q that is part of the same query edge is dependent.[1] (2) Any pair of variables in a query Q that is part of the same predicate is dependent. A third rule, which is omitted exists for queries that contain projections and can be found in reference [19]. Reference [19] has further shown that one can determine whether or not the result $Q(G)$ of a query Q adheres to an f-tree $\mathcal{T}$ by a simple rule called the *path condition*: any set of dependent variables need to be on the same root-to-leaf path on $\mathcal{T}$. Equivalently, variables that are *not* on the same root-to-leaf path must be conditionally independent on their least common ancestor a_i and the set of variables on the path from the root to a_i. For example, the f-tree in Figure 2.1a satisfies the path condition, and so can factorize the output of our 2-path query. Instead, the f-tree in Figure 2.1c does not satisfy the path condition, because b and c are dependent but are on different root-to-leaf paths, so $Q(G)$ cannot be factorized to adhere to this f-tree.

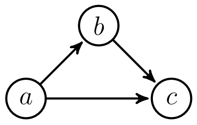


Figure 2.2: A triangle query

D-representations generalize f-representations to use named expressions (aka definitions) that allow re-using repeated factorized representations. Formally:

Definition 4 (D-representation [19]) *A d-representation is a set of named expressions* $\{N_1 = D_1, ..., N_k = D_k\}$*, where each* N_i *is a unique symbol and each* D_i *is a union of Cartesian products that can use other named expressions.*

For example, in the representations in Figure 1.3d for the output of the 4-hop money flow query, the sub-relation of the 2-hop path from v_3 to v_5 would be a named expression that is reused k times. Similarly, we will be interested in d-representations whose structures can be expressed at the variable level by an extension of f-trees called d-trees:

Definition 5 (D-tree [19]) *A d-tree* $\mathcal{T}^{\uparrow}$ *over a set* $\mathcal{S}$ *of variables is an f-tree* $\mathcal{T}$ *where each node* A_i *is labeled with the subset of the ancestors of* A_i *that depend on any of the descendants of* A_i*. We refer to these subset of variables as* dependency labels*.*

We will in particular be interested in nodes in $\mathcal{T}^{\uparrow}$ that have empty dependency labels, and draw them as dashed nodes. These are the variables whose children have no dependency on their ancestors. When we perform joins starting from the root to the leaves,

[1]Similarly a query node or query edge variable and any variable that refers to an attribute of these nodes and edges are dependent.

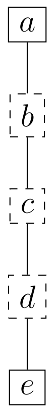


Figure 2.3: A d-tree $\mathcal{T}^{\uparrow}$ for Query 1.2b.

these variables and their children form sub-queries whose results we can group and reuse. For example, the results of our 4-hop query in Figure 1.3d uses the d-tree from Figure 2.3. Variable c here has an empty dependency label, so for different matches of c we can give a name, e.g., $D_{c=v_3}$ to the result of the sub-query for $(c)\rightarrow(d)\rightarrow(e)$, and for each prefix tuple c value matches v_3, we can avoid computing this sub-query and reuse $D_{c=v_3}$. In this d-tree in fact all of the variables have empty dependency labels, so we do similar grouping for b and d as well though these do not result in further compaction in the output size. We draw the root always as a non-dashed circle because values that match the root variable will be unique, as the root is the first variable in the d-tree. Therefore, we could not reuse results grouped by the root variable. We also draw the leaves as non-dashed circles as there are no further sub-query results to group after a leaf variable. As a result, caching and reusing with d-representations requires a d-tree of at least a height of 2 such that there are variables with a depth d that satisfies $0 < d < height$.

We end this section with a note on several size bounds for sizes of the f-representation from reference [19]. Reference [19] has shown that for any select-join query Q over a database D (so an input graph G in our context), the worst case size of its d- and f-representations of its output can be tightly characterized by two exponents $O(|D|^{s^{\uparrow}(Q)})$ and $O(|D|^{s(Q)})$, respectively, such that $s^{\uparrow}(Q) \leq s(Q)$. $s(Q)$ is further at most the fractional edge cover $\rho(Q)$ of Q, which is known as the AGM bound [3], and represents the worst-case flat output size. Reference [19] has also shown that there are queries in which $s^{\uparrow}(Q) < s(Q)$ and $s(Q) < \rho(Q)$, indicating asymptotic size differences between d-, f- and flat representations of output results of queries.

2.2 List-based Query Processor

2.2.1 Volcano-style Query Processor

Traditional Volcano-style query processors adopt flat tuple structure and perform tuple-at-a-time processing. Figure 2.4 shows a query plan, akin to the left-deep plan in RDBMS, that evaluates Query 2.3 which asks for all posts with length greater than 50 liked by friends of Alice. Figure 2.5a shows the flat tuple structure adhering to the given plan where each variable is represented as a flat value. Such flat representation, in the context of factorization, is a Cartesian product of each singleton value in the tuple. One benefit of Volcano-style processing is that it is efficient in terms of how much data is copied to intermediate tuples. Imagine $Scan(a)$ matches a single value a_1 which extends to k_1 different b values each of which further extends to k_2 different c values. Although the final output contains $k_1 \times k_2$ tuples, a_1 is only copied once during processing. Such saving of data copying is more significant for longer join paths. On the other hand, it is also well studied that Volcano-style query processors do not achieve good CPU cache locality since the processing of two consecutive nodes is interleaved with many other function calls.

Query 2.3: 2-hop path query.

```
MATCH (a:PERSON)−[:KNOWS]→(b:PERSON),
      (b:PERSON)−[:LIKES]→(c:POST)
WHERE a.name = Alice AND c.length > 50
```

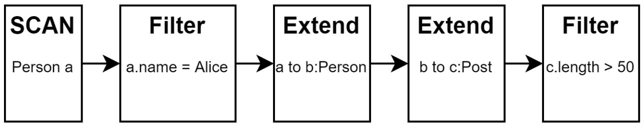
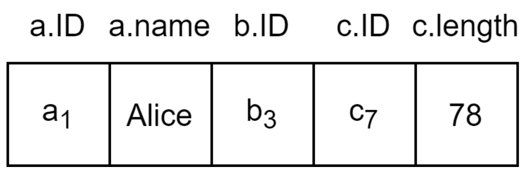


Figure 2.4: Left deep plan for the Query 2.3.

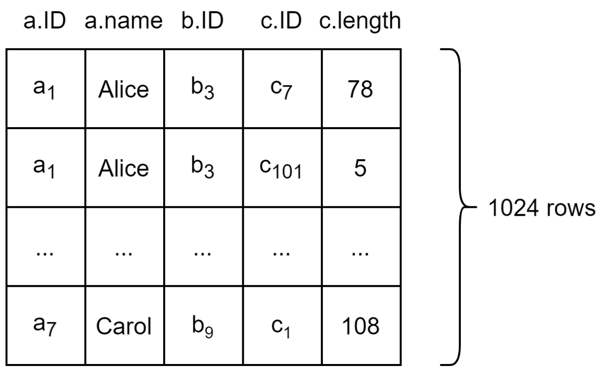
2.2.2 Block-based Query Processor

Block-based query processors aim to improve CPU cache locality by having operators process blocks of data at a time. The blocks contain a fixed number of, e.g. 1024, tuples per block. Figure 2.5b shows the intermediate tuple structure of block-based processor for the plan in Figure 2.4. One could easily reason that the set of tuples represented by a block is



a.ID	a.name	b.ID	c.ID	c.length
a_1	Alice	b_3	c_7	78

(a) Flat tuple structure



a.ID	a.name	b.ID	c.ID	c.length
a_1	Alice	b_3	c_7	78
a_1	Alice	b_3	c_{101}	5
...	...	...	...	...
a_7	Carol	b_9	c_1	108

(b) Block-based tuple structure

Figure 2.5: Flat and block-based tuple structure examples.

the union of all rows and each row is a Cartesian product of all its singleton values. Therefore, intermediate tuples in blocked-based processor have the same factorization structure as in Volcano-style processor.

One shortcoming of block-based processing is that it may perform a lot of data copying for queries with many-to-many joins. Consider example plan in Figure 2.4 in a block-based processor. a_1 would be copied once for each different (b, c) suffix as shown in Figure 2.5b. Another drawback is that block-based processors do not exploit data layout that is common in GDBMSs. Specifically, the materialization of node IDs contained in an adjacency list can be avoided since it is already stored consecutively in GDBMSs.

2.2.3 List-based Query Processor

Reference [9] introduced `LBQP` for GDBMSs that extends block-based processors while addressing the two shortcomings mentioned above. `LBQP` introduces a new data structure named list group which consists of a key variable, usually a node ID variable, and other variables that have one-to-one relationships with the key variable. Values of each variable is stored in list-like data structure named value vector. Variables in the same list group have a shared state which contains a currentIdx and a filter mask. If currentIdx is set to a non-negative integer i, the list group represents the i-th flat tuple in the list group. We refer to these list groups as *flat list groups*. Otherwise, list groups represent a list of values and are referred as *unflat list groups*. Filter mask is used to identify which values are valid since predicates might be applied on variables in a list group. Intermediate tuples in `LBQP`

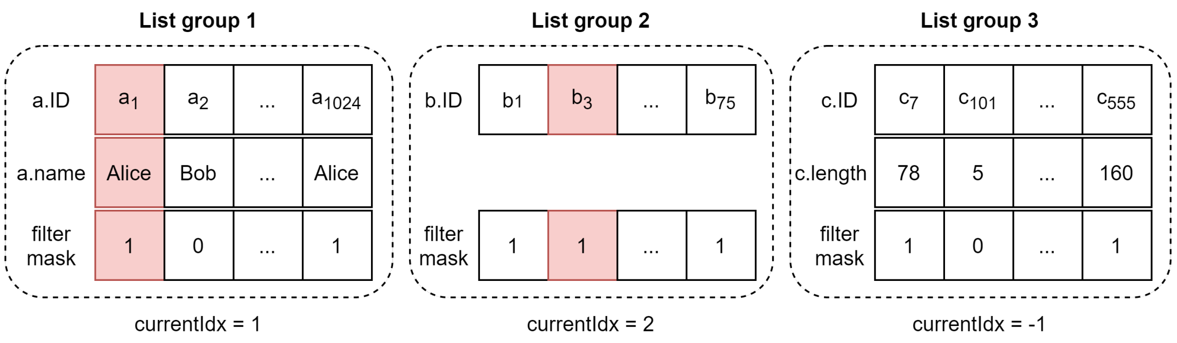


Figure 2.6: List-based tuple structure

is a collection of list groups. We will explain momentarily how this representation avoids the data repetition problem of block-based processors under many-to-many joins.

