TU WIEN Informatics

Integration von Worst-Case Optimal Join Algorithmen in Spalten-Orientierte Datenbanken

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Martin Ledl, BSc
Matrikelnummer 01634019

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Mitwirkung: Dipl.-Ing. Dr.techn. Matthias Lanzinger

Wien, 20. Dezember 2021

Martin Ledl	Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Integration of Worst-Case Optimal Join Algorithms into Column-Stores

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Martin Ledl, BSc
Registration Number 01634019

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler
Assistance: Dipl.-Ing. Dr.techn. Matthias Lanzinger

Vienna, 20th December, 2021

Martin Ledl	Reinhard Pichler

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Martin Ledl, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 20. Dezember 2021

Martin Ledl

Danksagung

Ich möchte mich an dieser Stelle recht herzlich bei meinem Betreuer Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler für die Hilfe zu Fragen aus dem Bereich der Datenbanktheorie bedanken. Weiters hat mir Ihr konstruktives Feedback zu den diversen Kapiteln und Ausarbeitungen dieser Arbeit sehr dabei geholfen, das Hintergrundwissen sowie meine Ausarbeitung und Ergebnisse verständlich zu verschriftlichen. Mein Dank gilt auch Dipl.-Ing. Dr.techn. Matthias Lanzinger, der mich auf das Thema dieser Arbeit aufmerksam gemacht hat und der mich vor allem bei der Evaluierung der Ergebnisse unterstützt hat. Abschließend möchte ich mich recht herzlich bei meinen Eltern Aloisia und Josef Ledl bedanken, die mich während meines Studiums bedingungslos unterstützt haben.

Acknowledgements

At this point I would like to thank my supervisor Univ.Prof. Mag.rer.nat. Dr.techn. Reinhard Pichler for the provided help to questions in the field of database theory. Furthermore, your constructive feedback on various chapters and elaborations helped me a lot to write background knowledge, the contribution of this thesis as well as its results in an understandable way. Special thanks further go to Dipl.-Ing. Dr.techn. Matthias Lanzinger, who drew my attention to the topic of this thesis and who supported me in various belongings, especially with the evaluation of this work. Finally, I would like to thank my parents Aloisia and Josef Ledl, who supported me unconditionally during my studies.

Kurzfassung

Business Intelligence und andere analytische Systeme müssen mit immer größer werdenden Datenmengen umgehen und Abfragen so schnell wie möglich beantworten können. Die Auswertung von Joins ist eine zentrale Herausforderung für Datenbanksysteme, um gestellte Abfragen effizient beantworten zu können. Diese Anforderung hebt die Notwendigkeit von Optimierungen des Berechnungsprozesses von Datenbankabfragen hervor. Deshalb wurde viel Forschung in diesem Bereich betrieben, um Konzepte zur ressourcenschonenden und laufzeiteffizienten Berechnung von Join-Abfragen zu entwickeln.

Der Forschungsbereich von Spalten-orientierten Datenbanksystemen hat eine neue Datenbankarchitektur hervorgebracht, die Mängel von traditionellen Zeilen-orientierten Systemen verbessern soll. Diese Systeme zerteilen Datenbanken vertikal und speichern Spalten unabhängig voneinander. Über Jahrzehnte wurden viele Konzepte erforscht und entwickelt, die die Effizienz von Spalten-orientierten Datenbanksystemen verbessern. Ein anderer Forschungsbereich, der sich mit Abfragen-Optimierung beschäftigt, erforscht Worst-Case Optimal Joins. Dieser Bereich hat theoretische Konzepte und Algorithmen hervorgebracht, die enge Laufzeitgrenzen auf Join-Algorithmen definieren. Beide Forschungsbereiche haben, unabhängig voneinander, erfolgreiche Arbeit im Zusammenhang Join-Optimierung geleistet. Diese Arbeit ist die Erste, die das Potenzial der Join-Optimierung erforscht, die aus einer Kombination von Konzepten aus beiden Forschungsbereichen folgt.

Weiters diskutiert diese Arbeit Forschungsergebnisse aus beiden Bereichen und stellt einen Ansatz zur Integration von Worst-Case Optimal Join Algorithmen vor. Der Integrationsprozess diskutiert den Abfragen-Compiler eines spezifischen Spalten-orientierten Datenbanksystems sowie die manuelle Übersetzung eines Worst-Case Optimal Join Algorithmus in die interne Sprache des Systems. Abschließend behandelt diese Arbeit die Integration dieser manuellen Übersetzung in den Compiler des Systems, sowie die experimentelle Auswertung des daraus resultierende Datenbanksystems.

Diese Auswertung hat gezeigt, dass unser System die getesteten natürlichen Join-Abfragen effizienter auswertet als das originale Datenbanksystem, wenn eine Schräglage in den Daten vorliegt. Weiters hat sich die Laufzeitdifferenz zwischen den beiden evaluierten Systemen, mit zunehmender Tabellengröße, erhöht. Die Arbeit belegt ebenfalls, dass unser System Abfragen auf Datenbanken mit Tabellengrößen auswerten kann, die zu Speicherfehlern beim originalen System führen. Diese Auswertung unterstreicht die praktische Relevanz von Worst-Case Optimal Joins im Kampf gegen Schräglagen in Daten.

Abstract

Business intelligence and analytical systems have to handle an increasing amount of data and require to be capable of answering queries as fast as possible. Therefore, the evaluation of joins has become a major challenge to be tackled by any database system in order to efficiently evaluate queries. To that end, plenty of research has been conducted to introduce concepts to compute join queries in a resource and runtime efficient way.
The research area of column-oriented database systems aims for overcoming shortcomings of traditional row-wise systems by designing a new database architecture from scratch. Such column-stores vertically partition a database and store columns independently of each other. Furthermore, various techniques that enhance performance of column-oriented database systems have been introduced in this field over the past decades. Another research area that aims for optimizing query performance deals with worst-case optimal joins. Theoretical concepts and algorithms that define tight bounds on join algorithms' runtime have been introduced in this field. Both research areas successfully came up with join optimization concepts independently of each other. Since a combination of concepts from both fields yield great potential for further performance optimization, this thesis aims for integrating worst-case optimal joins into column-stores and is the first work to perform and evaluate such combination.
Moreover, this thesis discusses research conducted in both areas and introduces a way of integrating worst-case optimal join algorithms into column-oriented database systems based on the outlined knowledge. The integration process discusses the query compiler of a specific column-store system and the manual translation of a class of worst-case optimal join algorithms into its internal language. Finally, the result of the manual translation is integrated into the column-store's query compiler and the resulting system is experimentally evaluated against the original system.
The experimental evaluation showed that our system outperforms the original one on a given set of natural join queries in settings with skewed data. Moreover, the difference in runtime performance between both systems continuously increased with higher input relation sizes. This thesis further showed that the column-store with worst-case optimal join algorithm integrated can evaluate natural join queries with big input relation sizes which cause memory allocation errors for the original systems. Finally, the evaluation underlines the practical potential of worst-case optimal join algorithms for fighting skew in input data.

Contents

CHAPTER 1

Introduction

An increasing number of business intelligence and analytical systems are essential to various domains of today's life. These systems are required to process an increasing amount of data and have to serve growing database workloads, which inevitably lead to performance problems. Moreover, the structure of data to be queried can impact the overall query performance. Therefore, querying skewed data can have severe performance implications for database systems. Due to these circumstances, there has been a pressing need for performance enhancement in query processing and many promising concepts have been developed to tackle these challenges. Furthermore, a lot of interdisciplinary research has been conducted over the last decades to improve the overall performance of database systems. Two interesting research areas in this context are column-oriented database systems and Worst Case Optimal Join (WCOJ) algorithms, which both have independently achieved great performance benefits for query processing.

Joins represent key operations in every database system and their evaluation is expensive. Therefore, enhancing performance of join operations will be beneficial for the overall query performance. Thus, research in the field of worst-case optimal joins aims for defining tight bounds on join algorithms' runtime. Ngo et al. [NRR13] state that traditional join plans are suboptimal, because join queries are evaluated by obtaining the best pairwise join plan. WCOJ algorithms, which evaluate conjunctive queries in time that matches the worst-case output size of the query, lead to provably asymptotically better performance than the pairwise join paradigm [Ngo18]. Moreover, the performance of WCOJ algorithm depends on the input relations of the query. Ngo et al. [NRR13] introduced a class of WCOJ join algorithms that has a recursive structure. Their `Generic-Join` algorithm 3.1 tightly bounds the runtime of a query to the maximum output size and is a generalization of two worst-case optimal join algorithms which are the Leapfrog Triejoin [Vel14] and an algorithm proposed by Ngo et al. [NPRR12]. This `Generic-Join` algorithm 3.1 aims for evaluating `Multi-Way-Joins` in a worst-case optimal manner and is proven to be asymptotically better than pairwise join

evaluation. Therefore, the relevance of WCOJ algorithms to commercial and academic data management and analytical systems increased, since such systems have to deal with big amounts of data and query performance becomes more and more crucial [Ngo18]. Furthermore, quite a number of notable implementations and integration of WCOJ algorithms have been carried out in the area of database system research. Aberger et al. [ALT+17] introduced `EmptyHeaded`, which is a relational engine for graph processing as well as `LevelHeaded` [ALOR17], which is an in-memory query processing engine that is not restricted to graphs. Both engines evaluate queries according to the WCOJ approach. Besides that, Freitag et al. [FBS+20] present an implementation of a class of WCOJ algorithms into their UMBRO DBMS.

Another approach to tackle the increasing amounts of data is to introduce a database paradigm that empowers to efficiently deal with data-intensive applications. To that end, column-oriented database systems, also known as column-stores, have been designed to efficiently handle big amounts of data. The column-store architecture has been built from scratch to enhance performance of analytical systems and column-stores are therefore designed in a fundamentally different way than row-oriented database systems. Column-stores compose various features to increase performance on data-intensive workloads. A fundamental difference to row-stores is that column-oriented database systems vertically partition a database and store columns separately instead of row-wise tuples. This `column-at-a-time` storage pattern has the advantage that only attributes relevant to the query need to be loaded from disk instead of entire rows. This enables vectorized processing and column-stores further aim for applying operations on cache-line size blocks of data in order to reduce cache misses and disk reads as well as to utilize modern Central Processing Units (CPUs). Column-stores benefit from compiler optimization techniques, since vectorized operations can be implemented as tight loops over arrays, for instance. Such pieces of code suit compiler optimizations well, which can lead to performance enhancements on modern CPUs through Single Instruction Multiple Data (SIMD) parallelism, for example. Moreover, some column-stores are able to store compressed data and directly operate on such data. An important architectural feature of column-stores with respect to join processing is late materialization, where the construction of tuples within the query execution is delayed to the latest possible point during execution. To that end, operators in column-stores produce and maintain position lists that can be utilized for materialization. [ABH+13]

Specific column-store implementations can be found in the academic and commercial area. One popular open-source system, developed in academia, is MonetDB [IGN+12]. MonetDB is actively used in health-care, telecommunications and science nowadays and integrates many design concepts and architectural features that are typical for column-oriented database systems. MonetDB was designed for data warehouse applications and is leveraged in high-performance business intelligence and data mining applications. [IGN+12]

Column-stores and WCOJ algorithms are two promising concepts for enhancing performance of query processing. Both research areas provide concepts that successfully

led to more efficient systems dealing with data-intensive tasks independently of each other. Nevertheless, the traditional pairwise join evaluation paradigm of column-stores can cause performance drawbacks for data-intensive workloads with skewed data. A promising approach to tackle this problem, that has not been researched yet, is the integration of WCOJs into column-oriented database systems. Furthermore, the potential benefits of combining concepts from both research areas has not been investigated yet. Integrating WCOJs into column-oriented database systems is promising due to the `column-at-a-time` nature and architectural features of column-oriented database systems. Moreover, column-stores capable of evaluating joins in a `Multi-Way` manner as introduced by WCOJ algorithms promise to answer queries more efficiently in settings with skewed data.

The goal of this thesis is to show that the integration of WCOJ algorithms into column-stores leads to increased query performance. The contribution of this work answers an open question of Ngo et al. [NRR13] from the database systems research area. Their open question is about whether the algorithmic ideas presented in their survey can gain runtime efficiency in database systems. Furthermore, the potential performance benefits of combining the two introduced research fields is evaluated on a specific set of queries. This evaluation points out the potential power of column-stores capable of computing joins according to the WCOJ approach. Moreover, the impact of skew is discussed and it is shown that the performance difference between a column-store with pairwise join evaluation and one with WCOJ algorithms integrated will increase with increasing amount of input data as well as increasing skew in the input data.

In order to be able to integrate WCOJs into column-oriented database systems, a fundamental knowledge base of both research areas has been established by conducting literature reviews in both fields. Moreover, the integration is carried out on the example of the MonetDB column-store and in order to find the most efficient way of integrating a class of WCOJ algorithms into the system, it has been analyzed through reverse engineering. This helps to understand the original query compiler's structure and encourages to choose the most suitable integration possibility. The class of WCOJ algorithms to be integrated is given by the `Generic-Join` algorithm 3.1 that needs to be translated manually into a MonetDB Assembly Language (MAL) program, which is MonetDB's internal language. At first, a direct translation is carried out which will be improved upon based on the corresponding evaluation results. The integration process is based on the knowledge about MonetDB's query compiler and the translated `Generic-Join` algorithm 3.1. The result are two MonetDB systems, one with direct translation and the other one with optimized translation integrated. Performance benchmarks of these systems are conducted and the results are compared to the original MonetDB system in order to point out performance benefits resulting from integrating WCOJ algorithms into column-stores. Finally, the impact of skew in data is evaluated. The resulting system and its evaluation has been made publicly available on Github and can be accessed under `https://github.com/mledl/MonetDB_WCOJ`.

This thesis is structured as follows. Chapter 2 introduces fundamental knowledge

about column-oriented database systems and discusses important features and concepts applied in column-stores. Furthermore, the MonetDB column-store and its internal MAL code are visited. Chapter 3 covers fundamental knowledge about concepts in the area of WCOJs and discusses the `Generic-Join` algorithm 3.1 that will be integrated into MonetDB. Moreover, Chapter 4 is having a look at related work and outlines the structure of MonetDB's query compiler. Additionally, the translation of the `Generic-Join` algorithm 3.1 into a MAL program is discussed in detail. Finally, the integration of this class of WCOJ algorithms in order to compile queries into the translated MAL code is pointed out. Chapter 5 aims for discussing the results obtained from running benchmarks on the respective systems resulting from the integration process. Moreover, the results are compared to a baseline which is given by the original MonetDB system.

CHAPTER 2

Column-oriented Database Systems

This chapter aims for giving an introduction into column-oriented database systems, which are also referred to as column-stores. This includes a comparison of column-oriented database systems to more traditional row-oriented database systems and a discussion of typical architectures. Moreover, this chapter explores different advanced features of column-stores that cause important performance benefits. Besides giving an overview of column-oriented database systems in general, MonetDB [Monb], a specific column-store implementation, is discussed in more detail. Since MonetDB is a very well known column-store in academia, the integration of worst-case optimal joins, which is elucidated in Chapter 4, has been carried out using this specific system.

2.1 Row-oriented DBMS vs. Column-oriented DBMS

Column-oriented as well as row-oriented database systems are two different architectural approaches of a database system with the purpose to store, manipulate, maintain and access various kinds of data. In general, both concepts aim for serving similar purposes, but differ in system design and technical implementation of database related features. The column-store architecture has been designed from scratch to enhance performance of analytical systems. The main fundamental difference to a row-oriented database system is the storage model. Traditional row-oriented systems store data one row at a time as shown in Table 2.1. Thus, when querying the table there is only a single data object that holds the table's rows which means that it also stores/loads attributes that are not needed to serve a given query. In contrast to that, column-stores vertically partition a database and store the columns separately instead of row-wise tuples as shown in Table 2.2 and 2.3. In a column-oriented database system, each attribute is stored as an

Name	Gender	Salary	Job
Andi	m	1510	Caretaker
Joana	f	2000	Software Engineer
Sally	f	1770	Farmer
Markus	m	1920	Teacher

Table 2.1: Physical storage layout of row-oriented DBMS.

Name
Andi
Joana
Sally
Markus

Gender
m
f
f
m

Salary
1510
2000
1770
1920

Job
Caretaker
Software Engineer
Farmer
Teacher

Table 2.2: Physical storage layout of column-oriented DBMS with virtual ids.

independent data object in order to prevent the system from loading an unnecessary amount of data. [ABH+13]

Facing a column-store with columns as shown in Table 2.2, a query to obtain the average salary of females would only need to load the columns *Gender* and *Salary* from storage to answer the query. A traditional row-oriented system is just capable of loading whole rows and therefore has to transfer attributes that are not required as well. Since, reading data from storage is an expensive operation in general and the amount of attributes and data to be processed has been increasing over the past years, this operation has become a performance bottleneck of database systems. Thus, there are queries and workloads for which column-stores tend to be more efficient than row-stores, since they only load the required attributes as independent data blocks from storage. This motivates to briefly look at settings where column-oriented database systems are beneficial and on the other hand where traditional row-stores are more suitable.

Column-stores outperform row-stores when a larger number of records is accessed, since the data transfer time from storage will dominate the overall seek time. However, inspecting the combination of submitted query and physical storage layout more precisely, it can be obtained that row-stores dominate column-stores on queries that just access a few rows with many attributes, because a column-oriented database system has to search for various columns and load all of them, whereas a row-oriented database system can load the required rows at once. [ABH+13]

2.2 Internals and Advanced Features

This section aims for introducing internal concepts that are of importance to an efficient column-oriented database system. Furthermore, advanced features that lead to performance increase of column-stores and which have to be kept in mind when integrating

Name	
1	Andi
2	Joana
3	Sally
4	Markus

Gender	
1	m
2	f
3	f
4	m

Salary	
1	1510
2	2000
3	1770
4	1920

Job	
1	Caretaker
2	Software Engineer
3	Farmer
4	Teacher

Table 2.3: Physical storage layout of column-oriented DBMS with explicit ids.

worst case optimal joins into a column-oriented database system are discussed. Abadi et al. [ABH+13] elaborate on efforts to introduce some of these techniques to traditional row-oriented database systems. The researchers conclude that these ideas can just be fully beneficial for column-stores due to their completely different design.

2.2.1 Explicit IDs vs. Virtual IDs

A relation in a column-oriented database system is represented by n columns and each column has tuple identifiers in order to obtain which entries belong to the same record across all columns of a relation. Such identifiers can be either explicit ids as shown in Table 2.3 or virtual/implicit ones as shown in Table 2.2. Explicit ids are represented through an additional id column per attribute that increases the data size on disk and will therefore lead to additional effort when reading data from storage. To overcome this issue, modern column-oriented database systems use the tuple's position within the column as a virtual id. To achieve this, some column-stores like MonetDB store attributes as dense arrays and elements belonging to the same record are located at the same index in each of a relation's attributes. [IGN+12]

2.2.2 Late Materialization

As discussed in the previous section, column-oriented database systems mainly differ from row-oriented database systems in their physical storage layout. Nevertheless, they are similar at the view and logical level and therefore column-stores need to present a query's output through data tuples too. To this end, column-oriented database system apply a materialization strategy to combine the separately stored columns into data tuples to determine intermediate results. Moreover, a column-store requires some kind of materialization strategy to serve a query when more than one attribute of a relation is accessed in order to present the output as data tuples.

The main variation between materialization strategies is the point within a query plan when columns are combined into data tuples. This tuple reconstruction takes advantage of explicit or virtual tuple identifier, introduced in Section 2.2.1, to obtain matching positions within multiple attributes and stitch together the intermediate data tuples. Furthermore, column-stores mainly apply either an early materialization or a late materialization strategy. Early materialization combines columns into data tuples when

an intermediate representation is required or even before, whereas late materialization delays tuple reconstruction as long as possible.

The late materialization strategy has two main performance advantages. First, values within columns are stored contiguously in memory using appropriate data structures. For instance, MonetDB [IGN+12] stores a single column as array in the C programming language. This empowers the system to apply compression algorithms such as Run Length Encoding (RLE) on columns in order to shrink the required memory footprint. Moreover, late materialization enables column-stores to only construct relevant tuples. Another advantage is that such column-oriented data structures can be looped through faster than using tuple iterators. That way entire cache lines are filled with values of the same column which maximizes efficiency, because only relevant data is kept in cache. This is important due to the fact that the bandwidth between CPU and memory has become a bottleneck in modern systems [ABH+13]. Additionally, column-stores can take advantage of vectorized processing of columns which is enabled by modern CPUs. [AMDM07] Figure 2.1 depicts a simple example MAL code of a query with selection, projections and a join operation and graphically illustrates how materialization using position lists works. Furthermore, it shows how the output tuples are constructed by materializing the corresponding columns. A detailed discussion of MAL code snippets is given in Section 2.3.5.

2.2.3 Joins

Joins are crucial operations to any database system, because they are the most expensive ones in query processing. Therefore, join operations yield a great potential for performance improvements and have to be dealt with appropriately to provide efficient query processing. Join performance in column-oriented database systems is tightly coupled to the applied materialization strategy. A join operation in a column-store with early materialization works similar to one in a row-store, because using early materialization, tuples are stitched together and then passed to the join operator. This yields similar performance as a join operation in a row-oriented database system.

To achieve the expected higher performance in a column-oriented database system are making use of a late materialization strategy, the structure of the resulting position lists needs to be considered. In general, any join algorithm in column-stores outputs a set of position pairs representing positions in the corresponding input relations for which the join predicate matches. Figure 2.1 shows how joins and materialization are handled on behalf of a simple example within a column-store system. The query is given through a set of MAL instructions. The resulting left position list will be sorted and the right one will be unsorted for many join algorithms, but there are also algorithms that output two unsorted position lists. Since, many join algorithms are implemented to iterate over the left input relation of the join in order, the left output `joinResultRb` will be sorted. The right result `joinResultSb` will be unsorted, because it results from matches between the left and right input relation of the join. Such unsorted position list can lead to performance bottlenecks whenever other columns of the joined relation are

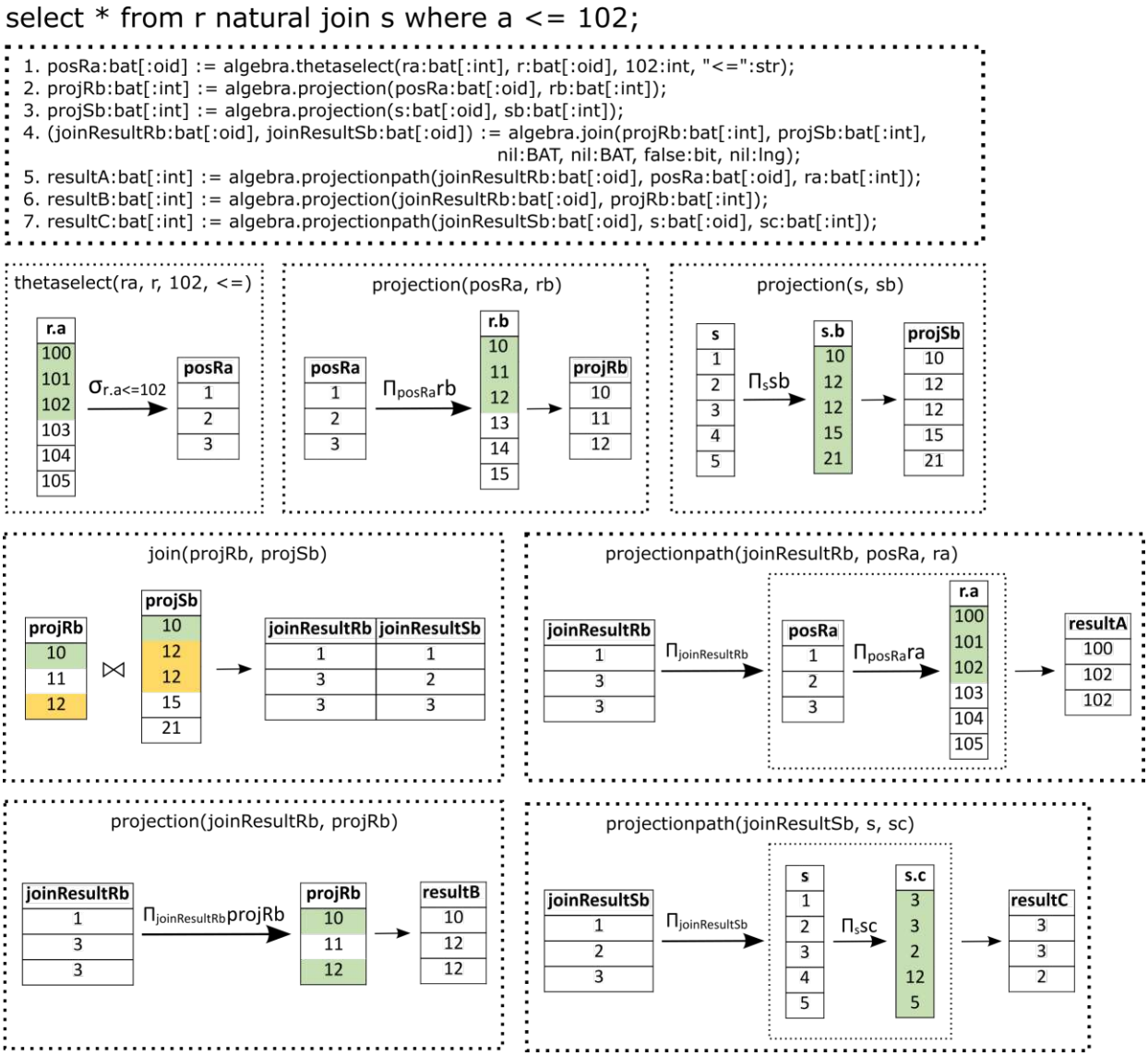


Figure 2.1: Example of materialization and join processing based on sample MAL code from MonetDB with relations R(a,b) and S(b,c) modified from [ABH+13].

required right after the join. To materialize a column using an unsorted list of positions, the storage needs to be accessed randomly instead of sequentially as it would be done with a sorted position list. The shortcoming here is that most storage devices suffer from larger random access time which can significantly slow down the database system.

