Hybrid worst-case optimal query planning

de Brouwer, M.A.J.

Award date:
2020

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Databases research group

Hybrid Worst-Case Optimal Query Planning

Master Thesis

Mike de Brouwer

Supervisors:
dr. Nikolay Yakovets

Assessment Committee:
dr. Nikolay Yakovets
dr. George Fletcher
dr. Vlado Menkovski

Eindhoven, March 2020

Abstract

In this thesis, we study the problem of optimizing subgraph queries using a new class of join algorithms that are worst-case optimal. Worst-case optimal join algorithms solve a subgraph query one vertex at a time using multiway intersections. Traditional binary join algorithms are provable sub-optimal on certain classes of queries. We present the current research into worst-case optimal algorithms and so-called hybrid plans. Hybrid plans combine binary and worst-case optimal algorithms. We study how these hybrid plans perform. For this purpose, we designed a query planner that generates a large set of binary, worst-case optimal, and hybrid plans. Intending to experimentally establish if including hybrid plans in our plan space is worthwhile. We compare our plan space to other prior works. To create our hybrid plans, we adapted the leapfrog triejoin algorithm to work with binary algorithms in hybrid plans. We implemented our approach into Avantgraph, a main memory graph database engine. For our experimental evaluation, we created a query miner. Using these mined queries, we run extensive experiments on our entire plan space to demonstrate the effectiveness of our approach. We run our experiments on our full plan space to allow us to discuss a wide variety of plans. Our experimental evaluation shows promising results in favor of hybrid plans. We compare hybrid plans to both worst-case optimal and binary plans. Our data shows hybrid plans outperforming both binary and worst-case optimal plans across a wide variety of query types.

Contents

Chapter 1

Introduction

Graph databases have gained popularity in recent years due to their usefulness when dealing with certain types of data. Graph databases are inherently focused on dealing with relationships between entities. Therefore if your workload relies heavily on relationships, such as social networks, using graph databases could be more optimal than using other types of databases such as relational databases.

Subgraph matching (SGM) queries are an essential class of queries supported by graph databases. These queries find instances of the subgraph denoted by the query $Q(V_Q, E_Q)$ in the directed labeled graph $G(V, E)$. Examples of these subgraph patterns could be finding cycles or cliques of people in a social network. We study a general class of SGM queries where V_Q and E_Q have labels. SGM queries find mappings of the subgraph pattern onto the given graph by binding each vertex in the query to a vertex in the graph. These bindings are found by joining edge tables together in various ways. When evaluating a query edge with a certain label, the edge table we query contains only the edges in G that contain that label. SGM queries can be evaluated using two main approaches:

- **Edge by edge:** this approach executes a series of binary joins to evaluate Q. Each binary join expands the query pattern until Q is matched. This is the traditional approach to computing subgraph queries.
- **Vertex by vertex:** this approach picks a *query vertex ordering* (QVO) of V_Q. According to this QVO, it matches one vertex at a time, using multi-way intersections, until Q is matched. In essence, it binds each vertex in V_Q one-by-one. We refer to this as binding propagation. These joins are the core computation performed by the recent *worst-case optimal* algorithms (WCO) [10, 13, 12, 9, 11].

Among all the operations executed in databases, joins are the most costly. Therefore efficient join algorithms are required. We refer to plans that only use binary joins as *BJ* plans, with only intersections as *WCO* plans, and plans with both operations as *hybrid* plans.

1.1 Problem Statement

Recent theoretical results [4] have shown that BJ plans can be suboptimal on cyclic subgraph queries. An example of a cyclic subgraph query is finding all triangles of friends. BJ plans have run times that are asymptotically worse than the worst-case output size of these cyclic queries. The worst-case output size of a query is known as the *AGM* (Atserias, Grohe, Marx) bound [4]. WCO join plans can execute these queries within the AGM bound. The AGM bound will be elaborated on later. WCO join algorithms correct for this suboptimality. Since binary joins are provable suboptimal on certain query classes, are they still relevant? Completely ignoring binary joins would mean to ignore decades of research into their optimization. Therefore our research

focuses on so called hybrid plans. These plans combine both WCO and binary join algorithms to form plans that can combine the strengths of both. This brings us to the first of our research questions.

Problem 1: When, if ever, do binary joins have a better performance relative to WCO joins? Both in BJ plans and hybrid plans.

Here, we are not only interested in which algorithm runs asymptotically faster. We are also interested in real-time performance, which involves more than theoretical run time guarantees. Notions such as cache efficiency and how well the algorithms can be made to execute over multiple cores and or made to work with *single instruction, multiple data* SIMD processing also play a role here. SIMD allows a processor to perform the same operation on multiple data simultaneously.

Problem 2: Does combining WCO and binary joins into hybrid plans produce plans that are better performing than their pure counterparts? Specifically, under what circumstances would hybrid plans be better?

Pure plans are plans where all algorithms are either binary or WCO. In the above problems, performance relates to the cost model that the plans are measured by.

Our research will focus on exploring different classes of queries. For each query, an extensive plan space will be computed and experimented with to answer the above questions.

1.2 Motivation

We are interested in finding out in what situation exactly WCO and binary joins perform best. If we can combine their strengths into hybrid plans, the resulting plans should be an improvement over traditional plan spaces. Therefore our plan space should consist of as many as possible WCO, binary, and hybrid plans. We run all generated plans without a cost-based plan optimizer involved. Implementing a cost-based plan optimizer before fully understanding the plan space risks pruning the plan space based on unfounded notions of what should be optimal. However, the number of possible plans is vast. Therefore some pruning is required to keep experiments possible. What exactly is safe to prune will have to established both theoretically and experimentally.

To make these plans, we require both binary and WCO algorithms. There are many options here. Deciding which ones we implement and experiment with should depend on their performance and ease of implementation.

To properly answer our research questions, we will need to experiment with a variety of query types. For each query type, we also require a high number of queries. Certain plans might perform well on one query while not on another query of the same type depending on the labels involved. We also require a graph database instance to execute these queries on, a graph with a large variety of labels and possible queries would be preferred here. Real instead of synthetic data would also be a significant advantage.

Eventually, the goal is to get enough insight into what makes WCO and binary algorithms, and their combinations, perform well under varying circumstances such that the following can happen. A planner with a cost-based optimizer can be built that confidently generates optimal or near-optimal plans, be it hybrid, WCO, or binary plans. Further insight into WCO algorithms and ways to improve them. Currently, WCO algorithms are still relatively new and under active research. While researching hybrid plans, we will also learn possible improvements for general WCO algorithms.

1.3 Objectives

Our goal is to test hybrid plans in a graph database engine. For this purpose, we use Avantgraph. Avantgraph is a main memory graph database engine developed by the Databases group at the Eindhoven University of Technology written in C++. Avantgraph uses pipelined execution plans according to the Volcano-style [6]. In order to test hybrid plans, the following needs to happen:

- We need to implement both binary, and WCO join algorithms with a common interface such that they can interact with each other.
- A planner needs to be built that can compute binary, WCO, and hybrid plans so that we can experiment with them.
- A directed labeled graph database instance consisting of real data is required. To be able to execute queries over this database instance, we will have to build a query miner to retrieve queries.
- A pipeline has to be built that allows us to experiment. This pipeline should be able to take as input several queries and a directed labeled graph. It should then compute for each query all plans and execute them. Relevant data needs to be recorded and stored such that our research questions can be discussed.

Binary join performance: The goal here is to measure the performance of binary joins compared to WCO joins in both pure plans and in hybrid plans. Here we will be interested in their performance in two different cases. In the first case, we want to see how well the algorithms perform on different query types as a whole. This gives us a general insight into what algorithms perform better where. For certain query types, such as cycles or cliques, there are many plans. These plans are the same for each query of that type, but they execute differently if the query is over different labels or a different database instance. If, for a query type, the same plan is always performing best in different instances versus a different plan performing best on different instances will once again give us insight into what makes specific algorithms and or plans better.

Hybrid plan performance: The goal here is to measure the performance of hybrid plans compared to pure WCO and BJ plans. Hybrid plans combine both types of algorithms to compute the SGM tuples. We enumerate and execute a high number of possible hybrid plans. Both left deep and bushy plans. We are interested in how often hybrid plans outperform pure plans and why they outperform them. Running many queries over different query types and interpreting the results through a cost model will tell us how often they outperform pure plans. Afterward, we will analyze individual queries of interest through metrics such as intermediate results sizes, and the number of computations required. This analysis will hopefully give us insights into why and when hybrid plans outperform pure plans. Eventually, this information could be used in a cost-based planner that has hybrid plans in their plan space, which we will leave as future work.

WCO join performance: While our main objective is to explore hybrid plans and not to improve on current WCO join algorithms, we will still explore potential improvements to the current state of the art of WCO joins. However, these improvements will most likely be left as future work. We use the leapfrog triejoin algorithm [13] as our WCO algorithm. This algorithm is designed for a relational database but functions well on a graph database. It was chosen partly for that reason because it made implementation easier. We also choose it because it showed good benchmarks, it includes the novel idea of leapfrogging, and it can intersect more than two relations at the same time. Leapfrog triejoin will be introduced thoroughly in Section 2.3.2. In Section 3.2.1, we explain how we integrated leapfrog triejoin in a graph database that uses hybrid plans.

1.4 Contributions

We summarize the contributions of this thesis in several main areas: adapting and combining binary and WCO algorithms into hybrid plans, a hybrid plan enumerator; leapfrog triejoin implementation in a graph database suited for hybrid plans, a query miner that uses our plans; and the benchmarking and discussion of our plan space and their components. This thesis focuses primarily on these hybrid plans to find and study instances where they are more optimal than either BJ or WCO plans. To study these, we have implemented leapfrog triejoin [13] and several binary join algorithms in Avantgraph. Using leapfrog triejoin as our WCO algorithm, we can combine it with BJ algorithms such as hash joins, sort-merge joins, and nested loop joins to create hybrid plans. This thesis does not focus on cost estimation for plans. Instead, we enumerate a large number of sensible plans. Work has gone in finding out exactly what plans are not sensible and why not. This still leaves a large number of plans that we can experiment with using various workloads.

1. We adapt BJ and WCO algorithms to function together in hybrid plans. Avantgraph uses a pipelined approach to resolve queries, the interface with which the algorithms retrieve results from each other has to be standardized (Section 3.2).

2. We provide a dynamic programming algorithm to enumerate our plan space efficiently. Afterwards, these plans will possibly be pruned to make experimentation possible. What plans are safe to prune will also be established (Section 3.1).

3. As our WCO algorithm, we use the leapfrog triejoin algorithm, which is developed for a relational database. We will adapt the leapfrog triejoin algorithm to function in a graph database and adapt it as needed to work with hybrid plans (Section 3.2.1).

4. To be able to experiment with the query types, a query miner has to be created to retrieve queries from any dataset we run. This query miner will be implemented in Avantgraph and use plans created by our enumerator (Section 4.2.1).

5. To experiment with our algorithms, the mined queries are executed over our plan space. We record data such as running time, intermediate result sizes, and the number of tuples generated by each operator for each plan such that we have a basis for discussion (Chapter 4).

6. Through the work in this thesis, several insights into possible improvements to one of the areas described above but which are not implemented will be discussed as future work (Chapter 5).

The structure of this thesis is as follows. In chapter 2, we elaborate on some notation used throughout this thesis and introduce the context for this work. Chapter 3 contains our contribution. The chapter explains the process of creating hybrid plans and the choices made when selecting what plans to remove. Furthermore, it explains how we implemented our various operators used in our plans. Chapter 4 goes over the results of the experiments ran over numerous different workloads. Finally, Chapter 5 concludes the thesis and presents avenues for future work.

Chapter 2

Preliminaries

The graphs that we use are directed labeled graphs $G(V, E)$ where V is the set of vertices and E is the set of edges. Internally this data is stored based on the *SPO* (subject, predicate, object) format. Each directed edge in the graph can be seen as a SPO triple where the subject is the source of the edge, the predicate is the label of the edge and the target of the edge is the object.

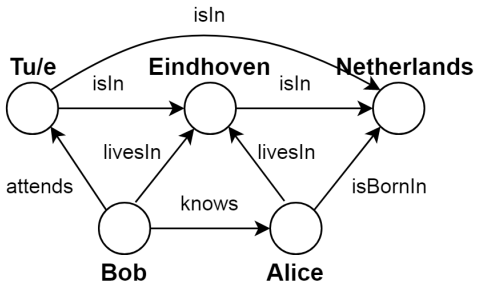


Figure 2.1: **Example of a directed, labeled graph.**

Definition 2.0.1. (Directed, labeled graph G). A directed, labeled graph is a 6-tuple denoted as $G = \langle V, L_V, \lambda_V, E, L_E, \lambda_E \rangle$

- V is the set of all vertices in the graph.
- $E \subset V$ x V is the set of all directed edges in the graph. Each edge connects a subject node to a object node.
- L_V is the set of all vertex labels.
- L_E is the set of all edge labels.
- λ_V is a function mapping to each vertex $\in V$ a single label $\in L_V$.
- λ_E is a function mapping to each edge $\in E$ a single label $\in L_E$.

Example 2.0.1. A small example of a directed, labeled graph is shown in Figure 2.1. This graph is a snippet of a potentially much larger graph. Each vertex in the graph has a label, for example, *Bob* or *Eindhoven*. Similarly each edge has a label that indicates a relationship between two vertices, for example *Bob livesIn Eindhoven*.

Definition 2.0.2. *Subgraph matching query* (SGM). We assume a SGM query $Q(V_Q, E_Q)$ is directed, connected, labeled and has m query vertices $a_1, \ldots, a_m$ and n query edges. Where $V_Q \subset V$ and $V_E \subset E$. A SGM query can be seen as a graph itself that we are trying to find in the larger total graph. A directed edge from vertex a_i to a_j will be denoted as $a_i \rightarrow a_j$. We denote a labeled as as $l(a_i \rightarrow a_j)$, where l is the label. A solution to a SGM query is the set of tuples that map the subgraph $G(V_Q, E_Q)$ onto the graph $G(V, E)$.

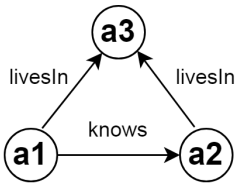


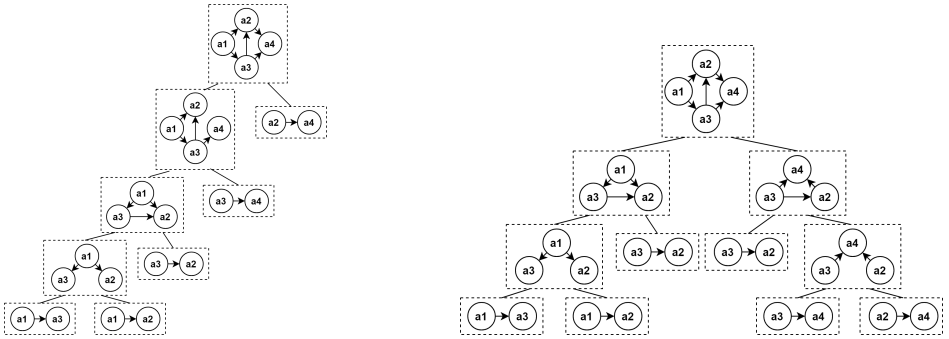
Figure 2.2: **Example of a SGM query.**

Example 2.0.2. SGM queries want to find patterns in the larger graph. For example, the query in Figure 2.2 wants to find the pattern where the following relationships hold: $knows(a1 \rightarrow a2)$, $livesIn(a1 \rightarrow a3)$, and $livesIn(a2 \rightarrow a3)$. Executing this query on the graph in Figure 2.1 yields the output tuple $(Bob, Alice, Eindhoven)$.

Definition 2.0.3. *Query vertex ordering* (QVO). A QVO is an ordering of the vertices in V_Q. The QVO decides in what order the vertices in the SGM query are bound by WCO join algorithms.

The cardinality of an edge table, or relation, will be denoted as $|E|$ where E denotes the edge table.

We implement our system on top of Avantgraph, Avantgraph is a main memory graph database engine developed by the Databases group at the Eindhoven University of Technology written in C++. All edge tables in Avantgraph are sorted by PSO and POS. This allows us to retrieve for each label all edges in sorted order for both directions of the relation.



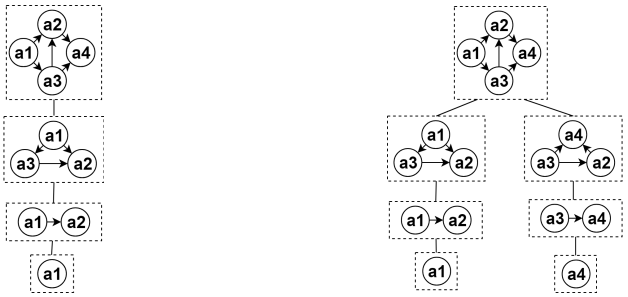
(a) **BJ Left Deep Plan** (b) **BJ Bushy Plan**

Figure 2.3: **Example Binary plan for the subgraph query at the top of the plans**

We present plans in our plan space as a rooted tree as follows:

- The root is labeled by the full query Q
- Leaf nodes are either a single $e \in E_Q$ or a single $v \in V_Q$
- Each internal node $Q_k = (V_k, E_k)$ is a subset of the full query where k denotes the number of vertices. If Q_k has a single child $Q_{k-1} = (V_{k-1}, E_{k-1})$ then Q_{k-1} is a subset of Q_k with one vertex and its incident edges missing. This represents a WCO join where we extend the subgraph Q_{k-1} by one vertex into Q_k. If Q_k has two children Q_{c1} and Q_{c2} where $Q_k = Q_{c1} \cup Q_{c2}$ and $Q_k \neq Q_{c1}$, $Q_k \neq Q_{c2}$. This represent a binary join to compute Q_k where both Q_{c1} and Q_{c2} are subsets of Q_k

Example 2.0.3. Figure 2.3 shows two BJ plans for a subgraph query with four nodes that can be represented by 4 joins over the edges ($a1 \rightarrow a2$, $a1 \rightarrow a3$, $a2 \rightarrow a3$, $a2 \rightarrow a4$, $a3 \rightarrow a4$). The BJ plan in figure 2.3a is a left-deep plan that joins in one edge at a time expanding the subgraph pattern. Figure 2.3b shows a bushy BJ plan that joins subgraphs together. Figure 2.4 shows a WCO and a hybrid plan of the same query.



(a) **WCO Plan** (b) **Hybrid Plan**

Figure 2.4: **Example WCO and hybrid plan for the diamond SGM query at the top of the plans**

We briefly go over the binary join algorithms we use in Avantgraph.

Hash join

The hash join algorithm joins two relations as follows:

- We build a hash table on one of the join relations.
- Once the hash table is built, scan the other relation, for each value scanned look up those values in the hash table. When values are found, we have a join.

Since we work in an in-memory database, we have to be careful with creating hash tables on large intermediate results. Therefore to not run out of memory, large results are partitioned and run in blocks. This is a trade-off between memory usage and runtime that can be fine-tuned based on the system specs.

Let us define M as the size of the relation we are building the hash table on and N the size of the relation that we are scanning. Then the expected runtime of the hash join is $O(N + M)$. The build step goes through the relation once, and probing occurs in constant time. The hash join has unfavorable memory access and works best when the hash table fits entirely into main memory.

Sorted Merge Join

All the base labels are stored in a sorted state. Therefore any join between two base labels can quickly be done by a sort-merge join. The sort-merge join scans both relations at the same time in order, any tuples that are equal on the join predicate are merged to form the result relation.

M and N are as defined before than the expected runtime of the sort-merge join is $O(N + M)$ as we are merely stepping through both relations once. This does not take into account the sorting step, which we do outside of query execution.

Indexed Nested Loop Join

An index nested loop join is a regular nested loop join where the inner relation is sorted and can, therefore, be queried efficiently. We can find any key in logarithmic time using a standard b-tree sorting method. Because of our sorted base labels, we use this method whenever we join a base label with an intermediate result. The intermediate result is the outer join that we only have to iterate through once. For each item in the outer join, we search the inner join, outputting tuples for each value found.

M and N are as defined before than the expected runtime of the index nested loop join is $(N \cdot logM)$, where M is the size of the inner relation that is being queried N times.

2.1 Related Work

Over the past decade, research into WCO algorithms has started with the discovery of the tight AGM bound. Many WCO algorithms have been proposed with sometimes wildly different methods of accomplishing worst-case optimality. One algorithm was even proven to be worst-case optimal after it had already been implemented in a commercial database system. Among these algorithms are *NPRR* [10], *leapfrog triejoin* [13], *minesweeper* [12] and the *generic-join* [11]. There are also several works for the topic that this thesis focuses on, hybrid plans, these works are EmptyHeaded [3], and an approach built on top of Graphflow [7] by Mhedhbi and Salihoglu [8].

This section will briefly explain the intuition behind these algorithms and any contributions of relevance to this thesis. Special attention will also be given here and in further chapters into areas where we differ from their approaches, especially in regards to the works that focus on hybrid plans. Note that the leapfrog triejoin algorithm will be covered in more detail since we use it ourselves.

2.2 AGM Bound

The AGM bound gives us an upper bound on the output size of a query. This bound allows us to reason about worst-case optimality. Illustrating the workings of the AGM bound can also give us insight into when BJ plans violate the AGM bound, and thus when WCO joins might be preferred.

We would like a tight bound on the result size of a SGM query $Q(V_Q, E_Q)$. To illustrate this bound, we use the following cyclic query containing three vertices, $(a1, a2, a3)$ and the relationships between them:

$$Q_\Delta := E_1 \bowtie E_2 \bowtie E_3 \tag{2.1}$$

where $E_1 = a1 \rightarrow a2$, $E_2 = a2 \rightarrow a3$ and $E_3 = a3 \rightarrow a1$ are copies of all edges in the graph. In this case without labeled edges the query can be seen as a repeated self-join. Now the question is what would the upper bound of the output size be of this query as a function of $|E_1|$, $|E_2|$ and $|E_3|$. It can easily be seen that $|Q_\Delta| \leq |E_1| \cdot |E_2| \cdot |E_3|$. One can quickly see as well that because of the triangular nature of the query any result of a join of two relations can afterwards only be further filtered by the third relation. So we arrive at $|Q_\Delta| \leq min(|E_1| \cdot |E_2|, |E_2| \cdot |E_3|, |E_1| \cdot |E_3|)$. The authors Atserias, Grohe and Marx [4] show that with the help of fractional covers this bound can be further reduced to $\sqrt{|E_1| \cdot |E_2| \cdot |E_3|}$, This bound is a tight bound.

Fractional Edge Cover

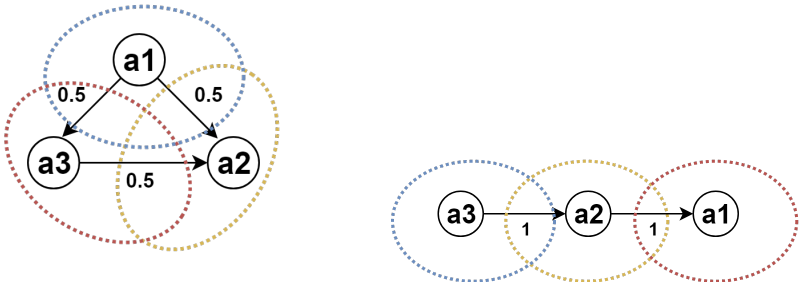
A SGM Q can be converted to a hypergraph instance H trivially when working with graph databases. $H(Q) = (V_Q, E_Q)$, each vertex and edge in Q is a vertex and edge in $H(Q)$. An optimal solution to the fractional edge cover problem of $H(Q)$ provides the tight AGM bound on the output size of Q_Δ. For every query, there can be found a graph $G(V, E)$ for which the output size is bounded by Equation 2.2

$$|Q| \leq \prod_{i=1}^{n} |E_i|^{u_i} \tag{2.2}$$

Where E_i is an edge in the query, and u_i comes from the fractional edge cover problem and is a set of weights assigned to each edge in H such that the total weight on each vertex is at least 1. This is a linear programming problem where each edge has a weight that ranges between 0 and 1. For Query 2.1 the following would have to hold:

$$\begin{aligned} a_1 &= E_1 + E_3 \geq 1 \\ a_2 &= E_1 + E_2 \geq 1 \\ a_3 &= E_2 + E_3 \geq 1 \end{aligned} \tag{2.3}$$

Each edge can be assigned a weight of 0.5 to satisfy these constraints to result at a bound of $\sqrt{|E_1| \cdot |E_2| \cdot |E_3|}$. We visualize this process in Figure 2.5a. Comparing this to Figure 2.5b, which shows the fractional edge cover of a chain with three vertices, we can see that here the sum of edge covers is equal to 2, which is exactly the worst-case when doing a binary join over two relations.



(a) **Fractional edge cover satisfying Query 2.1**

(b) **Fractional edge cover of a chain containing three vertices**

Figure 2.5: **Each colored box contains the weights for that vertex and has to equal to at least 1**

2.3 Worst Case Optimal Algorithms

While the authors of the AGM bound presented a tight bound on the result size of a join query, it does not present a WCO algorithm. The first WCO algorithm is NPRR [10] named after its authors Ngo, Porat, Ré, Rudra. For an extensive survey of worst-case optimality, including a proof of the AGM bound, and the generic-join WCO algorithm, see (Ngo, Ré, Rudra) [11]. The authors of this paper are the same authors that created the NPRR WCO algorithm. The paper briefly explains both NPRR and leapfrog triejoin, both of which are used to introduce their generic-join WCO algorithm. The authors describe two benefits of WCO join algorithm over traditional suboptimal join algorithms.

- The main ideas behind WCO algorithms are an optimal way of avoiding skew.
- WCO algorithms challenge the idea of only doing "*one join at a time*", as is done in traditional database systems. [11]

Both NPRR and leapfrog triejoin are instances of the same generic-join algorithm. To introduce the concept behind both algorithms, we will use the triangle Query 2.1. For clarity, we will name the three vertices of the triangle query A, B, and C, with the edges being as follows. $E_1 = A \rightarrow B$, $E_2 = B \rightarrow C$ and $E_3 = C \rightarrow A$

2.3.1 NPRR

NPRR was the first algorithm designed to be worst-case optimal. Leapfrog triejoin was implemented in a commercial relational database system called Logicblox [1] before it was realised that the algorithm was worst-case optimal. Both NPRR and leapfrog triejoin require a QVO and for all relations in the query to be sorted according to that QVO.

NPRR uses the power of two choices to handle skew. Large intermediate results can be caused by particular nodes having a high degree, or as we will call it here, they are *heavy*. Nodes with a low degree will be called *light*. By distinguishing between heavy and light nodes, we can use different join techniques to deal with them. A particular node a_i is called heavy if:

$$|\sigma_{A=a_i}(E_1 \bowtie E_3)| \geq |Q_\Delta[a_i]| \tag{2.4}$$

where

$$|Q_\Delta[a_i]| := \pi_{B,C}(\sigma_{A=a_i}(Q_\Delta)) \tag{2.5}$$

In other words, a particular node a_i is heavy if its contribution to the intermediate result size of $B \bowtie C$ is greater than its contribution to the size of the final output. The left-hand side of Equation 2.4 can be computed from an index of the input relations. The right-hand side will have to be approximated since we cannot know the intermediate result size until computing the join. NPRR uses $|E_2|$ as a proxy for $|Q_\Delta[a_i]|$, note that $|Q_\Delta[a_i]| \subseteq E_2$. The power of two choices is in how we handle heavy and light nodes. NPRR uses the following ways to compute $|Q_\Delta[a_i]|$, remember Q_Δ is a triangle query:

1. **When a_i is light:** Compute $\sigma_{A=a_i}(E_1) \bowtie \sigma_{A=a_i}(E_3)$ and filter the results by probing against E_2.
2. **When a_i is heavy:** Consider each tuple in $(b, c) \in E_3$ and check if $(a_i, b) \in E_1$ and $(a_1, c) \in E_2$

This ensures that when we are dealing with heavy nodes, we reduce the computation required. Algorithm 1 shows how to compute the triangle query Q_Δ by using the power of two choices. Making the distinction between heavy and light nodes to deal with nodes of high skew efficiently is the essence of the NPRR algorithm.

2.3.2 Leapfrog Triejoin

We will first introduce leapfrog triejoin in the same style as NPRR using the triangle query as a basis. Afterward, we will go further into depth about the relational leapfrog triejoin, as described in the paper by Todd Veldhuizen [13]. Leapfrog triejoin is a WCO join algorithm that is currently being used in the commercial database system LogicBlox [1].

Leapfrog triejoin deals with skew by delaying the computation. It is a multiway join algorithm that transforms the join into a series of nested intersections. Instead of explicitly dealing with the heaviness of a node by changing intersection methods, it gradually filters candidate bindings by looking ahead. Algorithm 2 computes all candidates for $a_i \in A$ by intersection. By then computing the intersection $L_B^a \leftarrow \pi_B \sigma_{A=a} E_1 \cap \pi_B E_2$ we find the candidate set for b for the given binding of a_i. To put it another way, these values of b are the only values that could participate with a_i in the output (a_i, b, c). Then the candidate set for c is computed by $L_C^{a,b} \leftarrow \pi_C \sigma_{B=b} E_2 \cap \pi_C \sigma_{A=a} E_3$, which again means that these values of c are the only possible values in (a_i, b_i, c). When a_i is skewed towards the heavy side, the candidates b and then c help gradually reduce the skew toward building up the final solution.