`LBQP` also improves on traditional block-based processors by replacing fixed-length blocks with variable-length adjacency lists. This allows `LBQP` to avoid materializing neighbor node IDs by simply keeping pointers to adjacency lists storing these node IDs. This optimization significantly reduces data copying, especially on star queries.

Example 1 Figure 2.6 shows the intermediate tuple with three different list groups that adheres to the plan in Figure 2.4. Scan(a) first scans a sequence of node IDs for variable a, i.e. $a_1, a_2, ..., a_{1024}$ and their corresponding name attributes. $a.ID$ and $a.name$ is written in the same list group since $a.name$ has a one-to-one relationship with $a.ID$ and thus can share the same state. A filter is then applied to each tuple in $list\ group_1$ whose evaluation result is written to the filter mask field. A join operator iteratively flattens the list group it extends from by updating the currentIdx field and writes to a new list group. In our example, jo_1 first sets $currentIdx = 1$ in **list group 1** and extends from a_1 to a set of b neighbors that are stored in **list group 2**. Similarly, jo_2 computes a set of c values, together with their length attributes, for b_3 that is under $currentIdx = 2$ of **list group 2**. The sizes of $b.id$ and $c.id$ are determined by the size of forward adjacency lists of a_1 and b_3 respectively and do not necessarily to be a fixed length. The last **list group 3** remains unflat since no further join extends from it. Finally, filter $c.length > 50$ is applied on c.length field in unflat **list group 3** whose filter mask is updated accordingly.

Recall that both Volcano-style and block-based processors represent tuples as a Cartesian product of singleton values. `LBQP`, however, represents tuples as the Cartesian product of all list groups, e.g. the tuples in Figure 2.6 can be expressed as $a_1 \times Alice \times b_3 \times (c_7 \cup ... \cup c_{555})$.

Chapter 3

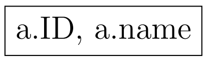
Intermediate Tuple Structure

In this chapter, we establish the set of f-trees that the tuple structure of `LBQP` adheres to. As we will show, `LBQP`'s tuple structure with no notion of reuse adheres to a limited set of f-trees. Hence, we introduce a trie-like tuple structure made of nodes we refer to as `FNode`(s). The tuple structure allows us to pass d-representations between operators.

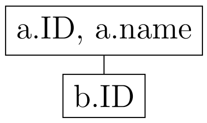
3.1 `LBQP` Tuple Structure

Recall from the previous chapter that `LBQP` is made of a list of groups and each group can be either *flat* or *unflat*. A flat group over a set of attributes $\mathcal{S} = \{A_1, A_2, ..., A_k\}$ represents a set with a single tuple $(\langle A_1 : a_1\rangle \times \cdots \times \langle A_k : a_k\rangle)$. An unflat group over $\mathcal{S}$ represents a set of tuples: $\cup_i\{(\langle A_1 : a_{1_i}\rangle \times \cdots \times \langle A_k : a_{k_i}\rangle)\}$. The final result set is obtained by applying a Cartesian product over the set of tuples obtained from each list group.

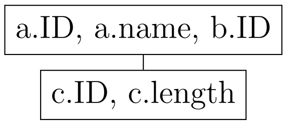
Next, we consider which f-trees the tuple structure of `LBQP` adheres to in the plan in Figure 2.4. The output of the `Scan` and `Filter` operators is a single unflat list group containing the `a.ID` and `a.name` attributes shown as **list group 1** in Figure 2.6. The equivalent f-tree and f-representations are shown in Figures 3.1a and 3.1d, respectively. The join operator jo$_1$ (a$\rightarrow$b) flattens the **list group 1** and for a fixed `a.ID` value extends to the set of `b.ID` values that are stored in a new **list group 2** which are passed to jo$_2$. The final output tuples of jo$_1$ follow the f-tree and f-representations as shown in Figures 3.1b and 3.1e, respectively. Similarly, jo$_2$ flattens the **list group 2** and for a fixed `b.ID` value extends to the set of `c.ID` and c.length values. The final output tuples of jo$_2$ follow the f-tree and f-representations as shown in Figures 3.1c and 3.1f, respectively.



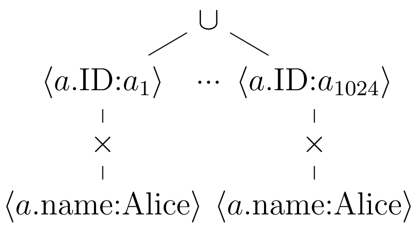
(a) F-tree for `Scan-Filter`'s output.



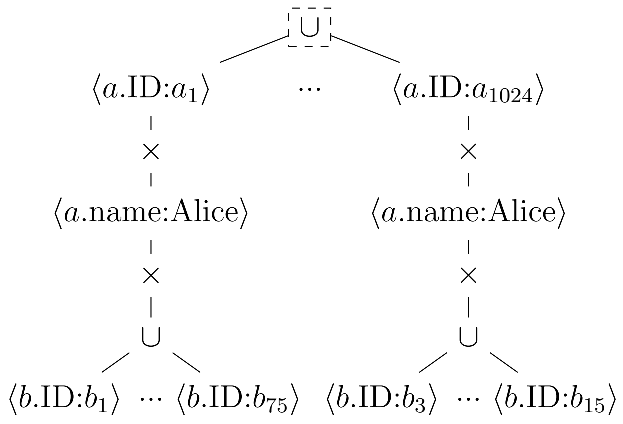
(b) F-tree for $\texttt{JOIN}_{jo_1}$'s output.



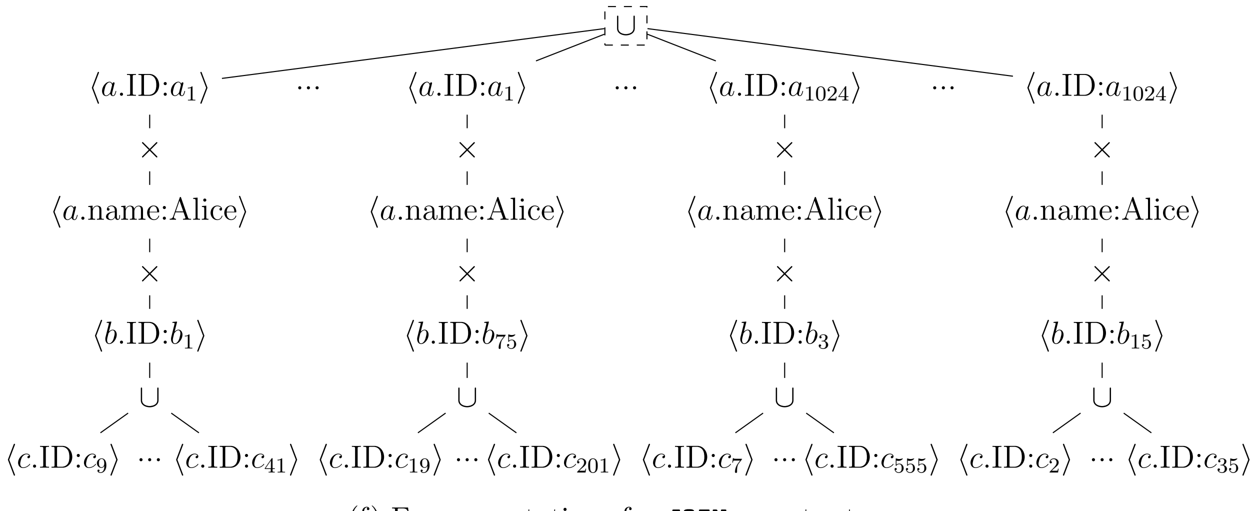
(c) F-tree for $\texttt{JOIN}_{jo_2}$'s output.



(d) F-tree for `Scan` output.


∪
⟨a.ID:a1⟩ ··· ⟨a.ID:a1024⟩
×
⟨a.name:Alice⟩ ⟨a.name:Alice⟩
∪
⟨b.ID:b1⟩ ··· ⟨b.ID:b75⟩ ⟨b.ID:b3⟩ ··· ⟨b.ID:b15⟩

(e) F-representations for $\texttt{JOIN}_{jo_1}$ output.


∪
⟨a.ID:a1⟩ ··· ⟨a.ID:a1⟩ ··· ⟨a.ID:a1024⟩ ··· ⟨a.ID:a1024⟩
×
⟨a.name:Alice⟩ ⟨a.name:Alice⟩ ⟨a.name:Alice⟩ ⟨a.name:Alice⟩
⟨b.ID:b1⟩ ⟨b.ID:b75⟩ ⟨b.ID:b3⟩ ⟨b.ID:b15⟩
∪
⟨c.ID:c9⟩ ··· ⟨c.ID:c41⟩ ⟨c.ID:c19⟩ ··· ⟨c.ID:c201⟩ ⟨c.ID:c7⟩ ··· ⟨c.ID:c555⟩ ⟨c.ID:c2⟩ ··· ⟨c.ID:c35⟩

(f) F-representations for $\texttt{JOIN}_{jo_2}$ output.

Figure 3.1: F-trees and f-representations for the plan in Figure 2.4 in `LBQP`.

