

Fully Dynamic Evaluation for Conjunctive Queries with Free Access Patterns



Haozhe Zhang
St Cross College
University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2022

Acknowledgements

Firstly and foremost, I want to express my deep gratitude to Prof. Dan Olteanu, my supervisor, for his guidance, invaluable advice, and the many hours we spent on the project. His vast knowledge and abundant experience have encouraged me throughout my academic research and daily life. Also, I wish to thank my friend, Dr. Ahmet Kara, for his technical assistance and endless discussions regarding the project. Without his support, the thesis would not have been possible. Furthermore, I want to thank my colleague, Dr. Milos Nikolic, for his enormous help during my study.

I am deeply grateful for the support and companionship of my colleagues from the Oxford FDB group and UZH Dast group. It has been a pleasure to have their presence over the past few years.

Lastly, I would especially like to thank my family. My wife, Yaqi, was extremely supportive throughout this entire process. She made numerous sacrifices to help me get to this point. My son, Siran, constantly took me out of the work to take the necessary breaks. My parents, Jun and Xiaopeng, have always been my biggest supporters. They have always encouraged me to pursue my dreams and never gave up on me. My parents-in-law, Hui and Wenxiang, deserve special thanks for their tremendous understanding and support. Without such a team, it would have been impossible for me to complete my studies.

Abstract

We study the problem of answering conjunctive queries with free access patterns under updates. A free access pattern is a partition of the free variables of the query into input and output. The query returns tuples over the output variables given a tuple of values over the input variables.

We introduce a fully dynamic evaluation approach for such queries. It is fully dynamic in the sense that it supports both inserts and deletes of tuples to the input relations. Our approach computes a data structure that supports the enumeration of the output tuples and maintains it under single-tuple updates to the input data. We also give a syntactic characterization of those queries that admit constant time per single-tuple update and whose output tuples can be enumerated with constant delay given an input tuple. Finally, for triangle and hierarchical queries with free access patterns, we chart the complexity trade-offs between the preprocessing time, update time and enumeration delay for such queries. The trade-offs are strongly or weakly Pareto optimal for triangle and a class of hierarchical queries. Their optimality is predicated on the Online Boolean Matrix-Vector Multiplication conjecture.

Contents

1	Introduction	1
1.1	Problem Setting	2
1.2	Contributions	3
1.3	Organization	5
2	Preliminaries	6
2.1	Data Model and Query Language	6
2.2	Conjunctive Queries with Free Access Patterns	9
2.3	Query Classes	9
2.4	Variable Orders	10
2.4.1	Variable Orders and Tree Decompositions	10
2.4.2	Classes of Variable Orders	12
2.5	Width Measures	13
2.6	View Trees	15
2.7	Computational Model	16
2.8	Lower Bounds	17
3	Overview of the Main Results	18
3.1	Fully Dynamic Evaluation for CQAP Queries	18
3.1.1	Query Fractures	18
3.1.2	Complexities	20
3.2	Preprocessing-Update-Enumeration Trade-offs	21
3.2.1	Queries with Hierarchical Fractures	21
3.2.2	Triangle Queries	23
3.3	A Dichotomy Result	24
3.4	Current Landscape of Conjunctive Query Evaluation	25

4	The Case of General Queries	26
4.1	Preprocessing	26
4.1.1	Extended Variable Orders	26
4.1.2	View Tree Construction	27
4.1.3	Indicator Projections	29
4.2	Enumeration	33
4.2.1	View Iterators	33
4.2.2	Enumeration Procedures	34
4.2.3	Multiple View Trees	36
4.3	Update	37
4.3.1	Dynamic Width	38
4.4	Complexity Analysis	41
5	Trade-Offs in Dynamic Evaluation for CQAP Queries with Hierarchical Fractures	45
5.1	Data Partitioning	46
5.2	Preprocessing	46
5.2.1	From Canonical to Access-Top Variable Orders	47
5.2.2	Variable Orders Describing Evaluation Strategies	49
5.2.3	View Trees Encoding the Query Result	54
5.2.4	Proofs	56
5.3	Enumeration	65
5.3.1	Union View Iterators	66
5.3.2	Enumeration Procedure	69
5.3.3	Enumeration from View Trees	72
5.3.4	Enumeration Delay	72
5.4	Update	73
5.4.1	Determining the Relation Part for a Single-Tuple Update	74
5.4.2	Processing a Single-Tuple Update	76
5.4.3	Processing a Sequence of Single-Tuple Updates	79
5.5	Complexity Analysis	87
5.6	Optimality Result	92
6	Trade-Offs in Dynamic Evaluation for Triangle CQAP Queries	95
6.1	The C_3 Query	95
6.1.1	Preprocessing	95
6.1.2	Enumeration	98
6.1.3	Update	99

6.2	Queries in \mathbf{C}_2	101
6.2.1	Preprocessing	101
6.2.2	Enumeration	102
6.2.2.1	Hop View Iterators	104
6.2.3	Update	108
6.3	Queries in \mathbf{C}_1	108
6.3.1	Preprocessing	108
6.3.2	Enumeration	110
6.3.3	Update	111
6.4	The \mathbf{C}_{count} Query	111
6.5	The \mathbf{C}_{lookup} Query	113
6.6	Complexity Analysis	113
6.7	Optimality Result	114
7	Dichotomy Result	120
8	Related Work	126
9	Extensions	131
10	Conclusion and Future Work	134
	Bibliography	134

Chapter 1

Introduction

Conjunctive queries play a significant role in relational databases. Many data management tasks can be expressed as the evaluation of conjunctive queries, i.e., computing the results of the queries. This has received a fair amount of attention [74, 65, 58]. For many other tasks, it is also required to support repeated access to the results of conjunctive queries for particular *access patterns* [56, 57, 52]. For instance, flight booking companies may need to build interfaces allowing users to retrieve flights from their database by entering values for specific input fields, such as the departure and arrival cities.

Example 1.1. *Consider the database of a flight booking company with the two relations: `airports(AirportId, Name, City)`, which stores each airport's ID, name and the city where it is located, and `flights(DEPAirportId, ARRAirportId, FlightNo)`, which stores each flight's number and the IDs of its departure and destination airports.*

The company allows users to check the flights for specific departure and arrival cities. This task is captured by the following SQL query with parameters `%A1` and `%A2`:

```
SELECT FlightNo, A1.City, A2.City
FROM airports AS A1, airports AS A2, flights
WHERE A1.AirportId = flights.DEPAirportId AND
A2.AirportId = flights.ARRAirportId AND
A1.City = "%A1" AND A2.City = "%A2"
```

The parameters `%A1` and `%A2` are replaced by the departure and arrival cities entered by the users. For instance, suppose a user wants to check the flights from London to Zurich, the company needs to answer this query with `%A1` and `%A2` replaced by London and Zurich. This query may be asked by many other users with different values for `%A1` and `%A2`.

We formalize such restricted access by conjunctive queries with output access patterns (CQAP for short) [26]: the free variables of a CQAP query are partitioned into *input* and *output*. Given a tuple of values over the input variables, the query yields the tuples of

values over the output variables such that the conjunction expressed in the query holds. We call such querying an *access request*. In Example 1.1, the variable `FlightNo` is the output variable and the variables `A1.City` and `A2.City` are the input variables.

Another problem that can be cast as a CQAP query is the triangle detection analysis in social networks. Consider a social network database with the binary relation `Friend(user1, user2)` such that (X, Y) is in `Friend` if users `X` and `Y` are friends. Suppose the analysis requires that given any two users `X` and `Y`, listing the users that form triangles with `X` and `Y` in the dataset. This can be captured by the following SQL query with the parameters `%X` and `%Y`:

```
SELECT R1.user1, R2.user1, R3.user1
FROM Friend AS R1, Friend AS R2, Friend AS R3
WHERE R1.user2 = R2.user1 AND R2.user2 = R3.user1 AND R3.user2 = R1.user1
AND R1.user1 = %X AND R2.user1 = %Y
```

The variables `R1.user1` and `R2.user1` are the input variables, while `R3.user1` is the output variable.

So far, we have focused on the query evaluation on static databases. In real world, however, databases are subject to frequent changes, e.g., flights can be canceled, and new users can be added. Many applications depend on real-time analytics over the evolving data. This requires dynamic evaluation for queries, i.e., maintaining the results of queries under data updates. Previous works have focused on the dynamic evaluation for conjunctive queries [49, 23, 50, 14, 38, 41]. There is, however, no prior work considered the dynamic evaluation for queries with access patterns.

1.1 Problem Setting

In this thesis, we consider the problem of fully dynamic evaluation for CQAP queries. That is, answering access requests for the CQAP queries under single-tuple updates. It is fully dynamic in the sense that it supports both inserts and deletes of tuples to the input relations. We tackle the problem using the typical incremental view maintenance (IVM) framework: we precompute a data structure that supports answering any access requests and maintain it under updates.

We refine the overall computation time of dynamic evaluation for CQAP queries into three components. The *preprocessing time* is the time to compute the data structure before receiving any updates. Given a tuple over the input variables, the *enumeration delay* is the time between the start of the enumeration process and the output of the first tuple, the time between outputting any two consecutive tuples, and the time between

outputting the last tuple and the end of the enumeration process [30]. The *update time* is the time used to update the data structure for one single-tuple update. Such update time can be amortized over a sequence of single-tuple updates; the *amortized update time* is the average time taken by the sequence of updates.

Following this framework, prior work can be used as part of two simple approaches, namely the eager approach and the lazy approach. The eager approach precomputes the result of the CQAP query over both the input and output variables and creates index on them for the access pattern. Upon each update, we update the result and the index using existing IVM approaches, such as the classical first order IVM [23]. With this approach, access requests can be answered efficiently, but the data structure needs to be updated for each update.

The lazy approach, on the other hand, has no precomputation and only updates the base relations. Upon each access request, it uses existing enumeration algorithm, such as the work by Bagan et al. [9], for the residual query obtained by setting the input variables to constants in the original query. This approach does not need to update any data structure, but needs time-consuming preparation for each access request, e.g., to remove dangling tuples and possibly create a data structure to support enumeration.

1.2 Contributions

We have two insights on improving the eager and lazy approaches. The first is to avoid fully materialized query results. For a CQAP query, the eager approach materializes and maintains the query result over both input and output variables. Such materialized result can easily support any access patterns over these free variables (not just the access pattern of the query), but this is not necessary. We can maintain a data structure that is specialized for the access pattern of the query. Such data structure can be more succinct than the fully materialized query results and require less preprocessing and update time.

Following this idea, we developed a fully dynamic evaluation approach for CQAP queries. It is fully dynamic in the sense that it supports both inserts and deletes of tuples to the input relations. Our approach computes and maintains a data structure that is specialized for the access pattern of the query to support the enumeration of the output tuples. The construction of such data structure is based on the two new notions *access-top variable orders* and *query fractures*. They capture the nature of free access patterns of CQAP queries.

An access-top variable order is a decomposition of the query into a rooted forest of variables, where: the input variables are above all other variables; and the free (input

and output) variables are above the bound variables. This variable order is compiled into a tree of views, which is a data structure that compactly represents the query output.

Since access to the query output requires fixing values for the input variables, the query can be fractured by breaking its joins on the input variables and replacing each of their occurrences with fresh variables within each connected component of the query hypergraph. This does not violate the access pattern, since each fresh input variable can be set to the corresponding given input value. Yet this may lead to structurally simpler queries whose dynamic evaluation admits lower complexity.

The second insight is to exploit the trade-off between the update time and enumeration delay. The lazy and the eager approaches are on the two extremes: the lazy approach admits constant update time but high enumeration delay, while the eager approach admits the opposite. Consider a sequence of interleaved updates and access requests to a database. In terms of the overall computation time, i.e., the overall time spent on processing the updates and answering the access requests, the lazy approach works well when updates occur much more frequently than access requests, and the eager approach works well in the opposite scenario. Assume we have an approach that shows a full continuum between the two extremes, we can tune the update time and enumeration delay depending on the frequencies of updates and access requests to achieve a better overall computation time.

Following this idea, we developed such an approach which charts the preprocessing time - update time - enumeration delay trade-off for the dynamic evaluation of two class of CQAP queries: (1) the triangle queries and (2) queries whose fractures are hierarchical. It shows that as the preprocessing and update times increase, the enumeration delay decreases. The core idea of the approach is to partition the data into the heavy and lights parts based on the degrees of the values and apply adaptive strategies to them. By tuning the threshold of the heavy-light partitioning, our approach can reduce the enumeration delay by increasing the update time, or the other way around. Our approach is strongly and weakly Pareto optimal for all triangle CQAP queries and a particular class hierarchical query, called CQAP_1 .

A further contribution of the thesis is a syntactic characterization of those CQAP queries that admit linear-time preprocessing and constant-time update and enumeration delay. We called this class CQAP_0 . All queries outside CQAP_0 do not admit constant-time update and delay regardless of the preprocessing time, unless a widely held (namely, the Online Matrix-Vector) conjecture fails. This dichotomy generalizes the similar dichotomy for q -hierarchical queries *without access patterns* [14], which are in CQAP_0 and have all free variables as output. The class CQAP_0 further contains queries with input variables that may be not only non- q -hierarchical but also cyclic. For instance, the

edge triangles detection problem is in CQAP_0 : given an edge (u, v) , check whether it participates in a triangle. The smallest query patterns not in CQAP_0 subsume the non- q -hierarchical ones as the former are sensitive to the interplay of the output and input variables. Showing they do not admit constant-time update and delay requires different and more hardness reductions from the Online Matrix-Vector Multiplication problem.

This thesis is based on our published works [41], [42], [43], and [44]. The first two works [41] and [42] tackle the dynamic evaluation of the triangle queries with arbitrary free variables. The work [43] focuses on both the static and dynamic evaluation of the hierarchical queries with arbitrary free variables. We developed algorithms that reveal the trade-offs between the preprocessing time, update time and enumeration delay by applying adaptive strategies to the data with different degrees. In the work [44], we extend our results to queries with access patterns.

In summary, these are contributions of this thesis:

- a fully dynamic evaluation approach for arbitrary CQAP queries,
- a fully dynamic evaluation approach which uncovers the trade-offs between the preprocessing time, update time and enumeration delay for triangle CQAP queries and CQAP queries whose fractures are hierarchical, and
- the dichotomy for CQAP queries that admits linear-time preprocessing and constant-time update and enumeration delay.

1.3 Organization

This thesis is structured as follows. Chapter 2 introduces the notations, definitions and properties that are necessary to understand the thesis. Chapter 3 gives a brief overview of the three main results of the thesis. These results are presented in detail in the following four chapters: Chapter 4 introduces our dynamic evaluation approach for arbitrary CQAP queries. Chapters 5 and 6 introduce our dynamic evaluation approach that uncovers the trade-offs between the preprocessing time, update time and enumeration delay for CQAP queries whose fractures are hierarchical and triangle CQAP queries, respectively. Chapter 7 proves the dichotomy result. Finally, Chapter 8 overviews the related work and the current landscape for the evaluation of CQAP queries. Chapter 9 lists the possible extensions of the thesis. At the end, Chapter 10 concludes the thesis and discusses additional challenges for further work.

Chapter 2

Preliminaries

In this chapter, we introduce the notations, definitions and properties that are used in this thesis.

2.1 Data Model and Query Language

We first define the data model and query language.

Data Model. A schema $\mathcal{X} = (X_1, \dots, X_n)$ is a tuple of distinct variables. Each variable X_i has a discrete domain $\text{Dom}(X_i)$. We treat schemas and sets of variables interchangeably, assuming a fixed ordering of variables. A tuple \mathbf{x} of values has schema $\mathcal{X} = \text{Sch}(\mathbf{x})$ and is an element from $\text{Dom}(\mathcal{X}) = \text{Dom}(X_1) \times \dots \times \text{Dom}(X_n)$. We use uppercase letters for variables and lowercase letters for data values. Likewise, we use bold uppercase letters for schemas and bold lowercase letters for tuples of data values.

We express relations as factors over the *sum-product semiring* $(\mathbb{Z}, +, \times, 0, 1)$ of integers [1]. A relation R over schema \mathcal{X} is a function $R : \text{Dom}(\mathcal{X}) \rightarrow \mathbb{Z}$ mapping tuples over \mathcal{X} to integers, such that $R(\mathbf{x}) \neq 0$ for finitely many tuples \mathbf{x} . A tuple \mathbf{x} is in R , denoted by $\mathbf{x} \in R$, if $R(\mathbf{x}) \neq 0$. The value $R(\mathbf{x})$ represents the multiplicity of \mathbf{x} in R . The size $|R|$ of R is the size of the set $\{\mathbf{x} \mid \mathbf{x} \in R\}$. A database is a set of relations and has size given by the sum of the sizes of its relations. This data model generalizes the bag semantics.

Given a tuple \mathbf{x} over schema \mathcal{X} and $\mathcal{S} \subseteq \mathcal{X}$, $\mathbf{x}[\mathcal{S}]$ is the restriction of \mathbf{x} onto \mathcal{S} . Given two tuples \mathbf{x} and \mathbf{x}' over schemas \mathcal{X} and respectively \mathcal{X}' such that $\mathcal{X} \cap \mathcal{X}' = \emptyset$, $\mathbf{x} \circ \mathbf{x}'$ is a tuple over the schema $\mathcal{X} \cup \mathcal{X}'$ that concatenates the values in \mathbf{x} and \mathbf{x}' . For a relation R over schema \mathcal{X} , schema $\mathcal{S} \subseteq \mathcal{X}$, and tuple $\mathbf{t} \in \text{Dom}(\mathcal{S})$: $\sigma_{\mathcal{S}=\mathbf{t}}R = \{\mathbf{x} \mid \mathbf{x} \in R \wedge \mathbf{x}[\mathcal{S}] = \mathbf{t}\}$ is the set of tuples in R that agree with \mathbf{t} on the variables in \mathcal{S} ; $\pi_{\mathcal{S}}R = \{\mathbf{x}[\mathcal{S}] \mid \mathbf{x} \in R\}$ stands for the set of tuples in R projected onto \mathcal{S} , i.e., the set of distinct \mathcal{S} -values from the tuples in R with non-zero multiplicities.

Query Language. A conjunctive query (CQ) has the form

$$Q(\mathcal{F}) = R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n).$$

For each $i \in [n]$, R_i is a relation symbol, and $R_i(\mathcal{X}_i)$ is an atom. Moreover, we denote by $\text{vars}(Q) = \bigcup_{i \in [n]} \mathcal{X}_i$ the set of variables; $\text{free}(Q) = \mathcal{F}$ the set of free variables; $\text{atoms}(Q) = \{R_i(\mathcal{X}_i) \mid i \in [n]\}$ the set of the atoms; and $\text{atoms}(X)$ the set of the atoms containing variable X . Let $\mathcal{B} = \text{vars}(Q) \setminus \text{free}(Q)$ be the set of bound variables of Q . The CQ Q is evaluated over the semiring $(\mathbb{Z}, +, \times, 0, 1)$. Given a tuple \mathbf{x} over \mathcal{F} , the multiplicity of \mathbf{x} in the result of Q is computed by:

$$Q(\mathbf{x}) = \sum_{\mathbf{x}' \in \text{Dom}(\mathcal{B})} R_1((\mathbf{x} \circ \mathbf{x}')[\mathcal{X}_1]) \times \dots \times R_n((\mathbf{x} \circ \mathbf{x}')[\mathcal{X}_n]).$$

Example 2.1. Consider the relations $R(A, B)$ and $S(B, C)$ over semiring $(\mathbb{Z}, +, \times, 0, 1)$:

$$\begin{array}{l} \frac{A \ B \ \rightarrow \ R(A, B)}{a_1 \ b_1 \ \rightarrow \ 1} \\ a_1 \ b_2 \ \rightarrow \ 2 \end{array} \qquad \begin{array}{l} \frac{B \ C \ \rightarrow \ S(B, C)}{b_1 \ c_1 \ \rightarrow \ 2} \\ b_2 \ c_1 \ \rightarrow \ 2 \end{array}$$

The two tables show the tuples in R and S with non-zero multiplicities. They represent that (a_1, b_1) appears once and (a_1, b_2) appears twice in R , and (b_1, c_1) and (b_2, c_1) appear twice in S .

Consider the query $Q(A, C) = R(A, B), S(B, C)$. The multiplicity of a tuple (a, c) over the free variables A and C in the query result is computed by:

$$Q(a, c) = \sum_{b \in \text{Dom}(B)} R(a, b) \times S(b, c).$$

The query result is shown in the table below:

$$\frac{A \ C \ \rightarrow \ Q(A, C)}{a_1 \ c_1 \ \rightarrow \ R(a_1, b_1) \times S(b_1, c_1) + R(a_1, b_2) \times S(b_2, c_1) = 6}$$

It represents that (a_1, c_1) appears six times in the query result. □

Updates and Delta Queries. An update δR to a relation R is a relation over the schema of R . A single-tuple update, written as $\delta R = \{\mathbf{x} \rightarrow m\}$, maps the tuple \mathbf{x} to a non-zero multiplicity $m \in \mathbb{Z}$ and other tuple to 0; that is, $|\delta R| = 1$. The data model and query language make no distinction between inserts and deletes – these are both updates represented as relations in which tuples have positive and negative multiplicities,

respectively. This allows us to preprocess inserts and deletes uniformly, which is unknown how to do this in bag semantics.

Allowing negative multiplicities can be useful in some applications, such as modelling a matrix multiplication as a join of relations, as discussed in Section 6.7. Though, if we want to simulate bag semantics, we can force the multiplicity of each tuple in the update to be non-negative: when there is a delete request to delete a tuple that is not in the database, we can ignore the request.

Given a relation R over the schema \mathcal{X} and an update δR , applying δR to R means summing the update with the relation, which results in a new relation $R' = R + \delta R$ over the schema \mathcal{X} such that:

$$R'(\mathbf{x}) = R(\mathbf{x}) + \delta R(\mathbf{x}), \quad \text{where } \mathbf{x} \in \text{Dom}(\mathcal{X}).$$

Given a query Q and an update δR to a relation R in Q , the *delta query* δQ defines the change in the query result after applying δR to the database. For a CQ $Q(\mathcal{F}) = R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n)$, the delta query δQ for an update δR_1 to R_1 is computed by replacing R_1 with δR_1 in Q :

$$\delta Q(\mathcal{F}) = \delta R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n).$$

The query result Q' after applying δR_1 to the database is computed by summing the query result Q with the delta query δQ . That is, $Q' = Q + \delta Q$.

Example 2.2. Consider the relations and the query $Q(A, C) = R(A, B), S(B, C)$ in Example 2.1. Given an update $\delta R = \{(a_1, b_2) \rightarrow 3\}$, which represents inserting three (a_1, b_2) to R , the delta query δQ is computed as follows:

$$\frac{A \ C \rightarrow \delta Q(A, C)}{a_1 \ c_1 \rightarrow \delta R(a_1, b_2) \times S(b_2, c_1) = 6}$$

The database after applying the update δR is as follows:

$$\begin{array}{l|l} A \ B \rightarrow R'(A, B) & B \ C \rightarrow S(B, C) \\ \hline a_1 \ b_1 \rightarrow 1 & b_1 \ c_1 \rightarrow 2 \\ a_1 \ b_2 \rightarrow R(a_1, b_2) + \delta R(a_1, b_2) = 5 & b_2 \ c_1 \rightarrow 2 \end{array}$$

$$\frac{A \ C \rightarrow Q'(A, C)}{a_1 \ c_1 \rightarrow Q(a_1, c_1) + \delta Q(a_1, c_1) = 12}$$

2.2 Conjunctive Queries with Free Access Patterns

A *conjunctive query with free access patterns* (CQAP for short) has the form

$$Q(\mathcal{O}|\mathcal{I}) = R_1(\mathcal{X}_1), \dots, R_n(\mathcal{X}_n).$$

The free variables are partitioned into *input* variables \mathcal{I} and *output* variables \mathcal{O} . An empty set of input or output variables is denoted by a dot (\cdot). The CQs are CQAP queries with $\mathcal{I} = \emptyset$. Given a database \mathcal{D} and a tuple \mathbf{i} over \mathcal{I} , the output of Q for the input tuple \mathbf{i} is denoted by $Q(\mathcal{O}|\mathbf{i})$ and is defined by $\pi_{\mathcal{O}}\sigma_{\mathcal{I}=\mathbf{i}}Q(\mathcal{D})$: This is the set of tuples \mathbf{o} over \mathcal{O} such that the assignment $\mathbf{i} \circ \mathbf{o}$ to the free variables satisfies the body of Q .

2.3 Query Classes

We next introduce the classes of queries that we consider in this thesis.

Definition 2.3 (Triangle CQAP Queries). *The triangle CQAP queries have the form:*

$$Q(\mathcal{O} | \mathcal{I}) = R(A, B), S(B, C), T(C, A), \quad \text{where } \mathcal{O} \cup \mathcal{I} \subseteq \{A, B, C\}.$$

Example 2.4. *Consider a triangle query $Q(\mathcal{O} | \mathcal{I}) = R(A, B), S(B, C), T(C, A)$.*

- *When $\mathcal{O} = \emptyset$ and $\mathcal{I} = \emptyset$, i.e., $Q(\cdot | \cdot)$, the query result is computed by:*

$$Q(\cdot | \cdot) = \sum_{a \in \text{Dom}(A)} \sum_{b \in \text{Dom}(B)} \sum_{c \in \text{Dom}(C)} R(a, b) \times S(b, c) \times T(c, a),$$

which is the number of triangles in the database.

- *When $\mathcal{O} = \{A, B, C\}$ and $\mathcal{I} = \emptyset$, i.e., $Q(A, B, C | \cdot)$, the query is to list the triangles and their multiplicities in the database.*
- *When $\mathcal{O} = \emptyset$ and $\mathcal{I} = \{A, B, C\}$, i.e., $Q(\cdot | A, B, C)$, the query is to check the multiplicity of a given triangle in the database.*

Definition 2.5 (Hierarchical Queries). *A query Q is hierarchical if for any two variables $A, B \in \text{vars}(Q)$, either $\text{atoms}(A) \subseteq \text{atoms}(B)$, $\text{atoms}(B) \subseteq \text{atoms}(A)$, or $\text{atoms}(B) \cap \text{atoms}(A) = \emptyset$.*

Example 2.6. *The query $Q() = R(A, B), S(B, C)$ is hierarchical, since $\text{atoms}(A) \subseteq \text{atoms}(B)$, $\text{atoms}(C) \subseteq \text{atoms}(B)$ and $\text{atoms}(A) \cap \text{atoms}(C) = \emptyset$. The query $Q() = R(A, B), S(B, C), T(C, D)$ is not hierarchical, since $\text{atoms}(B) \not\subseteq \text{atoms}(C)$, $\text{atoms}(C) \not\subseteq \text{atoms}(B)$ and $\text{atoms}(B) \cap \text{atoms}(C) \neq \emptyset$.*

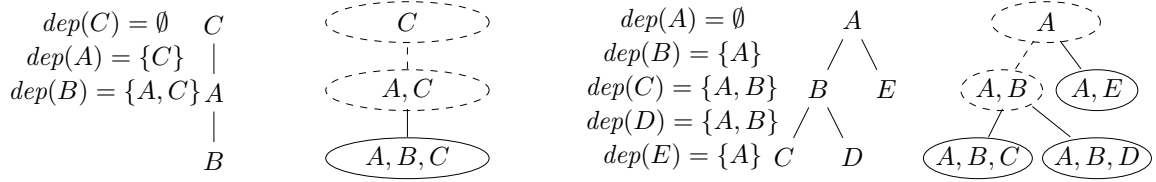


Figure 2.1: First and third from left: variable orders ω_1 and ω_2 for the two queries $Q_1(A, B|\cdot) = R(A, B), S(B, C), T(A, C)$ and $Q_2(B|A) = R(A, B, C), S(A, B, D), T(A, E)$; second and fourth from left: tree decompositions \mathcal{T}_1 and \mathcal{T}_2 for the hypergraphs of Q_1 and Q_2 . The two tree decompositions can be reduced by removing the dashed vertices and connecting the remaining vertices with edges.

2.4 Variable Orders

A variable order defines a partial ordering of the variables in a query. They are used as logical plans for the evaluation of conjunctive queries [65].

Definition 2.7 (Variable Order [65]). *A variable order ω for a CQAP query Q is a (rooted) forest with one node per variable in Q such that the variables of each atom in Q lie along the same root-to-leaf path in T .*

We introduce notation for a variable order ω for a CQAP query Q . Given a CQAP query, two variables *depend* on each other if they occur in the same atom of the query. The *dependency* dep_ω of ω is a function that maps each variable X in ω to its *dependent set*, i.e., the subset of X 's ancestor variables in ω on which the variables in the subtree rooted at X depend. We denote by $vars(\omega)$ the set of variables in ω . For each variable X in $vars(\omega)$, we call the set $\{X\} \cup dep_\omega(X)$ the *bag* of X using the language of tree decomposition [67]; we denote by $anc_\omega(X)$ the set of ancestor variables of X in ω .

Example 2.8. *Figure 2.1 (left) a variable order ω_1 for the query $Q_1(A, B|\cdot) = R(A, B), S(B, C), T(A, C)$. The dependency sets are $dep(C) = \emptyset$, $dep(A) = \{B\}$ and $dep(B) = \{A, C\}$.*

Figure 2.1 (third from left) also shows a variable order ω_2 of the query $Q_2(B|A) = R(A, B, C), S(A, B, D), T(A, E)$. The dependency sets of the variables are $dep(A) = \emptyset$, $dep(B) = \{A\}$, $dep(C) = \{A, B\}$, $dep(D) = \{A, B\}$ and $dep(E) = \{A\}$. \square

2.4.1 Variable Orders and Tree Decompositions

Variable orders have a close connection with the widely used notion of *tree decompositions*. Let us first recall the definition of the hypergraph of a query and the tree decomposition of a hypergraph.

Definition 2.9 (Hypergraph). *The hypergraph of a query Q is a pair $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where the vertices \mathcal{V} are the set of variables in Q and the hyperedges \mathcal{E} are the schemas of the atoms in Q .*

Definition 2.10 (Tree decomposition [67]). *A tree decomposition of a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is a pair $(T, (B_t)_{t \in V(T)})$, where T is a tree, $V(T)$ denotes the (multi-)set of vertices of T , and every B_t is a subset of \mathcal{V} , called the bag of t , such that the following properties hold:*

- *coverage: for each hyperedge $e \in \mathcal{E}$, there exists a bag B_t such that $e \subseteq B_t$, and*
- *connectivity: for each vertex $v \in \mathcal{V}$, the set of vertices $\{t \in T \mid v \in B_t\}$ induces a connected subtree of T .*

A tree decomposition is called *reduced* if there is not a bag which is the subset of another bag.

Example 2.11. *Consider the queries Q_1 and Q_2 in Example 2.8. Figure 2.1(second and fourth from left) shows tree decompositions \mathcal{T}_1 and \mathcal{T}_2 of the hypergraphs of the two queries. In \mathcal{T}_1 , all three variables have to be in the same bag to satisfy the connectivity property. The tree decomposition \mathcal{T}_1 can be reduced by removing the dashed vertex. The tree decomposition \mathcal{T}_2 can be reduced by removing the dashed vertices and connecting the remaining three vertices with edges as $\{A, B, C\} - \{A, B, D\} - \{A, E\}$. \square*

For a query Q and its hypergraph \mathcal{H} , there exists algorithms for bidirectional translations between the variable orders of Q and the tree decompositions of \mathcal{H} [65], which proves they are related concepts. Intuitively, for a variable order ω and its corresponding tree decomposition \mathcal{T} , each variable A of ω corresponds to a vertex t of \mathcal{T} , and the bag $dep(A) \cup \{A\}$ of A is exactly the bag B_t of t , i.e., $dep(A) \cup \{A\} = B_t$.

Example 2.12. *Consider the queries Q_1 and Q_2 in Example 2.8. Figure 2.1 shows the variable orders ω_1 and ω_2 of the two queries, and the tree decompositions \mathcal{T}_1 and \mathcal{T}_2 of the hypergraphs of the two queries. The variable orders ω_1 and ω_2 can be translated to the corresponding tree decompositions \mathcal{T}_1 and \mathcal{T}_2 , or in the opposite direction. Between ω_1 and \mathcal{T}_1 , the translation is as follows: the bag $\{C\}$ at the root variable C in ω_1 equals the bag $\{C\}$ at the root vertex in \mathcal{T}_1 , the bag $\{A, C\}$ at variable A in ω_1 equals the bag $\{A, C\}$ in \mathcal{T}_1 , and the bag $\{A, B, C\}$ at variable B in ω_1 equals the bag $\{A, B, C\}$ in \mathcal{T}_1 . The bijection between the bags in ω_2 and \mathcal{T}_2 is similar. \square*

In this thesis, we present our approaches and results using variable orders. They can also be presented using tree decompositions. Nevertheless, we use variable orders, since they are more natural for our approaches: In the preprocessing step, our approaches use variable orders to guide the construction of the data structures that represent the results of the queries; during the construction, the bound variables are eliminated one by one (projected away by computing the marginalization over them), and the variable orders define the partial orders of eliminating the bound variables in the queries. In the enumeration step, our approaches traverse the data structures in the order of the variable orders to compute the query results. Our approaches, as well as worst-case optimal join algorithms such as LeapFrog TrieJoin [73], proceed one variable at a time and not one bag at a time. Variable order based approaches express more naturally computation by variable elimination.

2.4.2 Classes of Variable Orders

We next introduce the classes of variable orders for CQAP queries used in the thesis.

Definition 2.13 (Canonical variable order). *A variable order ω for a query Q is canonical if for each relation R in Q , the schema of R is exactly the variables in the path from the root to the lowest variable of R in ω .*

Hierarchical queries are precisely those conjunctive queries that admit canonical variable orders.

Example 2.14. *Consider the queries Q_1 and Q_2 in Example 2.8 and their variable orders ω_1 and ω_2 in Figure 2.1. The variable order ω_1 is not canonical, since the schema of the relation R is $\{A, B\}$, but variables in the path from the root to the lowest variable of R are $\{A, B, C\}$. The variable order ω_2 is canonical, since the schema of the three relations R , S , and T are exactly the three root-to-leaf paths in the variable order, i.e., $\{A, B, C\}$, $\{A, B, D\}$ and $\{A, E\}$. \square*

Definition 2.15 (\mathcal{C} -top variable order). *Given a variable order ω for a query Q and a set of variables $\mathcal{C} \subseteq \text{vars}(Q)$, a variable order ω is \mathcal{C} -top if a variable not in \mathcal{C} is never an ancestor of a variable in \mathcal{C} .*

A variable order is *free-top* or *input-top* if it is \mathcal{C} -top where \mathcal{C} is the set of free variables or the set of input variables, respectively. A variable order is *access-top* if it is free-top and input-top.

Example 2.16. Consider the queries Q_1 and Q_2 in Example 2.8 and their corresponding variable orders ω_1 and ω_2 in Figure 2.1. The variable order ω_1 is input-top, since the query has no input variable; ω_1 is not free-top, since the bound variable C is on top of the free variables A and B .

The variable order ω_2 is input-top, since the input variable A is on top of other variables, and ω_2 is free-top, since the free variables A and B are on top of other variables; ω_2 is thus access-top, since it is both input-top and free-top. \square

The \mathcal{C} -top variable orders are related to the concept of \mathcal{C} -connex tree decompositions [9], which is a variant of tree decompositions defined for a set of variables \mathcal{C} .

Definition 2.17 (\mathcal{C} -connex tree decomposition [9]). Given a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{C} \subseteq \mathcal{V}$, a \mathcal{C} -connex tree decomposition of \mathcal{H} is a triple $(T, (B_t)_{t \in V(T)}, \mathcal{A})$, where:

- $(T, (B_t)_{t \in V(T)})$ is a tree decomposition of \mathcal{H} , and
- \mathcal{A} is a connected subset of $V(T)$ such that $\bigcup_{t \in \mathcal{A}} B_t = \mathcal{C}$.

In other words, a tree decomposition is \mathcal{C} -connex if its vertices that cover the variables in \mathcal{C} are connected.

Similar to tree decompositions and variable orders, \mathcal{C} -connex tree decompositions and \mathcal{C} -top variable orders are closely related. Intuitively, for a \mathcal{C} -top variable order ω and its corresponding \mathcal{C} -connex tree decomposition \mathcal{T} , the variables \mathcal{C} in ω correspond to the vertices covering \mathcal{C} in \mathcal{T} . In ω , these variables \mathcal{C} are on top of the other variables, so they are connected via the root of the variable order. Their corresponding vertices in \mathcal{T} are also connected due to the definition of \mathcal{C} -connex tree decompositions.

Example 2.18. Consider the query Q_2 in Example 2.8 and its corresponding variable order ω_2 and tree decomposition \mathcal{T}_2 in Figure 2.1. As discussed in Example 2.16, ω_2 is access-top, i.e., it is both $\{A\}$ -top and $\{A, B\}$ -top. The corresponding tree decomposition \mathcal{T}_2 is $\{A\}$ -connex and $\{A, B\}$ -connex: \mathcal{T}_2 is $\{A\}$ -connex, since the root vertex covers the variable A ; \mathcal{T}_2 is $\{A, B\}$ -connex, since its two bags $\{A\}$ and $\{A, B\}$ cover the variables A and B , and are connected. \square

2.5 Width Measures

We next introduce a width measure, namely the *static width*, for a variable order ω of a CQAP query Q . It captures the complexity to evaluate Q . The definition of the static width relies on the notion of *fractional edge cover number*.

Definition 2.19 (Fractional edge cover number[6]). *Given a conjunctive query Q and $\mathcal{F} \subseteq \text{vars}(Q)$, a fractional edge cover of \mathcal{F} is a solution $\lambda = (\lambda_{R(\mathcal{X})})_{R(\mathcal{X}) \in \text{atoms}(Q)}$ to the following linear program:*

$$\begin{aligned}
& \text{minimize} && \sum_{R(\mathcal{X}) \in \text{atoms}(Q)} \lambda_{R(\mathcal{X})} \\
& \text{subject to} && \sum_{R(\mathcal{X}) \in \text{atoms}(Q) \text{ s.t. } X \in \mathcal{X}} \lambda_{R(\mathcal{X})} \geq 1 && \text{for all } X \in \mathcal{F} \text{ and} \\
& \text{where} && \lambda_{R(\mathcal{X})} \in [0, 1] && \text{for all } R(\mathcal{X}) \in \text{atoms}(Q)
\end{aligned}$$

The optimal objective value of the above program is called the fractional edge cover number of \mathcal{F} and is denoted as $\rho_Q^*(\mathcal{F})$.

An *integral edge cover* of \mathcal{F} is a feasible solution to the variant of the above program with $\lambda_{R(\mathcal{X})} \in \{0, 1\}$ for each $R(\mathcal{X}) \in \text{atoms}(Q)$. The optimal objective value of this program is called the *integral edge cover number* of \mathcal{F} and is denoted as $\rho_Q(\mathcal{F})$. If Q is clear from the context, we omit the index Q in the expressions $\rho_Q^*(\mathcal{F})$ and $\rho_Q(\mathcal{F})$.

We next give the definition of the static width of a variable order ω .

Definition 2.20 (Static width). *Given a variable order ω for a query Q , the static width $w(\omega)$ of ω is:*

$$w(\omega) = \max_{X \in \text{vars}(\omega)} \rho_Q^*(\{X\} \cup \text{dep}_\omega(X)).$$

At each variable X in $\text{vars}(\omega)$, the fractional edge cover number $\rho_Q^*(\{X\} \cup \text{dep}_\omega(X))$ characterizes the time to compute a query that joins the atoms of Q covering the variables $\{X\} \cup \text{dep}_\omega(X)$ using a worst-case optimal algorithm [58]. The static width w of ω is the maximum fractional edge cover number over all variables in $\text{vars}(\omega)$.

Example 2.21. *Consider the queries Q_1 and Q_2 in Example 2.8 and their variable orders ω_1 and ω_2 in Figure 2.1. We first show the computation of the static width of ω_1 : to cover the bag $\{B\} \cup \text{dep}(B) = \{A, B, C\}$ of B , we assign $\lambda_{R(\mathcal{X})} = 0.5$ to each of the three atoms $R(A, B)$, $S(B, C)$ and $T(A, C)$ in Q_1 , and this gives us $\rho^*(\{A, B, C\}) = 0.5 + 0.5 + 0.5 = 1.5$. Since this is the largest fractional edge cover number over the bags of all variables in ω_1 , the static width of ω_1 is $w(\omega_1) = 1.5$. The static of ω_2 is $w(\omega_2) = 1$, since every bag can be covered by one single atom in Q_2 . \square*

The static width of a variable order is closely related to the *fractional hypertree width* of a tree decomposition.

Definition 2.22 (Fractional hypertree width [35]). *Given a query Q , its hypergraph H and a tree decomposition $\mathcal{T} = (T, (B_t)_{t \in V(T)})$ of H , the fractional hypertree width of \mathcal{T} is defined as $\max_{t \in V(T)} \rho_Q^*(B_t)$, denoted by $\text{fhtw}(\mathcal{T})$.*

Fractional hypertree width was originally defined for Boolean queries. The work [65] then introduced the width measure s^\uparrow , which was shown to generalize fhtw to CQs. In this thesis, we use fhtw to refer to this generalization.

The static widths of variable orders and the fractional hypertree widths of tree decompositions are closely related. For a variable order ω and its corresponding tree decomposition $\mathcal{T} = (T, (B_t)_{t \in V(T)})$, the static width $\mathbf{w}(\omega)$ is the same as the fractional hypertree width $\text{fhtw}(\mathcal{T})$: as discussed in Section 2.4.1, each variable A in the variable order ω corresponds to a vertex t of the tree decomposition, and $\text{dep}(A) \cup \{A\} = B_t$, so we have:

$$\mathbf{w}(\omega) = \max_{X \in \text{vars}(\omega)} \rho^*(\{X\} \cup \text{dep}_\omega(X)) = \max_{t \in V(T)} \rho^*(B_t) = \text{fhtw}(\mathcal{T}).$$

2.6 View Trees

In the preprocessing and update steps, our approaches construct and maintain data structures that represent the output of CQAP queries. Such data structures are called *view trees*. Intuitively, a view tree is a logical project-join plan in the classical database systems literature, but where each intermediate result is materialized.

Definition 2.23 (View tree [62]). *A view tree is a (rooted) tree with one view per node. The view at a node is either defined as the join of the views at its children, or the result of marginalizing out one or more variables away from its child view.*

Our algorithms construct view trees following access-top variable orders. We discuss in detail the procedure for constructing a view tree following an access-top variable order ω for a query Q in Section 4.1. The procedure has two steps:

1. it extends ω with atoms $\text{atoms}(Q)$ by adding these atoms as leaves of the variable order such that each atom is the child of its lowest variable, and then
2. it traverses ω bottom-up and at each variable X , it constructs the views of X by joining the views of the children of X and projecting the result to $\text{dep}_\omega(X)$.

Example 2.24. *Consider the query $Q(B|A) = R(A, B, C), S(A, B, D), T(A, E)$. Figure 2.2 shows the variable order ω for Q and the view tree constructed following ω . To construct the view tree, we first add the atoms of Q to ω under their lowest variables: $R(A, B, C)$ below C , $S(A, B, D)$ below D , and $T(A, E)$ below E . We then traverse the*

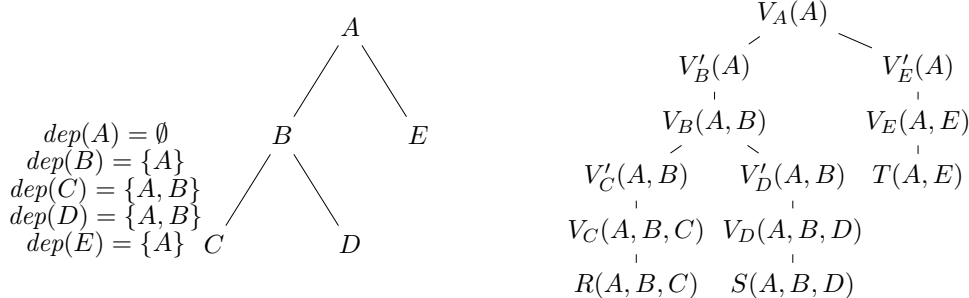


Figure 2.2: Left: variable order ω for the query $Q(B|A) = R(A, B, C), S(A, B, D), T(A, E)$ in Example 2.8; right: view tree constructed following ω .

variable bottom-up to construct views at each node: at variable C , since $R(A, B, C)$ is its only child atom, we create the view $V_C(A, B, C) = R(A, B, C)$ and then create the view $V'_C(A, B) = V_C(A, B, C)$, which projects away C . We build the views V_D, V'_D, V_E , and V'_E similarly at variables D and E . At variable B , we build the view $V_B(A, B) = V'_C(A, B), V'_D(A, B)$ by joining its two children views, and the view $V'_B(A) = V_B(A, B)$ by projecting away B . Finally, we create the view $V_A(A) = V'_B(A), V'_E(A)$ by joining its two children views. \square

2.7 Computational Model

We consider the RAM model of computation. Each relation (or materialized view) R over schema \mathcal{X} is implemented by a data structure that stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple \mathbf{x} with $R(\mathbf{x}) \neq 0$ and needs $O(|R|)$ space. This data structure can: (1) look up, insert, and delete entries in constant time, (2) enumerate all stored entries in R with constant delay, and (3) report $|R|$ in constant time. For a schema $\mathcal{S} \subset \mathcal{X}$, we use an index data structure that for any $\mathbf{t} \in \text{Dom}(\mathcal{S})$ can: (4) enumerate all tuples in $\sigma_{\mathcal{S}=\mathbf{t}}R$ with constant delay, (5) check $\mathbf{t} \in \pi_{\mathcal{S}}R$ in constant time; (6) return $|\sigma_{\mathcal{S}=\mathbf{t}}R|$ in constant time; and (7) insert and delete index entries in constant time.

We next exemplify a data structure that conforms to the above computational model. Consider a relation (materialized view) R over schema \mathcal{X} . A hash table with chaining stores key-value entries $(\mathbf{x}, R(\mathbf{x}))$ for each tuple \mathbf{x} over \mathcal{X} with $R(\mathbf{x}) \neq 0$. The entries are doubly linked to support enumeration with constant delay. The hash table can report the number of its entries in constant time and supports lookups, inserts, and deletes in constant time on average, under the assumption of simple uniform hashing.

To support index operations on a schema $\mathcal{F} \subset \mathcal{X}$, we create another hash table with chaining where each table entry stores a tuple \mathbf{t} of \mathcal{F} -values as key and a doubly-linked list of pointers to the entries in R having the \mathcal{F} -values \mathbf{t} as value. Looking up an index

entry given \mathbf{t} takes constant time on average under simple uniform hashing, and its doubly-linked list enables enumeration of the matching entries in R with constant delay. Inserting an index entry into the hash table additionally prepends a new pointer to the doubly-linked list for a given \mathbf{t} ; overall, this operation takes constant time on average. For efficient deletion of index entries, each entry in R also stores back-pointers to its index entries (one back-pointer per index for R). When an entry is deleted from R , locating and deleting its index entries in doubly-linked lists takes constant time per index.

2.8 Lower Bounds

The optimality results introduced in this thesis are based on the hardness of the following Online Boolean Matrix-Vector Multiplication (OMv) problem.

Definition 2.25 (Online Boolean Matrix-Vector Multiplication [36]). *We are given an $n \times n$ matrix \mathbf{M} and receive n column vectors of size n , denoted by $\mathbf{v}_1, \dots, \mathbf{v}_n$, one by one; after seeing each vector \mathbf{v}_i , we output the product $\mathbf{M}\mathbf{v}_i$ before we see the next vector.*

It is strongly believed that the OMv problem cannot be solved in subcubic time.

Conjecture 26 (OMv Conjecture, Theorem 2.4 in [36]). *For any $\gamma > 0$, there is no algorithm that solves the OMv problem in time $\mathcal{O}(n^{3-\gamma})$.*

The OMv conjecture has been used to exhibit conditional lower bounds for many dynamic problems, including those previously based on other popular problems and conjectures, such as 3SUM and combinatorial Boolean matrix multiplication [10].

Chapter 3

Overview of the Main Results

In this chapter, we give a brief overview of the three main results of the thesis. We give the necessary definitions and theorems in this chapter to make the statements precise. The algorithms, discussions and the proofs are given in the later chapters.

3.1 Fully Dynamic Evaluation for CQAP Queries

The first contribution of the thesis is a fully dynamic evaluation approach for arbitrary CQAP queries.

3.1.1 Query Fractures

The first step of the approach is to decompose the given CQAP query into a set of sub-queries. The intuition is that we break the hypergraph of the CQAP query at its input variables, so that it becomes a set of smaller hypergraphs, each of which is a sub-query. The results of these sub-queries are conditionally independently on the input variables. That is, given fixed values over the input variables, the result of the CQAP query is the Cartesian product of the results of these sub-queries. We call such a decomposition the *fracture* of the query.

Definition 3.1 (Query fracture). *The fracture of a CQAP query Q is a CQAP query Q_{\dagger} constructed from Q in three steps:*

- *For each input variable I and each relation R containing I , replace I by a new variable I_R .*
- *Compute the connected components of the hypergraph of the modified query.*
- *If several new variables I_1, \dots, I_n originating from the same input variable I end in the same connected component Q' , replace them by one input variable I_1 .*

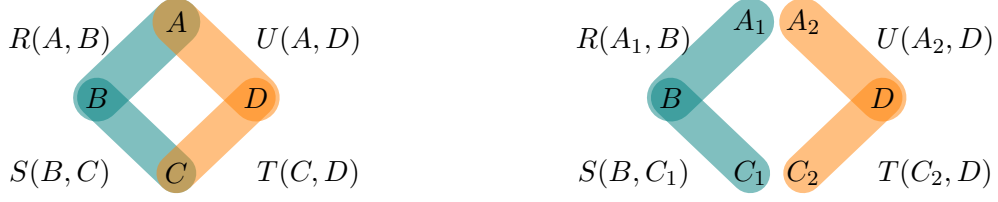


Figure 3.1: (Left) Hypergraph of the 4-cycle query Q_1 in Example 3.2. (Right) Fracture of Q_1 . It has two connected components $R(A_1, B), S(B, C_1)$ and $U(A_2, D), T(C_2, D)$.

The first two steps break the hypergraph of the CQAP query into a set of smaller hypergraphs by replacing the input variables by new variables. Multiple new variables originating from the same input variable might end in the same connected component. This might increase the static width of the query, since if multiple new variables are in the same bag, we need to cover all new variables in different relations instead of only one original input variable (see Example 3.3). Hence, we have the third step to merge the new variables that are in the same connected component.

Example 3.2. Consider the 4-cycle query

$$Q_1(B, D|A, C) = R(A, B), S(B, C), T(C, D), U(A, D).$$

Figure 3.1(left) shows the hypergraph of Q_1 . To compute the fracture of Q_1 , we replace the two input variables B and D by the new variables B_1 and B_2 in relations R and S , and by D_1 and D_2 in relations T and U , respectively. This results in the fracture Q_1^\dagger in Figure 3.1 (right):

$$Q_1^\dagger(B, D|A_1, A_2, C_1, C_2) = R(A_1, B), S(B, C_1), T(C_2, D), U(A_2, D).$$

It has two connected components $R(A_1, B), S(B, C_1)$ and $U(A_2, D), T(C_2, D)$. The two sub-queries defined by the two connected components are

$$Q_1^1(B, D|A_1, A_2) = R(A_1, B), S(B, C_2) \text{ and } Q_1^2(B, D|A_1, A_2) = U(A_2, D), T(C_2, D).$$

The 4-cycle query Q_1 is not hierarchical, but its fracture Q_1^\dagger is. \square

We demonstrate in the next example that the third step of the fracture computation is important as it can reduce the static width of the fracture.

Example 3.3. Consider now the triangle query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. To compute the fracture of Q , we replace the input variable A by the new variables A_1 and A_2 in relations R and T , respectively, which results in:

$$Q'(B, C|A_1, A_2) = R(A_1, B), S(B, C), T(C, A_2).$$

It has only one connected component, so we replace the two new variables A_1 and A_2 , which originate from the same input variable A , by a single new variable A_1 , which results in the fracture Q_{\dagger} :

$$Q_{\dagger}(B, C|A_1) = R(A_1, B), S(B, C), T(C, A_1).$$

The fracture Q_{\dagger} is the same as the original query Q .