Algorithm 1: Computing Q_Δ with the power of two choices

input: E_1, E_2, E_3 in sorted order
$Q_\Delta \leftarrow \emptyset$
$L \leftarrow \pi_A(E_1) \cap \pi_A(E_3)$
for *each* $a \in L$ **do**
  if $|\sigma_{A=a}E_1| \cdot |\sigma_{A=a}E_3| \geq |E_2|$ **then**
    for *each* $(b, c) \in E_2$ **do**
      if $(a, b) \in E_1$ *and* $(a, c) \in E_3$ **then**
        Add (a, b, c) to Q_Δ
  else
    for *each* $b \in \pi_B(\sigma_{A=a}E_1) \wedge c \in \pi_C(\sigma_{A=a}E_3)$ **do**
      if $(b, c) \in E_2$ **then**
        Add (a, b, c) to Q_Δ
Return Q_Δ

Algorithm 2: Computing Q_Δ by delaying computation

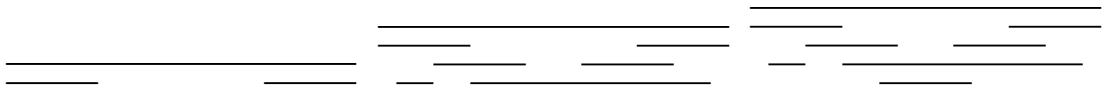
input: E_1, E_2, E_3 in sorted order
$Q_\Delta \leftarrow \emptyset$
$L \leftarrow \pi_A(E_1) \cap \pi_A(E_3)$
for *each* $a \in L$ **do**
  $L_B^a \leftarrow \pi_B\sigma_{A=a}E_1 \cap \pi_B E_2$
  for *each* $b \in L_B^a$ **do**
    $L_C^{a,b} \leftarrow \pi_C\sigma_{B=b}E_2 \cap \pi_C\sigma_{A=a}E_3$
    for *each* $c \in L_C^{a,b}$ **do**
      Add (a, b, c) to Q_Δ
Return Q_Δ

Since leapfrog triejoin is the algorithm we use as our WCO algorithm in this work, we go more in-depth about how it works and some of its implementation details as intended by the original author. In a later chapter, we will present how we adapt and use this algorithm for our purpose, which is using the algorithm in a graph database making use of hybrid plans.

The defining components of leapfrog triejoin are the binding propagation, the multiway nature of it, and leapfrogging. The ability to join multiple relations simultaneously complements the binding propagation. The algorithm being multiway makes it suitable for queries such as star queries or more complex queries that include vertices with high arity.

Leapfrogging is the principle of skipping over data that is guaranteed not to result in a binding, this allows the algorithm to skip over potentially vast amounts of data when the given relations are sparse or vary in size. However, since leapfrog triejoin is also multiway, we can start to skip data on relations that are not sparse but have a low overlap. Figure 2.6 illustrates relations by using a line over a range of values. When the line is filled in, it indicates that the relation contains that value. In Figure 2.6a, we are joining two relations that have high overlap, yet we are still able to skip a large part of the data. However, when we start to expand the number of relations that we are joining, such as in Figures 2.6b and 2.6c, we see that any parts between two relations that do not overlap result in data that we can skip. Since these separate comparisons scale exponentially, it can easily result in a low amount of comparisons required for the join on otherwise

not very sparse relations. In Figure 2.6c this even results in no bindings being found since two of the relations do not overlap. Moreover, because of leapfrogging, the algorithm does not have to go through all the data to establish this.



(a) Two relations (b) Four relations (c) Five relations

Figure 2.6: **Illustration of the potential disjointness of larger number of sets. Each line indicates a relation and the values it contains over an arbitrary range**

We will now go into a more in-depth overview of the *leapfrog triejoin algorithm*, henceforth LFTJ. LFTJ is a fairly straightforward algorithm that iterates through all relations in a join simultaneously to provide the result without any intermediate results. An analysis of this algorithm was performed by Todd Veldhuizen [13], which shows LFTJ to have a running time of $O(Q^* log|N|)$ where Q^* is the AGM bound and $|N|$ is the largest cardinality among the relations of the join.

The basic building block of LFTJ is a unary join called a leapfrog join. This join simultaneously joins a number of unary relations $R_1(x), \ldots, R_k(x)$. To begin, all unary relations are sorted and are presented by linear iterators. These iterators are required to have the following interfaces:

Table 2.1: **Leapfrog triejoin iterator interface**

int key()	Returns the key at the current. iterator position
next()	Proceeds the iterator to the next key.
seek(int seekKey)	Position the iterator to the first key that is $\geq$ seekKey, move the iterator to the end if no such key exists.
bool atEnd()	Returns true when the iterator is at the end.

The *key()* and *atEnd()* methods are required to run in $O(1)$ time, and the *next()* and *seek()* methods are required to run in $O(logN)$ time, where N is the cardinality of the relation that the iterator is on. We can achieve these guarantees by using standard data structures such as B-trees.

Initially, the leapfrog join algorithm holds the k iterators in sorted order in an array $Iter[0, \ldots, k-1]$ by calling Algorithm 3. The idea is to, at each step, take the current lowest value from the iterators and perform a seek operation for the highest value in the iterators. If this value does not exist, then the seek operation goes to the next highest value in the iterator. When the lowest iterator equals the highest iterator, we know we have a value to join on. This process is illustrated more clearly in Figure 2.7 and Algorithm 4.

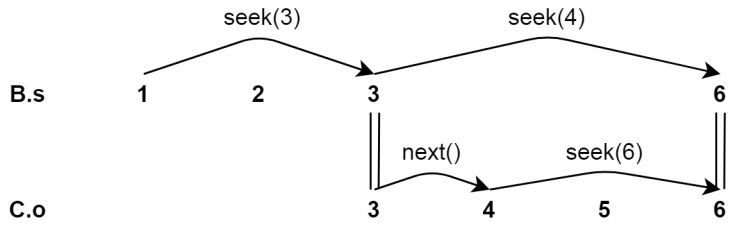


Figure 2.7: **Example of the leapfrog join when joining the target nodes of relation C with the source nodes of relation B. Initially, the k iterators are pointed at (1,3). Then the iterator for B searches for 3, finds it, and we now have a join value. Next() is then called on C, which puts the iterator at the value 4. When B then searches for 4 it arrives at 6. C then searches for 6 and finds another join value. Now both iterators are at the end, so the leapfrog join stops.**

Algorithm 3: Leapfrog-init()

```
1  if any iterator has atEnd() true then
2  |  atEnd := true;
3  else
4  |  atEnd := false;
5  |  sort the array Iter[0..k-1] by keys at which the iterators are positioned;
6  |  p := 0;
7  |  leapfrog-search()
```

Algorithm 4: Leapfrog-search()

```
1  x' := Iter[(p-1) mod k].key();
2  while true do
3  |  x := Iter[p].key();
4  |  if x = x' then
5  |  |  key := x;
6  |  |  return;
7  |  else
8  |  |  Iter[p].seek(x');
9  |  |  if x = x' then
10 |  |  |  atEnd := true;
11 |  |  |  return;
12 |  |  else
13 |  |  |  x' = Iter[p].key();
14 |  |  |  p = p + 1 mod k;
```

Algorithm 5: Leapfrog-next()

```
1  Iter[p].next();
2  if Iter[p].atEnd() then
3  |  atEnd := true;
4  else
5  |  p := p + 1 mod k;
6  |  leapfrog-search();
```

Using the runtime guarantees of the iterator interface, we can analyze the complexity of the leapfrog join. Assuming we have a join over relations $R_1, \ldots, R_k$ with $|R_i|$ being the cardinality

of relation R_i. Let $N_{min} = \min(R_1, \ldots, R_k)$ be the cardinality of the smallest relation in the join and $N_{max} = \max(R_1, \ldots, R_k)$ be the largest. The leapfrog join algorithm works in a fixed pattern where each iterator is advanced once every k steps. An iterator can be advanced at most $|R_i|$ times. Therefore, the number of steps the join takes can be at most $k \cdot N_{min}$. This also illustrates why the leapfrog join is a powerful multiway join algorithm. If one were to join multiple relations of vastly different sizes, then the algorithm will have its runtime dependent on the smallest relation. Furthermore, if we join more than two relations at the same time, we will start skipping more data to also arrive at a lower runtime. So while k increases by joining more relations, the actual number of steps in the join can decrease.

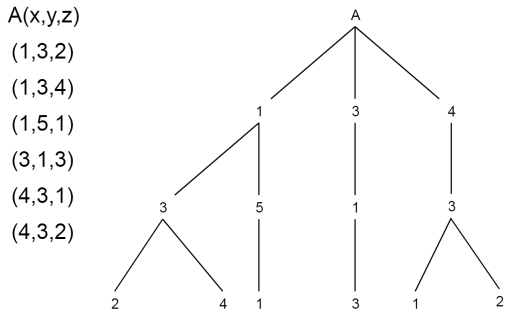


Figure 2.8: **Example of a trie on a relation A with a QVO of (x,y,z). Each tuple of relation A is a path through the trie. Here the relation has an arity of 3**

The algorithm is extended in the following way to process queries containing relations of arity > 1. Each relation in the join is represented by tries, as depicted in Figure 2.8. The iterator interface presented in Table 2.1 is extended with the *open*() and *up*() functions presented in Table 2.2. These functions are called to position all iterators at the attribute that we are currently binding. Therefore the tries have to be sorted according to the QVO.

Table 2.2: Leapfrog triejoin extended iterator interface

open()	Proceed to the first key at the next depth
up()	Return to the parent key at the previous depth.

$$Q(a, b, c) := R(a, b) \bowtie S(b, c) \bowtie T(c, a) \tag{2.6}$$

Now the full algorithm is realized in three layers, the triejoin, leapfrog join, and the trie iterators. On initialization, the triejoin constructs a leapfrog join instance for each variable in the query. Each leapfrog join instance is provided with an array of pointers to the trie iterators of the relations it is joining. Only one instance of a trie iterator is created for each relation. For example, for Query 2.6, the leapfrog join on variable a has access to the trie iterators for both R and T, and the trie iterator for S is provided to the leapfrog joins b and c. The triejoin can use the *open*() and *up*() functions to navigate the trie iterators to the right variable.
Then on a per variable basis, according to the QVO, the triejoin calls the appropriate leapfrog join to provide a binding. When a binding is found for a variable the *open*() function is called on all trie iterators for that variable. The triejoin then goes to the next variable in the QVO and calls that leapfrog join to do the same. If no binding is found for a variable, we backtrack by calling the *up*() function on the trie iterators of that variable and then calling the leapfrog join of the previous variable to provide another binding. This continues until a join on all variables is found or until we hit the end of the first variable.

Consider Query 2.6 and the QVO [a,b,c], the following leapfrog joins would be employed:

$$R(a, \_), T(\_, a)$$
$$R_a(b), S(b, \_)$$
$$S_b(c), T_a(c)$$

Where $R(a, \_) = \{a : \exists b.(a, b) \in R\}$ and $R_a(b) = \{b : (a, b) \in R\}$. The topmost leapfrog join finds bindings for variable a by joining relations R and T. Once such a binding is found the leapfrog join for variable b is called which joins relations R and S. Since relation R was already involved in the join for variable a the *open*() function has been called and the trie iterator has gone down a depth on its trie. So now we iterate over all values for b in relation R given the binding for a. Then for variable C, we join relations T and S similarly using previous bindings.

2.3.3 Generic Join

Algorithm 6 shows the Generic-Join algorithm, which is a generic recursive worst-case optimal algorithm that both Algorithms 1 and 2 are based on.

Here we use the following notation, as before the hypergraph for a query Q is denoted by $H(Q) = (V_Q, E_Q)$ and $I \subseteq V_Q$ is an arbitrary subset of vertices of $H(Q)$. We then define

$$E_{QI} := \{F \in E_Q | F \cap I \neq \emptyset\} \tag{2.7}$$

In other words E_{QI} is all the edges in E_Q that are incident to a vertex in I.

Query decomposition lemma. Let $Q = \bowtie_{F \in E_Q} R_F$ be a natural join query represented by a hypergraph $H(Q) = (V_Q, E_Q)$, and x by any fractional edge cover for $H(Q)$. Let $V_Q = I \oplus J$ be an arbitrary partition of V_Q such that $1 \leq |I| < |V_Q|$; and,

$$L = \bowtie_{F \in E_{QI}} \pi_I(R_F) \tag{2.8}$$

L is the query processed for only the vertices in I. Then, using fractional edge cover,

$$\sum_{t_i \in L} \prod_{F \in E_{QJ}} |R_F \ltimes t_I|^{X_F} \leq \prod_{F \in E_Q} |R_F|^{X_F} \tag{2.9}$$

Algorithm 6: Generic-Join($\bowtie_{F \in E_Q} R_F$)

input: Query Q and hypergraph $H(Q) = (V_Q, E_Q)$
$Q \leftarrow \emptyset$
if $|V_Q| = 1$ **then**
  Return $\bigcap_{F \in E_Q} R_F$
Pick I arbitrarily such that $1 \leq |I| < |V_Q|$
$L \leftarrow$ Generic-Join($\bowtie_{F \in E_{QI}} \pi_I(R_F)$)
for *every* $t_I \in L$ **do**
  $Q[t_I] \leftarrow$ Generic-Join($\bowtie_{F \in E_{QJ}} \pi_J(R_F \ltimes t_I)$)
  $Q \leftarrow Q \cup t_I \times Q[t_I]$
Return Q

The idea of Algorithm 6 is to recursively decompose the full query until only joins on single vertices are left. The results from these joins can then combined into the output.

Now we show how Algorithm 1 is Algorithm 6 for the triangle query Q_Δ by solving sub-queries $Q[t_I]$ by using the power of two choices. In particular, we compute $Q[t_i]$ in the following way

$$Q_{[t_I]} = R_J \bowtie (\bowtie_{F \in E_{Q_J} - \{J\}} \pi_J(R_F \ltimes t_I)) \tag{2.10}$$

We solve the subquery $Q[t_I]$ using the power of two choices. Here we set $I = \{A\}$, which in turn implies that $J = \{B, C\}$. This means step 5 of algorithm 6 becomes:

$$L = \pi_A(E_1) \bowtie \pi_A(E_3) = \pi_A(E_1) \cap \pi_A(E_3) \tag{2.11}$$

Which is the same L as in Algorithm 1. Now we loop over all $a \in L$ as in Algorithm 1. We then compute the subqueries using the power of two choices, similarly as in Algorithm 1.
Algorithm 2, the leapfrog triejoin algorithm, is an instantiation of Algorithm 6 where $V_Q = [n]$ and $I = \{1, \ldots, n-1\}$. Taking $I = \{A, B\}$, implying $J = \{C\}$ we can solve the triangle query in the following way. Step 5 of Algorithm 6 returns $L = \{(a, b) | a \in L_A, b \in L_B^a\}$. In other words step 5 returns binding pairs for a and b for the chosen I. Now for each binding pair (a, b) we wish to obtain $\{(a, b) \times L_C^{a,b}\}$ where

$$L_C^{a,b} = \pi_C(\sigma_{B=b}(E_2)) \bowtie \pi_C(\sigma_{A=a}(E_3)) = \pi_C(\sigma_{B=b}(E_2)) \cup \pi_C(\sigma_{A=a}(E_3)) \tag{2.12}$$

Which are the exact steps performed in algorithm 2.

2.3.4 Minesweeper

Another worst-case optimal join algorithm is Minesweeper [12]. Minesweeper computes the output of a join by starting from the full potential output space and then reducing from there. The main idea here is that often there are fewer tuples part of the output than are not. Therefore we can quickly rule out large areas of the potential output space. The main loop of the Minesweeper algorithm looks at so-called '*free tuples*'. These tuples are selected from the output space. Minesweeper starts by selecting a free tuple arbitrarily. The input relations are then probed to verify whether this tuple is part of the output; if this is not the case, the algorithm returns a '*gap box*'. These gap boxes can be seen as multi-dimensional rectangles in the output space in which no output tuple exists. If the tuple is part of the output, a gap box containing just that tuple is created. A data structure called the '*constraint data structure*' or CDS stores these gap boxes. We use the CDS to select further free tuples efficiently. The algorithm proceeds until gap boxes cover the entire output space. Minesweeper also requires its input relation to be indexed sorted according to a QVO.

Minesweeper differs from the other WCO algorithm presented so far in that it works backward from the total output space down to the result. Furthermore, while it is not very evident at first glance the algorithm still requires a QVO and essentially solves the query vertex-by-vertex. The QVO is required for the CDS. Each gap box is encoded as a list of intervals on the attributes of the query. For instance for the triangle query with three attributes, E_1, E_2 and E_3, an arbitrary gap box could be the following

$$gapbox = \{*, (1, 4), 3\}$$

This gap box indicates that for $E_3 = 3$, $1 < E_2 < 4$ there are no result tuples. The $*$ indicates that any value in the domain of that attribute is possible.

Interestingly the authors note that their algorithm performs better on acyclic queries compared to leapfrog triejoin while performing worse on cyclic queries. They also note that for β-acyclic queries, Minesweeper is instance optimal up to a log N factor, which is interesting for graph databases

where all acyclic queries are β-acyclic [5].

The authors of the survey [11] present as part of their conclusion an open question that is of interest here as well; it is the following:

- *A natural question to ask is whether the algorithmic ideas that were presented in this survey can gain runtime efficiency in database systems. This is an intriguing open question: on the one hand, we have shown asymptotic improvements in join algorithms, but on the other, there are several decades of engineering refinements and research contributions in the traditional dogma.* [11]

WCO join algorithms have a theoretical advantage over binary join algorithms, yet they do not have the weight of decades of research behind them. Therefore the overarching question whether or not they have practical relevance is an important topic of research. While this is not a question we can fully answer in this thesis, the work done here will be a step along the way.

2.4 Hybrid Approaches

We now introduce two different approaches that utilize a hybrid plan space combining both binary and WCO algorithms. First, we present EmptyHeaded [3] and then present an approach by Mhedhbi and Salihoglu [8].

2.4.1 EmptyHeaded

EmptyHeaded is a high-level relational engine that combines both WCO and binary algorithms and is designed to leverage single-instruction multiple data (SIMD) parallelism to its fullest potential. While it is a relational engine, it is optimized for SGM queries. EmptyHeaded uses the generic-join introduced earlier as their WCO join algorithm. The authors note that unoptimized set intersections often account for 95% of the overall runtime in the generic worst-case optimal join algorithm. Therefore they put focus on optimizing intersection through the use of SIMD instructions.

To generate their plan space EmptyHeaded uses *generalized hypertree decompositions* (GHDs). A GHD D is a tree decomposition of the query Q. Each node in D is a subquery of Q. Each subquery is then evaluated using the generic worst-case join algorithm and materialized into an intermediate result. Then starting from the leaves, these intermediate results are joined into their parents until we resolve the full query. A GHD of a single node is a query plan that binds all vertices in order by a WCO algorithm. Joining different nodes together is done by a binary algorithm, and therefore EmptyHeaded's plan space is hybrid. For each query, there are many valid GHDs. Selecting the lowest cost GHD is one of the goals of the query optimizer. The cost of a GHD is defined as the sum of the runtime of its nodes.

A problem with generating the plan space this way is that a GHD consisting of a single node executes a full WCO plan with an arbitrary QVO. Since the QVO can have a significant impact on the run time of a query, this can skew the results. Similarly, while the order of the subqueries in the GHD being joined together is determined by a heuristic, the QVO inside the subqueries is once again arbitrary. The plan space also does not allow for plans which use WCO joins after binary joins. Emptyheaded always computes the sub-queries using WCO algorithms and then uses binary joins. Another issue is that the plans generated depend only on the input query and not on the database instance. Thus it does not optimize in the query planner for the data it is joining. However note that Emptyheaded selects the data layout and intersection methods used at runtime depending on the data, but this does not affect the execution order of the query.

(a) **Barbell Query** (b) **Possible GHD of a barbell query**

Figure 2.9: **GHD example**

Example 2.4.1. Figure 2.9a shows a barbell query. The query can be seen as two 3-cycles joined by a single edge in the middle. Figure 2.9b shows a possible GHD for the barbell query, each node in the GHD denotes a join. This particular GHD would be solved as follows: first, both of the 3-cycles $(a1, a2, a3)$ and $(a4, a5, a6)$ are computed using a WCO algorithm. These intermediate results are materialized and then for each $a3 \to a4 \in E4$ output all 3-cycles that contain $a3$ and $a4$ in the correct position. This plan runs in $O(N^{\frac{3}{2}} + OUT)$ as we are merely computing the 3-cycles and then enumerating the output.

Because density skew in the graph data can make it challenging to apply SIMD, EmptyHeaded created an optimizer that selects among different data layouts to optimize for SIMD. Five different SIMD set intersection methods are implemented in EmptyHeaded. One of these is similar to the 'leapfrogging' in leapfrog triejoin but is implemented to make use of SIMD instructions. They note from their experiments that leapfrogging is rarely worth it compared to block-wise set intersections using SIMD. Only when the ratio between the cardinality of two intersection sets is higher than 32:1, do they select the leapfrogging algorithm. Because leapfrogging performs a set intersection in time with the lowest cardinality. However, they assume set intersections with only two sets at once. In our implementation, we allow any number of sets to be intersected simultaneously, which should give different results. Namely on smaller ratios leapfrogging should perform well.

2.4.2 Graphflow

This approach has a plan space that will be more similar to ours. It is built on top of the Graphflow DBMS [7]. Their WCO algorithm is the generic-join algorithm, similar to the one in EmptyHeaded. As a binary join, they use a classic hash join. The authors generate their plan space by enumerating all k-vertex sub-queries Q_k of Q for $k = 2, \ldots, m$, by using two methods.

- A binary join of two smaller sub queries
- Extending a sub query Q_{k-1} by one vertex

This generates a plan space of all possible WCO plans, binary plans, and a large number of hybrid plans. They show that this plan space is a superset of the plan space generated by EmptyHeaded. Note that this method allows for plans that mix WCO and binary algorithms in any order, unlike the plan space from EmptyHeaded. When extending a subquery Q_{k-1} by a vertex, they perform two-way in tandem intersections. So if adding a vertex requires three or more relations to be joined then these relations are not joined at the same time which means it cannot abuse disjointness between all the sets as illustrated in figure 2.6.

The main contributions of the authors are the following:

- The aforementioned hybrid plan space enumerator
- a cost-based optimizer that can compare the costs of WCO and binary algorithm when evaluating the cost of hybrid plans.

- An adaptive technique that can alter the QVO during query execution based on computed results so far.

To rank WCO plans the optimizer uses a cost metric they call *intersection cost* (i-cost). The i-cost represents the amount of intersection work a plan will perform. The i-cost is defined as the size of adjacency lists that will be accessed and intersected by different WCO plans. The hash join performs a different computation then the WCO join. Therefore, the cost of a hash join needs to be normalized compared to the i-cost so that cost estimates can be made for hybrid plans that can be compared with both binary and WCO plans. The idea here is to set the weights on the following function experimentally: $w_1 n_1 + w_2 n_2$. Where n_1 is the number of hashed tuples, and n_2 is the number of probes into the hash table. Once these weights are set such that WCO joins and hash joins can be compared accurately, hybrid plans can be given a cost as well.

Whenever a chain of two or more vertex bindings exist in either a hybrid or WCO plan, an adaptive operator can be used. The idea is to adhere to the QVO generated by the query planner for the first two vertices in the chain and to then use the actual results to potentially update the QVO. The initial QVO is determined by taking the cumulative i-costs, but these i-costs are estimations that can now be updated by using the partial result. Take, for example, a QVO of $(a1, a3, a4, a2)$ on an arbitrary query with four vertices. It is possible to change the QVO to $(a1, a3, a2, a4)$ after binding vertices $a1$ and $a3$ if this is determined to be more efficient. Changing the QVO is determined on a per tuple basis of the partial result. Their experiments show a speedup on most queries ran by using adaptive QVO's. However, the main goal of adaptive QVO's is to protect the planner from choosing a bad QVO. Since the difference between the best and worst adaptive plans is smaller than the difference between the best and worst fixed plans.

Chapter 3

Hybrid Plan Space and Algorithms

So far, we have discussed both WCO and BJ algorithms in separation and introduced hybrid plans. In this chapter, we will elaborate on the reasoning for creating hybrid plans, and on how we enumerate them. We do not create a cost-based plan optimizer. Instead, we enumerate a large number of plans, only removing some inefficient approaches. We then, through experimentation, find the strengths and weaknesses of different hybrid approaches. We did not create a plan optimizer because it allows us to explore the entire plan space to find these strengths and weaknesses. Implementing a cost based plan optimizer before fully understanding the plan space risks pruning the plan space based on unfounded notions of what should be optimal.

Combining WCO algorithms with traditional binary algorithms is nothing new [8][3]. However, we implemented a different WCO algorithm into our hybrid plans. We also look at a large variety of plans in our experiments due to not including an optimizer yet. Furthermore, this chapter introduces our implementation of the leapfrog triejoin and how we adapt it to work with hybrid plans in a graph database. In hybrid plans, the different algorithms have to communicate with each other to pass results tuples between them. For this to happen, the interfaces across all algorithms have to be made similar. Especially for the leapfrog triejoin algorithm, we had to make changes to allow it to be used in any place in a hybrid plan.

3.1 Full Plan Space

When dealing with just WCO plans, we can describe the full plan space can as all possible QVO's, which are all permutations of the vertices in the query. Similarly, for binary plans, we have to enumerate all possible valid edge permutations to create all left-deep plans and enumerate all valid subgraph joins to create all bushy plans. We now wish to extend that by combining vertex by vertex algorithms with edge by edge algorithms. For our WCO algorithm, we use the leapfrog triejoin. This algorithm is optimal both when edge by edge algorithms violate worst-case optimality, for instance, cycles and cliques. Furthermore, when many relations have to be joined at the same time, for instance, star queries. However, on many query types, traditional join algorithms are still competitive, especially when worst-case optimality is not violated, chain queries, for example. But what about more complex queries that have characteristics that might apply to both, in these situations hybrid plans could be the way to go. By applying each join algorithm in the section where it is optimal, we should get the best of both worlds.

Example 3.1.1. The bowtie query in Figure 3.1 is a nice example of this. The query is two 3-cycles joined together. We know that BJ algorithms violate worst-case optimality when computing

(a) **Bowtie Query** (b) **With fractional edge cover**

Figure 3.1:

3-cycles. Therefore a WCO algorithm should be optimal here. However, since it is two 3-cycles joined together, perhaps computing both 3-cycles with a WCO algorithm and then joining them together with a BJ algorithm could be quicker than a pure BJ or WCO approach.

Using the hypergraph of the query as shown in Figure 3.1b shows us that the bowtie query has a fractional edge cover of $(\frac{3}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{3}{4})$, with cost totaling $\frac{3}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{3}{4} = 2.5$. Assuming no labels, this tells us that the worst-case output of this query is $N^{2.5}$ tuples. A WCO algorithm can run this query in $O(N^{2.5})$ time as guaranteed by its worst-case optimality. If we, however, combine both WCO and BJ algorithms, we can create the following plan: we first compute the 3-cycles (a1, a2, a3) and (a4, a5, a6) independently and then join those together with the BJ algorithm. Each 3-cycle has a cost of $\frac{5}{4}$ thus a worst-case output of $N^{\frac{5}{4}}$. Which, in the worst-case results in $N^{\frac{5}{4}} \cdot N^{\frac{5}{4}} = N^{2.5}$ outputs, but on average we could see an improvement here. When joining both 3-cycles, we are likely to see a lower amount of output tuples than the worst case. Therefore because this plan is able to run in $O(N^{\frac{5}{4}} + OUT)$, we can see run time improvements here. Chapter 4 will dive deeper into this.

Each 3-cycle here has a worst-case output size of $N^{\frac{5}{4}}$ instead of $N^{\frac{3}{2}}$ because of our WCO algorithm filtering on edges not present in the cycle. For instance when computing the 3-cycle (a1, a2, a3) when expanding by the vertex a3, we also filter on edges $(a3 \rightarrow a4)$ and $(a3 \rightarrow a5)$, so it does not compute all actual cycles of (a1, a2, a3), it filters further.

To create hybrid plans, we follow a bottom-up approach to enumerate all possible plans efficiently. We are combining edge by edge and vertex by vertex approaches to create hybrid plans. Therefore the final plan and its sub-plans are represented by the edges it satisfies in the query. We represent our queries as edges instead of vertices because we need to accommodate for both types of algorithms. A vertex can be represented by the edges incident to it. However, vertices cannot, in a straightforward way, represent an edge. Meaning it is not possible to present a subgraph of a query by just its vertices because it is possible that not all edges incident to the vertices are in that subgraph.

Example 3.1.2. A triangle query has three edges; each edge is represented by an id. We represent a full plan for the triangle query in the form of $(1, 1, 1)$. This indicates that each edge of the query is represented in the plan. The sub-plan $(1, 0, 0)$ would indicate that only the first edge is present in the plan. The idea is to store for each edge configuration the possible plans. Approaching it from the bottom up, we can then combine plans using joins to reach the full query eventually. Note, however, that we only consider connected edge configurations to remove any cartesian products.