Note that `LBQP` is a pipelined processor that avoids materialization and passes tuples between operators a block at a time. jo$_1$ and jo$_2$ pass each time the set of tuples found under a branch of the root union in the dashed box as shown in Figures 3.1e and 3.1f. For example, jo$_1$'s first produced block is $\langle a.\text{ID}{:}a_1\rangle\times\langle a.\text{name}{:}\text{Alice}\rangle \times \cup_{b.ID}\{\langle b_1, ..., b_{75}\rangle\}$ and its last produced block is $\langle a.\text{ID}{:}a_{1024}\rangle\times\langle a.\text{name}{:}\text{Alice}\rangle \times \cup_{b.ID}\{\langle b_3, ..., b_{15}\rangle\}$. Similarly, jo$_2$'s first produced block $\langle a.\text{ID}{:}a_1\rangle\times\langle a.\text{name}{:}\text{Alice}\rangle\times \langle b.\text{ID}{:}b_1\rangle \times \cup_{c.ID}\{\langle c_9, ..., c_{41}\rangle\}$ and its last produced block is $\langle a.\text{ID}{:}a_{1024}\rangle\times\langle a.\text{name}{:}\text{Alice}\rangle\times \langle b.\text{ID}{:}b_{15}\rangle \times \cup_{c.ID}\{\langle c_2, ..., c_{35}\rangle\}$.

`LBQP`'s tuples as f-representations adhere to f-trees with a height $h \leq 1$. The f-trees have a single root node and possibly a set of leaf nodes. This leads to two problems. First, `LBQP` cannot represent d-representations which require an f-tree with a height $h \geq 2$ in order to reuse results. Non-leaf and non-root nodes results can be potentially reused and if $h \leq 1$, there are no such nodes and hence no opportunities for reuse. Second, `LBQP` is sub-optimal in the size of intermediate results generated even when constrained to f-representations only. `LBQP` cannot generate the most compact representations for certain queries. Consider for instance the 4-hop query $a{\rightarrow}b{\rightarrow}c{\rightarrow}d{\rightarrow}e$ where the most compact f-tree in terms of asymptotic worst-case analysis is shown in Figure 3.2 which `LBQP` plans cannot generate. As such, we need a new tuple structure to adhere to all possible d-trees.

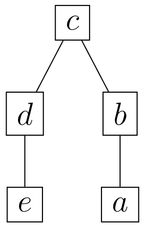


Figure 3.2: An f-tree for $a{\rightarrow}b{\rightarrow}c{\rightarrow}d{\rightarrow}e$

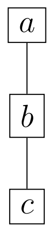
3.2 `FNode` Tries for Intermediate Results in `DPQP`

In order to be able to pass tuples with a d-representation adhering to any possible d-tree, we use a Trie structure to represent intermediate tuples that we call `FNode`. An `FNode` extends a list group with an added `FNode` children list denoted `FNodeChildren`. Let c be the number of children as indicated by the corresponding d-tree and T be the number of tuples in the list group. `FNodeChildren`'s size is $c \times T$. The tuple set stored in a trie rooted by a given `FNode` is denoted by `result(FNode)` and is defined as follows:

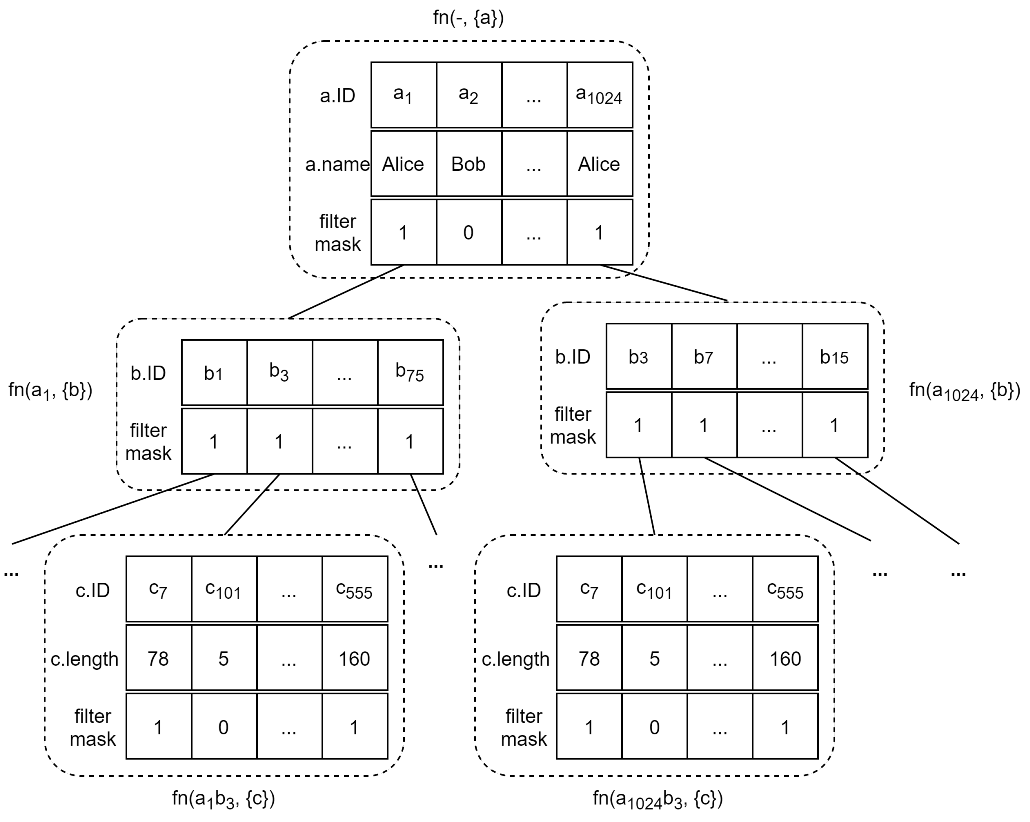
$$\texttt{result(FNode)} = \cup_{i=1}^{T} t_i \times (\Pi_{j=c.(i-1)}^{c.i-1} \texttt{result(FNodeChildren[}j\texttt{]))}$$

Next, we present examples of f- and d-representations using `FNode`(s).

3.2.1 F-representations using `FNode`(s)



(a) $\mathcal{T}_4$

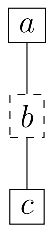


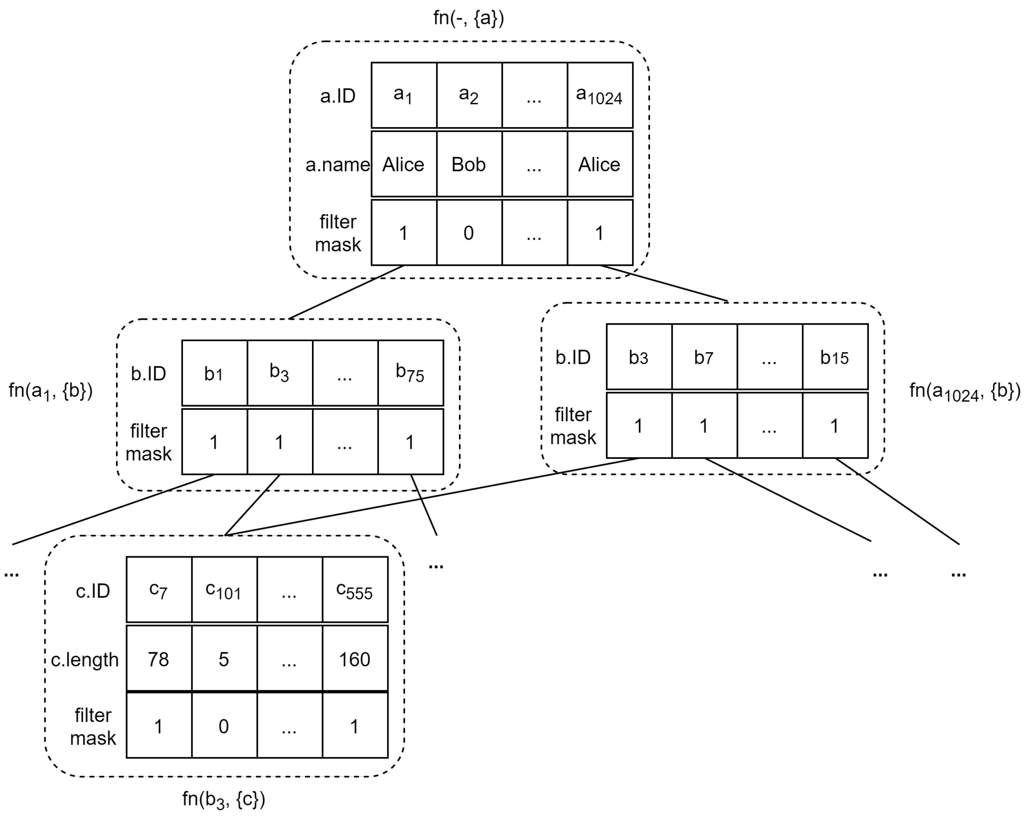
(b) F-representations using `FNode`(s) adhering to $\mathcal{T}_4$

Figure 3.3: An f-tree $\mathcal{T}_4$ and its f-representations using `FNode`(s).

Figure 3.3 shows the results of Query 2.3 as f-representations using `FNode`s that adhere to the f-tree $\mathcal{T}_4$. In the figure, each `FNode` is denoted by $fn(p, V)$ e.g., $fn(a_1b_3, \{c\})$. p, for prefix, represents the tuple that the f-representation rooted in the `FNode` depends on. V denotes the set of attributes stored in the `FNode`. For $fn(a_1b_3, \{c\})$, $p = \langle a : a_1\rangle \times \langle b : b_3\rangle$ shortened to a_1b_3 and $V = \{c\}$ and hence $fn(a_1b_3, \{c\})$ stores the values obtained given p for attributes `c.ID` and `c.length` (i.e., the attributes of c mentioned in Query 2.3). For the root `FNode`, $p = -$ and in this case the root is $fn(-, \{a\})$. Notice in this f-representations, the FNodes $fn(a_1, \{b\})$ and $fn(a_{1024}, \{b\})$ contain ⟨b.ID: a_3⟩ value which points to two different FNode instance children that are identical in the values they store.