To overcome this storage access bottleneck, some column-stores use a hybrid materialization strategy. In difference to late materialization, the hybrid strategy materializes all relevant columns of the right relation instead of just the ones that make up the join

predicate. Relevant columns can be for example the ones that need to be materialized after the join as well as the predicate column. The left relation just passes the join predicate column to the join operator. This operation results in a set of tuples for the right relation and a position list for the left one which can be utilized in order to materialize other relevant columns of the left relation. Using hybrid materialization, tuples need not be constructed from unsorted position lists. [ABH+13]

2.2.4 Vectorized Processing

Database textbooks mainly mention two strategies for query execution which are tuple-at-a-time pipelining (Volcano-Style iterator model) and materialization. In systems that apply tuple-at-a-time pipelining, a single tuple is propagated through the query plan tree at a time. This has the advantage of minimizing intermediate results. Whereas in systems that apply the materialization strategy, query operators fully read inputs from storage and write results to it. Thus, every query operator fully reads its entire input and generates an intermediate result from it. This materialization strategy aims for making operators and their interaction simple as well as more CPU efficient, since operators are applied to the input at once and the full result is written to storage and consumed by another operator. Nevertheless, this efficiency is gained at the cost of potentially huge intermediate results which can be an issue for data-intensive applications. MonetDB implements such materialization strategy through its Binary Association Table (BAT) algebra, which is discussed in Section 2.3. [ABH+13]

Another query execution strategy that has been introduced in the field of column-oriented database systems is vectorized execution. Vectorized execution bridges the gap between tuple-at-a-time pipelining and materialization. Instead of propagating a single tuple through the query plan tree as in tuple-at-a-time pipelining, a vector of N tuples is handled at once. At data processing level, operators use primitive functions that are capable of processing data in such vector-at-a-time manner. Processing a vector of size N at once instead of the full input avoids the materialization of large intermediate results. Moreover, each operator processes N tuples at the same time which enables this strategy to benefit from operators looping over arrays which can benefit from compiler optimizations.[ABH+13]

Integrating vectorized processing into column-stores introduces various advantages to the database system. Choosing the vector size N wisely can increase cache locality, since vectors required for answering a query will fit into the CPU cache and therefore decrease the number of additional memory access. Furthermore, vectorized primitive functions are realized as tight loops over arrays for data processing and are therefore target of very efficient compiler optimizations like the generation of SIMD instructions. Abadi et al. [ABH+13] discuss more interesting advantages of vectorized execution in column-stores as well as the application of vectorized processing in row-store systems. [ABH+13]

2.2.5 Data Compression

Abadi et al. [AMF06] mention the application of data compression schemes as a big advantage of column-stores. Column-oriented database systems are capable of dealing with a single attribute/column at a time. This enables the application of different compression schemes in a quite natural way, whereas row-oriented database systems would have to extract attributes from multiple tuples which makes compression more complicated. Column-oriented database systems natively support the compression of attributes across multiple tuples. Moreover, column-stores generally achieve higher compression ratios than row-stores, because consecutive attribute elements are often similar to each other whereas neighboring attributes in tuples of row-oriented systems are not. These two properties encourage the applicability of various compression algorithms. One well-known example is Run Length Encoding (RLE) where repeating values are expressed by pairs of value and run-length. A discussion of various compression algorithms has been done by Abadi et al. in [ABH+13] and [AMF06]. Apart from that, column-stores can take advantage of modern CPU's super-scalar properties by applying vectorized code for decompression to increase the system's efficiency. Additionally, CPU overhead is reduced, since iterating over a page of columns tends to be more efficient than iterating over a page of tuples. Another quite common way to improve CPU performance is to make database operators work directly on compressed data, which makes decompression obsolete for some operators. [AMF06]

A query execution engine capable of operating on compressed data can improve performance by more than an order of magnitude and therefore compression-aware systems' performance benefits overwhelm other techniques that realise factor-wise improvements. This performance gain originates from a combination of decreased memory I/O (since the system has to read less data) and the fact that systems working on compressed data get rid of the need for decompression. Operators that directly work with compressed data need to be implemented with overall common properties of compression algorithms in mind in order to support various compression schemes. As an example of operating on compressed data, think of numeric data that has been compressed using RLE. Summing up, RLE can be replaced with a product of the specific value and the corresponding run-length which reduces a certain number of additions to just a single multiplication. This example shows that the execution engine has to be aware of the applied compression scheme to perform the most suitable operation. [ABH+13]

To sum up, compression of attributes in column-stores as well as operating on compressed data can lead to significant performance improvements. Implementing compression the right way in column-oriented database systems is key to leverage the potential performance benefits. [ABH+13]

2.2.6 Database Cracking

Database systems maintain non-discriminative indices that serve tuple localization purposes. Workload-specific knowledge and additional idle time is required for establishment

and maintenance of such indices. Without workload-specific information, a database system would have to generate every possible index to gain maximal data access response performance which is infeasible. To achieve reasonable data access response times, indices need to be created upfront. Moreover, keeping indices up-to-date comes at additional performance costs as well.To boost performance, Idreos et al. [IKM07] introduce the idea of database cracking, also known as adaptive indexing, in the field of column-stores. The central idea of database cracking is to dynamically build and maintain a discriminative cracker index during processing of a query. This means that each query is processed, the corresponding result set is returned and as side effect columns relevant to the query are cracked into smaller pieces according to the query's predicates. The cracker index assembles all these smaller chunks and just holds data that has already been touched by a previous query. Furthermore, future queries will benefit from knowledge introduced by earlier ones and physically stored data will be self-organized with respect to the query workload. This self-organization property of column-stores together with the fact that a cracker index only composes data that has been touched by queries is significantly beneficial for data access response time. [IKM07]

Read-only queries can obviously benefit from database cracking, however queries that perform updates (insert, update or delete) on columns need to be handled carefully, since cracking is carried out on a copy of the original column. Therefore, it is crucial for the database system to maintain the original as well as the cracker column in sync while performing such update operations. Idreos et al. [IKM07] introduce a way to handle updates within cracking architectures on the example of MonetDB. Due to the fact that the original columns maintain the insertion order, updates are translated into appending any new tuples in general. Whereas MonetDB maintains delta tables that represent pending insertions, deletions and updates. Original columns are updated on transaction commit according to these delta tables. Cracking columns and index are modified by cracking optimizer that merge delta tables into the query plan, which is also supported by corresponding MAL operations.

Idreos et al. [IKM07] integrate a database cracking architecture on top of the MonetDB column-store. If a range query is processed on an attribute for the first time, the involved column is copied and referred to as cracker column of the corresponding attribute. All reorganization (cracking) of an attribute is performed on its cracker column and the original column keeps its insertion state. This allows fast tuple reconstruction using the original column and is therefore used for efficient projection handling, since the column is in insertion order. Cracker columns on the other hand are most suitable for performing fast value selections, because these operations can select whole matching cracked pieces of columns.

Figure 2.2 depicts an example of how cracking of a single column is handled in their implementation. Query Q1 is the first to access column A and therefore a new cracker column is instantiated and partitioned according to the two query predicates. Cracked pieces contain values in the range specified by a predicate. The result of Q1 is then returned as a view of the second piece at no additional costs. The query Q2 benefits from

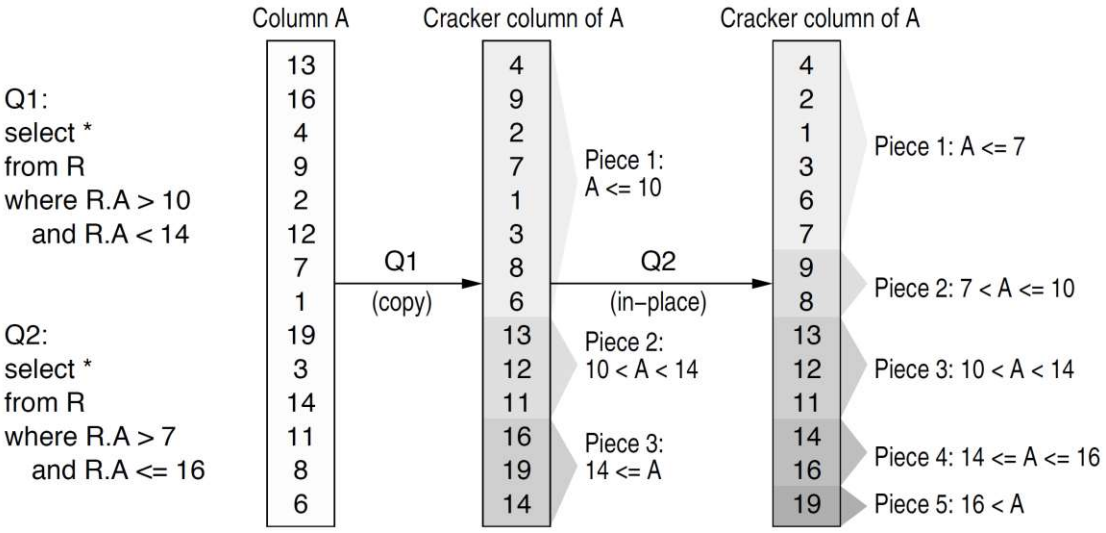


Figure 2.2: Example of cracking a single column using database cracking with MonetDB. [IKM07]

already present knowledge in the cracker index and the system only has to adjust the first and third pieces in-place in order to update the cracker column A. The result of Q2 is a column slice covering pieces 2-4 and again, without additional costs. This underlines the power of database cracking and the importance of knowledge about earlier queries to future ones. [IKM07]

To conclude the section on advanced features and internals of column-oriented database systems, it is important to understand that modern column-stores are composed of way more architectural features than just the column-oriented storage layout. Modern column-stores like MonetDB and C-Store [SAB+05] pioneered many features that have been discussed in this section and combine many of the introduced design principles. These systems proposed a complete redesign of database kernels with decades of DBMS research in mind. [ABH+13]

2.3 MonetDB

The aim of this section is to introduce the column-store system MonetDB [Monb], to discuss its architecture and the concrete implementation of internals mentioned in Section 2.2. More specifically, this section discusses the different building blocks of MonetDB, explores the internal relational algebra representation of a query as well as the MonetDB execution model. MonetDB is an open-source system that is very popular in the academic field and plenty of major research on column-stores has been conducted on that specific column-oriented database system. Due to its popularity and relevance to academia, this work aims for showing the suitability of WCOJ algorithms

for column-stores on the example of MonetDB in Chapter 4 and 5. To that end, the MonetDB system is outlined in more detail in this section.

2.3.1 Introduction

MonetDB is an open-source DBMS that has been developed by the Centrum Wiskunde & Informatica (CWI) database research group in Amsterdam since 1993 [BMK09]. A variety of different architectural features that boost column-store performance have been introduced, while working and researching in the field of column-oriented database systems. Many of the resulting concepts have been integrated into MonetDB. Among others, important research has been conducted on different optimization techniques like optimizing column-stores for modern hardware (CPUs) [MBK02] [BKM08], database cracking / adaptive indexing [IKM07], vectorized execution and lightweight compression [BZN05] to name some.

The amount of research done on column-stores on the example of MonetDB shows the importance of this particular system for academia. Moreover, MonetDB has been around for more than two decades and was designed primarily for data warehouse applications. Therefore, it has to deal with large databases in order to efficiently query a huge amount of data to serve business intelligence and analytical systems. MonetDB's architecture can be divided into three layers as shown in Figure 2.3. Research and innovations on each of the three layers led to significant performance improvements over more traditional row-wise DBMS. Important innovations include vertical fragmentation of storage layout, database cracking, query optimization at runtime and a query execution architecture that is optimized for modern CPUs. [IGN+12]

2.3.2 Physical Storage Layout

MonetDB vertically fragments relational tables and stores each attribute as an independent table with columns <*surrogate, value*>. These separated tables are referred to as BATs. The surrogate column represents the Object Identifier (OID) which is also called the head column. The second column, referred to as tail, stores the actual attribute values. Furthermore, OIDs represent the position of the corresponding value within a column. This is possible, since OIDs follow a dense ascending sequence of numbers. The positional encoding has the positive effect that the surrogate column need not be materialized and that tuples belonging to the base BAT can be reconstructed by the values' position within a column. Table representation of attributes with and without surrogate column is depicted in Table 2.3 and Table 2.2. Positions within base BATs represent the insertion order. The concept of OIDs in column-stores has already been discussed in Section 2.2.1.

Due to the fact, that the OID column need not be materialized, these BATs can be represented as typed C arrays and a collection of them internally represents a relation within MonetDB. Such columns are internally stored as memory mapped files. Additionally, MonetDB applies a late materialization strategy and therefore, intermediate results are materialized at the latest possible time. Ivanova et al. [IKNG10] introduce a recy-

cling architecture which aims for reusing intermediate BATs. The `column-at-a-time` storage layout in combination with late materialization enable bulk processing, which makes use of vectorized and cache optimized operations to minimize query evaluation overhead. [IGN+12]

2.3.3 System Architecture

MonetDB's system architecture is composed of three layers which are frontend, backend and the kernel. A schematic picture of each layer's responsibilities is shown in Figure 2.3. The original MonetDB implementation is a fully-fledged relational DBMS with client interfaces for JDBC, ODBC and Application Programming Interfaces (APIs) for programming languages such as C, Python, Java and some more. This implementation provides a Structured Query Language (SQL) frontend following the SQL:2003 standard. Furthermore, MonetDB's query execution engine is designed to exploit the columnar storage pattern as well as to make use of CPU features and cache systems of modern hardware [BKM08]. In addition to that, the basic MonetDB system provides the possibility to add new data types and algorithms to the system by implementing extension modules, that can be developed in C programming language or MAL. This empowers developers and researchers to implement functionality that goes beyond the traditional SQL approach. [IGN+12] In the following, the three different architectural layers and their responsibilities will be discussed.

Frontend

The frontend layer is specific to the supported query language and the user-specific data model. Figure 2.3 depicts the standard SQL frontend. Nevertheless, there can be compiler frontends for various query languages developed on top of the MonetDB column-store. Boncz et al. [BGvK+06] introduce MonetDB/XQuery which is a database system that fully supports the W3C XQuery language for XML.

The main requirements a frontend has to satisfy are the translation of submitted queries from query language to MAL and to map the user-space data model to BATs. The overall process follows the 3 steps depicted in Figure 2.3. At first, the query is parsed and transformed into some internal relational algebra representation in case of the SQL frontend. The relational algebra representation of the query is further optimized using domain-specific rules which aim for reducing the amount of data that has to be processed. Finally, the optimized representation is compiled into a set of MAL instructions and handed over to MonetDB's backend. Since the backend takes generated MAL code, it is compatible with any frontend generating such. [IGN+12]

Backend

The backend layer is composed of a MAL optimization framework, a MAL interpreter and an interface to communicate with the kernel to access BATs. The optimization framework consists of multiple modules which are arranged into different pipelines. The

pipeline to be used can be configured before system startup. These optimizers target programming language specific optimizations rather than database query specific ones and aim for transforming the MAL program into a more efficient, but equivalent version. Moreover, the optimizer modules need to be shared between different frontends which is enabled by a common binary-relational algebra backend. The pipelining structure enables to chain optimization modules in different orders and to add newly introduced modules into the right spot in the optimization process. [IGN+12]

Kernel

The kernel layer, also referred to as Goblin Database Kernel (GDK), is a C library that provides Atomicity, Consistency, Isolation and Durability (ACID) properties on a Decomposed Storage Model (DSM) [Mon21] and highly optimized implementations of binary relational algebra operators that are utilized to process BATs. These operators are capable of performing operational optimizations which aim for deciding on the actual implementation of an operator or algorithm to be used. This decision is based on the input's properties and due to bulk processing evaluation, each operator accesses the whole input and can obtain important properties for performing operational optimizations. For example, a join operator can decide whether to execute a merge-join if attributes are sorted or stick with a simple hash-join instead. [IGN+12]

The GDK makes use of main-memory database algorithms that are built on virtual-memory Operating System (OS) primitives and multi-threaded parallelism. It provides various facilities, like GDK routines for session management, BAT routines to define primitive operators and BAT Buffer Pool (BBP) routines to manage BBPs. Moreover, the kernel provides routines to manipulate primitive types, utilize the heap and access inserted as well as deleted elements within a transaction. The relational model is mapped to BATs (vertical fragmentation) in order to achieve data independence and better performance by utilizing techniques such as operator-at-a-time processing for instance. [Mon21]

Additionally, a MonetDB backend that acts as an BAT algebra virtual machine is depicted in Figure 2.4. This BAT algebra virtual machine can posses a variety of different frontend modules on top, which support various data models and query languages. Moreover, Figure 2.4 shows a sample application of the BAT algebra operator `select` on a small table that consists of the two columns `name` and `age`. Each attribute consists of virtual, dense surrogates and a memory-mapped values array. The `name` attribute's string values are handled as simple memory-mapped array where each entry relates to the positional offset within the memory-mapped strings. This means that the second string value starts at offset of length of first string entry. Furthermore, the applied `select` operation produces a new dense array of matching positions in the targeted relation. These positions can be used to reconstruct the matching tuples. Moreover, making use of arrays in virtual memory exploits the fast in-hardware address-to-disk-block mapping

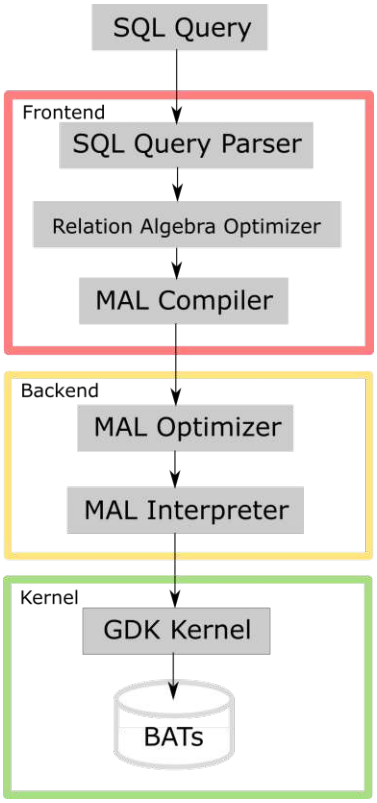


Figure 2.3: MonetDB's three layered architecture with SQL Frontend.

which is implemented in a CPU's Memory Management Unit (MMU). This results in a O(1) positional database lookup mechanism. [BKM08]

2.3.4 Relational Algebra

Since the contribution of this work focuses on SQL as query language, this subsection explores the internal representation of SQL queries using relational algebra. Therefore, Listing 2.1 shows the representation of the triangle query in relational algebra. This query representation can be interpreted as a logical plan on how the query will be processed and represented as `statement tree` internally in the query compiler. Tree nodes represent algebraic operators and edges between them indicate data dependencies between operators. During compilation, this tree structure is recursively traversed and the algebraic operators are translated into the corresponding sequence of MAL instructions. Query compilation is discussed in more detail in Chapter 4. [Mon21]

MonetDB makes use of a bulk query algebra, a simplified form of the traditional relational set algebra, in order to allow faster implementation on modern hardware [BKM08]. Besides the algebraic operators `table`, `join` and `project` used for representing the triangle query in Listing 2.1, there are the basic operators `select`, `group by` and

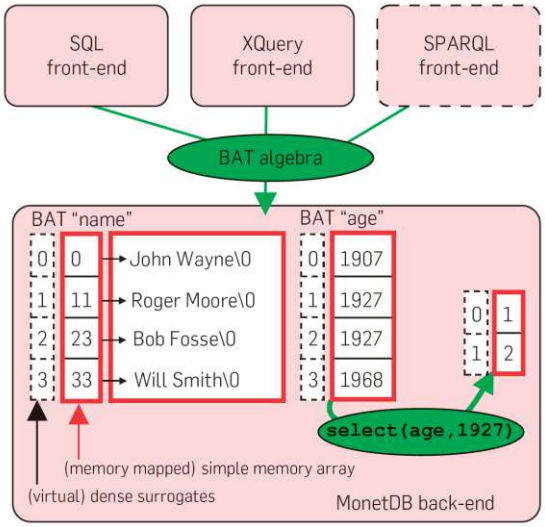


Figure 2.4: MonetDB's backend as BAT algebra virtual machine [BKM08].

`topN`. [Mon21] Listing 2.1 depicts a logical plan which serves the translation of a query into a MAL program. In the following, the main algebraic operators are discussed.

table The `table` operator aims for reading columns of a single relation from storage. To that end, the specific table is referenced via its `name` and the `schema` it belongs to. Moreover, the square brackets are used to specify the columns to be read.

join The `join` operator is used representative for the family of join operations, since MonetDB can decide at runtime which concrete join implementation to use. The operator takes two relations as well as the attributes to be joined on. `Multi-Way-Joins` are broken down into a sequence of `Two-Way-Joins` which results in nested `join` operators.

project The main purpose of the `project` operator is to apply operations on input columns, specified within squared brackets. This operator can also be used to sort data and to rename columns as well. In Listing 2.1, the `project` operator specifies the columns `r.a`, `r.b` and `s.c` as the ones used for result reconstruction after nested `join` operations.

group by The `group by` operator is used to perform aggregation on multiple columns using a specified aggregation function.

select The `select` operator is applied in order to filter columns based on specified predicates.

topN The `topN` operator specifies to only take the top n tuples from the result.

```
project (
| join (
| | join (
| | | table("sys"."s") [ "s"."b", "s"."c" ] COUNT ,
| | | table("sys"."t") [ "t"."a", "t"."c" ] COUNT
| | ) [ "s"."c" = "t"."c" ],
| | table("sys"."r") [ "r"."a", "r"."b" ] COUNT
| ) [ "r"."b" = "s"."b", "r"."a" = "t"."a" ]
) [ "r"."a", "r"."b", "s"."c" ]
```

Listing 2.1: Relational algebra representation of a query which is actually the triangle query introduced in Section 3.1.2.