For binary joins we can easily combine $(1, 0, 0)$ and $(0, 1, 0)$ to form a plan for $(1, 1, 0)$. Or combine $(1, 0, 1)$ and $(0, 1, 0)$ to form a plan for the full query $(1, 1, 1)$, this process is illustrated in figure

3.2. It gets slightly more involved in the case of vertex by vertex joins. For each vertex, we need to keep track of the edges incident to that vertex. With that knowledge we can combine sub plans with vertex by vertex joins, in the triangle query each vertex has two incident edges so combining sub-plans $(1, 0, 1)$ and $(0, 1, 0)$ to form a plan for the full query $(1, 1, 1)$ can be accomplished using a vertex by vertex join.

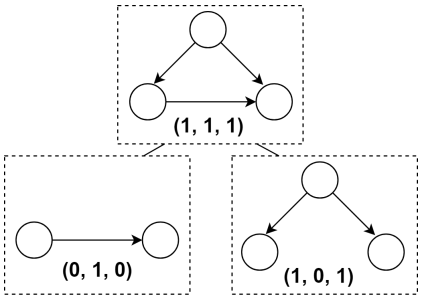


Figure 3.2: **Combining the sub plans with edges (1, 0, 1) and (0, 1, 0) to form the full plan (1, 1, 1)**

3.1.1 Plan Restrictions

In this section, we elaborate on some restrictions on our plan space. The plan space grows exponentially with the size of the query. Since many of these plans are theoretically inefficient, we can safely exclude them from our experiments. Below we list the restrictions we put on our plan space.

1. We only want to consider connected sub-plans to avoid cartesian products. The size of a cartesian product is the product of the cardinalities of the input relations. Plans which involve these cartesian products are likely to have massive intermediate result sizes and are therefore inefficient.

2. We do not consider single edges created by WCO algorithms when creating hybrid plans. We create a single edge with a WCO plan by binding the two vertices of that edge. In essence, it would filter the edge table based on the incident edges of the two vertices in question. To then use that edge in a hybrid plan would be inefficient in most cases compared to merely joining in the base labels. There could be cases where the edges produced by a WCO join would be filtered substantially compared to the base label. In those cases, it can be possible that the computation time required to produce the filtered edge table is lower than the extra computation time that joining in the normal base label would produce. However, disallowing this removes a large number of potential plans from our plan space, which is required to run our experiments on larger queries. Experimenting with filtered edge tables could be of interest in future work.

3. When adding a vertex to a sub-plan, we only add that vertex by using all edges from that vertex to that sub-plan. Consider the case as seen in Figure 3.3. This is necessary because we only want to use our WCO algorithm when it can join all edges incident to the vertex at once. This also means we do not end up with plans where we add one incident edge of a vertex through a binary join. To then, subsequently, add the vertex through a WCO algorithm. This would mean performing inefficient duplicate work.

4. We push so-called lonely variables to the back of all our WCO query plans. We define lonely variables as vertices that are not present in any join. These vertices are only connected by one edge to the subgraph; an example of this can be seen in Figure 3.4, here variables $a4$ and $a5$ are lonely variables. An issue arises that creates inefficiencies when we enumerate

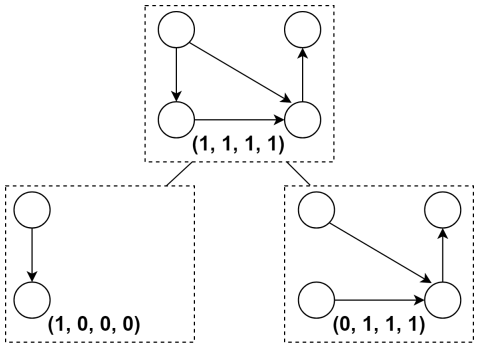


Figure 3.3: **We expand the plan on the bottom left by adding the bottom right vertex. This vertex is connected to three other vertices in the final query, therefore by adding the vertex all three edges have to be added to the sub plan as well.**

bindings on these lonely variables before the end of the query plan. For instance, take as an example a QVO of $(a1, a2, a4, a3, a5)$, which is a valid QVO according to our previous rules. However, the following happens, first bindings are found for $a1$ and $a2$. These are found by joining the appropriate relations. The binding for $a4$, however, does not require a join, it simply enumerates all possible values given the binding for $a2$. When we then close the triangle by binding $a3$, we have to do so for each of the valid bindings of $(a1, a2, a4)$. Now for each unique binding combination of $(a1, a2)$, we have to close the triangle as many times as there are enumerations of $a4$. If we instead follow the QVO $(a1, a2, a3, a4, a5)$, we only have to compute the bindings for $a3$ once for each combination of $(a1, a2)$. To remove these unnecessary computations, we move all lonely variables to the end of our WCO plans. This is not required for binary join plans because they enumerate these results regardless, so this is not an optimization for them.

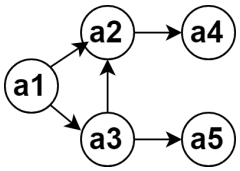


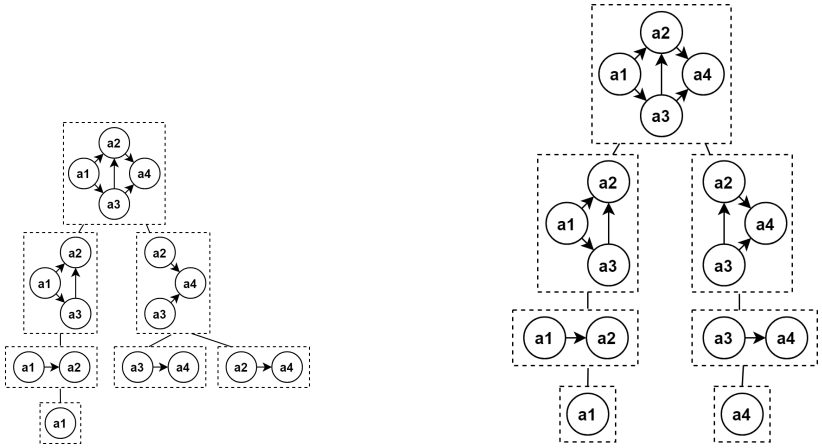
Figure 3.4: **Example query with lonely variables $a4$ and $a5$.**

This example also showcases perfectly the power of WCO algorithms. In this case, how the leapfrog triejoin algorithm delays the computation to reduce the number of intermediate results by enumerating the lonely variables last.

5. A binary join has a left and a right child, for each unique pairing of left and right children, we generate only one plan. This means that if left/right is already in our plan space, we do not include right/left as well. This vastly cuts down on our plan space, with only a minor negative impact on our experiments, since our binary join operators try to optimize based on the children regardless of which one is on the left or right. We potentially create a worse plan when a hash join builds the hash map on the larger child instead of the smaller one because of the ordering created. This cannot always be avoided without prior knowledge of the size of intermediary results.

6. There are situations where we want to include the same edge multiple times in our plan. Take, for instance, the query in Figure 3.5, two potential plans can be seen here. In Figure 3.5a the edge $a3 \rightarrow a2$ only appears once but in Figure 3.5b it appears twice. This might seem inefficient since we are doing some computations twice, but it reduces computation time for the final binary join. The edge $a3 \rightarrow a2$ closes the cycle and can only reduce the

number of intermediary results computed. Therefore whenever we have an edge that closes more than one cycle in the query, we create plans of the form seen in Figure 3.5b where we join the cycles on that edge. In Chapter 4, we will look into how these plans perform relative to each other.



(a) **Plan without duplicated edge** (b) **Plan with duplicated edge**

Figure 3.5: **Diamond query**

7. We prune any hybrid plan that joins two sub-plans that intersect on at least three vertices. Figure 3.6 shows an example of such a join. Pruning these types of plans greatly reduces our plan space on some queries. Initially, we included these plans in our experiments; through these experiments, we can confidently say that these plans are safe to prune.

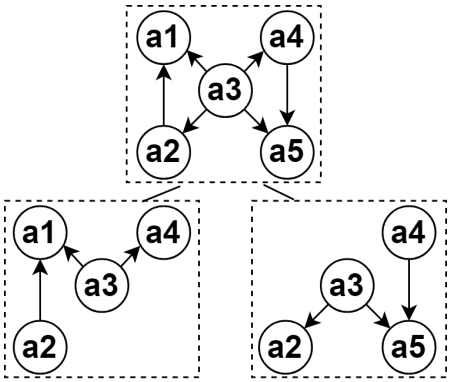


Figure 3.6: **Hybrid plan for the bowtie query joining on three vertices.**

3.1.2 Plan Enumeration

Now we can present the full algorithm we use to generate a wide variety of plans while still reducing the number of plans generated by eliminating known suboptimal plans. The algorithm combines the bottom-up enumeration of all plans with the restrictions presented before to accomplish this. We split the algorithm into the following cohesive pieces that we will explain in turn. It is shown in Algorithm 7, we indicate what lines correspond to each of the steps below.

1. Finding all valid subsets of edges.
2. Create all one edge plans, lines 2-22.

3. Fill out the rest of the memoization table until we arrive at the final plan, lines 23-41.

Valid Subsets

Finding all valid subsets of edges given the query is the first step of our algorithm. A valid subset is a set of edges that together form a connected subpattern of the query. For the bowtie query seen in Figure 3.1a the edges $(a2 \rightarrow a1, a2 \rightarrow a3, a1 \rightarrow a3)$ form a valid subpattern, this pattern is the left triangle of the query. But for instance the edges $(a2 \rightarrow a1, a2 \rightarrow a3, a5 \rightarrow a4)$ would not form a valid subpattern since they are not connected.

When creating all valid subsets, we also have to take into account the plan restriction rule regarding lonely variables. These restrictions only apply when extending a subpattern by a vertex; they do not apply to edge extensions. We can split up a query into a subpattern without the lonely variables and one with the lonely variables. We do not want to create subpatterns that have lonely variables without the complete subpattern involved as well. Using Figure 3.4 the lonely variables are vertices $(a4, a5)$ and the complete subpattern are the edges $(a1 \rightarrow a2, a1 \rightarrow a3, a3 \rightarrow a2)$. Thus a subset containing either edge $a2 \rightarrow a4$ or edge $a3 \rightarrow a5$ without containing the full subpattern is not a valid subset when extending by a vertex.

Since there is a difference in the valid subsets when extending by an edge or by a vertex, we create two sets of valid subsets.

To compute all valid subsets for edge extensions, we find all permutations of the possible orderings of edges and then for each permutation check if it is a valid subset. We verify if a subset is valid by building the subgraph one connected edge at a time. We start with an arbitrary edge in the subset; then, we loop over all remaining edges until we find one that connects to the current subgraph. We repeat this procedure until we either have completed the full subset; in that case, the subset is connected. Or until we have iterated over the remaining edges and find that none are connected to the subgraph, in that case, the subset is not connected.

To compute all valid subsets for vertex extensions, we have to take into account lonely variables and therefore use the following steps. These steps are an extension of the previous steps:

- Find all lonely variables and keep their edges apart for now.
- For the subpattern that is left, which contains the edges that are not incident to lonely variables, we compute all valid connected subsets, as done for the edge extension case.
- Add the lonely variable edges to the valid subsets.
- Compute each possible permutation of the lonely variable edges. Add each permutation combined with the full subpattern to the valid subsets.

One Edge Plans

Now we can start generating our plans. For each valid subset, we keep track of the plans generated for that subset. Lines 2-22 in Algorithm 7 show how we generate the one edge plans.

We separate one edge plans from the total plan space because we create them through a different method for both WCO and binary join algorithms. For a binary join algorithm, a single edge plan does not require a join; it is merely a base label scan. For a WCO algorithm a single edge plan requires you to bind both vertices defining that edge. Essentially we are creating a filtered edge table by doing this. These bindings can be done in two orders. For each order, we create a plan, where we first bind one vertex and then the other. In the case that the edge is incident to a lonely variable we only create one order where the lonely variable vertex is bounded second. We do this

because we cannot start a join plan on a vertex that requires only enumeration. Note that this means that we allow hybrid plans where we bind a lonely variable before all non lonely variables are bound. The implications of not allowing lonely variables in single edge plans are not as clear immediately as they are with pure WCO plans. Thus we decided to include these plans in our experiments.

For every single edge, we create two or three plans. The generated plans are stored together based on what subset they are.

Filling Out the Memoization Table

Now that we have plans for all single edges, we can generate all further sub plans from the bottom up. Lines 23-41 in Algorithm 7 show how we generate the remaining sub plans, each step will be elaborated on after.

After Algorithm 7 is finished, the entry in allPlans for the full query contains all the full plans generated. The algorithm first goes over all subsets containing two edges up to the final subset containing the entire query. For each number of edges, we build all possible plans that adhere to our rules. We can then, from the bottom up, combine these plans to compute our final plans efficiently. The rules in the algorithm refer to the ones presented in the previous section. The algorithm is split up into three main parts. For each sub-plan, we compute all WCO, binary, and shared edge plans. This way all hybrid plans get created at the same time. For instance, binary sub plans can get extended by WCO algorithms to form hybrid plans.

Function overview:

- v_I on line 28 denotes all incident edges of the vertex v.
- findEdge(subset): Return the edge that make up the subset.
- findVertices(subset): Return the vertices in the subgraph denoted by the subset.
- validSubsets(i): Computes all connected sub plans that contain i number of edges using the edge extension case.
- validWCOSubsets(i): The same as above for the vertex extension case.
- createWCOJoin(vertex, plan): Given a plan and a vertex, it creates a WCO join plan that extends the plan by the vertex.
- constructPossibleJoins(plan_s1, plan_s2): Given two plans create all possible plans that use binary joins to join these two plans. This means if there are multiple vertices to join on, we enumerate over all possible permutations of join orderings.
- sharedEdgePlans(subplan): computes all pairings of sub-plans that are two cycles with one common edge and combine them into the given sub-plan, see rule 6 in the previous section.

Example 3.1.3. To use Figure 3.4 as an example, we will go through the steps in the algorithm. For ease of notation, the full query is represented by $(1, 1, 1, 1, 1)$ to indicate that each edge is in the subset and by $(0, 0, 0, 0, 0)$ to indicate the empty subset. The edges are ordered in the following way; ($a1 \to a2$, $a1 \to a3$, $a3 \to a2$, $a2 \to a4$, $a3 \to a5$). The example will first go over the steps that generate the valid sub plans and subsequently how those sub plans are used to create our plan space.

We will now illustrate how we create the sub plans, remember that we have to create two different sets, one for the vertex extension case and one for the edge extension case. In the figures below, the colored subgraphs denote subgraphs that are in the vertex extension set. While all subgraphs

Algorithm 7: Bottom up Algorithm to Generate the Plan Space

```
   input : A SGM query Q(V_Q, E_Q)
1  allPlans ← ∅;
   // One edge plans
2  for Each subset in validSubsets(1) do
3      edge ← findEdge(subset);
4      allPlans[subset].add(edge);
5      v_1, v_2 ← findVertices(subset);
6      currentPlan ← ∅;
7      if v_1 is a lonely variable then
8          currentPlan ← createWCOjoin(v_2, currentPlan);
9          currentPlan ← createWCOjoin(v_1, currentPlan);
10         allPlans[subset].add(plan)
11     else if v_2 is a lonely variable then
12         currentPlan ← createWCOjoin(v_1, currentPlan);
13         currentPlan ← createWCOjoin(v_2, currentPlan);
14         allPlans[subset].add(plan);
15     else
16         currentPlan ← createWCOjoin(v_2, currentPlan);
17         currentPlan ← createWCOjoin(v_1, currentPlan);
18         allPlans[subset].add(plan);
19         currentPlan ← ∅;
20         currentPlan ← createWCOjoin(v_1, currentPlan);
21         currentPlan ← createWCOjoin(v_2, currentPlan);
22         allPlans[subset].add(plan);
   // Filling out the memoization table
23 for i ← 2 to |E_Q| by 1 do
24     for Eachsubset ∈ validSubsets(i) do
           // creating WCO plans
25         if subset ∈ validWCOSubsets() then
26             for v ∈ V_Q do
27                 if Removing v follows rules 1 and 3 then
28                     Separate the subset s into s_1, s_2 where s_1 ← v_I and s_2 ← s \ s_1;
29                     for plan ∈ allPlans[s_2] do
30                         allPlans[subset].add(createWCOJoin(vertex, plan));
           // creating Binary plans
31         for All possible ways to split the subset into s1, s2 do
32             if Both subsets follow rule 1 then
33                 for plan_s1 ∈ allPLans[s1] do
34                     for Each plan_s2 ∈ allPlans[s2] do
35                         for join ∈ constructPossibleJoins(plan_s1, plan_s2) do
36                             allPlans[subset].add(join);
           // Rule 6, similar to binary joins
37         for s1, s2 ∈ sharedEdgePlans(subset) do
38             for Each plan_s1 ∈ allPLans[s1] do
39                 for Each plan_s2 ∈ allPlans[s2] do
40                     for Each join ∈ constructPossibleJoins(plan_s1, plan_s2) do
41                         allPlans[subset].add(join);
   output: allPlans array containing our generated plan space
```

are part of the edge extension set. The vertex extension case is only required for the WCO plan space due to having to push back lonely variables. This set of sub plans will always be a subset of the edge extension case. The steps to compute both sets are as follows:

- For each possible number of edges we compute all connected subsets. This step computes all the subgraphs as seen in the figures below. After this step we have to compute which of these subgraphs are valid in the vertex extension case.
- The lonely variables are vertices $(a4, a5)$, their edges are respectively $(a2 \rightarrow a4, a3 \rightarrow a5)$. Keep these edges apart for now.
- The remaining edges are $(a1 \rightarrow a2, a1 \rightarrow a3, a3 \rightarrow a2)$, which are denoted as the subset $(1, 1, 1, 0, 0)$. For this sub pattern we have to compute all connected subsets. As these sub patterns are part of the vertex extension case they are shown in color in the figures below. Figures 3.7, 3.8 and 3.9 show these sub graphs in color.
- We now add the lonely variable edges to the valid subsets. This means adding the subsets $((0, 0, 0, 1, 0), (0, 0, 0, 0, 1))$ to the valid subsets.
- Each possible permutation of the lonely variable edges are the subsets $((0, 0, 0, 1, 0), (0, 0, 0, 0, 1), (0, 0, 0, 1, 1))$. Each of these is combined with the full sub pattern $(1, 1, 1, 0, 0)$ to form the valid subsets $((1, 1, 1, 1, 0), (1, 1, 1, 0, 1), (1, 1, 1, 1, 1))$ as shown in figures 3.10 and 3.11 again in color.

In total we now have the following valid subsets:

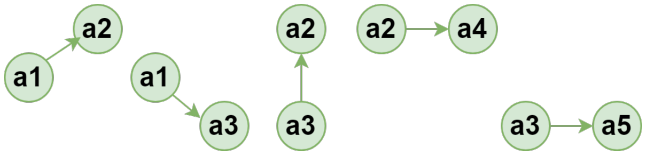


Figure 3.7: **All valid sub plans containing one edge. The colored plans are a part of the vertex extension set**

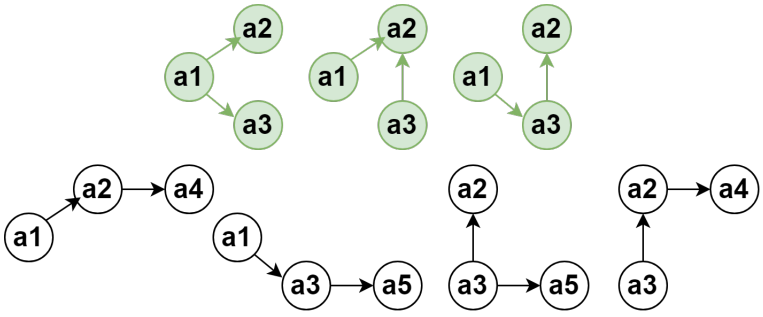


Figure 3.8: **All valid sub plans containing two edges. The colored plans are a part of the vertex extension set**

These subsets are then used in our dynamic programming algorithm to create our plan space. There are two distinct steps; first we create all one edge plans, and afterward, we enumerate our final plan space. Figure 3.7 shows all the one edge subsets. For each of these, we create both binary and WCO plans according to lines 3-22 in Algorithm 7. Note that the two subsets containing the edges $a2 \rightarrow a4$ or $a3 \rightarrow a5$ only create one WCO plan since one of the vertices in the subset is a lonely variable.

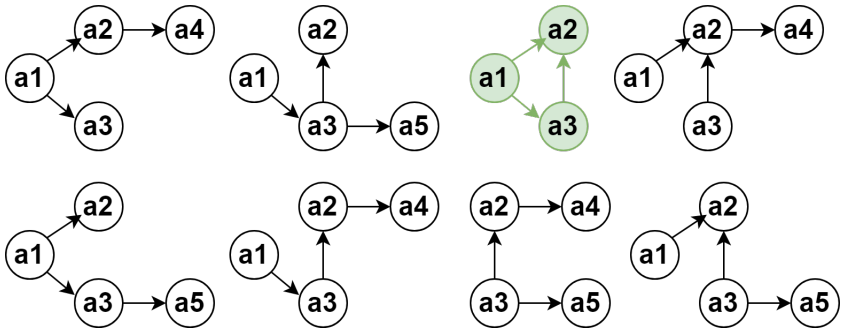


Figure 3.9: **All valid sub plans containing three edges. The colored plans are a part of the vertex extension set**

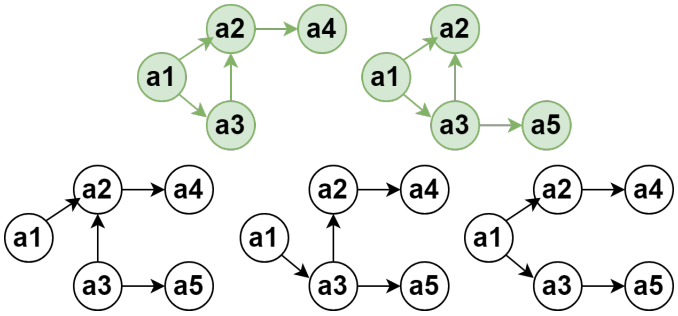


Figure 3.10: **All valid sub plans containing four edges. The colored plans are a part of the vertex extension set**

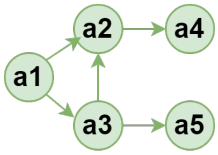


Figure 3.11: **The full SGM query**

Lines 23-41 in Algorithm 7 has three separate steps that are performed for each subset. We iterate over all our generated subsets from the bottom up, starting at all subsets containing two edges. When creating WCO plans, we only want to create plans for the valid WCO subsets. Let us take the subset in figure 3.12 as an example.

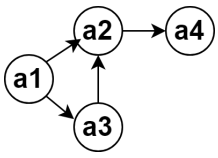


Figure 3.12: **A valid WCO sub set**

This subset consists of the vertices $(a1, a2, a3, a4)$. For each of these vertices, we remove the vertex and all its incident edges and see if rules 1 and 3 described previously still hold. In short, rule 1 dictates that the resulting subset needs to be connected, and rule 3 dictates that we can only add the vertices using all incident edges. Figure 3.13 shows the subsets that result from removing these vertices. The second subset from the left violates rule 1 as it is not connected. None of the other subsets violate rule 3. Therefore, we continue with these subsets.

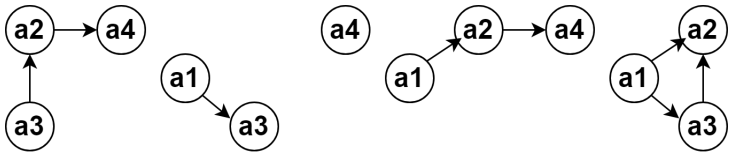


Figure 3.13: **All subsets created by removing a single vertex from figure 3.12**

Now let us continue using the left subset as an example. In this subset, the vertex $a1$ was removed and thus is the vertex we wish to add using a WCO join. Since we are currently creating plans for a subset containing four edges, we have already created all plans for one, two, and three edge subsets. We now retrieve all the plans we have created so far for the subset in this example. For every one of those plans, we create a new plan. This new plan has at the top a WCO join that extends by vertex $a1$ and as its child the plan for the subset. Note that in general the plans for the subset can be both binary, WCO, or hybrid plans.

When creating binary plans, we iterate over all generated subsets. For each of these subsets, we compute all possible ways to split a subset into two separate non-empty connected subsets. Doing this with Figure 3.12 produces, in part, the following sets as depicted in Figure 3.14. The left example extends a cycle by an edge. Here the cycle can be computed by a binary, WCO, or hybrid plan in this case. The right example, however, joins two subsets, both containing more than one edge, which will generate a bushy plan. More importantly, there are two join variables, vertex $a2$ and $a3$. Therefore we have to go over three separate loops, lines 33-35 in Algorithm 7. Two of these loops go over all plans generated so far for the subsets $s1$ and $s2$, thus creating each possible pairing. However, since there are two join vertices, we have to create two new plans for each pairing, one that first joins $a2$ and then $a3$ and vice versa.

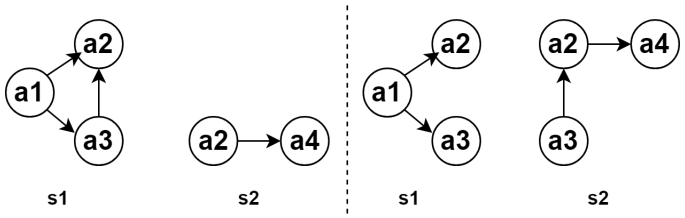


Figure 3.14: **Two possible ways to split the subset into connected subsets from figure 3.12**

The final step in Algorithm 7 does not apply here for this example. Line 37 computes all subsets $s1$ and $s2$ that are both cycles and share a common edge. This is not possible here, Figure 3.5 shows a query where this step would be applicable.

After doing the above steps for each subset up to and including the full query we will have generated our full plan space.

3.1.3 Plan Space Compared To Other Approaches

This section focuses on the differences between the plan spaces generated by our approach compared to that of Graphflow and EmptyHeaded.

- EmptyHeaded does not consider all possible QVO's for their query plans. We include all possible QVO's that push lonely variables to the end of the QVO.
- The QVO's used by Graphflow are a superset of ours since they do not consider lonely variables. Furthermore, they employ adaptive QVO's that can cause the initial plan to be

changed on a per tuple basis. This means that potentially many different QVO's can be employed to solve a single query. While this does not create different plans, it allows for executions of the plan that we do not have in our plan space.

- Graphflow does not consider BJ plans that first open a triangle and then close it. We do consider these plans for completion, as this would eliminate all binary plans on 3-cycle queries. However, more importantly, it forbids the following: any hybrid plan that first uses WCO join and then binary joins on cyclic queries, which we think are plans worth exploring.
- Emptyheaded does not allow for plans where a WCO join follows a binary join in a hybrid plan. Their query plans always compute subsets of the query using WCO joins and then combine these into the full query using binary joins.
- We do not consider plans similar to the own we show in Figure 2.9b. Specifically, we do not consider binary plans that enumerate over an edge table matching each edge with the intermediate results of both children. These types of plans are in both Graphflow and EmptyHeaded's plan spaces.
- Graphflow starts each WCO plan with an edge instead of a single vertex. This means that a join algorithm does not bind those vertices. Instead, their edge is enumerated. Notably, their WCO plan for a 3-cycle is a plan that extends an edge by a single vertex. We consider that plan to be a hybrid plan. Our WCO plan binds all three vertices in turn.

3.2 Implementation of Algorithms in Avantgraph

The database engine relies on various operators to do the work. A composition of these various operators resolves each query. The primary focus of this thesis is on join operators, and the various ways we can combine them, and to a lesser extend on data access operators needed for the join operators. These operators are implemented according to Volcano-style execution plans [6]. This means that queries are executing in a pipelined fashion where each operator has the functions *open*(), *next*() and *close*(). The *open*() and *close*() functions simply prepare and clean up the operator. The *next*() function is the workhorse that, when called, returns the next output tuple of that operator. To execute the query, we open the root operator and then repeatedly call *next*() to retrieve each output tuple until all results are retrieved. The power of this approach is the low memory consumption due to the pipelining.

In this section, we will go over the leapfrog triejoin algorithm and how we have implemented it as an operator. We will also quickly cover how we decide what binary join operator to use in our plans.

3.2.1 Leapfrog Triejoin in Avantgraph

Leapfrog Triejoin Adapted for Graphs

Tries are the base data structure of the leapfrog triejoin; each relation in the dataset is represented by one. When executing a query, these tries need to be sorted according to the QVO chosen for that query. In a relational database, each relation can have a large number of attributes. The potential number of different depths of each trie can, therefore, be high. Pre sorting each trie according to all possible variable orderings is infeasible. We are left with two options. Either we sort at runtime, which costs valuable time, or we sort certain orderings and restrict our plan space to that. Graph databases do not have this issue. The relations in a graph are the edges. Each edge has only two vertices. Therefore for each relation, we only have to sort and store two orderings, which can be done when loading in the data and does not have to be done at query runtime. Only ever having tries with a depth of two also simplifies the implementation logic. Avantgraph works using a triple store, and we retrieve our data by labels. Every edge table is stored twice, once for every ordering of the relation. We denote these orderings by *(predicate, source, object)* PSO and

(predicate, object, source) POS; where predicates are the labels that go from the source to the object in our directed, labeled graph. We stored edge tables are pre-sorted in the following way: PSO is sorted on the source values, and POS is sorted on the object values. We need to store the data in a way such that we can retrieve the data under the runtime guarantees needed for the iterators.