3.2.2 D-representations using FNode(s)


(a) $\mathcal{T}_{\uparrow 4}$



(b) D-representations using FNode(s) adhering to $\mathcal{T}_{\uparrow 4}$.

Figure 3.4: An d-tree $\mathcal{T}_{\uparrow 4}$ and its d-representations using FNode(s)

Figure 3.3 shows the results of Query 2.3 as d-representations using FNodes that adheres to $\mathcal{T}_{\uparrow 4}$. In this representations, unlike the f-representations in Figure 3.3b, the FNodes $fn(a_1, \{b\})$ and $fn(a_{1024}, \{b\})$ contain ⟨b.ID: a_3⟩ value which point to two same FNode instance $fn(b_3, \{c\})$. For $fn(b_3, \{c\})$, p =⟨$b : b_3$⟩ shortened to b_3. As indicated by d-tree $\mathcal{T}_{\uparrow 4}$ where b is in a dashed box, all FNodes storing c attributes depend only on the b.ID attribute enabling reuse.

Chapter 4

Operators and Plans

This chapter presents `DPQP`'s major operators inherited from `LBQP` and introduces a new operator named `FGroup`. We also describe how to obtain a `DPQP` query plan and integrate caching into it.

4.1 Operators

`DPQP` is able to reuse existing operators in `LBQP` such that a list group is an `FNode` with its `FNodeChildren` set to `NULL`. We first describe the major operators inherited from `LBQP`. Note that join operators need to flatten input `FNode`s as necessary.

- **Scan**: scans a set of consecutive node IDs and any needed attributes. The output of the operator is a single unflat `FNode` which stores: 1) the `ID` attribute as a pair of start and end offsets in a value vector; and 2) all needed attributes in their corresponding value vectors.
- **Extend**: is an index nested loop join operator that takes a partial match t as input and extends t by an unmatched query edge using an adjacency list index. Consider an example of extending a partial match t with query edge $v_s \rightarrow v_d$ where v_s has been matched in t. `Extend` will first flatten the `FNode` containing $v_s.ID$ as necessary which leads to a flat tuple v_{si} and then read the forward adjacent list index of v_{si} and write neighbor node IDs to a value vector of an unflat `FNode` to produce a new partial match t' with query edge $v_s \rightarrow v_d$ matched. `Extend` also reads all necessary attributes of the matched query edge and neighbor query node.

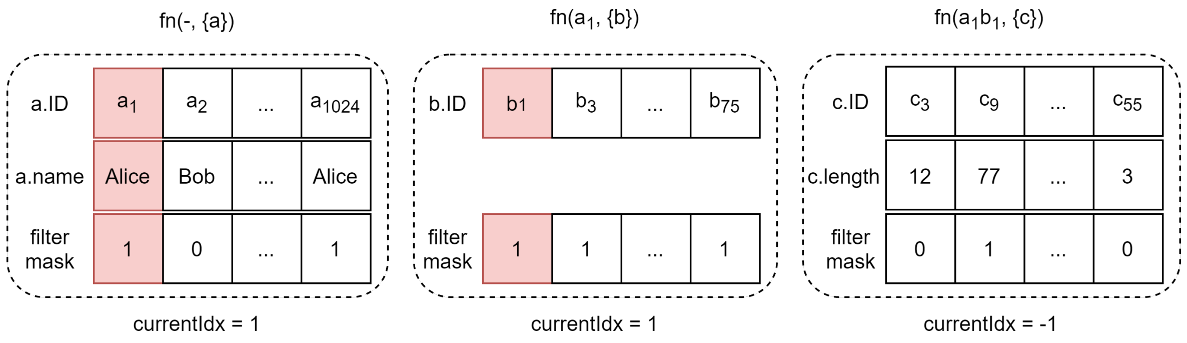


Figure 4.1: Input partial match t to FGroup(c by b).

- **Filter**: takes as input a predicate and a partial match t which has matched all variables required to evaluate the predicate. Regardless of the factorized structure of t, the filter predicate should be evaluated on each flat representation of t. Therefore, if the predicate includes variables from multiple FNodes, DPQP will ensure at most one FNode is unflat so that expression evaluators do not need to calculate a Cartesian product between different FNodes and thus simplify the implementation. Evaluation result is written to the filter mask of the unflat FNode.

- **Intersect**: is a multi-way extend operator that uses worst case optimal join algorithm described in reference [14]. Intersect takes as input a partial match t and extends t by two or more unmatched query edges using adjacency list index. Similar to Extend operator, Intersect flattens matched query vertices in t of those unmatched query edges as necessary. Intersect first sorts node IDs in each adjacency list index and then performs a merge sort to get a set of neighbor node IDs matching input query edges. Result node IDs are written to a value vector of an unflat FNode.

4.1.1 FGroup Operator

FGroup(u by v) is a grouping operator that constructs the nested trie structure between FNodes. FGroup(u by v) takes as input a partial match t with k FNodes that contains a flat FNode storing attributes of v denoted by fn_v and an unflat FNode storing attributes of u denoted by fn_u. It groups fn_u as the currentIdx-th child of fn_v. FGroup blocks computation until all children of fn_v have been computed, i.e. currentIdx of fn_v equals to vector size of fn_v. It then outputs a new partial match t' with $k-1$ FNodes but matching the same set of variables as t. We demonstrate the computation of FGroup with

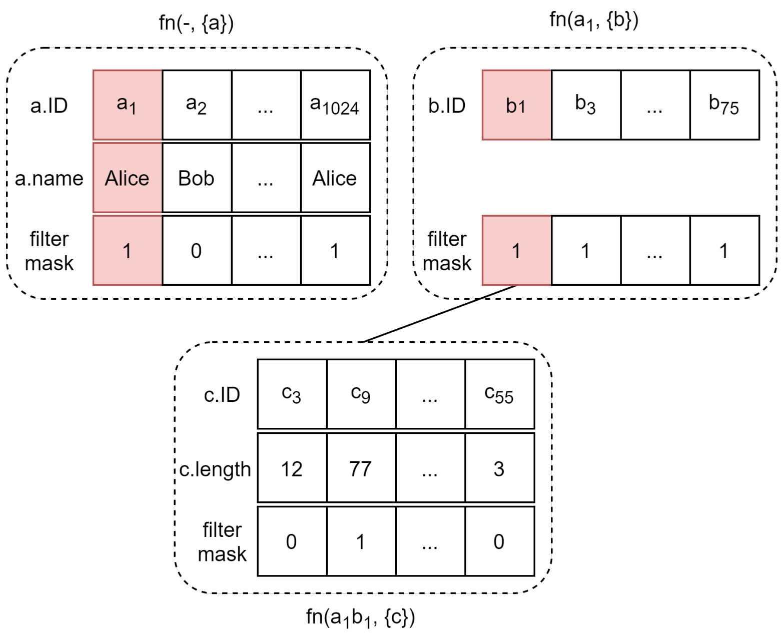


Figure 4.2: State of t after grouping $fn(a_1b_1, \{c\})$.

an example.

Example 2 Imagine we append an `FGroup`(c by b) operator at the end of the plan shown in Figure 2.4, a sample input partial match t to `FGroup`(c by b) is shown in Figure 4.1. `FGroup` performs a grouping operation which sets $fn(a_1b_1, \{c\})$ as a child of the first tuple in $fn(a_1, \{b\})$, i.e. b_1, as shown in Figure 4.2. Readers can verify this grouping does not change the set of flat tuples represented in Figure 4.1. `FGroup` blocks further computation until all children `FNodes` under $b_1, b_3, ..., b_{75}$ are computed as shown in Figure 4.3. In the output partial match t', $fn(a_1, \{b\})$ represents the sub-query result of the sub-tree rooted at b in $\mathcal{T}_4$ in Figure 3.3a as a factorized structure for its prefix a_1.

4.2 Query Plan

This section first studies `DPQP` query plans without caching which inserts `FGroup` operators into `LBQP` plans to form f-representations. We then describe how to integrate caching. We consider query plans for select-join queries that only contain `Extend` operators and other

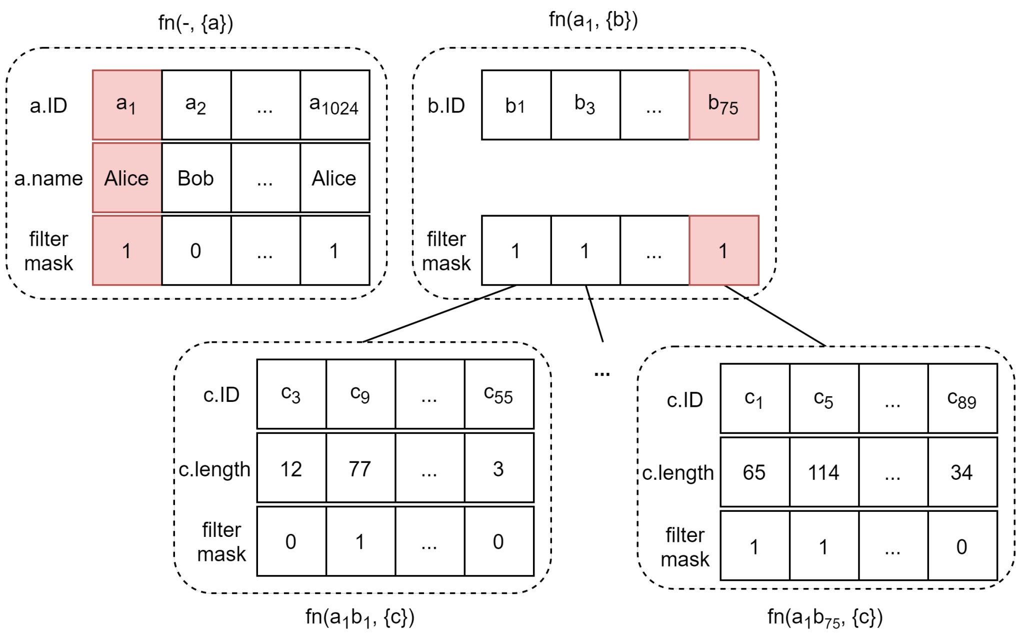


Figure 4.3: Output partial match t' of `FGroup`(c by b).

operators introduced in Section 4.1. Our plans are linear plans that scan some nodes and perform a sequence of extends. Bushy plans that join two intermediate non-base relations are not supported and left for future work.