The two new variables A_1 and A_2 in Q' breaks the cycle formed by the three relations R , S and T , and this increases the static width. Hence, it is important that we replace the new variables A_1 and A_2 by a single variable A_1 in the fracture Q_{\dagger} , which reduces the static width from 2 to 1.5: the static width of $Q'(B, C|A_1, A_2)$ is 2, since the two new variables A_1 and A_2 are in the same bag in any variable order, and we need at least two edges to cover them. The static width of $Q_{\dagger}(B, C|A_1)$ is 1.5, since the new variable A_1 is in the same bag with the two input variables B and C , and we can assign the weight 0.5 to each edge R , S and T to cover them. \square

Once the fracture of the given CQAP query is computed, we can evaluate each sub-query in the fracture independently. For each sub-query, we compute a view tree that supports the enumeration of the tuples over the output variables for a tuple over the input variables and maintain the view tree under single-tuple updates. The result of the CQAP query is the Cartesian product of the results of the sub-queries. The overall complexity is the sum of the complexities of the sub-queries, or asymptotically the max complexity of the sub-queries.

3.1.2 Complexities

The static width w introduced in Chapter 2 measures the complexity for building the view trees for the query. We define another width measure, the dynamic width δ , to measure the complexity for maintaining these view trees. We give the formal definition of the dynamic width in Definition 4.9. Intuitively, the dynamic width δ is the maximal static width over the delta queries for updates to all base relations.

We state the complexities of our approach using the static width and dynamic width in the following theorem:

Theorem 3.4. *Given a CQAP with static width w and dynamic width δ and a database of size N , the query can be evaluated with $\mathcal{O}(N^w)$ preprocessing time, $\mathcal{O}(N^\delta)$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

We discuss in detail the procedures of this approach and give the proof of the complexity result in Chapter 4.

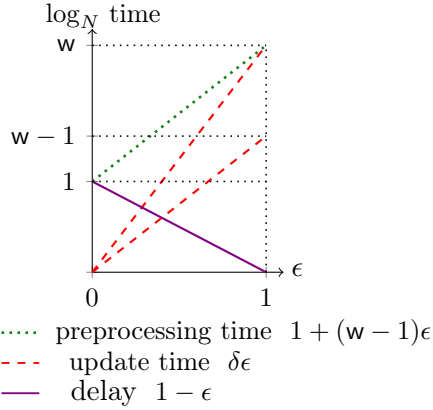


Figure 3.2: Preprocessing time, enumeration delay and amortized update time for a CQAP query with a hierarchical fracture, static width w and dynamic width δ (δ can be w or $w - 1$, hence the two red lines for the update time).

3.2 Preprocessing-Update-Enumeration Trade-offs

The second contribution of the thesis is a dynamic evaluation approach which uncovers the trade-offs between the preprocessing time, update time and enumeration delay for CQAP queries whose fractures are hierarchical and triangle CQAP queries.

Our approach has two core ideas. First, we partition the base relations into heavy and light parts based on the degrees of the values. This transforms a query over the base relations into a union of queries over heavy and light relation parts. Second, we employ different evaluation strategies for different heavy-light combinations of parts of the base relations. This allows us to confine the worst-case behavior caused by high-degree values in the database during query evaluation. By tuning the threshold for the heavy-light partitioning, our approach can reduce update time with the cost of more enumeration delay, or the other way around.

3.2.1 Queries with Hierarchical Fractures

For CQAP queries with hierarchical fractures, the complexities in Theorem 3.4 can be parameterized to uncover trade-offs between preprocessing, update and enumeration.

Theorem 3.5. *Consider any CQAP query Q with static width w and dynamic width δ , a database of size N , and $\epsilon \in [0, 1]$. If Q 's fracture is hierarchical, then Q admits $\mathcal{O}(N^{1+(w-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortized update time for single-tuple updates.*

These trade-offs are illustrated in Figure 3.2. For hierarchical queries, the dynamic width δ can be w or $w - 1$ (Proposition 5.27), hence the two red lines for the update time.

Optimality. Our trade-offs defined in Theorem 3.5 are optimal for two classes of CQAP queries with hierarchical fractures, namely the $CQAP_0$ and $CQAP_1$ queries, unless the OMv conjecture is false. These two classes of queries are queries whose results can be efficiently maintained: the $CQAP_0$ queries are those admitting constant update time, i.e. $\delta = 0$, while the $CQAP_1$ queries are those admitting slighter higher update time, i.e. $\delta = 1$. We next give the characterization of the two classes of queries.

Definition 3.6 (Variable dominance). *Given a query Q , for any two variables A and B in $\text{vars}(Q)$, B dominates A if $\text{atoms}(A) \subset \text{atoms}(B)$.*

Definition 3.7 ($CQAP_0$ queries). *Given a query Q , Q is free-dominant (input-dominant) if for any two variables A and B in $\text{vars}(Q)$, it holds: if A is free (input) and B dominates A , then B is free (input). A query is in $CQAP_0$ if its fracture is hierarchical, free-dominant, and input-dominant.*

Definition 3.8 ($CQAP_1$ queries). *Given a query Q , Q is almost free-dominant (almost input-dominant) if: (1) For any atom $R(\mathcal{X}) \in \text{atoms}(B)$, there is another atom $S(\mathcal{Y}) \in \text{atoms}(B)$ such that each free (input) variable dominated by B is contained in $\mathcal{X} \cup \mathcal{Y}$; (2) B is not already free-dominant (input-dominant). A query is in $CQAP_1$ if its fracture is hierarchical and is almost free-dominant, or almost input-dominant, or both.*

Example 3.9. *The query $Q_1(A, C | B, D) = R(A, B), S(B, C), T(C, D), U(A, D)$ is input-dominant, free-dominant, but not hierarchical. Its fracture $Q_{\dagger}(A, C | B_1, B_2, D_1, D_2) = R(A, B_1), S(B_2, C), T(C, D_1), U(A, D_2)$ is hierarchical but not input-dominant: C dominates both B_2 and D_1 and A dominates both B_1 and D_2 , yet A and C are not input. It is however almost input-dominant: A is not input and for any of its atoms $R(A, B_1)$ and $U(A, D_2)$, there is another atom $U(A, D_2)$ and respectively $R(A, B_1)$ such that both $R(A, B_1)$ and $U(A, D_2)$ cover the variables B_1 and D_2 dominated by A ; a similar reasoning applies to C . This means that Q_1 is in $CQAP_1$.*

The query $Q_2(A | B) = S(A, B), T(B)$ is in $CQAP_0$, since its fracture $Q_{\dagger}(A | B_1, B_2) = S(A, B_1), T(B_2)$ is hierarchical, free-dominant, and input-dominant.

The query $Q_3(B | A) = S(A, B), T(B)$ is in $CQAP_1$. Its fracture is the query itself. It is hierarchical, yet not input-dominant, since B dominates A and is not an input variable. It is, however, almost input-dominant: for each atom of B , there is one other atom such that together they cover A . Indeed, atom $S(A, B)$ already covers A , and it also does so together with $T(B)$; atom $T(B)$ does not cover A , but it does so together with $S(A, B)$. This means that Q_3 is almost input-dominant, and thus in $CQAP_1$.

The following are the smallest hierarchical queries that are not in $CQAP_0$ but in $CQAP_1$: $Q(A | \cdot) = R(A, B), S(B)$; $Q(B | A) = R(A, B), S(B)$; $Q(\cdot | A) = R(A, B), S(B)$.

□

We discussed in detail the procedures of our approach on CQAP queries with hierarchical fractures and give the proofs on its complexity and optimality results in Section 5.

3.2.2 Triangle Queries

Recall the *triangle CQAP queries* are of the form:

$$Q(\mathcal{O} \mid \mathcal{I}) = R(A, B), S(B, C), T(C, A), \quad \text{where } \mathcal{O} \cup \mathcal{I} \subseteq \{A, B, C\}.$$

Based on their access patterns, we characterize triangle CQAP queries into the categories: the \mathbf{C}_{lookup} , \mathbf{C}_{count} , \mathbf{C}_1 , \mathbf{C}_2 , and \mathbf{C}_3 queries. The \mathbf{C}_{lookup} query has the access pattern $(\cdot \mid A, B, C)$, that is, all three variables are input variables. The \mathbf{C}_{count} query has the access pattern $(\cdot \mid \cdot)$, i.e., no output and input variables. The other queries, that is, the \mathbf{C}_k queries, where $k \in \{1, 2, 3\}$, have the access patterns with

- k input variables, or
- k output variables and no input variable.

The following table summarizes the triangle CQAP queries in each category. The other \mathbf{C}_1 and \mathbf{C}_2 queries, e.g., $Q(B \mid \cdot)$ or $Q(B, C \mid \cdot)$, are skipped, since the join of the relations is symmetric in the variables A , B , and C .

\mathbf{C}_{lookup}	\mathbf{C}_{count}	\mathbf{C}_1	\mathbf{C}_2	\mathbf{C}_3
$Q(\cdot \mid A, B, C)$	$Q(\cdot \mid \cdot)$	$Q(A \mid \cdot)$	$Q(A, B \mid \cdot)$	$Q(A, B, C \mid \cdot)$
		$Q(\cdot \mid A)$	$Q(\cdot \mid A, B)$	
		$Q(B \mid A)$	$Q(C \mid A, B)$	
		$Q(B, C \mid A)$		

The \mathbf{C}_{count} query $Q(\cdot \mid \cdot) = R(A, B), S(B, C), T(C, A)$ is named the count query since it computes the number of triangles in the database. The \mathbf{C}_{lookup} query $Q(\cdot \mid A, B, C) = R(A, B), S(B, C), T(C, A)$ is named the triangle lookup query since it serves to look up the multiplicity of a given triangle in the database, i.e., the multiplicity of a given tuple over A , B and C in the join of the relations R , S and T . The \mathbf{C}_{lookup} query is in CQAP_0 and others are outside CQAP_0 .

For the triangle CQAP queries, we have the trade-offs as defined in Theorem 3.10:

Theorem 3.10. *Given a triangle CQAP query Q , a database of size N , and $\epsilon \in [0, 1]$. If Q is the \mathbf{C}_{lookup} query, it admits constant preprocessing time, update time, and enumeration time; otherwise, the query Q admits $\mathcal{O}(N^{\frac{3}{2}})$ preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ amortized update time, and the enumeration delay given in the table below.*

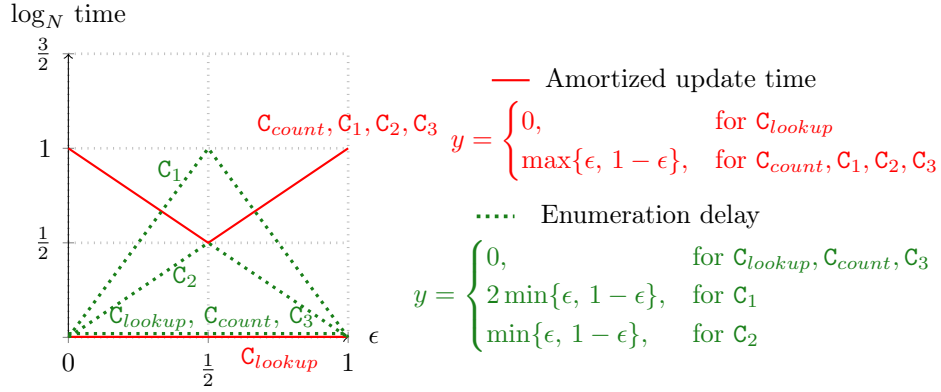


Figure 3.3: Trade-offs between the amortized update time and enumeration delay for the dynamic evaluation of triangle CQAP queries. N is the database size. The complexities are parameterized by ϵ .

	C_{count}	C_1	C_2	C_3
Enumeration Delay	$\mathcal{O}(1)$	$\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$	$\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$	$\mathcal{O}(1)$

The trade-offs are summarized in Figure 3.3. It shows the trade-offs between the amortized update time and the enumeration delay based on the parameter ϵ , which ranges from 0 to 1.

Optimality. Our trade-offs for all triangle CQAPs are optimal, conditioned on the OMv conjecture.

We discussed in detail the procedures of our approach for triangle CQAP queries and give the proofs on its complexity and optimality results in Section 6.

3.3 A Dichotomy Result

The third contribution of the thesis is a dichotomy result for the queries in $CQAP_0$: the queries in $CQAP_0$ are exactly those CQAP queries admitting linear-time preprocessing and constant-time update and enumeration delay. All queries outside $CQAP_0$ do not admit constant-time update and delay regardless of the preprocessing time, unless the OMv conjecture [36] fails.

The dichotomy result is stated in the following theorem.

Theorem 3.11. *Consider an arbitrary CQAP query Q and a database of size N .*

- *If Q is in $CQAP_0$, then it admits $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ enumeration delay, and $\mathcal{O}(1)$ update time for single-tuple updates.*

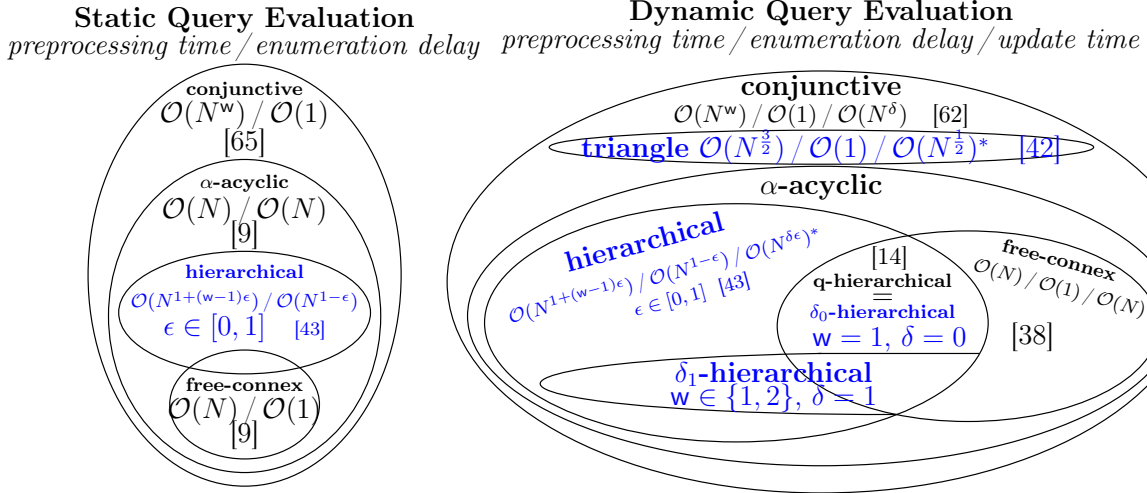


Figure 3.4: Landscapes of static and dynamic query evaluation. w : static width; δ : dynamic width; *: amortized time.

- If Q is not in $CQAP_0$ and has no repeating relation symbols, then there is no algorithm that computes Q with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, for any $\gamma > 0$, unless the OMv conjecture fails.

This dichotomy implies that our approach for general CQAP queries (Theorem 3.4) is optimal for the queries in $CQAP_0$. The proof of the theorem is given in Chapter 7.

3.4 Current Landscape of Conjunctive Query Evaluation

Our approaches, Theorems 3.4, 3.5 and 3.10, can also be applied to both the static and dynamic evaluation of CQs (without access patterns): for CQs without access patterns, we consider them as CQAP queries that have no input variable, and for the static setting, we compute the view trees in the same way as in the dynamic setting, and enumerate the results from the view trees.

Figure 3.4 summarizes the landscape of static and dynamic evaluations of CQs and where our results stand. The blue results in the figure are the new results of this thesis. The other results except the α -acyclic in the static setting can be recovered by Theorem 3.4. The δ_0 - and δ_1 -hierarchical queries are the hierarchical queries that are in $CQAP_0$ and respectively $CQAP_1$. They have dynamic widths 0 and 1. The δ_0 -hierarchical queries are precisely the q-hierarchical queries from prior work [14]. All free-connex hierarchical queries are either δ_0 - or δ_1 -hierarchical.

Chapter 4

The Case of General Queries

In this chapter, we introduce a fully dynamic evaluation approach for arbitrary CQAP queries whose complexity matches Theorem 3.4. Our dynamic evaluation technique comprises three distinct, yet independent stages: preprocessing, enumeration and updates. In the preprocessing stage, we construct a set of view trees that represents the result of the query over both the input and output variables. These view trees allow the enumeration of tuples over the output variables for any access request over the input variables, and can be updated efficiently upon single-tuple updates to the base relations. We present the preprocessing, enumeration and update stages in detail in Sections 4.1, 4.2 and 4.3, respectively. We then prove Theorem 3.4 in Section 4.4.

We consider in the following a fixed CQAP query $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$, and a database of size N .

4.1 Preprocessing

In the preprocessing stage, we construct a set of view trees, one for each connected component in the fracture Q_{\dagger} of Q , which together represent the result of Q_{\dagger} over both its input and output variables. These view trees are constructed following an access-top variable order ω of Q_{\dagger} . In the following, we discuss the case $Q = Q_{\dagger}$, which means ω consists of a single tree; otherwise, we apply the preprocessing stage to each tree in ω .

4.1.1 Extended Variable Orders

The first step of the preprocessing stage is to extend the access-top variable order ω with the base relations of Q . An *extended variable order* is obtained by adding the atoms $atoms(Q)$ of Q to ω as leaves such that each atom is the child of its lowest variable.

In the following sections, when we refer to a variable order, we mean the variable order extended with the atoms. Consider an extended variable order ω of Q . For each

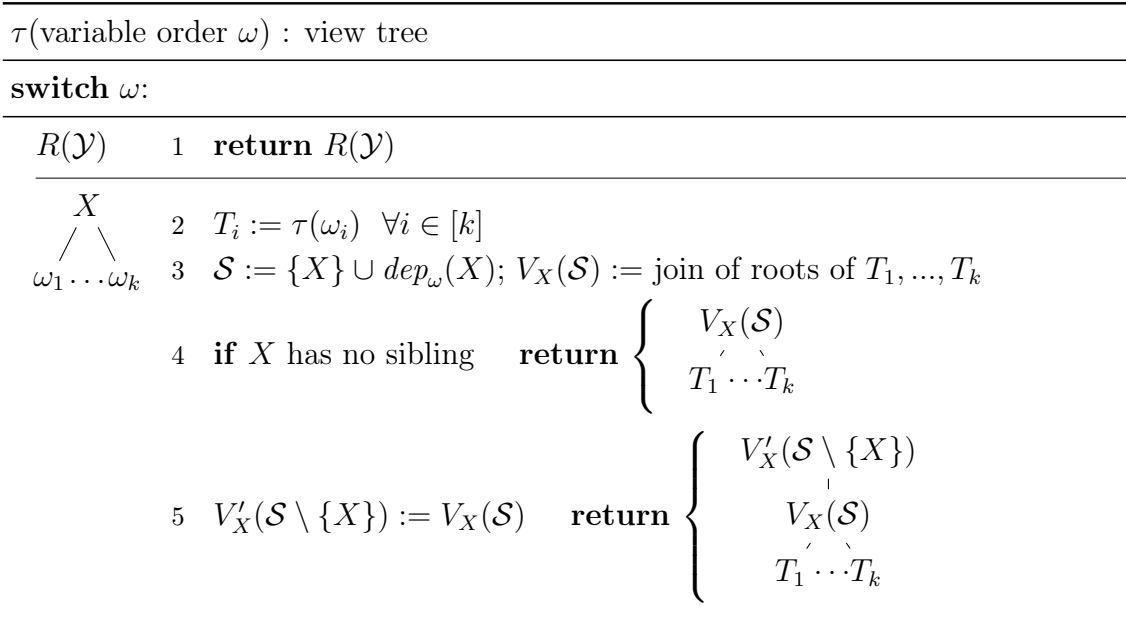


Figure 4.2: Construction of a view tree following a variable order ω . At each variable X in ω , the function creates a view V_X whose schema consists of X and the dependency set of X . If X has siblings, it adds a view on top of V_X that marginalizes out X .

the view tree $\tau(\omega)$ can be constructed in $\mathcal{O}(N^{\mathbf{w}(\omega)})$ time. By choosing a variable order whose static width is $\mathbf{w}(Q)$, the preprocessing time of our approach becomes $\mathcal{O}(N^{\mathbf{w}(Q)})$, as stated in Theorem 3.4.

The next example demonstrates our view tree construction for a query in CQAP_0 .

Example 4.2. Consider again the query $Q(B, C, D, E \mid A) = R(A, B, C), S(A, B, D), T(A, E)$ in Example 4.1 and the two sub-queries $Q_1(B, C, D \mid A_1) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E \mid A_2) = T(A_2, E)$ in its fracture. Figure 4.3 depicts an access-top variable order (left) for Q_1 and its corresponding view tree (middle). The variable order has static width 1. Each variable in the variable order is mapped to a view in the view tree, e.g., B is mapped to $V_B(A_1, B)$, where $\{B, A_1\} = \{B\} \cup \text{dep}(B)$. The views V'_C and V'_D are auxiliary views. The views V'_C, V'_D , and V_{A_1} marginalize out the variables C, D and respectively B from their child views. The view V_B is the intersection of V'_C and V'_D . Hence, all views can be computed in $\mathcal{O}(N)$ time. Since the query fracture is acyclic, the view tree does not contain indicator projections.

The only access-top variable order for the connected component Q_2 of Q_{\dagger} is the top-down path $A_2 - E - T(A_2, E)$, as shown in the right of Figure 4.1. The views mapped to A_2 and E are $V_{A_2}(A_2)$ and respectively $V_E(A_2, E)$. They can obviously be computed in $\mathcal{O}(N)$ time. \square

The next example considers a CQAP_1 whose preprocessing time is quadratic.

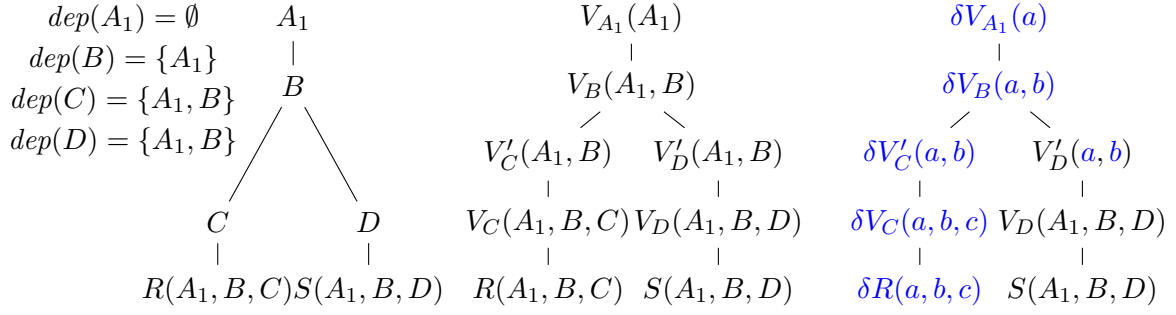


Figure 4.3: (Left) Access-top variable order for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree constructed from the variable order; (right) the delta view tree under a single-tuple update to R .

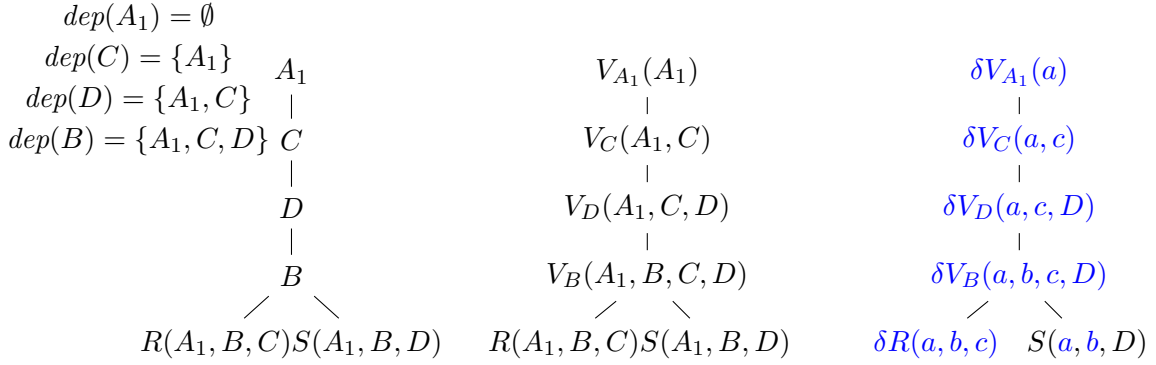


Figure 4.4: (Left) Access-top variable order for $Q_1(B, D|A_1, C) = R(A_1, B, C), S(A_1, B, D)$; (middle) the view tree corresponding to the variable order; (right) the delta view tree under a single-tuple update to R .

Example 4.3. Consider the CQAP₁ $Q(E, D|A, C) = R(A, B, C), S(A, B, D), T(A, E)$ and its fracture $Q_{\dagger}(E, D|A_1, A_2, C) = R(A_1, B, C), S(A_1, B, D), T(A_2, E)$. The fracture has the two connected components $Q_1(B, D | A_1, C) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E | A_2) = T(A_2, E)$. The hypergraphs of Q and its fracture are the same as for the query in Example 4.2, as shown in Figure 4.1. Figure 4.4 depicts an access-top variable order (left) for Q_1 and its corresponding view tree (middle). The variable order has static width 2. The view V_B joins the relations R and S , which takes $\mathcal{O}(N^2)$ time. The views V_D, V_C , and V_A are constructed from V_B by marginalizing out one variable at a time. Hence, the view tree construction takes $\mathcal{O}(N^2)$ time. The view tree for Q_2 is the same as in Example 4.2 and can be constructed in linear time. \square

4.1.3 Indicator Projections

In the previous section, we discussed the procedure to construct a view tree for a CQAP query. This procedure can be sub-optimal when the query is cyclic.

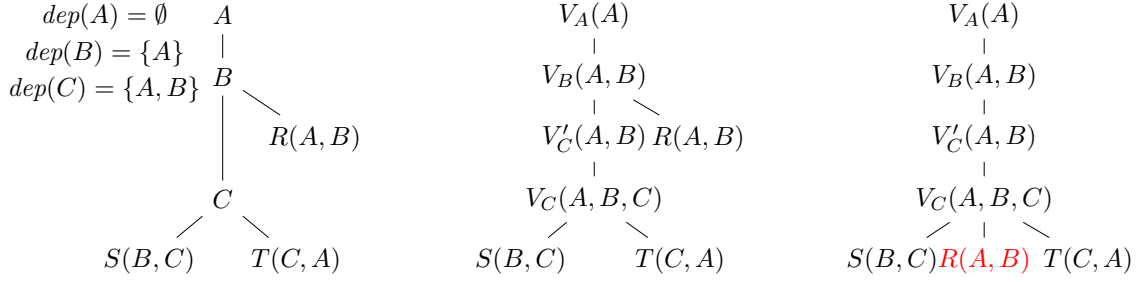


Figure 4.5: (Left) Access-top variable order for the cyclic CQAP query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$; (middle) the view tree constructed following the variable order; (right) the view tree obtained from the left view tree by placing R under V_C .

Example 4.4. Consider the triangle query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. The fracture Q_+ of Q is the query itself. Figure 4.5 shows an access-top variable order for Q (left) and the view tree constructed following the variable order (middle).

The view $V_C(A, B, C)$ can be computed in $\mathcal{O}(N^2)$ time, since there are $\mathcal{O}(N)$ B -values in S and $\mathcal{O}(N)$ C -values in A . This is larger than the time characterized by the static width, i.e., $\mathcal{O}(N^{1.5})$.

To avoid this large view V_C , we can change the view tree by placing R under V_C , as shown in the right view tree in Figure 4.5. The relation R forms a cycle with S and T , which constraints the size of V_C to be $\mathcal{O}(N^{1.5})$, and thus the time to compute V_C is $\mathcal{O}(N^{1.5})$ using a worst-case optimal join algorithm. □

The above example shows that placing a relation under a different view may create a cycle of relations and reduce the size and the computation time of the view. This strategy, however, does not work when the relation is in multiple cycles of relations in different parts of a view tree. For example, consider the 4-loop query with a diagonal relation:

$$Q_{\square}(A, B, C, D|\cdot) = R(A, B), S(B, C), T(C, D), U(D, A), W(A, D).$$

The diagonal relation W is in two triangle sub-queries. We need to place W under the two different views that compute the two triangle sub-queries, to reduce the time to compute these views. If we duplicate W and place it under the two views, the multiplicities of the tuples in W will be multiplied twice instead of once, which results in a wrong answer. To resolve this issue, we introduce the concept of *indicator projection*.

indicators(CQAP Q , variable order ω) : extended variable order

switch ω :

$R(\mathcal{Y})$	1 return $R(\mathcal{Y})$
$\begin{array}{c} X \\ / \quad \backslash \\ \omega_1 \cdots \omega_k \end{array}$	2 $\hat{\omega}_i := \text{indicators}(\omega_i) \quad \forall i \in [k]$
	3 $\mathcal{S} := \{X\} \cup \text{dep}_\omega(X)$; $\mathcal{R} := \text{atoms}(\omega)$
	4 $\mathcal{I} := \{I_{\mathcal{Z}}R(\mathcal{Z}) \mid R(\mathcal{Y}) \in (\text{atoms}(Q) \setminus \mathcal{R}) \text{ and } \mathcal{Z} = \mathcal{Y} \cap \mathcal{S} \neq \emptyset\}$
	5 $\{I_1, \dots, I_\ell\} := \text{GYO}^*(\mathcal{I} \cup \mathcal{R})$
	6 return $\left\{ \begin{array}{c} X \\ / \quad \backslash \\ \hat{\omega}_1 \cdots \hat{\omega}_k \quad I_1 \cdots I_\ell \end{array} \right.$

Figure 4.6: Adding indicator projections to a variable order ω of a CQAP Q . Each variable X in ω gets as new children the indicator projections of relations that do not occur in the subtree rooted at X but form a cycle with those that occur. $\text{GYO}^*(\mathcal{I} \cup \mathcal{R})$ uses the GYO reduction [31] to determine the originated indicators in \mathcal{I} that form a cycle with the atoms in \mathcal{R} .

Definition 4.5 (Indicator Projection). *For a relation R over schema \mathcal{X} and $\mathcal{Y} \subseteq \mathcal{X}$, the indicator projection $I_{\mathcal{Y}}R$ is a relation over \mathcal{Y} such that [1]:*

$$\text{for all } \mathbf{y} \in \text{Dom}(\mathcal{Y}) : I_{\mathcal{Y}}R(\mathbf{y}) = \begin{cases} 1 & \text{if there is } \mathbf{t} \in R \text{ such that } \mathbf{y} = \mathbf{t}[\mathcal{Y}] \\ 0 & \text{otherwise.} \end{cases}$$

Intuitively, the indicator projection $I_{\mathcal{Y}}R$ is a simulation of the bag semantics of $R(\mathcal{Y})$ using our data model. Instead of moving relations in a view tree, we put the indicator projections of relations under views. These indicator projections do not affect the results of the query, but can reduce the time to compute the views.

Extend a Variable Order with Indicator Projections. In the preprocessing step for a cyclic query, after extending the given variable order with atoms, we further extend the variable order with indicator projections. The view tree constructed following the extended variable order will then have these indicator projections participating in the construction of its views.

Given a CQAP query Q and a variable order ω for Q , the function `indicators` in Figure 4.6 extends ω with indicator projections. At each variable X in ω , we compute the set \mathcal{I} of all possible indicator projections (Line 4). Such indicators $I_{\mathcal{Z}}R$ are for relations R whose atoms are not included in the subtree rooted at X but share a non-empty set \mathcal{Z} of variables with $\{X\} \cup \text{dep}_\omega(X)$. We choose from this set those indicators that form a cycle with the atoms in the subtree of ω rooted at X (Line 5). We achieve this using a variant

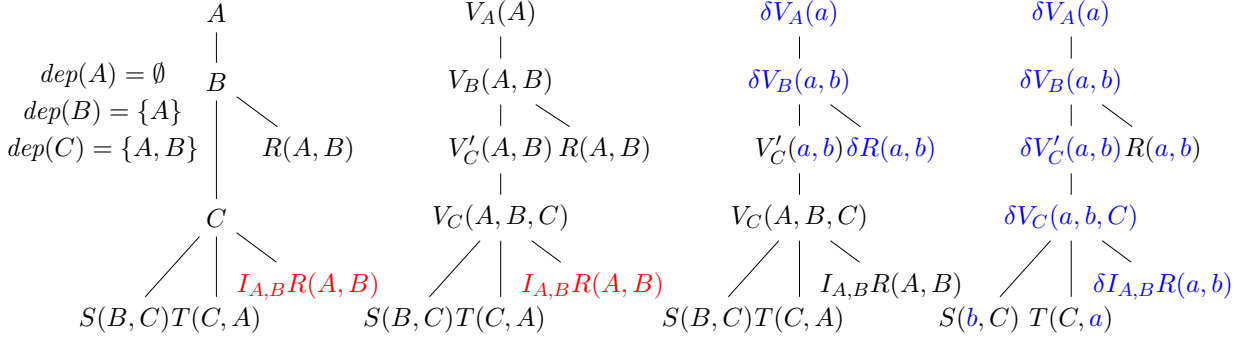


Figure 4.7: (Left) Access-top variable order for the cyclic CQAP query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$; (second from left) the view tree constructed from the variable order; (two from right) the two delta view trees under a single-tuple update to R .

of the GYO reduction [11]. Given the hypergraph formed by the hyperedges representing these indicators \mathcal{I} and the atoms \mathcal{R} , GYO repeatedly applies two rules until it reaches a fixpoint: (1) Remove a node that only appears in one hyperedge; (2) Remove a hyperedge that is included in another hyperedge. If the result of GYO is a hypergraph with no nodes and one empty hyperedge, then the input hypergraph is (α -)acyclic. Otherwise, the input hypergraph is cyclic and the GYO's output is a hypergraph with cycles. Our GYO variant, dubbed GYO* in Figure 4.6, returns those indicator projections from \mathcal{I} that contribute to this non-empty output hypergraph. The chosen indicator projections become children of X (Line 6). The following example demonstrates the extension of a variable order with indicator projections.

Example 4.6. Consider again the triangle query $Q(B, C|A) = R(A, B), S(B, C), T(C, A)$. The fracture Q_+ of Q is the query itself.

Figure 4.7 shows the access-top variable order ω for Q . The input variable A is on top of the output variables B and C . At variable C , the function indicators from Figure 4.6 creates an indicator projection $I_{A,B}R$ since the relation R is not under C but forms a cycle with the relations S and T . By adding $I_{A,B}R$ below C , the time to compute V_C reduces from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{\frac{3}{2}})$, and the time to construct the view tree reduces from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{\frac{3}{2}})$.

In the preprocessing stage, we construct the view tree following the variable order as shown in Figure 4.7 (second from left). The view V_C joins the relations R and S and the indicator projection $I_{A,B}R$, which can be computed in $\mathcal{O}(N^{\frac{3}{2}})$ time using a worst-case optimal join algorithm. The view V_B can be computed in linear time by looking up each tuple from V'_C in R . The views V'_C and V_A are constructed by marginalizing out one variable at a time in time $\mathcal{O}(N^{\frac{3}{2}})$ and $\mathcal{O}(N)$ time, respectively. Hence, the view tree construction takes $\mathcal{O}(N^{\frac{3}{2}})$ time.

□

We show an example in Section 4.3 that indicator projections can reduce the update time for cyclic queries.

4.2 Enumeration

The view trees constructed by the function τ for any access-top VO for Q_{\dagger} allow for constant-delay enumeration of the tuples in $Q(\mathcal{O}|\mathbf{i})$ given any tuple \mathbf{i} over the input variables \mathcal{I} . The enumeration relies on iterators with access patterns constructed over the materialized views.

4.2.1 View Iterators

We first introduce iterators for views. A *view iterator* can enumerate the tuples in a materialized view using the standard iterator interface. We define $\text{it}_V(\mathcal{O}|\mathcal{I})$ as a view iterator it over a view V with schema $\mathcal{O} \cup \mathcal{I}$, where \mathcal{O} is the *output schema* and \mathcal{I} is the *context schema*; the output schema \mathcal{O} and the context schema \mathcal{I} of a view iterator are not necessarily disjoint. We write $\text{it}_V(\mathcal{O})$ when \mathcal{I} is empty.

The view iterator implements an interface with three methods: $\text{open}(ctx)$, $\text{next}()$ and $\text{contains}(\mathbf{o})$.

- The $\text{open}(ctx)$ method initializes the iterator using the tuple ctx over \mathcal{I} as context. This method sets the range of the iterator to those \mathcal{O} -tuples that are consistent with ctx in V , that is, either paired with or part of ctx in V .
- The $\text{next}()$ method returns an \mathcal{O} -tuple consistent with ctx in V . It returns **EOF** when the \mathcal{O} -tuples in the range of the iterator are exhausted. The returned \mathcal{O} -tuples are distinct.
- The $\text{contains}(\mathbf{o})$ method checks whether the \mathcal{O} -tuple \mathbf{o} is consistent with ctx in V ; in other words, whether \mathbf{o} will be enumerated by the iterator.

All three methods operate in constant time, as per our computational model (cf. Chapter 2).

Example 4.7. Consider a materialized view $V(A, B)$. The iterator $\text{it}_V(A, B)$ enumerates the (A, B) -tuples in the view. The iterator $\text{it}_V(B|A)$ enumerates the distinct B -values paired with a given A -value in V . The iterator $\text{it}_V(B|A, B)$ takes as input a given (A, B) -tuple (a, b) and returns b if the tuple (a, b) exists in V ; otherwise, it returns **EOF**. The iterator $\text{it}_V(A)$ is invalid as the union of its output variable A and context schema \emptyset do not match the schema of V , i.e., $\{A\} \cup \emptyset \neq \{A, B\}$. □

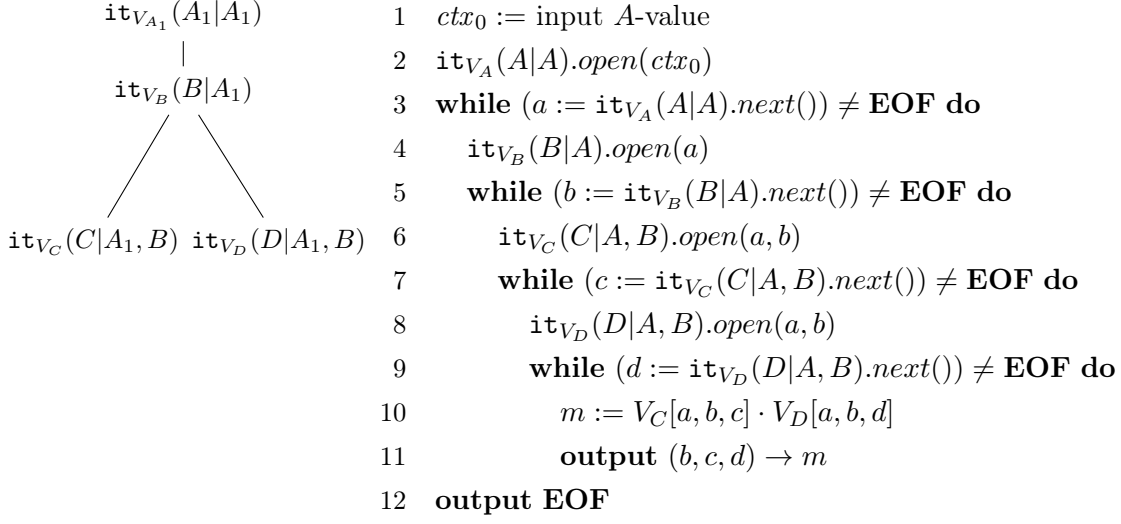


Figure 4.8: Left: View iterators created over the view tree for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ from Example 4.2; Right: Enumeration procedure for Q_1 using the created view iterators.

4.2.2 Enumeration Procedures

When the result of a query is in a single view, such as the query $Q_2(E|A_2) = T(A_2, E)$ from Example 4.2, we create a single view iterator over the view to enumerate the query result. When the result is distributed in multiple views in a view tree, we need to create a list of view iterators over these views. The result of the query is the concatenations of the output values enumerated by the created iterators.

We use the access-top variable order of the view tree to guide the construction of the iterators: we create iterators over the views that correspond to the free variables in the variable order and then generate an enumeration procedure for the query using the created iterators. We first give the intuition in the following example and then the algorithm.

Example 4.8. Consider the query $Q(B, C, D, E|A)$ from Example 4.2 and the two connected components $Q_1(B, C, D|A_1)$ and $Q_2(E|A_2)$ of its fracture. For the query Q_2 , we create the view iterator $it_{V_T}(E|A_2)$ to enumerate the values over E for a given A_2 -value.

For the query Q_1 , we create a list of view iterators. Figure 4.3 (middle) depicts the view tree for Q_1 . The view iterators are created following a top-down manner. At the root view V_{A_1} , we create $it_{V_{A_1}}(A_1|A_1)$ to check if a given input A -value exists in V_{A_1} . If it exists, the iterator returns the same A -value, which then serves as the context for the iterators created below. The iterator $it_{V_B}(B|A_1)$ at view V_B enumerates the B -values that are paired with a in V_B . Such (A_1, B) -values serve as the context for $it_{V_C}(C|A_1, B)$ and $it_{V_D}(D|A_1, B)$, which enumerate C - and respectively D -values. We skip creating

iterators over auxiliary views $V'_C(A_1, B)$ and $V'_D(A_1, B)$ because we already have iterators for A and B (these auxiliary views are created only for efficient update; they do not play a role in the enumeration). These iterators are then organized into nested loops as shown in the right of Figure 4.8. The enumeration procedure returns **EOF** when all the iterators are exhausted, i.e., all tuples have been enumerated. The multiplicity of an output tuple is the product of the multiplicities of its values in the views V_C and V_D .

The time needed to fetch the next value from each iterator is $\mathcal{O}(1)$; this is also the enumeration delay of the procedure.

We have now constant-delay enumeration procedures for the tuples in $Q_1(B, C, D|a)$ and the tuples in $Q_2(E|a)$ for any A -value a . We can enumerate with constant delay the tuples in $Q(B, C, D, E|a)$ as follows. We ask for the first tuple (b, c, d) in $Q_1(B, C, D|a)$ and then iterate over the distinct E -values in $Q_2(E|a)$. For each such E -value e , we report the tuple (b, c, d, e) . Then, we ask for the next tuple in $Q_1(B, C, D|a)$ and restart the enumeration over the tuples in $Q_2(E|a)$, and so on.

□

Nesting view iterators, as in Example 4.8, is valid, since the context schema of each iterator is subsumed by the input variables of the query and the output variables of preceding iterators. In other words, the iterators for the free variables are above those for the bound variables and the iterators for the input variables are above those for the output variables. This is guaranteed, since the view tree is built over the access-top variable order. The nesting order of the view iterators is not always unique; e.g., we can swap the two innermost loops in the procedure from Figure 4.8.

We next give the algorithm. Figure 4.9 shows the function `BUILDITERATORS` for building the list of view iterators for a given view tree of a CQAP query Q with access pattern $(\mathcal{O}|\mathcal{I})$. It recursively constructs the view iterators by traversing the view tree T in a top-down fashion. Consider the root view $V_X(\mathcal{X})$ of T constructed at variable X in the corresponding variable order. If $X \notin \mathcal{X}$, then V_X is an auxiliary view that allows for efficient maintenance under updates (c.f. Figure 4.2) but has no role in enumeration, thus we recur on its child (Lines 2-3). The function creates a view iterator with output variable X over V_X if X is a free variable; if X is an output variable, X is removed from the context schema, and the iterator enumerates the X -values that are consistent with the given tuple over \mathcal{X} in V_X ; otherwise, X is an input variable, X is kept in the context schema of the iterator, so the iterator serves to check whether the given value over X is consistent with the other given values in \mathcal{X} in V_X (Line 4). The function recursively creates iterators in each subtree and concatenates them into a list of iterators (Lines 5-6).

We then generate the enumeration procedure by organizing the view iterators into nested loops based on a pre-order traversal of the view tree. We open the iterators with

```

BUILDITERATORS(view tree  $T$ , access pattern  $(\mathcal{O}|\mathcal{I})$ ) : list of iterators

```

```

switch  $T$ :

```

```

 $R(\mathcal{Y})$    1  return [] // empty list

```

```

 $V_X(\mathcal{X})$   2  if  $X \notin \mathcal{X}$  or // skip auxiliary views and
/ \           $V_X$  is an indicator projection // skip auxiliary views and
 $T_1 \cdots T_k$  3  return BUILDITERATORS( $T_1, (\mathcal{O}|\mathcal{I})$ ) // skip auxiliary views and
indicator projections

```

```

4   $it_X = \begin{cases} [(\mathbf{new\ it}_{V_X}(X|\mathcal{X}))] & , \text{if } X \in \mathcal{I} \\ [(\mathbf{new\ it}_{V_X}(X|\mathcal{X} \setminus \{X\}))] & , \text{if } X \in \mathcal{O} \\ [] & , \text{otherwise} \end{cases}$  // empty list

```

```

5   $it_{child_i} = \text{BUILDITERATORS}(T_i, (\mathcal{O}|\mathcal{I})), \forall i \in [k]$ 
6  return  $it_X ++ it_{child_1} ++ \dots ++ it_{child_k}$  // concatenation of the lists

```

Figure 4.9: Create a list of view iterators with support for the access pattern $(\mathcal{O}|\mathcal{I})$ in a view tree T . We use brackets to denote a list. The operator $++$ concatenates two lists.

values from their ancestor views as context, thus ensuring they enumerate only those values guaranteed to be in the query output. Each concatenation of the outputs of the iterators forms the values of an output tuple. Since the *open* and *next* calls of the view iterators take constant time, the enumeration delay is constant.

Multiplicity Computation. Once we get an output tuple from the enumeration procedure as shown above, we need to compute the multiplicity of the tuple in the view tree. Figure 4.10 shows the COMPUTEM function for computing the multiplicity of a tuple \mathbf{t} in a view tree T . The tuple \mathbf{t} is over both the output and input variables of the query, i.e., the concatenation of the given tuple over the input variables and the enumerated values over the output variables.

The function traverses the view tree T based on a pre-order. If \mathbf{t} is in the root view V (Line 1), the function returns the multiplicity of \mathbf{t} in V (Line 2). Otherwise, i.e., the tuple \mathbf{t} is stored below V , possibly distributed in different branches (Line 3), the function recurs to each subtree and returns the product of their multiplicities (Lines 4-5). The computation time is determined by the number of views in the view tree, which is $\mathcal{O}(1)$.

4.2.3 Multiple View Trees

We discussed how to enumerate tuples from one view tree. In case of queries with several connected components $Q_1(\mathcal{O}_1 | \mathcal{I}_1), \dots, Q_n(\mathcal{O}_n | \mathcal{I}_n)$, we form a nesting chain for the enumeration from their view trees, as shown below:

COMPUTEM(view tree T , tuple \mathbf{t}): multiplicity

switch T :

$R(\mathcal{Y})$ 1 **return** $R[\mathbf{t}]$ // assert $\text{Sch}(\mathbf{t}) = \mathcal{Y}$

$V_X(\mathcal{X})$ 2 **if** $\mathcal{X} = \text{Sch}(\mathbf{t})$

 / \ 3 **return** $V[\mathbf{t}]$

$T_1 \cdots T_k$ 4 **else** // $\mathcal{X} \subsetneq \text{Sch}(\mathbf{t})$

 5 $\mathcal{V}_i :=$ variables in T_i

 6 **return** $\prod_{i \in [k]} \text{COMPUTEM}(T_i, \pi_{\mathcal{V}_i} \mathbf{t})$

Figure 4.10: For a tuple \mathbf{t} over the free variables of a CQAP query, compute the multiplicity of \mathbf{t} in the view tree T .

1 **foreach** $\mathbf{o}_1 \rightarrow m_1 \in Q_1(\mathcal{O}_1 | \mathbf{i}_1)$
2 ...

3 **foreach** $\mathbf{o}_n \rightarrow m_n \in Q_n(\mathcal{O}_n | \mathbf{i}_n)$
4 **report** $\mathbf{o}_1 \cdots \mathbf{o}_n \rightarrow m_1 \times \cdots \times m_n$

Consider a tuple \mathbf{i} over the input variables \mathcal{I} of Q . It holds $Q(\mathcal{O} | \mathbf{i}) = \times_{i \in [n]} Q_i(\mathcal{O}_i | \mathbf{i}_i)$ where $\mathbf{i}_i[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q . The multiplicity of an output tuple is the product of the multiplicities of the output tuples from Q_1, \dots, Q_n . Since we can enumerate with constant delay the tuples in $Q_i(\mathcal{O}_i | \mathbf{i}_i)$ for any tuple \mathbf{i}_i over \mathcal{I}_i with constant delay, we can enumerate the tuples in $Q(\mathcal{O} | \mathbf{i})$ with constant delay by nesting the enumeration procedures for $Q_1(\mathcal{O}_1 | \mathbf{i}_1), \dots, Q_n(\mathcal{O}_n | \mathbf{i}_n)$.

4.3 Update

We now explain how to update the view trees constructed by the function τ in Figure 4.2. Consider a single-tuple update $\delta R = \{\mathbf{x} \rightarrow m\}$ to an input relation R ; m is positive in case of insertion and negative in case of deletion.

We update each view tree that has an atom $R(\mathcal{X})$ at a leaf using the function $\text{APPLY}(T, \delta R)$ in Figure 4.11: it propagates the update δR in the view tree T from the leaf R to the root view. For each view on this path, it updates the view result with the change computed using the standard delta rules [23]. If T does not refer to R , the procedure has no effect.

The update δR may affect indicator projections $I_Z R$. A new single-tuple update $\delta I_Z R = \{\mathbf{x}[\mathcal{Z}] \rightarrow k\}$ to $I_Z R$ is triggered in the following two cases. If δR is an insertion

APPLY(view tree T , update δR) : delta view															
switch T :															
$K^{sig'}(\mathcal{X})$	<ol style="list-style-type: none"> 1 if $K^{sig'} = R$ 2 $R(\mathcal{X}) := R(\mathcal{X}) + \delta R(\mathcal{X})$ 3 return δR 4 return \emptyset 														
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="padding: 5px;">$V(\mathcal{X})$</td> <td style="padding: 5px;">5 $V_i(\mathcal{X}_i) := \text{root of } T_i, \text{ for } i \in [k]$</td> </tr> <tr> <td style="padding: 5px;">/ \</td> <td style="padding: 5px;">6 if $\exists j \in [k]$ s.t. $R \in T_j$</td> </tr> <tr> <td style="padding: 5px;">$T_1 \cdots T_k$</td> <td style="padding: 5px;">7 $\delta V_j := \text{APPLY}(T_j, \delta R)$</td> </tr> <tr> <td></td> <td style="padding: 5px;">8 $\delta V(\mathcal{X}) := \text{join of } V_1(\mathcal{X}_1), \dots, \delta V_j(\mathcal{X}_j), \dots, V_k(\mathcal{X}_k)$</td> </tr> <tr> <td></td> <td style="padding: 5px;">9 $V(\mathcal{X}) := V(\mathcal{X}) + \delta V(\mathcal{X})$</td> </tr> <tr> <td></td> <td style="padding: 5px;">10 return δV</td> </tr> <tr> <td></td> <td style="padding: 5px;">11 return \emptyset</td> </tr> </table>		$V(\mathcal{X})$	5 $V_i(\mathcal{X}_i) := \text{root of } T_i, \text{ for } i \in [k]$	/ \	6 if $\exists j \in [k]$ s.t. $R \in T_j$	$T_1 \cdots T_k$	7 $\delta V_j := \text{APPLY}(T_j, \delta R)$		8 $\delta V(\mathcal{X}) := \text{join of } V_1(\mathcal{X}_1), \dots, \delta V_j(\mathcal{X}_j), \dots, V_k(\mathcal{X}_k)$		9 $V(\mathcal{X}) := V(\mathcal{X}) + \delta V(\mathcal{X})$		10 return δV		11 return \emptyset
$V(\mathcal{X})$	5 $V_i(\mathcal{X}_i) := \text{root of } T_i, \text{ for } i \in [k]$														
/ \	6 if $\exists j \in [k]$ s.t. $R \in T_j$														
$T_1 \cdots T_k$	7 $\delta V_j := \text{APPLY}(T_j, \delta R)$														
	8 $\delta V(\mathcal{X}) := \text{join of } V_1(\mathcal{X}_1), \dots, \delta V_j(\mathcal{X}_j), \dots, V_k(\mathcal{X}_k)$														
	9 $V(\mathcal{X}) := V(\mathcal{X}) + \delta V(\mathcal{X})$														
	10 return δV														
	11 return \emptyset														

Figure 4.11: Updating views in a view tree T for a single-tuple update δR to relation part R . If R is a leaf of T , the function updates R and its ancestor views in a bottom-up fashion and returns the change of the root view. Otherwise, the empty set is returned.

and $\mathbf{x}[\mathcal{Z}]$ is a value not already in $\pi_{\mathcal{Z}}R$, then the new update is triggered with $k = 1$. If δR is a deletion and $\pi_{\mathcal{Z}}R$ does not contain $\mathbf{x}[\mathcal{Z}]$ after applying the update to R , then the new update is triggered with $k = -1$. This update is propagated up to the root of each view tree, like for δR .