It is also more natural to think and reason about binding propagation in graphs then it is in relational databases since SGM queries are graphs themselves. In a relational database, we would be binding one attribute at a time, which we cannot easily visualize as a subgraph. The hypergraphs for relational queries, where relations can have more than two vertices, are more complicated, which makes the notions of cyclic and acyclic queries less straightforward. Meanwhile, a SGM query for a graph database is the same as its hypergraph.

Implementation of Leapfrog Triejoin in Avantgraph

We implemented the leapfrog triejoin algorithm using two layers of operators; the iterator interface that walks the base relations of the graph, and the leapfrog join, which joins the iterators together. In our implementation, we have removed the need for the third layer, which is the triejoin. This layer controls the other two layers to bind the correct variables. We still require this control logic. We have, however, moved this logic to be self-contained in the leapfrog join. Thus, the leapfrog join maintains both the variable ordering in the QVO and the depth of the iterators. In a pure leapfrog triejoin implementation, this is not required, nor advised since it complicates things. Nevertheless, for the algorithm to work properly in hybrid plans with other algorithms, it needs to be self-contained with well-defined interfaces and behavior. Using the triejoin to direct all leapfrog join operators would not allow us to use binary joins in place of leapfrog joins to create hybrid plans.

Example 3.2.1. As an example query with edge labels a, b, and c, we will use the following cyclic query of length three. We visualize this query in Figure 3.15:

$$Q = a \bowtie b \bowtie c \tag{3.1}$$

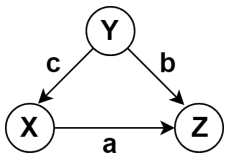


Figure 3.15: **Query subgraph of a 3 cycle with edge labels a, b and c**

To execute this query, we have to decide on a QVO over the vertices, or variables, X, Y, and Z. What variable ordering we choose has a significant impact on the run time of the query. For each vertex in the query, we create a leapfrog join instance, which has access to the iterators for the appropriate relations. For each relation in the query only a single iterator is needed. In this case, the leapfrog join for X requires the iterators for a and c, Y requires b and c, and finally, Z requires a and b. If we were to pick a QVO of (X, Y, Z), then we would first bind vertex X, then Y, and then Z. This ordering dictates how we need our data sorted, the leapfrog join for X wants to intersect the sources of a with the objects of c. Therefore, the data for iterator a is retrieved from the PSO storage, and the data for iterator c is retrieved from the POS storage.

Figure 3.16 shows the full execution plan for this query. The nodes X, Y,, and Z represent the leapfrog join for their respective variables. The leaves represent the trie iterators. The first letter

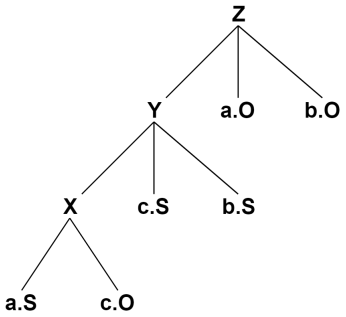


Figure 3.16: **Execution plan for query 3.1 using QVO (X,Y,Z)**

indicates the relation of the iterator, and the second letter indicates whether we are joining the source(S) or object(O) of the relation. Note, that for instance, both a.S and a.O represent the same trie iterator but at a different depth. The order in which S or O appear in the execution plan tell us which order of the trie iterator we work with, either PSO or POS. In the case of relation a, we first encounter a.S, so the trie iterator retrieves the data for relation a from the PSO data storage.

Each one of the leapfrog joins and trie iterators has the usual *open*(), *next*() and *close*() functions. We retrieve bindings from leapfrog joins by calling their *next*() function. Leapfrog joins can have other leapfrog joins as their children. This is part of the self-contained logic. Instead of an overarching control unit calling *next*() to retrieve bindings from the appropriate leapfrog joins according to the QVO, the leapfrog joins themselves call next on their child leapfrog join. In this case, we have a QVO of (X, Y, Z). Calling *next*() on the leapfrog join for variable Z causes a call to *next*() on variable Y and then on X. This self-contained logic is required in the case of hybrid plans. In hybrid plans, a binding might have to be retrieved from a different join algorithm. By requiring all join algorithms to adhere to the same interface makes it possible for a leapfrog join to call next on any other algorithm in the same way as it calls next on another leapfrog join.

We execute this plan by first calling *open*() and then repeatedly *next*() on the operator root, which in this case is the leapfrog join for variable Z. Once *next*() returns false, which means there are no more bindings we call *close*() to clean up the execution plan.

Leapfrog Join

The leapfrog join is implemented as an operator according to the *open*(), *next*() and *close*() principles of Volcano style plan execution. *open*() prepares the data for the algorithm and *close*() cleans up the operator. $Next()$ is the workhorse that, when called, computes the next binding. In addition it also has the *search*() function similar to the one in Algorithm 4. The leapfrog join requires an array of pointers to the trie iterators it needs to join, and a pointer to the child join if it exists. The array of trie iterators is sorted at all times according to the values that they point to. The *open*() function differs from the $leapfrog - init()$ function described in Algorithm 3 because we require logic capable of handling child joins. It initializes the leapfrog join by calling *open*() on its child join, and if there is no child join calls *open*() on its iterators. We also do not yet call *search*(), we only look for bindings by calling *next*(). With the leapfrog join handling the overall logic of the algorithm, it has to call on the join below it in the execution plan itself, but the bottom join in the plan has no child join. Therefore, the leapfrog join has different control flows depending on if there is a child join or not.

The case without a child join is the simplest; all we do is join the trie iterators. Calling *next*() in this case follows Algorithm 5 closely. We check if the current iterator, the one located at the smallest value, is at the end. If that is the case, then there are no bindings to be found anymore,

else we call the *search*() function. This function differs at a small point from Algorithm 4. Instead of using the *atEnd* variable to indicate to the triejoin that the leapfrog join is done, the leapfrog join has to handle this itself. When the leapfrog join cannot find a binding, we reduce the depth on all iterators in the join, and return to the parent join in the execution plan that we could not find a binding. In this case where it concerns the bottom join in the execution plan it would mean that no more results will be produced.

In the case with a child join, we do not only have to join the iterators to produce valid bindings. We also have to handle the logic of directing the child join. Binding propagation produces results by binding vertices one by one in order according to the chosen QVO. This means that for every binding we are trying to compute, we can filter the data based on the previous bindings computed. Therefore if a leapfrog join that has a child join cannot find a binding, it is not done yet. At that point, it has to direct its child join to compute another binding, so that there is new data to be joined. Because of this reliance on previous bindings, when *open*() is called, we first have to *open*() the child join before any work in this leapfrog join can be done. When *next*() is called for the first time, we first have to call *next*() on our child join so that we have the bindings required. Now that the bindings are in place we can call *open*() on our trie iterators, we then attempt to join the values returned by these iterators. When no more bindings can be computed, we do not yet return to the parent join that no binding can be found. Instead, we again call *next*() on our child join to produce a new binding. With this new binding, we once again open our iterators to try to find bindings again. Only when our child join cannot compute bindings anymore, do we return there are no more bindings. This is why the bottom-most join being done indicates that no more results will be produced.

Trie Iterator

The iterator interface described in Tables 2.1 and 2.2 is extended with the *close*() function. The *open*() function positions the iterator on the relevant data. Depending on the execution plan, we retrieve the SPO or SOP data. The *close*() function is there to close the iterator when it is done. As described earlier, the iterator interface needs certain runtime guarantees. To reach these guarantees, the graph is stored using a tree data structure. When an iterator is instantiated, it simply walks the appropriate branch of the tree. Therefore, no data copying is necessary, and the leapfrog join can access the data without requiring more memory than needed to store the data. Because we are working with graph data, the tries we construct will always have a depth of two, one depth for the source and one for the object of the relation. The iterator itself keeps track of what depth it is on. When initialized, the depth starts at -1. The leapfrog join manipulates the depth of the iterator. Whenever *open*() is called, the depth is increased by one, and whenever *up*() is called the depth is decreased by one. This process is illustrated in Figure 3.17

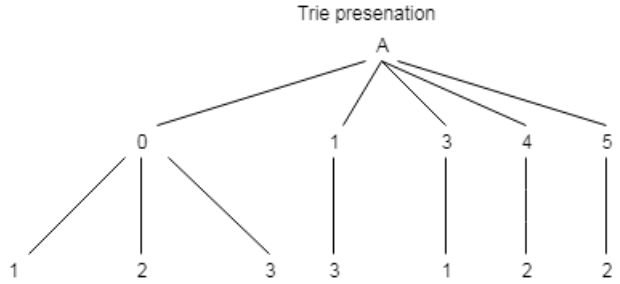


Figure 3.17: **Trie of arity two as would be used in a graph database. Initially, the depth is at the root. The first time** *open*() **is called, the iterator is positioned at the first key, 0, on depth 0. Calling** *open*() **again positions the iterator on depth 1, at key 1. Now if** *up*() **and then** *next*() **is called the iterator on depth 0 is positioned at key 1**

For every edge in the query, we require a single iterator. If multiple edges are of the same label, then each edge needs a separate iterator. So far, we have introduced only a single type of trie iterator that always has a depth of two and represents a single edge in the query. This works for pure leapfrog triejoin plans. However, when we introduce other join algorithms, this fails on one point. Now we have a situation where for an edge, one vertex is bound by a leapfrog join and the other by another join algorithm. When this happens, there are two cases:

- The leapfrog join is first in the execution plan: nothing changes, we can still use the standard trie iterator. However, we now only use depth 0. Since depth 1 would have been used for joining the variable now being bound by another join algorithm.
- The leapfrog join comes after the other join algorithm: normally, the trie iterator would have been opened by the first leapfrog join and then opened again by the second leapfrog join. This would position the iterator at depth 1 with a binding on depth 0. Now instead, we have to make a trie iterator interface that can retrieve the binding created by the child non-leapfrog join and retrieve the appropriate data that way.

3.2.2 Binary Join Operators

We utilize different binary join algorithms, depending on the situation. We can have two different types of inputs to our binary join operators. Either we are joining a base label, or we are joining an intermediate result. Base labels have the advantage that they have a known size at query runtime and that they can be pre-sorted. Intermediate results can be hugely variable in output size and also has incurred costs if we need to sort or hash the data. For these reasons, we use the following structure to decide what algorithm to use:

1. Sort-merge join: if both sides require base table scans. The data is already sorted, which allows the sort-merge join to execute quickly.
2. Indexed nested loop join: if only one side is a base table. Here we again take advantage of one side being pre-sorted
3. Hash join if both sides are intermediate results. In this case, we would have other options. A naive approach would be a nested loop join, which would be wildly inefficient. For other options, we first have to do some work on the data. In the case of a hash join, we have to build a hash table on one side of the join. In the case of a sort-merge join, we would have to sort both sides of the join. Building a hash table and sorting the data can be considered equally expensive computationally. However, a hash join only requires one hash table to be made. In the case where the intermediate results have a significant difference in size, we can build this hash table on the smaller result and have less work that way. Having the hash table on the smaller relation also speeds up the hash join algorithm. However, we do not know in advance the sizes of the intermediate relations and therefore, cannot guarantee that we build the hash table on the smaller relation.

Chapter 4

Experimental Study

4.1 Experimental Setup

To verify the strengths and weaknesses of our approach, we performed numerous experiments on real data. When comparing different algorithms to each other, especially ones as different as edge-by-edge and vertex-by-vertex approaches as here, relying only on runtime as a metric is not feasible. Implementation and hardware-specific details can affect the results of our experiments enough to matter in instances where running times are close together. Therefore for our metrics, we look at both runtime, intermediate result sizes, and combined tuple output for our plans. The runtime is described as the total time it takes to execute the query plan. Tuple output is defined as the number of tuples an operator produces, the combined tuple output for the plan is the output of all operators combined.

We look at both intermediate result sizes and tuple output. While they are closely connected, there is a crucial distinction. Larger intermediate result sizes generally implies more tuples produces and, therefore, a higher running time of the query. However, query plans, especially for the larger SGM queries, have multiple layers. Intermediate result sizes only describes how a single layer will affect its parents. It has no information on how costly producing the intermediate results was. Two join plans can arrive at the same number of intermediate results at different actual costs. A layer here is a sub result, for example, computing the 3-chain before closing it into a 3-cycle in a binary plan. For a simple query like a 3-cycle, computing the 3-chain dominates the overall running time of the query, having a significant intermediate result here impacts the query substantially. However, what if we look at larger SGM queries, a significant intermediate result on the first layer does not immediately imply that this plan should be slower than a plan on the same query that produces a low number of intermediate results on the first layer. The total tuple output of the latter plan could be higher than the former plan. Thus when trying to compare queries through intermediate results alone, we would have to look at all layers simultaneously, which is essentially the total number of tuples computed by the query plan.

We define a tuple produced as a single call to our operators, so each tuple produced by a call to $next()$. If an operator has to do much work to produce the next tuple, it will still count as one tuple for that operator. However, that work is carried out by the children of that operator in the plan. Note that this metric is similar to the i-cost introduced by Mhedhbi and Salihoglu [8], also described in chapter 2. Total tuple output also allows us to contrast between the different join algorithms and look at how long each algorithm takes to produce a tuple on different workloads. By looking at how long each algorithm takes to produce a tuple, we can define the tuple throughput of those algorithms. Using the tuple throughput, we can then compare different join algorithms.

Our experiments aim to show us insights into the following questions. These questions are an

extension of our research questions:

1. Which plans generally perform better on certain types of queries such as cycles, chains, stars? We are not only interested in what plan types, these being binary, WCO, and hybrid plans, being the best. We are interested in what type of plans within these plan types are the best. For example, for hybrid plans in what order do we see WCO and binary join algorithm appear.
2. How can we compare the tuple throughput of binary and WCO algorithms? Can we compare the throughput between query types and different data instances? Or only on one specific instance, if at all. Finding a ratio of the tuple throughput between binary and WCO algorithms could provide us insights to build query planners for hybrid plans.
3. Does including hybrid plans in our plan space give us significant enough improvements? Including hybrid plans, increases the size of our plan space significantly. If these plans cannot run faster than pure plans, the extra computation required might not be worth it.
4. Under which circumstances, if ever, do traditional binary joins outperform WCO plans on SGM queries?
5. How much benefit can we see from including shared edge plans? These plans increase the size of the plan space, but do they outperform the same plan that does not include the shared edge?

Before diving into the experiments, we hypothesize the results we might see:

Q1: Which plans generally perform better on certain types of queries such as cycles, chains, stars?

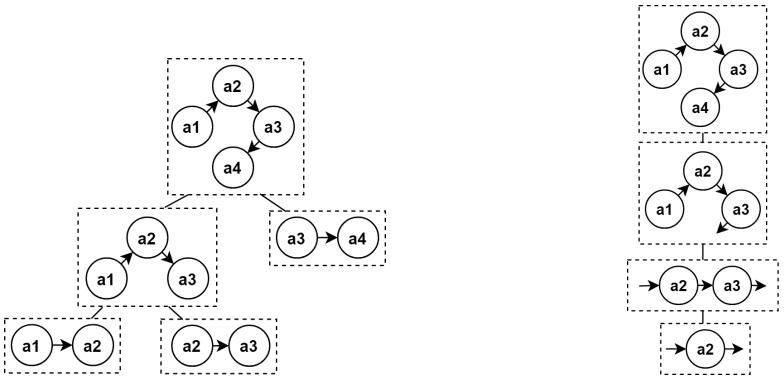
We will first discuss the likely differences between binary and WCO plans. Since hybrid plans are a combination of both, we will discuss them afterward. We will limit the discussion here to the general classes of cyclic, chain, and star queries. Many query types, such as bowtie and barbell queries, can be described as combinations of these general classes.

WCO plans are fast because they pre-filter to produce smaller intermediary results, which is especially apparent in cyclic queries where binary plans are not worst-case optimal. However, binary joins have less overhead, and therefore, a higher tuple throughput. On specific instances where intermediary results are not reduced much by a WCO plan, a binary plan could potentially be quicker. Leapfrog join also performs multiway joins which should show performance gains on SGM queries that contain vertices of arity > 2.

Binary join algorithms are not asymptotically sub-optimal on chain queries. Therefore the only real benefit the leapfrog triejoin might have here is its ability to pre-filter to produce smaller intermediate results. The smaller the intermediate results, the lower the amount of computation required by further joins in the query plan. Figure 4.1 shows a binary and WCO plan for a chain query of length four. The binary plan only has to compute two joins while the WCO plan has to compute four. However, the WCO plan looks ahead, meaning it joins edges that are not relevant for the current pattern.

In Figure 4.1b on the second layer, we compute the bindings for vertices $(a2, a3)$. Computing these two vertices already computes the entire query. All edges from the final query are already present in the joins. So we are not computing all possible instances of the vertices $(a2, a3)$, we are just computing the ones present in the final result. The only step after that is to enumerate the values of vertices $a1$ and $a4$ based on the bindings for vertices $(a2, a3)$. Vertices $a1$ and $a4$ are the lonely variables, as presented in the last chapter. The WCO plan does this enumeration at the end to delay the computation and reduce intermediate result sizes. However, with the cost

of extra overhead in the more costly join computations. The binary plan is most likely going to perform better when the intermediate result sizes stay consistent throughout the plan. While the WCO plan should outperform the binary plan if it can comparatively cut a lot of the intermediate results.



(a) **Binary plan** (b) **WCO plan**

Figure 4.1: **Possible plans for a 4-chain query**

The idea behind hybrid plans is to combine the best of both worlds. If binary plans perform well when WCO plans cannot pre-filter much compared to binary plans, then we should see left-deep hybrid plans doing well on queries such as bowties. Binding the middle vertex in a bowtie query significantly filters the number of intermediary tuples. After binding the middle vertex and several other vertices, the intermediary results can be lowered enough such that a binary algorithm can compute the last couple of edges. This notion can be generalized to any query type. Whenever the intermediary results are small enough and stay small, we will most likely see hybrid plans with binary algorithms extending the final edges being faster than pure WCO plans.

Another aspect of hybrid plans is the ability to combine entire sub-patterns. We define these plans as bushy hybrid plans. An example would be combining both cycles in a bowtie pattern. Both cycles would be computed separately, most likely through a WCO plan, and then combined using a binary algorithm, also discussed previously in Chapter 3. This approach is probably not optimal for some data instances. For instance, if the left cycle has a low amount of results, the right cycle a high amount of results, and the resulting bowtie query has a low amount of results, then a WCO plan should be more appropriate. Consider a WCO plan that computes the left cycle first and then binds the other remaining values. Such a plan would have to compute a low number of left cycles, which are then completed into the full query without ever having to compute the large number of right cycles. It will be interesting to see what the experiments will show here and for other queries where combining bushy hybrid plans are a possibility.

The paper by Mhedhbi and Salihoglu [8] also researches hybrid plans notes several plans as being the fastest in their experiments that contradict our hypothesis here. Notably, their fastest hybrid plans for a 6-cycle query first computes a 4-chain using binary joins and extends this sub-pattern with a WCO algorithm into the 6-cycle. This goes against our hypothesis of expecting WCO algorithms to precede binary algorithms. Nevertheless, such a plan can make sense since it closes the cycle using a WCO algorithm as opposed to a binary algorithm.

Q2: How can we compare the tuple throughput of binary and WCO algorithms?

As already noted in the discussion above, we hypothesize that everything revolves around intermediate result sizes, and thus total tuple output required. Any plan that can reduce the intermediate result sizes by a substantial amount is likely to be faster. The question then is what is substantial,

the different algorithms we use each have different levels of overhead. When the intermediate result sizes are close between different plans, this overhead can cause the plan with higher intermediate results to still perform faster. Being able to quantify this overhead in comparison to intermediate result sizes will allow us to define the tuple throughput of different algorithms. This will be needed to create a cost-based plan optimizer that can reliably pick the best plan, whether it be hybrid, WCO, or binary.

A vital metric will be the degree that the WCO joins can pre-filter at each step of the plan compared to a binary join plan. The goal is to estimate the intermediary result sizes produced by binary and WCO algorithms and then be able to use their tuple throughput to decide what algorithm would be better. Experimentally a ratio between the tuple throughput of binary and WCO algorithm can be established. Similar to this are the weights used by [8] to compare the costs of intersections to a hash join.

Q3: Does including hybrid plans in our plan space give us significant enough improvements?

Whether or not hybrid plans are worth the effort is likely to be query specific. Without having prior information on the intermediate result sizes of a query on a given database instance, it will be hard to rule out entire query types as being bad for hybrid plans. For instance, for a query such as a 3-cycle, a WCO plan should perform better than a hybrid plan on an instance with low selectivity. For a 3-cycle the only hybrid plans that we generate enumerate over one edge of the query and then extend that by one vertex through an intersection. If the query has a high selectivity, we will enumerate over possible edges that a WCO plan would filter out, which would not be the case for a low selective query.

It will be interesting to see from the experiments where hybrid plans do well and where they do not. Being able to pinpoint on which classes of queries and on which data instances hybrid plans do well and on which they do not would allow us to compute them only when relevant.

Q4: Under which circumstances, if ever, do traditional binary joins outperform WCO plans on SGM queries?

We have listed many advantages that WCO join algorithms have over binary join algorithms, the question is then "are binary join algorithms still required?". We think the strength of them will be apparent in hybrid plans, but outside of some edge cases, on the average pure WCO join plans should outperform binary join plans consistently on all query types. The main place where we will most likely see the benefit of binary join algorithms are in hybrid plans where the binary algorithm can be used just for the section of the query where circumstances allow it to outperform WCO algorithms. Then it would not be required for the binary algorithm to be better across the entire plan but just for a subsection of it. Most likely this will be seen near the end of plans with low selectivity where the intermediate results sizes have shrunk considerably. Whether we will see bushy hybrid plans perform well is left to be seen.

Q5: How much benefit do we get from including shared edge plans?

Shared edge plans filter aggressively by including the same edge multiple times in a plan. The same edge appears twice because we are joining two sub-patterns with a binary join that both include that edge. As discussed in Section 3.1, the plans that are shown in Figure 3.5 are an example of this. The plan with the duplicate edge should be quicker, which has also been noted by [8]. However, we do not think that these plans will surpass other hybrid plans, but we still include them for a fair comparison.

Data

For our data set, we use YAGO2s. This data set contains over 240 million triples derived from Wikipedia, Wordnet, and GeoNames [2]. This data set allows us to test our approach on realistic data that takes many different forms. While at the same time testing how it performs on such a large data set. To be able to experiment using this data set, many queries were mined for a set of SGM queries.

Hardware

We use a single machine with an Intel Xeon X5675 @ 3.07GHz CPU and 188GiB of RAM. Our experiments only ever utilize one physical core at a time.

4.2 Queries

For our experiments, we use a variety of queries, both cyclic and acyclic, with various connectivity. Figure 4.2 shows all the SGM queries used in our experimental evaluation. Our base queries are named by following the number of edges by the type of query, i.e., 3-cycle means a cyclic query with three edges. These queries were mined from the YAGO2s dataset by a query miner written in Avantgraph. In our experiments, we only consider labeled data. We record the tuple throughput of all operators in each plan and the running times of our plans. For the join operators, the tuple throughput is equal to the intermediate result size of that sub-pattern. This way, we can verify how well our approach does on different data sets. Our main goal is to evaluate our approach between queries and on different data for the same query.

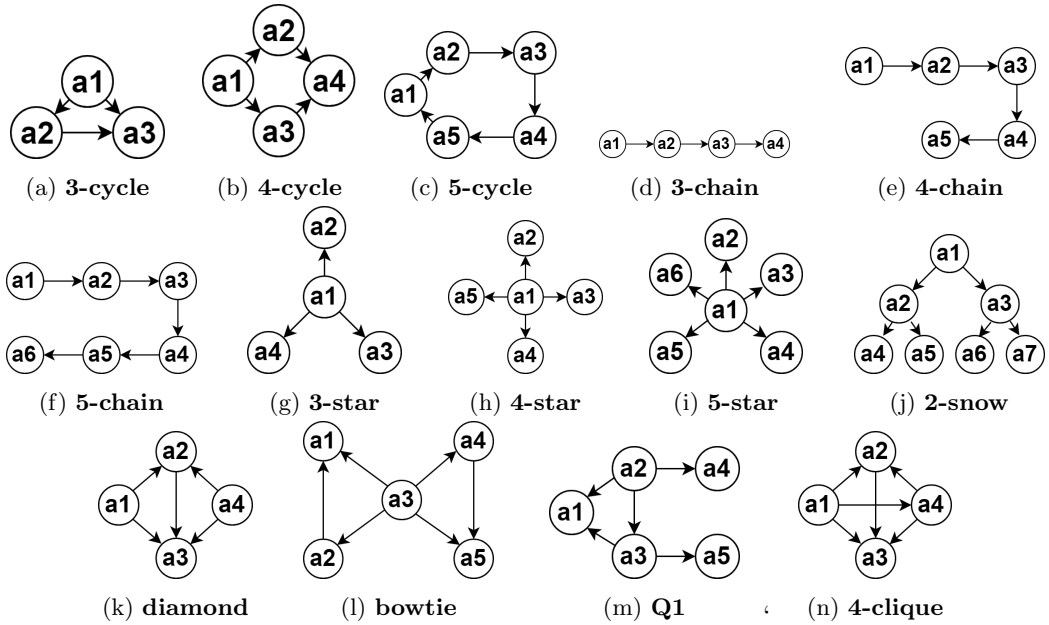


Figure 4.2: **SGM queries used in experimental evaluation**

For each query type, we run between 25 to 100 different queries of that type on the YAGO2s dataset [2]. These queries were selected at random from all mined queries. For all queries, we run all the plans in our plan space. We then analyze the results of the queries to answer our research questions in the following ways.

- We present a general overview containing relative statistics on the running times of the queries. These running times were computed by taking the average of ten runs of a query

plan. Since we are merely interested in the fastest plans of each type for a query, we used a dynamic timeout. The timeout interrupts the execution of a query plan when it takes twice as long as the fastest plan of that type so far.

- To analyze specific queries, we rerun a smaller subset of queries where we execute each plan up to a global timeout. Using these results, we can now look into the entire plan space and how different types of plans execute. Here we will look at all metrics, run times, total tuple output, and intermediate results sizes. We can compare these metrics between different plans. For instance, the plans in Figure 3.5 will be compared this way.

We will now first introduce our query miner written in Avantgraph.

4.2.1 Query Miner

To be able to run the queries that we want, we created a query miner in Avantgraph. This section will briefly elaborate on how we created this query miner and how it functions. We wrote the query miner into Avantgraph in a straight forward manner. It executes the following steps to mine all the queries that we require:

- Load in the YAGO2s data set.
- Retrieve all labels with a cardinality over a certain threshold. This threshold is set such that we approximately get the number of labels that we want to limit run time.
- Using these labels, we enumerate over all possible 3-cycle, 3-chains, and 3-star queries. This creates L^3 number of possible queries, where L is the number of labels. Using these base queries, we can then step-wise create more complex queries without needing to enumerate all possible combinations.
- For each query, we create one WCO plan with an unaltered QVO. We then run all these plans and record whether there is any output. To reduce the run time, we interrupt any query execution after a couple of seconds. This means that we cannot guarantee the correct number of results. However, we are only interested in whether there is an output or not. It is also a possibility that a query has results but that it took longer than the timeout to compute the first output tuple. In that case, we lose a result, but that query would have taken to long for the experiments anyhow.
- For each query with output, we write to a text file the labels and the number of results.
- To create more complex queries, we build upon the results of the previous queries. For instances to create a 4-chain query, we take all the 3-chain queries generated before and extend them with the available labels. This results in $|3 - chain| * L$ number of queries, where $|3 - chain|$ is the number of 3-chain results previously mined.
- We keep building to create more complex queries while keeping the number of queries we have to execute to manageable levels.