4.2.1 Plan with Path F-tree

Consider a path f-tree $\mathcal{T}$ and an `LBQP` plan which is a chain of `Scan`, `Extend` and `Filter` operators whose join order follows a top-down traversal of $\mathcal{T}$. Every node in $\mathcal{T}$, except the root, leads to the insertion of an `FGroup` operator done as follows. We traverse $\mathcal{T}$ in bottom-up order and insert `FGroup` operators with the following rules. Consider a node v_c and its parent v_p,

- If v_c is the leaf node in $\mathcal{T}$, we insert `FGroup`(v_c by v_p) after `Extend`($v_p \rightarrow v_c$) (or after the last `Filter` follows `Extend`($v_p \rightarrow v_c$) if there is any).
- Otherwise, since we perform grouping with bottom-up traversal on $\mathcal{T}$, the child of v_c must have been grouped so there is one and only one `FGroup`(x by v_c) where v_c is the parent. We insert `FGroup`(v_c by v_p) after `FGroup`(x by v_c).

Example 3 Consider the `LBQP` plan shown in Figure 2.4 whose f-tree $\mathcal{T}_4$ is shown in Figure 3.3a. We traverse $\mathcal{T}_4$ in bottom-up order and first consider node c and its parent b. Since c is the leaf node, `FGroup`(c by b) is inserted after the `Filter` following `Extend`($b \rightarrow c$). For variable b and its parent a, we insert `FGroup`(b by a) after the `FGroup` operator where b is parent i.e. `FGroup`(c by b). Figure 4.7a shows the final plan. We further illustrate execution with the following example. Example 2 has demonstrated the execution of `FGroup`(c by b) for the batch of tuples under prefix a_1 whose output partial match t' is shown in Figure 4.3. `FGroup`(b by a) takes t' as input and groups $fn(a_1, \{b\})$ as the first child `FNode` of $fn(\emptyset, \{a\})$. It then blocks further computation and asks for the next batch of tuples under a different a prefix whose computation is similar to a_1. The output of `FGroup`(b by a), i.e. $fn(\emptyset, \{a\})$, is shown in Figure 3.3b which is also the final query result.

Note that the join order of a `DPQP` plan follows the top-down traversal of the f-tree while grouping order follows the bottom-up traversal of the f-tree. `FGroup` operators take the advantage of the fact that grouping is done in the reverse order of extension and avoid full materialization. Recall that in the description of `Extend` operator, every set of neighbor node IDs is obtained from a prefix t and thus can be grouped as the child of t. Consider the sample input shown in Figure 4.1, the set of $c.ID$ in $fn(a_1b_1, \{c\})$ was computed in Extend($b \rightarrow c$) for prefix (a_1, b_1). It is guaranteed that $c_3, c_9, ..., c_{55}$ is a complete set of $c.ID$ for (a_1, b_1) and there will be no other $c.ID$ under the same prefix. Therefore, we can safely group $c_3, c_9, ..., c_{55}$ as a child of (a_1, b_1) and continue to process the next batch of $c.ID$ which will be the child of a different prefix. Imagine we change the `FGroup` operator to `FGroup`(b by c) in the above example which groups in a different order from the reverse join order. In order to find the set of $b.ID$ for a particular $c.ID$ e.g. c_3, we have to first materialize all tuples since there might be another `FNode` containing c_3 but under a different prefix.

4.2.2 Plan with Bushy F-tree

We now describe the general case where f-trees can be bushy. Consider a bushy f-tree $\mathcal{T}$ and an `LBQP` plan whose join order follows the pre-order traversal of $\mathcal{T}$. we perform a post-order traversal on $\mathcal{T}$ and focus on nodes with multiple children. Given a node v_p with multiple children, for each root-to-leaf path in the sub-tree under v_p, we consider a sub-path p_{sub} on the root-to-leaf path where all nodes on p_{sub} have not yet been grouped. Since p_{sub} is a path f-tree, we could apply the rules described in Section 4.2.1 with a slight modification to insert `FGroup` operators for each node on p_{sub}. We continue to evaluate the next node with multiple children until all nodes, except for the root, are grouped. The modification is stated as follow

- Recall that v_c is a child of v_p. If v_c is a non-leaf node, since we perform grouping with post-order traversal on $\mathcal{T}$, each child of v_c must have been grouped. Therefore there is **at least one** `FGroup`(x by v_c) where v_c is the parent. We insert `FGroup`(v_c by v_p) after **the last** `FGroup`(x by v_c).

This modification captures the case that v_p might have multiple children in a bushy f-tree and v_p should be grouped only if all of its children have been grouped.

Query 4.1: Star query

```
MATCH (a:PERSON)—[:KNOWS]→(b:PERSON),
      (b:PERSON)—[:LIKES]→(c:POST)
      (b:PERSON)—[:LIKES]→(d:CMT)
WHERE a.name = Alice AND c.length > 50
```

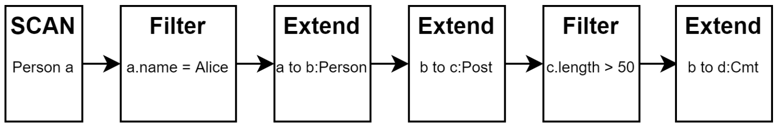


Figure 4.4: List-based plan for Query 4.1.

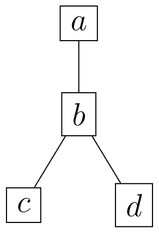


Figure 4.5: F-tree $\mathcal{T}_5$ for the plan in Figure 4.4.

Example 4 Consider Query 4.1 which extends Query 2.3 by adding an extra query edge $b \rightarrow d$ whose `LBQP` plan is shown in Figure 4.4 which follows the join order of the plan in Figure 2.4 and append an Extend($b \rightarrow d$) at the end. Figure 4.5 shows the f-tree $\mathcal{T}_5$ corresponding to the plan in Figure 4.4. Note that c and d are on different root-to-leaf

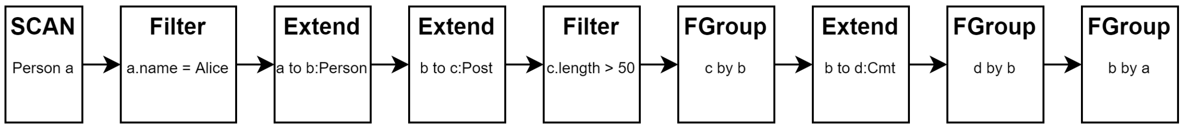
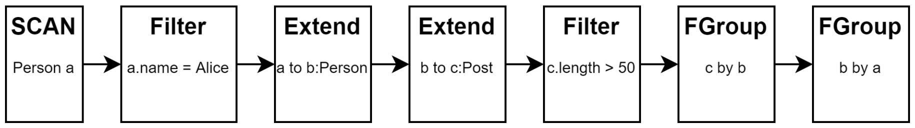


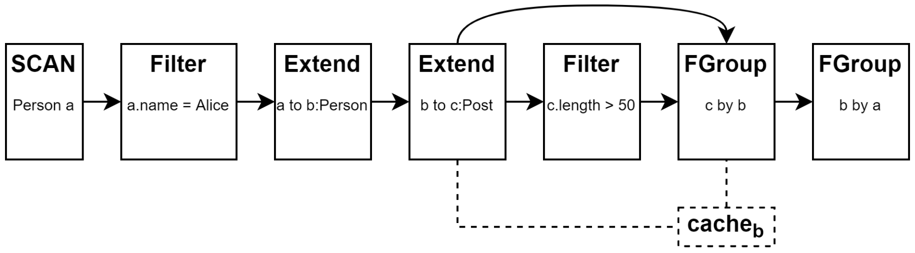
Figure 4.6: DPQP plan with f-representations adhering to f-tree $\mathcal{T}_5$.

paths because, for a fixed value of b, the set of c values and d values can be uniquely determined. In other words, c and d are conditionally independent for prefix b. Since $\mathcal{T}_5$ is a bushy f-tree, we first group nodes under the sub-tree rooted at b. For root-to-leaf path $b - c$ where c is a leaf node, we insert FGroup(c by b) after the Filter following Extend($b \rightarrow c$). Similarly, an FGroup(d by b) is inserted after Extend($b \rightarrow d$). We next consider the ungrouped sub-path $a - b$. Since the sub-tree underneath b has been grouped, we can identify two FGroup operators i.e. FGroup(c by b) and FGroup(d by b) where b is the parent FNode. FGroup(b by a) is then inserted after the last FGroup, i.e. FGroup(c by d) since b is grouped only if the sub-tree under b has been grouped. Figure 4.6 shows the final DPQP plan with f-representations whose execution is similar to the plan in Figure 4.7a.

4.3 D-representation-based Caching



(a) DPQP plan with f-representations adhering to f-tree $\mathcal{T}_4$ in Figure 3.3a.



(b) DPQP plan with d-representations adhering to d-tree $\mathcal{T}^{\dagger}{}_4$ in Figure 3.4a.

Figure 4.7: DPQP plans with f- and d-representations.

In the section above we have demonstrated how to modify an LBQP plan to obtain a DPQP plan with f-representations. We further show how to integrate caching to obtain a plan with d-representations and thus avoid repetition in both data representations and computation. We first describe on which variables caching can be applied. Given a plan using f-representations and its corresponding f-tree, we could infer a d-tree by computing dependency labels for each node in the f-tree using Definition 3

- If v is a leaf node, we don't apply cache since there is no sub-query result to cache for a leaf.
- If v is the root node, no cache is applied since root node IDs, are guaranteed to be unique and thus has no repetition.
- If all nodes in the sub f-tree rooted at v depend only on v, then caching can be applied and the same children `FNode`s for v can be reused.