The function $\text{APPLY}(T, \delta R)$ in Figure 4.11 propagates the update δR in the view tree T from the leaf R to the root view. For each view on this path, it updates the view result with the change computed using the standard delta rules [23]. If T does not refer to R , the procedure has no effect.

4.3.1 Dynamic Width

Recall that the time to compute a view V_X is $\mathcal{O}(N^{\mathbf{w}})$, where $\mathbf{w} = \rho_Q^*(\{X\} \cup \text{dep}_{\omega}(X))$. In case of an update to a relation or indicator $R(\mathcal{Y})$ under V_X in the view tree, the variables in \mathcal{Y} are set to constants. The time to update V_X is then $\mathcal{O}(N^{\delta})$, where $\delta = \rho_Q^*(\{X\} \cup \text{dep}_{\omega}(X)) \setminus \mathcal{Y}$.

We next give the formal definition of the dynamic width in Definition 4.9. The definition of static width is copied here for convenience: the blue texts are the differences between the two definitions.

Definition 4.9. *The static width $\mathbf{w}(\omega)$ and dynamic width $\delta(\omega)$ of an extended variable*

order ω are:

$$\begin{aligned} w(\omega) &= \max_{X \in \text{vars}(\omega)} \rho_Q^*(\{X\} \cup \text{dep}_\omega(X)) \\ \delta(\omega) &= \max_{X \in \text{vars}(\omega)} \max_{R(\mathcal{Y}) \in \text{atoms}(\omega_X)} \rho_Q^*(\{X\} \cup \text{dep}_\omega(X) \setminus \mathcal{Y}) \end{aligned}$$

where $\text{atoms}(\omega_X)$ is the set of atoms (base relations and indicators) in the subtree of ω rooted at X .

The dynamic width is defined similar to the static width, with one simplification: we consider every case of an atom $R(\mathcal{Y})$ being replaced by a single-tuple update, so its variables \mathcal{Y} are all set to constants and can be discarded in the computation of the fractional edge cover number.

Example 4.10. Figure 4.3 (right) shows the delta view tree for the view tree to the left under a single-tuple update $\delta R(a, b, c)$ to R . We update the relation $R(A, B, C)$ with $\delta R(a, b, c)$ in constant time. The ancestor views of δR (in blue) are the deltas of the corresponding views, computed by propagating δR from the leaf to the root. They can also be effected in constant time. Overall, maintaining the view tree under a single-tuple update to any relation takes $O(1)$ time.

Consider now the delta view tree in Figure 4.4 (right) obtained from the view tree to its left under the single-tuple update $\delta R(a, b, c)$. We update $V_B(A_1, B, C, D)$ with $\delta V_B(a, b, c, D) = \delta R(a, b, c), S(a, b, D)$ in $O(N)$ time, since there are at most N D -values paired with (a, b) in S . We then update the views $V_D, V_C,$ and V_{A_1} in $O(1)$ time. Updates to S are handled analogously. Overall, maintaining the view tree under a single-tuple update to any input relation takes $O(N)$ time.

Consider now a single-tuple update δR to R for the query in Example 4.6, the base relation R and the indicator projection $I_{A,B}R$ are affected by the update. We compute two delta view trees shown on the right in Figure 4.7 for changes in R and respectively $I_{A,B}R$. In the delta view tree for changes to R (the left one), computing the delta $\delta V_B(a, b) = V'_C(a, b), \delta R(a, b)$ requires a constant lookup in V'_C ; computing $\delta V_A(a) = \delta V_B(a, b)$ takes constant time. In the delta view tree for changes to $I_{A,B}R$ (the right one), computing the delta $\delta V_C(a, b, C) = S(b, C), T(C, a), \delta I_{A,B}R(a, b)$ requires intersecting the C -values that are paired with b in S and with a in T , which takes $O(N)$ time; computing $\delta V'_C(a, b) = \delta V_C(a, b, C)$ requires aggregating away $O(N)$ C -values; computing δV_B and δV_A takes constant time. Overall, a single-tuple update to R takes $O(N)$ time. The delta view trees for changes to S and T are analogous. Hence, the update time of the query Q is $O(N)$. \square

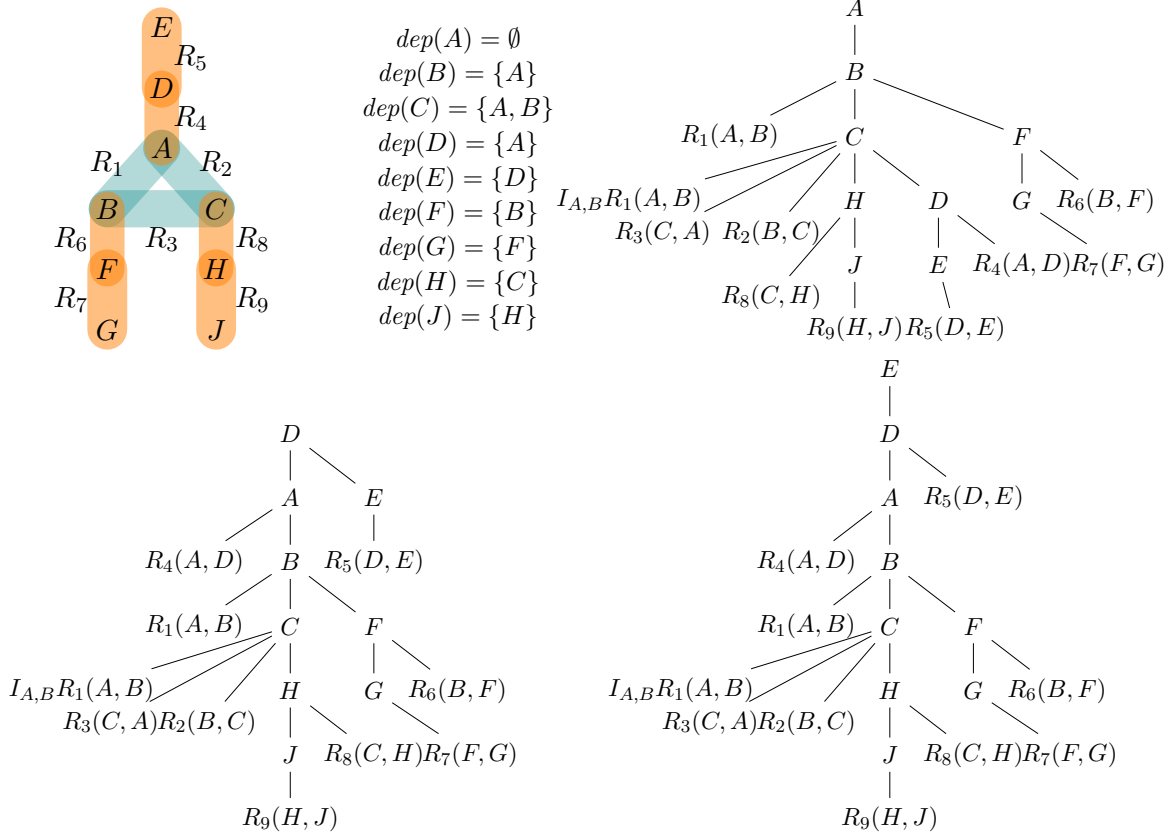


Figure 4.12: Top left: The hypergraph of the query Q in Example 4.11. Remaining three: the optimal access-top variable orders of the query Q with the roots A , D and E , respectively. All other access-top variable orders are analogous to these three variable orders. The dependent sets of the two variable orders in the second row are omitted.

In the following example, we show that the indicator projections can reduce the update time for a query no matter which variable order is chosen as the strategy for the dynamic evaluation.

Example 4.11. Consider the following query:

$$Q(A, B, C, D, E, F, G, H, J \mid \cdot) = R_1(A, B), R_2(B, C), R_3(C, A), R_4(A, D), R_5(D, E), \\ R_6(B, F), R_7(F, G), R_8(C, H), R_9(H, J)$$

It is a triangle query with three tails. Its fracture is same as the query itself. Figure 4.12 shows the hypergraph (top-left) of the query and three access-top variable orders of the query. They are the optimal variable orders that are rooted at variables A , D and E . That is, other variable orders rooted at the corresponding variable do not admit smaller static and dynamic widths. Since the query is symmetric, the optimal variable orders rooted at other variables are analogous to these three variable orders.

Consider the variable order in the top right of Figure 4.12. The indicator projection $I_{A,B}R_1$ is created under variable C to form a cycle with R_2 and R_3 and thus to reduce the time to compute and update the view $V_C(A, B, C) = I_{A,B}R_1(A, B), R_2(B, C), R_3(C, A)$ at C . The time to update V_C is $\mathcal{O}(N^\delta)$, where δ is the fractional edge cover number of $\{A, B, C\}$ minus the schema of an atom below C . If we choose the atom to be $R_9(H, J)$, the remaining variables are still $\{A, B, C\}$. With the indicator projection $I_{A,B}R_1$, the fractional edge cover number is $\rho^*(A, B, C) = \frac{3}{2}$ (by assigning a weight of $\frac{1}{2}$ to each atom $I_{A,B}R_1, R_3$ and R_2). Without $I_{A,B}R_1$, the fractional edge cover number is $\rho^*(A, B, C) = 2$. Hence, the indicator projection $I_{A,B}R_1$ reduces the time to update V_C from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{\frac{3}{2}})$.

The two variable orders in the second row of Figure 4.12 are similar to the aforementioned variable order: they all have the variables A, B , and C in one root-to-leaf path, followed by the atom R_9 , which has no intersection with A, B , and C . The indicator projection $I_{A,B}R_1$ created under variable C forms cycles with R_2 and R_3 and thus reduces the time to update the view $V_C(A, B, C)$ at C from $\mathcal{O}(N^2)$ to $\mathcal{O}(N^{\frac{3}{2}})$ in the same way. Hence, the indicator projections can reduce the update time of the query Q for all variable orders. \square

4.4 Complexity Analysis

We next discuss the complexities of the preprocessing, enumeration and update stages of our dynamic evaluation approach for arbitrary CQAP queries. Recall the theorem that states the complexities of our approach:

Theorem 3.4. *Given a CQAP with static width \mathbf{w} and dynamic width δ and a database of size N , the query can be evaluated with $\mathcal{O}(N^{\mathbf{w}})$ preprocessing time, $\mathcal{O}(N^\delta)$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

Before proving it, we discuss how we select the static and dynamic widths of CQAP queries. We are interested in the dynamic evaluation of queries, so we consider the update time of the queries more important than the preprocessing time. We consider the standard lexicographic ordering \leq on pairs of dynamic and static widths: $(\delta_1, \mathbf{w}_1) \leq (\delta_2, \mathbf{w}_2)$ if $\delta_1 \leq \delta_2$ or $\delta_1 = \delta_2$ and $\mathbf{w}_1 \leq \mathbf{w}_2$. Given a set \mathcal{S} of variable orders, we define $\min_{\omega \in \mathcal{S}} (\delta(\omega), \mathbf{w}(\omega)) = (\delta, \mathbf{w})$ such that $\forall \omega \in \mathcal{S} : (\delta, \mathbf{w}) \leq (\delta(\omega), \mathbf{w}(\omega))$. The dynamic and static width of a CQAP query Q is the minimum pair over all access-top variable orders of the fracture Q_{\dagger} :

Definition 4.12. *The dynamic width $\delta(Q)$ and static width $\mathbf{w}(Q)$ of a CQAP query Q are:*

$$(\delta(Q), \mathbf{w}(Q)) = \min_{\omega \in \text{acc-top}(Q_{\dagger})} (\delta(\omega), \mathbf{w}(\omega))$$

Intuitively, Definition 4.12 first minimizes for the dynamic width and then for the static width. It remains an open question whether the variable order of a query Q that gives the minimum dynamic width is the same variable order that gives the minimum static width.

We are now ready to prove Theorem 3.4.

Proof. Given a CQAP Q with static width $\mathbf{w}(Q) = \mathbf{w}$ and dynamic width $\delta(Q) = \delta$ and a database of size N , we show that our approach presented in Section 4 evaluates Q with $\mathcal{O}(N^{\mathbf{w}})$ preprocessing time, $\mathcal{O}(N^{\delta})$ update time, and $\mathcal{O}(1)$ enumeration delay. Consider an access-top variable order ω for the fracture Q_{\dagger} with $\mathbf{w}(\omega) = \mathbf{w}$ and $\delta(\omega) = \delta$. In the following, we analyze each of the three stages preprocessing, update, and enumeration.

Preprocessing. Without loss of generality, assume that ω consists of a single tree. Otherwise, we do the analysis below for each of the constantly many trees in ω . The preprocessing stage consists of materializing the view tree $T = \tau(\omega)$ where τ is the function given in Figure 4.2. We show by induction on the structure of T that every node in T can be materialized in $\mathcal{O}(N^{\mathbf{w}})$ time.

Base Case: Each leaf atom or indicator projection in T can be materialized in linear time.

Induction Step: Consider an auxiliary view V'_X in T for $X \in \text{vars}(\omega)$. By construction, this view results from its single child view V_X by marginalizing out variable X . By induction hypothesis, the view V_X can be computed in $\mathcal{O}(N^{\mathbf{w}})$ time, hence its size has the same complexity bound. We can compute V'_X by scanning over the tuples in V_X and maintaining during the scan the count $|\sigma_{S=s} V_X|$ for each tuple \mathbf{s} in $\pi_S V_X$. This can be done in $\mathcal{O}(N^{\mathbf{w}})$ overall time.

Consider now a view $V_X(\mathcal{S})$ in T with $X \in \text{vars}(\omega)$ and $\mathcal{S} = \{X\} \cup \text{dep}_{\omega}(X)$. Let $V_1(\mathcal{S}_1), \dots, V_k(\mathcal{S}_k)$ be the child nodes of V_X . Each child node can be a view, an atom, or an indicator projection. By induction hypothesis, the child nodes of V_X can be materialized in $\mathcal{O}(N^{\mathbf{w}})$ time. Consider any variable Y that occurs in the schemas of at least two child nodes of V_X . This means that $Y \in \mathcal{S} = \{X\} \cup \text{dep}_{\omega}(X)$. Hence, any variable that does not occur in \mathcal{S} cannot be a join variable for the child views of V_X . We first marginalize out the variables in the child views that do not occur in \mathcal{S} . This can be done in $\mathcal{O}(N^{\mathbf{w}})$ time. Let $V'_1(\mathcal{S}'_1), \dots, V'_k(\mathcal{S}'_k)$ be the resulting views. The view V_X can now be rewritten as $V_X(\mathcal{S}) = V'_1(\mathcal{S}'_1), \dots, V'_k(\mathcal{S}'_k)$. Since the views V'_1, \dots, V'_k result from joining the atoms

(and indicator projections) in ω , we can upper-bound the computation time for V_X by $\mathcal{O}(N^p)$ where $p = \rho_Q^*(\mathcal{S})$ [59]. It follows from the definition of \mathbf{w} that p is upper-bounded by \mathbf{w} . We conclude that the view V_X can be computed in $\mathcal{O}(N^{\mathbf{w}})$ time.

Enumeration. Assume that \mathcal{I} and \mathcal{O} are the input and respectively output variables of Q and let \mathcal{I}_\dagger be the input variables of Q_\dagger . We show that for any input tuple \mathbf{i} over \mathcal{I} , the tuples in $Q(\mathcal{O}|\mathbf{i})$ can be enumerated with constant delay using the view trees constructed in the preprocessing stage. Let $\omega_1, \dots, \omega_n$ be the trees in ω (ω is a forest) and $\tau(\omega_1) = T_1, \dots, \tau(\omega_n) = T_n$ the view trees constructed from $\omega_1, \dots, \omega_n$. For $j \in [n]$, let $Q_j(\mathcal{O}_j|\mathcal{I}_j)$ with $\mathcal{O}_j = \mathcal{O} \cap \text{vars}(\omega_j)$ and $\mathcal{I}_j = \mathcal{I}_\dagger \cap \text{vars}(\omega_j)$ be the CQAP query that joins the atoms appearing at the leaves of T_j . We first explain how for any $j \in [n]$ and \mathbf{i}_j over \mathcal{I}_j , the tuples in $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ can be enumerated with constant delay using the view tree T_j . Since the view tree is constructed following an access-top variable order, it holds that all views V_X where X is free (input) are above the views V_Y where Y is bound (output). Hence, we can construct a list of the view iterators over the view tree by the function BUILDITERATORS (Figure 4.9), and organize them into nested loops, where the iterators are opened with values from their ancestor views as context, thus ensuring they enumerate only those values guaranteed to be in the query output. Since the *open* and *next* calls of the view iterators take constant time, the enumeration delay is constant.

Assume that we can enumerate the tuples in $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ with constant delay for any $j \in [n]$ and tuple \mathbf{i}_j over \mathcal{I}_j . Consider a tuple \mathbf{i} over \mathcal{I} . It holds $Q(\mathcal{O}|\mathbf{i}) = \times_{j \in [n]} Q_j(\mathcal{O}_j|\mathbf{i}_j)$ where $\mathbf{i}_j[X'] = \mathbf{i}[X]$ if $X = X'$ or X is replaced by X' when constructing the fracture of Q . We enumerate the tuples in $Q(\mathcal{O}|\mathbf{i})$ by interleaving the enumeration procedures for $Q_1(\mathcal{O}_1|\mathbf{i}_1), \dots, Q_n(\mathcal{O}_n|\mathbf{i}_n)$. That is, we first retrieve the first complete tuple \mathbf{o}_j from $Q_j(\mathcal{O}_j|\mathbf{i}_j)$ for each $j \in [n]$ and report $\mathbf{o}_1 \cdots \mathbf{o}_n$. Then, we iterate over the remaining tuples in $Q_n(\mathcal{O}_n|\mathbf{i}_n)$. For each such tuple \mathbf{o}'_n , we report $\mathbf{o}_1 \cdots \mathbf{o}'_n$. After all tuples in $Q_n(\mathcal{O}_n|\mathbf{i}_n)$ are exhausted, we move to the next tuple in $Q_{n-1}(\mathcal{O}_{n-1}|\mathbf{i}_{n-1})$ and restart the enumeration for $Q_n(\mathcal{O}_n|\mathbf{i}_n)$, and so on.

We conclude that the time to report the first tuple in $Q(\mathcal{O}|\mathbf{i})$, the time to report a next tuple after the previous one is reported, and the time to signalize the end of the enumeration after the last tuple is reported is constant.

Updates. We show that the view trees constructed in the preprocessing stage can be updated in $\mathcal{O}(N^\delta)$ time under single-tuple updates to the base relations. Consider a single-tuple update to a base relation R . We first update each view tree referring to an atom of the form $R(\mathcal{X})$. Updating a view tree amounts to computing the deltas of the views on the path from $R(\mathcal{X})$ to the root of the view tree. We have shown above

that for each variable X , the views V_X and V'_X can be materialized in $\mathcal{O}(N^p)$ time where $p = \rho_Q^*(\{X\} \cup \text{dep}_\omega(X))$. Since the update fixes the values in \mathcal{X} , the time to compute the delta of these views under the update becomes $\mathcal{O}(N^d)$ where $d = \rho_Q^*((\{X\} \cup \text{dep}_\omega(X)) \setminus \mathcal{X})$. A single-tuple update to R can trigger a single-tuple update to each indicator view of the form $I_{\mathcal{Z}}(R(\mathcal{Z}))$. Analogously to the reasoning above, we conclude that the time to compute the deltas of the views under such updates is $\mathcal{O}(N^d)$ where $d = \rho_Q^*((\{X\} \cup \text{dep}_\omega(X)) \setminus \mathcal{Z})$. It follows from the definition of the dynamic width δ of ω , that in both cases the exponent d is upper-bounded by δ . This implies that the overall update time is $\mathcal{O}(N^\delta)$. \square

Chapter 5

Trade-Offs in Dynamic Evaluation for CQAP Queries with Hierarchical Fractures

For triangle CQAP queries and CQAP queries with hierarchical fractures, we introduce a dynamic approach that uncovers the trade-offs between the preprocessing time, update time, and enumeration delay. We focus on the queries with hierarchical fractures in this chapter, and the triangle CQAP queries in the next chapter.

The approach partitions the data into heavy and light parts and employ adaptive evaluation strategies to them. The trade-offs are achieved by tuning the threshold for the heavy-light partitioning. The complexities of our dynamic evaluation technique for CQAP queries with hierarchical fractures is stated in Theorem 3.5.

In this chapter, we first give the formal definition of the data partitioning (Section 5.1), and then the preprocessing (Section 5.2), enumeration (Section 5.3), and update (Section 5.4) stages of our approach. We then prove Theorem 3.5 and discuss the complexities in Section 5.5. At the end of the chapter, we show the optimality of our approach in Section 5.6.

We consider in the following a fixed CQAP query $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$ which is hierarchical, and a database of size N . To simplify the presentation, we assume that the fracture has one connected component, which implies that the fracture is the query itself and the variable order of the query is a single tree. Otherwise, we can evaluate each connected component separately. Whenever we refer to a (canonical or access-top) variable order of a query Q , we mean the (canonical or access-top) variable order extended with the atoms of Q .

5.1 Data Partitioning

We partition relations based on the frequencies of their values. For a database \mathcal{D} , relation $R \in \mathcal{D}$ over schema \mathcal{X} , schema $\mathcal{S} \subset \mathcal{X}$, and threshold θ , a *partition* of R on \mathcal{S} with threshold θ is a pair $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ of relations such that

- (union) $R(\mathbf{x}) = R^{\mathcal{S} \rightarrow H}(\mathbf{x}) + R^{\mathcal{S} \rightarrow L}(\mathbf{x})$ for each $\mathbf{x} \in \text{Dom}(\mathcal{X})$
- (domain partition) $\pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow H} \cap \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow L} = \emptyset$
- (heavy part) $\forall \mathbf{t} \in \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow H}, \exists K \in \mathcal{D}: |\sigma_{\mathcal{S}=\mathbf{t}}K| \geq \frac{1}{2}\theta$
- (light part) $\forall \mathbf{t} \in \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow L}$ and $\forall K \in \mathcal{D}: |\sigma_{\mathcal{S}=\mathbf{t}}K| < \frac{3}{2}\theta$

We call $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ a *strict partition* of R on \mathcal{S} with threshold θ if it satisfies the union and domain partition conditions and the following strict versions of the heavy and light part conditions:

- (strict heavy part) $\forall \mathbf{t} \in \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow H}, \exists K \in \mathcal{D}: |\sigma_{\mathcal{S}=\mathbf{t}}K| \geq \theta$
- (strict light part) $\forall \mathbf{t} \in \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow L}$ and $\forall K \in \mathcal{D}: |\sigma_{\mathcal{S}=\mathbf{t}}K| < \theta$

The relation $R^{\mathcal{S} \rightarrow H}$ is called *heavy* and the relation $R^{\mathcal{S} \rightarrow L}$ is called *light* on the partition key \mathcal{S} . Due to the domain partition, the relations $R^{\mathcal{S} \rightarrow H}$ and $R^{\mathcal{S} \rightarrow L}$ are disjoint. For $|\mathcal{D}| = N$ and a strict partition $(R^{\mathcal{S} \rightarrow H}, R^{\mathcal{S} \rightarrow L})$ of R on \mathcal{S} with threshold $\theta = N^\epsilon$ for $\epsilon \in [0, 1]$, we have: (1) $\forall \mathbf{t} \in \pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow L} : |\sigma_{\mathcal{S}=\mathbf{t}}R^{\mathcal{S} \rightarrow L}| < \theta = N^\epsilon$; and (2) $|\pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow H}| \leq \frac{N}{\theta} = N^{1-\epsilon}$. The first bound follows from the strict light part condition. In the second bound, $\pi_{\mathcal{S}}R^{\mathcal{S} \rightarrow H}$ refers to the \mathcal{S} -tuples that have high degrees (more or equal to θ) in some relation in the database. The database can contain at most $\frac{N}{\theta}$ such tuples; otherwise, the database size would exceed N .

A relation R can be partitioned on multiple partition keys $\mathcal{S}_1, \dots, \mathcal{S}_n$, which are not necessarily disjoint. We write $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n}$ to denote the relation part obtained after partitioning $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_{n-1} \rightarrow s_{n-1}}$ on \mathcal{S}_n , where $s_i \in \{H, L\}$ for $i \in [n]$. The domain of $R^{\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n}$ is the intersection of the domains of $R^{\mathcal{S}_i \rightarrow s_i}$, for $i \in [n]$. We call $\mathcal{S}_1 \rightarrow s_1, \dots, \mathcal{S}_n \rightarrow s_n$ the *HL-signature* for R .

Example 5.1. Consider a relation R with the schema (A, B, C) and the partition keys $\{A\}$ and $\{A, B\}$. The relation part $R^{A \rightarrow H, AB \rightarrow L}$ is obtained by first partitioning R on $\{A\}$ and then on $\{A, B\}$, and it contains the tuples with A -values that have high degrees in R and (A, B) -values that have low degrees in R . The HL-signature for $R^{A \rightarrow H, AB \rightarrow L}$ is $A \rightarrow H, AB \rightarrow L$. \square

5.2 Preprocessing

For preprocessing, we first partition the base relations into heavy and light parts based on the degrees of the values. This decomposes the query over the input relations into

a union of queries over the heavy and light parts; we call them the *skew-aware queries*. We then construct a set of variable orders extended with the relation parts such that each variable order corresponds to an evaluation strategy for a skew-aware query. For variable orders over light relation parts, we follow the general approach from Section 4.1 and construct view trees from access-top variable orders. For variable orders involving heavy relation parts, we construct view trees from variable orders that are not access-top, thus yielding non-constant enumeration delay but better preprocessing and update times. This trade-off is controlled by the parameter ϵ .

In the following of this section, we first give a function that turns canonical variable orders into optimal access-top ones (Section 5.2.1), and then explain how to obtain different variable orders from the canonical variable order of the hierarchical query by using the above function (Section 5.2.2). In Section 5.2.3 we describe the construction of view trees from variable orders. Proofs of the propositions in this section are given in Section 5.2.4.

5.2.1 From Canonical to Access-Top Variable Orders

Given a canonical variable order ω of $Q(\mathcal{O}|\mathcal{I})$, the function $\text{ACC-TOP}(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 5.1 returns an access-top variable order for Q with optimal static and dynamic width. The function proceeds recursively on the structure of ω . At a variable X , the function selects a set \mathcal{D} of variables from the subtree ω_X rooted at X based on the type of X : 1) if X is an input variable, the function sets $\mathcal{D} = \emptyset$; 2) if X is an output variable, the function defines \mathcal{D} to be the input variables in ω_X , and 3) if X is bound, the function sets \mathcal{D} to be the free variables in ω_X (Line 3). The function then takes out \mathcal{D} from ω_X and puts them on top of X (Lines 4-6). Line 5 makes sure the input variables are put on top of the output variables.

The deletion of a set \mathcal{D} of variables from a variable order ω is implemented by the function $\Delta(\omega, \mathcal{D})$ in Figure 5.2. The function traverses recursively over all variables in ω . If a variable X is not included in \mathcal{D} , the function does not change the structure of ω (Lines 3-4). In case $X \in \mathcal{D}$ and X has a parent Y , it appends the child trees of X to the variable Y (Lines 5-6). If $X \in \mathcal{D}$ and X has no parent, the child trees of X become independent (Line 7).

Example 5.2. *Figure 5.3 (left and middle) shows the hypergraphs of the query*

$$Q(B, C, D, E \mid A) = R(A, B, C), S(A, B, D), T(A, E)$$

and of its fracture

$$Q_{\dagger}(B, C, D, E \mid A_1, A_2) = R(A_1, B, C), S(A_1, B, D), T(A_2, E).$$

ACC-TOP(variable order ω , access pattern $(\mathcal{O}|\mathcal{I})$) : variable order

switch ω :

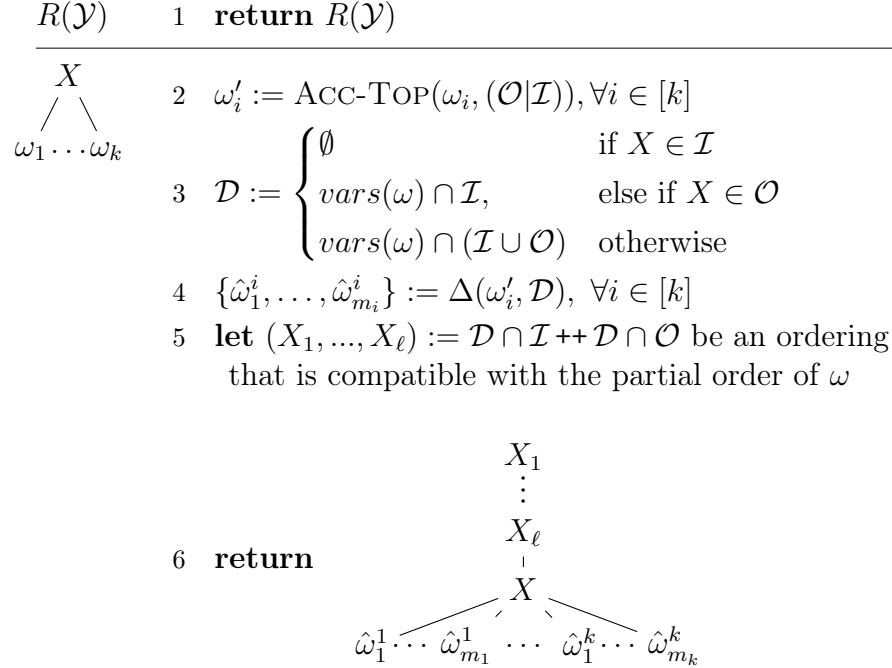


Figure 5.1: Construction of an access-top variable order from a canonical variable order ω of a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture. The operator $++$ concatenates the tuples $\mathcal{D} \cap \mathcal{I}$ and $\mathcal{D} \cap \mathcal{O}$. The function $\Delta(\omega', \mathcal{D})$, defined in Figure 5.2, deletes the variables in \mathcal{D} from the variable order ω' .

The fracture is hierarchical, free-dominant and input-dominant. Hence, Q and Q_\dagger are in CQAP_0 . The fracture can be decomposed into two queries $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E|A_2) = T(A_2, E)$, whose bodies are the two connected components in the fracture; Figure 5.3 (right) depicts the access-top variable orders for the two queries. They are same as the canonical variable orders of the two queries.

□

Example 5.3. Consider the query

$$Q(C, D | E) = R(A, B, C), S(A, B, D), T(A, E).$$

Figure 5.4 (left) shows the hypergraphs of the query. Its fracture is same as itself, which is hierarchical but not free-dominant. Figure 5.4 (middle) depicts the canonical variable order of the query. Figure 5.4 (right) depicts the access-top variable order for the query. The free variables C, D and E sit on top of the bound variables A and B . The input variable E sits on top of the output variables C and D .

□

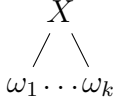
$\Delta(\text{variable order } \omega, \text{ variables } \mathcal{D})$: set of variable orders	
switch ω :	
$R(\mathcal{Y})$	1 return $\{R(\mathcal{Y})\}$
	2 $\{\omega_1^i, \dots, \omega_{m_i}^i\} := \Delta(\omega_i, \mathcal{D}), \forall i \in [k]$
	3 if $X \notin \mathcal{D}$
	4 return $\left\{ \begin{array}{c} X \\ \omega_1^1 \dots \omega_{m_1}^1 \dots \omega_1^k \dots \omega_{m_k}^k \end{array} \right\}$
	5 else if X has parent Y
	6 return $\left\{ \begin{array}{c} Y \\ \omega_1^1 \dots \omega_{m_1}^1 \dots \omega_1^k \dots \omega_{m_k}^k \end{array} \right\}$
	7 else return $\{\omega_1^1, \dots, \omega_{m_1}^1, \dots, \omega_1^k, \dots, \omega_{m_k}^k\}$

Figure 5.2: Deletion of a set \mathcal{D} of variables from a variable order ω . If $X \in \mathcal{D}$ and X has a parent Y , the child trees of X are appended to Y . If $X \in \mathcal{D}$ and X has no parent, the child trees of X become independent.

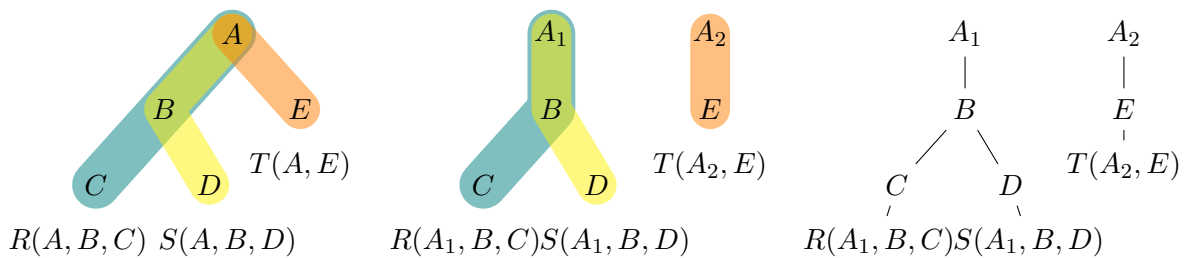


Figure 5.3: Left and middle: Hypergraphs of the query (left) and its fracture on input variable A (middle two) used in Example 5.2. Right two: The access-top variable orders returned by ACC-TOP in Figure 5.1, which are the same as the canonical variable orders.

The following proposition states that the access-top variable order constructed by ACC-TOP for a CQAP query Q has the optimal static and dynamic widths.

Proposition 5.4. *Given a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture and a canonical variable order ω for Q , $\text{ACC-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top variable order for Q with static width $w(Q)$ and dynamic width $\delta(Q)$.*

5.2.2 Variable Orders Describing Evaluation Strategies

Each variable order of a CQAP query stands for an evaluation strategy for the query. We next show how to derive a set of variable orders from the canonical variable order of a query such that each variable order depicts an evaluation strategy of one skew-aware

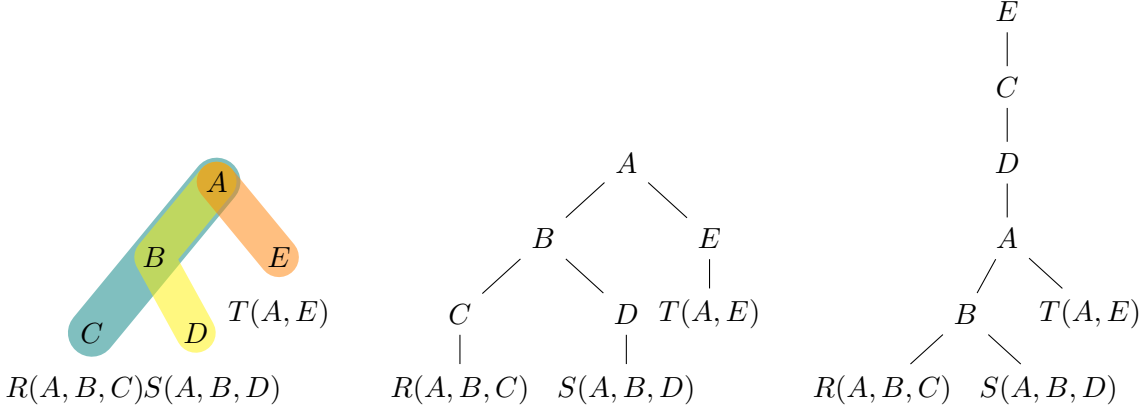


Figure 5.4: Left: Hypergraph of the query and its fracture used in Example 5.3. Middle: The canonical variable order of the query. Right: The access-top variable order returned by ACC-TOP in Figure 5.1.

query.

We start with a high-level explanation of the construction. Consider the canonical variable order ω of $Q(\mathcal{O}|\mathcal{I})$ and a subtree ω' of ω rooted at a variable X . The *induced query* $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ is defined over the join of the atoms at the leaves of ω' . The \mathcal{I}_X consists of the input variables in ω' and the root path of X . The set \mathcal{I}_O contains the output variables in ω' . Let ω'_{at} be an access-top variable order of $Q_X(\mathcal{O}_X|\mathcal{I}_X)$. We discuss the cases when Q_X is CQAP₀ or not CQAP₀.

If Q_X is CQAP₀, we use ω'_{at} for the evaluation of Q_X . The view tree following ω'_{at} can be constructed in linear time, can be updated in constant time and allows for constant-delay enumeration of the result of Q_X .

If Q_X is not CQAP₀, in this case, ω' contains a bound or output variable Y such that Q_Y is not CQAP₀. If X is not this variable Y , we recursively process the subtrees of ω' until we reach Y ; otherwise, i.e., if X is this variable Y , we distinguish two cases based on the degree of values over $\text{anc}_w(X) \cup \{X\}$. In the light case, we construct the view tree following the variable order ω'_{at} . This view tree can be constructed and maintained under updates efficiently, since the values over $\text{anc}_w(X) \cup \{X\}$ have bounded degree. In the heavy case, we use the variable order ω' . The view tree following ω' allows for constant update time and an enumeration delay that depends on the number of distinct values over $\text{anc}_w(X) \cup \{X\}$. Since these values have high degree, the number of distinct such values is bounded, which ensure efficient enumeration delay.

Given a canonical variable order ω of $Q(\mathcal{O}|\mathcal{I})$, the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 5.5 returns the set of all variable orders for Q obtained from ω . The atoms at the leaves of these variable orders are labelled by HL-signatures. When constructing view trees following these variable orders, these atoms will be materialized with corresponding

 $\Omega(\text{variable order } \omega, \text{access pattern } (\mathcal{O}|\mathcal{I})) : \text{set of variable orders}$

switch ω :

```

 $R^{sig}(\mathcal{Y})$  1 return  $\{R^{sig}(\mathcal{Y})\}$ 


---


 $X$  2  $key := \text{anc}_\omega(X) \cup \{X\}$ 
 $\begin{matrix} / \\ \backslash \end{matrix}$  3  $\mathcal{I}_X := \text{anc}_\omega(X) \cup (\mathcal{I} \cap \text{vars}(\omega))$ 
 $\omega_1 \dots \omega_k$  4  $\mathcal{O}_X := \mathcal{O} \cap \text{vars}(\omega)$ 
5  $Q_X(\mathcal{O}_X|\mathcal{I}_X) := \text{join of } \text{atoms}(\omega)$ 
6 if  $Q_X(\mathcal{O}_X|\mathcal{I}_X)$  is CQAP0
7 return  $\{\text{ACC-TOP}(\omega, (\mathcal{O}|\mathcal{I}))\}$ 
8 else if  $X \in \mathcal{I}$  or  $(X \in \mathcal{O} \text{ and } \text{vars}(\omega) \cap \mathcal{I} = \emptyset)$ 
9 return  $\left\{ \begin{matrix} X \\ / \backslash \\ \omega'_1 \cdot \omega'_k \end{matrix} \mid \omega'_i \in \Omega(\omega_i, (\mathcal{O}|\mathcal{I})), \forall i \in [k] \right\}$ 
10 else
11  $htrees := \left\{ \begin{matrix} X \\ / \backslash \\ \omega'_1 \cdot \omega'_k \end{matrix} \mid \omega'_i \in \Omega(\omega_i^{key \rightarrow H}, (\mathcal{O}|\mathcal{I})), \forall i \in [k] \right\}$ 
12  $ltree := \text{ACC-TOP}(\omega^{key \rightarrow L}, (\mathcal{O}|\mathcal{I}))$ 
13 return  $htrees \cup \{ltree\}$ 


---



```

Figure 5.5: Construction of a set of variable orders from a canonical variable order ω of a CQAP $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture. Each constructed variable order corresponds to an evaluation strategy of some part of the query result. The variable order $\omega^{key \rightarrow s}$ for $s \in \{H, L\}$ has the structure of ω but the HL-signature of each atom is extended by $key \rightarrow s$.

relation parts. That is, an atom $R^{sig}(\mathcal{Y})$ with $\mathcal{S} \rightarrow s \in sig$ will be materialized by a part of relation R that is heavy on \mathcal{S} if $s = H$ and light on \mathcal{S} if $s = L$. We assume that the atoms in the initial canonical variable order ω passed as input to the function Ω are labelled by the empty HL-signature \emptyset .

We now describe the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in more detail. The function proceeds recursively on the structure of ω and considers at each variable X , the induced query $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ (Line 4). If Q_X is CQAP₀, the function returns an access-top variable order constructed by the function $\text{ACC-TOP}(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 5.1 (Lines 5-6). If X is an input variable, or it is an output variable and ω does not contain any input variable, the query Q_X can be evaluated efficiently given that the induced queries defined at the children of X are evaluated efficiently. Hence, the function recursively computes a set of variable orders for each child tree of X . For each combination of these variable orders, it builds a new variable order where X is on top of the child variable orders (Lines 7-8). Otherwise,

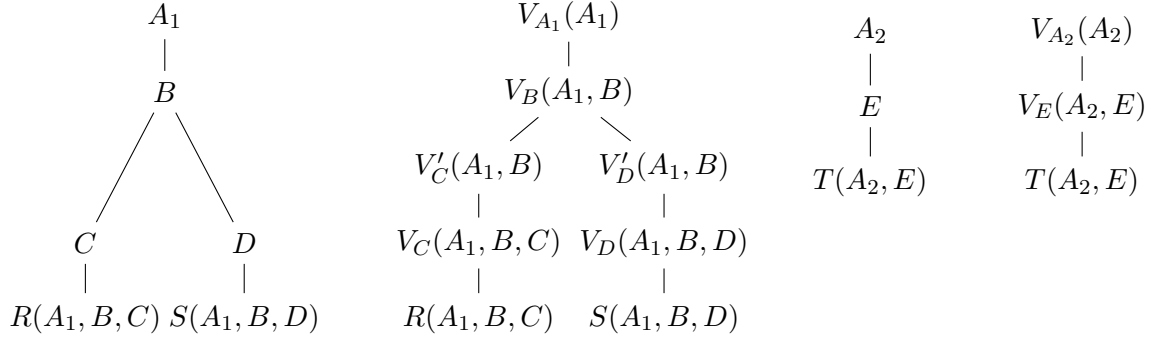


Figure 5.6: Variable orders constructed for $Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D)$ and $Q_2(E|A_2) = T(A_2, E)$ in Example 5.2 and their corresponding view trees.

if X is bound or an output variable and ω contains input variables, the function creates two evaluation strategies for Q_X based on the degree of values over $X \cup \text{anc}(X)$. For the values over $X \cup \text{anc}(X)$ that are *heavy*, i.e., the degrees of the values are above a given threshold, the function treats X as an input variable and proceeds recursively to resolve further variables located below X in the variable order and to potentially fork into more strategies (Line 10). For the values over $X \cup \text{anc}(X)$ that are *light*, the function constructs an access-top variable order for ω (Line 10).

Example 5.5. Consider the CQAP₀ query

$$Q(B, C, D, E | A) = R(A, B, C), S(A, B, D), T(A, E)$$

and the two sub-queries in its fracture:

$$Q_1(B, C, D|A_1) = R(A_1, B, C), S(A_1, B, D) \text{ and } Q_2(E|A_2) = T(A_2, E)$$

from Example 5.2. Figure 5.6 (left and middle right) shows the variable orders, i.e., the evaluation strategies, for the variable orders of the two queries returned by Ω . Since Q is in CQAP₀, the variable orders for evaluation are exactly the access-top variable orders of the two queries. \square

Example 5.6. Consider the query

$$Q(C, D | E) = R(A, B, C), S(A, B, D), T(A, E)$$

from Example 5.3. The canonical variable order of the query is the same as in Figure 5.4 (middle). Figure 5.7 shows on the left column the three variable orders returned by the function Ω in Figure 5.5.

We explain the construction of the variable orders returned by Ω . We start from the root A in the canonical variable order. The residual query $Q_A(\mathcal{O}_A|\mathcal{I}_A)$ is equal to $Q(\mathcal{O}|\mathcal{I})$.

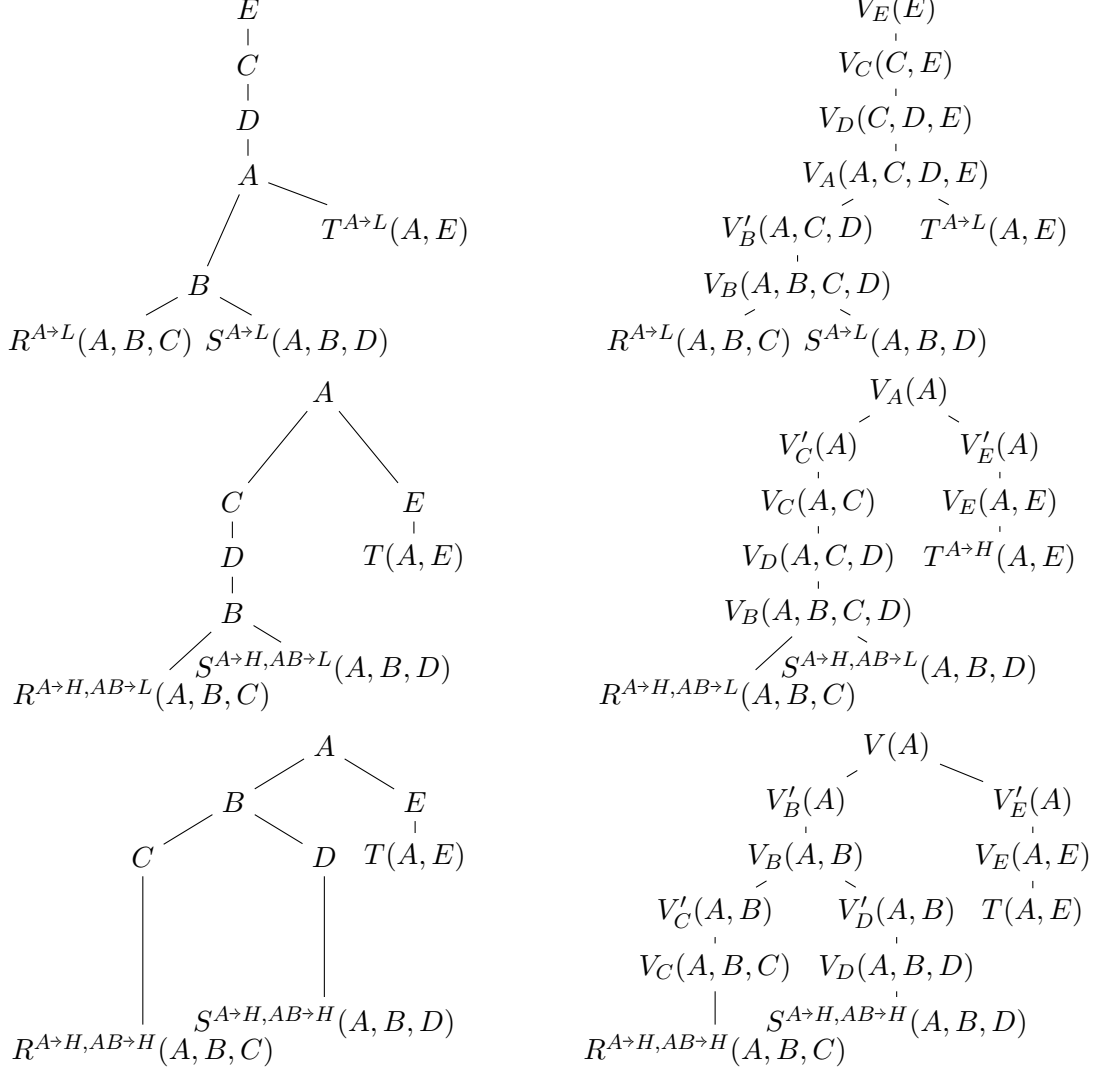


Figure 5.7: Left column: The variable orders constructed for the query $Q(C, D | E) = R(A, B, C), S(A, B, D), T(A, E)$ in Example 5.3. Right column: The view trees constructed following the variable orders on the left.

Since Q_A is not $CQAP_0$ and A is bound, we distinguish two cases based on the degree of A -values: In the light case for A , we create an access-top variable order for Q_A whose leaves are the light parts of the input relations partitioned on A (top left in Figure 5.7).

In the heavy case for A , we recursively process the subtrees of A in the canonical variable order and treat A as an input variable. The residual query $Q_E(\cdot | A, E) = T(A, E)$ is $CQAP_0$, thus we create an access-top variable order for Q_E whose leaf is $T^{A \to H}(A, E)$, i.e., the heavy part of T partitioned on A (middle left and bottom left variable orders in Figure 5.7). The residual query $Q_B(C, D | A) = R(A, B, C), S(A, B, D)$, however, is not $CQAP_0$. Since B is bound, we further distinguish two new cases based on the degree of the values over (A, B) . In the light case for (A, B) , we construct a variable order

VIEWTREES(canonical variable order ω , access pattern $(\mathcal{O}|\mathcal{I})$) : view trees

1 **return** $\{\tau(\omega') \mid \omega' \in \Omega(\omega, (\mathcal{O}|\mathcal{I}))\}$

Figure 5.8: Construction of all view trees for a canonical variable order ω of a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture.

whose leaves are $R^{A \rightarrow H, AB \rightarrow L}$ and $S^{A \rightarrow H, AB \rightarrow L}$, i.e., the parts of R and S that are heavy on A and light on (A, B) (middle left variable order in Figure 5.7). In the heavy case for (A, B) , we process the subtrees of B considering B as an input variable (bottom left variable order in Figure 5.7). The residual queries $Q_C(C|A, B) = R(A, B, C)$ and $Q_D(D|A, B) = S(A, B, D)$, are CQAP₀. Overall, we create three variable orders. \square

5.2.3 View Trees Encoding the Query Result

The translation from variable orders for CQAP queries with hierarchical fractures into view trees is the same as in our approach for arbitrary CQAP queries (Section 4.1). Given a variable order ω , the function $\tau(\omega)$ in Figure 4.2 returns a view tree following ω . The function VIEWTREES($\omega, (\mathcal{O}|\mathcal{I})$) in Figure 5.8 returns the set of all view trees for a query $Q(\mathcal{O}|\mathcal{I})$ with canonical variable order ω . For each variable order ω' returned by $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ from Figure 5.5, the function creates the corresponding view tree by calling $\tau(\omega')$ from Figure 4.2.

Materializing a view tree consists of computing the relation parts at the leaves and computing the joins defined by the views in the view tree. The preprocessing phase for a query $Q(\mathcal{O}|\mathcal{I})$ with canonical variable order ω consists of materializing all view trees in VIEWTREES($\omega, (\mathcal{O}|\mathcal{I})$).

Example 5.7. Figure 5.6 (middle left and right) shows the view trees constructed from the corresponding variable orders. Each variable in the variable order is mapped to a view in the view tree, e.g., B is mapped to $V_B(A_1, B)$, where $\{B, A_1\} = \{B\} \cup \text{dep}(B)$. The views V'_C, V'_D and V_{A_1} are auxiliary views that allow for efficient maintenance under updates to R and S : they marginalize out one variable from their child views. The view V_B is the intersection of V'_C and V'_D . Hence all views can be computed in linear time. \square

Example 5.8. Consider again the query

$$Q(B, C, D, E \mid A) = R(A, B, C), S(A, B, D), T(A, E)$$

from Example 5.3. Figure 5.7 shows next to each variable order for the query, the corresponding view tree. The query Q has static width 3. Computing the relation parts at the

leaves of the view trees takes time linear in N , where N is the database size. We explain how the views in the view trees can be computed in $\mathcal{O}(N^{1+2\epsilon})$ time.