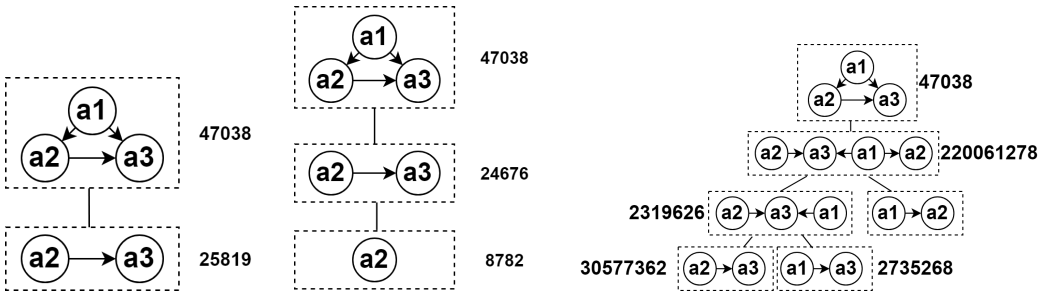
4.3 Plan Space Evaluation

Tables 4.1 and 4.2 show the general results for all our experiments. The first row of each query shows the average runtime over all the queries for that plan type, only using the fastest plan for each type per query. The second and third rows of the table show the largest and smallest ratio. We define this ratio as the relative runtime of that plan type compared to the binary plan, again using the fastest plan for each type. Thus the largest ratio indicates the query where the plan outperformed the binary plan the greatest and vice versa for the smallest ratio. For instance, the largest ratio for the hybrid plan for the 3-cycle is 0.0055. So for a certain query, the hybrid

plan executed 200x faster than the binary plan, and this was the largest ratio over all the 3-cycle queries. The queries where these ratios are at extremes are good candidates to look into for further insights. Since these queries present us with opposites and can more easily show us what causes the differences between the plan types. Note that a gap of 1.429, as seen for the hybrid plan for the 3-cycle query, means that the binary plan was faster than the hybrid plan here. The last column records for all queries the plan type of the fastest plan. The table then shows the number of times a certain plan type was that fastest plan in relative numbers. The 1 in the WCO plan column means that for every query in our experiment for that type the fastest plan was a WCO plan.

By comparing hybrid, WCO, and binary plans for the cases where the runtime ratios between them are at the extremes, we can see what we hypothesized. Binary plans should perform worse because of exploding intermediate results. However, they should perform decently well when this is not the case. Let us look at the 3-cycle query first, Figures 4.3 and 4.4 show these extremes, on the latter query the hybrid plan was actually 1.4x slower than the binary plan. The binary plan first has to compute a 2-chain by joining two base labels together. This violates worst-case optimality since it has a potential output size of N^2. The bold number next to the layers in the figure indicate the tuple output of operators. We can see in Figure 4.3c that the exploding intermediate result sizes cause the large runtime gap.

More interesting is comparing the plans in Figures 4.4a and 4.4c. In this case, the hybrid plan has to iterate over 277069 edge labels. And for each of these, perform an intersection on the edge tables of edges ($a1 \to a2, a1 \to a3$) given the bindings for $a2$ and $a3$. Yet, for all these intersections there are only 360 output tuples. Meanwhile, the binary plan keeps its intermediate results low. In this instance the WCO plan also ran 44x faster than the hybrid plan. This is notable because the plan we call a hybrid plan here is, in fact, the only WCO plan in the plan space of [8] for the 3-cycle query.



(a) **Hybrid plan** (b) **WCO plan** (c) **Binary plan**

Figure 4.3: **This figure shows the fastest plans of each plan type for the 3-cycle query with the largest runtime ratio between the binary and other plans. The bold numbers are the number of tuples produced by each step.**

Cyclic Queries

The data illustrates that WCO plans outperform binary plans by a wide margin on cyclic queries. The 3-cycle query is often used as the base example for the strength of vertex by vertex joins because binary joins violate worst-case optimality here. The 5x speedup between hybrid and WCO plans on the 3-cycle query is because the only hybrid plan in our plan space takes one edge from the query, goes over all the edges of that label, and tries extending each with a vertex by intersecting the other two edges. Which can cause much unnecessary computation time when the

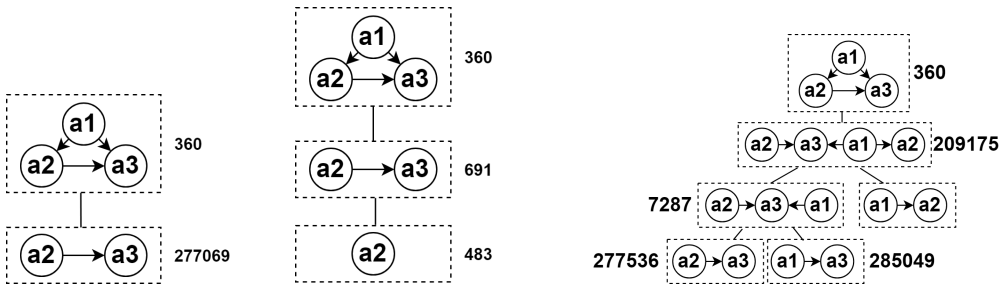
	Hybrid Plans	**WCO Plans**	**Binary Plans**
3-cycle	**3**	**6**	**6**
Average Fastest Runtime	0.116	0.019	1
Largest Ratio	0.0055	0.000105	1
Smallest Ratio	1.429	0.226	1
Fastest Plan type	0	1	0
4-cycle	**300**	**16**	**20**
Average Fastest Runtime	0.0401	0.0370	1
Largest Ratio	0.00157	0.00169	1
Smallest Ratio	0.333	0.313	1
Fastest Plan type	0.55	0.45	0
5-cycle	**3040**	**40**	**70**
Average Fastest Runtime	0.0224	0.0214	1
Largest Ratio	0.000938	0.000894	1
Smallest Ratio	0.188	0.182	1
Fastest Plan type	0.4	0.6	0
3-chain	**16**	**6**	**2**
Average Fastest Runtime	0.0791	0.0937	1
Largest Ratio	0.00093	0.00118	1
Smallest Ratio	0.599	0.734	1
Fastest Plan type	0.908	0.092	0
4-chain	**52**	**8**	**5**
Average Fastest Runtime	0.0779	0.0952	1
Largest Ratio	0.00139	0.00276	1
Smallest Ratio	1.205	1.477	1
Fastest Plan type	0.955	0.034	0.011
5-chain	**302**	**16**	**14**
Average Fastest Runtime	0.0588	0.0651	1
Largest Ratio	0.00274	0.0280	1
Smallest Ratio	0.540	0.545	1
Fastest Plan type	0.957	0.043	0
3-star	**21**	**6**	**3**
Average Fastest Runtime	0.0980	0.1140	1
Largest Ratio	0.0058	0.0058	1
Smallest Ratio	0.611	0.783	1
Fastest Plan type	0.847	0.153	0
4-star	**276**	**24**	**15**
Average Fastest Runtime	0.0902	0.0972	1
Largest Ratio	0.0159	0.0164	1
Smallest Ratio	0.410	0.499	1
Fastest Plan type	0.9375	0.0625	0

Table 4.1: **This table shows a general overview of the queries depicted in Figure 4.2. Each column represent the values for a specific plan type. The bold values indicate the number of plans of the column type in the plan space. The first three rows of values for each query type are relative to the binary plan, a lower value is better. The average fastest runtime calculates the ratio of the fastest plan of the column compared to the fastest binary plan for each query and then averages those. The second and third rows show the largest and smallest of these values. The fastest plan type indicates the ratio for all queries executed when that plan type was the fastest.**

query has highly selective since a large number of the edges that we enumerate over will not be

	Hybrid Plans	WCO Plans	Binary Plans
5-star	**4425**	**120**	**105**
Average Fastest Runtime	0.253	0.277	1
Largest Ratio	0.000032	0.000032	1
Smallest Ratio	0.366	0.412	1
Fastest Plan type	1	0	0
2-snow	**3414**	**96**	**117**
Average Fastest Runtime	0.0489	0.0529	1
Largest Ratio	0.000303	0.000299	1
Smallest Ratio	0.9163	1.0644	1
Fastest Plan type	0.94	0.06	0
diamond	**306**	**20**	**370**
Average Fastest Runtime	0.0478	0.0397	1
Largest Ratio	0.00210	0.00295	1
Smallest Ratio	0.416	0.430	1
Fastest Plan type	0.55	0.45	0
bowtie	**1841**	**56**	**1504**
Average Fastest Runtime	0.0425	0.0426	1
Largest Ratio	0.0016	0.0014	1
Smallest Ratio	0.2406	0.2441	1
Fastest Plan type	0.5278	0.4722	0
Q1	**142**	**12**	**114**
Average Fastest Runtime	0.0156	0.0162	1
Largest Ratio	0.0018	0.0018	1
Smallest Ratio	0.0682	0.0783	1
Fastest Plan type	0.7872	0.2128	0

Table 4.2: **This table shows a general overview of the queries depicted in Figure 4.2. Each column represent the values for a specific plan type. The bold values indicate the number of plans of the column type in the plan space. The first three rows of values for each query type are relative to the binary plan, a lower value is better. The average fastest runtime calculates the ratio of the fastest plan of the column compared to the fastest binary plan for each query and then averages those. The second and third rows show the largest and smallest of these values. The fastest plan type indicates the ratio for all queries executed when that plan type was the fastest.**



(a) **Hybrid plan** (b) **WCO plan** (c) **Binary plan**

Figure 4.4: **This figure shows the fastest plans of each plan type for the 3-cycle query, where the hybrid plan is slower than the binary plan. The bold numbers are the number of tuples produced by each step. The WCO plan executed 44x faster than the hybrid plan**

part of any solution.

The 4 and 5-cycle queries however show hybrid and WCO plans being about equal with WCO plans being slightly ahead. The fast hybrid plans we see here are plans that first compute chains using WCO joins. Remember, we already join the two edges outside of the chain, and therefore the intermediate result size of the chain is lower compared to a chain computed by a binary plan. Using this chain, we see two kinds of plans, one that extends the chain using binary joins by joining in edges one by one. We also see plans that extend the path up to a chain subquery that only misses one vertex, at which point we extend the vertex using a WCO algorithm. So we see hybrid plans that go back and forth on what type of algorithm it uses being good. We have already hypothesized that hybrid plans that first use WCO algorithms and then switch to binary later in the plan could outperform full WCO plans when the WCO algorithm cannot filter much relative to what the binary algorithm would produce. However, we do not see hybrid plans that first compute a chain using binary joins, which Graphflow does.

Our experiments show a nice example of this. The query in question is a 5-cycle query where the hybrid plan is the fastest. This hybrid plan first binds three vertices using WCO joins. Which it then extends by an edge using a binary join algorithm and finalizes the query using a WCO join. This plan is shown in Figure 4.5. On this query the fastest WCO plan follows the same QVO but binds the fourth vertex using a WCO algorithm as opposed to a binary algorithm as with the hybrid plan. The important result from this experiment is that the intermediate result size of the fourth vertex binding is the same for both plans. This is interesting because the binding in the full WCO plan joins in an extra edge when binding the fourth vertex. However, in this case the extra step which can filter the result size aggressively ended up not filtering anything at all. Therefore extra work was done for no benefit, and thus the hybrid plan was the fastest. This is not unique to a single query; we see this back on numerous queries where hybrid plans are the fastest.

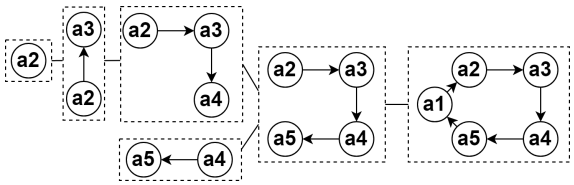


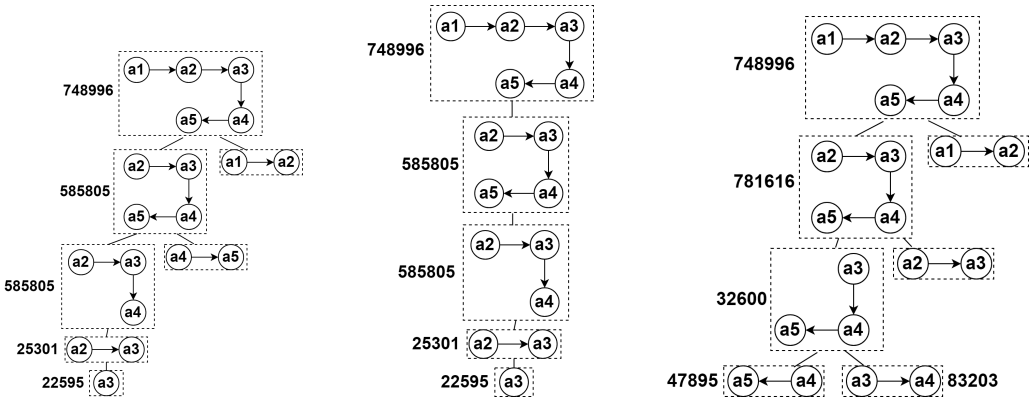
Figure 4.5: **Hybrid plan for the 5-cycle query**

Chain Queries

Chain queries are notable because it is the only query type in our experiments where a binary plan executed faster than WCO or hybrid plans on a particular query. However, this only happened on one single query, we might have expected more since chain queries are acyclic and binary join plans do not violate wors- case optimality here. Nevertheless, for all chain queries experimented with; 3-chain, 4-chain, and 5-chain queries, the binary plans were, on average, at least a magnitude slower than the other plan types. We can also see that on the larger queries, the gap between the binary and other plans become larger. Hybrid plans also beat WCO plans on average by about 10-20%.

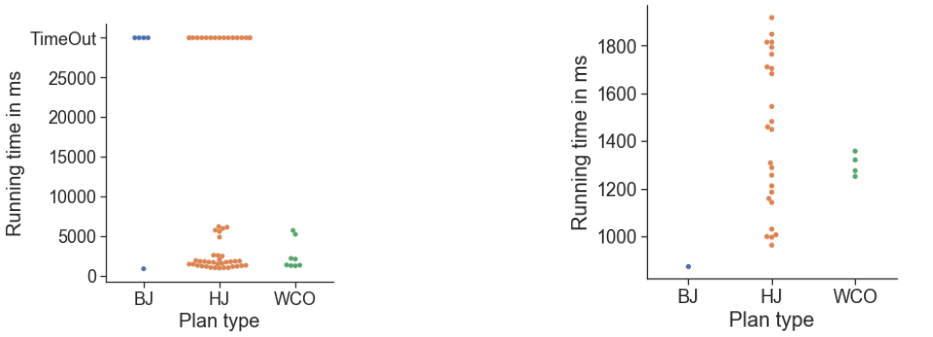
Let us look at the particular query where a binary plan was the fastest. Figure 4.6 shows the fastest plans for each plan type on this query. Immediately we can see that the reason for the low execution time of the binary plan is in the intermediate result sizes, especially how the first join required a low amount of tuples from the edge tables to perform the join. There is a low amount of skew on the vertices $a1$ and $a5$. In the WCO plan, we can see that enumerating them added very little to the total output size. Computing all unique tuples for the vertices $(a2, a3, a4)$ took the most work. Interestingly we can see from Figure 4.7 that there was only one binary plan

that computed the output within the timeout. All the other potential join orderings created large intermediate result sizes on the first join.



(a) **Hybrid plan** (b) **WCO plan** (c) **Binary plan**

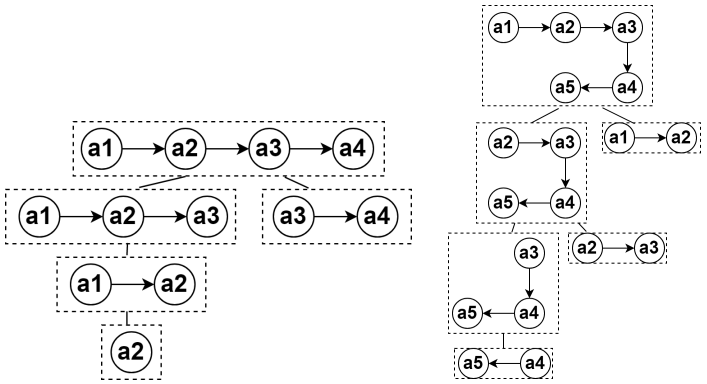
Figure 4.6: **The fastest plans of each plan type for the 4-chain query where the binary plan was the fastest. The bold number are the number of tuples produced by each step.**



(a) **Showing all plans** (b) **Focused on the fastest plans**

Figure 4.7: **4-chain query where the binary plan is the fastest**

As discussed in Section 3.1.2, we allow lonely variables to appear not at the end for hybrid plans because we were uncertain about the implications of not allowing this. Using edges that we create from binding both vertices of that edge using WCO joins creates edge tables that are filtered for that particular query. An edge that includes a lonely variable will only have one edge bound; thus it will still be filtered, albeit potentially less. Chain queries have lonely variables on both ends; a potential hybrid plan would be one that starts with a filtered edge that contains a lonely variable and then expand on that using binary or WCO joins. For every query we ran in our experiments, we analyzed which hybrid plan was the fastest; these are all shown in Appendix B. Figure 4.8 shows two of these plans. Note that in both plans, the plan starts with a filtered edge from two WCO joins that contains a lonely variable. Furthermore, while these plans are far outnumbered by other hybrid plans that bind lonely variables last, this shows that these plans can still be competitive. However, the hybrid plans that did bind lonely variables last were never far behind, so it can still be a safe option to eliminate them from our plan space.



(a) **3-chain 12/12** (b) **4-chain 1/0**

Figure 4.8: **Figure 4.8a shows a hybrid plan for a 3-chain query and Figure 4.8b for a 4-chain query. Both plans start with a filtered edge from two WCO joins that contains a lonely variable. The left value indicates the number of times this plan was the fastest hybrid plan; the right value indicates how often this plan was the fastest plan for the query, beating both WCO and binary plans as well.**

Star Queries

When looking at the star queries, we can see that the hybrid plans outperform the WCO plans. However, the difference is minimal. Not surprisingly, every single WCO and hybrid plan first computes the center vertex. Since it allows us to join many vertices at once and then enumerate the valid output by going over the remaining vertices that contain no joins since they are lonely variables. This enumeration step is logically the same for both the hybrid and WCO plans. The hybrid plan uses an index nested loop join, which means that for every binding of the center vertex, it does a binary search on the incident edge of each lonely variable and then enumerates all outputs. The leapfrog join works in a similar way where we go down a depth in the trie for each relation, as explained in section 3.2.1. Here, the extra overhead caused by the added complexity required for the leapfrog join makes the hybrid plans more efficient than a pure WCO plan. The actual important part of the query is the efficient binding of the center vertex, which is why both hybrid and WCO plans outperform binary plans.

Interestingly on the 5-star query, the binary plans start to perform better compared to the 3-star and 4-star queries. On the latter binary plans run a magnitude slower compared to both WCO and hybrid plans. However, on the 5-star query, they only execute 4x slower on average. Fast WCO and hybrid plans bind the center vertex and then merely enumerate the lonely variables. We would expect that the more edges participating in the multiway join of the center vertex, the more the intermediate results sizes would be reduced compared to the binary plan. We think that potentially two factors are at play here.

- A high number, about 60%, of our sampled 5-star queries timed out. Meaning at least one of the plan types could not finish the execution of any of their plans for that query within the timeout period. Not surprisingly, this plan type in all cases was the binary plan. Two problems are created here; the data set is now smaller, and therefore the data more variable. Furthermore, pruning specific queries creates a bias in our set of queries. Possibly the pruned queries all had far higher runtime ratios between binary and WCO/HJ plans. It is, however, also possible that the pruned queries would have shown the same results but with higher run times on all plans involved.

  Resampling a different set of queries would have been possible to increase the size of the data set. However, doing so would still only allow us to include queries that can have all

plan types execute at least one of their plans within the timeout. Therefore it would still have the same bias.

- The other explanation we hypothesize to be the following: when the number of edges incident to the center vertex in a star query increases, it also exponentially grows the unique pairs of edges that can be joined together as the first step in a binary plan. The high number of possible joins gives the binary plan a higher chance to start with a selective join that keeps intermediate result sizes down. The first join creating a small intermediary result, impacts the overall computations required immensely.

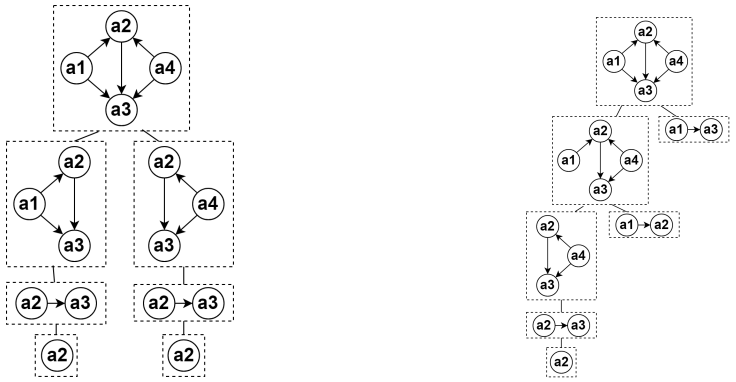
Notably, there is still a plan for the 5-star query where the binary join is over three magnitudes slower; this plan has seven output tuples. Skew plays a significant role here. In this particular instance, the fastest WCO plan only has to bind seven values in the center vertex. More importantly, the iterator performed less than 100 binary searches across the five edge tables to produce these results. Some of the relations were either sparse or had minimal overlap with each other. The binary plan, on the other hand, first has to compute a 2-chain. Computing the 2-chain allows either a heavily skewed node on the query to produce massive intermediate results or for a join to happen on two large sparse relations with little overlap. The latter happened here; one particular edge table required nearly half a million iterations in the initial 2-chain join. Furthermore, among all binary plans, this plan was the fastest. Different plans with other pairs of edge tables as the first join had even higher values.

So while the data suggests that on the average binary plans do better on 5-star queries compared to 3-star and 4-star queries, they still do not outperform WCO and hybrid plans that bind the center vertex. Also note that while on the average this happens, it does not happen on the extremes. Both for the largest and smallest ratio, the values indicate binary plans performing worse on the 5-star compared to the 3-star and 4-star queries. Which is why we suggest there is a bias in this experiment.

Combined Queries

Combined queries are queries that we define as a combination of star, chain, or cyclic queries. For example the bowtie query is two 3-cycles joined together. Combined queries often contain more edges and vertices; therefore, their plan space is richer as well. Table 4.2 shows the results for the combined queries in our experiments. Notably, we can see that hybrid plans perform, on average equal to WCO plans on bowtie and diamond queries. Hybrid plans perform better than WCO plans on Q1 and 2-snow queries, but on average only by a small margin. Both 2-snow and Q1 contain lonely variables. We have seen so far that enumerating these lonely variables often favors hybrid plans. Because the binary join can enumerate the lonely variables with lower overhead than the leapfrog join can, so this result is not surprising.

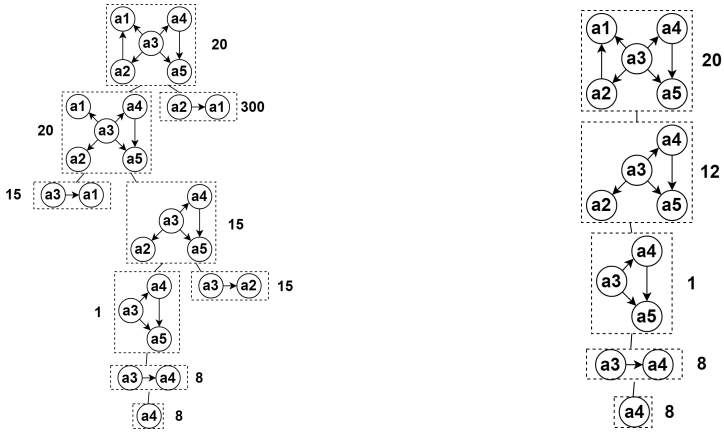
Hybrid plans performing poorly on bowtie and diamond can be explained by bushy plans being too costly. Figure 4.9 shows two plan types that were the fastest hybrid plan for some of the queries in our experiments. Figure 4.9a shows a bushy plan that computes two 3-cycles, which are then joined together. This approach was the fastest hybrid plan on 9 of our queries, executing faster than the WCO plan only twice. On the other hand, Figure 4.9b shows a left-deep hybrid plan that first computes a 3-cycle using WCO joins and subsequently extends the subpattern one edge at a time and closes the cycle using a binary join. This approach was the fastest hybrid plan on 77 queries, executing faster than the WCO plan 46 times. The benefits gained from computing both 3-cycles separately, thus reducing intermediate results, are outweighed by the downsides of bushy plans. Bushy plans require intermediate results to be materialized on at least one side of the join; in our case, we build a hash table. Left-deep plans can pipeline each result tuple into the next operator without having to materialize these results. We see a similar pattern for the bowtie query; for this query, bushy plans never execute faster than left-deep plans.



(a) **Bushy plan** (b) **Left-deep plan**

Figure 4.9: **Two possible hybrid plans for the diamond query.**

Note that the left-deep hybrid plans on both the bowtie and diamond queries close a cycle using binary joins. In the case of the diamond query, these plans execute faster than WCO plans more often than not. However, on both queries, the first cycle is always computed using WCO joins, only the second cycle is computed using binary joins. We hypothesized this to happen whenever the binary joins produce a similar amount of intermediate results as the WCO joins, which means the WCO joins are not able to filter aggressively. Let us look at one of the queries where the left-deep hybrid plan is faster than the WCO plan. Figure 4.10 shows both the fastest hybrid and WCO plan for a query where the hybrid plan is a left-deep plan that closes the second cycle using binary joins. In this case, the hybrid plan was 2x faster than the WCO plan. The data shows that the binary joins produced intermediate results that were not much greater than those produced by the WCO joins. The extra overhead from the WCO joins caused these plans to be slower while producing smaller intermediate results.



(a) **Hybrid plan** (b) **WCO plan**

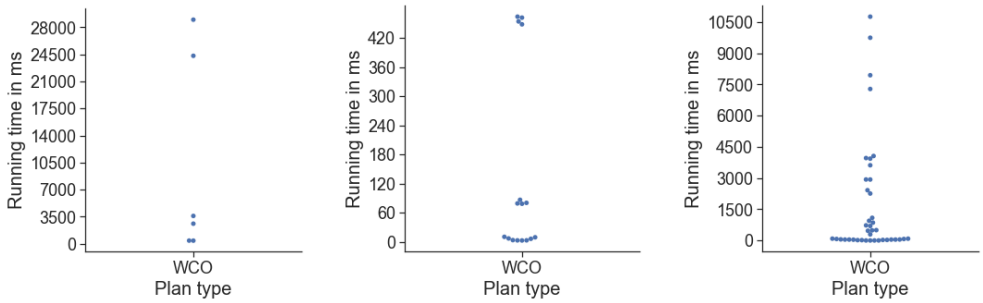
Figure 4.10: **Hybrid and WCO plan for a query where the hybrid plan was faster. The values in bold indicate the number of tuples produced by that step of the plan.**

The 2-snow and Q1 queries are interesting because they are more complex than chain queries but also contain lonely variables. There are hybrid plans that bind lonely variables with a WCO join before binding all other variables. On the less complex chain queries, we saw instances where these plans were faster than plans that pushed lonely variables to the end. We already advised when discussing the chain queries that we could safely remove these plans from our plan space.

The data for the 2-snow and Q1 queries further verifies this. For each fastest hybrid plan for the 2-snow query it first computed the vertices $(a1, a2, a3)$ and then subsequently enumerated the lonely variables. The same is true for the Q1 query. All these plans can be seen in Appendix B. The difference between these two queries and the chain queries is the presence of vertices with arity > 2. Binding these vertices early using WCO joins produces low intermediate results.

Importance of QVO's

Figure 4.11 sums up the importance of QVO's. For three queries, we show scatter plots of all WCO plans in our plan space. Each data point is a different QVO. The difference between a bad QVO and a good QVO can be orders of magnitude. A planner should optimize for this to guard itself against bad QVO's. Notably, EmptyHeaded does not optimize its QVO's, potentially risking bad plans. Graphflow, on the other hand, goes to great lengths to avoid bad QVO's. Besides selecting a QVO at the planning phase, they allow the QVO to change dynamically at runtime by updating their estimates with real data. Section 2.4.2 went in-depth regarding adaptive QVO's in Graphflow.



(a) **3-cycle** (b) **4-cycle** (c) **5-cycle**

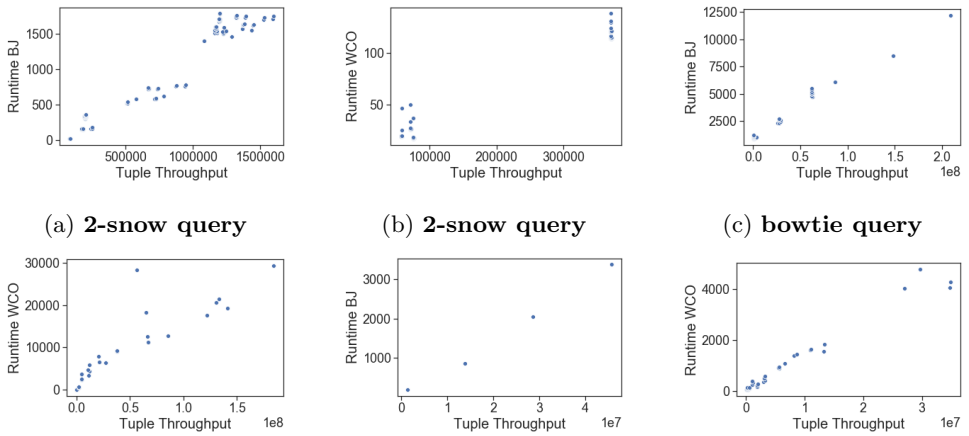
Figure 4.11: **Three examples of the importance of choosing the right QVO. Each data point in the scatter plot is a WCO plan in our plan space. Together they cover all valid QVO's.**

4.4 Comparing WCO and Binary Algorithms

We study the effects of computations on the runtime of WCO and binary plans. In essence, we want to compare the tuple throughput of WCO and binary algorithms. The goal here is to find a ratio between the tuple throughput of both types of algorithms that allows us to build a cost model for hybrid plans. To assign a cost to a hybrid plan, we require a method to accurately compare the cost of a binary to a WCO join. We aim to find out if we can find such a ratio can. Furthermore, find out if we can use these ratios across different query types and datasets, or if we have to recompute them on a per-query basis.