2.3.5 Query Execution & MonetDB Assembly Language (MAL)

MonetDB's GDK is programmed in MAL and represents an abstract machine. MAL is built around a closed low-level, two-column relational algebra that operates on BATs. To make MonetDB support n-ary relational algebra plans, such n-ary plans are translated into an equivalent two-column BAT algebra which can then be compiled into MAL programs that also operate on two-columns. A MAL program is a sequence of simple MAL instructions and every BAT algebra operator maps to such an instruction. Furthermore, any MAL instruction is only capable of handling simple expressions and therefore said to have zero degree of freedom, however they can be parametrized if necessary. In order to follow MonetDB's operator-at-a-time evaluation strategy, where each operation is completely processed over the entire input as well as to support evaluation of complex expressions, the system applies so-called bulk processing. To that end, complex operations are converted into a sequence of simple BAT algebra operations which perform simple operations on an entire column and can therefore be transformed into simple MAL instructions. [IGN+12] [BMK09]

Due to this way of handling complex expressions together with the array representation of BATs, each BAT algebra operation can be mapped to simple array operations. Thus, each BAT algebra operator can be implemented using simple array operations as shown in Listing 2.2. Since BAT algebra operators are implemented as tight for-loops, instruction cache misses are reduced and the system benefits from high instruction locality. Furthermore, this approach takes advantage of compiler optimizations as well. [BMK09]

```
BAT algebra operator:
R:bat[:oid,:oid]:=select(B:bat[:oid,:int], V:int)

C–Code:
for (i = j = 0; i < n; i++)
  if (B.tail[i] == V) R.tail[j++] = i;
```

Listing 2.2: Example of C implementation of BAT algebra operator `select` from [BMK09].

MAL programs are specifications of dataflow behaviour or intended computation, but the language was not introduced to be treated as a primary programming language. Moreover, a program's evaluation depends on the execution paradigm in a scenario. MAL programs can be either interpreted as an ordered sequence of assembler instructions or as a representation of a dataflow graph. [Monc]

MonetDB's type system is extensible to serve a wide range of database kernels and application requirements. It supports a set of hardwired scalar types which can be dealt with by the kernel without function call overhead. These types are `bit`, `bte`, `sht`, `int`, `lng`, `hge`, `oid`, `flt`, `dbl` and `str`. Furthermore, a set of user-defined types is already implemented and can be extended using MonetDB's type extension mechanism. MAL is a strongly typed language with exception of the polymorphic type `any_1` which is resolved at runtime. [Monc]

Moreover, MAL instructions are assignments where an expression on the right side of the assignments returns multiple results to the variables on the left side of it. They are organized into MAL modules, also known as namespaces. This organisation in namespaces leads to significant improvements for type resolution as well as performance improvements during the optimization phase. In general, an instruction signature is composed of the MAL module and the instruction name itself. Listing 2.3 depicts an example signature for the `thetaselect` instruction within the `algebra` MAL module. [Monc]

Namespaces that are relevant to this work are `algebra`, `group`, `iterator` and `bat`. The `algebra` module is a set of the most common algebraic BAT manipulation commands and all operators take parameter values without causing side effects, but produce new resulting values. The `bat` module composes commands required to manage BATs, such as creating new ones or attaching values to an existing one for instance. Additionally, the `group` module contains operators to craft and perform statistical operations on groups of BATs. This is very important to serve datamining purposes and among others, the `group` module can be used to retrieve as distinct values of a BAT. Finally, the `iterator` module combines functionalities to break down BATs into smaller pieces and iterate over them. [Monc]

2.3.6 Selection of Important MAL Instructions

Integrating WCOJ algorithms into MonetDB as discussed in Chapter 4 requires the manual translation of the relational algebra algorithm from Algorithm 3.1 into a MAL program in the first place. In a successive step, the MonetDB compiler needs to be modified in a way that it generates MAL code that follows the WCOJ approach of Algorithm 3.1. Thus, it is important to understand the MAL syntax in order to be able to choose the most suitable MAL instructions to translate certain lines of pseudocode into. To that end, the code listings in the remaining section will introduce the most important BAT algebra operators through small code examples.

```
algebra.thetaselect(b:bat[:any_1], s:bat[:oid], val:any_1, op:str)
    :bat[:oid]

X_78:bat[:oid] := algebra.thetaselect(bindRB:bat[:str], tidR:bat[:oid],
    X_71:str, "==":str);
```

Listing 2.3: Signature of a `thetaselect` MAL instruction with example usage.

Listing 2.3 shows the signature of the `thetaselect` instruction and its example usage. The listed version of the `algebra` module's `thetaselect` takes two BATs and two additional parameters. The two BATs are data columns with polymorphic values of type `any_1` and a candidate list. The two later parameters specify the value to be selected within the data column as well as the comparison operator to be applied. In the concrete example, the result is a BAT of OIDs which represent each position within `bindRB` that is equal to the value specified by variable `X_71`.

```
mvc:int := sql.mvc();
tidR:bat[:oid] := sql.tid(mvc:int,"sys":str,"r":str);
bindRB:bat[:str] := sql.bind(mvc:int,"sys":str,"r":str,"b":str,0:int);
projRB:bat[:str] := algebra.projection(tidR:bat[:oid],bindRB:bat[:str]);
```

Listing 2.4: Example of projecting attribute B of relation R.

Listing 2.4 depicts how to access the attribute `B` of relation `R`. The instruction `sql.mvc()` can be used to retrieve the `multi-version catalog context` which is required for correct statement dependencies. This for example means, that an `sql.bind` has to be executed before a `sql.update` in concurrent execution. The instruction `sql.tid(mvc,"sys","r")` creates a BAT that represents the tuple identifiers of table `R` which is associated with schema `sys`. A materialized BAT is returned if tuples have been deleted or the column has been cracked, otherwise tuple identifiers represent a continuous sequence. The instruction `bindRB := sql.bind(mvc,"sys","r","b",0)` binds the cache id of column b's BAT to the MAL variable `bindRB`. Finally, the instruction `algebra.projection(tidR,bindRB)` projects `bindRB` onto `tidR` and the result is a materialized BAT `projRB`. [Mon21]

Note that the `algebra.projection` operation can have duplicate entries, whereas the relational algebra projection π returns a set of distinct values.

```
X_78:bat[:oid] := algebra.thetaselect(bindRB:bat[:str], tidR:bat[:oid],
    X_71:str, "==":str);
X_80:bat[:str] := algebra.projectionpath(X_78:bat[:oid], tidR:bat[:oid],
    bindRB:bat[:str]);
```

Listing 2.5: Example of projecting two position lists at once.

The code example in Listing 2.5 performs two projections on BAT `bindRB` in one go. The benefit of executing `algebra.projectionpath` with two position lists over

individually executing `algebra.projection` for each position list is the fact that `algebra.projectionpath`'s intermediate column representations are very likely to not being materialized. This can lead to mentionable performance improvement. [Mon21]

```
C_81:bat[:oid] := algebra.intersect(projRB:bat[:str], projSB:bat[:str],
    nil:bat[:oid], nil:bat[:oid], false:bit, false:bit, nil:lng);
distB:bat[:str] := algebra.projection(C_81:bat[:oid], projRB:bat[:str]);
```

Listing 2.6: Example of intersecting two BATs and projecting the result.

The code snippet in Listing 2.6 shows how to intersect two BATs. The operator `algebra.intersect` takes two columns and performs a set intersection on them. In the example, the BATs `projRB` and `projSB` are intersected and only the resulting position list for `projRB` is returned. Projecting this position list onto `projRB` finally returns a new BAT `distB` that holds the intersection of attributes `B` of relations `R` and `S`. [Mon21]

```
barrier (h:oid, t:str) := iterator.new(distB:bat[:str]);
    /* some MAL instructions */
    redo (h:oid, t:str) := iterator.next(distB:bat[:str]);
exit (h:oid, t:str);
```

Listing 2.7: Example of iterating through every single value of a BAT.

Listing 2.7 shows how loops can be implemented in MAL code. This code construct is essential to integrate the concept of WCOJ algorithms into MonetDB, because the `Generic-Join` algorithm of Algorithm 3.1 requires the compiled MAL code to iterate through the values of certain columns. Loops can be realized by combining the control flow modifiers `BARRIER`, `EXIT` and `REDO`. The modifiers `BARRIER` and `EXIT` represent a guarded statement block and `REDO` is a conditional flow modifier that continues execution of the loop after the corresponding `BARRIER` statement for a potential next value if `iterator.next` returns a numeric value ≥ 0, a non-empty string or a value different from `nil`. Entering the loop through `BARRIER` follows the same conditions. Loops in MAL can be nested where each level of hierarchy is identified by its primary target variable which is `t` in Listing 2.7. The `EXIT` modifier marks the end of the loop and is reached if `REDO` returns no further valid value for another follow-up iteration. [Monc]

```
(X_65:bat[:oid], C_66:bat[:oid]) := group.groupdone(projB:bat[:str]);
distB:bat[:str] := algebra.projection(C_66:bat[:oid], projB:bat[:str]);
```

Listing 2.8: Example of projecting a BAT containing only distinct values.

The code snippet in Listing 2.8 depicts an example of how to obtain a BAT `distB` containing all the distinct values of BAT `projB`. The module `group` provides different grouping operations and the example makes use of the operation `groupdone` in

order to obtain a BAT of positions of unique values within `projB`. This BAT is represented by the second assigned variable `C_66`. Finally, the resulting position list can be used to project the distinct values of `projB` to the new BAT `distB` by executing `algebra.projection`. [Mon21]

```
X_110:bat[:str] := algebra.project(C_66:bat[:oid], t:str);
```

Listing 2.9: Example of projecting a single string value to a position BAT.

Listing 2.9 shows a special case of the more general `algebra.projection` operation which projects values of a BAT, specified by a position list, to a new one. In difference to that, `algebra.project` takes a BAT of positions and a single scalar value and creates a new BAT of the specified position list's shape and projects the scalar value to each position. [Mon21] This operation will be essential to the reconstruction process of resulting BATs when integrating the `Generic-Join` algorithm 3.1 into the MonetDB system.

```
tmpC:bat[:str] := bat.new(nil:str);
tmpC:bat[:str] := bat.append(tmpC:bat[:str], X_110:bat[:str], true:bit);
```

Listing 2.10: Example of creating string BAT and append another column to it.

Another relevant concept in MAL is the creation of a new BAT. Listing 2.10 shows how partial results can be combined into a final result by making use of MonetDB's `bat` module. A new typed BAT can be created by calling the `bat.new` operator and specifying its type via a parameter. To append one BAT to another one, the `bat` module provides an `append` operator. It is important to note, that any result is assigned to a new BAT in MAL. Since this behaviour is not compatible with looping over a column's values and appending partial results to the same BAT in each iteration, a new `append` operator that assigns the result to the same variable has to be introduced. The variable `tmpC` represents a BAT that combines its original values and the appended values from `X_110`. [Mon21]

2.3.7 Optimization

MonetDB's optimization framework covers multiple tiers and is designed modularly in order to be extendible with domain specific optimizer rules. More specifically, there are two different points in the query execution process where optimization takes place.

The first point of optimization is targeting the relational algebra representation of a query. This first group of optimizations are `domain-specific strategic optimizers` which aim for reducing the amount of data that has to be processed while executing the query. With respect to the relational algebra representation of a SQL query, such optimizations could be exploiting indices for efficient join processing or pushing

down selections in the execution order. These optimizations are applied in the MonetDB's frontend layer and result in an optimized logical query plan.

The optimized logical plan is then translated into a MAL program on which the second optimization part is applied. This second optimizations are `tactical optimizations` that are performed on the compiled MAL code itself and merely focus on programming language specific improvements. The MAL code optimization modules are arranged in pipelines that are composed of a sequence of multiple modules. This sequence can be rearranged and extended to further advance the MAL specific optimizations. The MonetDB documentation on MAL optimizers [Mona] discusses the different optimization strategies realized in the system. [IGN+12]

CHAPTER 3

Worst Case Optimal Joins

The aim of this chapter is to introduce the overall concept of WCOJ algorithms as well as the theoretical research on (tight) bounds on the output size of a join from which these algorithms have been derived. This chapter starts of with introducing how joins are processed in traditional DBMS in order to underline the fundamental difference in comparison to WCOJs. Furthermore, the general concept of WCOJs is discussed and advances in research on bounding the worst-case output size of a query are stated. Additionally, this chapter investigates how algorithms have been crafted that follow these proven bounds. Finally, this chapter discusses an algorithm from Ngo et al. [NRR13] that unifies a great amount of research on WCOJs and which is the fundamental basis for the contribution of this work. This algorithm is depicted in Algorithm 3.1 and its integration into column-oriented database systems will be outlined in Chapter 4.

3.1 Important Notation

This section introduces important notation with respect to join processing and WCOJ algorithms. The following definitions of queries and query representations will be used in the following sections and chapters of this work.

3.1.1 Hypergraph

Any natural join query $\mathcal{Q}$ can be modelled using a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, where $\mathcal{V}$ denotes the set of vertices and $\mathcal{E}$ the set of hyperedges $\mathcal{E} \subseteq 2^{\mathcal{V}} \backslash \{\emptyset\}$. The set $\mathcal{V}$ of vertices corresponds to the attributes and the set $\mathcal{E}$ of hyperedges represents the relations R_F involved in the join. For each hyperedge $F \in \mathcal{E}$ there exists a relation R_F on attribute set F. [NRR13] An example hypergraph representation of the triangle query $\mathcal{Q}_\triangle$ is depicted in Figure 3.1.

3.1.2 Triangle Query

The triangle query $\mathcal{Q}_\triangle$ is the simplest cyclic query one can construct from naturally joining three relations with two attributes each. Due to its simplicity, the triangle query is most frequently considered when explaining ideas on WCOJ algorithms in research [NRR13][Vel14][Ngo18][NPRR12]. The query below shows the triangle query with relations $R(A, B)$, $S(B, C)$ and $T(A, C)$ and attributes A, B, and C.

$$\mathcal{Q}_\triangle = R(A, B) \bowtie S(B, C) \bowtie T(A, C)$$

The triangle query $\mathcal{Q}_\triangle$ will be used for demonstration and explanation purposes throughout this work. The triangle query $\mathcal{Q}_\triangle$ can be represented as hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{A, B, C\}$ and $\mathcal{E} = \{R, S, T\}$. Thus, each relation like $R(A, B)$ will be interpreted as a hyperedge $\{A, B\}$ on the argument list's variables.

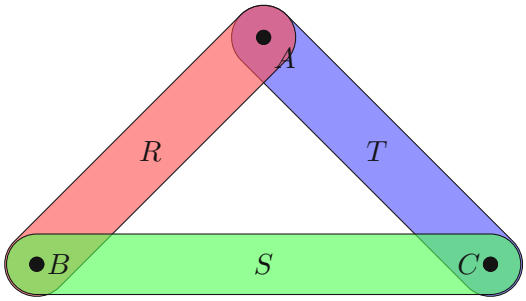


Figure 3.1: Hypergraph representation of the triangle query $\mathcal{Q}_\triangle$.

3.1.3 Fractional Edge Cover

The term `fractional edge cover` is important in the area of WCOJ, because major proposed bounds in this research field use this concept for their formulations and proofs. Ngo et al. [NRR13] define a `fractional edge cover` using a polyhedron as follows.

Given a hypergraph $H = (\mathcal{V}, \mathcal{E})$ and let $\boldsymbol{x} = (\chi_F)_{F \in \mathcal{E}}$ be any point in the following polyhedron:

$$\Big\{\boldsymbol{x} | \sum_{F: v \in F} \chi_F \geq 1, \forall v \in \mathcal{V}, \boldsymbol{x} \geq \mathbf{0}\Big\}$$

Every point $\boldsymbol{x}$ following this definition is referred to as a `fractional edge cover` of the hypergraph H.

Grohe [Gro13] gives a constructive textbook definition of the terms `fractional edge cover` and `fractional edge cover number` as well as how the query's structure can be used to obtain non-trivial bounds on the result's size. This definition is given as follows.

Let D be a database instance with relations $R_1, ..., R_m$ and $\mathcal{Q}$ be a natural join query, then the idea is to bound the size of the output of the query $\mathcal{Q}(D)$ in terms of

the input relations' sizes $N_i = |R_i(D)|$. If a single relation R_i contains all attributes relevant to the query $\mathcal{Q}$ then the output size is bounded by $|\mathcal{Q}(D)| \leq N_i$. Instead of one relation containing all attributes, imagine a set of k relations $R_{i_1}, ..., R_{i_k}$ are containing all attributes relevant to query $\mathcal{Q}$, then the output size is bounded by $|\mathcal{Q}(D)| \leq \prod_{j=1}^{k} N_{i_j}$. The relations $R_{i_1}, ..., R_{i_k}$ are referred to as an `edge cover` of query $\mathcal{Q}$.

To find the `edge cover` yielding the best bound, Grohe [Gro13] describes an integer linear program in variables $x_1, ..., x_m$ where $x_i = 1$ implies that relation R_i is part of the `edge cover`. This integer linear program for a query $\mathcal{Q}$ taking attributes $A_1, ..., A_n$ is defined as the following minimization problem.

$$\begin{aligned} &min \sum_i x_i * \log(N_i) \\ where &\sum_{i:A_j of R_i} x_i \geq 1, \forall j \in \{1, ..., n\} \\ &x_i \in \{0, 1\} : \forall i \in \{1, ..., m\} \end{aligned} \tag{3.1}$$

A solution to this integer linear program is a vector $\boldsymbol{x} = (x_1, ..., x_m) \in \{0, 1\}^m$ and for every such solution the following bounds holds.

$$|\mathcal{Q}(D)| \leq \prod_{i=1}^{m} N_i^{x_i} = 2^{\sum_i x_i * \log(N_i)}$$

The `edge cover number` of query $\mathcal{Q}$ in database D is the optimal solution of the integer linear program above, which is formalized as $\rho(\mathcal{Q}, D) = \sum_i x_i * \log(N_i)$. By replacing the integrability constraint $x_i \in \{0, 1\}$ with $0 \leq x_i$, the solution vector of the program becomes rational and such a rational solution $\boldsymbol{x} = (x_1, ..., x_m) \in \mathbb{Q}^m$ is called a `fractional edge cover` of query $\mathcal{Q}$. Likewise, the value of an optimal solution $\rho^*(\mathcal{Q}, D) = \sum_i x_i * \log(N_i)$ is referred to as `fractional edge cover number` of the query $\mathcal{Q}$. Since, the definitions above are based on the hypergraph representation of a query $\mathcal{Q}$ it holds that $\rho(\mathcal{Q}, D) = \rho(H(\mathcal{Q}), D)$ and $\rho^*(\mathcal{Q}, D) = \rho^*(H(\mathcal{Q}), D)$ [AGM08]. For further examples of `fractional edge covers` of different queries see Ngo et al. [NRR13].

3.2 Traditional Join Processing

The evaluation of relational database queries is a well-studied area in the database theory and systems fields. Especially, a great amount of research has been conducted on the evaluation of relational joins as well as various query optimization techniques over the past decades. Chapter 2 discusses many different techniques applied in column-store systems which contributed to the emerge of column-oriented database system.

Two important factors that drove research on query evaluation were the improvements on computing hardware and the drastic increase of data volumes to be store and processed

by DBMSs. Additionally, increasing complexity in relational data has led to the need for more sophisticated query evaluation strategies and algorithms. Since, relational join evaluation is known to be the most computational intensive part in the query processing pipeline, various optimization techniques and algorithms have been proposed in research and are applied by traditional DBMSs that can be found in everyday systems. Graefe [Gra93] already published a survey in 1993 which discusses many join algorithms that are still core components of up-to-day systems. These algorithms include `Nested Loop Joins`, `Hash Joins` and `Merge Joins` which are elaborated in the following.

3.2.1 Nested Loop Joins

A `Nested Loop Join` algorithm is the simplest join algorithm for binary matching. Computing the `Nested Loop Join` for the simple natural join query $R \bowtie S$, the algorithm iterates through all elements of the outer relation R and scans the inner input relation S for matching elements. The algorithm's name comes from its structure, because the matching is implemented as two nested loops, one iterating the outer and the other the inner relation. Advantages of this algorithm are its simplicity and that it can compute the cartesian product and any Theta-join of two relations. The big drawback of `Nested Loop Joins` is their poor performance, because for every element of the outer relation the whole inner relation is scanned for matches. Assuming that both relations are large with input size n, then the runtime of a simple `Nested Loop Join` is in $O(n^2)$. Due to this poor performance, this algorithm is commonly not applied in database systems, but often used for demonstration purposes.

Due to the algorithm's simplicity there is a lot of room for improvements. For single match operations, like intersections, iterating the inner relation can be terminated after finding the first match for a given element of the outer relation. Moreover, a `Block Nested Loop Join` algorithm increases performance by scanning the inner relation for each page of the outer one. This decreases the number of scans on the inner relation. Furthermore, scans of the inner input can be slightly boosted by scanning the inner relation alternatingly forward and backward. This approach would reuse the last page of the previous scan and would save read per inner scan. [Gra93]

3.2.2 Merge Joins

The `Merge Join` or `Sort Merge Join` algorithm joins two relations R and S and requires that both relations are sorted on the join attribute. Sorting the join columns groups all tuples by the join column's value and therefore partitions the relation. The merging step exploits this partitioning of both relations involved in a join. Thus, both relations are compared partition-wise when merging which avoids scanning relation R multiple times. [RG03]

The sorting of join attributes is the key consideration of this approach. Additional performance can be gained if the `Merge Join` algorithm operates on already sorted inputs, since no additional costs for sorting are required. Furthermore, it is not always

necessary to explicitly sort the join columns, because there might be some intermediate representation of the relation or join column which is already sorted accordingly. If multiple relations are joined on the same columns using the `Merge Join` algorithm, each intermediate result is sorted at the join attribute and successive `Merge Joins` can exploit this property to perform further joins efficiently. [Gra93]

Computing the `Merge Join` for the natural join query $R \bowtie S$ where R is of size m and S of size n, sorting both relations on the join column would be in $O(m * log(m))$ and $O(n * log(n))$ respectively. Merging the two sorted relations is in $O(m + n)$ if at least one of the relations is duplicate free. It is obvious that performing a `Merge Join` is most suitable if at least one input is already sorted. Thus, database systems can decide dynamically which concrete algorithm to be used according to relation's properties and the data flow. [RG03]

3.2.3 Hash Joins

The `Hash Join` algorithm's idea is to build an in-memory hash table for the smaller input relation, referred to as build input, and probes this hash table against the other input relation called the probing input. If the hash table fits in memory, the algorithm works without the necessity for temporary files and additional disk reads. [Gra93]

Similar to the `Merge Join`, the `Hash Join` algorithm aims for partitioning the input relations on the join attribute. This partitioning is achieved by hashing both relations on the join column using the same hash function. That way, it can be assured that tuples in partition j of one relation can only be joined with tuples in partition j of the other relation. For performing the join, a partition of the smaller relation is read and the corresponding partition of the other relation is scanned for matches. Due to partitioning the relations, the partitions are only scanned once during the join process. [RG03]

To evaluate the `Hash Join` algorithm's costs, the build and probing phase need to be examined separately. During the build phase, both relations have to be scanned and partitioned. Assuming the natural join query $R \bowtie S$ where R is of size m and S of size n this phase can be done in $O(m + n)$. Moreover, under the assumption that loaded partitions will not exceed the memory size during probing and each partition is only scanned once, the probing phase is also in $O(m + n)$ and so is the overall `Hash Join` algorithm. [RG03]

3.2.4 Joining Multiple Relations

The previously discussed join algorithms can be used to combine two relations at a time. Joining two relations is referred to as `Two-Way-Join`. However, conjunctive queries in general require to combine more than two relations during query evaluation. Ramakrishnan and Gerke [RG03] describe that the textbook way of evaluating join queries across multiple relations is to obtain the best pairwise join plan from the set of all possible join plans. Possible join plans for the triangle query $\mathcal{Q}_\triangle$ are shown in

Figure 3.2. Any of the three join plans in Figure 3.2 runs in time $\Omega(N^2)$, since any of the joins $R \bowtie S$, $R \bowtie T$ and $S \bowtie T$ produce quadratic effort. The third relation is joined using a semi-join, because all attributes are already included in the former join and therefore joining the third relation produces linear effort. Joining relations in the most suitable order leads to stronger bounds on the output size, thus to more efficient join evaluation. To take advantage of the best possible join order, query optimizer and cost functions are used to rate join plans. Joining multiple relations is further referred to as `Multi-Way-Join`.