Consider the variable order and view tree in the top row of Figure 5.7. At variable B , we create the view $V_B(A, B, C, D) = R^{A \rightarrow L}(A, B, C), S^{A \rightarrow L}(A, B, D)$, which joins the light parts of R and S partitioned on A . Computing $V_B(A, B, C, D)$ takes $\mathcal{O}(N^{1+\epsilon})$ time: For each value (a, b, c) in $R^{A \rightarrow L}$, we iterate over at most N^ϵ (a, b, d) values in $S_L^{A \rightarrow L}$. Since B has siblings in the variable order, we also create the auxiliary view $V'_B(A, C, D)$ that aggregates away B in time linear in the size of V'_B . At A , we compute $V_A(A, C, D, E)$ in $\mathcal{O}(N^{1+2\epsilon})$ time: We iterate over $\mathcal{O}(N^{1+\epsilon})$ values (a, c, d) in $V'_B(A, C, D)$ and for each such value, iterate over at most N^ϵ values (a, e) in $T^{A \rightarrow L}$. We do not need to create an auxiliary view that aggregates away A , since A does not have siblings in the variable order. At each variable above A , we create a view that aggregates away the variable below. Aggregating a variable away takes time linear in the size of the view. Hence, computing $V_D(C, D, E)$ takes $\mathcal{O}(N^{1+2\epsilon})$ time, computing $V_C(C, E)$ takes $\mathcal{O}(N^{1+\epsilon})$ time, and computing $V_E(E)$ takes $\mathcal{O}(N)$ time. Overall, materializing this view tree takes $\mathcal{O}(N^{1+2\epsilon})$ time.

We now consider the variable order and view tree in the second row. At B , we create the view $V_B(A, B, C, D) = R^{A \rightarrow H, AB \rightarrow L}(A, B, C), S^{A \rightarrow H, AB \rightarrow L}(A, B, D)$ in $\mathcal{O}(N^{1+\epsilon})$ time: For each value (a, b, c) in $R^{A \rightarrow H, AB \rightarrow L}$, we iterate over at most N^ϵ values (a, b, d) in $S^{A \rightarrow H, AB \rightarrow L}$. At E , we build $V_E(A, D, E)$ that aggregates away B in $\mathcal{O}(N^{1+\epsilon})$ time. At D , we build $V_D(A, D)$ and the auxiliary view $V'_D(A)$ in linear time. The other views can be computed in linear time by aggregating away variables and applying semi-join reduction. Hence, materializing the view tree in the second row takes $\mathcal{O}(N^{1+\epsilon})$ time.

Materializing the view tree in the bottom row takes linear time: All views are computed by aggregating away variables and applying semi-join reduction, which takes linear time.

Overall, we materialize the three view trees for Q in $\mathcal{O}(N^{1+2\epsilon})$ time. \square

The set of view trees constructed for a CQAP query with a hierarchical fracture in the preprocessing phase encode exactly the query.

Proposition 5.9. *Let $\{T_1, \dots, T_k\}$ be the set of view trees in $\text{VIEWTREES}(\omega, (\mathcal{O}|\mathcal{I}))$ for a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture and the canonical variable order ω for Q . Let $Q_{T_i}(\mathcal{O}|\mathcal{I})$ be the query defined by the conjunction of the leaf atoms in T_i . Then, $Q(\mathcal{O}|\mathcal{I}) \equiv \bigcup_{i \in [k]} Q_{T_i}(\mathcal{O}|\mathcal{I})$.*

Given a CQAP $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture, static width w , the preprocessing time of our approach is given by the time to materialize the view trees in $\text{VIEWTREES}(\omega, \mathcal{O}, \mathcal{I})$. The time to materialize these view tree is $\mathcal{O}(N^{1+(w-1)\epsilon})$.

Proposition 5.10. *Given a CQAP query with a hierarchical fracture, static width w , a database of size N , and $\epsilon \in [0, 1]$, the view trees in the preprocessing stage can be computed in $\mathcal{O}(N^{1+(w-1)\epsilon})$ time.*

5.2.4 Proofs

In this section we give the missing proofs of the formal statements in Section 5.2.

Proof of Proposition 5.4

Proposition 5.4. *Given a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture and a canonical variable order ω for Q , $\text{ACC-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top variable order for Q with static width $w(Q)$ and dynamic width $\delta(Q)$.*

Before proving Proposition 5.4, we introduce some useful notation. Let ω be a canonical variable order of a hierarchical CQAP query. Let \mathcal{F} , \mathcal{I} , and \mathcal{O} be the free, input, and respectively output variables of the query, and X a variable in ω . We denote by ω_X the subtree of ω rooted at X and by Q_X a query that joins the atoms at the leaves of ω_X . (and whose set of free variables is arbitrary).

We first give the intuition behind the proof. Recall the static and dynamic widths of a query Q are defined over access-top variable orders of Q . We next provide an alternative definition of the static and dynamic widths of variable orders: instead of the access-top variable orders, they are defined over *canonical* variable orders of Q , by simulating the ACC-TOP function. We then show these alternative definitions are equivalent to the original definitions.

We start by the alternative definitions of the static and dynamic widths. The following measures ξ and κ express the static and the dynamic width of ω_X without referring to access-top variable orders.

$$\xi(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in \text{bound}(\omega_X) \\ Z \in \text{out}(\omega_X)}} \{\rho_{Q_X}^*(\text{vars}(\omega_Y) \cap \mathcal{F}), \rho_{Q_X}^*(\text{vars}(\omega_Z) \cap \mathcal{I})\}$$

$$\kappa(\omega_X, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in \text{bound}(\omega_X) \\ Z \in \text{out}(\omega_X)}} \max_{R(\mathcal{Y}) \in \text{atoms}(\omega_Y)} \{\rho_{Q_X}^*((\text{vars}(\omega_Y) \cap \mathcal{F}) - \mathcal{Y}), \rho_{Q_X}^*((\text{vars}(\omega_Z) \cap \mathcal{I}) - \mathcal{Y})\}$$

If ω_X does not contain any bound or output variable, we have

$$\xi(\omega_X, \mathcal{I}, \mathcal{O}) = \kappa(\omega_X, \mathcal{I}, \mathcal{O}) = 0.$$

The next lemma expresses the static and dynamic width of the variable orders returned by the function ACC-TOP in terms of the measures ξ and κ .

Lemma 5.11. *Given a canonical variable order ω of a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture, a variable X in ω , and the induced query Q_X at variable X , the function $\text{ACC-TOP}(\omega_X, (\mathcal{I}|\mathcal{O}))$ constructs a variable order ω' such that $\omega^t = (\text{anc}_\omega(X) \circ \omega')^1$ is an access-top variable order for Q_X with $\mathbf{w}(\omega^t) = \max\{1, \xi(\omega_X, \mathcal{I}, \mathcal{O})\}$ and $\delta(\omega^t) = \kappa(\omega_X, \mathcal{I}, \mathcal{O})$.*

Proof. The function ACC-TOP traverses the given canonical variable order and pulls up free variables such that the resulting variable order becomes access-top. More precisely, if a variable X is bound and contains free variables in its subtree, the function puts all free variables below X on top of X such that the input variables are above the output variables. If the variable X is an output variable and contains input variables in its subtree, it puts all input variables that are under X on top of X .

If ω neither contains a bound variable above a free one nor an output variable above a bound one, the variable order remains unchanged. Since a canonical variable order has static width 1 and dynamic width 0, the statement in the lemma holds in this case.

Assume now that ω contains at least one bound variable above a free variable or at least one output variable above an input variable. Consider an arbitrary bound variable X in ω that has free variables in its subtree. Let \mathcal{F} be the set of free variables under X . Due to the structure of canonical variable orders, all variables in \mathcal{F} depend on X . By moving the variables in \mathcal{F} on top of X , the set \mathcal{F} is added to the dependency set of X in the resulting variable order ω^t . Hence, the fractional edge cover number of $\{X\} \cup \text{dep}_{\omega^t}(X)$ is $\rho^*(\{X\} \cup \mathcal{F})$. The dependency set of a variable Y in \mathcal{F} can only decrease since the set of the variables from Y to the root decreases. The dependency set of a variable Y below X changes if it contained a variable from \mathcal{F} in its subtree that is now positioned on top of Y . However, the fractional edge cover number of $\{Y\} \cup \text{dep}_{\omega^t}(Y)$ is upper-bounded by the fractional edge cover number of $\{X\} \cup \text{dep}_{\omega^t}(X)$.

In case X is an output variable that has a set \mathcal{V} of input variables in its subtree, the reasoning is similar. The fractional edge cover number of $\{X\} \cup \text{dep}_{\omega^t}(X)$ is $\rho^*(\{X\} \cup \mathcal{V})$ and upper-bounds the fractional edge cover numbers at the other variables in the resulting variable order ω^t .

Hence, the static width of ω^t is determined by the largest set of variables that is moved on top of a single variable by the function ACC-TOP .

For the dynamic width of ω^t , the reasoning is completely analogous. The dynamic width of ω^t is given by the largest set of variables that is moved on top of a single variable X after removing the variables of any atom containing X . \square

We are ready to prove Proposition 5.4.

¹In ω^t , variables $\text{anc}_\omega(X)$ are organized as a linked list on top of ω' , as shown in Figure 5.1 (Line 6).

Proof of Proposition 5.4. Consider a CQAP query Q whose fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I})$ is hierarchical. Let $\mathcal{F} = \mathcal{I} \cup \mathcal{O}$ and \mathbf{w} and δ be the static and respectively dynamic width of Q . By the definition of static and dynamic width, Q_{\dagger} has static width \mathbf{w} and dynamic width δ . Let ω be the canonical variable order of Q_{\dagger} . Without loss of generality, assume that Q_{\dagger} contains at least one atom with non-empty schema. Otherwise, ACC-TOP returns the set of atoms in Q_{\dagger} , which is already an optimal access-top variable order for Q_{\dagger} . Assume also that ω consists of a single connected component. Otherwise, we apply the same reasoning for each connected component. By Lemma 5.11, $\text{ACC-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top variable order ω^t for Q_{\dagger} with static width $\max\{1, \xi(\omega_X, \mathcal{I}, \mathcal{O})\}$ and dynamic width $\kappa(\omega_X, \mathcal{I}, \mathcal{O})$. We first show:

$$\max\{1, \xi(\omega, \mathcal{I}, \mathcal{O})\} \leq \mathbf{w} \quad (5.1)$$

First, assume that $\xi(\omega, \mathcal{I}, \mathcal{O}) = 0$. This means $\max\{1, \xi(\omega, \mathcal{I}, \mathcal{O})\} = 1$. Since Q_{\dagger} contains at least one atom with non-empty schema, we have $\mathbf{w} \geq 1$. Thus, Inequality (5.1) holds. Now, let $\xi(\omega, \mathcal{I}, \mathcal{O}) = \ell \geq 1$. We show that $\mathbf{w} \geq \ell$. It follows from $\xi(\omega, \mathcal{I}, \mathcal{O}) = \ell$ that at least one of the following two cases holds:

- Case (1.1): ω contains a bound variable Y such that $\rho_Q^*(\mathcal{F}') = \ell$, where $\mathcal{F}' = \text{vars}(\omega_Y) \cap \mathcal{F}$
- Case (1.2): ω contains an output variable Y such that $\rho_Q^*(\mathcal{I}') = \ell$, where $\mathcal{I}' = \text{vars}(\omega_Y) \cap \mathcal{I}$.

We first consider Case (1.1). By construction, the inner nodes of each root-to-leaf path of a canonical variable order are the variables of an atom. Hence, for each variable $Z \in \mathcal{F}'$, there is an atom in Q_{\dagger} that contains both Y and Z . This means that Y and Z depend on each other, i.e., all variables in \mathcal{F}' depend on Y . Let ω' be an arbitrary access-top variable order for Q_{\dagger} . Since all variables in \mathcal{F}' depend on Y , each of them is on a root-to-leaf path with Y . Since Y is bound and the variables in \mathcal{F}' are free, the set \mathcal{F}' is included in $\text{anc}_{\omega'}(Y)$. Thus, $\mathcal{F}' \subseteq \text{dep}_{\omega'}(Y)$. This means $\rho^*(\{Y\} \cup \text{dep}_{\omega'}(Y)) \geq \ell$, which implies $\mathbf{w}(\omega') \geq \ell$. It follows $\mathbf{w} \geq \ell$.

The reasoning for Case (1.2) is analogous. In any access-top variable order $\omega' = (T, \text{dep}_{\omega'})$ for Q_{\dagger} , all variables in \mathcal{I}' is included in $\text{anc}_{\omega'}(Y)$. Hence, $\mathcal{I}' \subseteq \text{dep}_{\omega'}(Y)$, which means $\rho^*(\{Y\} \cup \text{dep}_{\omega'}(Y)) \geq \ell$. This implies $\mathbf{w}(\omega') \geq \ell$, thus, $\mathbf{w} \geq \ell$.

It follows that the static width of the access-top variable order $\text{ACC-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ is $\mathbf{w}(Q)$.

Following similar steps, we can show:

$$\kappa(\omega, \mathcal{I}, \mathcal{O}) \leq \delta \quad (5.2)$$

Let $\kappa(\omega, \mathcal{I}, \mathcal{O}) = k$. We show that $\delta \geq k$. The definition of $\kappa(\omega, \mathcal{I}, \mathcal{O})$ implies that one of the following two cases hold:

- Case (2.1): ω contains a bound variable Y and an atom $R(\mathcal{Y})$ containing Y such that $\rho_Q^*(\mathcal{F}' - \mathcal{Y}) = k$, where $\mathcal{F}' = \text{vars}(\omega_Y) \cap \mathcal{F}$
- Case (2.2): ω contains an output variable Y and an atom $R(\mathcal{Y})$ containing Y such that $\rho_Q^*(\mathcal{I}' - \mathcal{Y}) = k$, where $\mathcal{I}' = \text{vars}(\omega_Y) \cap \mathcal{I}$.

We consider Case (2.1). Let $\omega' = (T, \text{dep}_{\omega'})$ be an arbitrary access-top variable order for Q_{\dagger} . The atom $R(\mathcal{Y})$ is included in $\text{atoms}(\omega'_Y)$, since it contains Y . All variables in \mathcal{F}' depend on Y . Since Y is bound and the variables in \mathcal{F}' are free, the set $\mathcal{F}' - \mathcal{Y}$ is included in $\text{anc}_{\omega'}(Y)$. Hence, $\mathcal{F}' - \mathcal{Y} \subseteq \text{dep}_{\omega'}(Y)$. This implies that $\rho^*((\{Y\} \cup \text{dep}_{\omega'}(Y)) - \mathcal{Y}) \geq k$, i.e., $\delta(\omega') \geq k$. It follows $\delta \geq k$.

To show Case (2.2), we reason analogously. We just treat the output variables like the bound variables and input variables like the free variables in Case (2.1).

Overall, we conclude that given a CQAP query Q and its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I})$, the function $\text{ACC-TOP}(\omega, (\mathcal{I}|\mathcal{O}))$ constructs an access-top variable order with static width $w(Q_{\dagger}) = w(Q)$ and dynamic width $\delta(Q_{\dagger}) = \delta(Q)$. \square

Proof of Proposition 5.9

Proposition 5.9. *Let $\{T_1, \dots, T_k\}$ be the set of view trees in $\text{VIEWTREES}(\omega, (\mathcal{O}|\mathcal{I}))$ for a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture and the canonical variable order ω for Q . Let $Q_{T_i}(\mathcal{O}|\mathcal{I})$ be the query defined by the conjunction of the leaf atoms in T_i . Then, $Q(\mathcal{O}|\mathcal{I}) \equiv \bigcup_{i \in [k]} Q_{T_i}(\mathcal{O}|\mathcal{I})$.*

Proof. The procedure VIEWTREES calls Ω to construct from the input canonical variable order ω a set of variable orders $\omega_1, \dots, \omega_k$ and constructs the set of view trees T_1, \dots, T_k following the variable orders. The variable order ω_i and the corresponding view tree T_i for $i \in [k]$ have the same leaf atoms. Let ω' be a subtree of ω rooted at a variable $X \in \text{vars}(\omega)$. We define $Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) = \bowtie_{R(\mathcal{X}) \in \text{atoms}(\omega')} R(\mathcal{X})$ to be the query defined by the conjunction of the leaf atoms in ω' .

The proof is by induction over the structure of the variable order ω . We show that for any subtree ω' rooted at X of ω , it holds:

$$Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) \equiv \bigcup_{\omega'' \in \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))} Q_{\omega''}(\mathcal{O}_X|\mathcal{I}_X), \quad (5.3)$$

where $\mathcal{O}_X = \mathcal{O} \cap \text{vars}(\omega')$ and $\mathcal{I}_X = \text{anc}(X) \cup (\mathcal{I} \cap \text{vars}(\omega'))$. This completes the proof.

Base case: If ω' is an atom, the procedure Ω returns that atom and the base case holds trivially.

Inductive step: Assume that ω' has subtrees $\omega'_1, \dots, \omega'_k$. Let $key = \mathbf{anc}(X) \cup \{X\}$, $\mathcal{I}_X = \mathbf{anc}(X) \cup (\mathcal{I} \cap \mathit{vars}(\omega'))$, and $\mathcal{O}_X = \mathcal{O} \cap \mathit{vars}(\omega')$. The procedure Ω distinguishes the following cases:

Case 1: $Q_X(\mathcal{O}_X|\mathcal{I}_X)$ is $CQAP_0$ (Figure 5.5 Line 6). The procedure $\Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))$ constructs an access-top variable order with leaves exactly the atoms of ω' . This implies Equivalence 5.3.

Case 1 does not hold and ($X \in \mathcal{I}$ or ($X \in \mathcal{O}$ and $\mathit{vars}(\omega') \cap \mathcal{I} = \emptyset$)) (Figure 5.5 Line 8). The procedure $\Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))$ constructs recursively a set of variable orders for each subtree in $\omega'_1, \dots, \omega'_k$. Using the induction hypothesis, we rewrite as follows:

$$\begin{aligned}
Q_{\omega'}(\mathcal{O}_X|\mathcal{I}_X) &= \bowtie_{i \in [k]} Q_{\omega'_i}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \\
&\stackrel{\text{IH}}{\equiv} \bowtie_{i \in [k]} \left(\bigcup_{\omega'' \in \Omega(\omega'_i, (\mathcal{O}_{X'}|\mathcal{I}_{X'}))} Q_{\omega''}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \right) \\
&\equiv \bigcup_{\forall i \in [k]: \omega''_i \in \Omega(\omega'_i, (\mathcal{O}_{X'}|\mathcal{I}_{X'}))} \bowtie_{i \in [k]} Q_{\omega''_i}(\mathcal{O}_{X'}|\mathcal{I}_{X'}) \\
&= \bigcup_{T \in \Omega(\omega', (\mathcal{O}_X|\mathcal{I}_X))} Q_T(\mathcal{O}_X|\mathcal{I}_X),
\end{aligned}$$

where X' is the root of ω' , $\mathcal{O}_{X'} = \mathcal{O} \cap \mathit{vars}(\omega')$ and $\mathcal{I}_{X'} = \mathbf{anc}(X') \cup (\mathcal{I} \cap \mathit{vars}(\omega'))$.

Cases 1 and 2 do not hold. (Figure 5.5 Line 10). The procedure Ω creates the variable orders $htrees \cup \{ltree\}$ defined as follows:

- $ltree = \text{ACC-TOP}(\omega^{key \rightarrow L}, (\mathcal{O}_X|\mathcal{I}_X))$, where $\omega^{key \rightarrow L}$ has the same structure as ω' but each atom is replaced by its part that is light on key ;
- $htrees$ are same as the variable orders built in the previous case except each atom is replace by a part that is heavy on key .

If a relation is partitioned on a set key of variables, then the parts of relation that are light and heavy on key are disjoint and together form the relation. This drive the following equivalence. For simplicity, we skip the schemas of queries:

$$\bigcup_{\forall i \in [k]: T_i \in \Omega(\omega'_i, (\mathcal{O}|\mathcal{I}))} \bowtie_{i \in [k]} Q_{T_i} \equiv Q_{ltree} \cup \bigcup_{\forall i \in [k]: T_i \in \Omega(\omega_i^{key \rightarrow H}, (\mathcal{O}|\mathcal{I}))} Q_{T_i} \quad (5.4)$$

Using the induction hypothesis, we obtain:

$$\begin{aligned}
Q_{\omega'} &= \bowtie_{i \in [k]} Q_{\omega'_i} \stackrel{\text{IH}}{\equiv} \bowtie_{i \in [k]} \left(\bigcup_{\omega'' \in \Omega(\omega'_i, (\mathcal{O}|\mathcal{I}))} Q_{\omega''} \right) \\
&\equiv \bigcup_{\forall i \in [k]: \omega''_i \in \Omega(\omega'_i, (\mathcal{O}|\mathcal{I}))} \bowtie_{i \in [k]} Q_{\omega''_i} \\
&\stackrel{(5.4)}{\equiv} Q_{ltree} \cup \bigcup_{\forall i \in [k]: \omega''_i \in \Omega(\omega_i^{key \rightarrow H}, (\mathcal{O}|\mathcal{I}))} Q_{\omega''_i} \\
&= Q_{ltree} \cup \bigcup_{T \in htrees} Q_T = \bigcup_{T \in \Omega(\omega', (\mathcal{O}|\mathcal{I}))} Q_T
\end{aligned}$$

□

Proof of Proposition 5.10

Proposition 5.10. *Given a CQAP query with a hierarchical fracture, static width \mathbf{w} , a database of size N , and $\epsilon \in [0, 1]$, the view trees in the preprocessing stage can be computed in $\mathcal{O}(N^{1+(\mathbf{w}-1)\epsilon})$ time.*

The proof uses the auxiliary Lemma 5.12 given below. We first explain how Proposition 5.10 is implied by Lemma 5.12. Consider a CQAP query Q with static width \mathbf{w} and hierarchical fracture Q_{\dagger} and an $\epsilon \in [0, 1]$. In the preprocessing stage, we apply for each connected component $Q'_{\dagger}(\mathcal{O}|\mathcal{I})$ of Q_{\dagger} the following steps. Let ω be the canonical variable order of Q'_{\dagger} . First, we call the function $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ in Figure 5.5, which creates a set of variable orders from ω . For each variable order ω' in this set, we call the function $\tau(\omega')$ in Figure 4.2, which creates a view tree T following ω' . By Lemma 5.12, the view tree T can be materialized in $\mathcal{O}(N^{(\mathbf{w}(Q'_{\dagger})-1)\epsilon})$ time. Since $\mathbf{w}(Q'_{\dagger})$ is upper-bounded by \mathbf{w} , this implies $\mathcal{O}(N^{(\mathbf{w}-1)\epsilon})$ overall preprocessing time.

It remains to prove Lemma 5.12.

Lemma 5.12. *Let ω be a variable order of a CQAP query $Q(\mathcal{O}|\mathcal{I})$, X a variable in ω , Q_X the induced query at X in ω , $\omega' \in \Omega(\omega_X, (\mathcal{O}, \mathcal{I}))$, $\omega^t = (\text{anc}_{\omega}(X) \circ \omega')$, N the size of the leaf relations in ω' , and $\epsilon \in [0, 1]$. The view tree $\tau(\omega^t)$ can be materialized in $\mathcal{O}(N^{1+(\mathbf{w}(Q_X)-1)\epsilon})$ time.*

Proof. The proof is by induction on the structure of ω_X . We show that for each variable Y in ω^t , the view V_Y in $\tau(\omega^t)$ as defined in Line 4 of the procedure τ can be materialized in $\mathcal{O}(N^{1+(\mathbf{w}(Q_X)-1)\epsilon})$ time. Each auxiliary view defined in Line 8 of the procedure τ results from its child view by marginalizing a single variable. Materialising these auxiliary views does not increase the overall asymptotic computation time.

Base case: Assume that ω_X is a single atom. In this case, the procedure Ω returns this atom. The atom can obviously be materialized in $\mathcal{O}(N)$ time. Hence, the statement in the lemma holds.

Inductive step: Assume that the root variable X in ω_X has the child nodes X_1, \dots, X_k . Let $key = \mathbf{anc}_\omega(X) \cup \{X\}$, $\mathcal{I}_X = \mathbf{anc}_\omega(X) \cup (\mathcal{I} \cap \mathit{vars}(\omega_X))$, $\mathcal{O}_X = \mathcal{O} \cap \mathit{vars}(\omega)$. The induced query at X is defined as $Q_X(\mathcal{O} \mid \mathcal{I}) = \mathit{join}$ of $\mathit{atoms}(\omega)$. Following the control flow in Ω , we distinguish between the following cases.

Case (1): $Q_X(\mathcal{O} \mid \mathcal{I})$ is a CQAP₀ query (Figure 5.5, Line 6).

In this case, the procedure Ω returns the variable order $\omega' = \mathbf{Acc-Top}(\omega_X, (\mathcal{O} \mid \mathcal{I}))$. By Proposition 5.4, $\omega^t = (\mathbf{anc}_\omega(X) \circ \omega')$ is an access-top variable order for Q_X with static width $\mathbf{w}(Q_X)$. Since Q_X is in CQAP₀, its static width can be at most 1 (Propositions 5.30 and 5.27). This means that for every variable $Y \in \mathit{vars}(\omega^t)$, the set $\{Y\} \cup \mathit{dep}_{\omega^t}(Y)$ can be covered by a single atom in Q_X . Hence, each view $V_Y(\{Y\} \cup \mathit{dep}_{\omega^t}(Y))$ can be computed in $\mathcal{O}(N)$ time. This completes the inductive step for Case (1).

Case (2): Q_X is not in CQAP₀ and ($X \in \mathcal{I}$ or ($X \in \mathcal{O}$ and $\mathit{vars}(\omega) \cap \mathcal{I} = \emptyset$)) (Figure 5.5, Line 8).

The set of variable orders returned by Ω is defined as follows: for each set $\{\omega_i\}_{i \in [k]}$ with $\omega_i \in \Omega(\omega_{X_i}, (\mathcal{O} \mid \mathcal{I}))$, the set contains a variable order ω' , which has the root node X and child trees $\omega_1, \dots, \omega_k$. Consider for each such variable order ω' , the variable order $\omega^t = (\mathbf{anc}_\omega(X) \circ \omega')$. By induction hypothesis, each view tree over ω_i can be materialized in $\mathcal{O}(N^{1+(\mathbf{w}(Q_{X_i})-1)\epsilon})$ time. Since $\mathbf{w}(Q_{X_i}) \leq \mathbf{w}(Q_X)$ for any $i \in [k]$, it follows that each view tree over ω_i can be materialized in $\mathcal{O}(N^{1+(\mathbf{w}(Q_X)-1)\epsilon})$ time. Consider now the view tree $\tau(\omega^t)$. The view at X is defined by $V_X(\mathcal{S}) = V_{X_1}(\mathcal{S}_1), \dots, V_{X_k}(\mathcal{S}_k)$, where $\mathcal{S} = \{X\} \cup \mathit{dep}_\omega(X)$ and V_{X_1}, \dots, V_{X_k} are the child views of V_X . By the construction of view trees, V_X is a free-connex query. Hence, it can be computed by first marginalizing the variables in V_{X_i} that are not included in \mathcal{S} for each $i \in [k]$ and then computing the join of the remaining relations. This gives overall $\mathcal{O}(N^{1+(\mathbf{w}(Q_X)-1)\epsilon})$ computation time. This completes the inductive step in this case.

Case (3): Q_X is not in CQAP₀ and X is an output variable dominating an input variable, or it is a bound variable dominating a free variable. (Figure 5.5, Line 10).

In this case, the procedure Ω constructs a set htrees of variable orders and a single variable order ltree . The construction of the variable orders in htrees differs from the variable orders constructed under Case (2) only in that they refer to base relations that are heavy on the variable set key . This does not affect the asymptotic computation time of the view trees. Hence, the view trees over the variable orders htrees can be computed in $\mathcal{O}(N^{1+(\mathbf{w}(Q_X)-1)\epsilon})$ time. The variable order ltree is defined as

$ltree = \text{ACC-TOP}(\omega_X^{\text{key} \rightarrow L}, (\mathcal{O}|\mathcal{I}))$, where $\omega_X^{\text{key} \rightarrow L}$ indicates that the base relations are light on key . Observe that key is included in the schemas of the leaf atoms of $ltree$. By Proposition 5.4, $ltree$ is an access-top variable order for Q_X with optimal static width. Then, it follows from Lemma 5.14 that the view tree $\tau(ltree)$ can be materialized in $\mathcal{O}(N^{1+(w(Q_X)-1)\epsilon})$ time. This completes the inductive step for *Case 3*. \square

We need to prove Lemma 5.14, which is used in the above proof. The proof of Lemma 5.14 uses the following auxiliary lemma.

Lemma 5.13 ([43]). *For any hierarchical query Q and $\mathcal{F} \subseteq \text{vars}(Q)$, it holds $\rho^*(\mathcal{F}) = \rho(\mathcal{F})$.*

Proof. We define an integral edge cover $\lambda = (\lambda_R)_{R \in \text{atoms}(Q)}$ for Q and then show that $\sum_{R \in \text{atoms}(R)} \lambda_R \leq \rho^*(Q)$. Let ω be an arbitrary canonical variable order for Q . Recall that each root-to-leaf path in ω corresponds to an atom R in Q , such that the leaf is R and the set of inner nodes is the schema of R . A maximal variable path p in ω is a path that starts at a root and ends at a variable X such that all children of X are atoms. We call X the end variable of p . We assume that ω has k maximal variable paths. For each maximal variable path p in ω with end variable X , we fix an arbitrary child atom R_p of X . For each atom R in Q , we define:

$$\lambda_R = \begin{cases} 1 & , \text{ if } R = R_p \text{ for some maximal variable path } p \\ 0 & , \text{ otherwise} \end{cases}$$

It follows from the definition of λ that $\sum_{R \in \text{atoms}(R)} \lambda_R = k$.

Let $\lambda' = (\lambda'_R)_{R \in \text{atoms}(Q)}$ be an arbitrary fractional edge cover for Q . Given any maximal variable path p where the end variable has the child atoms R_1, \dots, R_m , we define $s_p = \sum_{i \in [m]} \lambda'_{R_i}$. We complete the proof by showing the following two statements:

1. $\lambda = (\lambda_R)_{R \in \text{atoms}(Q)}$ is an integral edge cover for Q .
2. $k \leq \sum_{R \in \text{atoms}(R)} \lambda'_R$.

$\lambda = (\lambda_R)_{R \in \text{atoms}(Q)}$ is an integral edge cover for Q : Let X be an arbitrary variable in Q . The variable X must be included in at least one maximal variable path p . Since $\lambda_{R_p} = 1$, the variable X is covered by the edge cover λ . Since X was chosen arbitrary, it follows that λ is an integral edge cover for Q .

$k \leq \sum_{R \in \text{atoms}(Q)} \lambda'_R$: Let p be an arbitrary maximal variable path in ω with end variable X . Let R_1, \dots, R_m be the child atoms of X . Besides R_1, \dots, R_m , no other atom has X in its schema. Hence, it must hold $s_p \geq 1$. Since there are k maximal variable paths, this implies $k \leq k \cdot s_p \leq \sum_{R \in \text{atoms}(Q)} \lambda'_R$. \square

We are now ready to prove Lemma 5.14.

Lemma 5.14. *Let ω be a variable order, X a variable in ω such that $\text{anc}_\omega(X)$ is included in the schemas of all leaf atoms in ω_X and $\omega^t = (\text{anc}_\omega \circ \omega_X)$. If the leaf relations in ω_X are the light parts of a partition on $\{X\} \cup \text{anc}_\omega(X)$ with threshold $\mathcal{O}(N^\epsilon)$ for some $\epsilon \in [0, 1]$, the view tree $\tau(\omega^t)$ can be materialized in $\mathcal{O}(N^{1+(\mathbf{w}(\omega^t)-1)\epsilon})$ time.*

Proof. Let $T = \tau(\omega^t)$ and $\mathbf{w} = \mathbf{w}(\omega^t)$. We show: every view in T can be computed in $\mathcal{O}(N^{1+(\mathbf{w}-1)\epsilon})$ time.

The leaf atoms can obviously be materialized in $\mathcal{O}(N)$ time. Consider any view $V_Y(\mathcal{S})$ in T with $\text{atoms}(\omega_Y^t) = \{R_i(\mathcal{X}_i)\}_{i \in [k]}$. The view V_Y is defined over the join of its child views and it holds $\mathcal{S} = \{Y\} \cup \text{dep}_\omega(Y)$. By the construction of our view trees, V_Y can be computed by joining the atoms $R_1(\mathcal{X}_1), \dots, R_k(\mathcal{X}_k)$. Hence, we can write the view as

$$V_Y(\mathcal{S}) = R_1(\mathcal{X}_1), \dots, R_k(\mathcal{X}_k).$$

Let $\rho^*(\mathcal{S}) = m$. By Lemma 5.13, $\rho(\mathcal{S}) = m$. We can find an optimal edge cover for \mathcal{S} by using only atoms from the set $\{R_i(\mathcal{X}_i)\}_{i \in [k]}$. Let $\lambda = (\lambda_{R_i(\mathcal{X}_i)})_{i \in [k]}$ be an edge cover of \mathcal{S} with $\sum_{i \in [k]} \lambda_{R_i(\mathcal{X}_i)} = m$. Let $\mathcal{R}_0, \mathcal{R}_1 \subseteq \text{atoms}(\omega_X)$ consist of the atoms in ω_X that λ assigns to 0 and 1, respectively. We first compute a view $V(\mathcal{S})$ over the join of the atoms in \mathcal{R}_1 as follows. We choose an arbitrary atom from \mathcal{R}_1 and iterate over its tuples. For each such tuple \mathbf{t} , we iterate over the matching tuples in the other atoms in \mathcal{R}_1 . Since each atom in \mathcal{R}_1 includes $\text{anc}_\omega(X)$ in its schema and is the light part of a partition on $\text{anc}_\omega(X)$ with threshold $\mathcal{O}(N^\epsilon)$, it contains $\mathcal{O}(N^\epsilon)$ tuples matching \mathbf{t} . This means that the time to materialise V is $\mathcal{O}(N \cdot N^{(m-1)\epsilon}) = \mathcal{O}(N^{1+(m-1)\epsilon})$. Now, we can rewrite V_Y using the new view V :

$$V_Y(\mathcal{S}) = V(\mathcal{S}), R'_1(\mathcal{X}'_1), \dots, R'_\ell(\mathcal{X}'_\ell), \quad (5.5)$$

where $R'_1(\mathcal{X}'_1), \dots, R'_\ell(\mathcal{X}'_\ell)$ are the atoms in \mathcal{R}_0 . The query (5.5) is free-connex α -acyclic, which means that it can be computed in time linear in the input plus the output size of V_Y , using Yannakakis's algorithm [11]. The input size is upper-bounded by $|V| = \mathcal{O}(N^{1+(m-1)\epsilon})$. The size of the output is also $\mathcal{O}(N^{1+(m-1)\epsilon})$. Hence, the overall time to compute V_Y is $\mathcal{O}(N^{1+(m-1)\epsilon})$. Since $m = \rho^*(\mathcal{S})$ is upper-bounded by \mathbf{w} , we derive that the computation time for V_Y is $\mathcal{O}(N^{1+(\mathbf{w}-1)\epsilon})$. Each of the additional auxiliary views constructed in Line 8 of the procedure τ is obtained by marginalizing away a variable from its child view. This does not blow up the overall asymptotic computation time. \square

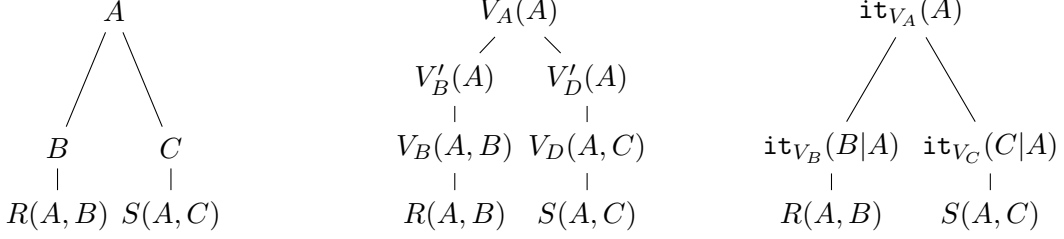


Figure 5.9: Variable orders constructed for $Q(A, B, C|\cdot) = R(A, B), S(A, C)$

5.3 Enumeration

In the enumeration algorithm for arbitrary CQAP queries (Section 4.2), we have introduced how to enumerate tuples from the view trees constructed in the preprocessing stage: for each view tree, we create iterators over the views that correspond to the free variables in the variable order of that view tree. We organize the iterators into nested loops based on a pre-order traversal of the view tree. We open the iterators with values from their ancestor views as context, thus ensuring they enumerate only those values guaranteed to be in the query output.

This approach, i.e., nesting the iterators, is valid for the view trees constructed following access-top variable orders, as in Section 4.1, where free variables are above the bound variables and input variables are above the output variables. The view trees constructed for the queries with hierarchical fractures, however, might not be access-top, and nesting view iterators may be invalid.

Example 5.15. Consider the query $Q(A, B, C|\cdot) = R(A, B), S(A, C)$. Figure 5.9 shows the access-top variable, the view tree constructed following the variable order, and the view iterators created over the view tree: These iterators can be nested in the enumeration procedure.

Assume now A is bound; the query becomes $Q(B, C|\cdot) = R(A, B), S(A, C)$. The variable order is not access-top anymore, and enumerating the query results, i.e., distinct (B, C) -tuples, is not possible by nesting the view iterators for B and C : the view iterators that enumerate B - and C -values have A in their context schemas, yet we do not create the iterator for A -values since A is bound. \square

We say a view iterator is *unsupported* if there are no other iterators enumerating values that fix every context variable in its context schema. In the above example, the view iterators for B and C are unsupported. View iterators might be unsupported when they are created following a non-access-top variable order. To resolve this issue, we define the *union view iterators*, which is a generalization of the view iterators.

```

uitV( $\mathcal{O}|\mathcal{I}$ ).open(relation ctx)

```

```

1 uitV( $\mathcal{O}|\mathcal{I}$ ).iterators := empty map // context tuple  $\mapsto$  view iterator
2 foreach ctxi  $\in$  ctx do
3   uitV( $\mathcal{O}|\mathcal{I}$ ).iterators[ctxi] := new uitV( $\mathcal{O}|\mathcal{I}$ )
4   uitV( $\mathcal{O}|\mathcal{I}$ ).iterators[ctxi].open(ctxi)

```

Figure 5.10: The method opens the union view iterator $\text{uit}_V(\mathcal{O}|\mathcal{I})$ for the input relation ctx over schema \mathcal{I} as context. The method creates for each tuple in ctx a view iterator and opens the view iterator for the corresponding tuple.

```

uitV( $\mathcal{O}|\mathcal{I}$ ).next() : (tuple, relation)

```

```

1 ctx1  $\mapsto$   $\text{it}_1(\mathcal{O}|\mathcal{I}), \dots, \text{it}_n(\mathcal{O}|\mathcal{I}) := \text{uit}_V(\mathcal{O}|\mathcal{I}).\text{iterators}$  // pattern matching
2 o := UNION( $\text{it}_1(\mathcal{O}|\mathcal{I}), \dots, \text{it}_n(\mathcal{O}|\mathcal{I})$ )
3 ctxo := {ctxi |  $i \in [n], \text{it}_i(\mathcal{O}|\mathcal{I}).\text{contains}(\mathbf{o})$ }
4 return (o, ctxo)

```

Figure 5.11: Fetch the next output tuple in the union view iterator, computed by the UNION algorithm, and the context tuples that are consistent with the output tuple.

5.3.1 Union View Iterators

We write $\text{uit}_V(\mathcal{O}|\mathcal{I})$ to denote a union view iterator over view V with output schema \mathcal{O} and context schema \mathcal{I} . It generalizes the view iterators in that the context of $\text{uit}_V(\mathcal{O}|\mathcal{I})$ is a *relation* (instead of a tuple) over schema \mathcal{I} . The empty context is the singleton relation with the empty tuple (the identity for the join operation). The $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{open}(ctx)$ method takes a relation ctx over schema \mathcal{I} as input, and then creates for each tuple $ctx_i \in ctx$ a view iterator and opens the view iterator for ctx_i . The $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{next}()$ method returns a pair $(\mathbf{o}, ctx_{\mathbf{o}})$, where \mathbf{o} is a tuple over \mathcal{O} that is consistent with at least one tuple in ctx in V and $ctx_{\mathbf{o}} \subseteq ctx$ represents the context tuples that are consistent with \mathbf{o} .

Figure 5.10 shows the $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{open}(ctx)$ method. It takes as input a relation ctx over \mathcal{I} and initializes an attribute $\text{uit}_V.\text{iterators}$ of mapping type (Line 1), which maps each tuple $ctx_i \in ctx$ to a new view iterator (Line 3). Each view iterator is opened for the corresponding tuple (Line 4). The time needed by the $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{open}(ctx)$ method is given by the time to create and open $|ctx|$ iterators.

The $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{next}()$ method first uses the UNION algorithm, as described in Figure 5.12, to get the next distinct output tuple from the $\text{uit}_V(\mathcal{O}|\mathcal{I}).\text{iterators}$. The UNION

```

UNION(iterators  $it_1, \dots, it_n$ ): tuple


---


1  if ( $n = 1$ )
2    return  $it_n.next()$ 
3  if ( $t_{[n-1]} := \text{UNION}(it_1, \dots, it_{n-1}) \neq \mathbf{EOF}$ )
4    if ( $it_n.contains(t_{[n-1]})$ )
5      return  $it_n.next()$ 
6    return  $t_{[n-1]}$ 
7  if ( $t_n := it_n.next()$ )  $\neq \mathbf{EOF}$ 
8    return  $t_n$ 
9  return  $\mathbf{EOF}$ 

```

Figure 5.12: Return the next distinct tuple from a set of iterators.

algorithm is an adaptation of prior work [30]. It takes as input n iterators with the same output schema, which enumerate tuples from possibly overlapping sets, and returns a tuple in the union of these sets, where the tuple is distinct from all tuples returned before. Upon each call, the function returns one tuple. If all iterators are exhausted, the function returns **EOF**.

We first explain the algorithm on two iterators it_1 and it_2 . For the next tuple t_1 of it_1 , the function checks whether t_1 will be enumerated by it_2 using $it_2.contains(t_1)$. If so, it returns the next tuple in it_2 ; otherwise, it returns t_1 . In case it_1 is exhausted, the function returns the next tuple in it_2 , or **EOF** in case it_2 is also exhausted.

In case of $n > 2$ iterators, the function considers the UNION of the first $n - 1$ iterators as the next tuple of one iterator and it_n as the second iterator, and then reduces the general case to the previous case of two iterators.

The delay of the UNION algorithm is upper-bounded by the product of the number of iterators and the delay of the slowest iterator.

Figure 5.11 shows the $uit_V(\mathcal{O}|\mathcal{I}).next()$ method. Once the method gets the next output tuple \mathbf{o} from the UNION function (Line 2), it looks up \mathbf{o} in all view iterators to compute the context $ctx_{\mathbf{o}} \subseteq ctx$ of \mathbf{o} , which is defined as the relation of the context tuples of the iterators that return true (Line 3). The method returns \mathbf{o} together with its context $ctx_{\mathbf{o}}$ (Line 4).

The delay of the $uit_V(\mathcal{O}|\mathcal{I}).next()$ method is given by the delay of finding the next output tuple using the UNION algorithm and the cost of getting the context of the output tuple. Assume $uit_V(\mathcal{O}|\mathcal{I})$ is opened for a relation ctx . Fetching the output tuple \mathbf{o} takes $\mathcal{O}(|ctx|)$ time using the UNION algorithm, since we create an iterator for each context

```

1   $ctx_0(A, C) := \{(a_0, c_0)\}$ ,   where  $a_0, c_0$  are input values
2   $uit_{V_A}(A|A).open(\pi_A(ctx_0))$ 
3  while  $((a, ctx_a) := uit_{V_A}(A|A).next()) \neq (\mathbf{EOF}, \emptyset)$  do
4     $uit_{V_C}(C|A, B, C).open(V_B(A, B) \bowtie ctx_0)$ 
5    while  $((c, ctx_c) := uit_{V_C}(C|A, B, C).next()) \neq (\mathbf{EOF}, \emptyset)$  do
6       $uit_{V_D}(D|A, B).open(\pi_{AB}(ctx_c))$ 
7      while  $((d, ctx_d) := uit_{V_D}(D|A, B).next()) \neq (\mathbf{EOF}, \emptyset)$  do
8        output  $(d)$ 
9  output EOF

```

Figure 5.13: Enumeration for $Q(D|A, C) = R(A, B, C), S(A, B, D)$ using the view tree from Figure 4.8.

tuple in ctx and thus $\mathcal{O}(ctx)$ iterators, and the delay of all iterators is $\mathcal{O}(1)$. Computing the set $ctx_{\mathbf{o}} \subseteq ctx$ for \mathbf{o} also takes $\mathcal{O}(|ctx|)$ time. Hence, the $uit_V(\mathcal{O}|\mathcal{I}).next()$ method runs in $\mathcal{O}(|ctx|)$ time.

Similar as the *contains* method for view iterators, the $uit_V(\mathcal{O}|\mathcal{I}).contains(\mathbf{o})$ checks whether a tuple \mathbf{o} over the output variables \mathcal{O} is in the output of $uit_V(\mathcal{O}|\mathcal{I})$. It can be performed by checking $it.contains(\mathbf{o})$ for each iterator it in $uit_V(\mathcal{O}|\mathcal{I}).iterators$. Hence, assume $uit_V(\mathcal{O}|\mathcal{I})$ is opened for a relation ctx , the $uit_V(\mathcal{O}|\mathcal{I}).contains(\mathbf{o})$ method runs in $\mathcal{O}(|ctx|)$ time.

Example 5.16. Consider again the view tree in Figure 4.8 (middle) but now for:

$$Q(D|A, C) = R(A, B, C), S(A, B, D),$$

which has a different access pattern. Figure 5.13 shows the enumeration procedure for query.

We construct three union view iterators, one for each free variable. The iterator $uit_{V_A}(A|A)$ serves to check if the given A -value exists in V_A (Lines 2-3). The iterator $uit_{V_C}(C|A, B, C)$ is unsupported as there is no binding for variable B . For this iterator, we provide a relation over schema (A, B, C) as context. To avoid enumerating dangling tuples, the context should include only those B -values guaranteed to have matching D -values in the final output. The ancestor view $V_B(A, B)$ provides such (A, B) -values, which we further restrict to those matching the given input values (Line 4). The $next()$ call on uit_{V_C} returns the input C -value together with a relation ctx_c containing the matching (A, B, C) -tuples in V_C if they exist; otherwise, it returns $(\mathbf{EOF}, \emptyset)$. The relation ctx_c serves as context for the iterator over D -values (Line 6).

```

BUILDUNIONITERATORS(view tree  $T$ , access pattern  $(\mathcal{O}|\mathcal{I})$ , relation  $supp$ )

```

```

switch  $T$ :

```

```

 $R(\mathcal{Y})$    1  return []

```

```

 $V_X(\mathcal{X})$   2  if  $X \notin \mathcal{X}$                                 // skip auxiliary maintenance views
/ \        3  return BUILDUNIONITERATORS( $T_1, (\mathcal{O}|\mathcal{I}), supp$ )
 $T_1 \cdots T_k$ 
4   $it_X = \begin{cases} [(\mathbf{new\ uit}_{V_X}(X|\mathcal{X}), supp)] & , \text{if } X \in \mathcal{I} \\ [(\mathbf{new\ uit}_{V_X}(X|\mathcal{X} \setminus \{X\}), supp)] & , \text{if } X \in \mathcal{O} \\ [] & , \text{otherwise} // \text{empty list} \end{cases}$ 
5   $supp_{child} := \begin{cases} supp & , \text{if } X \in (\mathcal{I} \cup \mathcal{O}) \\ V_X(\mathcal{X}) & , \text{otherwise} \end{cases}$ 
6   $it_{child_i} := \text{BUILDUNIONITERATORS}(T_i, (\mathcal{O}|\mathcal{I}), supp_{child}), \forall i \in [k]$ 
7  return  $it_X ++ it_{child_1} ++ \dots ++ it_{child_k}$  // concatenation of the lists

```

Figure 5.14: Create a list of union view iterators with support for the access pattern $(\mathcal{O}|\mathcal{I})$ in a view tree T . The operator $++$ concatenates two lists. The first call to BUILDUNIONITERATORS uses the support $\{()\}$.

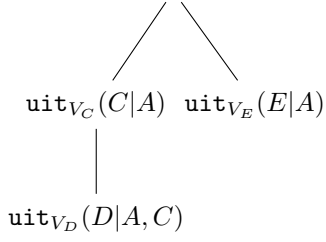
The open and next calls take time linear in the size of the context ctx used when opening the iterator. The size of the context for \mathbf{uit}_{V_A} is constant, while for \mathbf{uit}_{V_C} and \mathbf{uit}_{V_D} is at most the size of V_B . Thus, the enumeration delay is $\mathcal{O}(|V_B|)$. \square

5.3.2 Enumeration Procedure

We use the variable orders of the view trees to guide the construction of the iterators. For each view tree, we create iterators over the views that correspond to the free variables in the variable order of that view tree. The variable orders constructed for the hierarchical CQAP queries might not be access-top variable orders, so we need to create union view iterators over the view trees.

We extend the function BUILDITERATORS for the case of general queries from Figure 4.9 to the function BUILDUNIONITERATORS as shown in Figure 5.14. It builds a list of *union* view iterators for a given view tree of a CQAP query Q with access pattern $(\mathcal{O}|\mathcal{I})$. Each union view iterator comes paired with a support relation that provides the context for any variable with no binding. The support provided in the initial call to BUILDITERATORS is the singleton relation with the empty tuple (the identity for the join operation).

The function recursively constructs union view iterators, traversing the view tree T in a top-down fashion. Consider the root view $V_X(\mathcal{X})$ of T constructed at variable X in the



```

1  ctx0 := {e0} // where e0 is the input E-value
2  uitVE(E|A, E).open(VA(A) ⋈ ctx0)
3  while ((e, ctxe) := uitVE(E|A, E).next()) ≠ EOF do
4    uitVC(C|A).open(ctxe)
5    while ((c, ctxc) := uitVC(C|A).next()) ≠ EOF do
6      uitVD(D|A, C).open(ctxc ⋈ {c})
7      while ((d, ctxd) := uitVD(D|A, C).next()) ≠ EOF do
8        m := ∑a ∈ πActxd VD(a, c, d) · VC(a, c) · VE(a, e)
9        output (c, d) ↦ m
10 output EOF

```

Figure 5.15: Enumeration procedure for the connected component $Q(C, D|E) = R^{A \rightarrow H, AB \rightarrow L}(A, B, C), S^{B \rightarrow H, AB \rightarrow L}(A, B, D), T^{A \rightarrow H}(A, E)$.

corresponding variable order. The function creates a union view iterator over V_X if X is a free variable. Otherwise, if X is a bound variable, it uses V_X as the support relation for any union view iterator created for a free variable below X . The function recursively creates iterators in each subtree and concatenates them into a list of iterators with their support relation.

Once we construct the iterators over the view tree, we generate the enumeration procedure by organizing the iterators into nested loops based on a pre-order traversal of the view tree. We open the iterators with the output values and context relations from their ancestor views as context, thus ensuring they enumerate only those values guaranteed to be in the query output. Each concatenation of the outputs of the iterators forms the values of an output tuple.

The time for an iterator to report an output tuple, i.e., the *next* method of the iterator, is determined by the size of its input context relation. That is, the size of the support relations. Hence, the enumeration delay of the procedure is upper-bounded by the size of the support relations.

Example 5.17. Consider the view tree from Figure 5.7 (left in the second row), created for $Q(C, D|E) = R^{A \rightarrow H, AB \rightarrow L}(A, B, C), S^{B \rightarrow H, AB \rightarrow L}(A, B, D), T^{A \rightarrow H}(A, E)$. BUILDUNION-ITERATORS returns the following iterators for this view tree:

- $uit_{V_E}(E|A, E)$ with the support $V_A(A)$,
- $uit_{V_C}(C|A)$ with the support $V_A(A)$, and
- $uit_{V_D}(D|A, C)$ with the support $V_A(A)$.