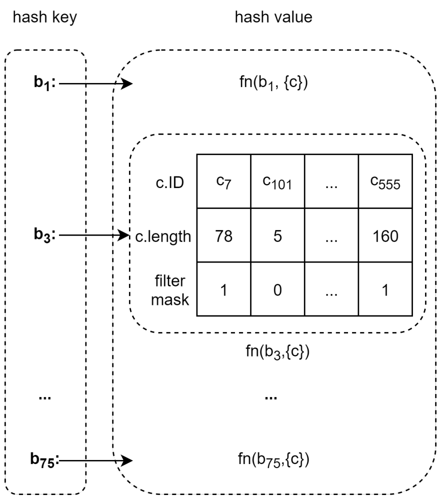


Figure 4.8: Cache of b.

If cache on v_p is enabled, for each of v_p's child v_c, there must be one and only one Extend($v_p \to v_c$) and `FGroup`(v_c by v_p). We integrate a shared cache between these `Extend` and `FGroup` operator pairs. Our cache is a hashmap from node IDs to `FNodes` where the hash key is also the prefix of the `FNode`. Consider an entry $(v_{p_i},\ fn(v_{p_i}, V_c))$, $fn(v_{p_i}, V_c)$ represents the sub-query result of sub-tree underneath v_c for prefix v_{p_i}. The cache is probed by `Extend` operator and updated by `FGroup` operator. Consider a value v_{p_i} which `Extend`($v_p \to v_c$) tries to extend from,

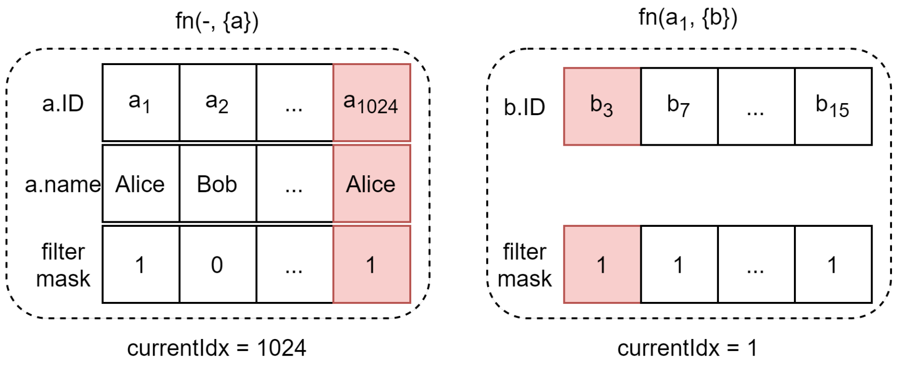


Figure 4.9: Input partial match to Extend($b \to c$).

- If v_{p_i} is not in the cache, we continue to process until `FGroup`(v_c by v_p). `FGroup` operator will group $fn(v_{p_i}, V_c)$ as the child of v_{p_i} and insert $(v_{p_i}, fn(v_{p_i}, V_c))$ into the cache.
- Otherwise, a cache hit indicates the sub-query result for prefix v_{p_i} has already been computed. We can safely skip the sequence of operators between `Extend`($v_p \to v_c$) and `FGroup`(v_c by v_p) and directly perform the grouping using the hash value under v_{p_i} from our cache.

We demonstrate with an example of how to integrate caching into the plan shown in Figure 4.7a which utilizes f-representations.

Example 5 Figure 3.4a shows the d-tree $\mathcal{T}^{\uparrow}{}_4$ obtained by analyzing dependence on Query 2.3 against f-tree $\mathcal{T}_4$. We focus on node b whose dependency labels are empty and thus caching can be enabled. For node b and its child c, we identify `Extend`($b \to c$) and `FGroup`(c by b) and configure a shared cache for the two operators. We further add a pointer from `Extend`($b \to c$) to `FGroup`(c by b) so that `Extend` can directly perform grouping on cache hit and skip $b \to c$ join as well as the filter $c.length > 50$. Figure 4.7b shows the final plan integrated with cache. We next demonstrate the execution when there is a cache hit. Similarly Example 2 as in this example, we have demonstrated the computation in `FGroup`(c by b) for the first batch of b values $(b_1, b_3, ..., b_{75})$ under prefix a_1. When caching is enabled for variable b, we insert into the cache for each b_i and its child `FNode` $fn(b_i, \{c\})$. The state of the cache of b after Example 2 is shown in Figure 4.8. Note that the `FNodes` in cache are identified with only b prefix since b has empty dependency labels. Now consider the next batch of b values for a different prefix a_{1024} as shown in Figure 4.9 which is the input of `Extend`($b \to c$). `Extend` first sets $currentIdx = 1$ and probes b_3 against the cache which results in a cache hit. It then skips subsequent join and filter operations and directly jumps to `FGroup`(c by b). `FGroup` puts the hash value of b_3 which is $fn(b_3, \{c\})$ as the first child of $fn(a_1, \{b\})$. Therefore, in the final d-representations in Figure 3.4b, (a_1, b_3) and and (a_{1024}, b_3) share the same $fn(b_3, \{c\})$ as child.

`DPQP` plans with caching not only enable d-representations which are more compact representation than f-representations but also avoid repeated computations of possible large sub-query results. If caching is enabled on a variable v, we guarantee to join only on unique $v.ID$ since any repeated $v.ID$ will be a cache hit whose result `FNode` is stored in cache hash table.

Chapter 5

Evaluations

We integrated DPQP into GraphflowDB which is an in-memory GDBMS. The version we used adopts LBQP as we reviewed in Section 2. If not specified, DPQP turns on caching whenever possible. The goal of our experiments is two-fold. First, we demonstrate DPQP's f-representations are more compact than LBQP's representations and DPQP's d-representations can be further compact than DPQP's f-representations. Second, we report the overheads of constructing factorized representations and analyze the performance gain of caching.

5.1 Experimental Setup

Hardware

We use a single machine with two Intel E5-2670 @2.6GHz CPUs and 512 GB of RAM. The machine has 16 physical cores and 32 logical ones. All experiments use a single thread. The maximum heap size of JVM is set to 200GB.

Datasets

Table 5.1 shows the datasets used in our benchmark. Our datasets include social networks, web graph and product co-purchasing network, which have a variety of graph topologies and sizes ranging from several million edges to over a hundred-million edges. We use 3 datasets from SNAP [12] and the LDBC social network dataset [2] generated with scale factors 10 which we denote by LDBC10.

Name	Num of vertices	Num of edges
Epinions	75.9K	508.8K
Amazon	403.4K	3.4M
Google	875.7K	5.1M
LDBC10	8.1M	176.6M

Table 5.1: Datasets used.

5.1.1 Query workload

For micro-benchmarks we generate 1-, 2-, 3-, and 4-hop queries without predicates. These queries serve as stress tests in order to evaluate how much space and computation can be saved on `DPQP` with d-representations. For end-to-end benchmarks, we use LDBC interactive complex (IC) queries which capture the workload characteristic in real-world scenarios.

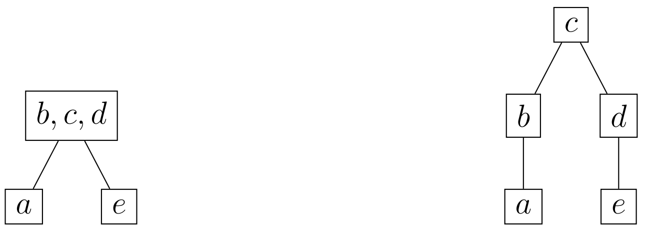
5.2 Size of Query Results

Datasets	Flat	LBQP	F-representations	D-representations
Epinions	119.8M	40.9M	40.5M	1M
Amazon	97.1M	39.1M	36.2M	7.2M
Google	182.1M	70.2M	66.2M	9.8M

Table 5.2: Number of singletons in 2-hop path query results with different representations.

We first study how much space can be saved when utilizing different factorized representations. Instead of measuring memory usage which might be inaccurate for different languages' implementations, we directly count the number of singletons (i.e. primitive data type instances) in the final results. For example, Figure 2.5a has 5 singletons and, Figure 2.5b contains $5 * 1024$ singletons. Table 5.2 shows the number of singletons of 2-hop path query results under different representations for 3 different datasets. As expected, flat representations have the most number of singletons while d-representations have the least and can be orders of magnitude smaller than the corresponding flat representations, e.g. on the Epinions dataset. Compared to flat representations, `LBQP`'s representations contain around 1/3 of singletons because it avoids repeating (a, b) prefix for each c value. F-representations are slightly more compact than `LBQP`'s representations while the difference is small. The intuition behind this is that the last `Extend` operator usually does the most amount of work. This can be understood as follows. Let d denote the average degree of nodes. For each prefix match to a leaf node, the leaf Extend operator creates on average d outputs. Therefore, the work done by leaf `Extend` operators dominate the total work. This behavior is a result of the fact that the extensions in 1-, 2-, 3-hop queries are over many-to-many relationships. However, in absence of caching, we cannot reduce the work on leaf `Extends` in `DPQP` with f-representations. We can only factor out non-leaf nodes, which leads to small gains.

5.3 Runtime Analysis of Plans with Full F-representations and D-representations



(a) `LBQP`'s f-tree. (b) `DPQP`'s f-tree.

Figure 5.1: Bushy f-trees for the 4-hop query in `LBQP` and `DPQP`.

We further study the runtime effects of utilizing full f-representations and d-representations. Specifically, we report the overheads of `FGroup` operators without caching and the benefit of f-representations for bushy f-trees. We also analyze how much benefit we can get from caching for different queries and database instances.