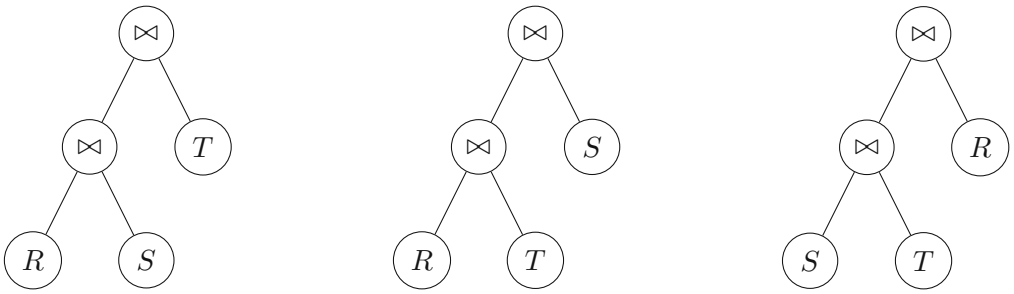


Figure 3.2: The three possible pairwise join plans for the triangle query $\mathcal{Q}_{\triangle}$.

3.2.5 Major Join Processing Approaches

Ngo et al. [NRR13] discuss a query's structural information as well as making use of cardinality information as the two major approaches for join processing. These two ways have been used independently and could not be united into a more powerful approach so far. Grohe and Marx [GM14] as well as Atserias, Grohe and Marx [AGM08] introduced tight bounds on the output size of a query as a function of the sizes of input relations. These bounds are further referred to as AGM bound and combine a query's structural information with the cardinality information. The AGM bound is discussed in more detail in Section 3.3. Moreover, Ngo [Ngo18] surveys and discusses research on more general constraints which can be of great interest to DBMSs. These are `functional dependencies` and `degree constraints`.

Structural Information The term `structural information` covers a query's structural properties such as bounded width or acyclicity. Such properties are very important to theoretical, algorithmic concepts that have been applied in database systems. For example, Yannakakis [Yan81] describes different algorithms for various problems on acyclic queries. A lot of research on `structural information` corresponds to various definitions of width and describes how far a query is away from being acyclic. A query is tractable, which means that there exists a polynomial time algorithm that evaluates the query, if the used notion of width is bounded by a constant. The runtime of structural approaches is $O(N^{w+1} * log(N))$ where w describes the notion of width and N the summarized input size. [NRR13]

Cardinality Information The term `cardinality information` covers a query's quantitative aspects. Structural approaches summarize the sizes of the query's input relations in a single number instead of taking various potentially different input sizes into account. Cardinality information is of great importance, because commercial DBMSs mainly focus on cardinality aspects of a query. `Cardinality information` is important for obtaining the most suitable join plan when processing a `Multi-Way-Join`, since the individual sizes of relations need to be considered. Furthermore, without putting enough focus on `structural information` as well, any join-project plan will be slower by a polynomial in the data size than the best possible one. [NRR13] `Cardinality information` or `cardinality constraint` can be formalized as an assertion $|R_F| \leq N_F$ for any $F \in \mathcal{E}$, which is roughly speaking a constraint on the input size of a certain relation [Ngo18].

Functional Dependencies Functional dependencies, also referred to as FD constraints, are very widespread in DBMSs. Simple functional dependencies are of the form $A \to B$ where A and B are variables. In general, the meaning of FD constraints is defined over relations between two sets of variables which are X and Y. Such FD constraints state that, if bindings in the set X of variables are fixed, there is at most one binding for every such variable in the other set Y. Since functional dependencies are more general than cardinality constraints, the AGM bound does not cover them. To handle FD constraints, Gottlob et al. [GLVV12] extend the AGM bound and showed that their bound is tight if all functional dependencies are simple ones. [Ngo18]

Degree Constraints Degree constraints describe a class of constraints that are more general than FD constraints. Such constraints ensure that for any fixed binding of variables in set X, there is at most a certain number of bindings of variables in set Y. Therefore, degree constraints are a generalization of functional dependencies and cardinality constraints, since cardinality constraints are degree constraints with an empty set X. [Ngo18]

To obtain the best possible join-project plan, it is necessary to combine `structural information` with `cardinality information` of a query which is done by the AGM bound. The theoretical research of Ngo et al. [NRR13], which proposes a class of WCOJ algorithms that is implemented in this work, takes `cardinality constraints` into account. The more general FD and degree constraints have been introduced briefly, since (future) research will aim for finding appropriate algorithms covering them.

3.2.6 Limitation of Traditional Join Plans

Traditional database systems try to obtain the best pairwise join-project plan when dealing with `Multi-Way-Joins`. Figure 3.2 depicts the three possible join-project plans for the triangle query $\mathcal{Q}_\triangle$, which are:

1. $(R \bowtie S) \bowtie T$
2. $(R \bowtie T) \bowtie S$
3. $(S \bowtie T) \bowtie R$

Ngo et al. [NRR13] point out the suboptimality of such traditional join plans by providing a family of instances for which any of the traditional join plans from Figure 3.2 produces a too large intermediate result and therefore runs in $\Omega(n^2)$. The general family of instances for $m \geq 1$ is defined as follows:

$$R = \{a_0\} \times \{b_0, ..., b_m\} \cup \{a_0, ..., a_m\} \times \{b_0\}$$
$$S = \{b_0\} \times \{c_0, ..., c_m\} \cup \{b_0, ..., b_m\} \times \{c_0\}$$
$$T = \{a_0\} \times \{c_0, ..., c_m\} \cup \{a_0, ..., a_m\} \times \{c_0\}$$

Figure 3.3 shows a graphical representation of such an instance with $m = 4$. Any relation R, S and T of the described family of instances is of size $N = 2 * m + 1$ and the output size of the query is $|\mathcal{Q}_\triangle| = 3 * m + 1$. In contrast to that, any pairwise join has a size of $N = m^2 + m$. Therefore, any join plan from Figure 3.2 has a runtime in $\Omega(n^2)$ for a large m. [NRR13]

The main cause for pairwise join plans to be suboptimal is skewed data. Figure 3.3 clearly shows a heavily skewed example instance towards the white dotted data values which represent a_0, b_0 and c_0 respectively. The remaining chapter aims for introducing bounds and algorithms that provide an optimal way of avoiding and dealing with skew. The provided family of instances is very important to this work, because it is an illustrative example of a situation where WCOJs tremendously outperform traditional ways of join processing. The evaluation described in Chapter 5 revisits this family of instances.

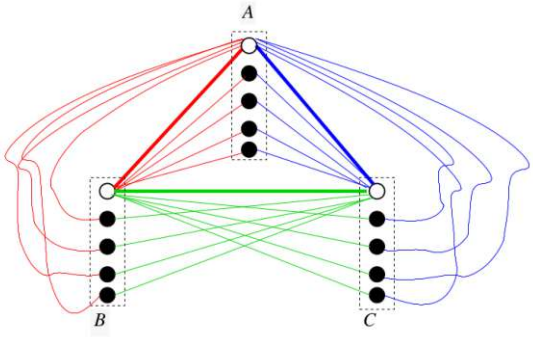


Figure 3.3: Illustration of suboptimal join-project only plans of the triangle query $\mathcal{Q}_\triangle$ with $m = 4$ from Ngo et al. [NRR13].

3.3 Worst-Case Optimal Joins

The term WCOJ algorithms refers to a class of join algorithms whose runtime match the worst-case output size of a query [Ngo18]. The concepts and techniques behind WCOJ algorithms originated from decades of research in different fields including graph theory, information theory, constraint satisfaction, geometric inequalities and database theory. The combination of different research from these areas led to new classes of join algorithms that evaluate `Multi-Way-Joins` in time proportional to the worst-case output size of the query. This query computation has been shown to be asymptotically better than pairwise evaluation [NRR13] [NPRR12] [Vel14]. In order to illustrate the power of WCOJ algorithms, consider the triangle query $\mathcal{Q}_\triangle$. Ngo [Ngo18] introduces an entropy argument from which he derives the AGM bound for the triangle query $\mathcal{Q}_\triangle$. Moreover, a runtime bound is presented based on the `fractional edge cover number` which illustrates the connection between WCOJ algorithms and `fractional edge covers`.

The entropy argument of Ngo [Ngo18] for the triangle query $\mathcal{Q}_\triangle$ is defined using a distribution $Domain(A) \times Domain(B) \times Domain(C)$ where triples (a, b, c) are uniformly selected from the query's output. Let H be an entropy function of this distribution, then $H[\boldsymbol{X}]$ describes the entropy of the marginal distribution on variables $\boldsymbol{X} \subseteq \{A, B, C\}$. Based on this definition, Ngo [Ngo18] shows that $H[A, B, C] \leq \log_2 |\mathcal{Q}_\triangle|$, $H[A, B] \leq \log_2 |R|$, $H[B, C] \leq \log_2 |S|$ and $H[A, C] \leq \log_2 |T|$ holds. Furthermore, he shows that if for any coefficients $\alpha, \beta, \gamma \geq 0$

$$H[A, B, C] \leq \alpha * H[A, B] + \beta * H[B, C] + \gamma * H[A, C] \tag{3.2}$$

holds for all entropy functions H, then the following output size bound can be derived for the triangle query $\mathcal{Q}_\triangle$:

$$|\mathcal{Q}_\triangle| \leq |R|^\alpha + |S|^\beta + |T|^\gamma \tag{3.3}$$

Furthermore, Ngo [Ngo18] shows that Equation 3.2 holds $iff \alpha + \beta \leq 1, \beta + \gamma \leq 1$ and $\alpha + \gamma \leq 1$ for $\alpha, \beta, \gamma \geq 0$ hold and presents a special case of the AGM bound for the triangle query $\mathcal{Q}_\triangle$ as follows:

$$\begin{aligned} \log_2 |\mathcal{Q}_\triangle| \leq \min\{&\alpha * \log_2 |R| + \beta * \log_2 |S| + \gamma * \log_2 |T| \\ &s.t. \alpha + \beta \leq 1, \beta + \gamma \leq 1, \alpha + \gamma \leq 1; \alpha, \beta, \gamma \geq 0\} \end{aligned} \tag{3.4}$$

The Inequalities 3.3 and 3.4 clearly show that the bound depends on the `fractional edge cover` of the query. To discuss the runtime of WCOJ algorithms on the example of the triangle query $\mathcal{Q}_\triangle$, consider a database instance D where $N = \max\{|R|, |S|, |T|\}$ and $\rho^*(H(\mathcal{Q}_\triangle, D)) = (\alpha^*, \beta^*, \gamma^*)$ be an optimal solution to Inequality 3.4. Then, the triangle query $\mathcal{Q}_\triangle$ can be evaluated in $\tilde{O}(N + |R|^{\alpha^*} * |S|^{\beta^*} * |T|^{\gamma^*})$ where $\tilde{O}$ hides a potential log factor in N. Thus, the runtime is proportional to the size of the database and an optimal solution of the `fractional edge cover` of the joined relations. Note

that the presented bound only takes cardinality constraints into account. Ngo [Ngo18] discusses the general case where degree constraints apply.

To analyze how the runtime bound is related to the `fractional edge cover` of the query hypergraph, lets consider $(\alpha^*, \beta^*, \gamma^*)$ to either be $(0,1,1)$ or $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$. In case of $(0,1,1)$ the runtime is in $\tilde{O}(|R|^{\alpha^*} * |S|^{\beta^*} * |T|^{\gamma^*}) = \tilde{O}(|S| * |T|)$ modulo preprocessing time $\tilde{O}(N)$, similar to the runtime of the traditional join plan $R \bowtie (S \bowtie T)$. The instance $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ leads to an optimal bound for Inequality 3.4 if the product of sizes of any two relations involved in the triangle query is greater than the size of the third relation. Then, the runtime is bound by $\tilde{O}(N + \sqrt{|R| * |S| * |T|})$ which is the worst-case optimal runtime. [Ngo18]

The following section discusses theoretical research on studying the worst-case output size of a query. Moreover, it introduces first algorithmic concepts that have been derived from formal proofs of such bounds.

3.3.1 Fundamental Research on Bounds and Algorithms

Ngo's survey [Ngo18] gives a great overview on milestones in the history of bounding the worst-case output size of a query and deriving algorithms from the corresponding proofs. It discusses various results from defining bounds on a query's output size and algorithms that emerged from these approaches. The survey introduces various milestones that led to the `Generic-Join` algorithm 3.1 from Ngo et al. [NRR13] and beyond.

The first known approach, related to defining worst-case optimal bounds on a query's output size, is the Loomis-Whitney inequality [LW+49] which is a geometric inequality shown in 1949. In terms of geometry, the Loomis-Whitney inequality bounds the measure of an n-dimensional set in terms of its $(n-1)$-dimensional projections onto the coordinate hyperplanes [NPRR12]. Ngo et al. [NRR13] formulate Loomis-Whitney queries LW_n from this geometric inequality where for every $i \in [n]$ there exists a relation $R_{[n]\setminus i}$. Less formally, in a Loomis-Whitney query every atom contains all but one variable. Furthermore, the Loomis-Whitney query is a generalization of the triangle query since $\mathcal{Q}_\triangle = LW_3$ and therefore, applying the discrete form of Loomis-Whitney inequality states $|\mathcal{Q}_\triangle| = \sqrt{|R| * |S| * |T|}$ which is also discussed by Ngo et al. [NRR13].

In 1981, Alon [Alo81] aimed for determining the maximum number of subgraphs G isomorphic to H, for two graphs G and H. Roughly speaking, the number of occurrences of a subgraph H within a larger graph G has been studied. Thus, Alon [Alo81] researched the worst-case output size in the area of graph theory and obtained a tight asymptotic bound on the number of subgraphs. In case of the triangle query $\mathcal{Q}_\triangle$, this bound has the same asymptotic behaviour as Loomis-Whitney's which is in $\Theta(N^{\frac{3}{2}})$, where the tables sizes are given by $N = |R| = |S| = |T|$. Moreover, Ngo [Ngo18] formulates Alon's bounds using the `factional edge cover number` of hypergraph H as $\Theta(N^{\rho^*(H)})$.

Another milestone has been achieved by Bollobás and Thomason [BT95] in 1995. The researchers proved a generalization of Loomis-Whitney's results from 1949. Ngo [Ngo18]

discusses the connection between the most general bound by Bollobás and Thomason [BT95] and the AGM bound [AGM08]. Moreover, the equivalence between the AGM bound and the geometric inequality from Bollobás and Thomason [BT95] has been shown by Ngo et al. [NPRR12] in 2012. The most important result of Bollobás and Thomason to the area of WCOJ algorithms was their inductive proof using Hölder's inequality from which the recursive algorithms by Ngo et al. in [NPRR12] and [NRR13] have been developed and analyzed.

Furthermore, main contributing research to the establishment of WCOJ algorithms were Grohe and Marx [GM06] [GM14] and Atserias et al. [AGM08] whose bound is known as the AGM bound. Grohe and Marx [GM06] [GM14] introduce a new notion of width, called the `fractional hypertree width` by combining the `hypertree width` with the `fractional edge cover number`. They apply the `fractional hypertree width` in the context of Constraint Satisfaction Problems (CSPs) to define and prove new bounds. They showed that CSPs with `bounded fractional hypertree width` can be solved in polynomial time. Moreover, the researchers proved that the maximum possible number of solutions of a sub-problem defined within a bag of a tree decomposition is bound by $O(N^{\rho^*(\mathcal{Q})})$ using Shearer's lemma and they state an almost worst-case optimal join-project plan which runs in $O(N^{\rho^*(\mathcal{Q})+1})$ [Ngo18].

The AGM bound of Atserias et al. [AGM08] makes use of the `fractional edge cover number` in order to define bounds on the size of conjunctive queries in terms of the size of its input relations. The researchers present a proof of this worst-case optimal bound and also show that this bound is asymptotically tight. Furthermore, they state that join-project plans that result from `fractional edge covers`, and therefore stuck to the AGM bound, are better than join plans. Additionally, they proved that there exist arbitrary large join queries $\mathcal{Q}$ and database instances D such that any join-only plan requires $\Omega(N^{\Omega(\log(|\mathcal{Q}|)})$ to compute $\mathcal{Q}(D)$ whereas a join-project plan evaluates $\mathcal{Q}(D)$ in $O(N^3)$.

The research of Ngo et al. [NPRR12] from 2012 builds on the results of Atserias et al. [AGM08] and aims for obtaining algorithms that follow these worst-case optimal bounds. They present multiple results that are of interest to database theory and systems research. First, the researchers showed that for some queries any join-project plan is polynomially slower than the optimal bound. Moreover, they present an algorithm that achieves asymptotically worst-case optimal running times for any join query. In the first place, Ngo et al. [NPRR12] prove that the AGM inequality, as given in Lemma 3.1, is equivalent to the discrete version of the geometric inequality of Bollobás and Thomason [BT95] and then focused on the Loomis-Whitney inequality which is a special case of the one by Bollobás and Thomason. The researchers developed a worst-case optimal algorithm for join queries that are Loomis-Whitney instances with $n \geq 3$. This class of queries contains the triangle query $\mathcal{Q}_\triangle$ and more general queries. Furthermore, they developed this algorithm to handle any instance of natural join queries. Moreover, their algorithm provides a constructive proof of the AGM bound without using Shearer's inequality. The key idea of the presented algorithm is the partitioning of join key values

on each side of the join into heavy and light ones. A value of a join attribute is said to be heavy if its fan-out is big enough so that joining all such join attributes would violate the size bound. Such violation would be a size above $N^{\frac{3}{2}}$ for example. [NPRR12]

Unfortunately, the algorithm and its analysis provided by Ngo et al. [NPRR12] are very complicated and not easy to implement. Interestingly, engineers at the commercial database system LogicBlox [AtCG+15] noticed an implementation of this complicated class of WCOJ algorithms using their framework. Benchmarks of this approach compared to LogicBlox's work-horse join algorithm called `Leapfrog-Triejoin` (LFTJ) revealed that their `Leapfrog-Triejoin` can achieve the same runtime and even outperforms the algorithm of Ngo et al. [NPRR12] on some test problems. Driven by these benchmark results, Veldhuizen [Vel14] analyzed the complexity of `Leapfrog-Triejoin`, which has already been implemented in 2009, and showed that this algorithm is worst-case optimal up to a logarithmic factor in the data size. Furthermore, Veldhuizen proved that `Leapfrog-Triejoin` is worst-case optimal for finer-grained classes of queries than the algorithm of Ngo et al. [NPRR12] and published these findings in 2014.

The simplicity of Veldhuizen's `Leapfrog-Triejoin` and the results of Ngo et al. [NPRR12] from 2012 lead to the establishment of the `Generic-Join` algorithm of Ngo et al. [NRR13]. The researchers underline how they bridge the gap between structural information of a query and cardinality constraints. Moreover, they analyzed the AGM bound and constructed the `Generic-Join` algorithm, depicted in Algorithm 3.1, from this analysis. This rather simple and natural recursive join procedure is integrated into a column-store system in this work. Therefore, this historical outline collected various interdisciplinary research that led to the establishment of this `Generic-Join` algorithm.

Beyond bridging the gap between structural information and cardinality constraints of a query which this work is built upon, there is research on more general constraints as discussed in Section 3.2.5.

3.3.2 AGM Bound

The major result of Atserias et al. [AGM08] as well as of Grohe and Marx [GM14] [GM06] is a tight bound on a join query's output size in terms of the sizes of the individual input relations and the `fractional edge cover number`. Thus, the AGM bound unifies the structural properties and cardinality constraints of a query into a single tight bound. Atserias et al. [AGM08] define the AGM bound for any join query $\mathcal{Q}$ as given in 3.1. The bound addresses how large $\mathcal{Q}(D)$ can get in terms of $|D|$ in the worst-case.

Lemma 3.1 ([AGM08])**.** *Let $\mathcal{Q}$ be a join query with schema σ and D let be an instance of σ. Then for any `fractional edge cover` $(\chi_R : R \in \sigma)$ of $\mathcal{Q}$ it holds that*

$$|\mathcal{Q}(D)| \leq \prod_{R \in \sigma} |R(D)|^{\chi_R}.$$

Atserias et al. [AGM08] as well as Ngo et al. [NRR13] present proofs of the AGM bound applying Shearer's inequality. It has to be noted that in the proof of the AGM

inequality from Lemma 3.1 the `fractional edge cover` need not necessarily be optimal. Atserias et al. [AGM08] further present a proof that is based on the linear program presented in Equation 3.1 (without taking the sizes of input relations into account) and the duality argument for linear programs. Moreover, they show that the upper bound of the AGM inequality from Lemma 3.1 is tight as given in Lemma 3.2.

Lemma 3.2 ([AGM08]). *Let $\mathcal{Q}$ a join query with schema σ and let $(\chi_R : R \in \sigma)$ be an optimal `fractional edge cover` of $\mathcal{Q}$. Then for any $N_0 \in \mathbb{N}$ there exists an instance D of σ s. t. $|D| \geq N_0$ and*

$$|\mathcal{Q}(D)| \geq \prod_{R \in \sigma} |R(D)|^{\chi_R}.$$

Atserias et al. [AGM08] state that due to the AGM bound and the fact that it is tight (see Lemma 3.1 and Lemma 3.2), for any join query $\mathcal{Q}$ and every database instance D, the query's output size on database instance D is bounded by $|\mathcal{Q}(D)| \leq |D|^{\rho^*(\mathcal{Q})}$. Moreover, there are arbitrarily large database instances D such that $|\mathcal{Q}(D)| \geq |D|^{\rho^*(\mathcal{Q})} * |\mathcal{Q}|^{-1}$ holds. Based on the AGM bound and their results, the researchers claim that a class of join queries having a bounded `fractional edge cover number` is equivalent to the fact that queries from that class can be evaluated in polynomial time by an explicit join-project plan.

In order to profess on the AGM bound, lets consider the following two edge covers for the triangle query $\mathcal{Q}_\triangle$ from Figure 3.1.

1. $\chi_R = \chi_S = \chi_T = \frac{1}{2}$
2. $\chi_R = \chi_S = 1, \chi_T = 0$

Both edge covers are valid because for each it holds that $\sum_{i:A_j of R_i} x_i \geq 1, \forall j \in \{1, ..., n\}$. Having a look at attribute B, this condition for the first cover is $\chi_R + \chi_S = \frac{1}{2} + \frac{1}{2} = 1 \geq 1$ and for the second one evaluates to $\chi_R + \chi_S = 1 + 1 = 2 \geq 1$. Comparing both covers under the AGM bound, the first cover gives a bound of $|\mathcal{Q}_\triangle| \leq \sqrt{|R| * |S| * |T|}$, whereas the second one states a bound of $|\mathcal{Q}_\triangle| \leq |R| * |S|$. Under the assumption that all relations have a size of at most N, the first edge cover gives a tight upper bound of $N^{\frac{3}{2}}$, while the second one gives a worse bound of N^2.