To analyze if this is possible, we create scatter plots over the data for specific queries. These plots can be seen in Figure 4.12. For each query, the plots show a data point for the total tuple throughput of the entire plan compared to the runtime. We do this for every plan that finished within the timeout window. These plots allow us to visually confirm whether or not the tuple throughput of BJ and WCO plans are consistent over various running times. If these scatter plots did not show an even upward trend, we could determine that there is a limited correlation between the runtime and the total tuple throughput.

Now we use the data in these plots and data from more queries. Using this data, we can compute ratios between the BJ and WCO plans. We compute these ratios as follows: we denote the total



(a) **2-snow query** (b) **2-snow query** (c) **bowtie query**

(d) **bowtie query** (e) **5-cycle query** (f) **5-cycle query**

Figure 4.12: **Scatter plots of three queries. Each data point in one plan. The y-axis is the runtime of that plan in ms. The x-axis is the total tuple throughput of that plan**

tuple output of BJ plans as tt_b and WCO plans as tt_w. Similarily we denote the runtime of BJ plans as r_b and WCO plans as r_w. Now for each possible pairings of WCO and BJ plans within a query, we compute a ratio as follows:

$$ratio = \frac{tt_b/r_b}{tt_w/r_w} \tag{4.1}$$

This gives us many ratios for each query. These ratios can be described as the tuple throughput of the BJ plans divided by the tuple throughput of the WCO plans. These ratios can tell us how expensive a BJ join is compared to a WCO join. However, this data is only valid if these ratios are consistent across different query types and data instances. To test this, we collected data from multiple query types, and we present this data in Table 4.3. Each row in this table is a single query. For each query, we compute a set of ratios, and for this set of ratios, we list the following: maximum value, minimum value, mean, median, variance, and standard deviation.

In this table, we need to look at how the ratios are within a query type and how they are between query types. Note that while the dataset here is small, it still gives us valuable insight. If we use a ratio in a cost-based plan optimizer to compare WCO to binary joins, then this ratio should hold, within a margin, for all queries to be useful. As we can see from the values in the table, this is not the case between query types. The mean ratio between, for instance, a bowtie and a 2-snow query can differ a magnitude. Even within a query type, we see significant differences, for instance, on the diamond query. Therefore, a ratio computed according to this method will not be useful to assign accurate costs to hybrid plans.

Instead, a more costly approach is required that computes query and data instance-specific heuristics to estimate the costs of WCO and binary joins. These heuristics would have to be computed by probing the relevant data or stored statistics. Not only to be able to estimate intermediate result sizes of each layer in the query plan but also to estimate the cost of the join itself.

Interestingly Graphflow uses the following approach. They experimentally run both WCO and binary plans, and for the WCO plans, log the (i-cost, runtime) pairs, and for the binary plans log the $(n_1, n_2$, runtime) pairs. Where n_1 and n_2 are metrics from their hash join operator as explained in Section 2.4.2. Using these pairs, they can decide on a set of weights w_1, w_2 to assign

Query Type	Max	Min	Mean	Median	Variance	SD
2-snow	1.059	0.162	0.328	0.312	0.01	0.099
2-snow	2.567	0.203	0.546	0.382	0.161	0.401
5-cycle	7.237	0.112	1.006	0.682	0.866	0.93
5-cycle	7.165	0.631	2.375	2.196	1.523	1.234
5-star	1.788	0.099	0.728	0.734	0.173	0.416
5-star	0.608	0.247	0.416	0.419	0.006	0.08
5-chain	0.963	0.161	0.558	0.637	0.053	0.231
5-chain	0.427	0.12	0.247	0.227	0.007	0.087
bowtie	11.768	0.198	3.305	2.688	7.911	2.813
bowtie	14.979	0.061	1.752	0.57	5.3	2.302
diamond	8.172	0.064	1.025	0.448	2.0	1.414
diamond	8.52	0.164	2.897	2.621	3.081	1.755
Q1	1.226	0.16	0.443	0.354	0.052	0.228
Q1	2.165	0.152	0.52	0.433	0.114	0.338

Table 4.3: **Each row is a specific query of the type in the left column. a set of ratios is computed according to Equation 4.1, and for this set of ratios we list the following: maximum value, minimum value, mean, median, variance, and standard deviation.**

to their hash join operator that best fits the running times of both pairs. The one difference of their approach to ours is that they log data on a per join basis while we observe the entire plan at once.

4.5 Hybrid Plans, Are They Worth It

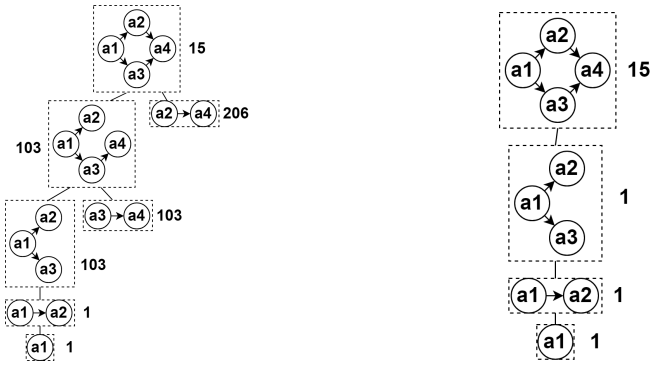
So far, we have looked at our queries by comparing binary, WCO, and hybrid plans while taking the binary plans as our baseline. From the results in Tables 4.1 and 4.2, we can see how hybrid and WCO plans compare to each other on average however we cannot find queries where one greatly outperforms the other. It is especially interesting to find out in what circumstances hybrid plans do outperform WCO plans.

Table 4.4 shows for each query type how the fastest hybrid and WCO plans compare to each other. We calculated all the runtime ratios between the fastest hybrid and WCO plans of each query within its type. The table shows the largest and smallest of these ratios. This data shows that for the majority of query types, the hybrid and WCO plans are within very close margins of each other. Unsurprisingly, the hybrid plans always ran slower on the 3-cycle and 4-clique queries in our experiments. For the 4-clique query, every vertex has an arity of 3, greatly benefitting the multiway leapfrog join algorithm. The star queries show the hybrid plans outperforming the WCO plans, as discussed before. This is caused by binary joins being able to enumerate lonely variables more efficiently in many cases. There are some query types, most notably the 4-cycle and diamond queries, where the runtime ratios are variable. However, even in these cases, we never see a speedup of more than 4x either way.
So far, we have seen that in the case where the WCO plan was faster than the hybrid plan, this was due to the hybrid plans computing a higher number of intermediate results. While the case where the hybrid plans were faster the intermediate results were kept low. Figure 4.13 shows an example where the WCO kept the intermediate results low but still executed twice as slow as the hybrid plan. Here we also close a cycle using binary algorithms instead of using WCO algorithms.

Why would closing a cycle using a binary algorithm ever be quicker than using a WCO algorithm? We are still enumerating over the same set of data, just in different ways. Our WCO algorithm, the leapfrog triejoin, binds the final vertex by intersecting the edges $a2 \rightarrow a4$ and $a3 \rightarrow a4$. It

	Largest Ratio	Smallest Ratio
3-cycle	0.734	0.0013
4-cycle	2.412	0.326
5-cycle	1.500	0.758
3-chain	1.770	0.925
4-chain	2.138	0.960
5-chain	1.743	0.996
3-star	1.328	0.921
4-star	1.308	0.996
5-star	1.195	1.001
2-snow	1.480	0.987
diamond	2.182	0.199
bowtie	1.085	0.839
Q1	1.404	0.957
4-clique	0.32	0.028

Table 4.4: **This Table shows the differences between hybrid and WCO plans for the queries as seen in Figure 4.2. For each query of every query type the fastest WCO plan was compared to the fastest hybrid plan. The values in the table are the relative runtime of the WCO plan compared to the hybrid plan. The largest ratio indicates where this ratio was the largest, thus where the hybrid plan was relatively the fastest among all queries of that type, similarly for the smallest ratio.**



(a) **Hybrid plan** (b) **WCO plan**

Figure 4.13: **This figure shows the fastest hybrid and WCO for a 4-cycle query, where the hybrid plan is 2x faster than the WCO plan. The bold numbers are the number of tuples produced by each step.**

only intersects the edges filtered by the previously computed bindings. On the other hand, the binary algorithm uses index nested loop joins. It extends vertex $a3$ by using a binary search on the edges $a3 \rightarrow a4$ for every binding for $a2$, this is done similarly for vertex $a2$. Afterward, we have a 4-chain, which is then filtered to a 4-cycle using a selection on vertex $a4$.

Instinctively we would say that the WCO plan should perform faster since the hybrid plan has more intermediate results. It comes down to the efficiencies of the algorithms. While both use binary searches to find values quickly, the leapfrog join can be cache inefficient. The leapfrog join iterates through the edge relations using binary searches for every iteration until a join is found. While the index nested loop join only uses one binary search for every binding. Afterward, a linear iterator enumerates all values of edges since they are pre-sorted. The number of binary searches here is probably larger for the leapfrog join even with the lower amount of intermediate results,

which results in the larger running time.

It then becomes interesting whether or not this is specific to our intersection method, which is leapfrogging, or if this would affect linear or other more cache-friendly intersection methods as well. The EmptyHeaded paper [3] includes a survey of multiple intersection methods implemented using SIMD instructions. One of their methods, called SIMDGalloping, uses the leapfrogging principle used by the leapfrog join. They note that the SIMDGalloping algorithm only outperforms a block-wise algorithm whenever the ratio of the cardinalities of both relations goes over 1:32. Because the cardinality of the smallest relation bounds the number of binary searches performed. Their experiments illustrate that binary searches need to skip enough work to be worthwhile. Note, however, that these experiments are performed on set intersections between at most two relations, and not on n-way intersections. As illustrated before when more sets are involved the overlap between all the sets becomes smaller and more data can be skipped in each binary search.

Bushy vs Left-deep Plans

Table 4.5 shows the ratio between bushy and left-deep hybrid plans for all the queries where bushy hybrid plans are in our plan space. This ratio is the relative number of times that a bushy or left-deep plan was the fastest hybrid plan. In this case, we define a bushy hybrid plan as being bushy if any subpart of the plan joins two sub plans.

	Bushy Plan	**Left-deep Plan**
4-cycle	0.033	0.967
5-cycle	0	1
5-chain	0	1
4-star	0	1
5-star	0	1
2-snow	0.02	0.98
diamond	0.102	0.898
bowtie	0	1
Q1	0	1
4-clique	0	1

Table 4.5: **This Table shows the ratio between bushy and left-deep hybrid plans for all the queries where bushy hybrid plans are in our plan space. The ratio is the relative number of times that a bushy, or left-deep plan was the fastest hybrid plan. In this case a bushy hybrid plan is defined as being bushy if any sub part of the plan joins two subplans.**

It is not surprising that bushy plans do not perform well on queries such as chain and star queries. However, even on queries where we could expect them to do well, such as bowtie and diamond queries, they do not perform well. Figure 4.14 shows two scatter plots for the diamond query containing all hybrid plans separated by bushy and left-deep plans. In Figure 4.14a, the fastest bushy plan executes 7.9x faster than the fastest left-deep plan. While in Figure 4.14b, we see the complete opposite. Here the left-deep plan is 198x faster.

This result does not come at a surprise when we compare intermediate result sizes for both queries. In both cases, the fastest bushy hybrid plan computes both cycles and joins them together. On the second query computing, the first cycle produced two tuples, while computing the second cycle produced 318644 tuples. The full query output is 24 tuples. On the other hand, the left-deep plan extended the two tuples from the left cycle one edge at a time. This process produced far fewer

(a) **Bushy plan 7.9x faster.** (b) **Left-deep plan 198x faster.**

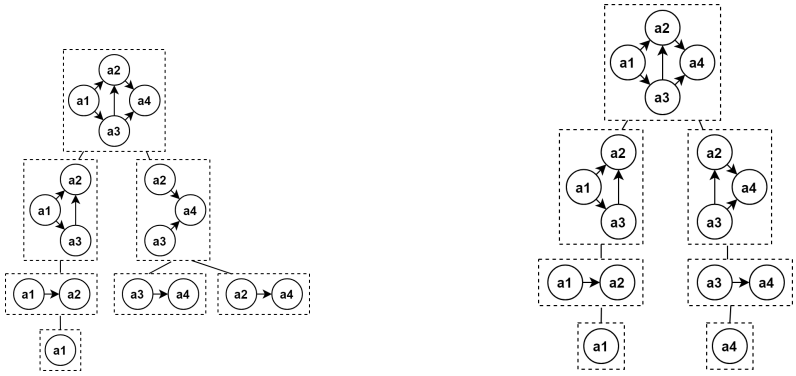
Figure 4.14: **Two scatter plots for the diamond query showing all bushy and left-deep hybrid plans. Each plan is a data point in the scatter plot.**

than the 318644 tuples for the bushy plan, hence the 198x speedup. We hypothesized that, in scenarios like this, bushy plans would not perform well.

On the first query where the bushy plan was faster, we see the following: computing the left cycle produced 441 tuples, and the right cycle 14046. The full query produced 55965 tuples. Meanwhile, the fastest left-deep plan started from an edge and extended that edge twice by one vertex. Interestingly even the fastest WCO plan, which computes the left cycle and then extends that to the full query, was 2x slower than the bushy plan. This tells us that bushy hybrid plans can be optimal in certain instances. For instance, when the final output size compared to the intermediate results of both sub plans is large enough to warrant the expensive hash join.

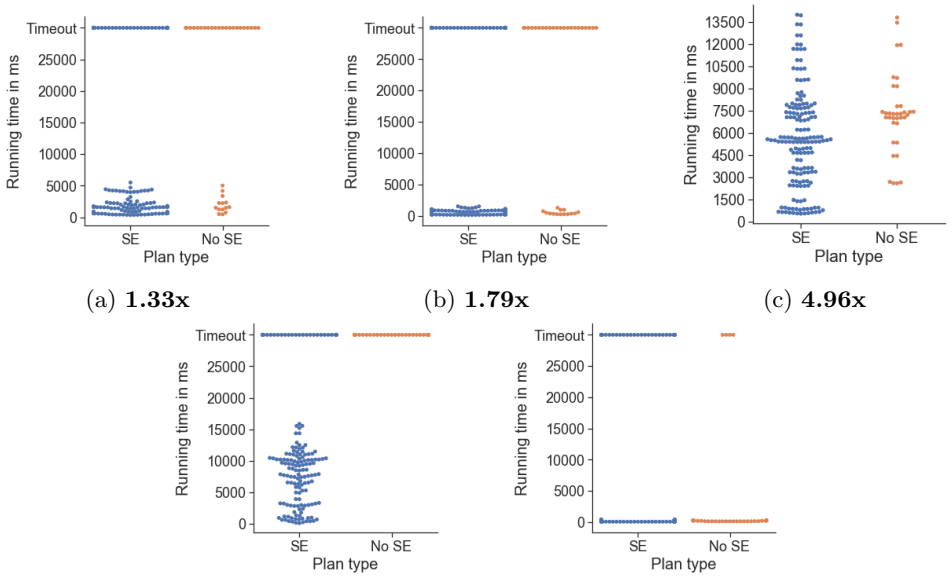
4.6 Shared Edge Plans

Shared edge plans in our experiments only happen on the diamond query, Figure 4.15 shows the two plans we are interested in here. Both are bushy hybrid plans however plan 4.15b has edge $a3 \rightarrow a2$ in both sub-plans while plan 4.15a does not. We hypothesized that the shared edge plan should be faster because it reduces the intermediate results sizes before performing the binary join. The reduction of the computation time of the binary join should far outweigh the potential increase in computation time for the sub-plan. We say potential here because it is possible that computing the 2-chain using a merge join requires more time than computing a 3-cycle using WCO joins. A merge join is required here because we cannot compute the 2-chain using WCO joins, binding each vertex of the 2-chain in turn already computes the 3-cycle. ‘
We classify shared edge plans on the diamond query as plans that compute both 3-cycles and then join them together using a binary join. These 3-cycles can be computed in several ways, but we do not consider plans that compute them using binary joins since we have already seen these to be suboptimal in previous experiments. Thus we consider any 3-cycle sub-plan that we compute through a hybrid or full WCO sub-plan. We define the no shared edge plans as plans that join a 3-cycle with a 2-chain using a binary join. We compute the 2-chain by a binary join, and we compute the 3-cycle as in the shared edge plan. Figure 4.16 shows scatter plots for five different diamond queries where we compare shared edge plans versus no shared edge plans. From this data, we can see that including shared edge plans in our plan space is a clear benefit. On one occasion, the no shared edge plans were unable to compute the query at all within the timeout window.



(a) **Plan without duplicated edge** (b) **Plan with duplicated edge**

Figure 4.15: **Two plans for the diamond query with the shared edge plan on the right, meaning an edge occurs in both sub plans, $a3 \rightarrow a2$ in this case**



(a) **1.33x** (b) **1.79x** (c) **4.96x**

(d) **N/A** (e) **1.86x**

Figure 4.16: **Scatter plots of five diamond queries. Timeout indicates plans that did not finish running. 'SE' means shared edge plans as in Figure 4.15b and 'no SE' are plans of the form in Figure 4.15a. The value in bold indicate how much faster the fastest 'SE' plan is compared to the fastest 'no SE' plan.**

Chapter 5

Conclusions and Future Work

5.1 Conclusion

Throughout this thesis, we have presented a framework to implement hybrid plans in a graph database. We have introduced concepts, such as worst-case optimality and the AGM bound, that are the theoretical basis for this framework. Furthermore, we have presented various algorithms that are worst-case optimal. The work in this thesis closely resembles the work done by [8, 3]. However, our approach is different in some aspects, such as our plan space and the WCO algorithm we use. As our WCO algorithm, we adapted and implemented the leapfrog triejoin algorithm [13]. All join algorithms used in our framework have been adapted to adhere to the same interfaces such that they can be used in hybrid plans. We evaluated our approach by building this framework into Avantgraph, an in-memory graph database.

For our experimental evaluation, we built a query miner in Avantgraph to extract several SGM queries from the YAGO2s dataset [2]. In our experiments, we have compared the performance, through several metrics, of WCO, binary, and hybrid plans. We have established that WCO plans outperform binary plans across all query types, including queries where binary plans do not violate worst-case optimality such as chain queries. Furthermore, we have seen hybrid plans outperform WCO plans on average by about 10%, and in some cases, by as much as 2.5x faster execution time. We have also seen several query types, such as the 3-cycle and 4-clique, where WCO plans run faster across all experiments. This is due to the high connectivity of the query favoring vertex by vertex joins.

Hybrid plans can be divided into two groups, bushy and left-deep hybrid plans. Bushy hybrid plans use binary joins to combine subpatterns of the SGM query. Meanwhile, left-deep hybrid plans only use binary joins to extend the subpattern by an edge. We have shown left-deep hybrid plans to outperform bushy hybrid plans. We see the effectiveness of binary joins in these left-deep hybrid plans when the binary join can replace a WCO join that is unable to reduce the intermediate result size substantially. Bushy hybrid plans require intermediate results to be materialized in order to compute the join efficiently. Our experiments show that joining the same edge in both subpatterns of the bushy plan produces faster plans by reducing the intermediate result sizes. Given the results presented here, the potential colossal plan space can be reduced significantly to make hybrid plans a worthwhile investment for any query planner.

The main goal of this thesis was to explore worst-case optimal plans in both their pure form and in a hybrid form. Using the insights learned in Chapter 4, the next logical step would be to create a cost-based planner to select fast plans out of the full plan space using cardinality information. The challenge here lies in the variety of algorithms used. A cost-based planner would have to normalize the different cost models for each approach so that they can be compared. We have

presented findings that show that this is not trivial. Some interesting research on this topic has already been conducted by [8].

5.2 Future Work

Efficient intersections are at the core of the leapfrog join algorithm. The iterators have to be able to find values in constant time to achieve worst-case optimality. The only way to accomplish this is by hashing your data. Instead, we execute within a log factor by using binary searches. However, we should look outside of what is theoretically the best but instead look at practicality. EmptyHeaded [3] already experimented with different intersection methods, their findings say that block-wise intersections making use of SIMD are more optimal in many cases. These intersection methods can be faster because they are more cache efficient, and they make use of the latest in processor technologies. Adapting the work in this thesis to work with these new intersections methods could be the next step.

Our experiments show that bushy hybrid plans do not perform well. Bushy plans require the materialization of intermediate results, and, in our case, require work to build a hash table. But what if we can replace bushy hybrid plans by pure WCO plans. Is there a way to treat subpatterns as vertices in such a way that we can continue to pipeline the results. Is there potentially a way to do this without having to materialize the intermediate results? Or are we bound to pre-computing the subpattern to be able to treat it as a vertex in a WCO plan? This line of research could be of interest in future work.

Bibliography

[1] Logicblox. https://developer.logicblox.com/. Accessed: 2020-02-04. 10

[2] Yago2s dataset. https://www.mpi-inf.mpg.de/departments/databases-and-information-systems/research/yago-naga/yago/. Accessed: 2019-11-8. 40, 57

[3] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. *arXiv e-prints*, page arXiv:1503.02368, Mar 2015. 8, 17, 20, 54, 57, 58

[4] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *arXiv e-prints*, page arXiv:1711.03860, Nov 2017. 1, 8

[5] Ronald Fagin. Degrees of acyclicity for hypergraphs and relational database schemes. *J. ACM*, 30(3):514550, July 1983. 17

[6] G. Graefe. Volcano: An extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. 3, 31

[7] Chathura Kankanamge, Siddhartha Sahu, Amine Mhedbhi, Jeremy Chen, and Semih Salihoglu. Graphflow: An active graph database. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD 17, page 16951698, New York, NY, USA, 2017. Association for Computing Machinery. 8, 18

[8] Amine Mhedhbi and Semih Salihoglu. Optimizing Subgraph Queries by Combining Binary and Worst-Case Optimal Joins. *arXiv e-prints*, page arXiv:1903.02076, Mar 2019. 8, 17, 20, 36, 38, 39, 42, 57, 58

[9] Hung Q. Ngo. Worst-Case Optimal Join Algorithms: Techniques, Results, and Open Problems. *arXiv e-prints*, page arXiv:1803.09930, Mar 2018. 1

[10] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case Optimal Join Algorithms. *arXiv e-prints*, page arXiv:1203.1952, Mar 2012. 1, 8, 9

[11] Hung Q. Ngo, Christopher Re, and Atri Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *arXiv e-prints*, page arXiv:1310.3314, Oct 2013. 1, 8, 9, 17

[12] Dung Nguyen, Molham Aref, Martin Bravenboer, George Kollias, Hung Q. Ngo, Christopher Ré, and Atri Rudra. Join Processing for Graph Patterns: An Old Dog with New Tricks. *arXiv e-prints*, page arXiv:1503.04169, Mar 2015. 1, 8, 16

[13] Todd L. Veldhuizen. Leapfrog Triejoin: a worst-case optimal join algorithm. *arXiv e-prints*, page arXiv:1210.0481, Oct 2012. 1, 3, 4, 8, 10, 12, 57

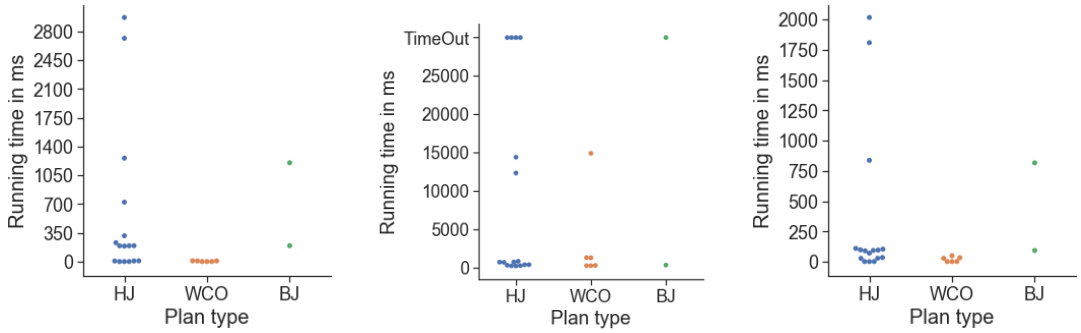
[14] Michael Waskom, Olga Botvinnik, Joel Ostblom, Saulius Lukauskas, Paul Hobson, Maoz-Gelbart, David C Gemperline, Tom Augspurger, Yaroslav Halchenko, John B. Cole, Jordi

Warmenhoven, Julian de Ruiter, Cameron Pye, Stephan Hoyer, Jake Vanderplas, Santi Villalba, Gero Kunter, Eric Quintero, Pete Bachant, Marcel Martin, Kyle Meyer, Corban Swain, Alistair Miles, Thomas Brunner, Drew O'Kane, Tal Yarkoni, Mike Lee Williams, and Constantine Evans. mwaskom/seaborn: v0.10.0 (january 2020), January 2020. 61

Appendix A

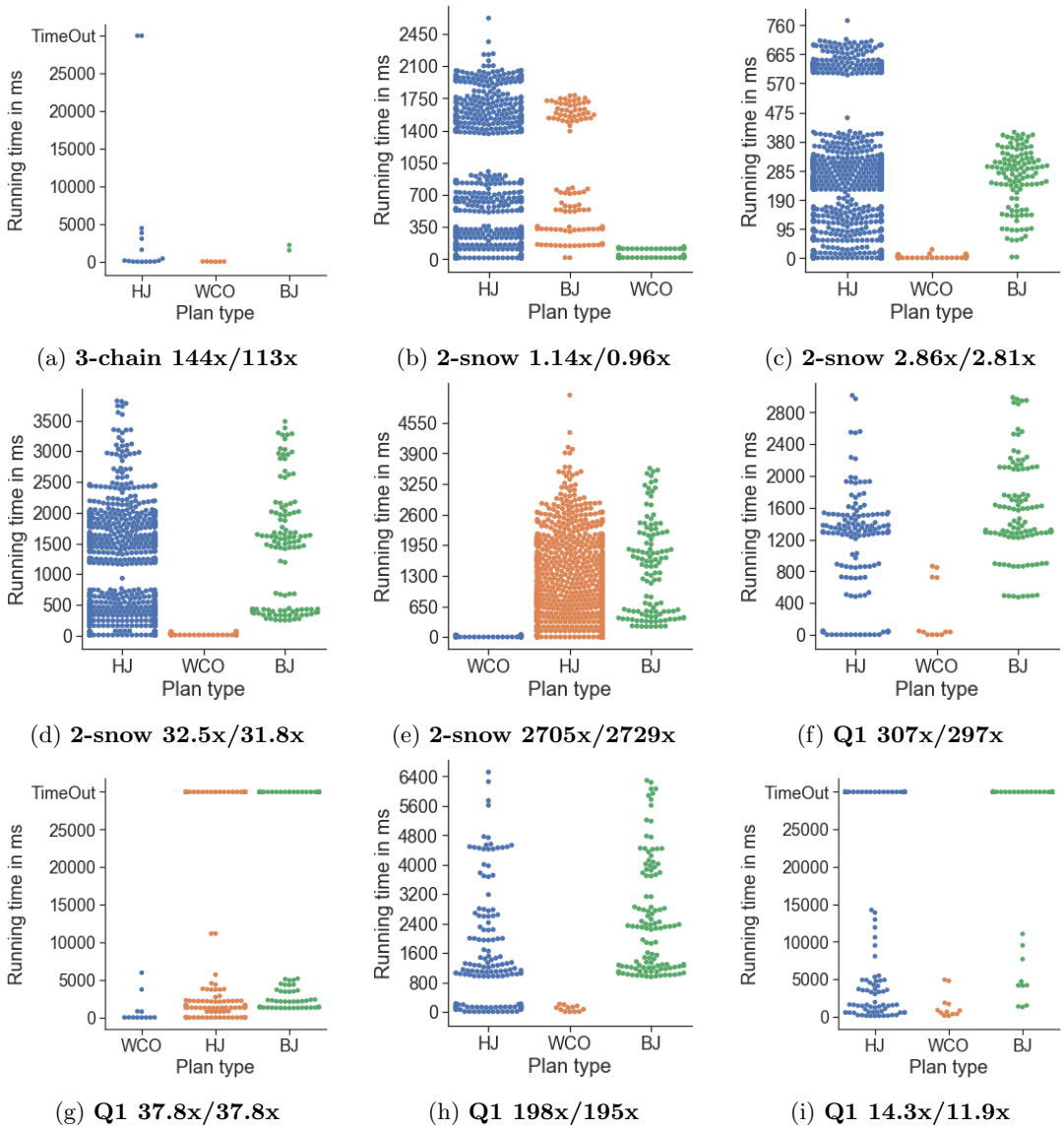
Scatter Plots of Queries

This appendix displays all the scatter plots for the queries that had the largest and smallest ratio in Tables 4.1, 4.2 and 4.4. The description of each scatter plot will indicate what type of query it is from and what the ratios between the fastest plans are. We made These plots using the seaborn python library [14], specifically using the catplot function. Each plan in our plan space is represented by one data point in the scatter plot. Plans of similar runtime are grouped and are displayed horizontally of each other. For any particular data point, the x-axis denotes the type of plan, and the position on the y-axis denotes the run time of the plan.