5.3.1 Runtime Overheads and Benefits of Full F-representations

Although `DPQP` with full f-representations generates more compact representations compared to `LBQP`, it also introduces computation overheads which come from `FGroup` operators. Recall that a `DPQP` plan with f-representations is obtained by inserting `FGroup`

Dataset	Query	LBQP	Full F-representations
Epinoins	**1-hop**	18.31	24.7 **(0.74x)**
	2-hop	309.3	394.4 **(0.78x)**
	4-hop	31836.8	900.8 **(35.35x)**
Amazon	**1-hop**	110.0	136.8 **(0.80x)**
	2-hop	898.0	1684.9 **(0.53x)**
	4-hop	8514.9	3427.9 **(2.48x)**
Google	**1-hop**	191.3	253.7 **(0.75x)**
	2-hop	2140.7	3473.9 **(0.62x)**
	4-hop	24947.4	7614.9 **(3.28x)**

Table 5.3: Runtime (in ms) of `DPQP` with full f-representations v.s. `LBQP` on 1-, 2-, and 4-hop queries

operators to a `LBQP` plan. In other words, a `DPQP` plan will do all the work in its corresponding `LBQP` plan plus the work in `FGroup` operators. Table 5.3 shows the runtime of `DPQP` plans with full f-representations v.s. `LBQP` on 1-, 2- and 4-hop queries. We pick left-deep plans for 1-, and 2-hop path queries and for the 4-hop query, we use the plan with join order c, b, a, d, e. Figure 5.1 shows f-trees adhering to the join order above for the 4-hop query in `LBQP` and `DPQP` respectively. Note that for `DPQP` we use full f-representations and thus no cache can be enabled. We can observe that for 1-, and 2-hop path queries, plans in `DPQP` with full f-representations are constantly slower compared to plans in `LBQP`. These overheads, as we explained, come from grouping operations whose number equals to the total number of extensions performed. On the other hand, `DPQP` with full f-representations can be up to 35.35 times faster on the 4-hop query. This is because LBQP can only factor out leaf nodes and Extend($d \rightarrow e$) has to be executed for each different (b, c, d) prefix. While in DPQP, we are able to explore the conditional independence between a, b and d, e for a fixed c value. As a result, Extend($d \rightarrow e$) is only executed for each different (c, d) which could be a much smaller number compared to the total number of different (b, c, d).

5.3.2 Runtime Benefits of D-representations

Datasets	Query	LBQP	DPQP
Epinoins	**2-hop**	309.3	68.8 **(4.5x)**
	3-hop	23766.7	105.3 **(255.6x)**
Amazon	**2-hop**	898.0	757.6 **(1.2x)**
	3-hop	9121.8	1587.8 **(5.8x)**
Google	**2-hop**	2140.7	191.3 **(1.3x)**
	3-hop	25139.6	2140.7 **(9.6x)**

Table 5.4: Runtime (in ms) of LBQP v.s. DPQP on 2- and 3-hop path queries.

Join Path Length

Table 5.4 shows the runtime of plans in LBQP v.s. DPQP on 2- and 3-hop path queries with left-deep plans. We omit the 1-hop path query since d-representations based processing cannot cache and reuse on the 1-hop query. For the 2-hop path query ($a \rightarrow b \rightarrow c$), a cache is enabled for node b. Similarly, for the 3-hop path query ($a \rightarrow b \rightarrow c \rightarrow d$), two caches are enabled on nodes b and c respectively. We first observe plans with d-representations are constantly faster and the longer the join path is, the more benefit we have. For example, the DPQP plan with d-representations is 4.6 times faster than LBQP plan on the 2-hop query

Dataset	**Query**	**LBQP**	**DPQP**
Epinions	**1-hop**	75.9K	75.9K **(1.0x)**
	2-hop	584.7K	127.9K **(4.6x)**
	3-hop	40.5M	177.7K **(227.9x)**
Amazon	**1-hop**	403.4K	403.4K **(1.0x)**
	2-hop	3.8M	806.7K **(4.7x)**
	3-hop	36.2M	1.2M **(29.9x)**
Google	**1-hop**	916.4K	916.4K **(1.0x)**
	2-hop	6.0M	1.6M **(3.7x)**
	3-hop	66.7M	2.3M **(29.0x)**

Table 5.5: Total number of extensions performed in `LBQP` v.s. `DPQP` on 1-, 2- and 3-hop path queries.

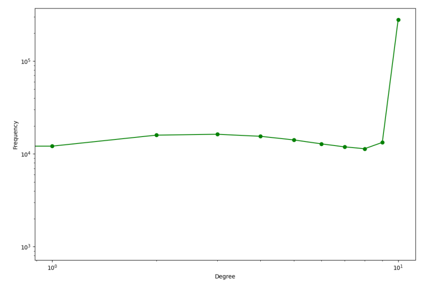
on Epinions dataset. While on 3-hop query, the plan with d-representations becomes 227.9 times faster. Consider the cache on node b for 2- and 3-hop queries. Intuitively, a cache hit of b on the 3-hop query is more beneficial since it skips subsequent extensions on both c and d while a cache hit on the 2-hop query only saves one following join.

In order to have a more accurate analysis, we count the total number of extensions performed, i.e., Extend operator calls on some prefix tuple, which is a proxy metric to measure the cost of a plan since our plans only involve node ID based extensions. Table 5.5 shows the total number of extensions performed in plans in `LBQP` v.s. `DPQP` on 1-, 2- and 3-hop path queries. We include 1-hop queries in the table so that the trend can be more clear. For `LBQP`, the total number of extensions increases quadratically as join path gets longer.

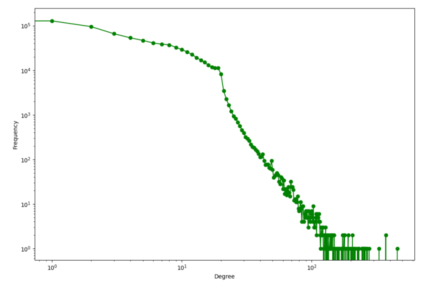
For example on Amazon dataset, 0.4M, 3.8M and 36.2M extensions are performed for 1-, 2- and 3-hop queries respectively. This is because the leaf Extend operator is executed repeatedly for different prefixes. On the other hand, the total number of extensions increase linearly for plans with d-representations. Consider the same Amazon dataset, plan with d-representations perform 0.4M more extensions for each hop increased in the join path. Recall that if caching is enabled on node v, then we guarantee to compute extensions only for unique $v.ID$.

In a worst-case analysis i.e., given a clique input graph with n vertices and m edges, a k-hop path query without predicates evaluated with an f-tree as a chain performs $n+m^{k-1}$ extensions while a d-representation as a chain with caching in every node except the root and leaf nodes leads to $k \times n$.

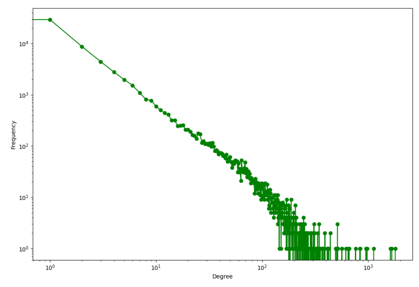
Graph Skewness



(a) Amazon out degree distribution



(b) Google out degree distribution



(c) Epinions out degree distribution

Figure 5.2: Out degree distributions for Amazon, Google and Epinions datasets.

Another observation from Table 5.4 is that the gain of utilizing cache varies drastically for different database instances. For example, for the same 3-hop path query, `DPQP` plan with d-representations is only 5.8 times faster compared to `LBQP` plan on Amazon dataset while being 255.6 times faster on Epinions dataset. Since our queries do not contain any predicates and match the full graph, an intuitive answer is to relates the gain difference to graph degree distributions. Figure 5.2 shows the distribution of out-going degrees for three different datasets. The degree distribution is uniform on Amazon dataset where almost all vertices have outgoing degrees less than 10. On the other hand, Google and Epinions datasets are more skewed where a few vertices have much higher outgoing degrees. In particular, we can see there are some vertices with more than 1k out-going edges on Epinions dataset from Figure 5.2c. A cache hit on such highly connected vertices yields more gains since more subsequent joins can be saved potentially. As a result, when doing subgraph matching without predicates on highly skewed graphs such as Epinions, we expect to have more performance gain from caching.

5.4 End-to-End Benchmarks

System	Range	IC01	IC02	IC03	IC04	IC05	IC06
LBQP	0.1%	1929.3	1552.0	23979.0	81.4	43548.5	7537.1
	1%	7111.5	6497.0	80443.9	365.7	188067.8	21074.8
DPQP	0.1%	234.0 **(8.6x)**	1206.0 **(1.3x)**	4494.3 **(5.3x)**	135.6 **(0.6x)**	5676.3 **(7.7x)**	815.7 **(9.2x)**
	1%	237.4 **(30.0x)**	2963.4 **(2.2x)**	5270.6 **(15.3x)**	404.2 **(0.9x)**	5855.0 **(32.1x)**	920.2 **(22.9x)**
	Range	**IC07**	**IC08**	**IC09**	**IC11**	**IC12**	
LBQP	0.1%	49.6	7.3	56284.2	216.3	1022.3	
	1%	174.5	49.0	181594.8	854.1	5188.0	
DPQP	0.1%	34.0 **(1.5x)**	11.2 **(0.7x)**	9322.5 **(6.0x)**	96.4 **(2.2x)**	902.7 **(1.1x)**	
	1%	169.7 **(1.0x)**	68.0 **(0.7x)**	9737.5 **(18.7x)**	208.9 **(4.1x)**	2098.3 **(2.5x)**	

Table 5.6: Runtime (in ms) of `LBQP` v.s. `DPQP` on LDBC IC queries

Table 5.6 shows our end-to-end runtime benchmark of `LBQP` v.s. `DPQP` on LDBC IC queries under different selectivity. Original IC queries start from a single point which leads to small results size and few or even none repeated node ID joins. We modify these queries to initially scan a range, specifically 0.1% and 1%, of vertices so that queries become big enough and contain repeated node IDs. We omit IC10 because our system does not support filtering on DATE type. Every `DPQP` plan has the same join order as its corresponding `LBQP` plan.