The AGM bound and the above example clearly show the bound's connection to the sizes of the individual input relations. In order to minimize the right hand side of the AGM inequality given in Lemma 3.1 the linear program in Equation 3.1 needs to be solved optimally and such optimal solution is referred to as `fractional edge cover number` $\rho^*(\mathcal{Q}, D)$. Then, this inequality can be denoted by $|Q| \leq 2^{\rho^*(\mathcal{Q}, D)}$. [NRR13]

3.3.3 From Theoretical Bounds to Algorithms

The historic outline of fundamental research on worst-case optimal bounds and algorithms clearly shows that current algorithms and bounds result from years of interdisciplinary research. Early geometric inequalities such as the one of Loomis-Whitney [LW+49] or its generalized version by Bollobás and Thomason [BT95] originated from bounding problems in the area of geometry. Atserias et al. [AGM08] study how large the output size of a join query can get with respect to its input relations in the worst-case. To formalize this problem, the researchers defined and proved tight bound on the output size of a query which is given in Lemma 3.1. Moreover, their proof is based on Shearer's entropy inequality. However, no algorithm could be derived from such non-constructive proof of the AGM inequality that achieves these optimal bounds.

Ngo et al. [NPRR12] introduce a worst-case optimal algorithm that yields a constructive proof of the AGM bound instead of making use of the information theoretic Shearer's inequality. At first the researchers showed the equivalence between the AGM inequality and the inequality by Bollobás and Thomason [BT95]. This equivalence implies that the Loomis-Whitney inequality is a special case of the AGM inequality. Based on that knowledge, the researchers describe an algorithm for Loomis-Whitney instances and in another iteration a more general algorithm for all join queries from which they prove the AGM inequality.

Another constructive proof of the AGM bound was given by Ngo et al. [NRR13]. The researchers obtained the `query decomposition lemma`, given in Lemma 3.3 from the recent, but complicated algorithmic approach of Ngo et al. [NPRR12] from 2012 and present a correctness proof. Furthermore, they applied this lemma to inductively prove the AGM inequality. This inductive proof structure paved the way to a class of recursive join algorithms which they called `Generic-Join` and which is given in Algorithm 3.1. All these efforts of deriving algorithms that run within the presented optimal bounds require a constructive proof in order to formulate algorithmic procedures for them.

3.4 Worst-case Optimal Join Algorithms

This section aims for discussing the recursive `Generic-Join` algorithm 3.1 introduced by Ngo et al. [NRR13] and how it is handling skew in the input data of a join query. Besides the most general `Generic-Join` algorithm, the researchers discuss their basic ideas through introducing two algorithms that are worst-case optimally bounded. These algorithms are referred to as `The Power of Two Choices` and `Delaying of Computation` and are introduced on the example of the triangle query $\mathcal{Q}_{\triangle}$. Nevertheless, these two algorithms can be used to compute more general natural join queries when not specified explicitly for the triangle query $\mathcal{Q}_{\triangle}$. The ideas behind these algorithms are an optimal way of dealing with skew in the input relations of a join query. As already mentioned in this chapter, the techniques presented here are closely related to geometry. Moreover, the algorithms break with the pairwise evaluation of `Multi-Way-Joins` by dealing with them in one go. Furthermore, Ngo et al. [NRR13] describe and analyzes

the algorithm from Ngo et al. [NPRR12] and the `Leapfrog-Triejoin` [Vel14] using their simplified framework and state that both join algorithms are a special case of their `Generic-Join` algorithm 3.1.

3.4.1 Heaviness of Attribute Values

In order to deal with skewed data, Ngo et al. [NRR13] discuss a classification of attribute values into light and heavy ones. The idea behind taking a value's heaviness into account is to handle nodes of low and high skew using different techniques. This separation is important, because heavy values have a larger fan-out than lighter ones and can therefore lead to larger intermediate results. An example of such a heavy value is illustrated in Figure 3.3. The value a_0 has a high fan-out and is therefore an example of skew. The researchers discuss two different approaches of handling light and heavy values on the example of the triangle query $\mathcal{Q}_\triangle$ and construct two algorithms for this query that follow their observations.

3.4.2 The Power of Two Choices

Ngo et al. [NRR13] introduce an algorithm on the example of the triangle query $\mathcal{Q}_\triangle$ called `The Power of Two Choices` which treats heavy and light values differently. They define an attribute value a_i to be heavy in context of the triangle query $\mathcal{Q}_\triangle$ if

$$|\sigma_{A=a_i}(R \bowtie T)| \geq |\mathcal{Q}_\triangle[a_i]| \tag{3.5}$$

where

$$\mathcal{Q}_\triangle[a_i] := \pi_{B,C}(\sigma_{A=a_i}(\mathcal{Q}_\triangle)). \tag{3.6}$$

Note that these definitions can be analogously made for values b_i and c_i of attributes B and C. Moreover, this approach requires the relations to be indexed and an estimation of the values' heaviness before computing the results. The inequality 3.5 defines a value to be classified as heavy if it contributes more to the intermediate relation $R \bowtie T$ than to the overall output size of the query. To evaluate the join for heavy values a_i, for each tuple $(b, c) \in S$ it will be checked if $(a_i, b) \in R$ and $(a_i, c) \in T$. Since this strategy just scans relation S, the computation of $\mathcal{Q}_\triangle[a_i]$ can be done in linear time. On the other hand, light values are treated the other way around. At first, the join $\sigma_{A=a_i}(R) \bowtie \sigma_{A=a_i}(T)$ is evaluated and the results are filtered by probing against values in S.

Ngo et al. [NRR13] formulate an algorithm that estimates a value's heaviness and then applies the above strategies for computing the output. Moreover, they prove that the algorithm is in $O(N^{\frac{3}{2}})$ for $|R| = |S| = |T| = N$ and state that their `Generic-Join` algorithm 3.1 is a generalization of the `The Power of Two Choices` algorithm beyond triangles.

3.4.3 Delaying of Computation

Another approach to distinguish between heavy and light values is introduced by Ngo et al. [NRR13] through their `Delaying of Computation` algorithm which has been

depicted on the example of the triangle query $\mathcal{Q}_\triangle$ as well. Instead of trying to estimate the heaviness of any value a_i before computing the join, this algorithm aims for gradually reducing skew while constructing the query's solution. This reduction can be achieved by intersecting the same attributes across multiple relations, which results in pairwise intersections for the triangle query $\mathcal{Q}_\triangle$.

In the first step, `Delaying of Computation` obtains a candidate set of b_j that can occur in an output tuple (a_i, b_j, c) with value a_i by computing the intersection $L_B^{a_i} = \pi_B(\sigma_{A=a_i}(R)) \cap \pi_B(S)$. Next, for any $b \in L_B^{a_i}$ a candidate set of c_k is computed following the same approach which results in a set $L_C^{a_i,b_j} = \pi_C(\sigma_{B=b_j}(S)) \cap \pi_C(\sigma_{A=a_i}(T))$. Finally, each combination of values from these set are added to the result. The computation of candidate sets for b_j and c_k aim for reducing the skew towards computing the answer to a given query.

Ngo et al. [NRR13] show that the `Delaying of Computation` algorithm has the same worst-case optimal runtime as the `The Power of Two Choices` algorithm. Both algorithms combine ideas of handling skew in the input relations into recursive algorithms that are special instances of the general `Generic-Join` algorithm 3.1.

3.4.4 Generic-Join Algorithm

The `Generic-Join` algorithm 3.1 describes a class of recursive join algorithms that has been constructed from an inductive proof of the AGM inequality as given in Lemma 3.1 using the `query decomposition lemma` stated in Lemma 3.3. `The Power of Two Choices` as well as the `Delaying of Computation` algorithm are two special instances that have been introduced on the example of the triangle query $\mathcal{Q}_\triangle$ of this general algorithm. These two specific algorithms have been discussed to introduce the idea of recursively computing `Multi-Way-Joins` and crafting the resulting tuples using candidate sets of attributes.

Furthermore, Veldhuizen's `Leapfrog-Triejoin` [Vel14] and the algorithm of Ngo et al. [NPRR12] from 2012 are specific instances of the `Generic-Join` algorithm 3.1. Ngo et al. [NRR13] state that it has been straightforward to design the `Generic-Join` algorithm 3.1 from their proof of the `query decomposition lemma` given in Lemma 3.3 and the query decomposition from Equations 3.8 and 3.9 which they applied in the inductive step of their constructive AGM inequality proof.

In order to prove the AGM inequality constructively, Ngo et al. [NRR13] introduce the `query decompositon lemma` 3.3. Roughly speaking, this `query decomposition lemma` can be interpreted as follows. Facing a partition $\mathcal{V} = I \uplus J$ of the original query hypergraph's vertices that partitions the hypergraph into sets I and J. Then, the set $L = \bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$ represents an intermediate result of the original query corresponding to its sub-query belonging to partition I. Any intermediate tuple $\boldsymbol{t}_I \in L$ needs to be further enriched with attribute values corresponding to the sub-query defined by partition J, which can occur in an output tuple together with $\boldsymbol{t}_I$. Moreover, the semi-join $R_F \ltimes \boldsymbol{t}_I$ ensures that only attributes of relation R_F contribute to the `fractional edge cover`

number, that can occur in an output tuple together with $\boldsymbol{t}_I$. This situation is formalized by the left-hand side of Inequality 3.7. Therefore, the inequality defined within the query decomposition lemma states, that the fractional edge cover number calculated following the partitioning approach is at most as large as the one calculated without partitioning the hypergraph.

Furthermore, the major components of the query decomposition lemma can be found in the Generic-Join algorithm 3.1. This algorithm is designed to recursively partition the original query hypergraph and reconstruct output tuples by only considering attribute values that can occur in output tuples, together with intermediate tuples $\boldsymbol{t}_I$. Therefore, major components of the Generic-Join algorithm 3.1 have been derived from the query decomposition lemma.

Lemma 3.3 (Query Decomposition Lemma)**.** *Let $Q = \bowtie_{F \in \mathcal{E}} R_F$ be a natural join query represented by a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, and let $\boldsymbol{x}$ be any fractional edge cover for $\mathcal{H}$. Further, let $\mathcal{V} = I \uplus J$ be an arbitrary partition of $\mathcal{V}$ such that $1 \leq |I| < |\mathcal{V}|$ and let $L = \bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$. Then the following inequality holds.*

$$\sum_{\boldsymbol{t}_I \in L} \prod_{F \in \mathcal{E}_J} |R_F \ltimes \boldsymbol{t}_I|^{\chi_F} \leq \prod_{F \in \mathcal{E}} |R_F|^{\chi_F} \tag{3.7}$$

Note that a specialized form of the query decomposition lemma for the triangle query $\mathcal{Q}_\triangle$ has been implicitly applied in the The Power of Two Choices algorithm.

Query Decomposition

Query decomposition can be used in order to decompose a given query according to any given partition of its hypergraph's vertices and represent the original query in terms of these partitions. Ngo et al. [NRR13] make use of query decomposition to recursively craft the resulting tuples in their Generic-Join algorithm 3.1 and define this query decomposition as follows.

Let $\mathcal{V} = I \uplus J$ be an arbitrary partition of $\mathcal{V}$ such that $1 \leq |I| < |\mathcal{V}|$ and let $L = \bowtie_{F \in \mathcal{E}_I} \pi_I(R_F)$. Then, for each tuple $\boldsymbol{t}_I \in L$ a new join query can be defined as

$$\mathcal{Q}[\boldsymbol{t}_I] := \bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes \boldsymbol{t}_I). \tag{3.8}$$

Based on these new join queries, the original query $\mathcal{Q}$ can be written as

$$\mathcal{Q} = \bigcup_{\boldsymbol{t}_I \in L} (\boldsymbol{t}_I \times \mathcal{Q}[\boldsymbol{t}_I]). \tag{3.9}$$

The proofs of the query decomposition lemma and the AGM inequality using query decomposition are explained by Ngo et al. [NRR13] in great detail. Moreover, the researchers show by induction that the runtime of the Generic-Join algorithm 3.1 is $\tilde{O}(m * n * \prod_{F \in \mathcal{E}} |R_F|^{\chi_F})$ where $\tilde{O}$ is hiding a potential log factor. They further note, that

Algorithm 3.1: Generic-Join($\bowtie_{F\in\mathcal{E}} R_F$) [NRR13]

Input: Query $\mathcal{Q}$, Hypergraph $H = (\mathcal{V}; \mathcal{E})$

$\mathcal{Q} \leftarrow \emptyset$;
if $|\mathcal{V}| = 1$ **then**
  return $\cap_{F\in\mathcal{E}} R_F$;
end
Pick I arbitrarily such that $1 \leq |I| < |\mathcal{V}|$;
$L \leftarrow$ Generic-Join($\bowtie_{F\in\mathcal{E}_I} \pi_I(R_F)$);
for *every* $\mathbf{t}_I \in L$ **do**
  $\mathcal{Q}[\mathbf{t}_I] \leftarrow$ Generic-Join($\bowtie_{F\in\mathcal{E}_J} \pi_J(R_F \ltimes \mathbf{t}_I)$);
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\mathbf{t}_I\} \times \mathcal{Q}[\mathbf{t}_I]$;
end
return $\mathcal{Q}$;

the algorithm of Ngo et al. [NPRR12] from 2012 is an instance of the `Generic-Join` algorithm 3.1 where it takes any $J \in \mathcal{E}$ and $I = \mathcal{V} - J$ and solves the sub-queries $\mathcal{Q}[\boldsymbol{t}_I]$ differently. Additionally, they show that this algorithm matches `The Power of Two Choices` algorithm. Furthermore, the `Leapfrog-Triejoin` algorithm [Vel14] is also an instance of the `Generic-Join` algorithm 3.1 where $\mathcal{V} = [n]$ and $I = 1, ..., n-1$. [NRR13]

CHAPTER 4

Integrating Worst-Case Optimal Joins into Column-Stores

The aim of this chapter is to discuss the integration of a class of WCOJ algorithms into column-oriented database systems. More specifically, the `Generic-Join` algorithm 3.1 that has been discussed in Chapter 3 is integrated in the MonetDB column-store which has been introduced in Chapter 2. This chapter starts with a discussion of related database system research that aims for utilizing WCOJ algorithms. Related work can cover integration efforts similar to this work, the introduction of new query processing engines, or even discussions on suitable data structures that can be leveraged to implement classes of WCOJ algorithms. After looking at related research, the methods applied in order to integrate the `Generic-Join` algorithm 3.1 into the MonetDB system are introduced. Furthermore, this chapter takes a look into MonetDB's query compiler to deeply understand the compilation process of a query into a sequence of MAL instructions. The obtained knowledge is important for the integration part of this work, because MonetDB's query compiler needs to be modified in a way that the generated MAL code follows the `Generic-Join` algorithm 3.1.

The key contribution of this chapter is to present a sequence of MAL instructions that is equivalent to the `Generic-Join` algorithm 3.1. This conversion will be discussed on the example of the triangle query $\mathcal{Q}_{\triangle}$. Moreover, MonetDB's query compiler will be modified to generate MAL programs that process `Multi-Way-Joins` in a worst-case optimal manner. This work focuses on the evaluation of natural join queries, since research discussed in this work presents WCOJ techniques for natural join queries. Finally, technical obstacles are discussed that were encountered during the integration process.

4.1 Related Work

This section discusses research that introduces ideas on how to integrate a class of WCOJ algorithms into database engines. To that end, Aberger et al. [ALT+17] present a high-performance graph-processing engine called `EmptyHeaded` that supports a query language similar to `Datalog`. `EmptyHeaded` handles `Multi-Way-Joins` according to a class of WCOJ algorithms and the researchers state that their engine outperforms comparable specialized graph processing engines up to three orders of magnitude on PageRank and graph pattern queries, for example. It even outperforms low-level systems, where users have to write imperative low-level code, which tends to be more efficient than high-level systems using a simpler query language. Aberger et al. [ALT+17] state that their main contribution is a novel architecture consisting of a query compiler that represents a logical query plan using Generalized Hypertree Decomposition (GHD) instead of a traditional `relational algebra`. Moreover, their execution engine exploits low-level data layout in order to increase SIMD parallelism. The combination of these two concepts enables `EmptyHeaded` to match the proposed worst-case optimal bounds. Additionally, the researchers mention that the application of GHD comes with additional bookkeeping information that is exploited to bound the size of intermediate results. Finally, they discuss how their optimizer deals with skew at several granularity levels by selecting set layouts and intersection algorithms according to data characteristics at runtime.

Besides their research on the `EmptyHeaded` engine, Aberger et al. [ALOR17] introduce an in-memory query processing engine called `LevelHeaded` that aims for evaluating both, business intelligence and linear algebra workloads efficiently. They compare popular engines with respect to their performance on business intelligence versus linear algebra workloads and show that their `LevelHeaded` engine performs well in both worlds. In order to unify these query workloads in the `LevelHeaded` engine, Aberger et al. [ALOR17] introduce a novel WCOJ query architecture. `LevelHeaded`'s architecture composes three major techniques. First, `LevelHeaded` translates generic SQL queries to hypergraphs, more specifically to GHDs. This eliminates attributes, since only key columns of SQL queries are added to the GHDs. Moreover, they discuss heuristics applied in order to obtain the best GHD representation. Besides this selection mechanism, the researchers introduce a technique to further push down selections below joins to boost performance. Furthermore, Aberger et al. [ALOR17] introduce a new cost-based optimizer that obtains the most suitable attribute order similarly to obtaining the best possible join order in traditional query optimizers. The researchers describe their cost heuristics and claim that choosing different attribute orders can result in an order of magnitude performance difference on the same query. The third introduced technique is a query execution optimizer for `group by` operators. They claim that this optimizer leads to resistance against skew for `group by` operations. Combining these techniques into an WCOJ query architecture, `LevelHeaded` outperforms other relational engines on linear algebra queries by at least a magnitude of order. Moreover, `LevelHeaded` is among the top 31% of best-of-the-breed solutions for business intelligence and linear

algebra queries combined.

Freitag et al. [FBS+20] present an implementation of WCOJ algorithms into their UMBRO DBMS that has been developed by their research group. The researchers claim, that their approach can be implemented in any general-purpose DBMS that supports hybrid transactional as well as analytical workloads. They identify two major shortcomings that prevent WCOJ algorithms finding their way into general-purpose DBMSs. First, they require suitable indices on all permutations of attributes that can take part in a join, which implies great maintenance and storage overhead. Secondly, `Multi-Way-Joins` perform worse than pairwise joins if there are no growing intermediate results. Thus, Freitag et al. [FBS+20] argue that an implementation of a WCOJ algorithm in a general-purpose DBMS needs an optimizer that only introduces `Multi-Way-Joins` if they are beneficial to the query evaluation. Moreover, such an implementation has to be capable of dynamically building and maintaining performant index structures without the need of persisting them to disk. To that end, the researchers introduce a hash-based WCOJ algorithm that does not rely on precomputed indices. Their algorithm exploits a hash trie data structure that aims for storing tuples in a trie based on the hash values of their join attributes. These hash tries can be dynamically built in linear time. Furthermore, the researchers introduce a heuristic query optimizer that generates hybrid query plans, which combine pairwise joins with `Multi-Way-Joins`, and take cardinality estimations into account. Their results show, that their proposed algorithm outperforms binary join plans as well as other systems with integrated WCOJ algorithms.

A recent paper of Navarro et al. [NRR20] from 2020 analyzes that the implementation of WCOJ algorithms requires enhanced indexing structures. Such implementations require additional data structures, heavily indexed databases or clever heuristics to compute the most suitable evaluation path given index structures. An example of such a heuristic is given by Aberger et al. [ALOR17] which is utilized to obtain the best possible attribute order. Thus, a gain of flexibility in attribute processing leads to heavier data structures and bigger storage footprints. To tackle these concerns, Navarro et al. [NRR20] aim for developing optimal join algorithms that are counterfeiting this additional storage requirements and that are independent of any attribute order. Their key contribution is a representation of input relations as point sets in n-dimensional grids. This representation is realized using quadtrees, which require an almost optimal amount of space. Furthermore, the researchers show that WCOJ algorithms can be derived directly from this representation of input relations. Moreover, they briefly discuss the evaluation of the triangle query $\mathcal{Q}_{\triangle}$ using the `Generic-Join` algorithm 3.1 and introduce how their quadtrees (qdags) handle this query. This is done by dividing the output space, which is a grid of size N^3, into 8 subgrids of size $(\frac{N}{2})^3$ and recursively evaluate each subgrid for the given query. Moreover, the researchers introduce their qdags as well as lazy qdags data structures that benefit from regularities in grids of a relation. Navarro et al. [NRR20] state, that their WCOJ algorithm is a competitive alternative to existing ones and further mention that their qdags representation is much more space efficient than existing index structures.

4.2 Methodology

This section elaborates on the methods used for integrating the `Generic-Join` algorithm 3.1 into MonetDB. Moreover, it gives a structured overview of what methods utilized in the integration process, including a brief introduction to each one. Different methodological approaches have to be combined in order to gain knowledge about the MonetDB compiler's structure and code as well as to translate the `Generic-Join` algorithm 3.1 into MAL code and to integrate this algorithm into MonetDB's query compiler.

1. **Literature Review**
   Background information and fundamental knowledge about column-oriented database systems, especially MonetDB, and WCOJ algorithms has been discussed in Chapter 2 and Chapter 3 respectively. This literature review builds a foundation of knowledge from both domains, which are combined in this work. Moreover, good theoretical understanding is necessary to successfully integrate WCOJ algorithms into column-oriented database systems.

2. **Reverse Engineering**
   MonetDB [Monb] provides a rich amount of user-specific documentation. Moreover, its architecture and internal concepts as well as MAL [Monc] are well documented. Nevertheless, the documentation is lacking information on how to get developers started with extending the MonetDB system, especially how a query is translated into a sequence of MAL instructions. Therefore, main concerns before being able to start integrating the `Generic-Join` algorithm 3.1 are to reverse MonetDB's source code, understand the compiler's internal representation of a query and how queries are translated in MAL code. This knowledge will be essential to actually being able to integrate the WCOJ algorithm into MonetDB.

3. **Translation of the Theoretical Algorithm to MAL Code**
   The `Generic-Join` algorithm 3.1 is represented using different relational algebra operators that have to be translated into equivalent MAL instructions. Based on the knowledge obtained in the reverse engineering step, the translation is crucial to estimate the integration's feasibility. This translation is carried out on the example of the triangle query $\mathcal{Q}_{\triangle}$. The manual conversion of relational algebra operators utilized in the `Generic-Join` algorithm 3.1 to MAL instructions tries to be as direct as possible in the first place. Finally, improvements are introduced based on the evaluation of this direct translation.

4. **Integration of the Theoretical Algorithm into the Query Compiler**
   The `Generic-Join` algorithm 3.1 is finally integrated into the MonetDB query compiler. Taking the gained knowledge from reverse engineering and manual translation into account, MAL features required for integrating the class of WCOJ algorithms into MonetDB need to be implemented and query optimizers need to

be adapted and extended accordingly. Furthermore, binary join evaluation has to be replaced with the `Multi-Way` approach introduced by the `Generic-Join` algorithm 3.1.

4.3 MonetDB Query Compiler

This section aims for giving a brief overview of how MonetDB compiles a given query into a MAL program. The discussion focuses on the internal representation of a query using an operator tree, where each operator produces a statement tree structure. Furthermore, the data structures used internally are outlined. This section does not shed a light on the query parser, since this work does not alter MonetDB's existing one. However, the integration of a class WCOJ algorithms into MonetDB requires adapting the translation process in order to implement the join processing schema introduced by the `Generic-Join` algorithm 3.1. The core of this translation process is a tree of operators that corresponds to the relational algebra representation, as shown in Listing 2.1. This operator tree is traversed, and each operator node generates a statement tree or a list of such that corresponds to the set of MAL instructions represented by that operator node. To outline this translation process, the operator tree as well as the statement trees will be discussed on the example of the triangle query $\mathcal{Q}_{\triangle}$. Note that the major insights outlined in this section have been obtained during the reverse engineering process of MonetDB's source code.