COMPUTEM(view tree T , tuple \mathbf{t} , context relations $contexts_{\mathbf{t}}$): multiplicity

switch T :

```

 $V_X(\mathcal{X})$  1  if  $Sch(\mathbf{t}) \subsetneq \mathcal{X}$ 
/ \      2     $\{A_1, \dots, A_k\} := \mathcal{X} \setminus Sch(\mathbf{t})$ 
 $T_1 \cdots T_k$  3     $\mathcal{A}_1 := \pi_{A_1}(\bowtie_{ctx \in contexts_{\mathbf{t}}} ctx)$  //  $A_1$ -values that satisfy all context relations
4    return  $\sum_{a \in \mathcal{A}_1} COMPUTEM(T, \mathbf{t} \circ a, contexts_{\mathbf{t}} \cup \{\{a\}\})$ 
5  else if  $\mathcal{X} \subsetneq Sch(\mathbf{t})$ 
6     $\mathcal{V}_i :=$  variables in  $T_i$ 
7     $contexts_i := \{\pi_{\mathcal{V}_i} R \mid R \in contexts_{\mathbf{t}}\}$ 
8    return  $\prod_{i \in [k]} COMPUTEM(T_i, \pi_{\mathcal{V}_i} \mathbf{t}, contexts_i)$ 
9  else //  $\mathcal{X} = Sch(\mathbf{t})$ 
10 return  $V[\mathbf{t}]$ 

```

Figure 5.16: Compute the multiplicity of the given tuple \mathbf{t} in the view tree T . The input $contexts_{\mathbf{t}}$ contains all the context sets returned during the enumeration of \mathbf{t} .

Figure 5.15 shows the enumeration procedure for these iterators. The returned support relations define the context to be used when opening each union view iterator. As discussed in the next section, to compute the multiplicity of the output tuple (c, d) for the input E -value e_0 , we sum over the multiplicities of the tuple concatenated with the A -values in the context relation ctx_d (Line 9).

□

Multiplicity. We extend the function COMPUTEM in Figure 4.10 for view trees that are not constructed over access-top variable orders. Figure 5.16 shows the extended COMPUTEM function for computing the multiplicity of a tuple \mathbf{t} in a view tree T . The parameter $context_{\mathbf{t}}$ contains the set of context relations returned by the *next* method of the union view iterators for the tuple \mathbf{t} , such as the relations ctx_e , ctx_c and ctx_d in Example 5.17.

The function traverses the view tree T based on a pre-order. At the root view $V(\mathcal{X})$ of T , there are three cases: (1) the view V has a variable A_1 that is not in the schema of the tuple \mathbf{t} (Line 1). This corresponds to the case when A_1 is bound and has been aggregated away from the views below V in the view tree. In this case, we treat A_1 as if it is free, and sum over the multiplicities of the concatenations of \mathbf{t} and the A_1 -values paired with \mathbf{t} in the view tree, i.e., the A_1 -values in the context: For each such A_1 -value from the context set (Lines 2-3), the function concatenates the value to \mathbf{t} , and applies COMPUTEM to compute the multiplicity of the new tuple. The multiplicity of \mathbf{t} is the

sum of the multiplicities of these new tuples (Line 4). (2) The second case is the opposite of the first case: the schema of \mathbf{t} has additional variables that are not in the schema of V (Line 5). This means the tuple \mathbf{t} is stored below V , possibly distributed in different branches. The function applies COMPUTEM recursively to each subtree and takes the product of the returned multiplicities (Lines 6-8). (3) When \mathbf{t} is in V , the function returns the multiplicity of \mathbf{t} in V (Lines 9-10).

The computation time of the multiplicity of a tuple \mathbf{t} is upper-bounded by the time for enumerating \mathbf{t} using the iterators. The time of the function COMPUTEM is determined by the number of multiplicities to be summed in the first case. That is, the size of the context relations. Since these context relations are all subsets of the support relations (as per the *next* method of union view iterators), their sizes are upper-bounded by the sizes of the support relations. Hence, COMPUTEM does not take time more than the time for the enumerating the tuple \mathbf{t} using the iterators.

5.3.3 Enumeration from View Trees

In the preprocessing stage, we might have constructed multiple view trees for different skew-aware queries. The result of the query is the union of the results of the skew-aware queries, i.e., the union of the results represented the view trees. Enumerating the result of the query, however, is not as straightforward as enumerating the results of the skew-aware queries separately: the results of the skew-aware queries might be overlapping; we need to avoid enumerating the same tuple multiple times, and the multiplicity of a tuple is the product of the multiplicities of the tuple in all view trees.

To enumerate from multiple view trees, we use the UNION algorithm again: we consider the enumeration in each view tree as an iterator, and use the UNION algorithm to enumerate the union of the results of the iterators. Checking whether a tuple will be enumerated from a view tree can be implemented by calling the *contain* method of the union view iterators created in this view tree. Recall the time for the UNION algorithm to report a tuple is determined by the product of the number of iterators and the delay of the slowest iterator. Since the number of iterators, i.e., the number of skew-aware queries, is constant, the enumeration delay is the time for enumerating one tuple from each view tree, which is upper-bounded by the size of the support in the view tree.

5.3.4 Enumeration Delay

We next discuss the complexities in the enumeration stage.

Proposition 5.18. *For any CQAP₀ query, its distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(1)$ delay.*

Proof. We want to show that for any CQAP₀ query, its distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(1)$ delay.

The fracture of any CQAP₀ query with access pattern $(\mathcal{O}|\mathcal{I})$ is hierarchical, $(\mathcal{O} \cup \mathcal{I})$ -dominant, and \mathcal{I} -dominant, per Definition 3.7. For each connected component of the fracture, we can construct a variable order where the free variables are above the bound variables and the input variables are above the output variables, see the Ω function from Figure 5.5. For the view tree constructed following that variable order, we can create a list of view iterators by doing a pre-order traversal of the view tree such that the iterators for input variables precede those for output variables in the list. By forming a nesting chain of these iterators, we can enumerate the distinct output tuples for the given input tuple with constant delay.

If the fracture consists of several connected components, we concatenate the list of iterators constructed for each connected component and form a nesting chain for the enumeration from their view trees. \square

Proposition 5.19. *For any hierarchical CQAP query Q , database of size N , and $\epsilon \in [0, 1]$, the distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay.*

Proof. We now sketch the proof that for any hierarchical CQAP query Q , database of size N , and $\epsilon \in [0, 1]$, the distinct output tuples given an input tuple can be enumerated with $\mathcal{O}(N^{1-\epsilon})$ delay.

Consider a CQAP query Q with hierarchical fractures. If Q is in CQAP₀, the distinct output tuples can be enumerated with $\mathcal{O}(1)$ delay, per Proposition 5.18. Otherwise, there exists a variable X such that either X is a bound variable and above a free variable or X is an output variable and above an input variable in the canonical variable order of Q . For each such case, we partition the relations in the subtree rooted at X and create different evaluation strategies over the heavy and light relation parts, see Figure 5.5.

In the light case, the created view trees follow access-top variable orders, thus admitting constant delay enumeration of the output tuples for a given input tuple. In the heavy case, the view defined X consists of at most $N^{1-\epsilon}$ heavy values, which define the support for the enumeration from child views. This size of the support determines the enumeration delay. \square

5.4 Update

In this section, we discuss the maintenance of the view trees computed in the preprocessing step under a single-tuple update δR to any base relation R . Our approach to effect

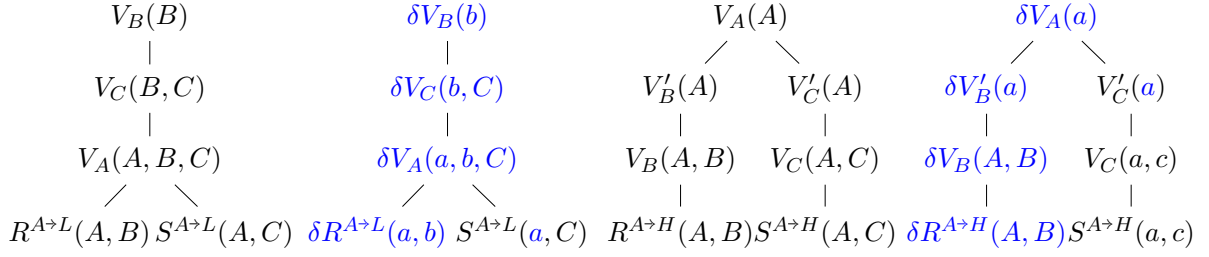


Figure 5.17: First and third from left: The view trees constructed for $Q(B, C) = R(A, B), S(A, C)$; The base relations are partitioned on the key A . Second and fourth from left: The delta view trees under a single-tuple update to R .

TRANSIENTHLS(tuple \mathbf{x}) : HL-signature	
1	$\{k_1, \dots, k_n\} := \{k \mid k \in \text{PARTITIONKEYS}, k \subseteq \text{Sch}(\mathbf{x})\}$
2	$\mathcal{K} :=$ parts of base relations
3	$s_i := \begin{cases} \text{sig}[k_i], & \text{if } \exists K^{sig} \in \mathcal{K} \text{ s.t. } \mathbf{x}[k_i] \in \pi_{k_i} K^{sig} \\ L, & \text{otherwise} \end{cases} \quad \text{for } i \in [n]$
4	return REMOVEHEAVYTAIL($\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\}$)

Figure 5.18: Computing an HL-signature for tuple \mathbf{x} by checking in which relation parts the values in \mathbf{x} are contained. PARTITIONKEYS consists of the set of all keys the base relations are partitioned on. $\text{sig}[k]$ returns the symbol the key k is mapped to in the HL-signature sig .

this update is as follows. We first identify which part of a relation R is affected by the update: We check the degrees of \mathbf{x} among the keys on which R is partitioned and find the relation part R^{sig} that has the matched degrees. Several view trees can refer to the same relation part. To simplify the reasoning about the maintenance task, we assume that each view tree has a copy of its relation parts. Then, for each view tree that contains R^{sig} , we update the view tree by calling the APPLY function in Figure 4.11.

As the database evolves under updates, the degree of values over a partition key may change: for example, a new tuple can change the degree of a partition key from light to heavy, or when a lot of new tuples are inserted, the size of the database may increase, and thus the threshold also increases. This may cause a partition key that was previously heavy to become light. For such cases, we periodically rebalance the relation partitions and views to account for new database sizes and updated degrees of values.

5.4.1 Determining the Relation Part for a Single-Tuple Update

Given an update $\delta R = \{\mathbf{x} \rightarrow m\}$, we have to find out which part of relation R is affected by the update. That is, we need to compute the HL-signature of the part of R on which

REMOVEHEAVYTAIL(HL-signature sig) : HL-signature

```

1   $\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\} := sig$ 
2   $heavyTail := \emptyset$ 
3  foreach  $i \in [n]$ 
4    if  $\exists j \in [n]$  s.t.  $s_j = L$  and  $k_j \subset k_i$ 
5       $heavyTail := heavyTail \cup \{k_i \rightarrow s_i\}$ 
6  return  $sig \setminus heavyTail$ 

```

Figure 5.19: Deletion of the heavy tail from an HL-signature sig . If $k \rightarrow L$ and $k' \rightarrow H$ are included in sig and k is a proper subset of k' , then $k' \rightarrow H$ is deleted from sig .

the update is to be applied.

Example 5.20. Consider the query $Q(B, C) = R(A, B), S(A, C)$. Figure 5.17 (first and third from left) shows the view trees constructed for the query in the preprocessing stage; the base relations are partitioned on the key A . Let $\delta R = \{(a, b) \rightarrow m\}$ an update to the base relation R . We need to compute the HL-signature of the A -value a to find out which part of relation R is affected. If a exists in $R^{A \rightarrow L}$ or does not exist in the database, a is light on the partition key A and thus affects the part $R^{A \rightarrow L}$; otherwise, i.e., a is in $R^{A \rightarrow H}$, a is heavy and thus affects $R^{A \rightarrow H}$.

The function TRANSIENTHLS(\mathbf{x}) in Figure 5.18 constructs an HL-signature by checking in which relation parts the values in \mathbf{x} are contained. The set PARTITIONKEYS (in Line 1) consists of all keys on which the input relations are partitioned. The function first creates an HL-signature $\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\}$ where each k_i is included in PARTITIONKEYS and is a subset of the schema of \mathbf{x} (Line 1). If there exists a relation part K^{sig} such that $\mathbf{x}[k_i]$ is included in the projection of K^{sig} onto k_i , s_i is defined as the symbol the key k_i is mapped to in sig (first case in Line 3). Otherwise, $\mathbf{x}[k_i]$ does not exist in the database yet, so it is light. Thus, in this case s_i is defined as L (first case in Line 3).

For hierarchical queries, recall that our preprocessing stage does not further partition a relation on a key k if the relation is already light on a subset of k . Hence, we apply the function REMOVEHEAVYTAIL in the last line of TRANSIENTHLS (defined in Figure 5.19) to remove from sig all pairs $k \rightarrow s$ such that there is $k' \rightarrow L$ in sig with $k' \subset k$. We call the HL-signature constructed by TRANSIENTHLS(\mathbf{x}) the transient HL-signature of \mathbf{x} .

When constructing relation parts from scratch, we determine the part a tuple needs to be included based on the degrees of the values in the tuple. Given a tuple \mathbf{x} and a threshold θ , the function ACTUALHLS(\mathbf{x}, θ) in Figure 5.20 computes an HL-signature sig based on θ . If the degree of the projection of \mathbf{x} onto a partition key is below θ in all

ACTUALHLS(tuple \mathbf{x} , threshold θ) : HL-signature

- 1 $\{k_1, \dots, k_n\} := \{k \mid k \in \text{PARTITIONKEYS}, k \subseteq \text{Sch}(\mathbf{x})\}$
- 2 $s_i := \begin{cases} L, & \text{if } \forall K \in \mathcal{D}: |\sigma_{k_i=\mathbf{x}[k_i]}K| < \theta \\ H, & \text{otherwise} \end{cases}$ for $i \in [n]$
- 3 **return** REMOVEHEAVYTAIL($\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\}$)

Figure 5.20: Computing an HL-signature for tuple \mathbf{x} by checking the degrees of the values in \mathbf{x} based on the threshold θ .

UPDATETREES(view trees \mathcal{T} , update δR)

- 1 $\delta R := \{\mathbf{x} \rightarrow m\}$
- 2 $sig := \text{TRANSIENTHLS}(\mathbf{x})$
- 3 **foreach** $T \in \mathcal{T}$ **do** APPLY($T, \delta R^{sig} = \{\mathbf{x} \rightarrow m\}$)

Figure 5.21: Updating a set \mathcal{T} of view trees for a single-tuple update $\delta R = \{\mathbf{x} \rightarrow m\}$ to relation R . If \mathbf{x} is already included in a part of R , all view trees referring to that part are updated. Otherwise, the HL-signature sig of \mathbf{x} is computed and all view trees referring to R^{sig} are updated.

input relations, sig maps the partition key to L (first case in Line 2). Otherwise, the partition key is mapped to H (second case in Line 2). The HL-signature constructed by ACTUALHLS(\mathbf{x}, θ) is called the transient HL-signature of \mathbf{x} based on θ .

5.4.2 Processing a Single-Tuple Update

Given a set \mathcal{T} of view trees and an update $\delta R = \{\mathbf{x} \rightarrow m\}$, the procedure UPDATETREES($\mathcal{T}, \delta R$) in Figure 5.21 maintains the view trees under the update. It first computes the transient HL-signature sig of \mathbf{x} (Line 2). Then, it applies $\delta R^{sig} = \{\mathbf{x} \rightarrow m\}$ to the view trees in \mathcal{T} (Line 3). There might be several view trees constructed in our preprocessing stage that refer to R^{sig} . Next, the function updates each view tree using the function APPLY($T, \delta R^{sig}$) from Figure 4.11 (Line 3). If T does not refer to R^{sig} , the procedure has no effect.

Example 5.21. *Figure 5.17(second and fourth from left) shows the delta view trees for the corresponding view trees under the single-tuple update $\delta R = \{(a, b) \mapsto m\}$ to R . The delta view trees for an update to S are analogous. The blue views in the view trees are the deltas to the corresponding views, computed while propagating δR from the affected relation part to the root view. The update δR affects the light part $R^{A \triangleright L}(A, B)$ of R if the tuple a, b is light on the partition key A . In this case, we update the relation part $R^{A \triangleright L}(A, B)$ with $\delta R^{A \triangleright A}(a, b) = \delta R(a, b)$, and propagate the change up the tree.*

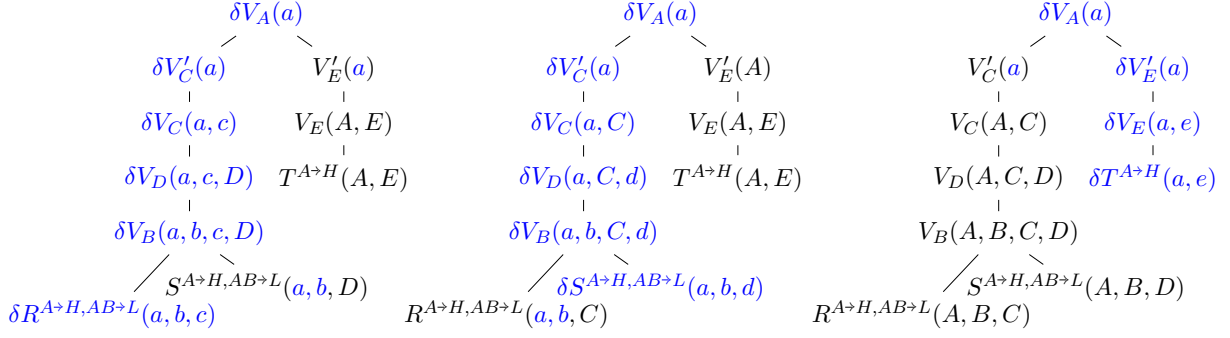


Figure 5.22: The delta view trees for the middle right view tree in Figure 5.7 under a single-tuple update to R , S , and T , respectively.

We update $V_A(A, B, C)$ with $\delta V_A(a, b, C) = \delta R^{A \rightarrow L}(a, b), S^{A \rightarrow L}(a, C)$ in $\mathcal{O}(N^\epsilon)$ time, since there are at most N^ϵ C -values paired with value a in $S^{A \rightarrow L}$. We then update $V_C(B, C)$ with $\delta V_C(b, C) = \delta V_A(a, b, C)$ in $\mathcal{O}(N^\epsilon)$ time, and similarly for the view $V_B(B)$ with $\delta V_B(b) = \delta V_C(b, C)$ in $\mathcal{O}(1)$ time.

In case δR affects the heavy part $R^{A \rightarrow H}(A, B)$, i.e., (a, b) is heavy on A , we update $V_B(A, B)$ with $\delta V_B(a, b) = \delta R^{A \rightarrow H}(a, b)$ in $\mathcal{O}(1)$ time and then update the other views $V'_B(A)$ and V_A similarly in $\mathcal{O}(1)$ time.

Overall, maintaining the two view trees under a single-tuple update to any relation takes $\mathcal{O}(N^\epsilon)$ time. \square

Example 5.22. Figure 5.22 shows the delta view trees for the middle right view tree in Figure 5.7 under the single-tuple update $\delta R = \{(a, b, c) \rightarrow m\}$ to R , $\delta S = \{(a, b, d) \rightarrow m\}$ to S , and $\delta T = \{(a, e) \rightarrow m\}$ to T .

For the delta view tree for the update δR , we update the view $V_B(A, B, C, D)$ with $\delta V_B(a, b, c, D) = \delta R^{A \rightarrow H, AB \rightarrow L}(a, b, c), S^{A \rightarrow H, AB \rightarrow L}(a, b, D)$ in $\mathcal{O}(N^\epsilon)$ time. We then update $V_D(A, C, D)$ with $\delta V_D(a, c, D) = \delta V_B(a, b, c, D)$ with constant time and similarly for the views $V_C(A, C)$, $V'_C(A)$ and $V_A(A)$. The computation of the delta view tree for the update δS is similar. For the update δT , we update the view $V_E(A, E)$ with $\delta V_E(a, e) = \delta T^{A \rightarrow H}(a, e)$ with constant time and similarly for the views $V'_E(A)$ and $V_A(A)$.

Overall, maintaining the view trees under a single-tuple update to any relation takes $\mathcal{O}(N^\epsilon)$ time. \square

We next state the complexity of processing a single-tuple update for a CQAP query with a hierarchical fracture in the following proposition.

Proposition 5.23. Given a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture, a dynamic width δ , a database of size N , and $\epsilon \in [0, 1]$, the view trees constructed in the preprocessing stage can be maintained under a single-tuple update to any input relation in $\mathcal{O}(N^{\delta\epsilon})$ time.

Proof. In the preprocessing stage, for a CQAP query Q with input variables \mathcal{I} , output variables \mathcal{O} , canonical variable order ω and delta width δ , we construct variable orders $\Omega(\omega, (\mathcal{O}|\mathcal{I}))$ and then construct view trees following these variable orders using the procedure τ . The procedure Ω traverses the variable order ω in a top-down manner. Consider any subtree ω' of ω rooted at X and the residual query Q_X at X in ω . The procedure Ω distinguishes different cases.

In case the residual query Q_X is in CQAP_0 , Ω creates an access-top variable order ω'_{at} for ω' . At each node X of ω'_{at} , τ creates a view V_X with schema $\{X\} \cup \text{dep}_{\omega'_{at}}(X)$ that joins the child views below. By construction, if X has only one child Y in ω'_{at} , the child view V_Y created at Y below V_X has the schema $\{X, Y\} \cup \text{dep}_{\omega'_{at}}(X)$ and V_X is computed by variable marginalisation, otherwise, i.e., V_X has multiple child views, these child views have the same schema $\{X\} \cup \text{dep}_{\omega'_{at}}(X)$ as V_X . Consider an update δR to a relation R . The update δR fixes the values of all variables on the path from the leaf R to the root to constants. While propagating an update through the view tree, the delta for each view V_X requires joining the update with the sibling child views of X . Each of these sibling child views (if it exists) has the same schema as view at X , as discussed above. Thus, computing the delta at each node makes only constant-time lookups in the sibling views. Overall, propagating the update through the view tree constructed for a CQAP_0 residual query takes constant time.

We now discuss the case Q is not in CQAP_0 . If X is an input variable, or X is an output variable and its ancestors have no input variable, the Ω procedure traverses to the subtrees of ω' and attaches the constructed variable orders to X . The τ procedure creates a view V_X at X with the schema $\{X\} \cup \text{dep}_{\omega'}(X)$ that joins the child views. By construction, the schema $\{X\} \cup \text{dep}_{\omega'}(X)$ is covered by any atom of ω' , and the same as discussed above, if X has only one child Y in ω'_{at} , the child view V_Y created at Y below V_X has the schema $\{X, Y\} \cup \text{dep}_{\omega'_{at}}(X)$ and V_X is computed by variable marginalisation, otherwise, i.e., V_X has multiple child views, these child views have the same schema $\{X\} \cup \text{dep}_{\omega'_{at}}(X)$ as V_X . Since an update to any base relation in ω' fixes all variable in V_X , the delta for V_X can be computed in constant time by constant-time lookups.

If X is a bound variable and ω' has free variables, or X is an output variable and ω' has input variables, the Ω procedure partitions the base relations of ω' on $\text{anc}(X) \cup \{X\}$. In the heavy case, Ω traverses to the subtrees of ω' as in the previous case except the base relations are replaced by the heavy parts of the relations. The delta for the view constructed at X can be computed in constant time.

In the light case, Ω builds an access-top variable order ω'_{at} of ω' with the light parts of the base relations as its leaves, and then τ constructs a view tree *ltree* following ω'_{at} . At variable X in ω'_{at} , τ creates a view V_X with schema $\mathcal{S}_X = \{X\} \cup \text{dep}_{\omega'_{at}}(X)$. Consider

an update δR that affects the light part of relation R . While propagating the update up, at V_X , the update δR does not fix all variables in \mathcal{S}_X and the unfixed variables are distributed in δ' views below V_X ($\delta' \leq \delta$ since the unfixed variables are covered by at most δ atoms, according to the definition of dynamic width). Computing the delta for V_X requires finding the values of these unfixed variables in the δ' views below V_X . Since the leaves of ω'_{at} are the light parts of the base relations, we can fetch the values of unfixed variables in each view in $\mathcal{O}(N^\epsilon)$ time and $\mathcal{O}(N^{\delta'\epsilon})$ time in δ' views. In the worst case, δ' can be as large as δ , and therefore the update time is $\mathcal{O}(N^{\delta\epsilon})$.

Overall, the update time for a single-tuple update to any input relation takes $\mathcal{O}(N^{\delta\epsilon})$ time. □

5.4.3 Processing a Sequence of Single-Tuple Updates

An update may change the degree of values over a partition key from light to heavy or vice versa. In such cases, we need to rebalance the partitioning and possibly recompute some views to account for a new database size and updated degrees of data values. Although such rebalancing steps may take time more than processing a single-tuple update, they happen periodically and their amortized cost remains the same as for a single-tuple update.

Major Rebalancing. We loosen the partition threshold to amortize the cost of rebalancing over multiple updates. Instead of the actual database size N , the threshold now depends on a number M for which the invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ always holds. If the database size falls below $\lfloor \frac{1}{4}M \rfloor$ or reaches M , we perform *major rebalancing*, where we halve or respectively double M , followed by strictly repartitioning the relation parts with the new threshold M^ϵ and recomputing the views. Figure 5.23 shows the major rebalancing procedure. For any base relation K and tuple \mathbf{x} contained in K , the procedure computes the HL-signature *sig* of \mathbf{x} based on the threshold θ and inserts \mathbf{x} into K^{sig} (Line 3). It then recomputes all views in the views trees (Line 4).

Proposition 5.24. *Consider a CQAP query $Q(\mathcal{O}|\mathcal{I})$ whose fracture is hierarchical, a database of size N , and $\epsilon \in [0, 1]$. Given that the view trees constructed for Q in the preprocessing stage can be constructed in $\mathcal{O}(N^p)$ time, major rebalancing of the views in the view trees takes $\mathcal{O}(N^p)$ time.*

Proof. Consider the major rebalancing procedure from Figure 5.23. The relation parts can be computed in $\mathcal{O}(N)$ time. The affected views can be recomputed in the time of the preprocessing stage of the query. □

MAJORREBALANCING(view trees \mathcal{T} , threshold θ)

- 1 $\mathcal{K} :=$ parts of base relations
- 2 **foreach** $K^{sig} \in \mathcal{K}$ **do**
- 3 $K^{sig} := \{\mathbf{x} \rightarrow K(\mathbf{x})$
 $\mid \mathbf{x}$ in base relation $K, \text{ACTUALHLS}(\mathbf{x}, \theta) = sig\}$
- 4 **foreach** $T \in \mathcal{T}$ **do** recompute views in T

Figure 5.23: Recomputing all relation parts and affected views in the view trees \mathcal{T} based on the threshold θ .

MINORREBALANCING(trees \mathcal{T} , value \mathbf{v} , threshold θ)

- 1 $\mathcal{K} :=$ parts of base relations
- 2 **foreach** $K^{sig} \in \mathcal{K}$ **do**
- 3 **foreach** $\mathbf{x} \in \sigma_{\text{Sch}(\mathbf{v})=\mathbf{v}} K^{sig}$ **do**
- 4 $sig' := \text{ACTUALHLS}(\mathbf{x}, \theta)$
- 5 **foreach** $T \in \mathcal{T}$ **do** APPLY($T, \delta K^{sig'} = \{\mathbf{x} \rightarrow K^{sig}(\mathbf{x})\}$)
- 6 **foreach** $T \in \mathcal{T}$ **do** APPLY($T, \delta K^{sig} = \{\mathbf{x} \rightarrow -K^{sig}(\mathbf{x})\}$)

Figure 5.24: Moving tuples \mathbf{x} containing \mathbf{v} to relation parts whose HL-signature matches the degree of \mathbf{v} in base relations.

The cost of major rebalancing is amortized over $\Omega(M)$ updates. After a major rebalancing step, it holds that $N = \frac{1}{2}M$ (after doubling), or $N = \frac{1}{2}M - \frac{1}{2}$ or $N = \frac{1}{2}M - 1$ (after halving). To violate the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ and trigger another major rebalancing, the number of required updates is at least $\frac{1}{4}M$. Hence, the amortized major rebalancing time is $\mathcal{O}(N^{p-1})$. For hierarchical queries, by Proposition 5.27, we have $\delta = w$ or $\delta = w - 1$; hence, the amortized major rebalancing time is $\mathcal{O}(M^{\delta\epsilon})$.

Minor Rebalancing. After an update $\delta R = \{\mathbf{x} \rightarrow m\}$ to relation R , we check the degrees of the values in \mathbf{x} . Consider a partition key k that is included in the schema of \mathbf{x} and the projection \mathbf{v} of \mathbf{x} onto k . If \mathbf{v} is included in a relation part that is light on k but the degree of \mathbf{v} is not below $\frac{3}{2}M^\epsilon$ in at least one base relations, all tuples including \mathbf{v} are moved to relation parts that are heavy on \mathbf{v} . Likewise, if \mathbf{v} is in a relation part that is heavy on k but the degree of \mathbf{v} is below $\frac{1}{2}M^\epsilon$ in all base relations, all tuples including \mathbf{v} are moved to relation parts that are light on \mathbf{v} . Figure 5.24 shows the *minor rebalancing* procedure that moves tuples including \mathbf{v} to relation parts whose HL-signature matches the degree of \mathbf{v} in the base relations. For each tuple \mathbf{x} in a relation part K^{sig} , it first computes the actual HL-signature sig' of \mathbf{x} based on the threshold θ (Line 4). It then inserts \mathbf{x} into $K^{sig'}$ (Line 5) and deletes it from K^{sig} (Line 6).

```

ONUPDATE(view trees  $\mathcal{T}$ , update  $\delta R$ )


---


1  UPDATETREES( $\mathcal{T}$ ,  $\delta R$ )
2  if ( $|\mathcal{D}| = M$ )
3     $M := 2M$ 
4    MAJORREBALANCING( $\mathcal{T}$ ,  $M^\epsilon$ )
5  else if ( $|\mathcal{D}| < \lfloor \frac{1}{4}M \rfloor$ )
6     $M := \lfloor \frac{1}{2}M \rfloor - 1$ 
7    MAJORREBALANCING( $\mathcal{T}$ ,  $M^\epsilon$ )
8  else
9     $\delta R := \{\mathbf{x} \rightarrow m\}$ 
10    $\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\} := \text{TRANSIENTHLS}(\mathbf{x})$ 
11   foreach  $i \in [n]$  do
12     if ( $s_i = L$  and  $\exists K \in \mathcal{D}: |\sigma_{k_i=\mathbf{x}[k_i]}K| \geq \frac{3}{2}M^\epsilon$ ) or
13       ( $s_i = H$  and  $\forall K \in \mathcal{D}: |\sigma_{k_i=\mathbf{x}[k_i]}K| < \frac{1}{2}M^\epsilon$ )
14       MINORREBALANCING( $\mathcal{T}$ ,  $\mathbf{x}[k_i]$ ,  $M^\epsilon$ )

```

Figure 5.25: Updating a set of view trees \mathcal{T} under a sequence of single-tuple updates to base relations. \mathcal{D} is the database. The global variable M is set to $2|\mathcal{D}| + 1$ in the preprocessing stage.

Proposition 5.25. *Consider a CQAP query $Q(\mathcal{O}|\mathcal{I})$ with a hierarchical fracture, a dynamic width δ , a database of size N , and $\epsilon \in [0, 1]$. Given that the view trees constructed for Q in the preprocessing stage can be maintained under a single-tuple update in $\mathcal{O}(N^u)$ time, minor rebalancing of the views in the view trees takes $\mathcal{O}(N^{u+\epsilon})$ time.*

Proof. Figure 5.24 shows the procedure for minor rebalancing of tuples containing the given value v to relation parts whose signature matches the degree of v in base relations. Minor rebalancing either moves $\mathcal{O}(\frac{3}{2}M^\epsilon)$ tuples that have \mathbf{v} to relation parts that are heavy on \mathbf{v} (light to heavy) or $\mathcal{O}(\frac{1}{2}M^\epsilon)$ tuples that have \mathbf{v} to relation parts that are light on \mathbf{v} (heavy to light). Each move is by an insert followed by a delete, which takes $\mathcal{O}(N^u)$ time. Since there are $\mathcal{O}(M^\epsilon)$ such moves and the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$ holds, the total time is $\mathcal{O}(N^u \cdot N^\epsilon) = \mathcal{O}(N^{u+\epsilon})$. \square

The cost of minor rebalancing is amortized over $\Omega(M^\epsilon)$ updates. This lower bound on the number of updates is due to the gap between the two thresholds in the heavy and light part conditions. The amortized time of minor rebalancing is thus $\mathcal{O}(N^u)$.

Figure 5.25 gives the trigger procedure ONUPDATE that maintains a set \mathcal{T} of view trees under a sequence of single-tuple updates to input relations. It first applies an update $\delta R = \{\mathbf{x} \rightarrow m\}$ to the view trees from \mathcal{T} using UPDATETREES from Figure 5.21 (Line 1). If this update leads to a violation of the size invariant $\lfloor \frac{1}{4}M \rfloor \leq N < M$, it invokes MAJORREBALANCING to recompute the relation parts and views (Lines 2-7).

Otherwise, it computes the transient HL-signature $\{k_1 \rightarrow s_1, \dots, k_n \rightarrow s_n\}$ of \mathbf{x} (Line 10). If for any s_i , we have $s_i = L$, but there exists a relation such that the degree of $\mathbf{x}[k_i]$ is at least $\frac{3}{2}M^\epsilon$, or it holds $s_i = H$ but the degree of $\mathbf{x}[k_i]$ is below $\frac{1}{2}M^\epsilon$ in all relations, it invokes MINORREBALANCING to move all tuples containing $\mathbf{x}[k_i]$ to the relation parts whose HL-signature matches the degree of $\mathbf{x}[k_i]$ in base relations (Lines 11-14).

We state the amortized maintenance time of our approach under a sequence of single-tuple updates.

Proposition 5.26. *Consider a CQAP query $Q(\mathcal{O}|\mathcal{I})$ whose fracture is hierarchical, a database of size N , and $\epsilon \in [0, 1]$. Given that the view trees constructed for Q in the preprocessing stage can be constructed in $\mathcal{O}(N^p)$ time and maintained in $\mathcal{O}(N^u)$ for a single-tuple update, maintaining the views in the view trees under a sequence of single-tuple updates takes $\mathcal{O}(N^u)$ amortized time per single-tuple update.*

Proof. We first give the intuition of the proof. By Proposition 5.24, a major rebalancing step requires $\mathcal{O}(N^p)$ time. This time is amortized over $\Omega(N)$ updates executed before the rebalancing step, thus the amortized time of major rebalancing is $\mathcal{O}(N^{p-1})$.

For hierarchical CQAP queries, since $\delta = w$ or $\delta = w - 1$ as per Proposition 5.27, we conclude that the amortized time for major rebalancing is $\mathcal{O}(N^{\delta\epsilon})$. This is exactly the time for processing a single-tuple update, i.e., $\mathcal{O}(N^u)$. Hence, for hierarchical CQAP queries, the amortized major rebalancing time is $\mathcal{O}(N^u)$.

For the minor rebalancing, by Proposition 5.25, a minor rebalancing step requires $\mathcal{O}(N^{u+\epsilon})$ time. This is amortized over $\Omega(N^\epsilon)$ previous updates, which results in $\mathcal{O}(N^u)$ amortized minor rebalancing time.

Overall, both minor and major rebalancing steps take amortized $\mathcal{O}(N^u)$ time, which is the same as the time to process a single-tuple update. Hence, we conclude that the amortized update time under a sequence of single-tuple updates takes $\mathcal{O}(N^u)$ amortized time per single-tuple update.

We next give the formal proof. Let $\mathcal{Z} = (\epsilon, M, \mathcal{T})$ be a state of a database D , where \mathcal{T} is the set of view trees constructed in the preprocessing stage and M is the threshold base of \mathcal{Z} , which is linear to the size of the database. Let $\mathcal{Z}_0 = (\epsilon, M_0, \mathcal{T}_0)$ be the initial state of a database D_0 and u_0, u_1, \dots, u_{n-1} a sequence of arbitrary single-tuple updates. The application of this update sequence to \mathcal{Z}_0 yields a sequence $\mathcal{Z}_0 \xrightarrow{u_0} \mathcal{Z}_1 \xrightarrow{u_1} \dots \xrightarrow{u_{n-1}} \mathcal{Z}_n$ of states, where \mathcal{Z}_{i+1} contains the result of executing the procedure ONUPDATE(\mathcal{T}_i, u_i) from Figure 5.25, for $0 \leq i < n$. Let c_i denote the actual execution cost of ONUPDATE(\mathcal{T}_i, u_i). For some $\Gamma > 0$, we can decompose each c_i as:

$$c_i = c_i^{\text{apply}} + c_i^{\text{major}} + c_i^{\text{minor}} + \Gamma, \quad \text{for } 0 \leq i < n,$$

where c_i^{apply} , c_i^{major} , and c_i^{minor} are the actual costs of the subprocedures UPDATE TREES, MAJOR REBALANCE, and MINOR REBALANCE, respectively, in ONUPDATE. If update u_i causes no major rebalancing, then $c_i^{major} = 0$; similarly, if u_i causes no minor rebalancing, then $c_i^{minor} = 0$. These actual costs admit the following worst-case upper bounds:

$$\begin{aligned} c_i^{apply} &\leq \gamma M_i^u, \\ c_i^{major} &\leq \gamma M_i^p \quad (\text{by Proposition 5.24}), \text{ and} \\ c_i^{minor} &\leq \gamma M_i^{u+\epsilon} \quad (\text{by Proposition 5.25}), \end{aligned}$$

where γ is a constant derived from their asymptotic bounds, and M_i is the threshold base of \mathcal{Z}_i . The costs of major and minor rebalancing can be superlinear in the database size.

The crux of this proof is to show that assigning a *sublinear amortized cost* \hat{c}_i to each update u_i accumulates enough budget to pay for expensive but less frequent rebalancing procedures. For any sequence of n updates, our goal is to show that the accumulated amortized cost is no smaller than the accumulated actual cost:

$$\sum_{i=0}^{n-1} \hat{c}_i \geq \sum_{i=0}^{n-1} c_i. \quad (5.6)$$

The amortized cost assigned to an update u_i is $\hat{c}_i = \hat{c}_i^{apply} + \hat{c}_i^{major} + \hat{c}_i^{minor} + \Gamma$, where

$$\hat{c}_i^{apply} = \gamma N_i^u, \quad \hat{c}_i^{major} = 4\gamma N_i^{p-1}, \quad \hat{c}_i^{minor} = \gamma N_i^u, \quad \text{and}$$

Γ and γ are the constants used to upper bound the actual cost of ONUPDATE. In contrast to the actual costs c_i^{major} and c_i^{minor} , the amortized costs \hat{c}_i^{major} and \hat{c}_i^{minor} are always nonzero.

We prove that such amortized costs satisfy Inequality (5.6). Since $\hat{c}_i^{apply} \geq c_i^{apply}$ for $0 \leq i < n$, it suffices to show that the following inequalities hold:

$$(\textit{amortizing major rebalancing}) \quad \sum_{i=0}^{n-1} \hat{c}_i^{major} \geq \sum_{i=0}^{n-1} c_i^{major} \quad \text{and} \quad (5.7)$$

$$(\textit{amortizing minor rebalancing}) \quad \sum_{i=0}^{n-1} \hat{c}_i^{minor} \geq \sum_{i=0}^{n-1} c_i^{minor}. \quad (5.8)$$

We prove Inequalities (5.7) and (5.8) by induction on the length n of the update sequence.

Major rebalancing.

- *Base case:* We show that Inequality (5.7) holds for $n = 1$. The preprocessing stage sets $M_0 = 2 \cdot N + 1$. If the initial database \mathbf{D}_0 is empty, then $M_0 = 1$ and u_0 triggers major rebalancing (and no minor rebalancing). The amortized cost

$\hat{c}_0^{major} = 4\gamma M_0^{p-1} = 4\gamma$ suffices to cover the actual cost $c_0^{major} \leq \gamma M_0^{1+p-1} = \gamma$. If the initial database is nonempty, u_0 cannot trigger major rebalancing (i.e., violate the size invariant) because $\lfloor \frac{1}{4}M_0 \rfloor = \lfloor \frac{1}{2}N \rfloor \leq N - 1$ (lower threshold) and $N + 1 < M_0 = 2 \cdot N + 1$ (upper threshold); then, $\hat{c}_0^{major} \geq c_0^{major} = 0$. Thus, Inequality (5.7) holds for $n = 1$.

- *Inductive step:* Assumed that Inequality (5.7) holds for all update sequences of length up to $n - 1$, we show it holds for update sequences of length n . If update u_{n-1} causes no major rebalancing, then $\hat{c}_{n-1}^{major} = 4\gamma M_{n-1}^{p-1} \geq 0$ and $c_{n-1}^{major} = 0$, thus Inequality (5.7) holds for n . Otherwise, if applying u_{n-1} violates the size invariant, the database size N_n is either $\lfloor \frac{1}{4}M_{n-1} \rfloor - 1$ or M_{n-1} . Let \mathcal{Z}_j be the state created after the previous major rebalancing or, if there is no such step, the initial state. For the former ($j > 0$), the major rebalancing step ensures $N_j = \frac{1}{2}M_j$ after doubling and $N_j = \frac{1}{2}M_j - \frac{1}{2}$ or $N_j = \frac{1}{2}M_j - 1$ after halving the threshold base M_j ; for the latter ($j = 0$), the preprocessing stage ensures $N_j = \frac{1}{2}M_j - \frac{1}{2}$. The threshold base M_j changes only with major rebalancing, thus $M_j = M_{j+1} = \dots = M_{n-1}$. The number of updates needed to change the database size from N_j to N_n (i.e., between two major rebalancing) is at least $\frac{1}{4}M_{n-1}$ since $\min\{\frac{1}{2}M_j - 1 - (\lfloor \frac{1}{4}M_{n-1} \rfloor - 1), M_{n-1} - \frac{1}{2}M_j\} \geq \frac{1}{4}M_{n-1}$. Then,

$$\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{major} &\geq \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} \hat{c}_i^{major} && \text{(induction hypothesis)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \sum_{i=j}^{n-1} 4\gamma M_{n-1}^{p-1} && (M_j = \dots = M_{n-1}) \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + \frac{1}{4}M_{n-1} 4\gamma M_{n-1}^{p-1} && \text{(at least } \frac{1}{4}M_{n-1} \text{ updates)} \\
&= \sum_{i=0}^{j-1} c_i^{major} + \gamma M_{n-1}^{1+p-1} \\
&\geq \sum_{i=0}^{j-1} c_i^{major} + c_{n-1}^{major} = \sum_{i=0}^{n-1} c_i^{major} && (c_j^{major} = \dots = c_{n-2}^{major} = 0).
\end{aligned}$$

Thus, Inequality (5.7) holds for update sequences of length n .

Minor rebalancing. When the degree of a tuple of values in a partition changes such that the heavy or light part condition no longer holds, minor rebalancing moves the affected tuples between the heavy and light parts of the partition. To prove Inequality (5.8),

we decompose the cost of minor rebalancing per relation part, partition key, and data values of the partition key.

$$c_i^{minor} = \sum_{R^{sig} \in \mathcal{R}} \sum_{k \in keys(sig)} \sum_{key \in Dom(k)} c_i^{R^{sig}, k, key} \quad \text{and}$$

$$\hat{c}_i^{minor} = \sum_{R^{sig} \in \mathcal{R}} \sum_{k \in keys(sig)} \sum_{key \in Dom(k)} \hat{c}_i^{R^{sig}, k, key}$$

We write $c_i^{R^{sig}, k, key}$ and $\hat{c}_i^{R^{sig}, k, key}$ to denote the actual and respectively amortized costs of minor rebalancing caused by update u_i , for a relation part R^{sig} and a tuple key with the schema key whose value comes from u_i . Consider the update u_i of the form $\delta R = \{\mathbf{t} \rightarrow m\}$. If update u_i triggers minor rebalancing, then the cost is $\sum_{R^{sig} \in \mathcal{R}} \sum_{k \in keys(sig)} c_i^{R^{sig}, k, \pi_k \mathbf{t}} = c_i^{minor}$; otherwise, $\sum_{R^{sig} \in \mathcal{R}} \sum_{k \in keys(sig)} c_i^{R^{sig}, k, \pi_k \mathbf{t}} = 0$. The amortized cost is $\sum_{R^{sig} \in \mathcal{R}} \sum_{k \in keys(sig)} \hat{c}_i^{R^{sig}, k, \pi_k \mathbf{t}} = \hat{c}_i^{minor}$ regardless of whether u_i causes minor rebalancing or not.

We prove that for a relation part R^{sig} a relation R , any partition key $k \in keys(sig)$ on which R is partitioned, and any $key \in Dom(k)$ the following inequality holds:

$$\sum_{i=0}^{n-1} \hat{c}_i^{R^{sig}, k, \pi_k \mathbf{t}} \geq \sum_{i=0}^{n-1} c_i^{R^{sig}, k, \pi_k \mathbf{t}}. \quad (5.9)$$

Since the number of relation partitions of a relation is constant, Inequality (5.8) follows directly from Inequality (5.9). We prove Inequality (5.9) by induction on the length n of the update sequence.

- *Base case:* We show that Inequality (5.9) holds for $n = 1$. Assume that update u_0 is of the form $\delta R = \{\mathbf{t} \rightarrow m\}$; otherwise, $\hat{c}_0^{R^{sig}, k, \pi_k \mathbf{t}} = c_0^{R^{sig}, k, \pi_k \mathbf{t}} = 0$, and Inequality (5.9) follows trivially for $n = 1$. If the initial database is empty, u_0 triggers major rebalancing but no minor rebalancing, thus $\hat{c}_0^{R^{sig}, k, \pi_k \mathbf{t}} = \gamma M_0^u \geq c_0^{R^{sig}, k, \pi_k \mathbf{t}} = 0$. If the initial database is nonempty, each relation is partitioned using the threshold M_0^ϵ . For update u_0 to trigger the minor rebalancing of R^{sig} , the degree of the tuple key over the partition key k in R^{sig} has to either decrease from $\lceil M_0^\epsilon \rceil$ to $\lceil \frac{1}{2} M_0^\epsilon \rceil - 1$ (heavy to light) or increase from $\lceil M_0^\epsilon \rceil - 1$ to $\lceil \frac{3}{2} M_0^\epsilon \rceil$ (light to heavy). The former happens only if $\lceil M_0^\epsilon \rceil = 1$ and update u_0 removes the last tuple with the tuple key from R^{sig} , thus no minor rebalancing is needed; the latter cannot happen since update u_0 can increase $|\sigma_{k=key} R^{sig}|$ to at most $\lceil M_0^\epsilon \rceil$, and $\lceil M_0^\epsilon \rceil < \lceil \frac{3}{2} M_0^\epsilon \rceil$. In any case, $\hat{c}_0^{R^{sig}, k, \pi_k \mathbf{t}} \geq c_0^{R^{sig}, k, \pi_k \mathbf{t}}$, implying that Inequality (5.9) holds for $n = 1$.
- *Inductive step:* Assuming that Inequality (5.9) holds for all update sequences of length up to $n - 1$, we show it holds for update sequences of length n . Consider that update

u_{n-1} is of the form $\delta R = \{\mathbf{t} \rightarrow m\}$ and causes minor rebalancing for the relation part R^{sig} ; otherwise, $\hat{c}_{n-1}^{R^{sig},k,\pi_k\mathbf{t}} \geq 0$ and $c_{n-1}^{R^{sig},k,\pi_k\mathbf{t}} = 0$, and Inequality (5.9) follows trivially for n . Let \mathcal{Z}_j be the state created after the previous major rebalancing or, if there is no such step, the initial state. The threshold changes only with major rebalancing, thus $M_j = M_{j+1} = \dots = M_{n-1}$. Depending on the existence of minor rebalancing steps since state \mathcal{Z}_j , we have two cases:

- *Case 1:* There is no minor rebalancing for R^{sig} caused by an update of the form since state \mathcal{Z}_j ; thus, $c_j^{R^{sig},k,\pi_k\mathbf{t}} = \dots = c_{n-2}^{R^{sig},k,\pi_k\mathbf{t}} = 0$. From state \mathcal{Z}_j to state \mathcal{Z}_n , the number of tuples with *key* either decreases from at least $\lceil M_j^\epsilon \rceil$ to $\lceil \frac{1}{2}M_{n-1}^\epsilon \rceil - 1$ (heavy to light) or increases from at most $\lceil M_j^\epsilon \rceil - 1$ to $\lceil \frac{3}{2}M_{n-1}^\epsilon \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than $\frac{1}{2}M_{n-1}^\epsilon$ since $M_j = M_{n-1}$ and $\min\{\lceil M_j^\epsilon \rceil - (\lceil \frac{1}{2}M_{n-1}^\epsilon \rceil - 1), \lceil \frac{3}{2}M_{n-1}^\epsilon \rceil - (\lceil M_j^\epsilon \rceil - 1)\} > \frac{1}{2}M_{n-1}^\epsilon$. Then,

$$\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{R^{sig},k,\pi_k\mathbf{t}} &\geq \sum_{i=0}^{j-1} c_i^{R^{sig},k,\pi_k\mathbf{t}} + \sum_{i=j}^{n-1} \hat{c}_i^{R^{sig},k,\pi_k\mathbf{t}} && \text{(induction hypothesis)} \\
&= \sum_{i=0}^{j-1} c_i^{R^{sig},k,\pi_k\mathbf{t}} + \sum_{i=j}^{n-1} \gamma M_{n-1}^u && (M_j = \dots = M_{n-1}) \\
&> \sum_{i=0}^{j-1} c_i^{R^{sig},k,\pi_k\mathbf{t}} + M_{n-1}^\epsilon \gamma M_{n-1}^u && \text{(more than } M_{n-1}^\epsilon \text{ updates)} \\
&\geq \sum_{i=0}^{j-1} c_i^{R^{sig},k,\pi_k\mathbf{t}} + c_{n-1}^{R^{sig},k,\pi_k\mathbf{t}} \\
&= \sum_{i=0}^{n-1} c_i^{R^{sig},k,\pi_k\mathbf{t}} && (c_j^{R^{sig},k,\pi_k\mathbf{t}} = \dots = c_{n-2}^{R^{sig},k,\pi_k\mathbf{t}} = 0).
\end{aligned}$$

- *Case 2:* There is at least one minor rebalancing step for R^{sig} caused by an update of the form $\delta R = \{\mathbf{t}' \rightarrow m'\}$ where $\pi_k\mathbf{t}' = \text{key}$ since state \mathcal{Z}_j . Let \mathcal{Z}_ℓ denote the state created after the previous minor rebalancing caused by an update of this form; thus, $c_\ell^{R^{sig},k,\pi_k\mathbf{t}} = \dots = c_{n-2}^{R^{sig},k,\pi_k\mathbf{t}} = 0$. The minor rebalancing steps creating \mathcal{Z}_ℓ and \mathcal{Z}_n inserts or deletes tuples with the \mathcal{S} tuples *key*. From state \mathcal{Z}_ℓ to state \mathcal{Z}_n , the number of such tuples either decreases from $\lceil \frac{3}{2}M_\ell^\epsilon \rceil$ to $\lceil \frac{1}{2}M_{n-1}^\epsilon \rceil - 1$ (heavy to light) or increases from $\lceil \frac{1}{2}M_\ell^\epsilon \rceil - 1$ to $\lceil \frac{3}{2}M_{n-1}^\epsilon \rceil$ (light to heavy). For this change to happen, the number of updates needs to be greater than M_{n-1}^ϵ since $M_\ell = M_{n-1}$ and $\min\{\lceil \frac{3}{2}M_\ell^\epsilon \rceil - (\lceil \frac{1}{2}M_{n-1}^\epsilon \rceil - 1), \lceil \frac{3}{2}M_{n-1}^\epsilon \rceil - (\lceil \frac{1}{2}M_\ell^\epsilon \rceil - 1)\} > M_{n-1}^\epsilon$. Then,

$$\begin{aligned}
\sum_{i=0}^{n-1} \hat{c}_i^{R^{sig},k,\pi_k \mathbf{t}} &\geq \sum_{i=0}^{\ell-1} c_i^{R^{sig},k,\pi_k \mathbf{t}} + \sum_{i=\ell}^{n-1} \hat{c}_i^{R^{sig},k,\pi_k \mathbf{t}} && \text{(induction hypothesis)} \\
&= \sum_{i=0}^{\ell-1} c_i^{R^{sig},k,\pi_k \mathbf{t}} + \sum_{i=\ell}^{n-1} \gamma M_{n-1}^u && (M_j = \dots = M_{n-1}) \\
&> \sum_{i=0}^{\ell-1} c_i^{R^{sig},k,\pi_k \mathbf{t}} + M_{n-1}^\epsilon \gamma M_{n-1}^u && \text{(more than } M_{n-1}^\epsilon \text{ updates)} \\
&> \sum_{i=0}^{\ell-1} c_i^{R^{sig},k,\pi_k \mathbf{t}} + c_{n-1}^{R^{sig},k,\pi_k \mathbf{t}} \\
&= \sum_{i=0}^{n-1} c_i^{R^{sig},k,\pi_k \mathbf{t}} && (c_\ell^{R^{sig},k,\pi_k \mathbf{t}} = \dots = c_{n-2}^{R^{sig},k,\pi_k \mathbf{t}} = 0).
\end{aligned}$$

Cases 1 and 2 imply that Inequality (5.9) holds for update sequences of length n .