(a) **3-chain 924x/903x** (b) **3-chain 1.62x/1.38x** (c) **3-chain 57x/52x**

Figure A.1: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

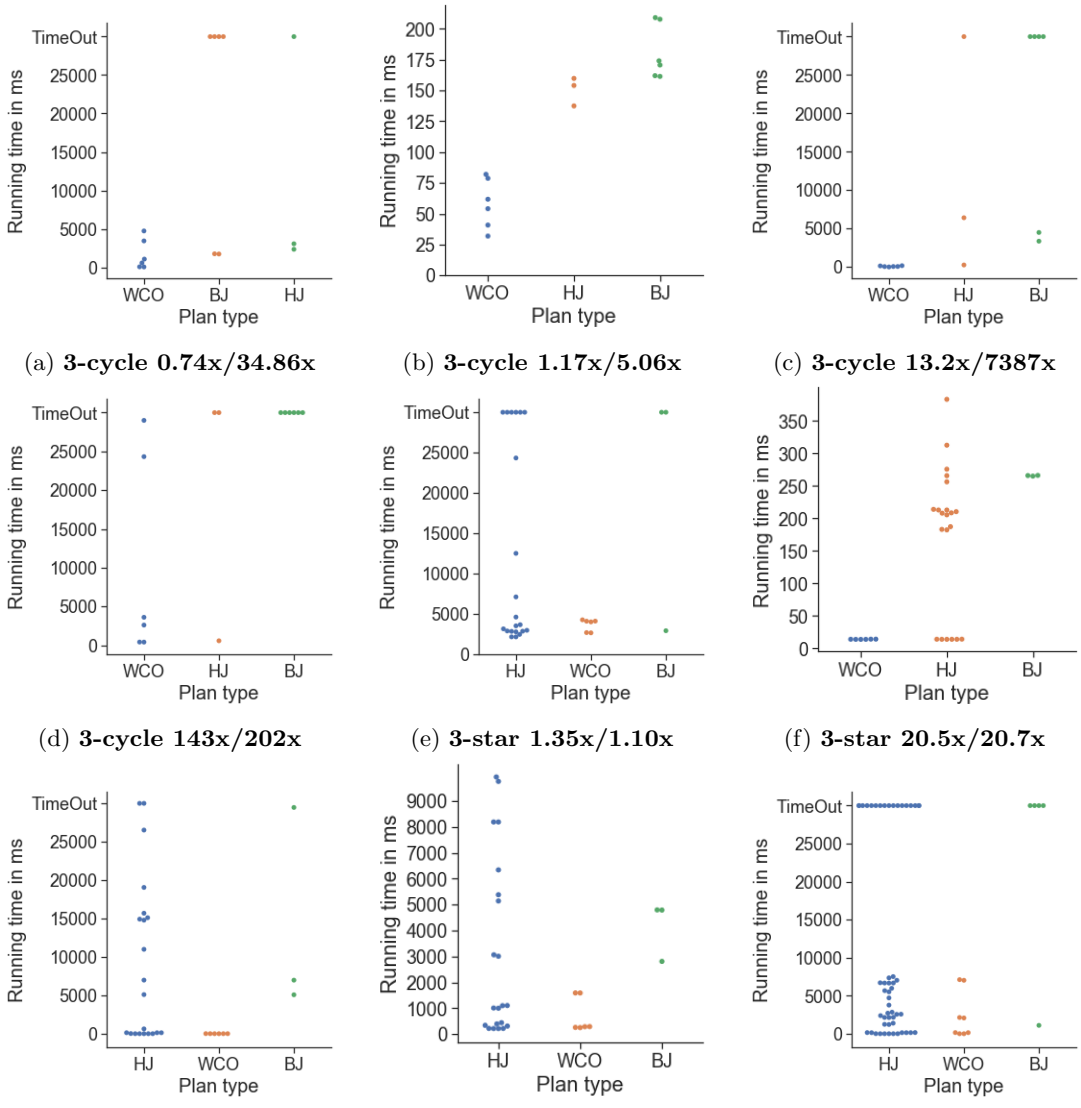


(a) **3-chain 144x/113x** (b) **2-snow 1.14x/0.96x** (c) **2-snow 2.86x/2.81x**

(d) **2-snow 32.5x/31.8x** (e) **2-snow 2705x/2729x** (f) **Q1 307x/297x**

(g) **Q1 37.8x/37.8x** (h) **Q1 198x/195x** (i) **Q1 14.3x/11.9x**

Figure A.2: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

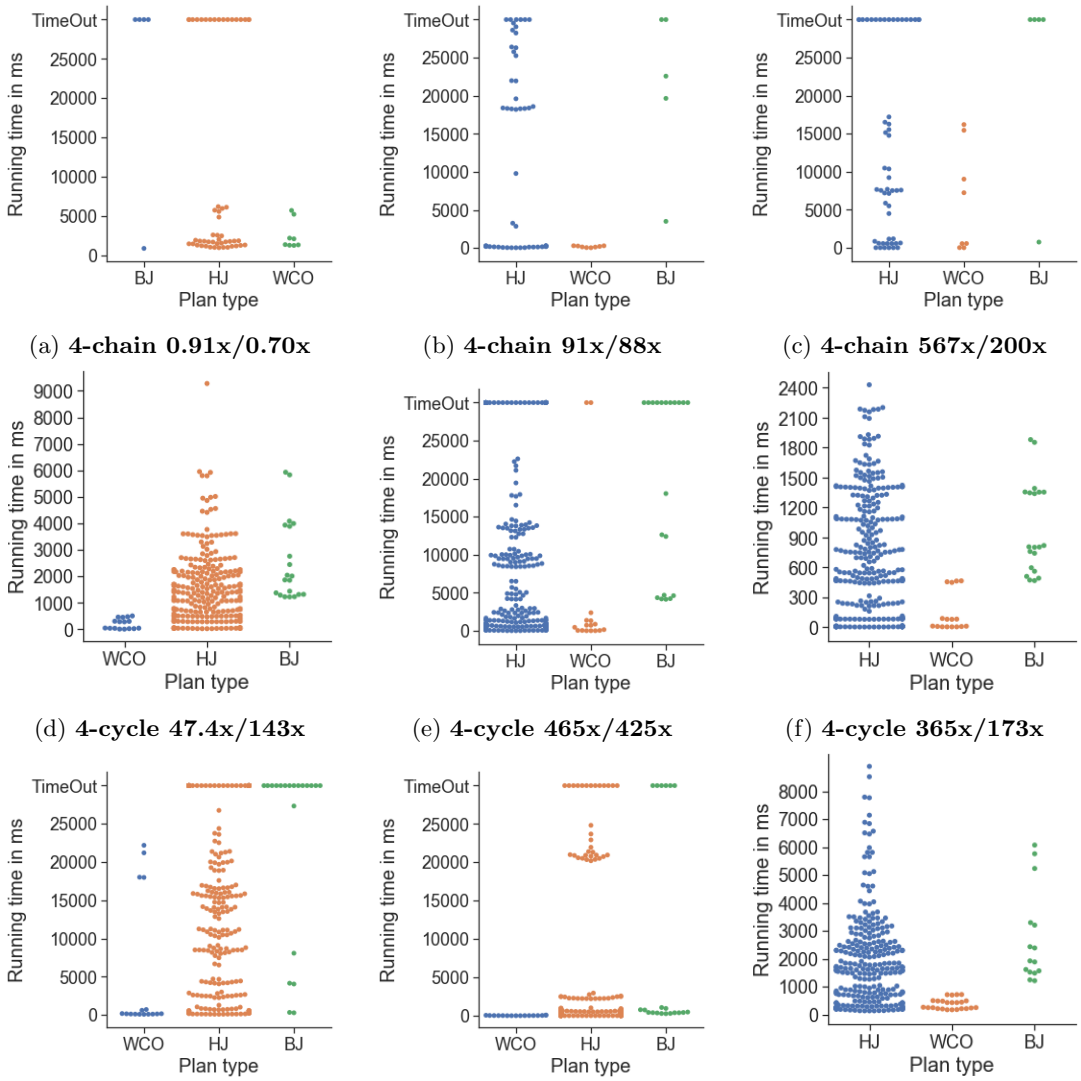


(a) **3-cycle 0.74x/34.86x** (b) **3-cycle 1.17x/5.06x** (c) **3-cycle 13.2x/7387x**

(d) **3-cycle 143x/202x** (e) **3-star 1.35x/1.10x** (f) **3-star 20.5x/20.7x**

(g) **3-star 164x/160x** (h) **3-star 12.86x/10.99x** (i) **4-chain 235x/197x**

Figure A.3: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

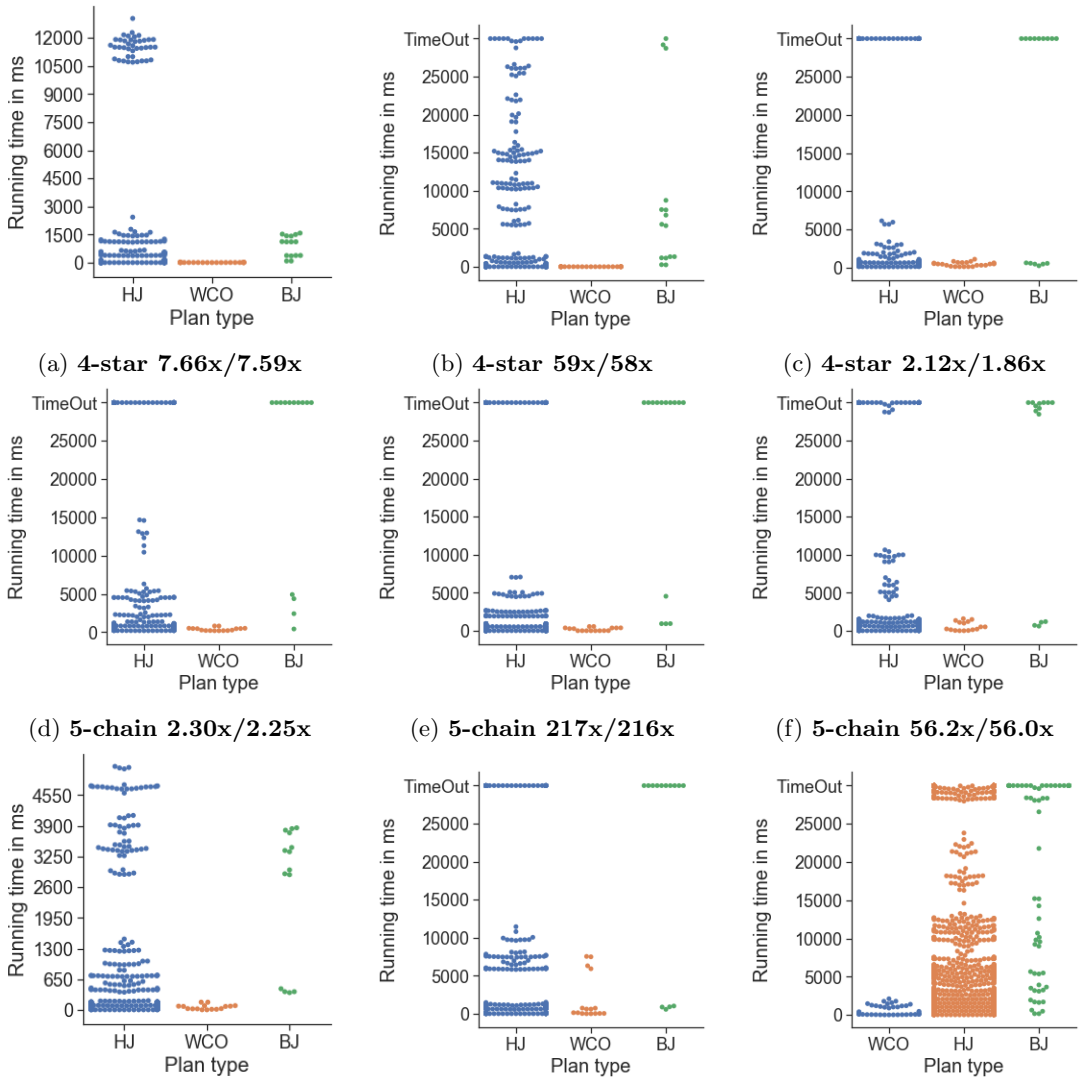


(a) **4-chain 0.91x/0.70x** (b) **4-chain 91x/88x** (c) **4-chain 567x/200x**

(d) **4-cycle 47.4x/143x** (e) **4-cycle 465x/425x** (f) **4-cycle 365x/173x**

(g) **4-cycle 2.96x/3.11x** (h) **4-cycle 458x/460x** (i) **4-star 9.11x/7.03x**

Figure A.4: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

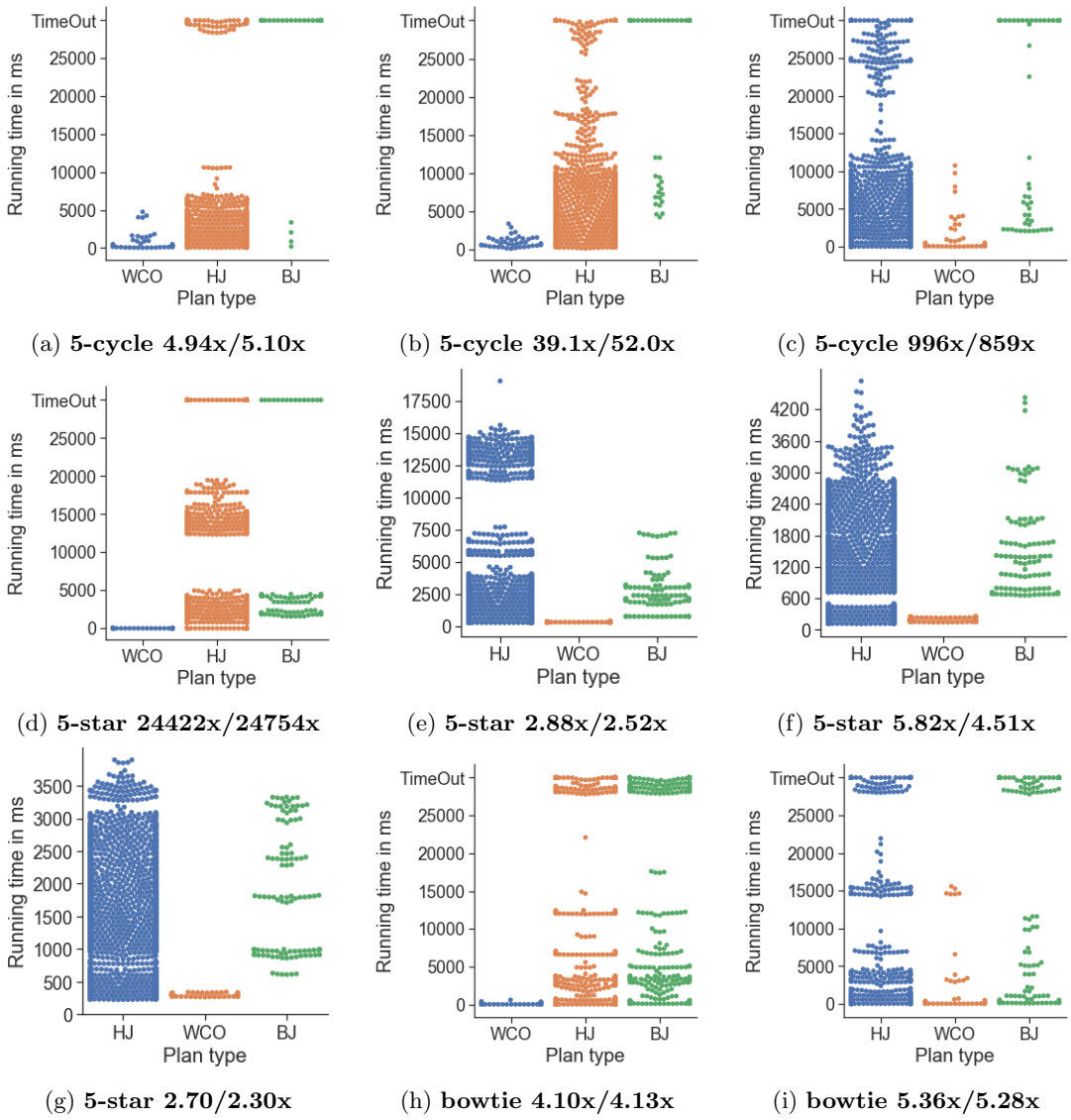


(a) **4-star 7.66x/7.59x** (b) **4-star 59x/58x** (c) **4-star 2.12x/1.86x**

(d) **5-chain 2.30x/2.25x** (e) **5-chain 217x/216x** (f) **5-chain 56.2x/56.0x**

(g) **5-chain 85x/78x** (h) **5-chain 262x/248x** (i) **5-cycle 33.6x/34.4x**

Figure A.5: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

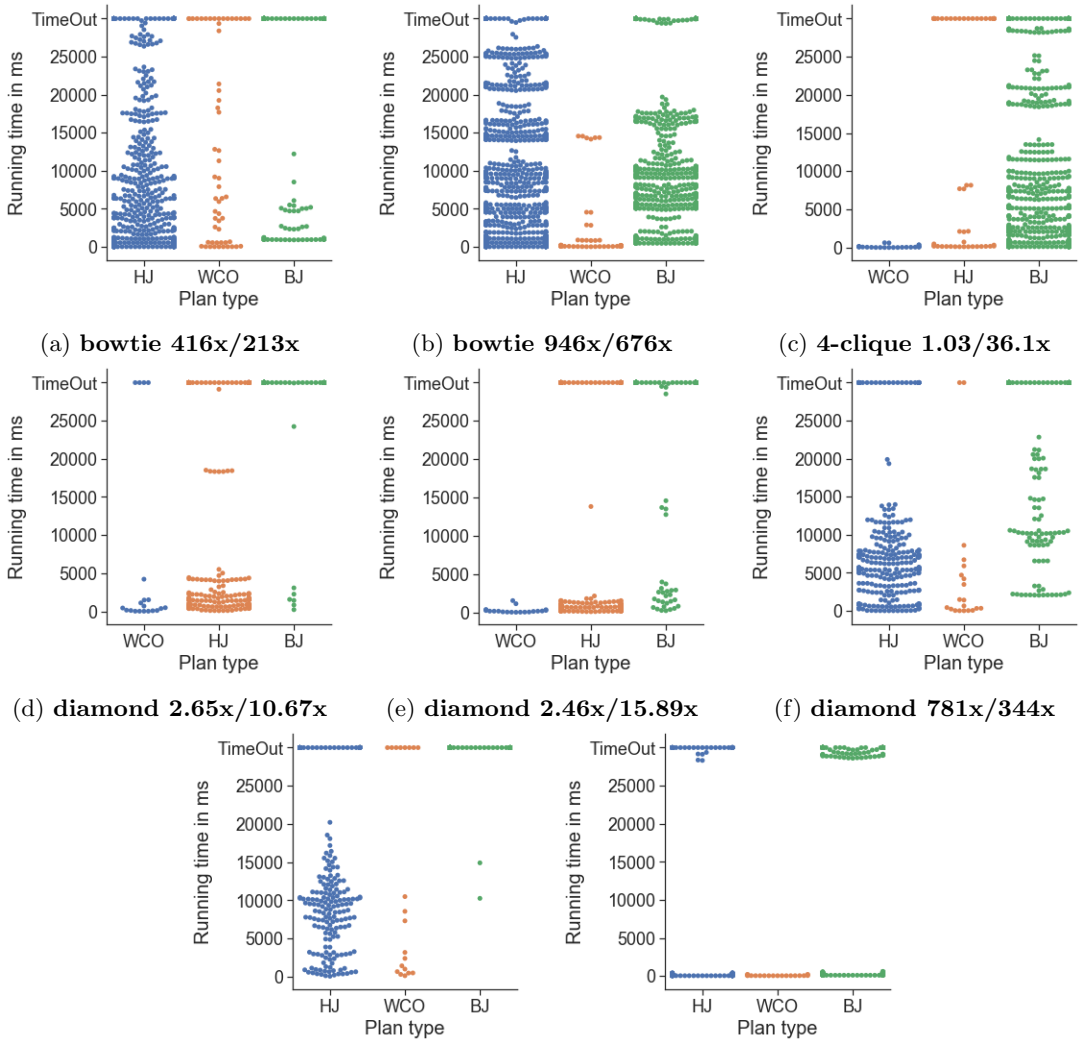


(a) **5-cycle 4.94x/5.10x** (b) **5-cycle 39.1x/52.0x** (c) **5-cycle 996x/859x**

(d) **5-star 24422x/24754x** (e) **5-star 2.88x/2.52x** (f) **5-star 5.82x/4.51x**

(g) **5-star 2.70/2.30x** (h) **bowtie 4.10x/4.13x** (i) **bowtie 5.36x/5.28x**

Figure A.6: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**



(a) **bowtie 416x/213x** (b) **bowtie 946x/676x** (c) **4-clique 1.03/36.1x**

(d) **diamond 2.65x/10.67x** (e) **diamond 2.46x/15.89x** (f) **diamond 781x/344x**

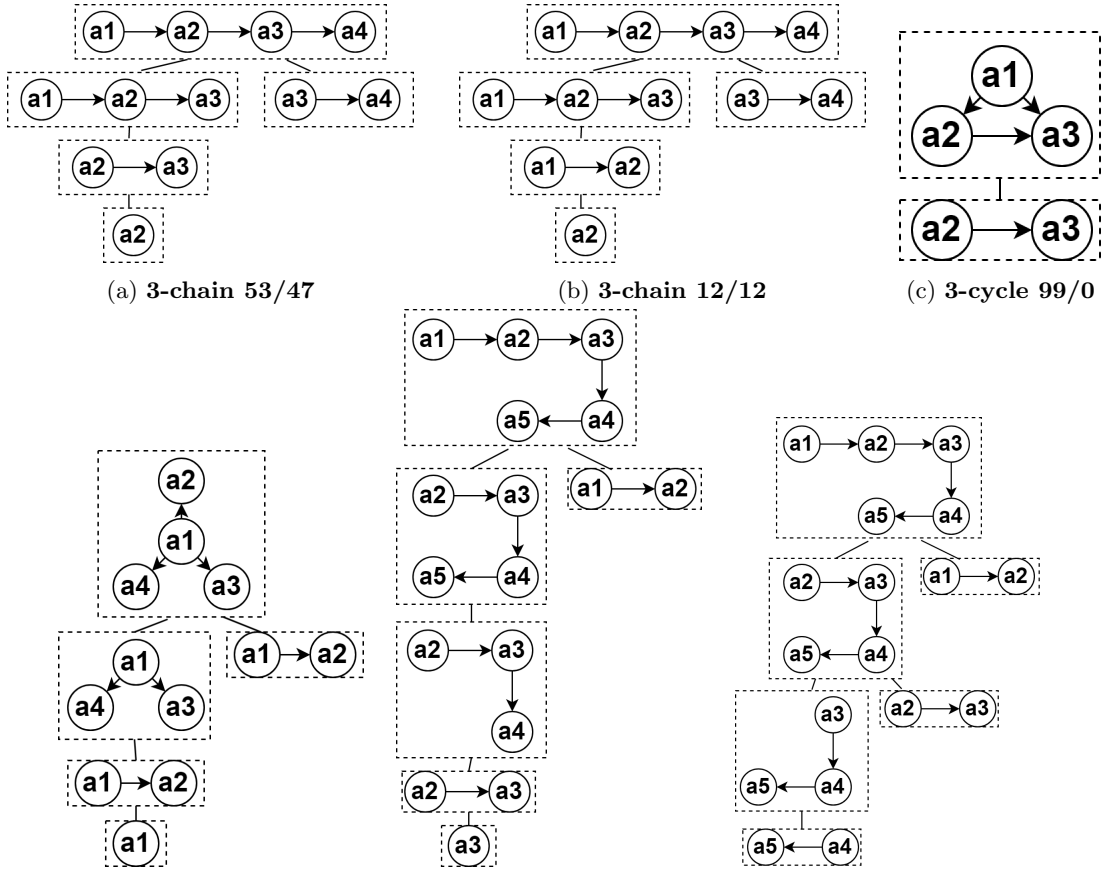
(g) **diamond 158x/75x** (h) **diamond 2.79x/2.67x**

Figure A.7: **Each scatter plot is for one query covering the entire plan space. Each data point is one plan in that plan space. Timeout indicates plans that did not finish running. The order of the plan types on the x-axis indicate which plan type was the fastest, from fastest on the left to slowest on the right. The values in bold indicate the speedup between the fastest plans of different types. The left value compares the hybrid plan to the binary plan, and the right value compares the WCO plan to the binary plan.**

Appendix B

All Fastest Hybrid Plans

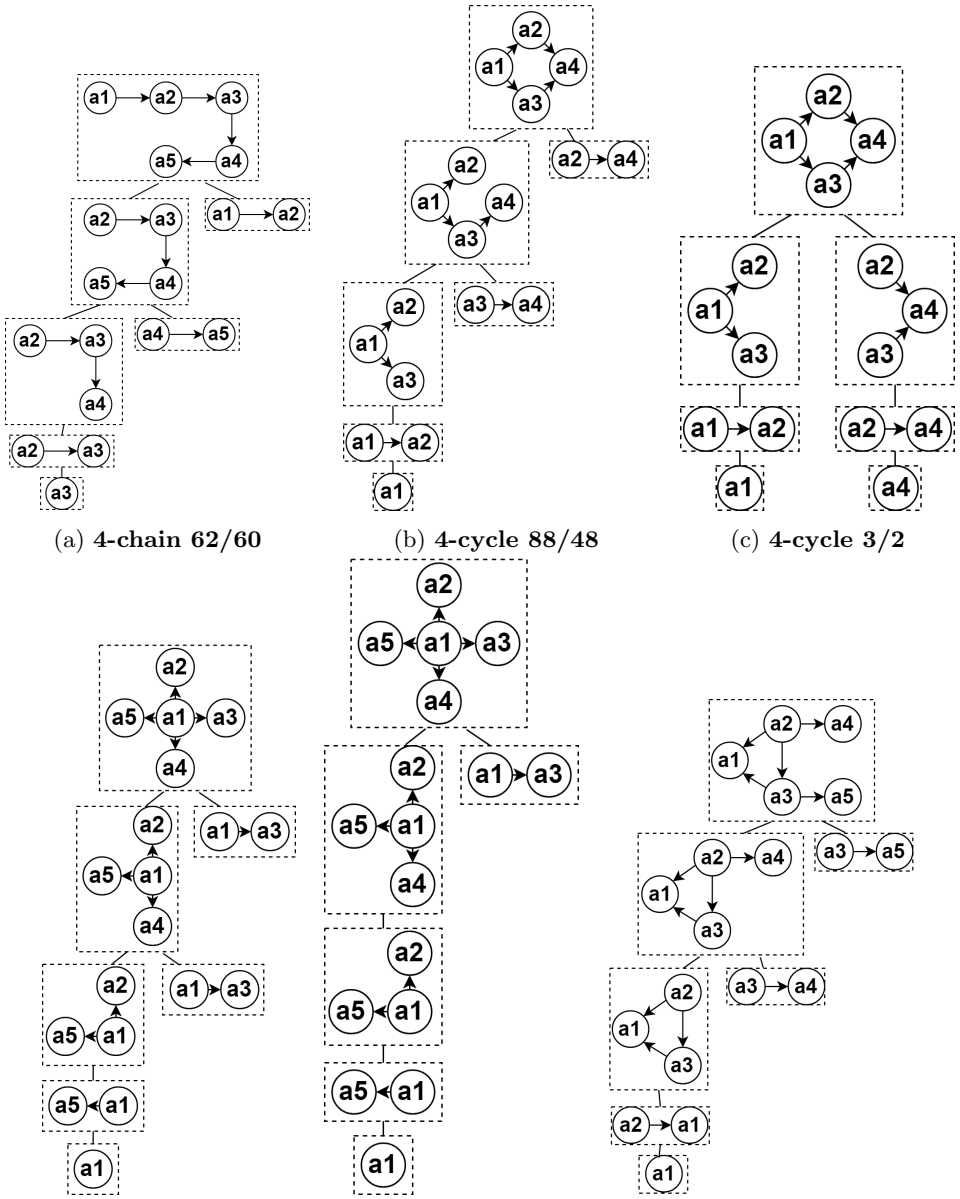
This chapter contains for each query type the hybrid plans that were the fastest hybrid plan for at least one query. We will also indicate the number of times a plan was the fastest hybrid plan. Furthermore, how often it was the fastest plan overall.