4.3.1 Query Translation Process

The following outlines MonetDB's query translation process under the assumption that the query has already been parsed and that it is represented using a combination of appropriate data structures. Moreover, implementation details are discussed in order to point out how a single operator can result in a sequence of MAL instructions and to elaborate on how information is passed through the translation process. Furthermore, the discussion aims for obtaining the most suitable possibility for replacing MonetDB's pairwise join processing with the `Multi-Way-Join` processing schema introduced by WCOJ algorithms. The second desirable outcome is a way of replacing the join translation semantics and still being capable of utilizing any other operator without the need for adaptation.

The query translation logic mainly resides in the `rel_bin.c` file of MonetDB's source code, since it contains the logic to traverse the operator tree and translate the parsed query into a MAL program. Nodes within this operator tree are represented by a C struct called `sql_rel`, which is shown in Listing 4.1. Besides a number of integer flags which are partly described in the source code, there is the `op` property of enum type `operator_type`, which describes the operator's type. Such types follow the naming convention `op_<operator>`. Figure 4.1 shows the operator tree of the triangle query $\mathcal{Q}_{\triangle}$, where nodes are reference by operator types (e.g. `op_basetable`). The three basetable operators correspond to the relations R, S and T which are joined in a pairwise

fashion, as also shown in Figure 4.1. Furthermore, the binary tree structure is established by making use of the `void` pointers `l` and `r` which define either a pointer to the left or right child node or address data to be accessed in leave nodes of the tree. Moreover, `exps` represents a pointer to a list of expressions utilized by some operators during translation, `p` can represent properties relevant to the optimizer and `nrcols` describes the number of columns involved.

```
typedef struct relation {
  sql_ref ref;
  operator_type op;
  void *l;
  void *r;
  list *exps;
  int nrcols;
  unsigned int
    flag:16,
    card:4,
    dependent:1,
    distinct:1,
    processed:1,
    outer:1,
    grouped:1,
    single:1,
    used:2;
  void *p;
} sql_rel;
```

Listing 4.1: MonetDB's `sql_rel` data structure which represents an operator tree's nodes. [Mon21]

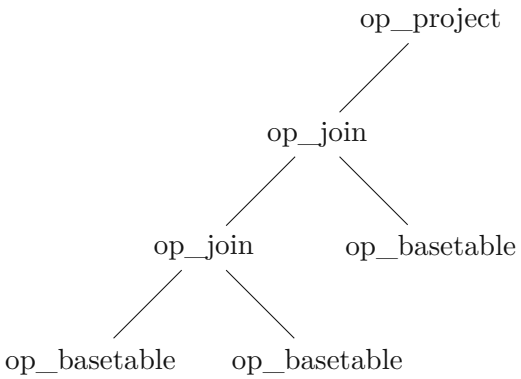


Figure 4.1: MonetDB's internal representation of the triangle query $\mathcal{Q}_\triangle$ as operator tree.

The operator tree is traversed in postorder and each tree node is executing operator specific functions in order to translate the current node given by an object of type `sql_rel`, as given in Listing 4.1 into a sequence of MAL instructions. The key data structure for this operator-wise translation is internally referred to as `stmt` and is shown

in Listing 4.2. There are statements of different types following the naming convention `st_<type>` (e.g. `st_bat`, `st_tid`, `st_result`, etc.). Moreover, properties `op1`, `op2` and `op3` are used to link to child statements. Due to the fact that most statement types just make use of `op1` and `op2`, many operator nodes return a binary tree of statements as result (e.g. `op_basetable`). Providing such concatenation of statements to subsequent operators or parent nodes in the operator tree enables to transport context about the translation through the whole process. Another important property of the `stmt` data structure is `op4` which can represent other types and is mainly utilized to address a list of statements. Operators can generate such statement lists during translation of multiple expressions or columns. This is done for instance in case of `op_basetable` when multiple columns of a single table are projected.

Moreover, the properties `tname` and `cname` relate to a table's and column's name, respectively. These fields can be important in situations, where a specific attribute needs to be obtained, as well as for debugging purposes. The `nr` property describes the number of the generic variable identifier the statement is assigned to (e.g. X_100). Furthermore, the fields `tvar` and `hvar` are used for list iterations, as shown in Listing 2.7. `tvar` represents the temporal variable assignment per iteration, and `hvar` relates to the corresponding `oid` value. Thus, `tvar` describes the value itself (e.g. a string value) and `hvar` its position within the iterated column. Additionally, the `cand` statement pointer can be utilized by operators as an optional candidate list.

The discussion about statements has been related to sharing translation context between subsequent operators so far. The context propagation is one of the main concerns this work has to deal with when modifying MonetDB's query compiler. In order to outline the full picture, we will briefly introduce how MonetDB manages MAL instructions that have been produced by an operator. The main connection between a `stmt` object, which aims for representing translation context and the generated MAL instructions, is the `q` property of type `InstrPtr`. The source file `sql_statement.c` contains a collection of functions that generate specific MAL instructions and return corresponding statement objects for context propagation. Each such function creates a new `InstrPtr` object by invoking `newStmt` or `newStmtArgs`. These functions can be parameterized with the MAL module and the corresponding instruction to be created. Moreover, the first parameter is the backend's `MalBlkPtr` instance, which abstractly represents the set of generated MAL instructions. Newly created instructions are added to this `MalBlkPtr` instance and can also be modified or accessed using predefined functions. This data structure represents the current state of the generated MAL program, and optimizations are also carried out on it. Furthermore, the MAL interpreter utilizes this data structure during execution.

```
typedef struct stmt {
  st_type type;
  struct stmt *op1;
  struct stmt *op2;
  struct stmt *op3;
```

```
  stmtdata op4;

  unsigned int
    nrcols:2,
    key:1,
    aggr:1,
    partition:1,
    reduce:1;

  struct stmt *cand;

  int flag;
  int nr;
  int tvar;
  int hvar;

  const char *tname;
  const char *cname;
  InstrPtr q;
} stmt;
```

Listing 4.2: MonetDB's `stmt` data structure which is used to represent statements. [Mon21]

4.3.2 Compiling the Triangle Query $\mathcal{Q}_\triangle$

The previous section discussed the translation process of a given query into a MAL program and describes the key data structures used in this process. This section outlines the translation process on the example of the triangle query $\mathcal{Q}_\triangle$ and discusses how natural joins are compiled with respect to this example.

The following explanation assumes that there exists a database with relations $R(A, B), S(B, C)$ and $T(A, C)$ running in MonetDB and that the triangle query $\mathcal{Q}_\triangle$ has been submitted to query this database. After the system has parsed the query, the query is transformed into an operator tree represented through nodes of type `sql_rel`, as described in the previous section. Figure 4.1 shows the resulting operator tree that is traversed in postorder during the translation process. Therefore, the operators are implemented in a way that they aim for translating their child nodes first, since they serve as input. This is realised by invoking the traversal logic on the `l` and `r` pointer of the current node. Following this recursive traversal logic, the first operator node to be translated is the left `op_basetable` of the second `op_join` node. This basetable operator generates MAL instructions which load the corresponding table's columns. The resulting MAL sequence for column b of relation R is shown in line 2 to 4 of Listing 2.4. Note that this is the optimized version of the resulting instructions, since intermediate MAL code for basetable operations contains delta operations to realise cracking. If delta operations are not required, the optimization pipeline removes them. Moreover, Figure 4.2 illustrates a statement of type `st_list` that results from translating the basetable operator for relation $T(A, C)$. This list contains statement trees for columns A

and C with three cascading `st_join` nodes. The deepest `st_join` statement, which takes two `st_bat` statements as input, produces a delta instruction between these two BATs. One of the BATs relates to the original column and the other represents introduced updates. The `st_join` node above binds the column of interest based on the table identifier, and the top level `st_join` statement generates a projection instruction that projects the column values and assigns the resulting BAT to a new variable.

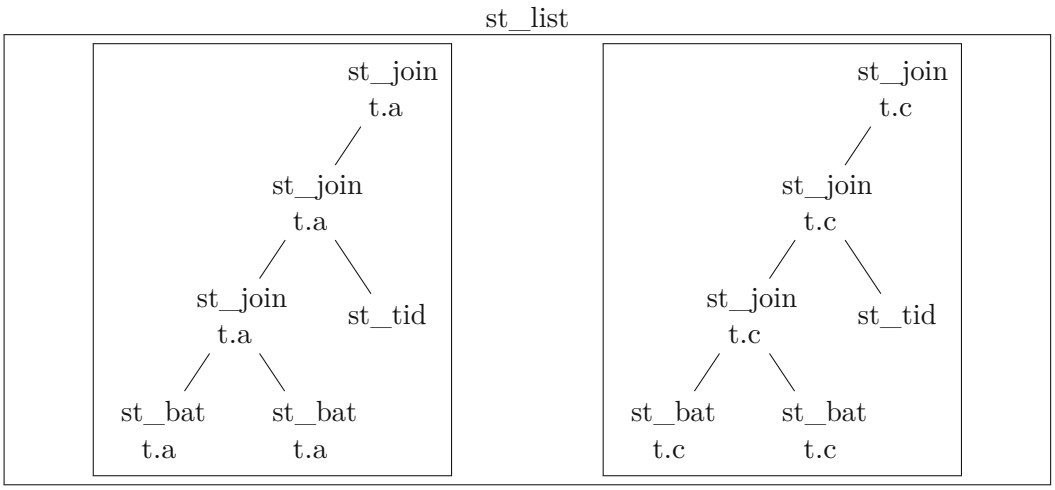


Figure 4.2: Resulting statement structure of type `st_list` after translating the basetable operator for relation T.

Note that the other two basetable operator nodes perform the same operations on the remaining columns R and S respectively. Continuing with the triangle query example, the next operator node to be considered is the deepest `op_join` node. This operator aims for performing a natural join between relations T and S, since the example query naturally joins all relations. Thus, the relations T and S are joined on attribute C. As shown in Figure 4.1 the deepest `op_join` node has two child nodes of type `op_basetable` which generate MAL code in order to load columns of relations T and S. These two child nodes are input to the join operator node, which aims for generating MAL instructions based on the context provided by the child nodes' resulting statement structure. Figure 4.3 depicts the resulting statement tree structure after executing this join operator. The `st_result` and the top level `st_join` statement have been generated by the current join operator. Based on the context of its input nodes, the join operator decides on the most suitable and efficient type of join to be translated to (e.g. equi-join, semi-join, etc.). This translation is represented by the top level `st_join` node, as shown in Figure 4.3. The `st_result` node serves materialization purposes of the join result and generates a projection instruction if required. As already introduced in the previous section, the statement tree structure from Figure 4.3 is the result of the corresponding `op_join` node and serves to propagate translation context to the parent operator node. Besides that, the corresponding `InstrPtr` pointer aims for storing information about the generated MAL instructions in the backend's `MalBlkPtr` data structure. The MAL instructions

resulting from translation of the discussed join are given in Listing A.1 on lines 13 to 15.

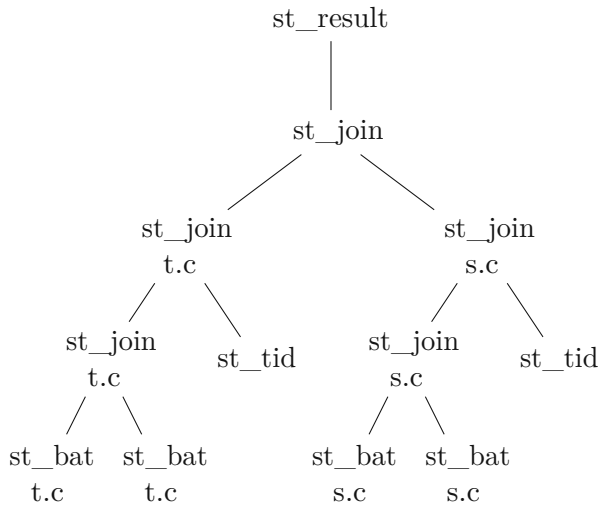


Figure 4.3: Resulting statement structure after translating the join operator for relations S and T.

The top level `op_join` operator node takes the resulting statement tree structure of the first pairwise join of relations S and T as well as the statement tree structure of the result of the basetable operator on relation R as inputs. The structures are given in Figure 4.3 and Figure 4.2 respectively. The resulting statement tree structure follows the same approach as discussed for the first join operation. Projections are generated according to the same logic, since each join operator is translated using the same methods given in the `rel_bin.c` source file. Many code fragments aim for looping over a statement of type `st_list` or expressions defined by the `sql_rel` structure. This is a recurrent pattern throughout the implementation of different operators, because many operators produce position lists as a result, such as the join operator does. Each attribute of the corresponding relation is iterated over in loops and projected on such a BAT of OIDs in order to obtain the join's materialized result. This has to be done since any attribute is managed and stored as separate BAT.

After translating both pairwise join operators into a sequence of MAL instructions, the translation process continues with the `op_project` operator in the root node of the operator tree, as shown in Figure 4.1. The root node of type `op_project` serves various purposes in general. First, it initiates the translation of its child. This process has been discussed on the example of the triangle query $\mathcal{Q}_{\triangle}$ in this section. Moreover, it aims for generating MAL code to select the top n results, as well as code to specify the order of results. Furthermore, this operator can generate MAL code that calculates distinct results.

The insights into MonetDB's query translation logic provided in this section build the foundation for planning the integration process of a class of WCOJ algorithms

given by the `Generic-Join` algorithm 3.1. The discussed example query underlines the feasibility of integrating the new way of join processing into MonetDB while still supporting any other SQL feature out of the box. The following sections of this chapter introduce and discuss an integration approach that replaces the pairwise join evaluation with `Multi-Way-Join` processing, as introduced by WCOJ algorithms.

4.4 Integrating the Generic-Join Algorithm into MonetDB

The aim of this section is to discuss the integration process of WCOJ algorithms into column-oriented database systems on the example of the `Generic-Join` algorithm 3.1 and the MonetDB column-store. In order to understand the difficulties encountered during the integration process, implementation challenges are outlined. These challenges compose problems that need to be solved in order to enable MonetDB to translate a given SQL query into a sequence of MAL instructions following the `Generic-Join` algorithm 3.1. Furthermore, the translation of relational algebra operators applied in the `Generic-Join` algorithm 3.1 into sequences of MAL instructions are discussed in great detail. A direct translation is carried out and evaluated in the first place. Based on the obtained results, certain parts of the direct translation are optimized. Finally, MonetDB's query compiler and optimizer are adapted in order to make the column-store generate MAL programs that follow the `Generic-Join` algorithm 3.1. Newly introduced and adapted data structures as well as adapted query optimizer and code generators are outlined.

4.4.1 Integration Challenges

This section briefly discusses the main obstacles encountered during the overall integration process of the `Generic-Join` algorithm 3.1 into MonetDB. This problem discussion is important in order to point out concepts that need to be supported by the column-store, as well as introduce features that have to be integrated into MonetDB. The following gives an overview of challenges to be tackled for a successful integration.

1. **Reverse Engineering of MonetDB's Codebase**
   A very time-consuming but essential challenge to be tackled was to gain knowledge about MonetDB by reversing its codebase. The main focus during the reverse engineering process was to understand the SQL frontend code and how queries are translated into a sequence of MAL instructions. A question of special interest is whether all MAL features required for realising the `Generic-Join` algorithm 3.1 are already supported by the MonetDB query compiler or not. Another point of great importance is the possibility of just changing the way joins are handled during translation while still supporting any other SQL feature without the need for code adaptations. Moreover, the extension modules `algebra`, `bat` and `group` have to be explored in order to be able to pick the most suitable MAL instructions during

the translation process and to understand which code constructs are possible. Due to the fact that MAL code optimizer are leveraged in order to transform a MAL program into a more efficient one, it is important to get to know the corresponding optimizer pipeline and its modules. This is of great interest, since newly introduced code constructs might be wrongly optimized due to the optimizer's lack of knowledge about this new code construct. For instance, an empty bindings optimizer would remove temporal variables if they are not explicitly supported. To sum up, the reverse engineering process aims for gaining knowledge about the translation of a SQL query into a MAL program in order to be able to introduce new code constructs and to adapt existing behaviour.

2. **Debugging mserver5**
   Since MonetDB's codebase needs to be modified and new concepts have to be introduced during the integration process, it is of importance to be able to debug the corresponding source code. To that end, knowledge about the programs, which MonetDB is composed of, is required. These programs are `monetdb`, `monetdbd` and `mserver5`. Since query compilation and execution is done by the `mserver5` program, the ability to debug this application is required. Unfortunately, the official MonetDB documentation [Monb] does not state a suitable debugging configuration. Thus, different configurations have been obtained during reverse engineering and were used to understand the meaning of certain variables and settings. Listing 4.3 depicts the most suitable debug configuration for the purposes of this work. The placeholder *path/to/db* relates to the location of the database to be started.

```
mserver5 --dbpath=path/to/db
  --set monet_vault_key=path/to/db/.vaultkey
  --set monet_mod_path=/usr/local/lib/monetdb5
  --set gdk_nr_threads=24
  --set max_clients=64
  --set sql_optimizer=default_pipe
```

Listing 4.3: Debug configuration of mserver5.

3. **Different Way of Translating Joins**
   The original MonetDB query compiler has been discussed in Section 4.3 in general, and specific data structures were pointed out on the example of the triangle query $\mathcal{Q}_{\triangle}$. The operator tree in Figure 4.1 illustrates the pairwise evaluation of joins. However, integrating a class of worst-case optimal joins into this translation process needs to replace this pairwise evaluation with a different approach that allows to generate MAL code that features the `Multi-Way-Join` processing behaviour of WCOJ introduced by the `Generic-Join` algorithm 3.1. Furthermore, one important goal of the integration process is to replace the join translation logic, while still supporting the original implementation of any other query language feature. This is desirable, since there will not be any need to adapt their original implementation, which reduces the overall implementation overhead of this work.

The new way of translating joins requires a suitable data structure that represents a query's hypergraph and transports all necessary information needed to translate natural join queries according to the `Generic-Join` algorithm 3.1. Moreover, the new translation logic has to be able to deal with the `stmt` data structure, as described in Section 4.3. Particularly, it has to consume statement tree structures that result from basetable operator nodes and produce a suitable statement structure in order to support any kind of parent operator node.

4. **Generate MAL Code with Loops**
   MAL provides the possibility of looping over every single entry of a BAT by utilizing the control flow operators `BARRIER`, `REDO` and `EXIT` together with the `iterator` module, as shown in Listing 2.7. This language construct is essential for this work, because the `Generic-Join` algorithm 3.1 needs to iterate over each resulting tuple $\boldsymbol{t}_I \in L$ in the results of the first recursive call on partition I and then recursively construct the output for each such tuple $\boldsymbol{t}_I \in L$ with respect to partition J. MonetDB provides different optimizations [Mona] to transform a MAL program into a more efficient version of it, but none of these optimizations requires loops. Therefore, MonetDB's SQL frontend does not provide statements that generate loop specific MAL instructions. To that end, a big challenge was to equip MonetDB with statements that generate loops, as shown in Listing 2.7. Moreover, these MAL constructs need to be integrated in the query compilation process in order to be capable of generating the nested loop structure that results from the recursive calls presented in the `Generic-Join` algorithm 3.1.

5. **Introduce Temporary Variables in MonetDB**
   The construction of output tuples for a given query according to the `Generic-Join` algorithm 3.1 can be obtained in line 9 of the algorithm. The corresponding MAL code version requires multiple nested loops, where each nested loop corresponds to an attribute that is joined when calculating a query's result, as shown in Listing A.3 of Appendix A. This nested loop structure calculates the output tuples by iterating over the distinct attribute values respectively and concatenates attribute values of outer loops with the joined attribute values of inner ones. One crucial aspect about this nested loop structure is, that every single iteration of an outer loop needs to append corresponding intermediate results of the inner loops to BATs. Such intermediate results can be calculated using temporal variables, as shown in line 40 and 41 of Listing A.3 in Appendix A. To that end, the MonetDB query compiler as well as its optimizer need to be extended in order to support temporal variables whose scope belongs to a given loop.

6. **Reuse Variable Identifiers**
   The original MonetDB generates MAL programs where the result of any MAL instruction is assigned to a new variable identifier. This suits its original compilation approach, which does not generate loops. The `Multi-Way-Join` evaluation schema of the `Generic-Join` algorithm 3.1 introduces the need for iterating

over single values of a given BAT. Moreover, each iteration's result has to be appended to a result BAT. Thus, such append operations require the possibility to append results to the same BAT in each iteration and therefore break with MonetDB's assignment policy. This has to be considered during the adaptation of methods that generate specific instructions which need to reassign the result to an existing variable. Moreover, this reassignment behaviour in combination with loops introduce local scopes within the nested loops which needs to be considered during the integration process as well.

4.4.2 Translating the `Generic-Join` Algorithm into MAL Instructions

The aim of this section is to discuss the most suitable and efficient MAL instructions that can be used to translate parts of the `Generic-Join` algorithm 3.1 into. The translation is committed to select MAL instructions that closely correspond to the relational algebra operators utilized in the formulation of the `Generic-Join` algorithm 3.1. Based on this direct translation in combination with the corresponding evaluation from Chapter 5, improvements are introduced and discussed. It is expected that improvements will be required, since algorithms formulated using relational algebra tend to utilize operators which are handy for formalization purposes. The efficiency of such operators in a real system is a completely different concern. Thus, the direct translation approach will include MAL instructions which are not beneficial for the efficiency of the resulting program and need to be replaced with more efficient alternatives (e.g. replace cross product with a more efficient set of MAL instructions).

The following discusses the translation of the `Generic-Join` algorithm 3.1 line by line and presents sequences of MAL instructions that closely correspond to the given line of relational algebra notation. At first, the most direct translation is discussed and afterwards potential improvements are described.

Direct Translation

In the first step of translating the `Generic-Join` algorithm 3.1 into a MAL program, a direct translation approach is carried out. This means, that the pseudocode instructions are translated into the closest corresponding MAL instructions. As already introduced, this can have specific performance implications. Furthermore, code segments that impact the data flow are discussed in order to point out how they are translated even though there is no one to one translation to existing MAL instructions. Listing A.2 in Appendix A shows the MAL program that results from this direct translation.

if $|\mathcal{V}| = 1$

The given line of code depicts the termination condition of the recursive structure of the `Generic-Join` algorithm 3.1. This condition holds whenever there is just a single attribute left in the set of vertices $\mathcal{V}$. The number of attributes in $\mathcal{V}$

is decreasing with recursion depth, since each recursive call of `Generic-Join` further partitions the provided query in two sub-hypergraphs of this given query. Whenever $|\mathcal{V}| = 1$ holds, the execution of one of the two recursive calls with respect to partitions I and J is terminated. The result of this termination is the intersection of any columns over all relations that correspond to the single attribute left in $\mathcal{V}$. Furthermore, the termination condition need not be translated into MAL code, but tells the query compiler to generate the intersection of all columns belonging to this single attribute in the right place.

return $\cap_{F \in \mathcal{E}} R_F$

This return statement is executed whenever the termination condition $|\mathcal{V}| = 1$ holds. Due to the fact that `Generic-Join` contains two recursive function calls, the overall execution can be illustrated using a tree structure. Facing such a tree representation of function calls, the given return statement is called in leave nodes of such tree. The overall structure of this tree and its depth is determined by the applied partition strategy in order to divide the set of vertices $\mathcal{V}$ into two sets I and J.