This shows that Inequality (5.6) holds when the amortized cost of $\text{ONUPDATE}(\mathcal{T}_i, u_i)$ is

$$\hat{c}_i = \gamma M_i^u + 4\gamma M_i^{p-1} + \gamma M_i^\epsilon + \Gamma, \quad \text{for } 0 \leq i < n,$$

where Γ and γ are constants. The amortized cost \hat{c}_i^{major} of major rebalancing is $4\gamma M_i^{p-1}$, and the amortized cost \hat{c}_i^{minor} of minor rebalancing is γM_i^ϵ . From the size invariant $\lfloor \frac{1}{4}M_i \rfloor \leq N_i < M_i$ follows that $N_i < M_i < 4(N_i + 1)$ for $0 \leq i < n$, where N_i is the database size before update u_i . This implies that for any database of size N , the amortized major rebalancing time is $\mathcal{O}(N^{p-1})$ and the amortized minor rebalancing time is $\mathcal{O}(N^u)$. From $w - 1 \leq \delta$ in Proposition 5.27, it follows that the overall amortized update time is $\mathcal{O}(N^{\delta\epsilon})$. □

5.5 Complexity Analysis

We can now prove Theorem 3.5 by combining the results of the previous sections. The theorem is copied here for convenience.

Theorem 3.5. *Consider any CQAP query Q with static width w and dynamic width δ , a database of size N , and $\epsilon \in [0, 1]$. If Q 's fracture is hierarchical, then Q admits $\mathcal{O}(N^{1+(w-1)\epsilon})$ preprocessing time, $\mathcal{O}(N^{1-\epsilon})$ enumeration delay, and $\mathcal{O}(N^{\delta\epsilon})$ amortized update time for single-tuple updates.*

Proof. Consider a CQAP query Q with static width w and dynamic width δ . Assume that the fracture Q_{\dagger} of Q is hierarchical. In the preprocessing stage, we construct a set of view trees representing the result of Q_{\dagger} . These view trees can be materialized in $\mathcal{O}(N^{1+(w-1)\epsilon})$ time (Propositions 5.10) and can be maintained with $\mathcal{O}(N^{\delta\epsilon})$ amortised time under single-tuple updates (Proposition 5.23). Given any input tuple, the view trees allow for the enumeration of the result of Q with $\mathcal{O}(N^{1-\epsilon})$ enumeration delay (Proposition 5.19). \square

The static width of a query is greater or equal to the dynamic width. For hierarchical queries, the difference can be 0 or 1.

Proposition 5.27. *For any CQAP query with a hierarchical fracture, static width w and dynamic width δ , it holds that either $\delta = w$ or $\delta = w - 1$.*

Proof. Let Q be a CQAP query with a hierarchical fracture, static width w and dynamic width δ . The fracture Q_{\dagger} of Q is hierarchical and has static width w and dynamic width δ . Lemma 5.13 and the definition of static and dynamic width imply that $w, \delta \in \mathbb{N}_0$ and $\delta \leq w$. It remains to show that $w - 1 \leq \delta$. For the sake of contradiction, assume that $w - 1 > \delta$. Let ω be an access-top variable order that gives the static width w and dynamic width δ for Q_{\dagger} . Our assumption $w - 1 > \delta$ implies:

$$\begin{aligned} & \forall X \in \text{vars}(Q_{\dagger}), \forall R(\mathcal{Y}) \in \text{atoms}(\omega_X) : \\ & \rho^*((\{X\} \cup \text{dep}_{\omega}(X)) - \mathcal{Y}) \leq \delta < w - 1. \end{aligned} \quad (5.10)$$

Since the static width of Q_{\dagger} is w , it holds:

$$\exists Y \in \text{vars}(Q_{\dagger}) : \rho^*(\{Y\} \cup \text{dep}_{\omega}(Y)) = w. \quad (5.11)$$

We show that Statements (5.10) and (5.11) are contradicting, which completes the proof. Let X be an arbitrary variable in $\text{vars}(Q_{\dagger})$ and $R(\mathcal{Y})$ any atom in $\text{atoms}(\omega_X)$. Let $\lambda = (\lambda_{K(\mathcal{X})})_{K(\mathcal{X}) \in \text{atoms}(Q)}$ be a fractional edge cover of $(\{X\} \cup \text{dep}_{\omega}(X)) - \mathcal{Y}$ such that

$$\sum_{K(\mathcal{X}) \in \text{atoms}(Q)} \lambda_{K(\mathcal{X})} = \rho^*((\{X\} \cup \text{dep}_{\omega}(X)) - \mathcal{Y}).$$

Due to Statement (5.10), it holds

$$\sum_{K(\mathcal{X}) \in \text{atoms}(Q)} \lambda_{K(\mathcal{X})} < w - 1. \quad (5.12)$$

Let $\lambda' = (\lambda'_{K(\mathcal{X})})_{K(\mathcal{X}) \in \text{atoms}(Q)}$ be defined as

$$\lambda'_{K(\mathcal{X})} = \begin{cases} 1, & \text{if } K(\mathcal{X}) = R(\mathcal{Y}) \\ \lambda_{K(\mathcal{X})}, & \text{otherwise} \end{cases}$$

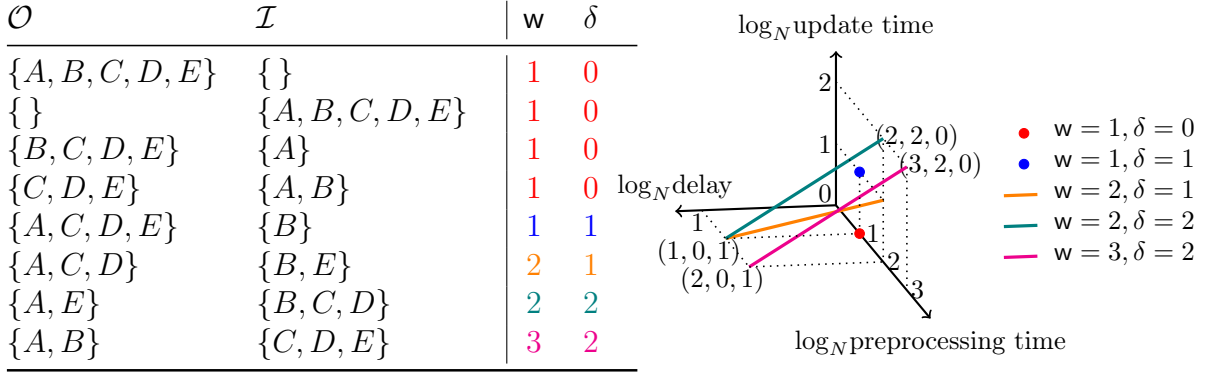


Figure 5.26: Left: The static width w and dynamic width δ of the query in Example 5.28 for different access patterns ($\mathcal{O} \mid \mathcal{I}$). Right: The trade-offs between the preprocessing time, update time, and enumeration delay of the corresponding queries on the left.

Clearly, λ' is a fractional edge cover of $\{X\} \cup \text{dep}_w(X)$. Moreover, due to Inequality (5.12), it holds that $\sum_{K(X) \in \text{atoms}(Q)} \lambda'_{K(X)} < w$. Since X was chosen arbitrarily from $\text{vars}(Q_+)$, this means that for any $X \in \text{vars}(Q)$, we have $\rho^*(\{X\} \cup \text{dep}_w(X)) < w$. However, this contradicts Statement (5.11). \square

The following example shows how the widths of a query can change dependent on the input and output variable sets of the query.

Example 5.28. Consider the query

$$Q(\mathcal{O} \mid \mathcal{I}) = R(A, B, C), S(A, B, D), T(A, E).$$

Figure 5.26 (left) gives static widths w and dynamic widths δ of the query for different access patterns ($\mathcal{O} \mid \mathcal{I}$). Figure 5.26 (right) shows the complexities for preprocessing time, update time, and enumeration delay for the query with different access patterns as a function of $\epsilon \in [0, 1]$. \square

Recall the eager and lazy approaches introduced in Chapter 1. Our trade-off may achieve a lower computation time when compared against the eager approach in case we only need to enumerate a part P of the result instead of the entire result. To achieve the overall minimum computation time, we may decide on how much to offload to update and what should be the enumeration delay as functions of the size of P . We demonstrate this in the following example.

Example 5.29. Consider the 4-cycle query:

$$Q(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D).$$

Assume that all four relations have size N .

The eager approach first precomputes the results of 4-cycle $Q(A, B, C, D)$ in $\mathcal{O}(N^2)$ time using a worst-case optimal join algorithm [58]. On a single-tuple update δR to relation R , it computes the delta

$$\delta Q(a, b, C, D) = \delta R(a, b), S(b, C), T(C, D), U(a, D).$$

This takes $\mathcal{O}(N)$ time: for each pair of C - and D -values that are paired with b in S and a in U , we look it up in T . For a given tuple (b, d) over B and D , the (A, C) -tuples in $Q(A, C \mid b, d)$ can be enumerated with constant delay. Overall, the eager approach takes $\mathcal{O}(N^2)$ preprocessing time, $\mathcal{O}(N)$ update time and $\mathcal{O}(1)$ enumeration delay.

The lazy approach has no precomputation and only updates the base relations for each update. To answer an access request for a given tuple (b, d) over B and D , it first calibrates the relations in the residual queries $Q(A, C) = R(A, b), S(b, C), T(C, d), U(A, d)$. This takes linear time. After that, it can enumerate the pairs of values over $\{A, C\}$ with constant delay. Overall, the lazy approach takes $\mathcal{O}(1)$ preprocessing time, $\mathcal{O}(1)$ update time and $\mathcal{O}(N)$ enumeration delay.

Our approach admits $\mathcal{O}(N^{1+\epsilon})$ preprocessing time, $\mathcal{O}(N^\epsilon)$ update time and $\mathcal{O}(N^{1-\epsilon})$ delay for any $\epsilon \in [0, 1]$. The complexities for the eager and lazy approaches can be recovered using our approach by setting $\epsilon = 1$ and respectively $\epsilon = 0$ (except for preprocessing in the lazy approach):

Approach	Preprocessing	Update	Delay
Eager	$\mathcal{O}(N^2)$	$\mathcal{O}(N)$	$\mathcal{O}(1)$
Lazy	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$
Ours	$\mathcal{O}(N^{1+\epsilon})$	$\mathcal{O}(N^\epsilon)$	$\mathcal{O}(N^{1-\epsilon})$

Consider now a sequence of updates, each followed by one access request to enumerate k out of the maximum possible $\mathcal{O}(N^2)$ pairs of values. The time to process an update and answer an access request is (**update + k * delay**), which means $\mathcal{O}(N + k)$ time for the eager approach, $\mathcal{O}(1 + (N + k))$ for the lazy approach and $\mathcal{O}(N^\epsilon + kN^{1-\epsilon})$ for our approach. The table below shows the complexity of processing an update and answering k access requests for a various of k . The five columns show the complexities for the approaches for various value of k . The last row states the value of ϵ , for which the complexities of our approach in the same columns are obtained. Depending on the value of k , we can tune our approach to minimize its complexity. For $1 \leq k < N$, our approach has consistently lower complexity than the lazy/eager approaches (the green cells), while for $k \geq N$ it matches the complexity of the lazy/eager approaches (the yellow cells).

	0	0.5	$\log_N k$ 1	1.5	2
<i>Eager/Lazy</i>	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N^{1.5})$	$\mathcal{O}(N^2)$
<i>Ours</i>	$\mathcal{O}(N^{0.5})$	$\mathcal{O}(N^{0.75})$	$\mathcal{O}(N)$	$\mathcal{O}(N^{1.5})$	$\mathcal{O}(N^2)$
ϵ	0.5	0.75	1	1	1

□

Recovery of Prior Results. Our result, Theorem 3.5, can recover prior results on CQs without access patterns, i.e., CQAP queries with no input variable, in both the static and dynamic settings.

We first discuss the static evaluation, i.e., compute the query results and then enumerate them: we find the access-top variable order that has the optimal static width w , and then precompute the data structure that represents the query results in $\mathcal{O}(N^w)$ time and enumerate the results with $\mathcal{O}(N^{1-\epsilon})$ delay. By appropriately setting ϵ , this trade-off recovers prior results on CQs restricted to hierarchical queries (Figure 5.27 left): by setting $\epsilon = 0$, both the preprocessing time $\mathcal{O}(N^{1+(w-1)\epsilon})$ and the delay $\mathcal{O}(N^{1-\epsilon})$ become $\mathcal{O}(N)$, as for α -acyclic queries [9]; by setting $\epsilon = 1$, we obtain $\mathcal{O}(N^w)$ preprocessing time and $\mathcal{O}(1)$ delay as for conjunctive queries [65]. For free-connex queries, $w = 1$ and the preprocessing time remains $\mathcal{O}(N)$ regardless of ϵ ; we then choose $\epsilon = 1$ to obtain $\mathcal{O}(1)$ delay [9]. For bounded-degree databases, i.e., where each value appears at most c times for some constant $c = N^\beta$, first-order queries admit $\mathcal{O}(N)$ preprocessing time and $\mathcal{O}(1)$ delay [28, 46]. We recover the $\mathcal{O}(1)$ delay using $\epsilon = 1$. The preprocessing time becomes $\mathcal{O}(N \cdot (N^\beta)^{w-1}) = \mathcal{O}(N)$ if our approach uses the constant upper bound c instead of the upper bound N^ϵ on the degrees.

In the dynamic case, our approach recovers prior work on conjunctive [62], free-connex [38], and q-hierarchical [14] queries by setting $\epsilon = 1$ (Figure 5.27 right). For general hierarchical queries, our approach then achieves the same complexities as prior work on conjunctive queries. For free-connex queries, we obtain linear-time preprocessing and update and constant-time delay since $w = 1$ and $\delta \in \{0, 1\}$. For q-hierarchical queries, we obtain linear-time preprocessing and constant-time update and delay since $w = 1$ and $\delta = 0$. Existing maintenance approaches, e.g., classical first-order IVM [23] and higher-order recursive IVM [50], DynYannakakis [38], and F-IVM [62], can achieve constant delay for general hierarchical queries yet after at least linear-time updates.

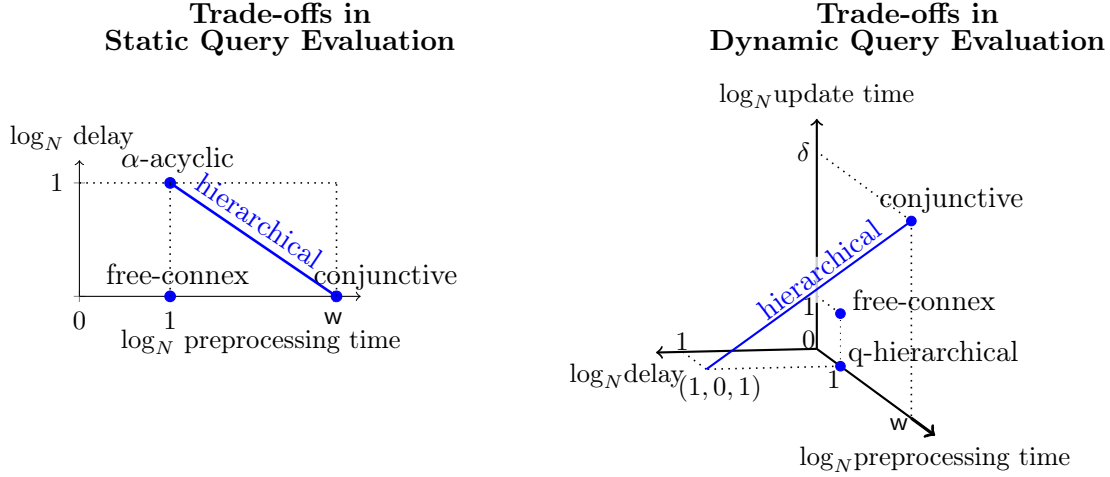


Figure 5.27: Trade-offs in static and dynamic evaluation for hierarchical queries. Our approach achieves each blue point and each point on the blue lines. Prior approaches are represented by one point in the trade-off space.

5.6 Optimality Result

Our trade-offs for CQAP queries with hierarchical fractures (Theorem 3.5) are optimal for $CQAP_0$ and $CQAP_1$ queries, unless the OMv conjecture is false. Before showing this, we first show that the $CQAP_0$ and $CQAP_1$ queries are exactly those queries with dynamic width 0 and 1, respectively.

Proposition 5.30. *A query is $CQAP_0$ if and only if it has dynamic width 0. A query is $CQAP_1$ if and only if it has dynamic width 1.*

The proposition directly follows from the following Lemmas 5.31 and 5.32.

Lemma 5.31. *Any $CQAP_i$ query with $i \in \{0, 1\}$ has dynamic width at least i .*

Lemma 5.32. *Any $CQAP_i$ query with $i \in \{0, 1\}$ has dynamic width at most i .*

It remains to prove Lemmas 5.31 and 5.32.

Proof of Lemma 5.31. Let Q be a $CQAP_i$ query for some $i \in \{0, 1\}$. Since the dynamic width of a CQAP query is greater or equal to 0, the case $i = 0$ is trivial. Assume now that Q is a $CQAP_1$ query and consider its fracture Q_{\dagger} . By definition of $CQAP_1$ queries, Q_{\dagger} is hierarchical and almost free-dominant or almost input-dominant. Assume first that Q_{\dagger} is almost free-dominant. This means that Q_{\dagger} contains a bound variable X and an atom $R(\mathcal{Y}) \in atoms(X)$ such that:

$$free(atoms(X)) \not\subseteq \mathcal{Y} \tag{5.13}$$

Let ω be an arbitrary access-top variable order for Q_{\dagger} . Since the schema of each atom in $atoms(X)$ contains X , all variables in $free(atoms(X))$ depend on X . Hence, each variable in $free(atoms(X))$ is on a root-to-leaf path with X . Since X is bound, the variables in $free(atoms(X))$ cannot be contained in ω_X . Hence, they are contained in $\mathbf{anc}_{\omega}(X)$. This implies that $free(atoms(X)) \subseteq (\{X\} \cup dep_{\omega}(X))$. By Assumption (5.13), $\rho((\{X\} \cup dep_{\omega}(X)) - \mathcal{Y})$ is at least 1. This implies that $\rho^*((\{X\} \cup dep_{\omega}(X)) - \mathcal{Y})$ is at least 1 (Lemma 5.13). It follows that $\delta(\omega) \geq 1$. Since ω is an arbitrary access-top variable order for Q_{\dagger} , we derive that the dynamic width of Q is at least 1.

The case that the fracture Q_{\dagger} is almost input-dominant is handled analogously. The query Q_{\dagger} contains an output variable X and an atom $R(\mathcal{Y}) \in atoms(X)$ such that:

$$in(atoms(X)) \not\subseteq \mathcal{Y} \quad (5.14)$$

Consider any access-top variable order ω for Q_{\dagger} . Since X is output, the variables in $in(atoms(X))$ are contained in $\mathbf{anc}_{\omega}(X)$. This means that $in(atoms(X)) \subseteq (\{X\} \cup dep_{\omega}(X))$. By Assumption (5.14), $\rho^*((\{X\} \cup dep_{\omega}(X)) - \mathcal{Y})$ is at least 1. It follows that $\delta(\omega) \geq 1$. Therefore, the dynamic width of Q is at least 1. \square

Proof of Lemma 5.32. Consider a CQAP₀ query Q and its fracture Q_{\dagger} . The query Q_{\dagger} admits a canonical variable order ω where all free variables are above the bound ones and all input variables are above the output variables. Hence, ω is access-efficient. Consider a variable X in ω and an atom $R(\mathcal{Y}) \in atoms(X)$. Since ω is canonical, it holds $dep_{\omega}(X) = \mathbf{anc}_{\omega}(X)$ and $R(\mathcal{Y})$ has $\{X\} \cup \mathbf{anc}_{\omega}(X)$ in its schema. Hence, $(\{X\} \cup dep_{\omega}(X)) - \mathcal{Y} = (\{X\} \cup \mathbf{anc}_{\omega}(X)) - \mathcal{Y} = \emptyset$. This means that $\rho^*((\{X\} \cup dep_{\omega}(X)) - \mathcal{Y})$ is 0. Since we chose X and $R(\mathcal{Y}) \in atoms(X)$ arbitrarily, this implies that the dynamic width of ω is 0. Hence, the dynamic width of Q_{\dagger} and, therefore, of Q is 0.

Now, assume that Q is a CQAP₁ with fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I})$. Let ω be the canonical variable order of Q_{\dagger} . By Lemma 5.11, the function ACC-TOP($\omega, \mathcal{O}, \mathcal{I}$) in Figure 5.1 (Section 5.2.1) constructs an access-top variable order ω^t for Q_{\dagger} with dynamic width $\kappa(\omega, \mathcal{I}, \mathcal{O})$, where

$$\kappa(\omega, \mathcal{I}, \mathcal{O}) = \max_{\substack{Y \in bound(\omega) \\ Z \in out(\omega)}} \max_{R(\mathcal{Y}) \in atoms(\omega_Y)} \{\rho^*((vars(\omega_Y) \cap \mathcal{F}) - \mathcal{Y}), \rho^*((vars(\omega_Z) \cap \mathcal{I}) - \mathcal{Y})\}$$

with $\mathcal{F} = \mathcal{I} \cup \mathcal{O}$. Recall that Q_{\dagger} is almost free- or almost input-dominant. Consider an arbitrary variable X in ω and an atom $R(\mathcal{Y})$ containing X . If X is bound, then $\rho^*((vars(\omega_X) \cap \mathcal{F}) - \mathcal{Y})$ can be at most 1. Similarly, if X is output, then $\rho^*((vars(\omega_X) \cap \mathcal{I}) - \mathcal{Y})$ can be at most 1. It follows that $\kappa(\omega, \mathcal{I}, \mathcal{O})$ is at most 1. This implies that ω^t is an access-top variable order for Q_{\dagger} with dynamic width at most 1. We conclude that the dynamic width of Q is at most 1. \square

Optimality Results for CQAP₀ Queries. Queries in CQAP₀ are those queries that have a dynamic width of 0 (Proposition 5.30). Both our approaches for arbitrary CQAP queries (Theorem 3.4) and for CQAP queries with hierarchical fractures (Theorem 3.5 with $\epsilon = 1$) admit constant update time and delay. This is matched by the lower bound conditioned on the OMv conjecture (first statement in Theorem 3.11). Thus, our result on CQAP₀ queries is optimal.

Optimality Results for CQAP₁ Queries. Theorem 3.5 can be refined for CQAP₁, since $\delta = 1$ and $w \leq 2$ for queries in this class. By setting $\epsilon = 0.5$, our approach admits the complexities as shown in Corollary 5.33.

Corollary 5.33. *(Theorem 3.5) Any CQAP₁ query Q admits $\mathcal{O}(N^{1+\frac{(w-1)}{2}})$ preprocessing time, $\mathcal{O}(N^{\frac{1}{2}})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}})$ amortized update time for single-tuple updates, where N is the database size and w is the static width of Q .*

This is matched by the lower bound conditioned on the OMv conjecture (second statement in Theorem 3.11). Thus, our result on CQAP₁ queries achieves weakly Pareto worst-case optimal: there can be no tighter upper bounds for *both* the update time and delay. This is summarized in the following corollary.

Corollary 5.34 (Theorem 3.11). *Given a CQAP₁ query Q without repeating relation symbols, $\gamma > 0$, and a database of size N , there is no algorithm that computes Q with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, unless the OMv conjecture fails.*

The class CQAP₁ contains both acyclic and cyclic queries, e.g., the 4-cycle query $Q(A, C \mid B, D) = R(A, B), S(B, C), T(C, D), U(A, D)$ with input variables B, D and output variables A, C . In contrast to CQAP₀, CQAP₁ is not maximal for the given complexities, e.g., counting triangles needs the same update time and delay [41, 42].

Chapter 6

Trade-Offs in Dynamic Evaluation for Triangle CQAP Queries

In this chapter, we introduce our approach for the dynamic evaluation of triangle CQAP queries. It uncovers the trade-offs between the update and enumeration complexities for triangle CQAP queries. The approach is similar to the one for CQAP queries with hierarchical fractures in Chapter 5, except that we use a different view tree structure for the triangle queries.

Recall we categorize the triangle CQAP queries into five categories, i.e., \mathcal{C}_{lookup} , \mathcal{C}_{count} , \mathcal{C}_1 , \mathcal{C}_2 , and \mathcal{C}_3 queries. Theorem 3.10 summarizes the trade-offs between the update and enumeration complexities for the triangle CQAP queries in different categories. In the following sections, we discuss the preprocessing, enumeration and update stages for each category. We discuss in detail the \mathcal{C}_3 and \mathcal{C}_2 queries, as they are the most representative ones, and discuss only the differences for the other categories. At the end of the chapter, in Section 6.7, we prove the optimality of our approach.

We consider in the following a constant $\epsilon \in [0, 1]$ and a database of size N .

6.1 The \mathcal{C}_3 Query

We focus on the queries in the \mathcal{C}_3 class. This class contains only one query, which is to enumerate all triangles in the database:

$$Q(A, B, C \mid \cdot) = R(A, B), S(B, C), T(C, A).$$

6.1.1 Preprocessing

We partition the relations R , S , and T on the variables A , B , and C , with the threshold N^ϵ , into the following relation parts:

$$R^{A \rightarrow s_A, B \rightarrow s_B}(A, B), S^{B \rightarrow s_B, C \rightarrow s_C}(B, C), \text{ and } T^{C \rightarrow s_C, A \rightarrow s_A}(C, A), \quad \text{where } s_A, s_B, s_C \in \{H, L\}.$$

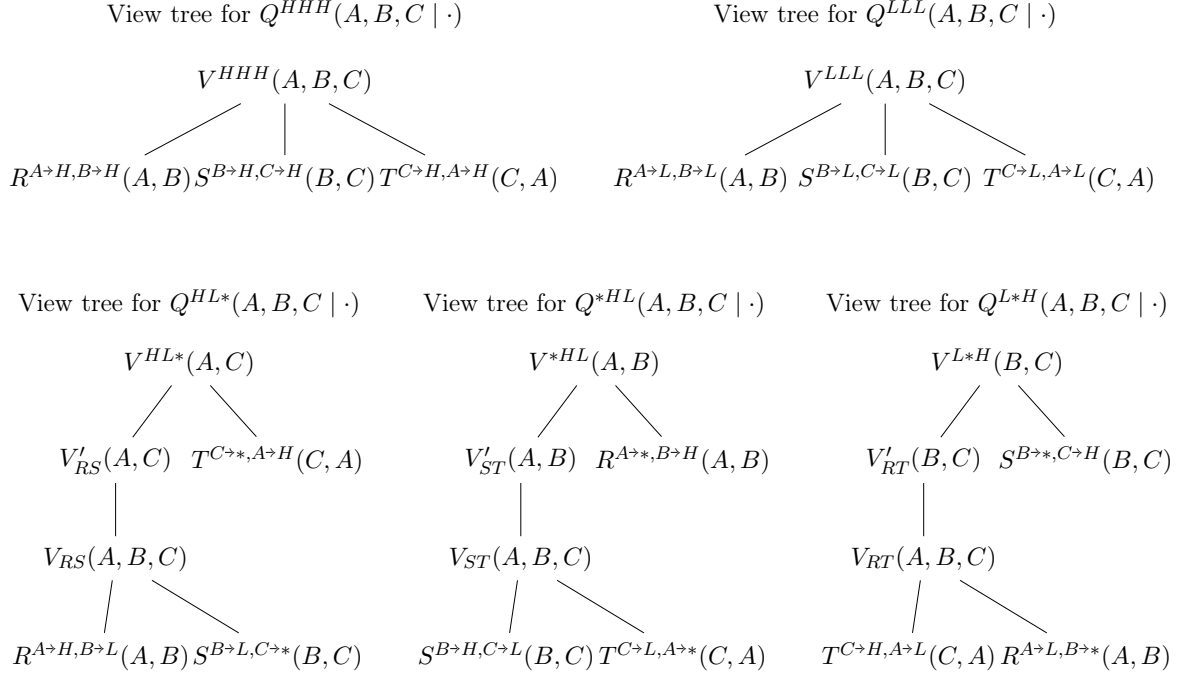


Figure 6.1: The view trees for maintaining the results of the skew-aware queries for the \mathbb{C}_3 query $Q(A, B, C | \cdot)$. The wildcard $*$ can be either H or L in a view tree.

The \mathbb{C}_3 query can then be decomposed into eight skew-aware queries expressed over these relation parts:

$$Q^{s_A s_B s_C}(A, B, C | \cdot) = R^{A \rightarrow s_A, B \rightarrow s_B}(A, B), S^{B \rightarrow s_B, C \rightarrow s_C}(B, C), T^{C \rightarrow s_C, A \rightarrow s_A}(C, A),$$

where $s_A, s_B, s_C \in \{H, L\}$. Each skew-aware query lists the triangles in the corresponding relation parts. For example, the skew-aware query $Q^{LLL}(A, B, C | \cdot)$ lists the triangles in the join of the relation parts $R^{A \rightarrow L, B \rightarrow L}(A, B)$, $S^{B \rightarrow L, C \rightarrow L}(B, C)$, and $T^{C \rightarrow L, A \rightarrow L}(C, A)$, which are the parts of R , S , and T with light A -, B -, and respectively C -values. The result of the \mathbb{C}_3 query Q is the union of the results of the skew-aware queries:

$$Q(A, B, C | \cdot) = \sum_{s_A s_B s_C \in \{H, L\}} Q^{s_A s_B s_C}(A, B, C | \cdot).$$

Since these results are disjoint, enumerating the result of Q is equivalent to enumerating the result of each of these queries one after the other.

To maintain the result of Q , we need to maintain the result of each skew-aware query. We adapt a maintenance strategy to each skew-aware query to allow for amortized update time that is sublinear in the database size.

Figures 6.1 and 6.2 present the view trees for the maintenance of the \mathbb{C}_3 query $Q(A, B, C | \cdot)$ under updates to the base relations. The results of the skew-aware queries Q^{HHH} and Q^{LLL} are materialized in listing form in the views V_{HHH} and respectively

Materialized View Definition	Computation Time
View tree for $Q^{HHH}(A, B, C \mid \cdot)$ $V^{HHH}(A, B, C) = R^{A \rightarrow H, B \rightarrow H}(A, B), S^{B \rightarrow H, C \rightarrow H}(B, C), T^{C \rightarrow H, A \rightarrow H}(C, A)$	$\mathcal{O}(N^{\frac{3}{2}})$
View tree for $Q^{LLL}(A, B, C \mid \cdot)$ $V^{LLL}(A, B, C) = R^{A \rightarrow L, B \rightarrow L}(A, B), S^{B \rightarrow L, C \rightarrow L}(B, C), T^{C \rightarrow L, A \rightarrow L}(C, A)$	$\mathcal{O}(N^{\frac{3}{2}})$
View tree for $Q^{HL*}(A, B, C \mid \cdot)$ $V_{RS}(A, B, C) = R^{A \rightarrow H, B \rightarrow L}(A, B), S^{B \rightarrow L, C \rightarrow *}(B, C)$ $V'_{RS}(A, C) = V_{RS}(A, B, C)$ $V^{HL*}(A, C) = V'_{RS}(A, C), T^{C \rightarrow *, A \rightarrow H}(C, A)$	$\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N)$
View tree for $Q^{*HL}(A, B, C \mid \cdot)$ $V_{ST}(A, B, C) = S^{B \rightarrow H, C \rightarrow L}(B, C), T^{C \rightarrow L, A \rightarrow *}(C, A)$ $V'_{ST}(A, B) = V_{ST}(A, B, C)$ $V^{*HL}(A, B) = V'_{ST}(A, B), R^{A \rightarrow *, B \rightarrow H}(A, B)$	$\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N)$
View tree for $Q^{L*H}(A, B, C \mid \cdot)$ $V_{RT}(A, B, C) = T^{C \rightarrow H, A \rightarrow L}(C, A), R^{A \rightarrow L, B \rightarrow *}(A, B)$ $V'_{RT}(B, C) = V_{RT}(A, B, C)$ $V^{L*H}(B, C) = V'_{RT}(B, C), S^{B \rightarrow *, C \rightarrow H}(B, C)$	$\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ $\mathcal{O}(N)$

Figure 6.2: The definition and time complexity for computing the materialized views for the \mathcal{C}_3 query $Q(A, B, C \mid \cdot)$. The wildcard $*$ can be either H or L in a view tree.

V_{LLL} , the root views of the view trees in the first row of Figure 6.1. The results of the remaining six skew-aware queries are distributed in the corresponding view trees, which allow for constant-delay enumeration. For example, consider the view tree for Q^{HL*} (left bottom in Figure 6.1). The result of Q^{HL*} is distributed among two auxiliary materialized views, V^{HL*} and V_{RS} . The former stores all (a, c) pairs that would appear in the result of Q^{HL*} , while the latter provides the matching B -values for each (a, c) pair. The two views together provide constant-delay enumeration of the result of Q^{HL*} . In addition to them, the view V'_{RS} serves to support constant-time updates to $T^{C \rightarrow *, A \rightarrow H}(C, A)$. The view trees for Q^{*HL} and Q^{L*H} are analogous.

In these two figures, the wildcard $*$ is either H or L in a view tree. We use such wildcard to avoid repeating the view trees with the same structure for different skew-aware queries. The wildcard is used in the remaining figures in this chapter.

We analyze the time to construct the views for the \mathcal{C}_{count} query. Figure 6.2 summarizes the time to compute these views. Strictly partitioning the input relations using the threshold N^ϵ takes $\mathcal{O}(N)$ time. Computing the skew-aware queries Q^{HHH} and Q^{LLL} using the worst-case optimal join algorithms takes $\mathcal{O}(N^{\frac{3}{2}})$ time [58].

Consider the view tree for the skew-aware query Q^{HL*} (top-left in Figure 6.1). To compute the view $V_{RS}(A, C) = R^{A \rightarrow H, B \rightarrow L}(A, B), S^{B \rightarrow L, C \rightarrow *}(B, C)$, one can iterate over all (B, C) -pairs (b, c) in $S^{B \rightarrow L, C \rightarrow *}$ and then find the A -values in $R^{A \rightarrow H, B \rightarrow L}$ for each b . Since B is light, the relation part $R^{A \rightarrow H, B \rightarrow L}$ contains at most N^ϵ distinct A -values for any B -value, which gives an upper bound of $|S^{B \rightarrow L, C \rightarrow *}| \cdot N^\epsilon$ on the size of V_{RS} . Meanwhile, since A is heavy, the relation part $R^{A \rightarrow H, B \rightarrow L}$ contains at most $N^{1-\epsilon}$ distinct A -values, which gives an upper bound of $|S^{B \rightarrow L, C \rightarrow *}| \cdot N^{1-\epsilon}$ on the size of V_{RS} . We can use either $R^{A \rightarrow H, B \rightarrow L}$ or $S^{B \rightarrow L, C \rightarrow *}$ as the outer relation. Hence, the number of steps needed to compute V_{RS} is upper-bounded by $\min\{|S^{B \rightarrow L, C \rightarrow *}| \cdot N^\epsilon, |S^{B \rightarrow L, C \rightarrow *}| \cdot N^{1-\epsilon}\} = \mathcal{O}(\min\{N \cdot N^\epsilon, N \cdot N^{1-\epsilon}\}) = \mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$.

Computing $V'_{RS}(A, C) = V_{RS}(A, B, C)$ requires aggregating the variable B away. This takes time linear to the size of the view V_{RS} , which is $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$ as explained in the paragraph above.

Computing the view $V^{HL*}(A, C) = V_{RS}(A, C), T^{C \rightarrow *, A \rightarrow H}(C, A)$ requires iterating over all (A, C) -pairs in V_{RS} and looking up the pair in $T^{C \rightarrow *, A \rightarrow H}(C, A)$. Each lookup takes constant time. Since V_{RS} has size $\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$, the time to compute Q^{HL*} is upper-bounded by $|V_{RS}| \cdot 1 = \mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$.

The analysis for the view trees for Q^{*HL} and Q^{L*H} is analogous.

Overall, since $\max_{\epsilon \in [0,1]} \{1 + \min\{\epsilon, 1 - \epsilon\}\} = \frac{3}{2}$, the materialized views for the \mathbb{C}_{count} query can be constructed in $\mathcal{O}(N^{\frac{3}{2}})$ time.

6.1.2 Enumeration

We discuss the enumeration for the query. We create view iterators over the view trees in Figure 6.1. For the skew-aware queries $Q^{HHH}(A, B, C | \cdot)$ and $Q^{LLL}(A, B, C | \cdot)$, since their results are materialized in the root views $V^{HHH}(A, B, C)$ and $V^{LLL}(A, B, C)$ of the view trees, we create the two view iterators $\text{it}_{V^{HHH}}(A, B, C)$ and respectively $\text{it}_{V^{LLL}}(A, B, C)$. They enumerate the results of Q^{HHH} and Q^{LLL} from the two root views with constant delay.

Figure 6.3 shows the enumeration procedure for the left-bottom view tree from Figure 6.1 for the skew-aware query $Q^{HL*}(A, B, C | \cdot)$. We create the view iterators for this view tree top-down. We create the iterator $\text{it}_{V^{HL*}}(A, B)$ at the root view V^{HL*} to enumerate its (A, C) -tuples (Lines 1-2) and the iterator $\text{it}_{V_{RS}}(B|A, C)$ at V_{RS} , which takes each (A, C) -tuple from V^{HL*} as input and enumerates the paired B -values in V_{RS} (Lines 3-4). Each concatenation of the outputs from the two iterators forms a tuple in the result of Q^{HL*} (Line 6). The multiplicity of the output tuple is the product of the multiplicities of the values of the output tuple in the relation parts (Line 5). Both iterators take constant time to report the next tuple, so the enumeration delay for $Q^{HL*}(A, B, C | \cdot)$ is constant.

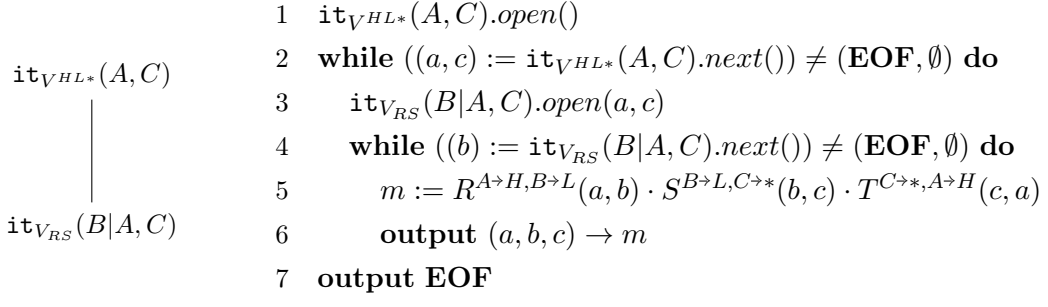


Figure 6.3: Enumeration for the skew-aware query $Q^{HL^*}(A, B, C | \cdot)$ using the bottom-right view tree from Figure 6.1.

The enumeration for the skew-aware queries $Q^{*HL}(A, B, C | \cdot)$ and $Q^{L*H}(A, B, C | \cdot)$ is analogous.

The result of the query $Q(A, B, C | \cdot)$ is the union of the disjoint results of the skew-aware queries. To enumerate the result of the query Q , we enumerate each tuple in the results of the skew-aware query one after the other. Since the enumeration delay for each skew-aware query is constant, the overall enumeration delay is also constant.

6.1.3 Update

We consider the update for the \mathbb{C}_3 query. Figure 6.4 shows the delta view trees for the view trees for the \mathbb{C}_3 query (Figure 6.1) under the single-tuple update $\delta R = \{(a, b) \rightarrow m\}$ to R . The delta view trees for an update to S and T are analogous.

In case the tuple (a, b) is heavy on both A and B , the update δR affects the part $R^{A \rightarrow H, B \rightarrow H}(A, B)$ of R . In this case, we update the relation part $R^{A \rightarrow H, B \rightarrow H}(A, B)$ with $\delta R^{A \rightarrow H, B \rightarrow H}(a, b) = \delta R(a, b)$ in the view trees for Q^{HHH} and Q^{HHL} , and propagate the delta from the affected relation parts to the root views. For the view tree for Q^{HHH} (top left), we update $V^{HHH}(A, B, C)$ with $\delta V^{HHH}(a, b, C) = \delta R^{A \rightarrow H, B \rightarrow H}(a, b), S^{B \rightarrow H, C \rightarrow H}(b, C), T^{C \rightarrow H, A \rightarrow H}(C, a)$ in $\mathcal{O}(N^{1-\epsilon})$ time, since $S^{B \rightarrow H, C \rightarrow H}(b, C)$ and $T^{C \rightarrow H, A \rightarrow H}(C, a)$ are heavy on C . For the view tree for Q^{HHL} (bottom middle), we update the root view $V^{HHL}(A, B)$ with $\delta V^{HHL}(a, b) = \delta R^{A \rightarrow H, B \rightarrow H}(a, b), V'_{ST}(a, b)$ in constant time.

In case the tuple (a, b) is light on both A and B , the update δR affects the relation part $R^{A \rightarrow L, B \rightarrow L}(A, B)$, and we need to update the view trees for Q^{LLL} and Q^{LLH} . In the view tree for Q^{LLL} , we update the view $V^{LLL}(A, B, C)$ with $\delta V^{LLL}(a, b, C) = \delta R^{A \rightarrow L, B \rightarrow L}(a, b), S^{B \rightarrow L, C \rightarrow L}(b, C), T^{C \rightarrow L, A \rightarrow L}(C, a)$ in $\mathcal{O}(N^\epsilon)$ time, since there are $\mathcal{O}(N^\epsilon)$ C -values in the views $S^{B \rightarrow L, C \rightarrow L}(b, C)$ and $T^{C \rightarrow L, A \rightarrow L}(C, a)$. In the view tree for Q^{LLH} , we update $V_{RT}(A, B, C)$ with $\delta V_{RT}(a, b, C) = \delta R^{A \rightarrow L, B \rightarrow L}(a, b), T^{C \rightarrow H, A \rightarrow L}(C, a)$ in $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ time, since $T^{C \rightarrow H, A \rightarrow L}(C, A)$ is light on A and heavy on C . Similarly, we update $V'_{RT}(B, C)$

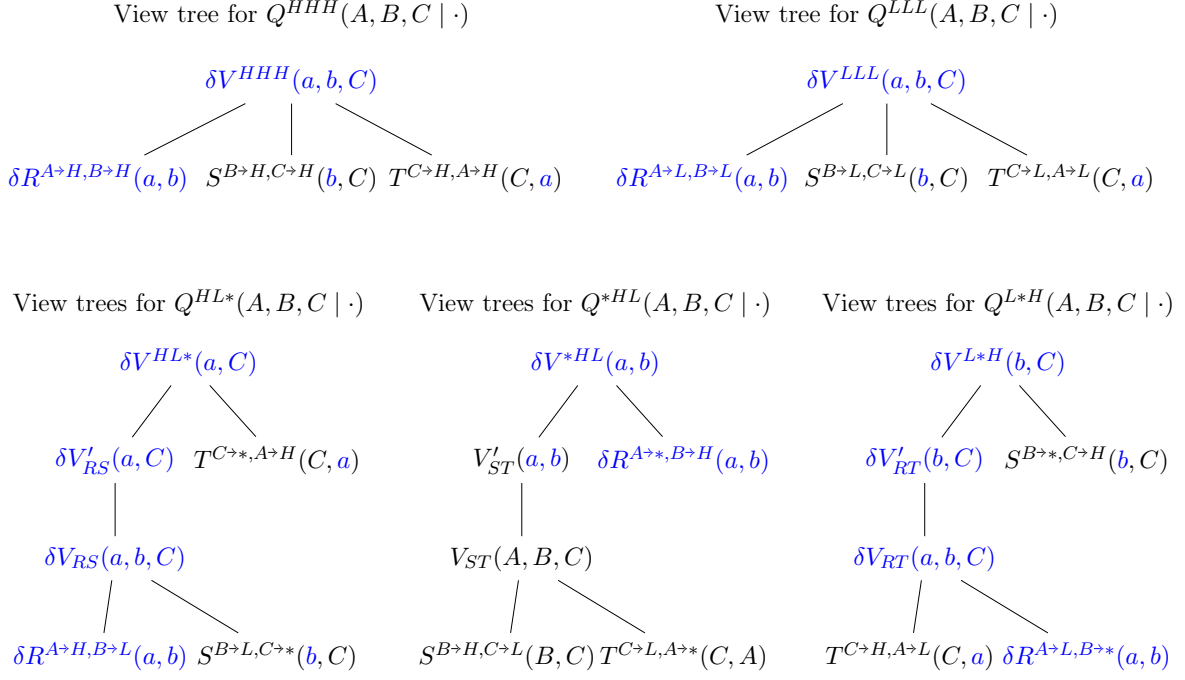


Figure 6.4: Delta view trees for the view trees in Figure 6.1 under the single-tuple update $\delta R = \{(a, b) \rightarrow m\}$ to R . The blue views in the view trees are the deltas to the corresponding views.

with $\delta V'_{RT}(b, C) = \delta V_{RT}(a, b, C)$ in constant time and $V^{LLH}(B, C)$ with $\delta V^{LLH}(b, C) = \delta V'_{RT}(b, C), S^{B \to L, C \to H}(b, C)$ in $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ time.

In case the tuple (a, b) is heavy on A and light on B , the update δR affects the relation part $R^{A \to H, B \to L}(A, B)$, and we need to update the view trees for Q^{HLH} and Q^{HLL} . We update $V_{RS}(A, B, C)$ with $\delta V_{RS}(a, b, C) = \delta R^{A \to H, B \to L}(a, b), S^{B \to L, C \to *}(b, C)$ in $\mathcal{O}(N^\epsilon)$ time. We then update $V'_{RS}(A, C)$ with $\delta V'_{RS}(a, C) = \delta V_{RS}(a, b, C)$ in constant time and $V^{HL*}(A, C)$ with $\delta V^{HL*}(a, C) = \delta V'_{RS}(a, C), T^{C \to *, A \to H}(C, a)$ in $\mathcal{O}(N^\epsilon)$ time.

In case the tuple (a, b) is light on A and heavy on B , the update δR affects the relation part $R^{A \to L, B \to H}(A, B)$, and we need to update the view trees for Q^{LHH} and Q^{LHL} . The updates are similar as those for Q^{LLH} and Q^{HHL} , which take $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ and $\mathcal{O}(1)$ time, respectively.

Overall, the update of the view trees for the \mathbb{C}_3 query takes $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ time.

Following the discussion in the previous sections, we conclude the following proposition for the trade-offs of the \mathbb{C}_3 query:

Proposition 6.1. *Given a \mathbb{C}_3 query Q , a database of size N , and $\epsilon \in [0, 1]$, the query Q can be evaluated with $\mathcal{O}(N^{\frac{3}{2}})$ time preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

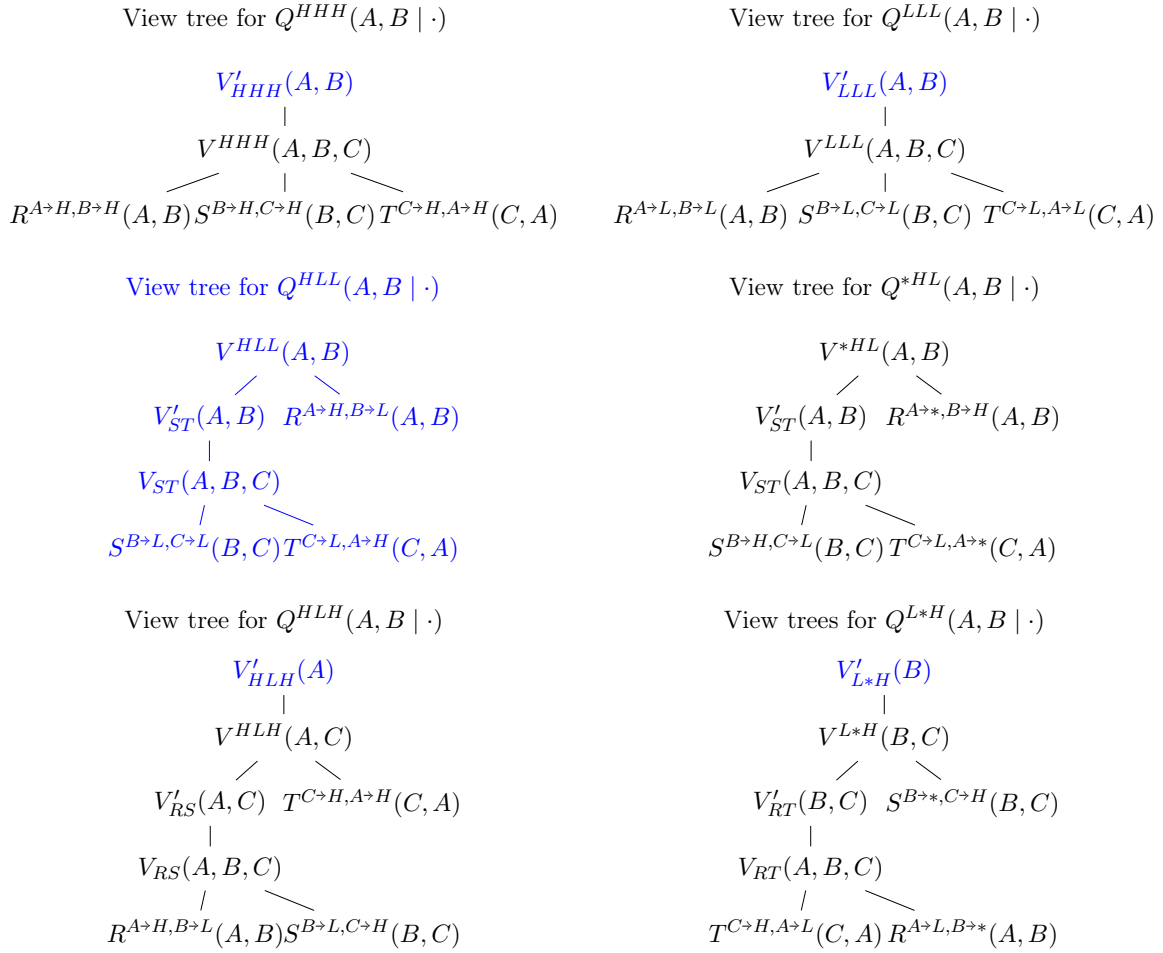


Figure 6.5: The view trees supporting the maintenance of the results of the skew-aware queries for the \mathbb{C}_2 query $Q(A, B | \cdot)$. These view trees are adapted from those for the \mathbb{C}_3 query. The adaptations are highlighted in blue. The wildcard $*$ can be either H or L .

6.2 Queries in \mathbb{C}_2

We now discuss the queries in \mathbb{C}_2 . These queries have two input variables, or two output variables and no input variable, which includes $Q(A, B | \cdot)$, $Q(\cdot | A, B)$, $Q(C | A, B)$ and their symmetries.

6.2.1 Preprocessing

We present the preprocessing stage of the first query

$$Q(A, B | \cdot) = R(A, B), S(B, C), T(C, A).$$

The preprocessing stage for the other queries is same, that is, we build the same view trees; only the enumeration of their results are different.