We first observe the gains of caching are between 1.1x and 9.2x on 0.1% selectivity and between 2.2x and 32.1x on 1% selectivity except for IC04, IC07, and IC08. Queries like IC01, IC03, IC05, IC06, and IC09 benefit significantly from caching since they contain long join paths with at least 3 hops and cache can be enabled on multiple nodes. For example IC05, whose improvement is most significant, is a 4-hop path query ($a \rightarrow b \rightarrow c \rightarrow d \rightarrow e$) with a predicate on a and e each. Three caches are enabled on IC05 for nodes b, c, and d and any cache hit not only saves subsequent joins but also the filter evaluation on e which is the dominant work. On the other hand, there is no gain or even overheads on IC04, IC07 and IC08. These queries do not benefit from caching since they only involve joins with many-to-one relationships and we can not cache any sub-query results while we still pay the cost of constructing `FNodes`. Consider query IC07 with is a 2-hop path query ($a \rightarrow b \rightarrow c$) where $a \rightarrow b$ is a join on a one-to-many relationship. Every b under different prefix a is guaranteed to be unique which makes maintaining cache a pure overhead. We also observe that for queries that benefit from d-representations, larger range scans lead to more performance gains. The underlying reason is that there is a higher number of repeated node IDs in subsequent joins for larger ranges.

Chapter 6

Related Work

We covered the FDB system and literature on the foundations of factorized databases in prior sections. We first broadly review work on compressed representation systems that are related to factorization. Then, we review related work on query processing using factorized representations. There is recent work that uses factorization to perform machine learning [23, 7] computations using factorized representations of relations and their incremental maintenance [16], which are less related to our work and not covered in detail.

D-representations, in addition to generalizing f-representations also capture several other representation systems that have been proposed in prior literature for different applications, such as world-set decompositions to represent a set of possible worlds in probabilistic databases [17], lossless fifth normal form decompositions [20], or factorized provenance polynomials [18]. Importantly, in addition to capturing these representations, d-representations are designed to be suitable to make query processing efficient by keeping intermediate results in compressed format.

Further work on the FDB system described query processing techniques for aggregations and ordering [4]. Similar to the original work on FDB, this work extends FDB with operators that take as input f-representations and output f-representations. As we discussed earlier, in addition to using d-representations, our approach instead keeps the traditional operators as is and uses pipelined operators that take in and output regular tuples and constructs d-representations at the ends of linear sub-plans. It is an interesting future work direction to study if our approach can be extended to evaluate all or classes of queries with aggregations and ordering.

Answer Graph [1] is a recent system 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 which is followed by cascading deletes in case a particular data node a_i, say matching variable (A_i) does not extend to (A_{i+1}). After this step, a second stage called the embedding generation stage generates a set of flat tuples by performing a left-deep join plan. The result of the first phase of Answer Graph is similar to our d-representations and keeps the same number of edges as intermediate data. However, the following enumeration phase does simple flattening. 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. In contrast to this approach, we do not need to handle cascading deletes as tuples that are not in the output never arrive at our `FGroup` operator. We also evaluate a larger class of queries and produce d-representations as outputs, so do not incur any size explosion.

Query processing on factorized representations of relations is ultimately a technique for identifying computations that will be repeated when evaluating queries and performing such computations once and reusing them. Several prior techniques from the literature on subgraph query evaluation have been designed with the same high-level goal. The postponing of Cartesian products optimization in the CFL subgraph matching algorithm [6] is a form of factorized query processing. Although the final results in CFL are flat tuples, the algorithm detects independent parts of a query and does perform the Cartesian product of these parts only when a set of partial matches are guaranteed to be in the output. The Cache Trie Join [10] (CTJ) algorithm extends the worst-case optimal Leapfrog Trie Join (LTJ) algorithm with caching. CTJ processes join queries one attribute (or query vertex) at a time using flat tuples. However, similar to d-representation-based processing, CTJ caches partial results for some prefixes and reuses them. The focus of this work is on keeping the memory footprint on LTJ low while benefiting from caching. CTJ is limited to only worst-case optimal join-style processing so does not decompose queries into bushy plans as our plans do. Reference [14] has introduced a simple caching optimization to the worst-case optimal Generic Join [15] algorithm, in the context of evaluating subgraph queries. The amount of caching is limited to only the matches of a single attribute, unlike CTJ, which caches entire sub-query results.

Another set of techniques are based on symmetry detection in the query. Examples include TurboISO and reference [21]. For example, given a in a subgraph query, TurboISO puts the query vertices that are guaranteed to match exactly the same data vertices into equivalence classes and only one of them is matched to data vertices. These matches are cached and re-used for other query vertices. It is possible to compute symmetric components once and name them in d-representations, the d-representations based query processing technique we described here or described in the original publication on d-representations and d-trees [19] do not exploit symmetry at the query level. In contrast, they find repeated computations for a prefix of partial matches of variables on parts of the query that are not necessarily symmetric. However, symmetry detection techniques are complementary to factorized query processing and can further result in avoiding further repeated work.

Chapter 7

Conclusions

Factorized databases utilize factorized data representations during query processing to obtain more compact final query results and faster runtimes. However, traditional factorized query processing requires full materialization at each operator and thus cannot be integrated easily into a pipelined query processor. In this thesis, we studied the integration of factorized query processing into GDBMSs in a pipelined fashion. We first review the tuple structure and architecture of `LBQP` on top of which we built our `DPQP`. The core idea of `DPQP` is to define a nested tuple structure, named `FNode` that mimics factorized representations and introduces `FGroup` operator which is responsible to construct nested `FNodes`. Each `FNode` represents a complete query result in factorized structure for a particular prefix. Once we obtain factorized intermediate tuple, we could further analyze conditional dependence information on the f-tree integrate caching accordingly. We demonstrated that `DPQP` plans, if caching can be enabled on some nodes, can significantly improve query runtime over `LBQP` plans on various datasets. The final query result is also orders of magnitude smaller.

We outline two possible directions of future work

- **Caching when conditional dependencies are violated by predicates:** Two variables are dependent if they appear in the same query edge or the same predicate. In many cases, complex predicates might prevent us from caching on very simple path queries. Consider a simple 2-hop path query ($a \rightarrow b \rightarrow c$) with a predicate $a.ID > c.ID$ and its left deep plan, no cache can be applied on b since a and c are dependent due to the given predicate. However, it's possible to first compute a sub-query result without predicates using cache and then apply the given predicate on the factorized results. Thus, an important line of future work is to study if it worth to pull up the filter of complex predicates and retain caching on certain variables.

- **Grouping in an order different from join order:** This thesis describes how to construct nested `FNode` with grouping in the reverse order of join order. Such constraint limits the factorization structure of final query results and could be problematic when a query contains GROUP BY or ORDER BY clauses whose desired factorization structure is different. It would be interesting to investigate how to perform grouping independent from join order and generalize for queries containing aggregations and ORDER BY clauses.

References

[1] Zahid Abul-Basher, Nikolay Yakovets, Parke Godfrey, Stanley Clark, and Mark Chignell. Answer graph: Factorization matters in large graphs. *EDBT*, 2020.

[2] Renzo Angles, János Benjamin Antal, Alex Averbuch, Peter A. Boncz, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba-Pey, Norbert Martínez-Bazan, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, Mirko Spasic, Benjamin A. Steer, Gábor Szárnyas, and Jack Waudby. The LDBC social network benchmark. *CoRR*, 2020.

[3] A. Atserias, M. Grohe, and D. Marx. Size Bounds and Query Plans for Relational Joins. *SICOMP*, 2013.

[4] Nurzhan Bakibayev, Tomáš Kočiský, Dan Olteanu, and Jakub Závodný. Aggregation and Ordering in Factorised Databases. In *VLDB*, 2013.

[5] Nurzhan Bakibayev, Dan Olteanu, and Jakub Závodný. Fdb: A query engine for factorised relational databases. *PVLDB*, 2012.

[6] Fei Bi, Lijun Chang, Xuemin Lin, lu Qin, and Wenjie Zhang. Efficient subgraph matching by postponing cartesian products. *SIGMOD*, 2016.

[7] Ryan Curtin, Ben Moseley, Hung Q. Ngo, XuanLong Nguyen, Dan Olteanu, and Maximilian Schleich. Rk-means: Fast clustering for relational data. *AISTATS*, 2020.

[8] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. *SIGMOD*, 2018.

[9] Pranjal Gupta, Amine Mhedhbi, and Semih Salihoglu. Integrating column-oriented storage and query processing techniques into graph database management systems. *VLDB*, 2021.

[10] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. Flexible caching in trie joins. *EDBT*, 2017.

[11] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedhbi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. *SIGMOD*, 2017.

[12] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford Large Network Dataset Collection, 2014.

[13] Amine Mhedhbi, Chathura Kankanamge, and Semih Salihoglu. Optimizing one-time and continuous subgraph queries using worst-case optimal joins. *TODS*, 2021.

[14] Amine Mhedhbi and Semih Salihoglu. Optimizing subgraph queries by combining binary and worst-case optimal joins. *PVLDB*, 2019.

[15] Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew strikes back: New developments in the theory of join algorithms. *SIGMOD*, 2013.

[16] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. F-ivm: Learning over fast evolving relational data. *SIGMOD*, 2020.

[17] Dan Olteanu, Christoph Koch, and Lyublena Antova. World-set decompositions: Expressiveness and efficient algorithms. *Theoretical Computer Science*, 403(2), 2008.

[18] Dan Olteanu and Jakub Závodný. On factorisation of provenance polynomials. In *TaPP*, 2011.

[19] Dan Olteanu and Jakub Závodný. Size bounds for factorised representations of query results. *TODS*, 2015.

[20] Raghu Ramakrishnan and Johannes Gehrke, editors. *Database Management Systems*. McGraw-Hill,, 2003.

[21] Xuguang Ren and Junhu Wang. Exploiting vertex relationships in speeding up subgraph isomorphism over large graphs. In *VLDB*, 2015.

[22] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M.Tamer Ozsu. The ubiquity of large graphs and surprising challenges of graph processing: Extended survey. *VLDB*, 2019.

[23] Maximilian Schleich and Dan Olteanu. Lmfao: An engine for batches of group-by aggregates. *PVLDB*, 2020.