Besides these control flow aspects that affect the nested loop structure of the resulting MAL program, the intersection of multiple columns across various relations that belong to the single attribute in $\mathcal{V}$ is of importance. Since queries usually join multiple relations on various attributes, the resulting MAL sequence requires cascading the intersection of multiple columns. Moreover, the MAL instruction `algebra.intersect` supports intersecting two BATs and produces BATs of OIDs representing matching positions within the corresponding inputs. This MAL instruction can generate the resulting BATs for the left, right or both input BATs. Since `algebra.intersect` just supports pairwise intersections, the translated MAL code cascades multiple intersection instructions. Moreover, intermediate BATs of OIDs need to be projected in order to obtain an intermediate result that can further be involved in a pairwise intersection with another column. Listing 4.4 shows an example sequence of MAL instructions that intersects three columns. Note that `group.groupdone` needs to be applied to the final result BAT of the cascaded intersections, since individual columns might contain duplicated values that would not be eliminated during intersection and projection.

```
1 interRTA:bat[:oid] := algebra.intersect(projRA:bat[:str],
      projTA:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
      false:bit, nil:lng);
2 projInterRTA:bat[:str] := algebra.projection(interRTA:bat[:oid],
      projRA:bat[:str]);
3 interA:bat[:oid] := algebra.intersect(projUA:bat[:str],
      projInterRTA:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
      false:bit, nil:lng);
4 projInterA:bat[:str] := algebra.projection(interA:bat[:oid],
      projUA:bat[:str]);
```

```
5  (X_111:bat[:oid], C_112:bat[:oid]) :=
       group.groupdone(projInterA:bat[:str]);
6  distA:bat[:str] := algebra.projection(C_112:bat[:oid],
       projInterA:bat[:str]);
```

Listing 4.4: MAL code snippet of intersecting three columns that correspond to attribute A. This snippet can result from translating "return $\cap_{F \in \mathcal{E}} R_F$" into MAL.

Pick I arbitrarily such that $1 \leq |I| < |\mathcal{V}|$

This line of pseudocode aims for partitioning the given hypergraph representation of a query into two sub-hypergraphs $\mathcal{H}_I = (I, \mathcal{E}_I)$ and $\mathcal{H}_J = (J, \mathcal{E}_J)$. Therefore, a subset of vertices I is selected arbitrarily according to the condition $1 \leq |I| < |\mathcal{V}|$ and J is a complementary subset to I such that $\mathcal{V} = I \uplus J$. The applied partitioning strategy impacts the resulting nested loop structure, because recursive calls handle both (sub-)hypergraphs separately. Integrating the `Generic-Join` algorithm 3.1 into a database system in a way that it achieves the best performance would require an optimal partitioning. Finding the most suitable partitioning strategy can depend on the database system itself, on the submitted query and other parameters. Thus, integrating such optimal partitioning strategy requires great effort and is outside the scope of this work.

For that reason, this work utilizes the same partitioning strategy for every query under the assumption that the selected strategy does not have major performance implications. In general, there will be performance differences which cannot be easily estimated upfront. These potential variations are neglected by this work, since the main focus is on the overall integration of WCOJ algorithms. The partitioning strategy applied by this work always picks I such that $|I| = 1$. During partitioning, the vertices of the current query hypergraph are sorted in descending order of their degrees, and the vertex of the highest degree will be selected. Thus, the set I consists of the attribute that appears in the most relations of the current sub-hypergraph. If there are multiple such attributes with the same highest degree, one of them will be arbitrarily selected. Since, $\mathcal{V}$ is partitioned into sets I and J, J contains all remaining vertices except the one with the highest degree.

$L \leftarrow$ Generic-Join $(\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F))$

This line of pseudocode depicts the first recursive call of the `Generic-Join` method. The parameter represents the sub-query related to partition I in relational algebra notation. Each recursive call defines the structure of the resulting MAL code, and the distribution of sub-hypergraphs is controlled by the applied partitioning strategy. The compiled MAL code of this work is always structured the same way, since a partitioning strategy is used that picks $|I| = 1$ as introduced in the previous paragraph. Moreover, the fact that $|I| = 1$ makes the recursive call $L \leftarrow$ Generic-Join $(\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F))$ meets the condition $|\mathcal{V}| = 1$ in any case, because the set I represents the set of vertices $\mathcal{V}$ that corresponds to the given recursive call of the `Generic-Join` function. As a result, each recursive method call taking

the sub-hypergraph related to partition I as parameter will intersect all columns that relate to the single attribute in I. The resulting code is shown in Listing 4.4. Note that when choosing another partitioning strategy that also allows partitions where $|I| > 1$, this recursive call would not immediately translate into cascading intersections in any case. Such settings would generate nested loops for recursive calls on the sub-hypergraph related to partition I.

for every $\mathbf{t}_I \in L$

The given for-loop aims for iterating over each value $\boldsymbol{t}_I$ present in the result L of the previously discussed recursive call on the sub-hypergraph corresponding to partition I. Listing 4.5 depicts how loops can be realized using MAL instructions. A new iterator on set L is defined using `iterator.new` that initially assigns values to variables `tIoid` and `tI`. The `BARRIER` dataflow modifier indicates the start of a loop and the `REDO` dataflow modifier assigns new values to `tIoid` and `tI` respectively and continues execution of the next loop iteration. If there are no further values present, the loop is exited. Furthermore, the value that corresponds to the current iteration can be accessed via the variable `tI`, as given in Listing 4.5. This variable will be utilized for constructing corresponding results by recursively calculating tuples that appear in the output tuple, together with the value referenced by `tI`.

The most important fact to be considered when translating loops is, that the loop's header and footer need to be transformed independently. Therefore, the given line of pseudocode just compiles into line 1 of Listing 4.5. Moreover, lines containing the `REDO` and `EXIT` dataflow modifiers have to be generated at a later point in the translation process. Since the loop block is the last code block in the `Generic-Join` algorithm 3.1, the MAL loop's footer instructions can be generated right before returning from the current function call. Furthermore, each nested loop, except the outermost one in this direct translation, requires allocating a temporal BAT upfront that will be utilized to propagate intermediate results produced by the translation of the cross product $\{\mathbf{t}_I\} \times \mathcal{Q}[\mathbf{t}_I]$ to the encapsulating loop respectively.

```
1  tmp:bat[:str] := bat.new(nil:str);
2  barrier (tIoid:oid, tI:str) := iterator.new(L:bat[:str]);
3    // further (nested) mal code
4    // result reconstruction using tmp bat
5    redo (tIoid:oid, tI:str) := iterator.next(L:bat[:str]);
6  exit (tIoid:oid, tI:str);
```

Listing 4.5: MAL code snippet iterating over all values $\boldsymbol{t}_I$ in L.

$\mathcal{Q}[\mathbf{t}_I] \leftarrow$ Generic-Join $(\bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes \mathbf{t}_I))$

The given line of pseudocode represents the second recursive call, which handles the sub-hypergraph corresponding to partition J. The specified parameter describes the sub-hypergraph belonging to partition J. In comparison to the first recursive call $L \leftarrow$ Generic-Join $(\bowtie_{F \in \mathcal{E}_I} \pi_I(R_F))$, the second one performs a semi-join operation between each relation R_F, where $F \in \mathcal{E}_J$, and the value $\boldsymbol{t}_I \in L$ of the current iteration. Note that partition J results from the chosen partitioning strategy. In the case of this work, J composes all attributes present in $\mathcal{V}$ except the attribute that occurs most frequently across all relations involved. The main challenge that has to be tackled with respect to generation of MAL code is the translation of $\pi_J(R_F \ltimes \mathbf{t}_I)$ for any $F \in \mathcal{E}_J$. This sequence of relational algebra states that the semi-joins of any relation R_F, where $F \in \mathcal{E}_J$ present in the hypergraph, and the value $\boldsymbol{t}_I \in L$ of the current iteration have to be calculated. Furthermore, the projection of attributes in J on the respective results need to be evaluated. Due to the vertical fragmentation of column-stores, projections have to be carried out on each column that will be used in later parts of the generated code separately.

Listing 4.6 depicts an example of how to translate the relational algebra statement $\pi_J(R_F \ltimes \mathbf{t}_I)$ for any $F \in \mathcal{E}_J$ into a sequence of MAL instructions. The example's setting is given by $\mathbf{t}_I = tupleB$, $J = \{A, C\}$ and $\mathcal{E}_J = \{R, S\}$. One obstacle encountered during manual translation of the given line of pseudocode was, that the `algebra` module's operators are applied on BATs, whereas the iterator returns the raw typed value (e.g. a single value of type string). In order to calculate the discussed semi-join $R_F \ltimes \boldsymbol{t}_I$, the raw value $\boldsymbol{t}_I$ needs to be transformed into a BAT that just contains this single entry. This is done by allocating a new BAT and appending the single value to it, as shown in line 1 and 2 of Listing 4.6. Moreover, lines 4 and 5 of Listing 4.6 depict the calculation of $R_S \ltimes t_{\{B\}}$ on the example of the triangle query $\mathcal{Q}_\triangle$. In line 4, the column B of relation S and $t_{\{B\}}$ are semi-joined. Note that the semi-join is replaced with an intersection instruction by MonetDB's optimization pipeline, since intersecting a column with another that just holds a single element is more efficient than performing a semi-join. The result is a list of matching positions for relation S that can further be utilized to project the result of the semi-join to the other attributes of relation S. Such a projection is carried out in line 5 of Listing 4.6. Furthermore, it is important to understand that the semi-join operation aims for shrinking the size of BATs by just selecting values that can occur together with value $\boldsymbol{t}_I$ in an output tuple.

```
1  X_70:bat[:str] := bat.new(nil:str);
2  tupleB:bat[:str] := bat.append(X_70:bat[:str], X_69:str, true:bit);
3
4  X_73:bat[:oid] := algebra.intersect(projSB:bat[:str],
       tupleB:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
       true:bit, nil:lng);
5  semiSC:bat[:str] := algebra.projectionpath(X_73:bat[:oid],
       tidS:bat[:oid], bindSC:bat[:str]);
6  X_79:bat[:oid] := algebra.intersect(projRB:bat[:str],
```

```
        tupleB:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
        true:bit, nil:lng);
7   semiRA:bat[:str] := algebra.projectionpath(X_79:bat[:oid],
        tidR:bat[:oid], bindRA:bat[:str]);
```

Listing 4.6: MAL code snippet to calculate the relational algebra statement $\pi_J(R_F \ltimes \mathbf{t}_I)$ for any $F \in \mathcal{E}_J$ where $\mathbf{t}_I = tupleB$, $J = \{A, C\}$ and $\mathcal{E}_J = \{R, S\}$. This example depicts one iteration of the outermost loop of the compiled triangle query shown in Listing A.2.

The given line of pseudocode represents the core idea behind `Multi-Way-Joins` which is to increase performance by decreasing the number of join candidates. This is realised by the discussed semi-join operation in combination with the recursive structure of the `Generic-Join` algorithm 3.1.

The discussed translation introduces potential performance drawbacks in comparison to the original MonetDB implementation. First, allocating a new BAT at each iteration for wrapping raw values in order to be able to utilize instructions of the `algebra` module introduces additional memory overhead and a certain number of avoidable calls of the `bat.append` instruction. Moreover, one of the intersection's input columns only holds a single value and there might be a more efficient way of obtaining column entries that can occur in an output tuple together with the given value.

$\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\mathbf{t}_I\} \times \mathcal{Q}[\mathbf{t}_I]$

The given line of pseudocode calculates the cross product between a set that just contains the value $\boldsymbol{t}_I$ of the current iteration and the result $\mathcal{Q}[\boldsymbol{t}_I]$ of the recursive call in the previously discussed line of pseudocode. The resulting tuples are added to the result set $\mathcal{Q}$ using the union operator. The major challenge with translating this relational algebra statement is to correctly calculate the columns of the resulting relation throughout the nested loop structure of the generated MAL code. Studying the `Generic-Join` algorithm 3.1 reveals that $\mathcal{Q}[\mathbf{t}_I]$ can be assigned in two different situations which will be handled differently in the final MAL code. The first situation is encountered in leave nodes of the function call tree resulting from the recursive calls to the `Generic-Join` algorithm 3.1. For calls corresponding to leave nodes, it holds that $|\mathcal{V}| = 1$ and therefore $\mathcal{Q}[\mathbf{t}_I] \leftarrow \cap_{F \in \mathcal{E}_I} R_F$. The second situation to be considered occurs whenever a recursive function call returns that corresponds to any intermediate node of the function call tree. The challenge of this case is to properly propagate intermediate results from nested loops to its encapsulating loop in order to simulate the union operator of the given line of pseudocode using a set of MAL instructions.

First, $\mathcal{Q}[\mathbf{t}_I] \leftarrow \cap_{F \in \mathcal{E}_I} R_F$ appears to be a straightforward case, since the conversion of the relational algebra statement $\cap_{F \in \mathcal{E}} R_F$ to MAL instructions has already been discussed. Moreover, the given situation only occurs in function calls where $|\mathcal{V}| = 1$. Moreover, $\cap_{F \in \mathcal{E}} R_F$ has already been translated to MAL code following the

cascading structure introduced in Listing 4.4. Next, the given tuple reconstruction pseudocode has to be converted into a sequence of MAL instructions that calculates the cross product between the result of $\mathcal{Q}[\mathbf{t}_I] \leftarrow \cap_{F \in \mathcal{E}_I} R_F$ and the current iterator value $\boldsymbol{t}_I$. Based on the result of the cross product, the corresponding output BAT of the current loop's attribute, as well as the BAT that propagates the intermediate results belonging to the attribute of the encapsulating loop, need to be updated. Listing A.2 shows a MAL program that calculates the result of the triangle query $\mathcal{Q}_\triangle$ for input relations $R(A,B), S(B,C)$ and $T(A,C)$. The innermost loop of this MAL program is shown in lines 38-58, and it iterates over unique values of attribute C that can occur in an output tuple together with the current iterator value belonging to attribute B. Furthermore, lines 46-49 show the translation of $\cap_{F \in \mathcal{E}_I} R_F$ where $I = \{A\}$ and $\mathcal{E}_I = \{R, T\}$ and the projection of its distinct result is assigned to variable X_111. Lines 51-55 depict the MAL instructions used for tuple reconstruction in the current case. At first, the cross product of X_111 and the tuple of the current iteration for attribute C is evaluated. Then, the result corresponding to attribute A is projected to variable X_114 in line 52. Moreover, `algebra.project` is utilized to project the raw value of the current iteration to a BAT having as many rows as the result of the cross product in line 53. After projecting the resulting columns, these intermediate results are appended to the corresponding BATs in lines 54 and 55. Note that the identifier `atmp` in line 54 is used to propagate the intermediate results for attribute A to the encapsulating loop.

The intersection scenario has been discussed in great detail. The second case, where MAL code needs to deal with tuple reconstruction, occurs when the `Generic-Join` algorithm 3.1 regularly returns set $\mathcal{Q}$ as given in line 11 of the algorithm. This return statement is invoked right after the tuple reconstruction logic of the recursively called `Generic-Join` method. Thus, returning from the second recursive call in line 8 of the `Generic-Join` algorithm 3.1 can result in consecutive MAL code sequences that serve tuple reconstruction and define the current loop's footer before returning from the call. The main challenge to be tackled in this scenario is the fact that the encapsulating loop requires intermediate results of a specific attribute calculated by its nested loop. This propagation of intermediate BATs is realized through the introduction of temporal variables per loop. That way, the scope of such local variables is restricted to the loop it has been defined in. Listing A.2 depicts such temporal variable in line 38. The identifier `atmp` represents a BAT that is assigned a fresh BAT in every iteration of the encapsulating loop. Furthermore, the intermediate results of attribute A are appended to `atmp` for all iterations of the inner loop in line 54. Since the inner loop (line 39-58) iterates over all distinct values in C that can occur in an output tuple with the current value of the encapsulating loop, it produces the result BAT for attribute C in line 55. After the inner loop terminated for a given iteration of the encapsulating one, the cross product of the intermediate BAT `atmp` and the value of the current iteration of the encapsulating loop is calculated. The construction of the output BATs per

attribute follows the schema discussed in the previous scenario. Due to the selected partitioning strategy of this work, tuple reconstruction is carried out in a cascading manner for queries that join multiple relations.

The introduced direct translation of tuple reconstruction might have one massive performance drawback, which is the `algebra.crossproduct` instruction. The cross product is calculated between a BAT with multiple rows and another one with just a single row. In general, the cross product is a very expensive operation and there might be more efficient ways to retrieve the same results without utilizing the `algebra.crossproduct` instruction. Moreover, replacing the cross product with some equivalent, but more efficient logic might require further adaptations of the overall translation of the tuple reconstruction process.

Improved Translation

The previous section discussed a direct translation of the `Generic-Join` algorithm 3.1. In the direct case, MAL instructions are selected that closely correspond to the relational algebra operators used in the `Generic-Join` algorithm 3.1. As already pointed out, this might not be the most efficient implementation of this class of WCOJ algorithms for MonetDB. Chapter 5 evaluates the direct translation in comparison to the performance of the original MonetDB system and points out that improvements over the presented direct translation are required. Therefore, this section investigates and discusses potential performance bottlenecks of the direct translation. Furthermore, alternative ways of translating certain lines of the `Generic-Join` algorithm 3.1 are introduced and outlined in detail.

$\mathcal{Q}[\mathrm{t}_I] \leftarrow$ Generic-Join $(\bowtie_{F \in \mathcal{E}_J} \pi_J(R_F \ltimes \mathrm{t}_I))$

This line of pseudocode introduces a set of MAL instructions that evaluates the given semi-join operation. This code snippet is shown in Listing 4.6 and utilizes the `algebra.intersect` instruction instead of calculating a semi-join, since it is an equivalent and more efficient solution. Interpreting the semi-join $R_F \ltimes \mathbf{t}_I$ according to the column-oriented database system paradigm reveals, that the semi-join operator can be realized using a combination of selection and projection. Due to the vertical fragmentation of column-stores, the intersection operation utilized in the direct translation calculates the intersection between the single value BAT representing $\boldsymbol{t}_I$ and the corresponding column of the given relation R_F. Moreover, the resulting BAT represents a list of positions, which is further used to project every other attribute of the same relation that is required in following parts of the translated code. Thus, projections of attributes are required in any case, and improvements can focus on replacing the semi-join with a simple selection. The MonetDB's `algebra` module provides selection logic through the `thetaselect` instruction, which takes a column, a single value and a selection operator as input. The column to be filtered needs to be defined by its table identifier and the bound attribute itself. Moreover, the types of the single selection value and

the BAT to be filtered must be the same and the selection operator is equality, because matching values need to be selected. Such matching values reference other attributes of the given relation that can occur in an output tuple with the given selection value. Listing 4.7 shows a MAL code example that is equivalent to the one provided in Listing 4.6, but improves upon the later one's direct translation. Two advantages are introduced by be improved version given in Listing 4.7. First, the `algebra.intersect` instruction is replaced with `algebra.thetaselect`, which might introduce slight performance improvements. More importantly, this replacement eliminates the need for wrapping the raw value of the current iteration into a temporal BAT. Getting rid of such temporal BATs will reduce the memory footprint of the translated MAL program and simplify the overall code as well.

```
1  X_75:bat[:oid] := algebra.thetaselect(bindSB:bat[:str],
       tidS:bat[:oid], rawValueB:str, "==":str);
2  semiSC:bat[:str] := algebra.projectionpath(X_75:bat[:oid],
       tidS:bat[:oid], bindSC:bat[:str]);
3  X_78:bat[:oid] := algebra.thetaselect(bindRB:bat[:str],
       tidR:bat[:oid], rawValueB:str, "==":str);
4  semiRA:bat[:str] := algebra.projectionpath(X_78:bat[:oid],
       tidR:bat[:oid], bindRA:bat[:str]);
```

Listing 4.7: MAL code snippet presenting the improved way of translating the semi-join $R_F \ltimes \boldsymbol{t}_I$. Improvements are achieved by replacing the need for wrapping a single raw value into a temporal BAT in order to be able to use `algebra.intersect` with utilizing the `algebra.thetaselect` instruction.

$\mathcal{Q} \leftarrow \mathcal{Q} \cup \{\mathbf{t}_I\} \times \mathcal{Q}[\mathbf{t}_I]$

The direct translation of tuple reconstruction has been discussed in great detail, and it has been pointed out that it is a very demanding challenge to be tackled in the overall translation process. The central instruction when performing a direct translation is the cross product. Moreover, the cross product is a very heavy and performance critical operation, which should be avoided during the integration of the `Generic-Join` algorithm 3.1 into the MonetDB system. To that end, the improved translation aims for replacing `algebra.crossproduct` with an equivalent set of MAL instructions and further adapts the column crafting process as well. In case of the `Generic-Join` algorithm 3.1, utilizing the relational algebra's cross product operator suits the algorithm's general formulation. However, leveraging the cross product implementation of a column-store to combine an arbitrary column with another one that consists of a single value might introduce performance shortcomings. In order to find the most suitable replacement, it is important to understand the outcome of the relational algebra statement $\{\mathbf{t}_I\} \times \mathcal{Q}[\mathbf{t}_I]$ in the context of MonetDB. The output of the `algebra.crossproduct` are two position lists of the same length. The first one represents all positions in the column that are related to the result $\mathcal{Q}[\mathbf{t}_I]$ of the second recursive invocation of `Generic-Join`. The second resulting BAT of OIDs is of the same length as the previous one and represents the

same positions as the previous list, which $\boldsymbol{t}_I$ needs to be projected to. Thus, both mentioned position lists are of the same shape and represent the same positions. Thus, a proper replacement of `algebra.crossproduct` is connected with no additional effort, because the required BAT of OIDs has already been calculated.

A simple observation made when inspecting the direct translation of the triangle query $\mathcal{Q}_\triangle$ in Listing A.2 leads to the insight, that there is no need to evaluate the cross product, since the required position list has already been calculated in line 48 for example. The variable `C_109` is used to obtain the distinct values of the current attribute utilizing the `algebra.projection` instruction. Moreover, `C_109` defines the shape of the cross product's result and can therefore be used to project the intermediate BAT to the single raw value that corresponds to $\boldsymbol{t}_I$. To that end, the `algebra.project` instruction can be utilized to project this single raw value to the position list given by `C_109`. This improvement eliminates the need for evaluating the cross product at no additional costs, since it has been kind of redundant.

Due to the elimination of the `algebra.crossproduct` instruction, the translation of the tuple reconstruction needs to be adapted as well. The cross product within each loop served to obtain the shapes of the intermediate results. Since it has been eliminated, the corresponding shape information needs to be propagated from nested loops to its encapsulating one. To achieve such propagation, another temporal variable has to be introduced per nested loop. Note that the outermost loop does not require any temporal variables, since there is no information to propagate further. Listing A.3 depicts such additional temporal identifier in line 41, for instance. The variable `tmpBCand` represents a BAT of OIDs which is utilized to append the intermediate shapes. The length of such temporal BAT can be interpreted as the number of tuples, which the current iterator value of attribute B appears in. Propagation is important, since the encapsulating loop is the only one being able to perform the according projection of its iterator value properly. Line 60 in Listing A.3 shows such projection, and its result is appended to the overall output BAT for attribute B in line 63. Note that the propagation of intermediate result BATs remains the same compared to the previously discussed direct translation approach.

To conclude the manual translation process of the `Generic-Join` algorithm 3.1 into a MAL program, it has to be underlined that following a direct translation approach turned out to be very helpful. It improved the basic understanding of MAL, the MonetDB query compiler and the `Generic-Join` algorithm 3.1. Based on the direct translation and gained knowledge, problematic areas in the translated code could be identified and analyzed. Moreover, the improved translation describes concepts that drastically improve performance, as evaluated in Chapter 5.

4.4.3 Making MonetDB Generate Worst-Case Optimal MAL Code

The previous section discussed the translation of the `Generic-Join` algorithm 3.1 into a MAL program. The aim of this section is to outline the integration of this class of WCOJ algorithms into MonetDB's query compiler based on the results of the manual translation. Therefore, this section discusses adaptations that have been made to MonetDB's query compiler, which has been introduced in Section 4.3. Moreover, it is pointed out how the `Generic-Join` algorithm 3.1 can be integrated in MonetDB while still natively supporting any other features out of the box. Additionally, the main data structures enabling the translation of joins into MAL code, that evaluates joins in a multi-way manner, are introduced.