We apply a similar strategy as for the \mathbf{C}_3 query. We partition the relations R , S and T on variables A , B and C , with the threshold N^ϵ , and then decompose $Q(A, B \mid \cdot)$ into skew-aware queries defined over the relation parts:

$$Q^{\mathbf{s}_A \mathbf{s}_B \mathbf{s}_C}(A, B \mid \cdot) = R^{A \rightarrow \mathbf{s}_A, B \rightarrow \mathbf{s}_B}(A, B), S^{B \rightarrow \mathbf{s}_B, C \rightarrow \mathbf{s}_C}(B, C), T^{C \rightarrow \mathbf{s}_C, A \rightarrow \mathbf{s}_A}(C, A),$$

where $\mathbf{s}_A, \mathbf{s}_B, \mathbf{s}_C \in \{H, L\}$. The result of the query $Q(A, B \mid \cdot)$ is the union of the results of these skew-aware queries.

Compared to the \mathbf{C}_3 query, this \mathbf{C}_2 query faces two new challenges. First, the results of the skew-aware queries are not disjoint anymore, which causes difficulties in the enumeration of distinct (A, B) -tuples among the results of the skew-aware queries. We can resolve this difficulty using the UNION algorithm, as we did for CQAP queries with hierarchical fractures, in Section 5.3.3. Second, among the view trees created for the \mathbf{C}_3 query from Figure 6.1, only the view tree for Q^{*HL} (middle) allows constant-delay enumeration of (A, B) -tuples, while the view trees for Q^{HL*} (left) and Q^{L*H} (right) allow constant-delay enumeration of (A, C) - and respectively (B, C) -tuples but not (A, B) -tuples. We need to adapt the view trees and use union view iterators to tackle this difficulty. We discuss in detail how to tackle these difficulties in the next section.

Figure 6.5 shows the view trees for the \mathbf{C}_2 query $Q(A, B \mid \cdot)$. These view trees are adapted from those for the \mathbf{C}_3 query; the adaptations are highlighted in blue. For the skew-aware queries $Q^{HHH}(A, B \mid \cdot)$, $Q^{LLL}(A, B \mid \cdot)$, $Q^{HLL}(A, B \mid \cdot)$ and $Q^{*HL}(A, B \mid \cdot)$ (first two rows), their results are materialized in the root views of the view trees; they allow constant-delay enumeration of the results.

For the skew-aware query $Q^{HLH}(A, B \mid \cdot)$ (bottom left), we construct a new root view $V'_{HLH}(A)$ on top of the original root view $V^{HLH}(A, C)$, which aggregates away the bound variable C . This is to allow the constant-delay enumeration of distinct A -values in the result of the query Q^{HLH} . Similarly, for the skew-aware query $Q^{L*H}(A, B \mid \cdot)$ (bottom right), we construct additionally a root view $V'_{L*H}(B)$ that aggregates the bound variable C away from the original root view $V^{L*H}(B, C)$ to allow the constant-delay enumeration of the distinct B -values. These two view trees do not support the constant-delay enumeration of the distinct (A, B) -tuples; They can be enumerated with sublinear delay.

The time to compute these additional views is upper-bounded $\mathcal{O}(N^{\frac{3}{2}})$, which is the same as the time to compute the views for the \mathbf{C}_3 CQAP queries. Hence, the overall time complexity of the preprocessing stage is $\mathcal{O}(N^{\frac{3}{2}})$.

6.2.2 Enumeration

We next discuss how to enumerate the results of \mathbf{C}_2 queries using the view trees in Figure 6.5. We show the enumeration for $Q(A, B \mid \cdot)$, $Q(\cdot \mid A, B)$ and $Q(C \mid A, B)$. The

	1	$ctx_0() := \{()\}$, empty input
$uit_{V'_{HLH}}(A)$	2	$uit_{V'_{HLH}}(A).open()$
	3	while $((a, ctx_a) := uit_{V'_{HLH}}(A).next()) \neq (\mathbf{EOF}, \emptyset)$ do
	4	$ctx_{V_{RS}} := V^{HLH}(A, C) \bowtie \{a\}$
	5	$uit_{V_{RS}}(B A, C).open(ctx_{V_{RS}})$
$uit_{V_{RS}}(B A, C)$	6	while $((b, ctx_b) := uit_{V_{RS}}(B A, C).next()) \neq (\mathbf{EOF}, \emptyset)$ do
	7	$m := \sum_{(_, c) \in ctx_b} R^{A \rightarrow H, B \rightarrow L}(a, b) \cdot S^{B \rightarrow L, C \rightarrow H}(b, c) \cdot T^{C \rightarrow H, A \rightarrow H}(c, a)$
	8	output $(a, b) \rightarrow m$
	9	output EOF

Figure 6.6: Enumeration for the skew-aware query $Q^{HLH}(A, B)$ using the bottom-left view tree from Figure 6.5.

enumeration for other symmetric \mathbf{C}_2 queries is analogous.

Query $Q(A, B | \cdot)$. Consider first the query

$$Q(A, B | \cdot) = R(A, B), S(B, C), T(C, A).$$

For the skew-aware queries $Q^{HHH}(A, B | \cdot)$, $Q^{LLL}(A, B | \cdot)$, $Q^{HHL}(A, B | \cdot)$, and $Q^{*HL}(A, B | \cdot)$, their results are materialized in the root views of the corresponding view trees, so we create view iterators over the root views to enumerate the results with constant delay. For the other skew-aware queries, we need to use the union view iterators.

Consider the skew-aware query $Q^{HLH}(A, B | \cdot)$. Figure 6.6 shows the enumeration procedure for the view tree for the query. We construct two iterators $uit_{V'_{HLH}}(A)$ and $uit_{V_{RS}}(B|A, C)$ for the two free variables A and B . The iterator $uit_{V'_{HLH}}(A)$ enumerates the A -values in the root view V^{HLH} (Lines 2-3). The iterator $uit_{V_{RS}}(B|A, C)$ is unsupported as there is no binding for variable C . For this iterator, we provide a relation over schema (A, C) as context. To avoid enumerating dangling tuples, the context should include only those C -values guaranteed to have matching (A, C) -tuples in the final output. The ancestor view $V^{HLH}(A, C)$ provides such (A, C) -tuples, which we further restrict to those matching the output A -value from $uit_{V_{RS}}(B|A, C)$ (Line 4). The $next()$ call on $uit_{V_{RS}}$ returns the input B -value together with a relation ctx_b containing the matching (A, C) -tuples in V_{RS} if they exist; otherwise, it returns $(\mathbf{EOF}, \emptyset)$. The tuples in the relation ctx_b are used to compute the multiplicity of the output tuple (Line 7).

The $open$ and $next$ calls take time linear in the size of the context ctx used when opening the iterator. The size of the context for $uit_{V'_{HLH}}$ is constant, while for $uit_{V_{RS}}$ is at most the size of the C -values paired with a fixed A -value in V^{HLH} . Given that V^{HLH} is over the relation parts that are heavy on A , the number of distinct C -values in V^{HLH} is at most $N^{1-\epsilon}$. Thus, the enumeration delay is $\mathcal{O}(N^{1-\epsilon})$.

```

hitV( $\mathcal{O}|\mathcal{I}$ ).open(relation ctx)

```

```

1 hitV( $\mathcal{O}|\mathcal{I}$ ).iterators := empty map    // context tuple  $\mapsto$  view iterator
2 foreach ctxi  $\in$  ctx do
3   hitV( $\mathcal{O}|\mathcal{I}$ ).iterators[ctxi] := new hitV( $\mathcal{O}|\mathcal{I}$ )    // lazily
4   hitV( $\mathcal{O}|\mathcal{I}$ ).iterators[ctxi].open(ctxi)    // lazily

```

Figure 6.7: The method opens the hop view iterator $\text{hit}_V(\mathcal{O}|\mathcal{I})$ for the input relation ctx over schema \mathcal{I} as context. The method lazily creates for each tuple in ctx a view iterator and opens the view iterator for the corresponding tuple. That is, these iterators are created and opened only on their first *next* call.

This enumeration delay can be improved by using another type of iterators, called *hop view iterators*, which we discuss next.

6.2.2.1 Hop View Iterators

The union view iterator relies on the UNION algorithm to enumerate the distinct tuples from possibility overlapping sets. An alternative method is to use the skip pointers [16], which allows “jumping” over already reported tuples when iterating over these sets. The skip pointer approach outperforms the union view iterators for some triangle CQAP queries, though it requires additional space to store the skip pointers and is more complex to implement. In this section, we define the *hop view iterators*, which uses the skip pointer approach to implement the interface of the union view iterators.

Extension of View Iterators. To support the skip pointer approach, we extend the view iterator introduced in Section 4.2.1 with two new methods: *isExhausted* and *exclude*. The *isExhausted()* method checks whether the next tuple of the iterator is **EOF**. The *exclude(**t**)* method excludes an arbitrary tuple **t** over \mathcal{O} in the enumeration of the iterator: excluded tuples are omitted by the *next* method during the enumeration.

The *exclude(**t**)* method can be implemented similarly to the delete function of a linked list: to exclude a tuple **t**, the iterator looks up the tuple **t** and records a pointer pointing from the predecessor of **t** to the successor of **t**. During the enumeration, when the *next* function is called, if such a pointer exists, the iterator returns the tuple pointed by the pointer; otherwise, it returns the next tuple. Looking up the input tuple **t** and recording the pointer both take constant time. Hence, the time complexity of *exclude(**t**)* is $\mathcal{O}(1)$.

Hop View Iterators. We now discuss the hop view iterators. They are similar to the union view iterators. Figure 6.7 shows the $\text{hit}_V(\mathcal{O}|\mathcal{I}).\text{open}()$ method. It differs from the

```

hitV( $\mathcal{O}|\mathcal{I}$ ).next() : (tuple, relation)

```

```

1 o := HOPUNION(hitV( $\mathcal{O}|\mathcal{I}$ ))
2 ctxo :=  $\pi_{\mathcal{I}}\sigma_{\mathcal{O}=\mathbf{o}}V$ 
3 return (o, ctxo)

```

Figure 6.8: Fetch the next output tuple in the union view iterator, computed by the HOPUNION function, and the context tuples that are consistent with the output tuple.

```

HOPUNION(hop view iterator hitV( $\mathcal{O}|\mathcal{I}$ ): tuple)

```

```

1 ctx1  $\mapsto$  it1( $\mathcal{O}|\mathcal{I}$ ), ..., ctxn  $\mapsto$  itn( $\mathcal{O}|\mathcal{I}$ ) := hitV( $\mathcal{O}|\mathcal{I}$ ).iterators
2 if n = 0
3   return EOF
4 t := it1( $\mathcal{O}|\mathcal{I}$ ).next()
5 foreach ctx  $\in$   $\pi_{\mathcal{I}}\sigma_{\mathcal{O}=\mathbf{t}}V$ :
6   itctx := hitV( $\mathcal{O}|\mathcal{I}$ ).iterators[ctx]
7   itctx.exclude(t)
8   if (itctx.isExhausted())
9     hitV( $\mathcal{O}|\mathcal{I}$ ).iterators.remove(ctx  $\mapsto$  itctx( $\mathcal{O}|\mathcal{I}$ ))
10 return t

```

Figure 6.9: Return the next distinct tuple from a set of iterators in hit_V($\mathcal{O}|\mathcal{I}$).iterators.

open method of the union view iterators in that the iterators in hit_V($\mathcal{O}|\mathcal{I}$).iterators are created lazily (Lines 3-4); that is, these iterators are created and opened only on their first next call. The lazy initialization allows the open method to run in constant time.

Figure 6.8 shows the hit_V($\mathcal{O}|\mathcal{I}$).next() method. Instead of using the UNION algorithm, the method uses the HOPUNION function (Figure 6.9) to enumerate the distinct tuples from the set of n iterators it₁, ..., it_n stored in hit_V.iterators. Let S_i be the set of tuples that it_i enumerates. The HOPUNION function first enumerates the tuples from S_1 using it₁, then those from $S_2 \setminus S_1$ using it₂, then those from $S_3 \setminus S_2 \setminus S_1$ using it₃, and so on. When a tuple is reported by an iterator, the function skips the tuple from other iterators, and when an iterator is exhausted, the function skips the iterator. The enumeration delay in this case would depend on the time needed to exclude a just reported tuple from the iterators and the time to exclude exhausted iterators.

Figure 6.9 shows the HOPUNION function. When there are no iterators stored in hit_V.iterators, the function returns **EOF** (Lines 2-3); otherwise, it gets the next tuple **t**

from the first iterator (Line 4). The tuples $\pi_{\mathcal{I}\sigma_{\mathcal{O}=\mathbf{t}}}V$ are those context tuples paired with \mathbf{t} in V ; The iterators opened for these context tuples will enumerate \mathbf{t} . By exploiting the skew information, the number of such iterators might be asymptotically than n , such as the \mathbf{C}_2 query we will discuss next. The function excludes \mathbf{t} from all these iterators to avoid reporting \mathbf{t} again (Lines 5-7). Excluding \mathbf{t} may leave an iterator exhausted. In this case, the HOPUNION function excludes the exhausted iterator from the $\text{hit}_V.\text{iterators}$ (Lines 8-9). Checking whether $\text{hit}_V.\text{iterators}$ has at least one iterator takes constant time (Lines 1-3). Getting the next tuple \mathbf{t} takes constant time (Line 4). Assume for any tuple \mathbf{t} , there are at most k context tuples paired with \mathbf{t} in V , i.e., $|\pi_{\mathcal{I}\sigma_{\mathcal{O}=\mathbf{t}}}V| = \mathcal{O}(k)$. The loop (Lines 5-9) runs $\mathcal{O}(k)$ times, and each loop iteration takes constant time to exclude \mathbf{t} from an iterator (Lines 6-7), constant time to check if the iterator is empty (Line 8), and constant time to exclude an exhausted iterator (Line 9). Hence, the delay of the function HOPUNION is linear to $\mathcal{O}(k)$.

Back to the *next* method. Once the function gets the tuple \mathbf{o} from the HOPUNION function, it computes the context relation for \mathbf{o} by selecting the context tuples that are paired with \mathbf{o} in V (Line 3), which also takes $\mathcal{O}(k)$ time. Overall, the *next* method takes $\mathcal{O}(k)$ time.

Improve the Enumeration Delay using Hop View Iterators. We now back to the enumeration of the skew-aware query $Q^{HLH}(A, B \mid \cdot)$. We improve the enumeration delay by using the hop view iterators instead of union view iterators. We create the hop view iterator $\text{hit}_{V'_{HLH}}(A)$ to enumerate the A -values a in the root view V^{HLH} with constant delay, and the iterator $\text{hit}_{V_{RS}}(B|A, C)$ and open it for the context relation $ctx_{V_{RS}} = V^{HLH}(A, C) \bowtie \{(a)\}$. Since the hop view iterator is opened for the context relation lazily, the context relation does not need to be materialized; it is enough as long as it supports constant-delay enumeration, which is the case per our computational model. Hence, opening $\text{hit}_{V_{RS}}(B|A, C)$ for $ctx_{V_{RS}}$ takes constant time. The iterator $\text{hit}_{V_{RS}}(B|A, C)$ can enumerate the distinct B -values with the delay determined by the number of context (A, C) -tuples in $ctx_{V_{RS}}$ that are paired with any B -value in V_{RS} . For a fixed B -value b , the number of such (A, C) -tuples is $\mathcal{O}(N^\epsilon)$: $ctx_{V_{RS}}$ contains only one fixed A -value a from $\text{hit}_{V'_{HLH}}(A)$ and $\mathcal{O}(N^\epsilon)$ C -values paired with b since the relation parts in this view tree are light on B . The number of such (A, C) -tuples is also bounded by $\mathcal{O}(N^{1-\epsilon})$, since the relation parts in the view tree are heavy on C . Hence, the enumeration delay for the B -values is the min of the two bounds: $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. Overall, the enumeration delay for skew-aware query Q^{HLH} is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. The enumeration for the skew-aware query $Q^{L*H}(A, B)$ is analogous.

We have discussed the enumeration for each skew-aware query of the query $Q(A, B \mid \cdot)$. Since the results of the skew-aware queries are not disjoint, to enumerate the distinct tuples, we use again the UNION algorithm presented in Section 5.3.3. The enumeration delay is the sum of the delays of all these skew-aware queries. Since the number of skew-aware queries is independent of the data size, the overall enumeration delay is the maximum delay of the individual skew-aware queries, which is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. Hence, the enumeration delay for $Q(A, B \mid \cdot)$ is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$.

Query $Q(\cdot \mid A, B)$. Consider now the other \mathbf{C}_2 query $Q(\cdot \mid A, B)$. The enumeration procedure is similar. We create the same iterators over the view trees, except now we set the input variables A and B as the context variables of the iterators. These iterators serve to check whether a given input tuple over (A, B) is in each view tree. The time for checking whether the given tuple exists in the view tree using the iterators is the same as that for enumerating a tuple from the iterators. Hence, the enumeration delay for $Q(A, B \mid \cdot)$ is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$.

Query $Q(C \mid A, B)$. We now discuss the enumeration for the query $Q(C \mid A, B)$. Consider the skew-aware query $Q^{HHH}(C \mid A, B)$. We create the two view iterators $\text{it}_{V'_{HHH}}(A, B \mid A, B)$ and $\text{it}_{V^{HHH}}(C \mid A, B)$ at V'_{HHH} and V^{HHH} . For a given tuple (a, b) over (A, B) , the iterator $\text{it}_{V'_{HHH}}(A, B \mid A, B)$ checks whether (a, b) exists in V'_{HHH} . If yes, the iterator $\text{it}_{V^{HHH}}(C \mid A, B)$ enumerates the C -values that are paired with (a, b) from V^{HHH} with constant delay. The enumeration for the skew-aware query $Q^{LLL}(Q \mid A, B)$ is analogous.

Similarly, for the skew-aware query $Q^{LLL}(C \mid A, B)$, we create the view iterator $\text{it}_{V^{LLL}}(A, B \mid A, B)$ to check whether an input (A, B) -tuple exists in V^{LLL} and the view iterator $\text{it}_{V_{ST}}(C \mid A, B)$ to enumerate the paired C -values from V_{ST} with constant delay. The enumeration for the skew-aware query $Q^{L*H}(C \mid A, B)$ is analogous.

Consider the skew-aware query $Q^{HLH}(C \mid A, B)$ and a given input tuple (a, b) over (A, B) . We create the hop view iterator $\text{hit}_{V'_{HLH}}(A \mid A)$ to check whether a is in $V'_{HLH}(A)$. We then create the hop view iterator $\text{hit}_{V_{RS}}(C \mid A, B, C)$ and open it for the context relation $ctx_{RS}(A, B, C) = V^{HLH}(A, C) \bowtie \{(a, b)\}$. The enumeration delay of the iterator $\text{hit}_{V_{RS}}(C \mid A, B, C)$ is determined by the size of ctx_{RS} . Since the relation parts in this view tree are heavy on C and light on B , the size of ctx_{RS} is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$, and thus the enumeration delay is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. Overall, the enumeration delay for $Q^{HLH}(C \mid A, B)$ is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. The enumeration for the skew-aware query $Q^{L*H}(C \mid A, B)$ is analogous.

Overall, we apply the UNION algorithm to enumerate the distinct tuples of $Q(C \mid A, B)$ from the skew-aware queries with $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ delay.

6.2.3 Update

The update of the view trees is similar to that of the view trees for the \mathcal{C}_3 query in Section 6.1.3. The view trees for the \mathcal{C}_2 query contain views that are not in the view trees for the \mathcal{C}_3 query, i.e., the views in blue in Figure 6.5. Though, these extra views are all views that aggregate away a variable from the views below, such as the view $V'_{HHH}(A, B)$. Maintaining these views under a single-tuple update take $\mathcal{O}(1)$ time. Hence, the update time for \mathcal{C}_2 queries is again $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$.

Following the discussion in the previous sections, we conclude the following proposition for the trade-offs of the \mathcal{C}_2 query:

Proposition 6.2. *Given a \mathcal{C}_2 query Q , a database of size N , and $\epsilon \in [0, 1]$, the query Q can be evaluated with $\mathcal{O}(N^{\frac{3}{2}})$ time preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ update time under single-tuple updates, and $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$ enumeration delay.*

6.3 Queries in \mathcal{C}_1

We next consider the \mathcal{C}_1 queries. These queries have one input variable, or one output variable and no input variable, including $Q(A \mid \cdot)$, $Q(\cdot \mid A)$, $Q(B \mid A)$ and $Q(B, C \mid A)$, and their symmetries.

6.3.1 Preprocessing

We present the preprocessing stage for the query

$$Q(A \mid \cdot) = R(A, B), S(B, C), T(C, A).$$

The preprocessing stage for the other three queries are same, that is, we build the same view trees; only the enumeration of their results are different.

We partition the relations R , S and T on variables A , B and C , with the threshold N^ϵ , and then decompose $Q(A \mid \cdot)$ into skew-aware queries defined over the relation parts:

$$Q^{s_A s_B s_C}(A \mid \cdot) = R^{A \rightarrow s_A, B \rightarrow s_B}(A, B), S^{B \rightarrow s_B, C \rightarrow s_C}(B, C), T^{C \rightarrow s_C, A \rightarrow s_A}(C, A),$$

where $s_A, s_B, s_C \in \{H, L\}$. The result of the query $Q(A \mid \cdot)$ is the union of the results of these skew-aware queries.

Figure 6.1 shows the view trees for \mathcal{C}_1 queries. These view trees are adapted from those for the \mathcal{C}_2 queries according to the access patterns of the \mathcal{C}_1 queries. The adaptations are highlighted in blue. In each of the view trees for $Q^{HHH}(A \mid \cdot)$, $Q^{LLL}(A \mid \cdot)$, $Q^{HHL}(A \mid \cdot)$, $Q^{LHL}(A \mid \cdot)$, and $Q^{HLL}(A \mid \cdot)$, we create on top of the view tree a new root view,

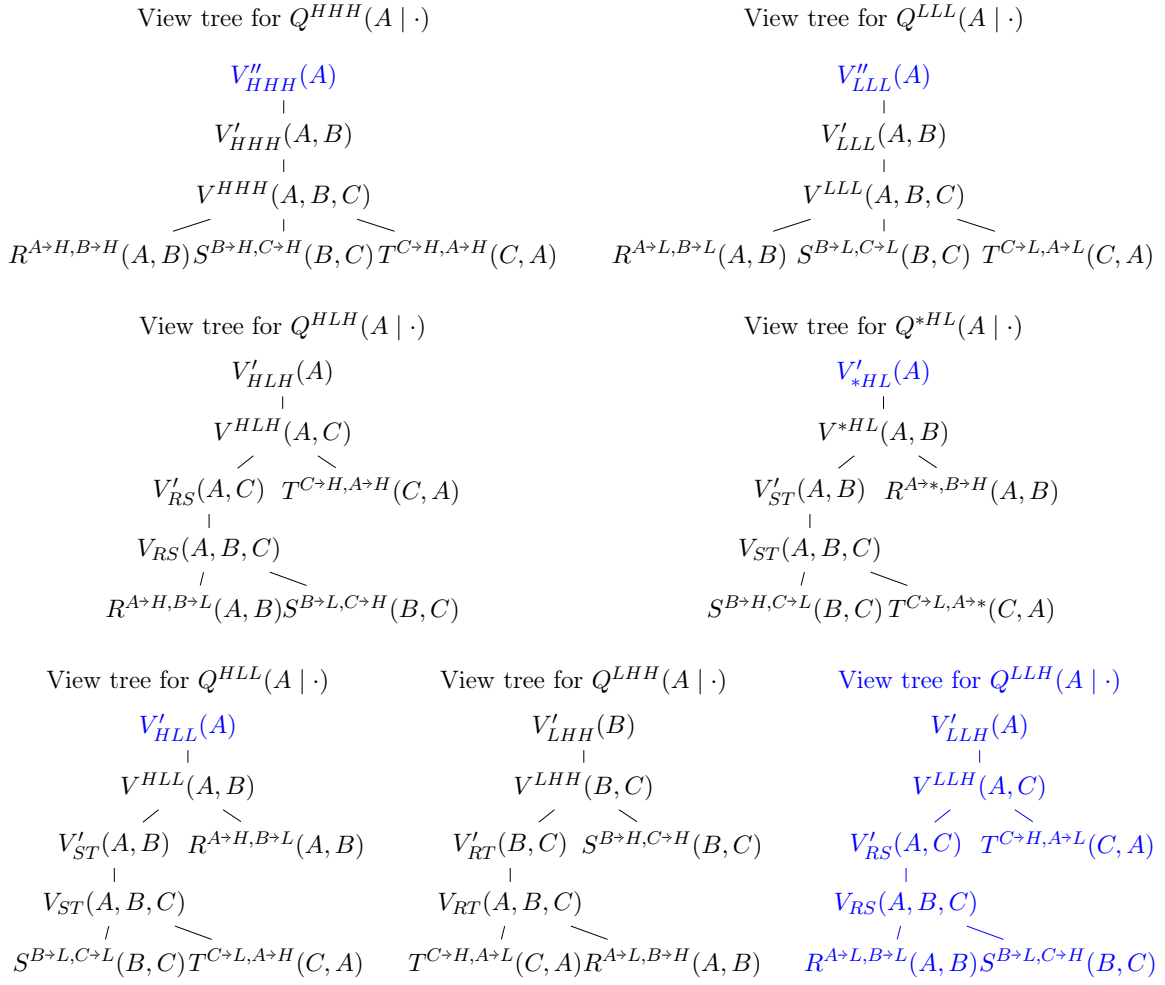


Figure 6.10: View trees supporting the maintenance and enumeration of the results of the skew-aware queries for the unary triangle CQAP queries. These view trees are adapted from the those for the \mathbf{C}_2 queries. The differences are highlighted in blue. The wildcard $*$ can be either H or L .

which aggregates away the variable B in the original root view. This allows the constant-delay enumeration of the distinct A -values in the view tree. For the skew-aware query $Q^{LLH}(A | \cdot)$, we create a new view tree, which is similar as the view tree for $Q^{HLH}(A | \cdot)$.

The additional views in the view trees for \mathbf{C}_1 queries are computed by aggregating away one variable from their child views. The time to compute these additional views is upper-bounded $\mathcal{O}(N^{\frac{3}{2}})$, which is the same as the time to compute the views for the \mathbf{C}_2 CQAP queries. Hence, the overall time complexity of the preprocessing stage is $\mathcal{O}(N^{\frac{3}{2}})$.

6.3.2 Enumeration

We discuss the enumeration for \mathbf{C}_1 queries using the view trees created in Figure 6.10. We show the enumeration for $Q(A \mid \cdot)$, $Q(\cdot \mid A)$, $Q(B \mid A)$ and $Q(B, C \mid A)$. The enumeration for other symmetric \mathbf{C}_1 queries is analogous.

Query $Q(A \mid \cdot)$. We first consider the \mathbf{C}_1 query $Q(A \mid \cdot)$. For all skew-aware queries except $Q^{LHH}(A \mid \cdot)$, the results of the queries are materialized in the root views of the corresponding view trees. We create view iterators at these root views to enumerate the corresponding query results with constant delay.

For the skew-aware queries $Q^{LHH}(A \mid \cdot)$, we create a hop view iterator $\text{hit}_{V_{RT}}(A \mid B, C)$ and open it with the context relation $V^{LHH}(B, C)$. The enumeration delay is determined by the number of (B, C) -tuples in V^{LHH} for any A -value. Since the relation parts in the view tree are light on A , an A -value pairs with $\mathcal{O}(N^\epsilon)$ B -values in $R^{A \rightarrow L, B \rightarrow *}(A, B)$ and $\mathcal{O}(N^\epsilon)$ C -values in $T^{C \rightarrow H, A \rightarrow L}(C, A)$, and thus overall $\mathcal{O}(N^{2\epsilon})$ (B, C) -tuples. Meanwhile, since relation parts are heavy on B and C , there are $\mathcal{O}(N^{1-\epsilon})$ B -values and $\mathcal{O}(N^{1-\epsilon})$ C -values, and thus $\mathcal{O}(N^{2-2\epsilon})$ (B, C) -tuples. Overall, the enumeration delay of the iterator is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

To enumerate the distinct tuples of $Q(A \mid \cdot)$ from the all skew-aware queries, we apply the UNION algorithm. The enumeration delay is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

Query $Q(\cdot \mid A)$. The enumeration for $Q(\cdot \mid A)$ is similar to that for $Q(A \mid \cdot)$. We create the same iterators except we now put A in the context schema of the iterators. The enumeration delay stays same: $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

Query $Q(B \mid A)$. We next consider the query $Q(B \mid A)$. For each of the skew-aware queries $Q^{HHH}(B \mid A)$, $Q^{LLL}(B \mid A)$, $Q^{*HL}(B \mid A)$, and $Q^{HLL}(B \mid A)$, the corresponding view tree contains a view that materializes the (A, B) -tuples in the join result of the relation parts, such as the view $V'_{HHH}(A, B)$ in the view tree for $Q^{HHH}(B \mid A)$. We create view iterators with the output schema B and context schema A at these views, such as $\text{it}_{V'_{HHH}}(B \mid A)$, to enumerate the results of the corresponding queries with constant delay.

For the skew-aware query $Q^{HLH}(B \mid A)$, we create an iterator $\text{it}_{V'_{HLH}}(A \mid A)$ for checking whether the view V'_{HLH} contains the given A -value a . We then create a hop view iterator $\text{hit}_{V_{RS}}(B \mid A, C)$ and open it with the context relation $ctx_{RS} = V^{HLH}(A, C) \bowtie \{a\}$. The enumeration delay of $\text{hit}_{V_{RS}}(B \mid A, C)$ is determined by the number of (A, C) -tuples in ctx_{RS} that are paired with any B -value. Since the relation parts are light on B and heavy on C , the number of such (A, C) -tuples is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$, and thus the enumeration delay is $\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$. The enumeration for $Q^{LLH}(B \mid A)$ is analogous.

For the skew-aware query $Q^{LHH}(B \mid A)$, we create a hop view iterator $\text{hit}_{V_{RS}}(B \mid A, C)$ and open it with the context relation $V^{LHH}(A, C)$. Since the relation parts are light on B and heavy on A and C , the enumeration delay is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

Overall, the enumeration delay for $Q(\cdot|A)$ is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

Query $Q(B, C|A)$. For the two skew-aware queries $Q^{HHH}(B, C|A)$ and $Q^{LLL}(B, C|A)$, we create the iterators $\text{it}_{V^{HHH}}(B, C|A)$ and $\text{it}_{V^{LLL}}(B, C|A)$ to enumerate their query results with constant delay.

For the skew-aware query $Q^{HLL}(B, C|A)$, we create the view iterators $\text{it}_{V^{HLL}}(A|A)$, $\text{it}_{V^{HLL}}(B|A)$ and $\text{it}_{V^{ST}}(C|A, B)$. The enumeration delay for each iterator is constant, thus the overall enumeration delay for $Q^{HLL}(B, C|A)$ is constant. The enumeration for $Q^{HLH}(B, C|A)$, $Q^{HHL}(B, C|A)$, $Q^{LHL}(B, C|A)$ and $Q^{LLH}(B, C|A)$ is analogous.

For the skew-aware query $Q^{LHH}(B, C|A)$, we can materialize the results by intersecting the (B, C) -tuples paired with the input A -value a in $V_{RT}(A, B, C)$, i.e., $\pi_{B,C}\sigma_{A=a}V_{RT}$, and the (B, C) -tuples in $V^{LHH}(B, C)$: for each (B, C) -tuple in $\pi_{B,C}\sigma_{A=a}V_{RT}$, we look it up in V^{LHH} . The computation time is determined by the number of (B, C) -tuples in $\pi_{B,C}\sigma_{A=a}V_{RT}$. Since the relation parts are heavy on B and C and light on A , the number of such (B, C) -tuples is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$, and thus the computation time is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$. Hence, the enumeration delay for $Q^{LHH}(B, C|A)$ is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

Overall, the enumeration delay over all skew-aware queries is $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$.

6.3.3 Update

The update of the view trees is similar to that of the view trees for the \mathbf{C}_3 query in Section 6.1.3. The extra views in the view trees for the \mathbf{C}_1 queries are all views that aggregate away a variable from the views below, which can be updated with $\mathcal{O}(1)$ time. Hence, the update time for \mathbf{C}_1 queries is $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$.

Following the discussion in the previous sections, we conclude the following proposition for the trade-offs of the \mathbf{C}_1 query:

Proposition 6.3. *Given a \mathbf{C}_1 query Q , a database of size N , and $\epsilon \in [0, 1]$, the query Q can be evaluated with $\mathcal{O}(N^{\frac{3}{2}})$ time preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ update time under single-tuple updates, and $\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$ enumeration delay.*

6.4 The \mathbf{C}_{count} Query

We next focus on the \mathbf{C}_{count} query:

$$Q(\cdot | \cdot) = R(A, B), S(B, C), T(C, A).$$

It computes the number of triangles in the join of the relations R , S and T in the database. We apply a strategy as for the \mathbf{C}_3 query. We partition each of R , S and T on the variables

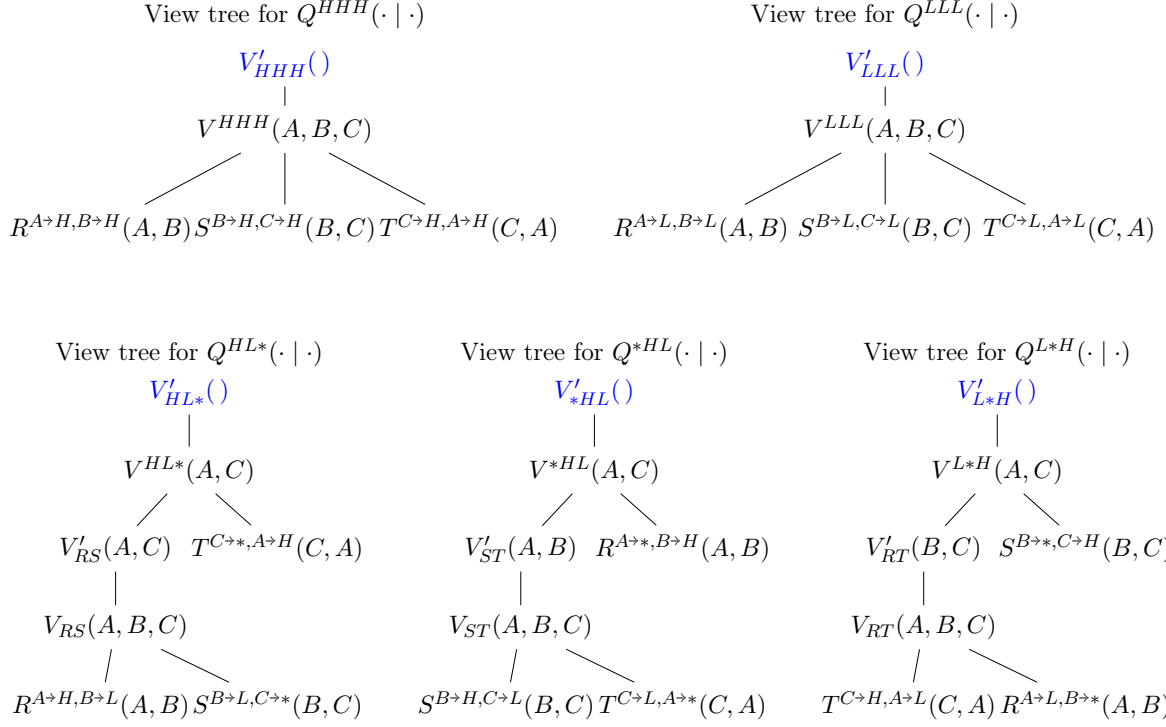


Figure 6.11: The view trees for maintaining the results of the skew-aware queries for the \mathbb{C}_{count} query $Q(\cdot | \cdot)$. The wildcard $*$ can be either H or L in a view tree.

A , B and C into the relation parts and decompose $Q(\cdot | \cdot)$ into the following skew-aware queries:

$$Q^{s_A s_B s_C}(\cdot | \cdot) = R^{A \rightarrow s_A, B \rightarrow s_B}(A, B), S^{B \rightarrow s_B, C \rightarrow s_C}(B, C), T^{C \rightarrow s_C, A \rightarrow s_A}(C, A),$$

where $s_A, s_B, s_C \in \{H, L\}$. Each skew-aware query counts the number of triangles in the corresponding relation parts. The result of the \mathbb{C}_{count} query Q is the sum of these skew-aware queries:

$$Q(\cdot | \cdot) = \sum_{s_A s_B s_C \in \{H, L\}} Q^{s_A s_B s_C}(\cdot | \cdot).$$

Figure 6.11 shows the view trees for the \mathbb{C}_{count} query. They are same as those for the \mathbb{C}_3 query, except we create for each view tree a new root view that marginalizes the variables in the old root view, which are highlighted in blue. Such new root views store the results of the skew-aware queries. The result of the \mathbb{C}_{count} query is the sum of these root views.

Proposition 6.4. *Given the \mathbb{C}_{count} query, a database of size N , and $\epsilon \in [0, 1]$, the view trees constructed in the preprocessing stage can be computed in $\mathcal{O}(N^{\frac{3}{2}})$ time.*

Proof. Computing the blue views in Figure 6.11 requires aggregating away the variables in the views below. This takes time linear to the size of the views below, which is

upper-bounded by $\mathcal{O}(N^{\frac{3}{2}})$. The computation of other views takes time upper-bounded by $\mathcal{O}(N^{\frac{3}{2}})$, as explained in Section 6.1. Overall, the preprocessing time is $\mathcal{O}(N^{\frac{3}{2}})$. \square

Computing the result of the \mathcal{C}_{count} query requires summing up the results in the root views of the view trees. This takes constant time. Hence, the enumeration delay is constant. The update time is the same as that of the \mathcal{C}_3 query, which is $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$. We conclude the following proposition for the trade-offs of the \mathcal{C}_{count} query:

Proposition 6.5. *Given the \mathcal{C}_{count} query Q , a database of size N , and $\epsilon \in [0, 1]$, the query Q can be evaluated with $\mathcal{O}(N^{\frac{3}{2}})$ time preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

6.5 The \mathcal{C}_{lookup} Query

We finally discuss the \mathcal{C}_{lookup} query:

$$Q(\cdot \mid A, B, C) = R(A, B), S(B, C), T(C, A).$$

It serves to check the multiplicity of a given triangle in the database.

The fracture of the query is:

$$Q(\cdot \mid A_1, A_2, B_1, B_2, C_1, C_2) = R(A_1, B_1), S(B_2, C_2), T(C_3, A_3).$$

It has three connected components, each of which is a single relation, hence each sub-query in the fracture is a CQAP_0 query. This makes the \mathcal{C}_{lookup} query a CQAP_0 query. So, we can apply our algorithm for arbitrary CQAP queries to the \mathcal{C}_{lookup} query, and achieve the following result:

Proposition 6.6. *Given the \mathcal{C}_{lookup} query Q , a database of size N , and $\epsilon \in [0, 1]$, the query Q can be evaluated with $\mathcal{O}(N)$ time preprocessing time, $\mathcal{O}(1)$ update time under single-tuple updates, and $\mathcal{O}(1)$ enumeration delay.*

6.6 Complexity Analysis

Following the Propositions in the previous sections, we can prove Theorem 3.10.

In the propositions in the previous sections, we showed the preprocessing time, enumeration delay and update time for each category of triangle CQAP queries. The update time is the time for a single-tuple update. We extend this to the amortized update time for a sequence of updates as we did for the CQAP queries with hierarchical fractures in Section 5.4.3, and prove the following proposition:

Proposition 6.7. *Consider a triangle CQAP query $Q(\mathcal{O}|\mathcal{I})$, a database of size N , and $\epsilon \in [0, 1]$. Given that the view trees constructed for Q in the preprocessing stage can be constructed in $\mathcal{O}(N^p)$ time and maintained in $\mathcal{O}(N^u)$ for a single-tuple update, maintaining the views in the view trees under a sequence of single-tuple updates takes $\mathcal{O}(N^u)$ amortized time per single-tuple update.*

Proof. The proof is similar to the proof of Proposition 5.26, except that we replace the update time for CQAP queries whose fractures are hierarchical with the update time for triangle queries. We only give the high level explanation. The minor rebalancing time and the major rebalancing time for the view trees constructed for the triangle queries are The minor rebalancing time is amortized over $\Omega(N^\epsilon)$ updates, thus the amortized time of minor rebalancing is $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$. The major rebalancing time is amortized over $\Omega(N)$ updates, thus the amortized time of major rebalancing is $\mathcal{O}(N^{\frac{1}{2}})$. Overall, both the minor and major rebalancing times are bounded by $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ ($\mathcal{O}(N^{\frac{1}{2}})$ is upper-bounded by $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$), which is the exactly the update time to process a single-tuple update. Hence, we conclude that the amortized update time for a sequence of updates is $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$. \square

We are now ready to prove Theorem 3.10. The theorem is copied here for convenience.

Theorem 3.10. *Given a triangle CQAP query Q , a database of size N , and $\epsilon \in [0, 1]$. If Q is the \mathcal{C}_{lookup} query, it admits constant preprocessing time, update time, and enumeration time; otherwise, the query Q admits $\mathcal{O}(N^{\frac{3}{2}})$ preprocessing time, $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ amortized update time, and the enumeration delay given in the table below.*

	\mathcal{C}_{count}	\mathcal{C}_1	\mathcal{C}_2	\mathcal{C}_3
Enumeration Delay	$\mathcal{O}(1)$	$\mathcal{O}(N^{\min\{\epsilon, 1-\epsilon\}})$	$\mathcal{O}(N^{\min\{2\epsilon, 2-2\epsilon\}})$	$\mathcal{O}(1)$

Proof. This theorem follows from Propositions 6.1, 6.2, 6.3, 6.5 and 6.6, and complemented by Proposition 6.7, which shows that the amortized rebalancing time is $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$. \square

6.7 Optimality Result

Our approach is optimal for CQAP₀ queries and thus for the \mathcal{C}_{lookup} query. For the other triangle CQAP queries, the following proposition shows that some combinations of update time and enumeration delay for the dynamic evaluation of triangle CQAP queries are not possible, conditioned on the OMv Conjecture 26.

Proposition 6.8. *Consider a triangle CQAP query Q and a database of size N . For any $\gamma > 0$, there is no algorithm for Q admitting arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and*

- $\mathcal{O}(1)$ enumeration delay if Q is a \mathcal{C}_{count} or \mathcal{C}_3 query, or
- $\mathcal{O}(N^{1-\gamma})$ enumeration delay if Q is a \mathcal{C}_1 query, or
- $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay if Q is a \mathcal{C}_2 query,

unless the OMv conjecture fails.

In the proof of this proposition, we reduce the OMv problem to the problem of dynamic evaluation of triangle CQAP queries: We show that if there is an algorithm that incrementally maintains a triangle CQAP query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and the enumeration delay for this query as stated in this proposition, for some $\gamma > 0$ and database with size N , then the OMv problem can be solved in subcubic time, which contradicts the OMv conjecture.

The proof relies on the Online Boolean Vector-Matrix-Vector Multiplication (OuMv) conjecture, which is implied by the OMv conjecture (Conjecture 1.2). First, we give the definition of the OuMv problem and state the corresponding conjecture.

Definition 6.9 (Online Boolean Vector-Matrix-Vector Multiplication (OuMv) [36]). *We are given an $n \times n$ Boolean matrix \mathbf{M} and receive n pairs of Boolean column-vectors of size n , denoted by $(\mathbf{u}_1, \mathbf{v}_1), \dots, (\mathbf{u}_n, \mathbf{v}_n)$; after seeing each pair $(\mathbf{u}_i, \mathbf{v}_i)$, we output the product $\mathbf{u}_i^T \mathbf{M} \mathbf{v}_i$ before we see the next pair.*

Conjecture 10 (OuMv Conjecture, Theorem 2.7 in [36]). *For any $\gamma > 0$, there is no algorithm that solves OuMv in time $\mathcal{O}(n^{3-\gamma})$.*

Proposition 6.8 shows the lower bounds for the triangle CQAP queries in different classes. We prove them separately. We start with the lower bound for the \mathcal{C}_{count} query.

Hardness for the \mathcal{C}_{count} Query

The following proof reduces the OuMv problem to the problem of incrementally maintaining the \mathcal{C}_{count} query. This reduction implies that if there is an algorithm that incrementally maintains a triangle CQAP query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(1)$ enumeration delay for some $\gamma > 0$ and database with size N , then the OuMv problem can be solved in subcubic time. This contradicts the OuMv conjecture and, consequently, the OMv conjecture.

SOLVEOUMV(matrix \mathbf{M} , vectors $\mathbf{u}_1, \mathbf{v}_1, \dots, \mathbf{u}_n, \mathbf{v}_n$)

```

1   $\mathcal{T} :=$  view trees constructed on the initial empty database
2  foreach  $(i, j) \in \mathbf{M}$  do
3       $\delta S := \{ (i, j) \mapsto \mathbf{M}(i, j) \}$ 
4      ONUPDATE( $\mathcal{T}, \delta S$ )
5  foreach  $r := 1, \dots, n$  do
6      foreach  $i := 1, \dots, n$  do
7           $\delta R := \{ (a, i) \mapsto (\mathbf{u}_r(i) - R(a, i)) \}$ 
8          ONUPDATE( $\mathcal{T}, \delta R$ )
9           $\delta T := \{ (i, a) \mapsto (\mathbf{v}_r(i) - T(i, a)) \}$ 
10         ONUPDATE( $\mathcal{T}, \delta T$ )
11  output  $(Q_0(\cdot | \cdot) \neq 0)$ 

```

Figure 6.12: The procedure SOLVEOUMV solves the OuMv problem using an incremental algorithm that maintains the \mathcal{C}_{count} triangle query Q_0 under single-tuple updates. The \mathcal{T} are the view trees constructed on the initial database with empty relations R , S and T . The procedure ONUPDATE is given in Figure 5.25 and maintains the result of the query under single-tuple updates.

Proof. The proof is inspired by the lower bound proof for maintaining non-hierarchical Boolean conjunctive queries [14]. Let Q_0 be the \mathcal{C}_{count} query. For the sake of contradiction, assume that there is an incremental maintenance algorithm \mathcal{A} that maintains Q_0 under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(1)$ enumeration delay, for some $\gamma > 0$. We show that this algorithm can be used to design an algorithm \mathcal{B} that solves the OuMv problem in subcubic time, which contradicts the OuMv conjecture.

The reduction. Figure 6.12 gives the pseudocode of the algorithm \mathcal{B} processing an OuMv input $(\mathbf{M}, (\mathbf{u}_1, \mathbf{v}_1), \dots, (\mathbf{u}_n, \mathbf{v}_n))$. We denote the entry of \mathbf{M} in row i and column j by $\mathbf{M}(i, j)$ and the i -th component of \mathbf{v} by $\mathbf{v}(i)$. The algorithm first constructs the view trees on the initial database $\mathbf{D} = \{R, S, T\}$ with empty relations R , S , and T . Then, it executes at most n^2 updates to the relation S such that $S = \{ (i, j) \mapsto \mathbf{M}(i, j) \mid i, j \in [n] \}$. In each round $r \in [n]$, the algorithm executes at most $2n$ updates to the relations R and T such that $R = \{ (a, i) \mapsto \mathbf{u}_r(i) \mid i \in [n] \}$ and $T = \{ (i, a) \mapsto \mathbf{v}_r(i) \mid i \in [n] \}$, where a is some dummy constant. By construction, $\mathbf{u}_r^T \mathbf{M} \mathbf{v}_r = 1$ if and only if there exist $i, j \in [n]$ such that $\mathbf{u}_r(i) = 1$, $\mathbf{M}(i, j) = 1$, and $\mathbf{v}_r(j) = 1$, which is equivalent to $R(a, i) \wedge S(i, j) \wedge T(j, a) = 1$ at the end of round r . Thus, the algorithm outputs 1 at the end of round r if and only if there is at least one triangle in the database, i.e., $Q_0(\cdot | \cdot) \neq 0$.

Time analysis. Constructing the view trees on the initial database with empty relations takes constant time. The construction of relation S from \mathbf{M} requires at most n^2 updates. Given that the amortized time for each update is $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ and the database size N stays $\mathcal{O}(n^2)$, the overall time for constructing relation S is $\mathcal{O}(n^2 \cdot n^{2 \cdot (\frac{1}{2}-\gamma)}) = \mathcal{O}(n^{3-2\gamma})$. In each round, the algorithm performs at most $2n$ updates and needs $\mathcal{O}(1)$ time to report the output tuple. Hence, the time to execute the updates in a single round is $\mathcal{O}(2n \cdot n^{2 \cdot (\frac{1}{2}-\gamma)}) = \mathcal{O}(n^{2-2\gamma})$. The time to report the output tuple is $\mathcal{O}(1)$. Thus, the overall execution time is $\mathcal{O}(n^{2-2\gamma})$ per round and $\mathcal{O}(n^{3-2\gamma})$ for n rounds. Hence, algorithm \mathcal{B} needs $\mathcal{O}(n^{3-2\gamma})$ time to solve the OuMv problem, which contradicts the OuMv conjecture and, consequently, the OMv conjecture. □

Hardness for the \mathcal{C}_3 Query

We next focus on the lower bound for the \mathcal{C}_3 query. We apply a similar strategy as for the \mathcal{C}_{count} query.

Proof. For the sake of contradiction, assume that there is an incremental maintenance algorithm \mathcal{A} that maintains a \mathcal{C}_3 query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(1)$ enumeration delay, for some $\gamma > 0$. We show that this algorithm can be used to design an algorithm \mathcal{B} that solves the OuMv problem in subcubic time, which contradicts the OuMv conjecture.

We design the algorithm \mathcal{B} as in the \mathcal{C}_{count} case, except now at the end of each round, instead of checking whether $Q_0 \neq 0$, we check whether the result of the \mathcal{C}_3 query is nonempty. The time needed to signalize nonemptiness is $\mathcal{O}(1)$ per round, and thus $\mathcal{O}(n)$ for n rounds. Since the time to execute the updates in a single round is $\mathcal{O}(n^{2-2\gamma})$, the overall execution time is $\mathcal{O}(n^{2-2\gamma})$ per round and $\mathcal{O}(n^{3-2\gamma})$ for n rounds. Hence, algorithm \mathcal{B} needs $\mathcal{O}(n^{3-2\gamma})$ time to solve the OuMv problem, which contradicts the OuMv conjecture and, consequently, the OMv conjecture. □

Hardness for the \mathcal{C}_1 Queries

We next prove the lower bound for the \mathcal{C}_1 queries.

Proof. For the sake of contradiction, assume that there is an incremental maintenance algorithm \mathcal{A} that maintains a \mathcal{C}_1 query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, for some $\gamma > 0$. We show that this algorithm can be used to design an algorithm \mathcal{B} that solves the OuMv problem in subcubic time, which contradicts the OuMv conjecture.

SOLVEOMV2(matrix \mathbf{M} , vectors $\mathbf{u}_1, \mathbf{v}_1, \dots, \mathbf{u}_n, \mathbf{v}_n$)

```

1   $\mathcal{T} :=$  view trees constructed on the initial empty database
2  foreach  $(i, j) \in \mathbf{M}$  do
3       $\delta S := \{ (i, j) \mapsto \mathbf{M}(i, j) \}$ 
4      ONUPDATE( $\mathcal{T}, \delta S$ )
5  foreach  $r := 1, \dots, n$  do
6      foreach  $i := 1, \dots, n$  do
7           $\delta R := \{ (a, i) \mapsto (\mathbf{u}_r(i) - R(a, i)) \}$ 
8          ONUPDATE( $\mathcal{T}, \delta R$ )
9           $\delta T := \{ (i, a) \mapsto (\mathbf{v}_r(i) - T(i, a)) \}$ 
10         ONUPDATE( $\mathcal{T}, \delta T$ )
11         foreach  $i := 1, \dots, n$  do
12             if  $Q_2(\cdot \mid a, i) \neq 0$ 
13                 output true
14         output false

```

Figure 6.13: The procedure SOLVEOMV2 solves the OMV problem using an incremental algorithm that maintains the \mathcal{C}_2 query $Q_2(\cdot \mid A, B)$ under single-tuple updates.

We design the algorithm \mathcal{B} as for the \mathcal{C}_{count} query, except now at the end of each round: in the case of the \mathcal{C}_1 query is $Q(A \mid \cdot)$, we check whether the query result is empty by triggering enumeration and checking whether at least one output tuple is reported; in case of other \mathcal{C}_1 queries, i.e., $Q(\cdot \mid A)$, $Q(B \mid A)$ or $Q(B, C \mid A)$, we check whether $Q(\cdot \mid a) \neq 0$, $Q(B \mid a)$ or $Q(B, C \mid a)$ is nonempty, for the dummy constant value a .