(a) **3-chain 53/47** (b) **3-chain 12/12** (c) **3-cycle 99/0**

(d) **3-star 98/83** (e) **4-chain 26/25** (f) **4-chain 1/0**

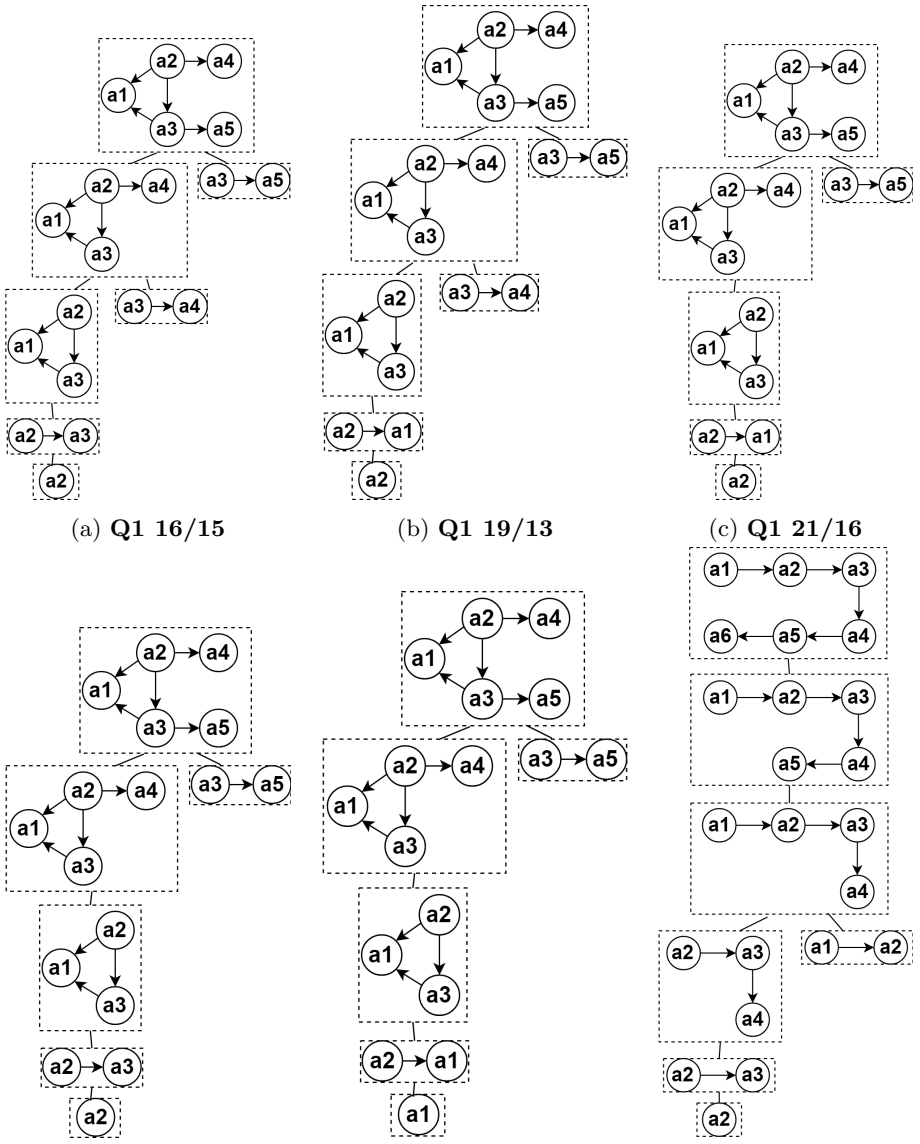
Figure B.1: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **4-chain 62/60** (b) **4-cycle 88/48** (c) **4-cycle 3/2**

(d) **4-star 39/38** (e) **4-star 25/22** (f) **Q1 3/3**

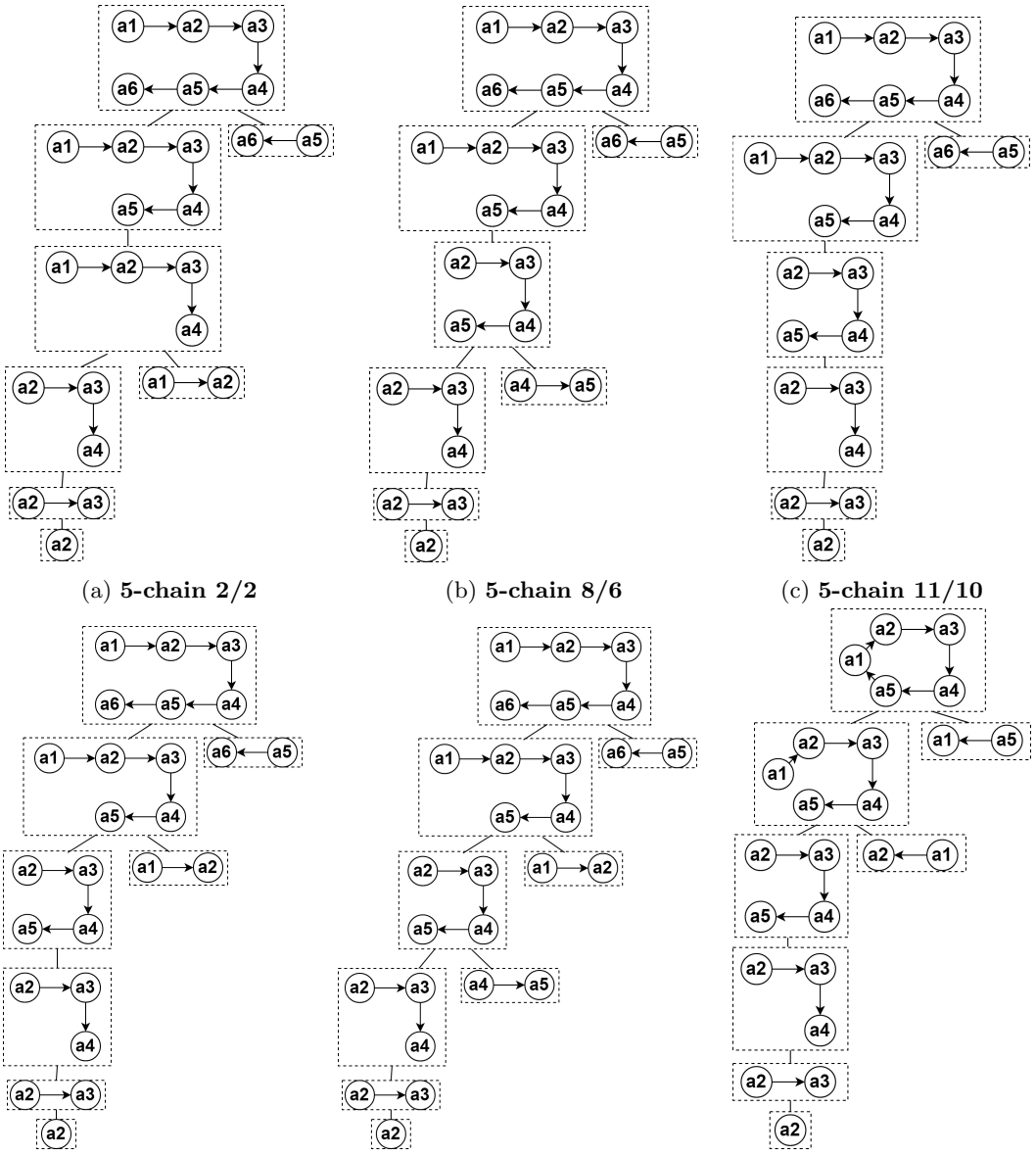
Figure B.2: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **Q1 16/15** (b) **Q1 19/13** (c) **Q1 21/16**

(d) **Q1 33/25** (e) **Q1 2/2** (f) **5-chain 2/2**

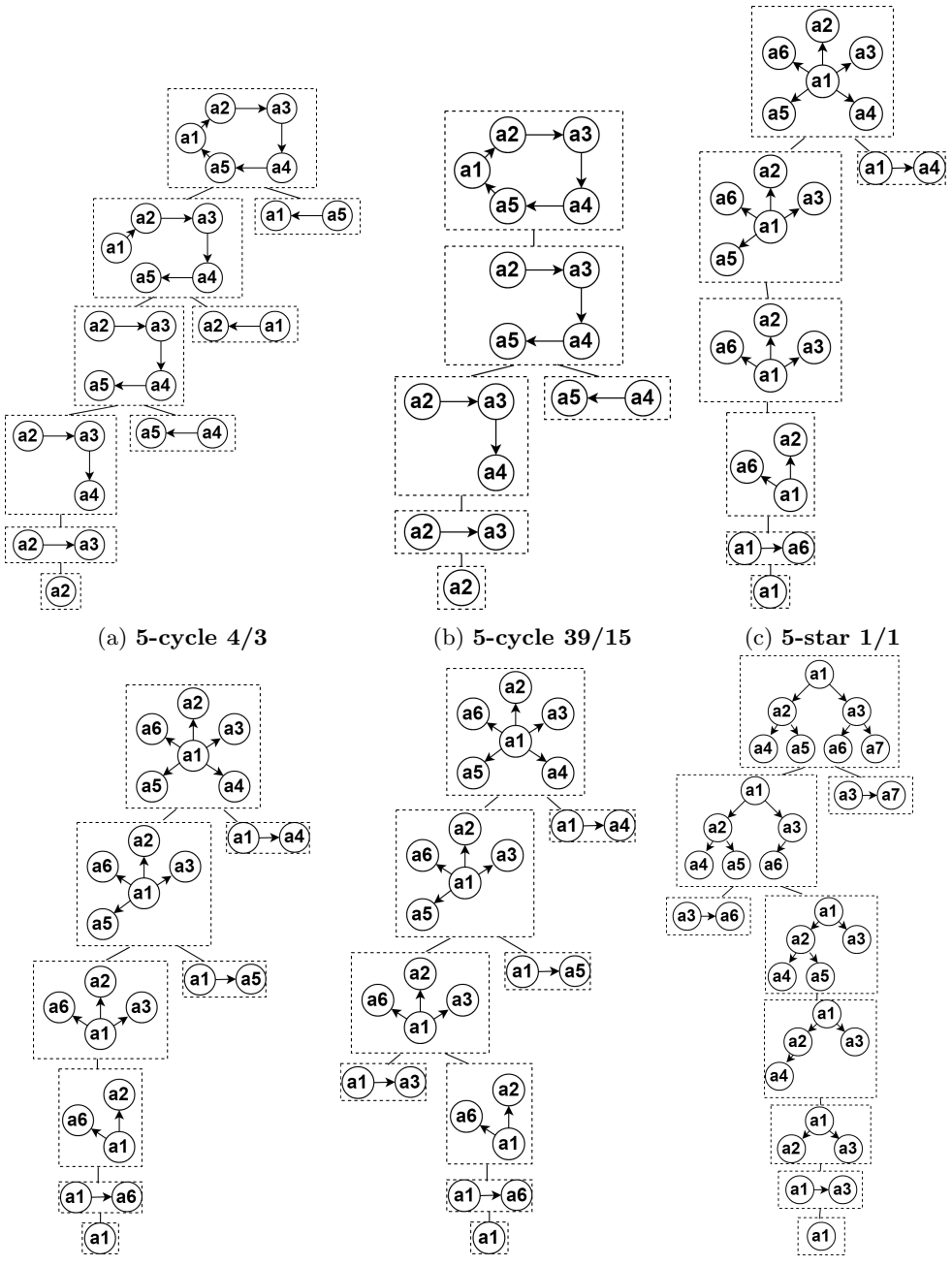
Figure B.3: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **5-chain 2/2** (b) **5-chain 8/6** (c) **5-chain 11/10**

(d) **5-chain 19/19** (e) **5-chain 27/27** (f) **5-cycle 2/0**

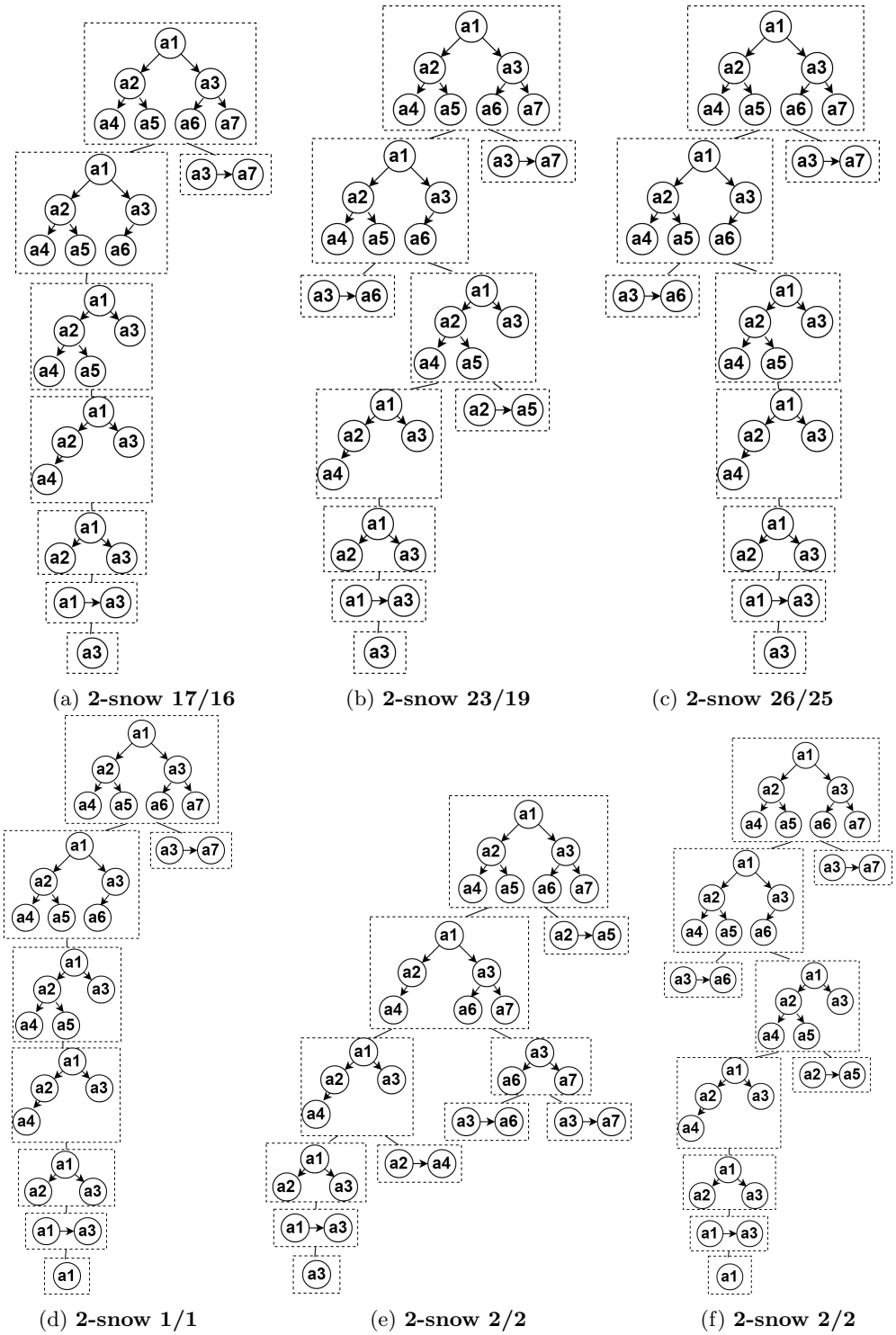
Figure B.4: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **5-cycle 4/3** (b) **5-cycle 39/15** (c) **5-star 1/1**

(d) **5-star 2/2** (e) **5-star 17/17** (f) **2-snow 1/1**

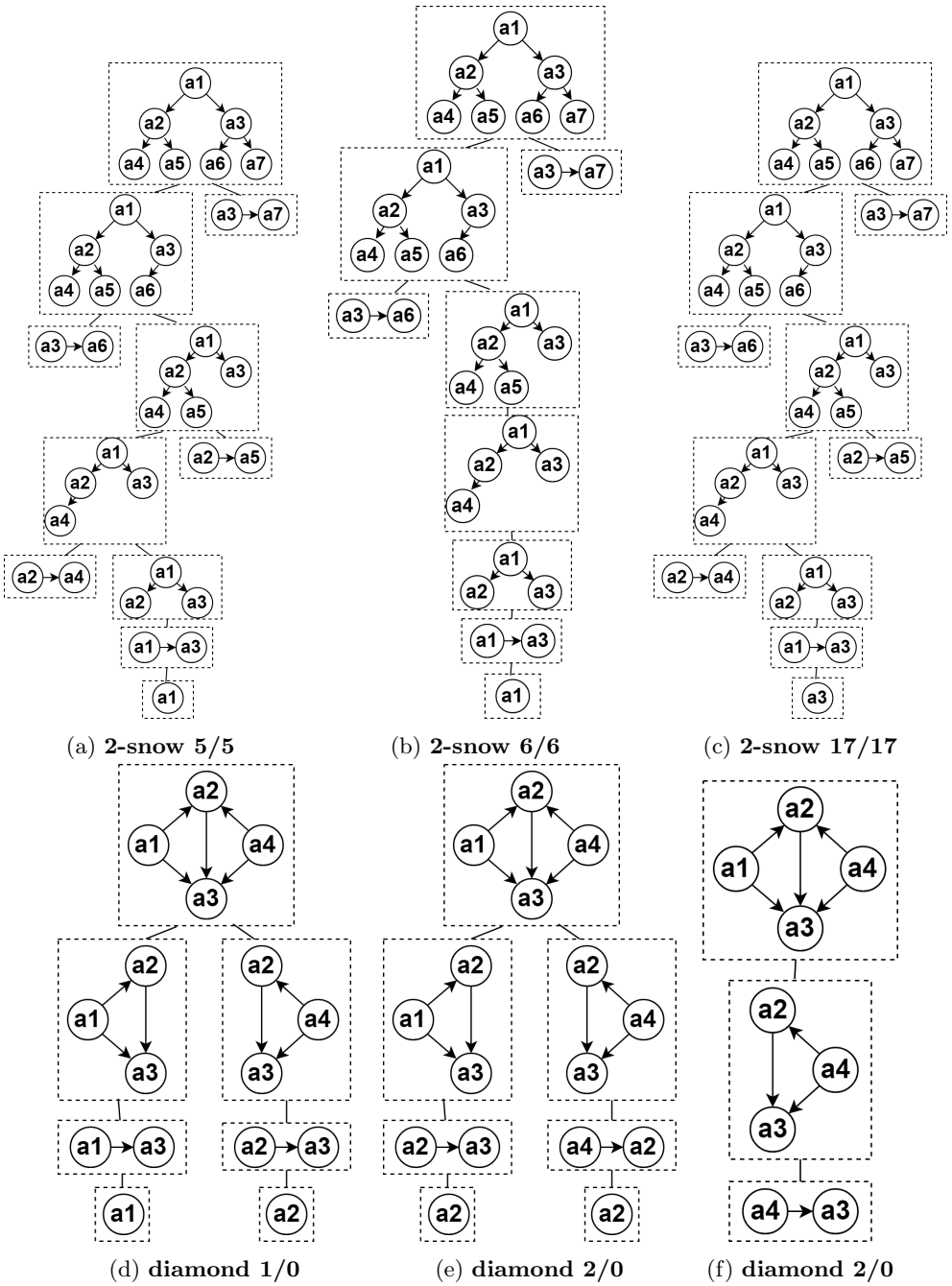
Figure B.5: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **2-snow 17/16** (b) **2-snow 23/19** (c) **2-snow 26/25**

(d) **2-snow 1/1** (e) **2-snow 2/2** (f) **2-snow 2/2**

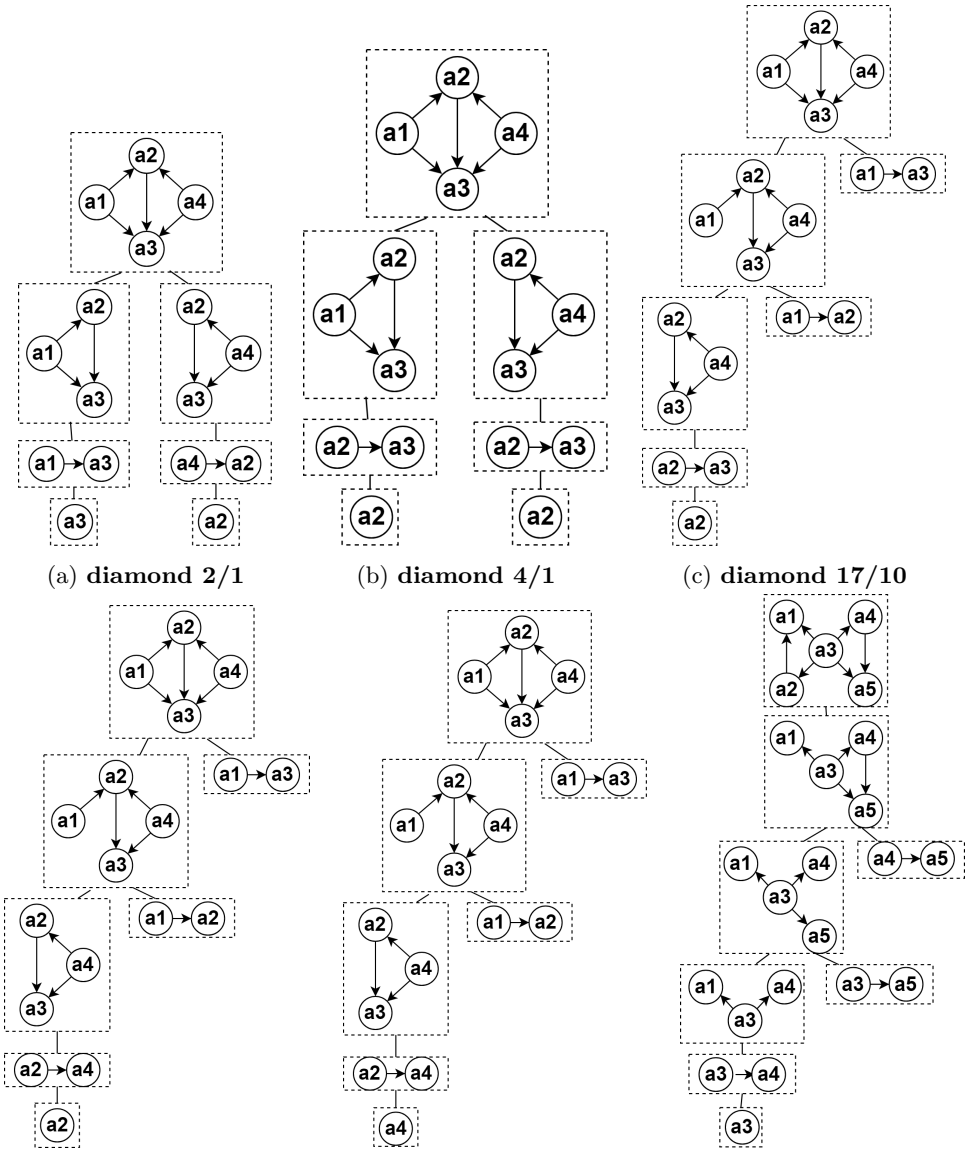
Figure B.6: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **2-snow 5/5** (b) **2-snow 6/6** (c) **2-snow 17/17**

(d) **diamond 1/0** (e) **diamond 2/0** (f) **diamond 2/0**

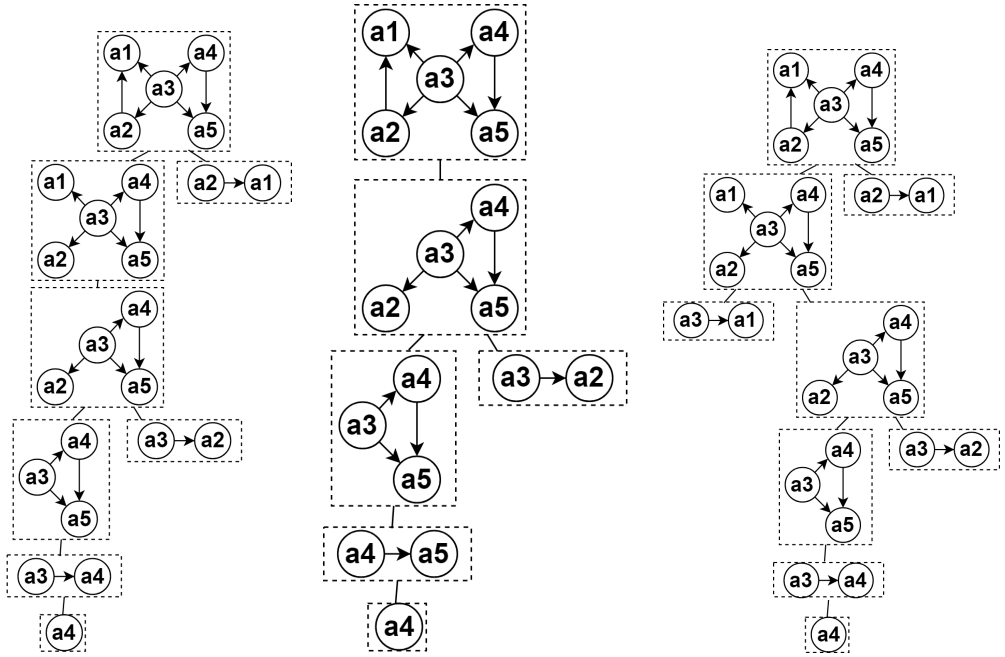
Figure B.7: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



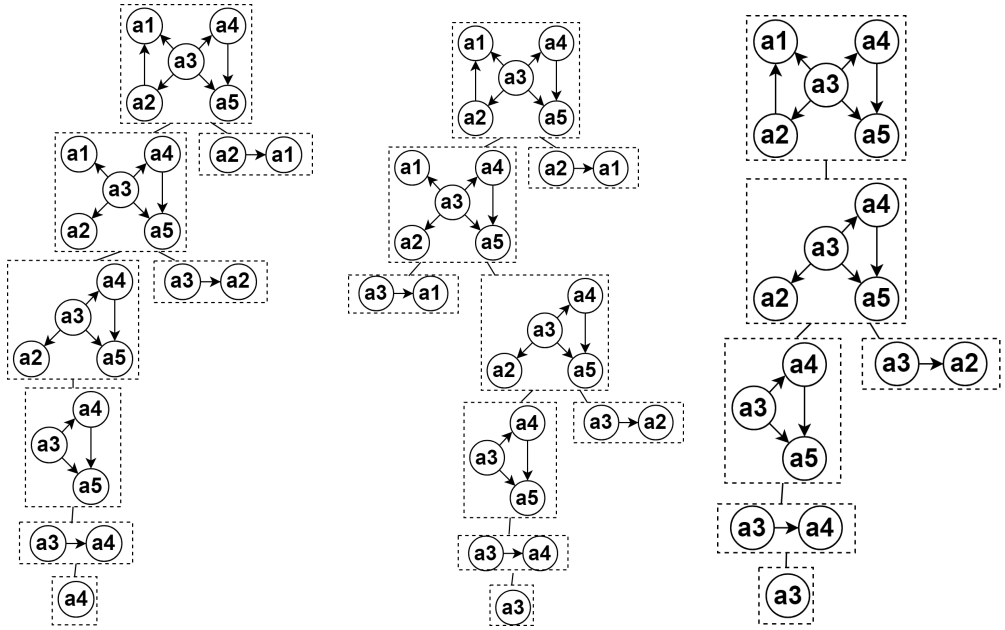
(a) **diamond 2/1** (b) **diamond 4/1** (c) **diamond 17/10**

(d) **diamond 29/16** (e) **diamond 31/20** (f) **bowtie 1/0**

Figure B.8: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**

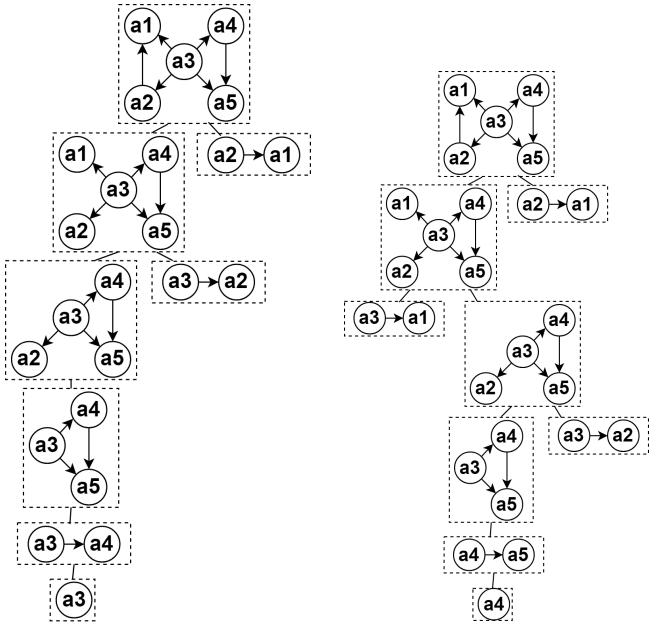


(a) **bowtie 1/0** (b) **bowtie 2/0** (c) **bowtie 2/2**



(d) **bowtie 4/2** (e) **bowtie 6/4** (f) **bowtie 8/2**

Figure B.9: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**



(a) **bowtie 11/9** (b) **bowtie 1/0**

Figure B.10: **Each sub-figure shows the fastest hybrid plans for each query type. The left value indicates the number of times this plan was the fastest hybrid plan for that type. The right value indicates how often this plan was the fastest plan overall for the query type, beating both WCO and binary plans as well.**