The time analysis is similar as that for the \mathcal{C}_{count} query, except that the algorithm now needs $\mathcal{O}(N^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{2(\frac{1}{2}-\gamma)}) = \mathcal{O}(n^{1-2\gamma})$ in each round to report the first output tuple or to signalize that the result is empty. Since the time to execute the updates in a single round is $\mathcal{O}(n^{2-2\gamma})$, the overall execution time is $\mathcal{O}(n^{2-2\gamma})$ per round and $\mathcal{O}(n^{3-2\gamma})$ for n rounds. Hence, algorithm \mathcal{B} needs $\mathcal{O}(n^{3-2\gamma})$ time to solve the OMV problem, which contradicts the OMV conjecture and, consequently, the OMV conjecture. □

Hardness for the \mathcal{C}_2 Queries

Finally, we discuss the lower bound for the \mathcal{C}_2 queries.

Proof. For the sake of contradiction, assume that there is an incremental maintenance algorithm \mathcal{A} that maintains a \mathcal{C}_3 query under single-tuple updates with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, for

some $\gamma > 0$. We show that this algorithm can be used to design an algorithm \mathcal{B} that solves the OuMv problem in subcubic time, which contradicts the OuMv conjecture.

In case of the \mathcal{C}_2 query $Q(A, B \mid \cdot)$, we design the algorithm \mathcal{B} as in the $\mathcal{C}_{\text{count}}$ case. The time needed to signalize nonemptiness is $\mathcal{O}(N^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{1-2\gamma})$ per round, and thus $\mathcal{O}(n^{2-2\gamma})$ for n rounds. This leads to $\mathcal{O}(n^{3-2\gamma})$ overall execution time for n rounds, which contradicts the OuMv conjecture and, consequently, the OMv conjecture.

In case of the other \mathcal{C}_2 queries, i.e., $Q(\cdot \mid A, B)$ or $Q(C \mid A, B)$, we design the algorithm \mathcal{B} as shown in Figure 6.13. Its differences from the algorithm \mathcal{B} in the $\mathcal{C}_{\text{count}}$ case is that at the end of each round, for each tuple (a, i) inserted to R , we check whether it creates triangles in the database via $Q_2(\cdot \mid a, i) \neq 0$ (Lines 11-14). In each round, we need to check the query result for n tuples, which takes $\mathcal{O}(n \cdot n^{1-2\gamma}) = \mathcal{O}(n^{2-2\gamma})$ time. For n rounds, we need $\mathcal{O}(n^3 - 2\gamma)$ times. This leads to $\mathcal{O}(n^{3-2\gamma})$ overall execution time for n rounds, which contradicts the OuMv conjecture and, consequently, the OMv conjecture. \square

Optimality of Our Approach. For $\epsilon = \frac{1}{2}$, our approach needs $\mathcal{O}(N^{\frac{1}{2}})$ amortized update time and, depending on the query, an enumeration delay such that the trade-off between these two measures is Pareto optimal. For the $\mathcal{C}_{\text{count}}$ and \mathcal{C}_3 queries, the delay is $\mathcal{O}(1)$. Our approach is strongly Pareto worst-case optimal for these queries: There can be no tighter upper bound for any of the update time or delay measures without loosening the upper bound for the other measure. For the \mathcal{C}_1 and \mathcal{C}_2 queries, the delay is $\mathcal{O}(N)$ and respectively $\mathcal{O}(N^{\frac{1}{2}})$. Our approach is only weakly Pareto worst-case optimal for the \mathcal{C}_1 and \mathcal{C}_2 queries: there are no tighter upper bounds for both the update time and delay measures. Nevertheless, either the update time or the delay may still be lowered for the \mathcal{C}_1 and \mathcal{C}_2 queries without contradicting the OMv conjecture.

Corollary 6.11 summarizes the above discussion on the worst-case optimality of our approach for the triangle CQAP queries.

Corollary 6.11 (Theorem 3.10 and Proposition 6.8). *Under a single-tuple update to the database \mathbf{D} , our approach with $\epsilon = \frac{1}{2}$ is strongly Pareto worst-case optimal for the $\mathcal{C}_{\text{count}}$ and \mathcal{C}_3 queries and weakly Pareto worst-case optimal for the \mathcal{C}_1 and \mathcal{C}_2 queries in the update-delay space, unless the OMv conjecture fails.*

Chapter 7

Dichotomy Result

The third contribution of this thesis is a dichotomy result for CQAPs, as stated in Theorem 3.11. The dichotomy states that the queries in CQAP_0 are precisely those CQAP queries that can be evaluated with constant update time and enumeration delay. We focus on the proof of this theorem in this chapter. The theorem is copied here for convenience.

Theorem 3.11. *Consider an arbitrary CQAP query Q and a database of size N .*

- *If Q is in CQAP_0 , then it admits $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ enumeration delay, and $\mathcal{O}(1)$ update time for single-tuple updates.*
- *If Q is not in CQAP_0 and has no repeating relation symbols, then there is no algorithm that computes Q with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, for any $\gamma > 0$, unless the OMv conjecture fails.*

We start the intuition behind the proof. The class CQAP_0 trivially contains all queries, whose variables are free and whose join variables are input. The smallest queries not included in CQAP_0 are:

- $Q_1(\mathcal{O}|\cdot) = R(A), S(A, B), T(B)$ with $\mathcal{O} \subseteq \{A, B\}$;
- $Q_2(A|\cdot) = R(A, B), S(B)$;
- $Q_3(\cdot|A) = R(A, B), S(B)$; and
- $Q_4(B|A) = R(A, B), S(B)$.

Each query is equal to its fracture. Query Q_1 is not hierarchical; Q_2 is not free-dominant; and Q_3 and Q_4 are not input-dominant. Prior work showed that there is no algorithm that achieves constant update time and enumeration delay for Q_1 and Q_2 , unless the OMv conjecture fails [14]. To prove the hardness statement in Theorem 3.11, we show

that this negative result also holds for Q_3 and Q_4 . Then, given an arbitrary CQAP query Q that is not in CQAP_0 , we reduce the evaluation of one of the four queries above to the evaluation of Q .

Proof. We start with an auxiliary lemma and a proposition.

Lemma 7.1. *If a CQAP query Q can be evaluated with $\mathcal{O}(f_p(N))$ preprocessing time, $\mathcal{O}(f_e(N))$ enumeration delay, and $\mathcal{O}(f_u(N))$ amortised update time, then its fracture Q_{\dagger} can be evaluated with the same asymptotic complexities, where N is the database size.*

Proof. Consider a CQAP query $Q(\mathcal{O}|\mathcal{I})$, its fracture $Q_{\dagger}(\mathcal{O}|\mathcal{I}_{\dagger})$, and a database \mathcal{D} for Q_{\dagger} of size N . We call a fresh variable A in Q_{\dagger} that replaces a variable A' in Q a *representative* of A . Let C_1, \dots, C_n be the sets of database relations that correspond to the connected components of Q_{\dagger} . We construct from \mathcal{D} the databases $\mathcal{D}_1, \dots, \mathcal{D}_n$, where each \mathcal{D}_i is constructed as follows. The database \mathcal{D}_i contains each relation in \mathcal{D} such that: (1) If $R \in C_i$ and R has a variable A in its schema that is a representative of a variable A' , the variable A is replaced by A' ; (2) the values in all relations not contained in C_i are replaced by a single dummy value d_i . The overall size of the databases is $\mathcal{O}(N)$. Given an input tuple \mathbf{t} over \mathcal{I} , we denote by $(Q(\mathcal{O}|\mathbf{t}), \mathcal{D}_i)$ the result of Q for input \mathbf{t} evaluated on \mathcal{D}_i . The result consists of the tuples over the output variables in C_i for the given input tuple \mathbf{t} , paired with the dummy value d_i over the output variables not in C_i . Intuitively, the result of Q_{\dagger} on \mathcal{D} can be obtained from the Cartesian product of the results of Q on $\mathcal{D}_1, \dots, \mathcal{D}_n$. To be more precise, consider a tuple \mathbf{t}_{\dagger} over \mathcal{I}_{\dagger} . We define for each $i \in [n]$, a tuple \mathbf{t}_i over \mathcal{I} such that $\mathbf{t}_i[A] = \mathbf{t}_{\dagger}[A']$ if A' is a representative of A . The result of $Q_{\dagger}(\mathcal{O}|\mathbf{t}_{\dagger})$ on \mathcal{D} is equal to the Cartesian product $\times_{i \in [n]} \pi_{\mathcal{O}_i}(Q(\mathcal{O}|\mathbf{t}_i), \mathcal{D}_i)$, where \mathcal{O}_i is the set of output variables of Q contained in C_i . Now, assume that we want to enumerate the result of $(Q_{\dagger}(\mathcal{O}|\mathbf{t}_{\dagger}), \mathcal{D})$. We start the enumeration procedure for each $Q(\mathcal{O}|\mathbf{t}_i), \mathcal{D}_i$ with $i \in [n]$. For each $\mathbf{t}'_1 \in Q(\mathcal{O}|\mathbf{t}_1), \mathcal{D}_1), \dots, \mathbf{t}'_n \in Q(\mathcal{O}|\mathbf{t}_n), \mathcal{D}_n)$, we return the tuple $\pi_{\mathcal{O}_1} \mathbf{t}'_1 \circ \dots \circ \pi_{\mathcal{O}_n} \mathbf{t}'_n$. This implies that the result of $(Q_{\dagger}(\mathcal{O}|\mathbf{t}_{\dagger}), \mathcal{D})$ can be enumerated with $\mathcal{O}(f_e(N))$ delay if Q admits $\mathcal{O}(f_e(N))$ enumeration delay.

We execute the preprocessing procedure for Q on each of the databases $\mathcal{D}_1, \dots, \mathcal{D}_n$ which takes $\mathcal{O}(f_p(N))$ overall time. Consider an update $\{\mathbf{t} \mapsto m\}$ to a relation R that is contained in the connected component C_i for some $i \in [n]$. We apply the update $\{\mathbf{t}_{\mathcal{I}} \mapsto m\}$ to relation R in \mathcal{D}_i , where $\mathbf{t}_{\mathcal{I}}$ is the tuple over \mathcal{I} defined as:

$$\mathbf{t}_{\mathcal{I}}[A] = \begin{cases} \mathbf{t}[A'] & \text{if } A' \text{ is a representative of } A \\ \mathbf{t}[A] & \text{otherwise} \end{cases}$$

The update takes $\mathcal{O}(f_u(N))$ amortised update time.

Overall, we obtain an evaluation procedure for Q_{\dagger} with $\mathcal{O}(f_p(N))$ preprocessing time, $\mathcal{O}(f_e(N))$ enumeration delay, and $\mathcal{O}(f_u(N))$ amortised update time. \square

Proposition 7.2. *Every CQAP₀ query has dynamic width 0 and static width 1.*

Proof. Consider a CQAP₀ query Q and its fracture Q_{\dagger} . We first show that the dynamic width of Q is 0. By definition, Q_{\dagger} is hierarchical, free-dominant, and input-dominant. Hierarchical queries admit canonical VOs. In canonical VOs, it holds: if a variable A dominates a variable B , then, A is on top of B . Hence, Q_{\dagger} admits a canonical VO that is access-top. Consider a variable X in ω and an atom $R(\mathcal{Y})$ in the subtree ω_X rooted at X . By the definition of canonical VOs, it holds: the dependency set of X consists of the ancestor variables of X ; \mathcal{Y} contains X and all ancestor variables of X . Hence, we have $\rho_{Q_X}^*((\{X\} \cup \text{dep}_{\omega}(X)) \setminus \mathcal{Y}) = \rho_{Q_X}^*((\{X\} \cup \text{anc}_{\omega}(X)) \setminus \mathcal{Y}) = \rho_{Q_X}^*(\emptyset) = 0$. This implies that the dynamic width of ω is 0. This means that the dynamic width of Q_{\dagger} , hence, the dynamic width of Q is 0.

It follows from Proposition 5.27 that the static width of Q is 1¹. □

We are ready to prove Theorem 3.11.

Complexity Upper Bound

We prove the first statement in Theorem 3.11. Assume that Q is in CQAP₀. By Proposition 7.2, Q has dynamic width 0. By definition of CQAP₀, the fracture Q_{\dagger} of Q is hierarchical. It follows from Proposition 5.27 that the static width of Q_{\dagger} , hence the static width of Q , is at most 1. Using Theorem 3.4, we conclude that Q can be evaluated with $\mathcal{O}(N)$ preprocessing time, $\mathcal{O}(1)$ update time, and $\mathcal{O}(1)$ enumeration delay.

Complexity Lower Bound

We prove the second statement in Theorem 3.11. The proof is based on a reduction of the Online Matrix-Vector Multiplication (OMv) problem (Definition 2.25) to the evaluation of non-CQAP₀ queries.

We start with the high-level idea of the proof. Consider the following simple CQAP queries, which are not in CQAP₀.

$$\begin{aligned} Q_1(\mathcal{O}|\cdot) &= R(A), S(A, B), T(B) \quad \mathcal{O} \subseteq \{A, B\} \\ Q_2(A|\cdot) &= R(A, B), S(B) \\ Q_3(\cdot|A) &= R(A, B), S(B) \\ Q_4(B|A) &= R(A, B), S(B) \end{aligned}$$

¹To simplify the presentation, we assume that Q contains at least one variable, so it has static width at least 1. Otherwise, it can trivially be evaluated with constant preprocessing time, update time, and enumeration delay.

Each query is equal to its fracture. Query Q_1 is not hierarchical; Q_2 is not free-dominant; Q_3 and Q_4 are not input-dominant. It is known that queries that are not hierarchical or free-dominant do not admit constant update time and enumeration delay, unless the OMv conjecture fails [14]. We show that the OMv problem can also be reduced to the evaluation of each of the queries Q_3 and Q_4 . Our reduction implies that any algorithm that evaluates the queries Q_3 or Q_4 with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay for any $\gamma > 0$ can be used to solve the OMv problem in subcubic time, which rejects the OMv conjecture. We then show that the evaluation of one of the queries Q_1 to Q_4 can be reduced to the evaluation of any CQAP query that is not in CQAP₀ and does not have repeating relation symbols.

In each of the following two reductions, our starting assumption is that there is an algorithm \mathcal{A} that evaluates the given query with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortized update time, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay for some $\gamma > 0$. We then show that \mathcal{A} can be used to design an algorithm \mathcal{B} that solves the OMv problem in subcubic time.

Hardness for Q_3 Given $n \geq 1$, let $\mathbf{M}, \mathbf{v}_1, \dots, \mathbf{v}_n$ be an input to the OMv problem, where \mathbf{M} is an $n \times n$ Boolean Matrix and $\mathbf{v}_1, \dots, \mathbf{v}_n$ are Boolean column vectors of size n . Algorithm \mathcal{B} uses relation R to encode matrix \mathbf{M} and relation S to encode the incoming vectors $\mathbf{v}_1, \dots, \mathbf{v}_n$. The database domain is $[n]$. First, algorithm \mathcal{B} executes the preprocessing stage on the empty database. Since the database is empty, the preprocessing stage ends after constant time. Then, it executes at most n^2 updates to relation R such that $R(i, j) = 1$ if and only if $\mathbf{M}(i, j) = 1$. Afterwards, it performs a round of operations for each incoming vector \mathbf{v}_r with $r \in [n]$. In the first part of each round, it executes at most n updates to relation S such that $S(j) = 1$ if and only if $\mathbf{v}_r(j) = 1$. Observe that $Q_3(\cdot|i)$ is true for some $i \in [n]$ if and only if $(\mathbf{M}\mathbf{v}_r)(i) = 1$. Algorithm \mathcal{B} constructs the result vector $\mathbf{u}_r = \mathbf{M}\mathbf{v}_r$ as follows. It asks for each $i \in [n]$, whether $Q_3(\cdot|i)$ is true, i.e., i is in the result of Q_3 . If yes, the i -th entry of the result of \mathbf{u}_r is set to 1, otherwise, it is set to 0.

Time Analysis. The size of the database remains $\mathcal{O}(n^2)$ during the whole procedure. Algorithm \mathcal{B} needs at most n^2 updates to encode \mathbf{M} by relation R . Hence, the time to execute these updates is $\mathcal{O}(n^2(n^2)^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{3-2\gamma})$. In each round r with $r \in [n]$, algorithm \mathcal{B} executes n updates to encode vector \mathbf{v}_r into relation S and asks for the result of $Q_3(\cdot|i)$ for every $i \in [n]$. The n updates and requests need $\mathcal{O}(n(n^2)^{\frac{1}{2}-\gamma}) = \mathcal{O}(n^{2-2\gamma})$ time. Hence, the overall time for a single round is $\mathcal{O}(n^{2-2\gamma})$. Consequently, the time for n rounds is $\mathcal{O}(nn^{2-2\gamma}) = \mathcal{O}(n^{3-2\gamma})$. This means that the overall time of the reduction is $\mathcal{O}(n^{3-2\gamma})$ in worst-case, which is subcubic.

Hardness for Q_4 The reduction differs slightly from the case for Q_3 in the way algorithm \mathcal{B} constructs the result vector $\mathbf{u}_r = \mathbf{M}\mathbf{v}_r$ in each round r . For each $i \in [n]$, it starts the enumeration process for $Q_4(B|i)$. If one tuple is returned, it stops the enumeration process and sets the i -th entry of \mathbf{u}_r to be 1. If no tuple is returned, the i -th entry is set to 0. Thus, the time to decide the i -th entry of the result of \mathbf{u}_r is the same as in case of Q_3 . Hence, the overall time of the reduction stays subcubic.

Hardness in the General Case Consider now an arbitrary CQAP query Q that is not in CQAP_0 and does not have repeating relation symbols. Since Q is not in CQAP_0 , this means that its fracture Q_{\dagger} is either not hierarchical, not free-dominant, or not input-dominant. If Q_{\dagger} is not hierarchical or it is not free-dominant and all free variables are output, it follows from prior work that there is no algorithm that evaluates Q_{\dagger} with $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time for any $\gamma > 0$, unless the OMv conjecture fails [14]. By Lemma 7.1, no such algorithm can exist for Q . Hence, we assume that Q_{\dagger} is hierarchical and consider two cases:

- (1) Q_{\dagger} is not free-dominant and all free variables are input
- (2) Q_{\dagger} is free-dominant but not input-dominant

Case (1). The query contains an input variable A and a bound variable B such that $\text{atoms}(A) \subset \text{atoms}(B)$. This means that there are two atoms $R(\mathcal{X})$ and $S(\mathcal{Y})$ with $\mathcal{Y} \cap \{A, B\} = \{B\}$ and $A, B \in \mathcal{X}$. Assume that there is an algorithm \mathcal{A} that evaluates Q_{\dagger} with arbitrary preprocessing time, $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ enumeration delay, and $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ amortised update time for some $\gamma > 0$. We will design an algorithm \mathcal{B} that evaluates Q_3 with the same complexities. This rejects the OMv conjecture. Hence, by Lemma 7.1, Q cannot be evaluated with these complexities, unless the OMv conjecture fails.

We define $\mathcal{R}_{(A,B)}$ to be the set of atoms that contain both A and B in their schemas and $\mathcal{S}_{(-A,B)}$ to be the set of atoms that contain B but not A . Note that there cannot be any atom containing A but not B , since this would imply that the query is not hierarchical, contradicting our assumption. We use each atom $R'(\mathcal{X}') \in \mathcal{R}_{(A,B)}$ to encode atom $R(A, B)$ and each atom $S'(\mathcal{Y}') \in \mathcal{S}_{(-A,B)}$ to encode atom $S(B)$ in Q_3 . Consider a database \mathcal{D} of size N for Q_3 and a dummy value d that is not included in the domain of \mathcal{D} . We write $(\mathcal{S}, A = a, B = b, d)$ to denote a tuple over schema \mathcal{S} that assigns the values a and b to the variables A and respectively B and all other variables in \mathcal{S} to d . Likewise, $(\mathcal{S}, B = b, d)$ denotes a tuple that assigns value b to B and all other variables in \mathcal{S} to d . Algorithm \mathcal{B} first constructs from \mathcal{D} a database \mathcal{D}' for Q_{\dagger} as follows. For each tuple (a, b) in relation R and each atom $R'(\mathcal{X}')$ in $\mathcal{R}_{A,B}$, it assigns the tuple $(\mathcal{X}', A = a, B = b, d)$ to relation R' . Likewise, for each value b in relation S and each atom

$S'(\mathcal{Y}')$ in $\mathcal{S}_{(-A,B)}$, it assigns the tuple $(\mathcal{Y}', B = b, d)$ to relation S' . The size of \mathcal{D}' is linear in N . Then, algorithm \mathcal{B} executes the preprocessing for Q_{\dagger} on \mathcal{D}' . Each single-tuple update $\{(a, b) \mapsto m\}$ to relation R is translated to a sequence of single-tuple updates $\{(\mathcal{X}', A = a, B = b, d) \mapsto m\}$ to all relations referred to by atoms in $\mathcal{R}_{(A,B)}$. Analogously, updates $\{b \mapsto m\}$ to S are translated to updates $\{(S', B = b, d) \mapsto m\}$ to all relations S' with $S'(\mathcal{Y}') \in \mathcal{S}_{(-A,B)}$. Hence, the amortised update time is $\mathcal{O}(N^{0.5-\gamma})$. Each input tuple (a) for Q_3 is translated into an input tuple $(\mathcal{I}_{\dagger}, A = a, d)$ for Q_{\dagger} where \mathcal{I}_{\dagger} is the set of input variables for Q_{\dagger} . Recall that all free variables of Q_{\dagger} are input. The answer of $Q_3(\cdot|a)$ is true if and only if the answer of $Q_{\dagger}(\cdot|(\mathcal{I}_{\dagger}, A = a, d))$ is true. The answer time is $\mathcal{O}(N^{0.5-\gamma})$. We conclude that Q_3 can be evaluated with $\mathcal{O}(N^{0.5-\gamma})$ enumeration delay and $\mathcal{O}(N^{0.5-\gamma})$ amortised update time, a contradiction due to the OMv conjecture.

Case (2). We now consider the case that the query Q_{\dagger} is free-dominant but not input-dominant. In this case, the we reduce the evaluation of Q_4 to the evaluation of Q_{\dagger} . The reduction is analogous to Case (1). The way we encode the atoms $R(A, B)$ and $S(B)$, do preprocessing, and translate the updates is exactly the same as in Case (1). The only difference is the way we retrieve the B -values in $Q_4(B|a)$ for an input value a . We translate a into an input tuple to Q_{\dagger} where all input variables besides A are assigned to d . Recall that Q_{\dagger} might have several output variables besides B . By construction, they can be assigned only to d . Hence, all output tuples returned by Q_{\dagger} have distinct B -values. These B -values constitute the result of $Q_4(B|a)$. We conclude that Q_4 can be evaluated with $\mathcal{O}(N^{0.5-\gamma})$ enumeration delay and $\mathcal{O}(N^{0.5-\gamma})$ amortised update time, which contradicts the OMv conjecture.

□

Chapter 8

Related Work

Our work lies at the intersection of two lines of research: querying under access patterns and dynamic evaluation. Our work is the first to investigate the dynamic evaluation for CQAP queries. These CQAP queries are typically called *parameterized queries* or *prepared statements* in database management systems.

Access Patterns. The term CQAP queries in this thesis is different from the term “queries with access patterns” overwhelmingly used in the literature [33, 75, 27, 12, 13]. In these works, *input* relations have input and output variables and there is no restriction on whether they are bound or free. Also, a variable may be input in a relation and output in another and this leads to technically interesting but generally hard reasoning about the answerability of the query [56, 57, 52]. In our setting, only the free variables are split into input and output; an input (output) variable is then input (output) for all relations that have that variable in the body of the query. To differentiate the two settings, we call our setting *free access patterns*. For given values over the input variables, CQAP queries become *residual queries*. As shown in Example 5.29, our CQAP approach can be however more efficient than the evaluation of residual queries. To support efficient answering, we precompute (subject to a trade-off) some mappings between possible values for input variables and the query output, whereas a residual query is to be computed from scratch for the given input values.

Prior work closest in spirit to ours investigated the space-delay trade-off for the static evaluation of full conjunctive queries with output access patterns [26]. This work constructs a succinct representation of the query result that allows for the enumeration of those tuples that conform with value bindings of the input variables. The representation relies on a tree decomposition of the query where the input variables form a connected subtree. This work does not support queries with projection nor dynamic evaluation. Follow-up work considers the problem of answering Boolean conjunctive queries with ac-

Class of Queries	Preprocessing	Delay	Extra Space	Source
f.c. α -acyclic CQ \neq	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(N)$	[9]
f.c. β -acyclic negative CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[20, 19]
f.c. signed-acyclic CQ	$\mathcal{O}(N (\log N)^{ Q })$	$\mathcal{O}(1)$	–	[20]
Acyclic CQ \neq	$\mathcal{O}(N)$	$\mathcal{O}(N)$	$\mathcal{O}(N)$	[9]
CQ \neq of f.c. treewidth k	$\mathcal{O}(\text{Dom} ^{k+1} + N)$	$\mathcal{O}(1)$	–	[9]
CQ	$\mathcal{O}(N^{w(Q)})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{w(Q)})$	[65, 1]
Full CQ with access patterns	$\mathcal{O}(N^{\rho^*(Q)})$	$\mathcal{O}(\tau)$	$\mathcal{O}(N + N^{\rho^*(Q)}/\tau)$	[26]
CQ on \underline{X} -structures (trees, grids)	$\mathcal{O}(N)$	$\mathcal{O}(N)$	–	[8]
FO on Bounded degree	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[28, 46]
FO on Bounded expansion	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[47]
FO on Local bounded expansion	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	–	[71]
FO on Low degree	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{2+\gamma})$	[29]
FO on Nowhere dense	$\mathcal{O}(N^{1+\gamma})$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\gamma})$	[68]
MSO on Bounded treewidth	$\mathcal{O}(N)$	$\mathcal{O}(1)$	–	[7, 48]

Figure 8.1: Prior work on the trade-off between preprocessing time, enumeration delay, and extra space for different classes of queries (Conjunctive Queries, First-Order, Monadic Second-Order) and static databases under data complexity; f.c. stands for free-connex. Parameters: Query Q with factorization width w [65] and fractional edge cover number ρ^* [6]; database of size N ; slack τ is a function of N and ρ^* ; $\gamma > 0$. Most works do not discuss the extra space utilization (marked by –).

cess patterns, where every free variable is fixed to a constant at query time, again in the static setting [25].

Hierarchical Queries. The Boolean conjunctive queries without repeating relation symbols that can be computed in polynomial time on tuple-independent probabilistic databases are hierarchical; non-hierarchical queries are hard for $\#P$ [72]. This was extended to non-Boolean queries with negation [32].

Hierarchical queries are the conjunctive queries whose provenance admits a factorized representation where each input tuple occurs a constant number of times; any factorization of the provenance of a non-hierarchical query would require a number of occurrences of the provenance of some input tuple dependent on the input database size [64].

In the MPC model, the hierarchical queries admit parallel evaluation with one communication step [51]. The r -hierarchical queries, which are conjunctive queries that become hierarchical by repeatedly removing the atoms whose complete set of variables occurs in another atom, can be evaluated in the MPC model using a constant number of steps and optimal load on every single database instance [37].

Hierarchical queries also admit one-step streaming evaluation in the finite cursor

Class of Queries	Preprocessing	Update	Delay	Extra Space	Source
q -hierarchical CQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[14, 38]
Triangle count	$\mathcal{O}(N^{\frac{3}{2}})$	$\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(N^{1+\min\{\epsilon, 1-\epsilon\}})$	[41]
Full triangle query	$\mathcal{O}(N^{\frac{3}{2}})$	$\mathcal{O}(N^{\frac{1}{2}})^{\dagger}$	$\mathcal{O}(1)$	$\mathcal{O}(N^{\frac{3}{2}})$	[42]
q -hierarchical UCQ	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[17]
FO+MOD on Bounded degree	$\mathcal{O}(N)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	–	[15]
MSO on Strings	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$	–	[61]

Figure 8.2: Prior work on the trade-off between preprocessing time, update time, enumeration delay, and extra space for different classes of queries (Conjunctive Queries, Count Queries, First-Order Queries with modulo-counting quantifiers, Monadic Second Order Logic) and databases under updates in data complexity. Parameters: Query Q ; database of size N ; $\epsilon \in [0, 1]$. Most works do not discuss the extra space utilization (marked by –). \dagger : amortized update time.

model [34]. Under updates, the q -hierarchical queries are the conjunctive queries that admit constant-time update and delay [14]. The q -hierarchical queries are a proper subclass of both the free-connex α -acyclic and hierarchical queries. In addition to being hierarchical, a second condition holds for a q -hierarchical query: if the set of atoms of a free variable is strictly contained in the set of another variable, then the latter is also free.

Figure 8.1 gives taxonomies of works on static query evaluation for hierarchical queries. Prior work exhibits a dependency between the space and enumeration delay for conjunctive queries with access patterns [26]. It constructs a succinct representation of the query result that allows for enumeration of tuples over some variables under value bindings for all other variables. It does not support enumeration for queries with free variables, as addressed in our work.

The result of any α -acyclic conjunctive query can be enumerated with constant delay after linear-time preprocessing if and only if it is free-connex. This is under the conjecture that Boolean multiplication of $n \times n$ matrices cannot be done in $O(n^2)$ time [9]. More recently, this was shown to hold also under the hypothesis that the existence of a triangle in a hypergraph of n vertices cannot be tested in time $\mathcal{O}(n^2)$ and that for any k , testing the presence of a k -dimensional tetrahedron cannot be decided in linear time [20]. The free-connex characterization generalizes in the presence of functional dependencies [22]. An in-depth pre-2015 overview on constant-delay enumeration is provided by Segoufin [70].

There are also enumeration algorithms for document spanners [5] and satisfying valuations of circuits [3].

Figure 8.2 shows the works on dynamic query evaluation for hierarchical queries. The q -hierarchical queries are the conjunctive queries that admit linear-time preprocessing

and constant-time update and delay [14, 38]. If a conjunctive query without repeating relation symbols is not q -hierarchical, there is no $\gamma > 0$ such that the query result can be enumerated with $\mathcal{O}(N^{\frac{1}{2}-\gamma})$ delay and update time, unless the Online Matrix Vector Multiplication conjecture fails. The constant delay and update time carry over to first-order queries with modulo-counting quantifiers on bounded degree databases, unions of q -hierarchical queries [17], and q -hierarchical queries with small domain constraints [15].

MSO queries on strings admit linear-time preprocessing, constant delay, and logarithmic update time. Here, updates can relabel, insert, or remove positions in the string. Further work considers MSO queries on trees under updates [60, 54, 4].

DBToaster [50], F-IVM [62], and DynYannakakis [38, 39] are recent systems implementing incremental view maintenance approaches.

Triangle Queries. The problem of incrementally maintaining the triangle count has received a fair amount of attention. Existing exact approaches require at least linear time in worst case. After each update to a database D , the naïve approach joins the relations R , S , and T in time $\mathcal{O}(N^{\frac{3}{2}})$ using a worst-case optimal algorithm [2, 59] and counts the result tuples. The number of distinct tuples in the result is at most $N^{\frac{3}{2}}$, which is a well-known result by Loomis and Whitney from 1949 (see recent notes on the history of this result [58]). The classical first-order IVM [23] computes on the fly a delta query δQ per single-tuple update in $\mathcal{O}(N)$ time. The recursive IVM [50] speeds up the delta computation by precomputing auxiliary views representing the update-independent parts. These views allow us to compute the delta query for single-tuple updates to the input relations in $\mathcal{O}(1)$ time, but maintaining these views still requires $\mathcal{O}(N)$ time.

Further away from our line of work is the development of dynamic descriptive complexity, starting with the DynFO complexity class and the much-acclaimed result on FO expressibility of the maintenance for graph reachability under edge inserts and deletes, cf. a recent survey [69]. The k -clique query can be maintained under edge inserts by a quantifier-free update program of arity $k - 1$ but not of arity $k - 2$ [76].

A distinct line of work investigates randomized approximation schemes with an arbitrary relative error for counting triangles in a graph given as a stream of edges, e.g., [10, 40, 21, 55, 24]. Each edge in the data stream corresponds to a tuple insert, and tuple deletes are not considered. The emphasis of these approaches is on space efficiency, and they express the space utilization as a function of the number of nodes and edges in the input graph and of the number of triangles. The space utilization is generally sublinear but may become superlinear if, for instance, the number of edges is greater than the square root of the number of triangles. The update time is polylogarithmic in the number of nodes in the graph.

Cutset optimisations. Cutset conditioning [66] and cutset sampling [18] are used for efficient exact and approximate inference in Bayesian networks. The idea is to *choose* a cutset, which is a subset of variables, such that conditioning on the variables in the cutset, i.e., instantiating them with possible values, yields a network with a small treewidth that allows exact inference. The set of input variables of a CQAP can be seen as a *given* cutset, while fixing the input variables to given values is conditioning. Query fracturing, as introduced in our work, is a query rewriting technique that does not have a counterpart in cutset optimisations in AI.

Chapter 9

Extensions

In this chapter, we present several extensions of the main results of this thesis.

Relations Over Task-Specific Semirings. In this thesis, we model the relations as factors over the sum-product semiring of integers. We can use it to simulate the bag semantics of relational databases. For other applications, other (semi)rings can be used. Different rings can be used as the domain of tuple multiplicities (or payloads). Previous work shows how the data-intensive computation of many applications can be captured by application-specific rings, which define sum and product operations over data values [62]. For example, the *relational data ring* supports payloads with listing and factorized representations of relations, and the *degree- m matrix ring* supports payloads that can be used for maintaining the mutual information of two discrete random variables and the gradients of square loss functions for linear regression models [62, 63].

Loomis Whitney queries The maintenance strategies for the triangle queries (without access patterns) naturally extend to Loomis Whitney (LW) queries. LW queries generalize triangle queries from cliques of degree three to cliques of degree $n \geq 3$; they encode the Loomis Whitney inequality [53]. Let A_1, \dots, A_n be the query variables and R_1, \dots, R_n relations over schemas $\mathbf{X}_1, \dots, \mathbf{X}_n$, where $\forall i \in [n] : \mathbf{X}_i = (A_{((i+j) \bmod n)+1})_{-1 \leq j \leq n-3}$. That is, the schema of R_1 is (A_1, \dots, A_{n-1}) , whereas the schema of R_n is $(A_n, A_1, \dots, A_{n-2})$. The n -ary LW query of degree n has the form

$$\diamond_n(\mathbf{x}) = R_1(\mathbf{x}_1) \cdots R_n(\mathbf{x}_n),$$

where $\mathbf{x} = (a_j)_{j \in [n]}$ and for all $i \in [n]$, $\mathbf{x}_i = (a_{((i+j) \bmod n)+1})_{-1 \leq j \leq n-3}$ is a value from the domain of the tuple \mathbf{X}_i of variables. As for triangle queries, a LW query of degree n and arity $0 \leq k \leq n - 1$ has the same body as for arity n but only keeps the first k values in the result. For instance, for $n = 4$ the binary LW query is

$$\diamond_2(A_1, A_2) = R_1(A_1, A_2, A_3), R_2(A_2, A_3, A_4), R_3(A_3, A_4, A_1), R_4(A_4, A_1, A_2).$$

In case $n = 3$, each LW query \diamond_k becomes the triangle query with k free variables, for $0 \leq k \leq 3$.

The extended approach achieves the following complexities for LW queries of degree n (stated without proof):

- The preprocessing and amortized update time are the same as for triangle queries: $\mathcal{O}(N^{\frac{3}{2}})$ preprocessing time and $\mathcal{O}(N^{\max\{\epsilon, 1-\epsilon\}})$ amortized update time.
- For the LW queries with $k = 0$ or n , the enumeration delay is constant; for k -ary LW queries where $0 < k < n$, the enumeration delay is $\mathcal{O}(N^{\min\{1, (n-k) \cdot (1-\epsilon)\}})$. The delay hence improves with increasing arity. For $n = 3$, we get exactly the same enumeration delay as for the triangle queries.
- The lower bound on the update-delay trade-off for triangle queries stated in Proposition 6.8 can be extended to LW queries. This means that at $\epsilon = \frac{1}{2}$, kekeis strongly Pareto worst-case optimal for the LW queries with $k = 0$ or n and weakly Pareto worst-case optimal for all other LW queries.

The result of the n -ary LW query \diamond_n of degree n has size $\mathcal{O}(N^{\frac{n}{n-1}})$ [53]. It can also be computed in the static setting in the same time, which is thus worst-case optimal [58]. We cannot recover the optimality in the static case, since it takes $\mathcal{O}(N^{\frac{1}{2}})$ amortized time per each single-tuple update and there are N tuples to insert. Since the combination of $\mathcal{O}(N^{\frac{1}{2}})$ amortized time and $\mathcal{O}(1)$ delay is strongly Pareto worst-case optimal, it means that no dynamic algorithm can achieve a lower amortized single-tuple update time for the n -ary LW query. This shows the limitation of single-tuple updates. To achieve the overall $\mathcal{O}(N^{\frac{n}{n-1}})$ time for N tuple inserts, one would need to process several inserts at the same time, that is, in bulk, such that the amortized time per insert should be $\mathcal{O}(N^{\frac{1}{n-1}})$. A characterization of the difference between bulk updates and single-tuple updates remains an interesting open problem.

Evaluation Trade-Offs for Any Acyclic CQs. Recall we present in Chapters 5 and 6 an approach that exposes trade-offs in the dynamic evaluation of triangle CQAP queries and CQAP queries with hierarchical fractures. This approach is based on the idea of partitioning the data into heavy and light parts and employing adaptive evaluation strategies to data with different degrees.

We extend this idea and develop a new approach [45] that exposes trade-offs between the preprocessing time and the enumeration delay in the static evaluation of any acyclic CQs. For the static evaluation of any acyclic CQs, there are two prior results that are the two extremes in the trade-off space: in one extreme, the seminal work [9] showed

that such queries can be evaluated with linear preprocessing time and linear enumeration delay. If the query is free-connex, the enumeration delay becomes constant. In the other extreme, prior work [65] showed that constant enumeration delay can be achieved for arbitrary acyclic queries at the expense of a preprocessing time that is characterized by the fractional hypertree width. We can recover the two extremes in the trade-off space, and for some queries, we reveal new trade-offs between the two extremes.

Chapter 10

Conclusion and Future Work

This thesis investigates the dynamic evaluation of conjunctive queries with output access patterns. We developed a dynamic evaluation approach for conjunctive queries with free access patterns. The approach relies on the two new notions access-top variable order and query fracture. We introduced the static and dynamic widths to capture the complexities of the preprocessing and respectively update steps.

We also establish a dichotomy: CQAP_0 queries are precisely those queries with constant-time update and delay unless the Online Matrix-Vector conjecture fails. This dichotomy is sensitive to the access pattern.

We further develop an algorithm that reveals the trade-offs between preprocessing, update, and enumeration for the triangle CQAP queries and CQAP queries with hierarchical fractures. Our algorithm has two core ideas. First, we partition the input relations into heavy and light parts based on the degrees of the values. This transforms a query over the input relations into a union of queries over heavy and light relation parts. Second, we employ different evaluation strategies for different heavy-light combinations of parts of the input relations. By tuning the threshold for the heavy-light partitioning, our algorithm can reduce update time with the cost of more enumeration delay, or the other way around. We show the strongly and weakly Pareto of our algorithm for the triangle CQAP queries and the CQAP_1 queries.

There are several lines of further work. The first is to generalize our trade-off for all CQAP queries and even to functional aggregate queries. The second is to extend the result to CQAP queries with group-by aggregates and order-by clauses. An open problem is to find lower bounds for queries beyond the triangle CQAP queries and the CQAP_0 and CQAP_1 queries, in particular, the lower bounds for CQAP_i queries for $i > 1$.

References

- [1] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, pages 13–28, 2016.
- [2] N. Alon, R. Yuster, and U. Zwick. Finding and Counting Given Length Cycles. *Algorithmica*, 17(3):209–223, 1997.
- [3] Antoine Amarilli, Pierre Bourhis, Louis Jachiet, and Stefan Mengel. A circuit-based approach to efficient enumeration. In *ICALP*, pages 111:1–111:15, 2017.
- [4] Antoine Amarilli, Pierre Bourhis, and Stefan Mengel. Enumeration on trees under relabelings. In *ICDT*, pages 5:1–5:18, 2018.
- [5] Antoine Amarilli, Pierre Bourhis, Stefan Mengel, and Matthias Niewerth. Constant-delay enumeration for nondeterministic document spanners. In *ICDT*, pages 22:1–22:19, 2019.
- [6] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. *SIAM J. Comput.*, 42(4):1737–1767, 2013.
- [7] Guillaume Bagan. MSO Queries on Tree Decomposable Structures Are Computable with Linear Delay. In *CSL*, pages 167–181, 2006.
- [8] Guillaume Bagan, Arnaud Durand, Emmanuel Filiot, and Olivier Gauwin. Efficient Enumeration for Conjunctive Queries over X-underbar Structures. In *CSL*, pages 80–94, 2010.
- [9] Guillaume Bagan, Arnaud Durand, and Etienne Grandjean. On Acyclic Conjunctive Queries and Constant Delay Enumeration. In *CSL*, pages 208–222, 2007.
- [10] Ziv Bar-Yossef, Ravi Kumar, and D Sivakumar. Reductions in Streaming Algorithms, with an Application to Counting Triangles in Graphs. In *SODA*, pages 623–632, 2002.

- [11] Catriel Beeri, Ronald Fagin, David Maier, and Mihalis Yannakakis. On the Desirability of Acyclic Database Schemes. *J. ACM*, 30(3):479–513, 1983.
- [12] Michael Benedikt, Julien Leblay, and Efthymia Tsamoura. Querying with Access Patterns and Integrity Constraints. *VLDB*, 8(6):690–701, 2015.
- [13] Michael Benedikt, Balder Ten Cate, and Efthymia Tsamoura. Generating Low-cost Plans from Proofs. In *PODS*, pages 200–211, 2014.
- [14] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering Conjunctive Queries Under Updates. In *PODS*, pages 303–318, 2017.
- [15] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering FO+MOD queries under updates on bounded degree databases. In *ICDT*, pages 8:1–8:18, 2017.
- [16] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering UCQs Under Updates and in the Presence of Integrity Constraints. In *ICDT*, pages 8:1–8:19, 2018.
- [17] Christoph Berkholz, Jens Keppeler, and Nicole Schweikardt. Answering ucqs under updates and in the presence of integrity constraints. In *ICDT*, pages 8:1–8:19, 2018.
- [18] Bozhena Bidyuk and Rina Dechter. Cutset sampling for bayesian networks. *J. Artif. Intell. Res.*, 28:1–48, 2007.
- [19] Johann Brault-Baron. A Negative Conjunctive Query is Easy if and only if it is Beta-Acyclic. In *CSL*, pages 137–151, 2012.
- [20] Johann Brault-Baron. *De la pertinence de l'énumération: complexité en logiques propositionnelle et du premier ordre*. PhD thesis, Université de Caen, 2013.
- [21] Luciana S Buriol, Gereon Frahling, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Christian Sohler. Counting Triangles in Data Streams. In *PODS*, pages 253–262, 2006.
- [22] Nofar Carmeli and Markus Kröll. Enumeration complexity of conjunctive queries with functional dependencies. In *ICDT*, pages 11:1–11:17, 2018.
- [23] Rada Chirkova and Jun Yang. Materialized Views. *Found. & Trends DB*, 4(4):295–405, 2012.
- [24] Graham Cormode and Hossein Jowhari. A Second Look at Counting Triangles in Graph Streams (Corrected). *Theor. Comput. Sci.*, 683:22–30, 2017.

- [25] Shaleen Deep, Xiao Hu, and Paraschos Koutris. Space-Time Tradeoffs for Answering Boolean Conjunctive Queries. *arXiv*, abs/2109.10889, 2021.
- [26] Shaleen Deep and Paraschos Koutris. Compressed Representations of Conjunctive Query Results. In *PODS*, pages 307–322, 2018.
- [27] Alin Deutsch, Bertram Ludäscher, and Alan Nash. Rewriting Queries using Views with Access Patterns under Integrity Constraints. *Theor. Comput. Sci.*, 371(3):200–226, 2007.
- [28] Arnaud Durand and Etienne Grandjean. First-order Queries on Structures of Bounded Degree are Computable with Constant Delay. *ACM Trans. Comput. Logic*, 8(4):21, 2007.
- [29] Arnaud Durand, Nicole Schweikardt, and Luc Segoufin. Enumerating Answers to First-order Queries over Databases of Low Degree. In *PODS*, pages 121–131, 2014.
- [30] Arnaud Durand and Yann Strozecki. Enumeration complexity of logical query problems with second-order variables. In *CSL*, pages 189–202, 2011.
- [31] Ronald Fagin, Alberto O. Mendelzon, and Jeffrey D. Ullman. A simplified universal relation assumption and its properties. *ACM Trans. Database Syst.*, 7(3):343–360, 1982.
- [32] Robert Fink and Dan Olteanu. Dichotomies for queries with negation in probabilistic databases. *ACM Trans. Datab. Syst.*, 41(1):4:1–4:47, 2016.
- [33] Daniela Florescu, Alon Levy, Ioana Manolescu, and Dan Suciu. Query Optimization in the Presence of Limited Access Patterns. *SIGMOD Rec.*, 28(2):311–322, 1999.
- [34] Martin Grohe, Yuri Gurevich, Dirk Leinders, Nicole Schweikardt, Jerzy Tyszkiewicz, and Jan Van den Bussche. Database Query Processing Using Finite Cursor Machines. *Theory Comput. Syst.*, 44(4):533–560, 2009.
- [35] Martin Grohe and Dániel Marx. Constraint Solving via Fractional Edge Covers. *ACM Trans. Alg.*, 11(1):4:1–4:20, 2014.
- [36] Monika Henzinger, Sebastian Krinninger, Danupon Nanongkai, and Thatchaphol Saranurak. Unifying and Strengthening Hardness for Dynamic Problems via the Online Matrix-Vector Multiplication Conjecture. In *STOC*, pages 21–30, 2015.
- [37] Xiao Hu and Ke Yi. Instance and output optimal parallel algorithms for acyclic joins. In *PODS*, pages 450–463, 2019.

- [38] Muhammad Idris, Martín Ugarte, and Stijn Vansummeren. The Dynamic Yannakakis Algorithm: Compact and Efficient Query Processing Under Updates. In *SIGMOD*, pages 1259–1274, 2017.
- [39] Muhammad Idris, Martín Ugarte, Stijn Vansummeren, Hannes Voigt, and Wolfgang Lehner. Conjunctive queries with inequalities under updates. *PVLDB*, pages 733–745, 2018.
- [40] Hossein Jowhari and Mohammad Ghodsi. New Streaming Algorithms for Counting Triangles in Graphs. In *COCOON*, pages 710–716, 2005.
- [41] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Counting triangles under updates in worst-case optimal time. In *ICDT*, pages 4:1–4:18, 2019.
- [42] Ahmet Kara, Hung Q. Ngo, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Maintaining triangle queries under updates. *ACM Trans. Database Syst.*, 45(3):11:1–11:46, 2020.
- [43] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Trade-offs in Static and Dynamic Evaluation of Hierarchical Queries. In *PODS*, pages 375–392, 2020.
- [44] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Conjunctive queries with output access patterns under updates. *CoRR*, abs/2206.09032, 2022.
- [45] Ahmet Kara, Milos Nikolic, Dan Olteanu, and Haozhe Zhang. Evaluation trade-offs for acyclic conjunctive queries. In *CSL*, October 2022.
- [46] Wojciech Kazana and Luc Segoufin. First-order Query Evaluation on Structures of Bounded Degree. *LMCS*, 7(2), 2011.
- [47] Wojciech Kazana and Luc Segoufin. Enumeration of First-order Queries on Classes of Structures with Bounded Expansion. In *PODS*, pages 297–308, 2013.
- [48] Wojciech Kazana and Luc Segoufin. Enumeration of Monadic Second-order Queries on Trees. *ACM Trans. Comput. Logic*, 14(4):25:1–25:12, 2013.
- [49] Christoph Koch. Incremental Query Evaluation in a Ring of Databases. In *PODS*, pages 87–98, 2010.
- [50] Christoph Koch et al. DBToaster: Higher-order Delta Processing for Dynamic, Frequently Fresh Views. *VLDB J.*, 23(2):253–278, 2014.

- [51] Paraschos Koutris and Dan Suciu. Parallel Evaluation of Conjunctive Queries. In *PODS*, pages 223–234, 2011.
- [52] Chen Li and Edward Chang. On Answering Queries in the Presence of Limited Access Patterns. In *ICDT*, pages 219–233, 2001.
- [53] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Journal: Bull. Amer. Math. Soc.*, 55(55):961–962, 1949. DOI: 10.1090/S0002-9904-1949-09320-5.
- [54] Katja Losemann and Wim Martens. MSO queries on trees: enumerating answers under updates. In *CSL-LICS*, pages 67:1–67:10, 2014.
- [55] Andrew McGregor, Sofya Vorotnikova, and Hoa T Vu. Better Algorithms for Counting Triangles in Data Streams. In *PODS*, pages 401–411, 2016.
- [56] Alan Nash and Bertram Ludäscher. Processing First-Order Queries under Limited Access Patterns. In *PODS*, pages 307–318, 2004.
- [57] Alan Nash and Bertram Ludäscher. Processing Unions of Conjunctive Queries with Negation under Limited Access Patterns. In *EDBT*, pages 422–440, 2004.
- [58] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *J. ACM*, 65(3):16:1–16:40, 2018.
- [59] Hung Q. Ngo, Christopher Ré, and Atri Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Rec.*, 42(4):5–16, 2013.
- [60] Matthias Niewerth. MSO queries on trees: Enumerating answers under updates using forest algebras. In *LICS*, pages 769–778, 2018.
- [61] Matthias Niewerth and Luc Segoufin. Enumeration of MSO queries on strings with constant delay and logarithmic updates. In *PODS*, pages 179–191, 2018.
- [62] Milos Nikolic and Dan Olteanu. Incremental View Maintenance with Triple Lock Factorization Benefits. In *SIGMOD*, pages 365–380, 2018.
- [63] Milos Nikolic, Haozhe Zhang, Ahmet Kara, and Dan Olteanu. F-IVM: learning over fast-evolving relational data. In David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2773–2776. ACM, 2020.

- [64] Dan Olteanu and Závodný. Factorised representations of query results: size bounds and readability. In *ICDT*, pages 285–298, 2012.
- [65] Dan Olteanu and Jakub Závodný. Size Bounds for Factorised Representations of Query Results. *ACM TODS*, 40(1):2:1–2:44, 2015.
- [66] Judea Pearl. *Probabilistic reasoning in intelligent systems - networks of plausible inference*. Morgan Kaufmann series in representation and reasoning. Morgan Kaufmann, 1989.
- [67] Neil Robertson and Paul D. Seymour. Graph minors. II. algorithmic aspects of tree-width. *J. Algorithms*, 7(3):309–322, 1986.
- [68] Nicole Schweikardt, Luc Segoufin, and Alexandre Vigny. Enumeration for FO Queries over Nowhere Dense Graphs. In *PODS*, pages 151–163, 2018.
- [69] Thomas Schwentick and Thomas Zeume. Dynamic Complexity: Recent Updates. *SIGLOG News*, 3(2):30–52, 2016.
- [70] Luc Segoufin. Constant delay enumeration for conjunctive queries. *SIGMOD Rec.*, 44(1):10–17, 2015.
- [71] Luc Segoufin and Alexandre Vigny. Constant Delay Enumeration for FO Queries over Databases with Local Bounded Expansion. In *ICDT*, pages 20:1–20:16, 2017.
- [72] Dan Suciu, Dan Olteanu, Christopher Ré, and Christoph Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.
- [73] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In *ICDT*, pages 96–106, 2014.
- [74] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *VLDB*, pages 82–94, 1981.
- [75] Ramana Yerneni, Chen Li, Jeffrey Ullman, and Hector Garcia-Molina. Optimizing Large Join Queries in Mediation Systems. In *ICDT*, pages 348–364, 1999.
- [76] Thomas Zeume. The Dynamic Descriptive Complexity of k-Clique. *Inf. Comput.*, 256:9–22, 2017.