Code Structure

The goal of the integration is to replace MonetDB's way of evaluating joins with the semantic given by the `Generic-Join` algorithm 3.1. Due to the fact, that MonetDB performs system queries at start up time and that the integration carried out during this work is experimental, the original query compiler code should not be altered with. That way, updates can be carried out using the original query compiler and user submitted queries can be evaluated using the adapted one. Moreover, this possibility of switching between the two ways of treating joins will be needed, because this work just focuses on natural join queries. To use different approaches for system and user submitted queries, a switch has been implemented in the `sql_relation2stmt` function in the `sql_gencode.c` source file. Listing 4.8 shows the corresponding code snippet. The depicted function is invoked with parameter `nme` set when system queries are executed, and therefore the presence of such a value indicates the use of MonetDB's original query translation approach. Furthermore, lines 7-11 of Listing 4.8 show an if-else construct using preprocessor expressions. If the flag `_WCOJ_` has been defined, a newly compiled MonetDB system will utilize the WCOJ approach in its `mserver5` program. Otherwise, the original MonetDB compiler will be used. This empowers to easily switch between the two implementations for testing and benchmarking purposes.

```
1  static stmt * sql_relation2stmt(backend *be, sql_rel *r, const char* nme)
2  {
3    ...
4    if(nme != NULL) {
5      s = output_rel_bin(be, r);
6    } else {
7      #ifdef _WCOJ_
8        s = output_rel_bin_wcoj(be, r);
9      #else
10       s = output_rel_bin(be, r);
11     #endif
12   }
13   ...
```

```
14 }
```

Listing 4.8: C code snippet that is used to translate system queries using MonetDB's original query compiler. Moreover, a preprocessor flag is used to compile the MonetDB column-store with the WCOJ or the original approach.

MonetDB's main query compiler logic is centred in the `rel_bin.c` source file. Roughly speaking, the translation dataflow logic is defined by various functions that reside in this file. In order to generate MAL instructions, methods provided by the `sql_statement.c` source file are leveraged. Since the integration aims for replacing the translation logic of joins while still supporting the translation of any other SQL feature, the `rel_bin.c` source file is copied and renamed to `rel_bin_wcoj.c`. The WCOJ processing approach given by the `Generic-Join` algorithm 3.1 is implemented in that source file and replaces MonetDB's pairwise join evaluation. The entry points to the query translation code are given by the methods `output_rel_bin` and `output_rel_bin_wcoj` in line 10 and 8 of Listing 4.8 respectively.

Integration using Hypergraph Representation of Queries

The idea on how and where the `Generic-Join` algorithm 3.1 can be suitably integrated into MonetDB has been outlined in the previous section. The aim of this section is to discuss the key aspects of the adapted query compiler, which empowers the translation of joins following the multi-way join approach of WCOJ algorithms. Due to the fact that the `Generic-Join` algorithm 3.1 operates on the hypergraph representation of a given query, the adapted query compiler will operate on a very similar data structure as well. This enables this work to stick closely to the dataflow given by the `Generic-Join` algorithm 3.1. Based on this code structure, the translation logic has to be implemented to generate MAL programs that follow the translations discussed in Section 4.4.2. In the following, the key data structures, as well as the main aspects of the adapted compiler implementation are outlined.

Listing 4.9 depicts the data structures used to represent a given query as hypergraph and further store additional information that is required to translate joins according to the `Generic-Join` algorithm 3.1. The parent data structure is the `Hypergraph` struct, that aims for representing the query as sets of vertices and edges. Therefore, the `list` pointers `vertices` and `edges` represent these sets as lists of type `Vertice` and `Edge` respectively. Note that MonetDB's way of handling single linked lists is reused in this implementation, as given in lines 4 and 5 of Listing 4.9. Moreover, the number of vertices and edges is stored respectively. Besides that, a hypergraph stores a pointer to the originally parsed query of type `sql_rel` in the `original_rel` property. This is not required, but used to be handy for testing purposes. Finally, the `partition` flag indicates whether the hypergraph represents partition I or J which is an important dataflow information.

The `Vertice` struct is mainly composed of `stmt` pointers utilized for tuple reconstruction. Each vertex of a hypergraph belongs to a specific attribute, as shown in

the example in Figure 3.1. Therefore, the `cname` property refers to the corresponding attribute name. The `list` pointer `attributes` represents a list of type `Attribute`, where each list element represents a column corresponding to the attribute of the current vertex across various relations. This list can be very useful for intersecting all columns belonging to the same attribute, for instance. Furthermore, the `result_stmt` field represents the statement corresponding to the result BAT for a given attribute. The `tmp_result` property relates to the statement that corresponds to intermediate result BAT of a nested loop, as introduced in Section 4.4.2. Similarly, the `tmp_candidate` field relates to the statement that represents the intermediate position BAT used to reconstruct columns for attributes belonging to encapsulating loops.

Moreover, the `Attribute` struct represents a given column of a table and adds additional `stmt` properties that relate to certain translations, which the current attribute has been involved in. Each such attribute is identified by the column name `cname` and the table name `tname`. Moreover, the `bat_type` field specifies the datatype of the corresponding BAT. This information is required to generate MAL code that creates a new temporal BAT. Note that the implementation of this work only supports string types, since this is sufficient for an experimental integration. However, the `bat_type` property could be used to support arbitrary datatypes. The two `stmt` pointer fields have similar purpose as the ones used in the `Vertice` struct, which is to indicate certain usages of the attribute during the translation process. To that end, the `current_stmt` field represents the latest translation the current attribute has been involved in. Moreover, the `intersection_stmt` property represents the statement returned from intersecting multiple attributes of column name `cname`.

Finally, the `Edge` struct represents the relations involved in the natural join query. Each edge can be identified by its corresponding table name `tname`. Furthermore, a `list` pointer `attributes` is maintained that composes all attributes of the current relation. This list of attributes has similar purposes as the one of the `Vertice` struct, but there are situations where retrieving all attributes of a given relation is beneficial. Moreover, the `ref` property relates to the operator node of type `op_basetable` that corresponds to the given relation.

```
1  typedef struct
2  {
3    sql_rel *original_rel;
4    list *vertices;
5    list *edges;
6    int nr_vertices;
7    int nr_edges;
8    int partition;
9  } Hypergraph;
10
11 typedef struct
12 {
13   const char *tname;
14   const char *cname;
```

```
15    sql_subtype bat_type;
16    stmt *current_stmt;
17    stmt *intersection_stmt;
18  } Attribute;
19
20  typedef struct
21  {
22    const char *cname;
23    list *attributes;
24    stmt *tmp_result;
25    stmt *result_stmt;
26    stmt *tmp_candidate;
27  } Vertice;
28
29  typedef struct
30  {
31    const char *tname;
32    list *attributes;
33    sql_rel *rel;
34  } Edge;
```

Listing 4.9: Data structures to represent a SQL query through its hypergraph. Note that these data structures further posses properties that aim for propagating specific information through the translation process.

The previous introduction of the key data structures used for representing queries as hypergraphs within the adapted query compiler underlines, that there are properties that are required directly for generating MAL code and others represent dataflow characteristics. This differentiation is also done in this work by separating the discussion of translation and integration processes. The advantage of this approach is that the integration benefits from the translation process, since the query compiler can be adapted having the full knowledge about the resulting MAL code. Therefore, the integration process can focus on dataflow properties and introduce features which are not supported by MonetDB yet.

One important detail about the integration process is the conversion of MonetDB's internal query representation into the hypergraph structure given in Listing 4.9. This conversion needs to be carried out right before translating the joins to MAL code according to the `Generic-Join` algorithm 3.1. The new join translation logic is entered when the original query translation logic traverses the operator tree and reaches the node of type `op_join` closest to the operator tree's root node. Figure 4.4 shows a sub operator tree of the one depicted in Figure 4.1 representing the two pairwise joins of the triangle query $\mathcal{Q}_{\triangle}$. Translation of the sub operator tree is done recursively and the leave nodes of type `op_basetable` are converted into the corresponding sets of `Vertice`, `Edge` and `Attribute`, as shown in Listing 4.9. After recursively generating these sets, the overall `Hypergraph` struct is created and returned.

After discussing the translation of the `Generic-Join` algorithm 3.1 to MAL code in Section 4.4.2 and introducing the `Hypergraph` data structure used in the adapted

MonetDB query compiler, all parts required for integrating the class of WCOJ algorithms have been established. Due to the fact that the `sql_statement.c` source file already implements functions that aim for generating the MAL instructions required by this work, these methods mainly have to be invoked appropriately in order to produce MAL programs equivalent to the `Generic-Join` algorithm 3.1. The newly introduced translation logic is centred around a recursive function called `rel2bin_generic_join` that takes a pointer to MonetDB's `backend` data structure, as well as a pointer to the converted `Hypergraph` as parameters. The earlier data structure composes further data structures required for the compilation process, such as the `mvc` that transports context and holds properties for interacting with the memory. Such property are fields of type `sql_allocator` which can be utilized to allocate memory for internal data structures. The `Hypergraph` parameter represents the (sub-)hypergraph that is translated at the current level of recursion. The first invocation provides the hypergraph of the overall parsed query, and every recursive call is done providing the sub-hypergraph corresponding to partition I or J. Since the recursive structure of the `Generic-Join` algorithm 3.1 is suitable for implementing logic that compiles given queries into MAL programs as introduced in Section 4.4.2, the control flow of `rel2bin_generic_join` closely matches the one of the `Generic-Join` algorithm 3.1. This circumstance combined with the already discussed manual translation and the `Hypergraph` data structure led to a straightforward implementation that empowers MonetDB to compile MAL programs that follow the discussed class of WCOJ algorithms.

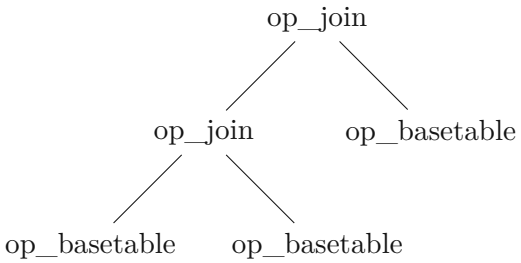


Figure 4.4: Operator sub-tree of the triangle query $\mathcal{Q}_\triangle$ that represents the pairwise join operations and which needs to be translated into a hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$.

Further Adaptations

Unfortunately, MonetDB's original query compiler does not support all features required to enable the generation of MAL programs according to the `Generic-Join` algorithm 3.1. Further adaptations made to MonetDB's optimizer and methods that aim for producing MAL code are outlined in the following.

One important feature introduced during the integration of the `Generic-Join` algorithm 3.1 into MonetDB's query compiler is the support of temporal identifiers. Temporal variables are utilized to assign newly created BATs that are a key component of the tuple reconstruction logic in the final MAL code. The requirement for such

a feature arose from the nested loop pattern that results from the `Generic-Join` algorithm 3.1. Propagating intermediate results from nested loops to the encapsulating ones requires such temporal BATs. This feature has not been supported yet, since the original MonetDB query compiler was not generating loops at all. Moreover, every MAL instruction's result is assigned to a new global variable. Fortunately, temporal BATs could be integrated easily, because the `sql_statement.c` source file provides methods to introduce empty BATs. The original optimization pipeline's empty bindings optimizer recognized such temporal BATs as empty bindings and deletes them during the optimization process. In order to allow temporal BATs that correspond to nested loops, the empty bindings optimizer defined in the `opt_emptybind.c` has been extended to keep temporal variables that are defined right before a loop header.

Besides temporal variables, the possibility to generate loops in MAL code has to be established. Moreover, loop header and footer need to be generated separately but still share the same variables. Therefore, the `sql_statement.c` source file has been extended with two methods `stmt_iterate_list` and `stmt_end_iterate_list` that take a `backend` as well as a `stmt` parameter. The later one defines the statement that corresponds to the set L in line 6 of the `Generic-Join` algorithm 3.1 in case of the `stmt_iterate_list` method which creates the loop header. The footer is generated by `stmt_end_iterate_list` and its `stmt` parameter represents the resulting statement of the loop header creation. That way, the footer is able to leverage the same t and h variable and the overall loop will be translated correctly.

Another required feature for tuple reconstruction in MAL breaks with MonetDB's policy of assigning every instruction's result to a new identifier. Collecting intermediate results of nested loops requires the possibility to append partial results per loop iteration to the same temporal BAT. Therefore, the target variable of the newly created identifier needs to be overwritten by the corresponding existing one. This can be easily achieved by calling `pushArgument` in the corresponding method of the `sql_statement.c` source file with the existing identifier's number. This overwrites the newly created instruction's identifier in the `MalBlkPtr` data structure with an existing number. Moreover, the resulting `stmt` instance needs to reflect these changes as well.

To conclude, the major adaptations of MonetDB's translation logic could be established well with deep knowledge about MonetDB's source code, as well as a deep understanding of the `Generic-Join` algorithm 3.1.

CHAPTER 5

Experimental Evaluation

This section aims for discussing the results obtained from benchmarking the MonetDB instance with integrated `Generic-Join` algorithm 3.1. Upfront of the actual experimental evaluation, the overall benchmarking setting is introduced. This includes the hardware on which the benchmarks were carried out, as well as the queries and databases the systems were tested against. First, the adapted MonetDB system with direct translation integrated was evaluated and compared to the running times of the original system. Based on these results, potentially problematic MAL constructs have been revisited, and the direct translation has been improved. Furthermore, the resulting MonetDB with optimized translation integrated was evaluated against the original MonetDB. Besides investigating the running times for various input relation sizes, the asymptotic runtime has been graphically evaluated by comparing the results using log-log plots. Moreover, the impact of different degrees of skew in the data has been explored.

5.1 Methodology

The experimental evaluation of the three different MonetDB instances has been carried out by running the same evaluation framework for each system separately. The three different instances are the original MonetDB system, MonetDB with direct translation and MonetDB with optimized translation integrated. These systems are tested against three different natural join queries with increasing number of rows per table on each run in order to measure the runtime behaviour of each system when answering the given queries on varying input relation sizes. The results of the run of MonetDB with direct translation, as well as the one with optimized translation are compared to the original MonetDB's results to obtain their potential performance. Moreover, the results of the system with optimized translation are analyzed in more detail to evaluate if the measured runtime meets the asymptotic boundaries discussed by Ngo et al. [NRR13]. Furthermore, the benchmark investigates if the MonetDB with optimized translation can even handle input

relation sizes which cause memory errors for the original MonetDB system. All these benchmarks aim for obtaining the system's performance with increasing number of rows in the input relations. Moreover, the evaluation of MonetDB with optimized translation results from averaging three independent benchmark runs. Another interesting insight to gain, is the running time behaviour over increasing fan-out in the join attributes of a given query. The term fan-out relates to the number of values that occur in a tuple together with all other values. A fan-out of 1 for relation R of the triangle query $\mathcal{Q}_{\triangle}$ means that values $a0$ and $b0$ occur in tuples with all values $a0...m$ and $b0....m$ respectively.

The benchmarking framework used to evaluate the adapted MonetDB systems is shown in Figure 5.1. The `benchmark.sh` file is a parametrizable shell script that measures runtime in milliseconds and is the core component of the benchmarking framework. It interacts with a running MonetDB system's database instance using the `mclient` program and utilizes the `gendb.py` script to generate the relations for a given run. The `gendb.py` can be configured using three arguments. The first indicates the query for which the relations need to be created, the `m` argument specifies the number of rows to be generated and `n` is utilized to specify the fan-out within the created relations. The `mclient` program is called, after the relations for the given run have been generated in order to connect to the running database and insert the created data into the running instance. Note that the `monetdbd` program of the MonetDB system to be benchmarked needs to start the corresponding `dbfarm`. Moreover, the `monetdb` program has to be called manually to start the database instance used for benchmarking. The set of queries submitted during the benchmarking process is defined in the `benchmark.sh` file and provided to `gendb.py` to generate the corresponding relations to be queried. Furthermore, the `benchmark.sh` script can be configured using the three parameters `m`, `n` and `f`. Parameter `m` is an integer array that corresponds to the number of rows of each input relation and is looped through to benchmark the current MonetDB system on all these input sizes. The parameter `n` represents the number of runs per query and value in `m` and can be used to perform benchmarks multiple times for greater reliability. Furthermore, parameter `f` specifies the fan-out used during the database generation using `gendb.py`. Finally, each benchmark run produces a log output file and a csv file with structured results that are suitable for plotting.

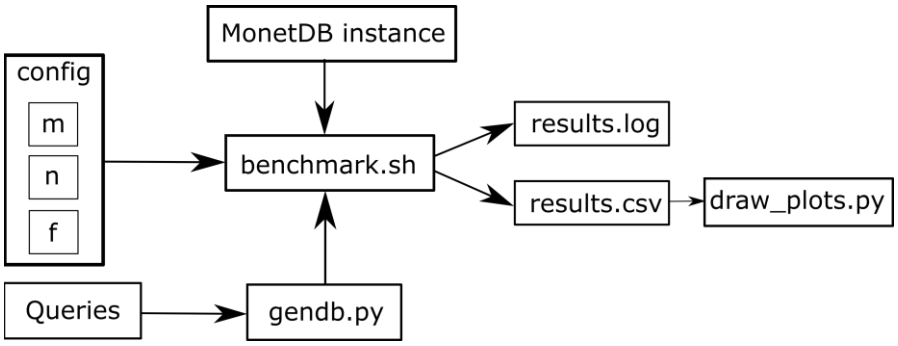


Figure 5.1: The benchmarking framework used for the experimental evaluation.

5.1.1 Hardware

The benchmarks discussed in this chapter were run on a desktop computer with a 64-bit Ubuntu 20.04.3 LTS OS installed with kernel version 5.11.0-41-generic. The machine is equipped with an AMD Ryzen 9 3900x processor with 12 CPU cores, 24 threads, a base clock of 3.8GHz and a 64MB L3 cache. The available memory of the machine is 32GB DDR4 and its disk capacity is 1TB.

5.1.2 Data

The benchmarks presented in this chapter are evaluating the corresponding MonetDB system's performance on three different natural join queries. Data is generated using the `gendb.py` utility that is able to create databases as csv files for the three supported queries. This csv file can be loaded into a database running within MonetDB using the `mclient` program. The three queries used for evaluation purposes are:

select count(*) from r natural join s natural join t;
This query is evaluated on a database with relations $R(A, B)$, $S(B, C)$ and $T(A, C)$. The given query is further referred to as `go3`.

select count(*) from r natural join s natural join t natural join u;
This query is evaluated on a database with relations $R(A, B, C)$, $S(B, C, D)$, $T(A, C, D)$ and $U(A, B, D)$. The given query is further referred to as `go4`.

select count(*) from r natural join s natural join t natural join u natural join v natural join w;
This query is evaluated on a database with relations $R(A, B, C, D, E)$, $S(B, C, D, E, F)$, $T(A, C, D, E, F)$, $U(A, B, D, E, F)$, $V(A, B, C, E, F)$ and $W(A, B, C, D, F)$. The given query is further referred to as `go6`.

Note that the attributes are distributed across multiple relations in a way that each relation consists of n attributes and any two arbitrary relations posses $n - 1$ overlapping attributes. This means that a natural join between two relations is done on $n - 1$ join attributes. Furthermore, each attribute is of type string, since the `gendb.py` utility generates string values according to the specified number of rows `m` and fan-out `n`. The resulting values are a concatenation of attribute name and a number (e.g. $a0$, $c123$, etc.).

The databases generated during the evaluation follow the schema introduced in Figure 3.3. This figure depicts a database instance for the triangle query $\mathcal{Q}_\triangle$ with $m = 4$ and fan-out $n = 1$, since $a0$, $b0$ and $c0$ occur in a tuple with each other value respectively. Furthermore, the evaluation is done on synthetically generated data instead of standard benchmarks. The main reason for that is the fact that WCOJ algorithms promise to outperform traditional pairwise join evaluation the most for database instances such as the one illustrated in Figure 3.3. In such settings, there are relations that are composed of attributes with low entropy, as introduced in the previous example. Database instances

that are most beneficial for WCOJ algorithms have been selected in order to show the potential performance improvements over traditional join evaluation. Furthermore, one could think of domains and settings in practice, where data is similarly structured as the synthetic data in this experimental evaluation.

5.1.3 Configuration

The configurations used for benchmarking the three different MonetDB variants are briefly outlined in this section. The input relation sizes used to compare MonetDB with direct translation to the baseline are given in column `m` of Table 5.1. Due to the fact that the MonetDB with direct translation performs worse than the original MonetDB system, the evaluation has been stopped at $m = 15000$. Likewise, the input relation sizes used for benchmarking MonetDB with optimized translation are shown in column `m` of Table 5.2. The array `m` of input relation sizes to be evaluated is specified in the `benchmark.sh` script. Furthermore, Listing 5.1 depicts the command executed to run the benchmark and shows the provided arguments. Note that the MonetDB instance to be evaluated needs to be started upfront. Moreover, all benchmarks that are not related to fan-out have been carried out with a fan-out of 1.

```
1  ./benchmark.sh -n 3 -f 1 -o results/monetdb_classic.csv
```

Listing 5.1: Command to start the benchmark for an already running MonetDB instance.

5.2 Discussion of Results

This section aims for presenting the results of the experimental evaluation. The baseline for this evaluation is the performance of the original MonetDB system as released in October 2020 and forked for this work in March 2021. The data used for benchmarking has already been described in the previous section. At first, a comparison of the MonetDB with direct translation integrated to the original MonetDB system is carried out. Based on the evaluation of the direct translation, potential improvements have been applied to the translation process that finally led to the optimized translation. This optimized translation is evaluated more extensively throughout this chapter, due to its potential of outperforming the original MonetDB system on the benchmark data for big relations. Moreover, this section compares the asymptotic running times of the MonetDB with optimized translation to the baseline system. Finally, the evaluation aims for exploring the impact of skew in the data by investigating the given set of queries for data with increasing fan-out.

5.2.1 Results of Direct Translation

The results of a single benchmark run on the original MonetDB system and the MonetDB with direct translation integrated are compared in Table 5.1 for queries `go3`, `go4` and

go6. Moreover, the results are plotted in Figure 5.2, where each plot corresponds to one of the given queries. A single benchmark run has been sufficient to obtain, that this work requires to improve upon the direct translation approach. Investigating the sub-plots from Figure 5.2 reveals that MonetDB with direct translation (`MonetDB_wcoj`) performs worse than the original MonetDB system (`MonetDB_classic`) even for small input relation sizes. Furthermore, the original MonetDB outperforms the direct translation approach the most for query `go3` which joins the least number of relations. The runtime gap between both systems is the smallest for query `go6`. Multiple improvements upon this translation have been discussed in Section 4.4.2 and it has been underlined that the `algebra.crossproduct` operation can be identified as the key factor responsible for the worse performance. Calling this cross product operator within nested loops causes multiple invocations of this computationally expensive operator. Moreover, inspecting Table 5.1 reveals that answering queries using MonetDB with direct translation can be two up to five times worse for the depicted values of `m`. It is obvious that these results are not satisfying and that this work needs to improve the translated MAL programs in order to be able to outperform the original MonetDB system. Therefore, no further evaluation of the direct translation approach has been performed, and the focus has been shifted to improve upon this direct approach.

m	**Classic**			**WCOJ**		
	go3 (s)	go4 (s)	go6 (s)	go3 (s)	go4 (s)	go6 (s)
10	0,011	0,024	0,041	0,011	0,036	0,066
100	0,012	0,025	0,042	0,023	0,049	0,088
500	0,03	0,059	0,089	0,095	0,163	0,279
1000	0,05	0,097	0,182	0,222	0,369	0,615
2000	0,108	0,254	1,791	0,52	0,992	1,676
5000	0,613	1,387	2,95	2,956	5,146	8,138
10000	2,219	5,614	11,431	10,26	18,872	30,642
15000	5,108	12,688	26,667	21,783	37,345	61,32

Table 5.1: Table representation of the comparison of running times of the original MonetDB system (Classic) and the MonetDB with direct translation (WCOJ). The displayed numbers result from a single benchmark run.

5.2.2 Results of Optimized Translation

The optimizations introduced in order to improve upon the results of the MonetDB with direct translation integrated have already been discussed in Section 4.4.2. Since these improvements lead to an efficient system, most evaluation has been carried out on this variant with optimized translation. The results presented in the following have been obtained by averaging the outcomes of three independent runs of the system. The

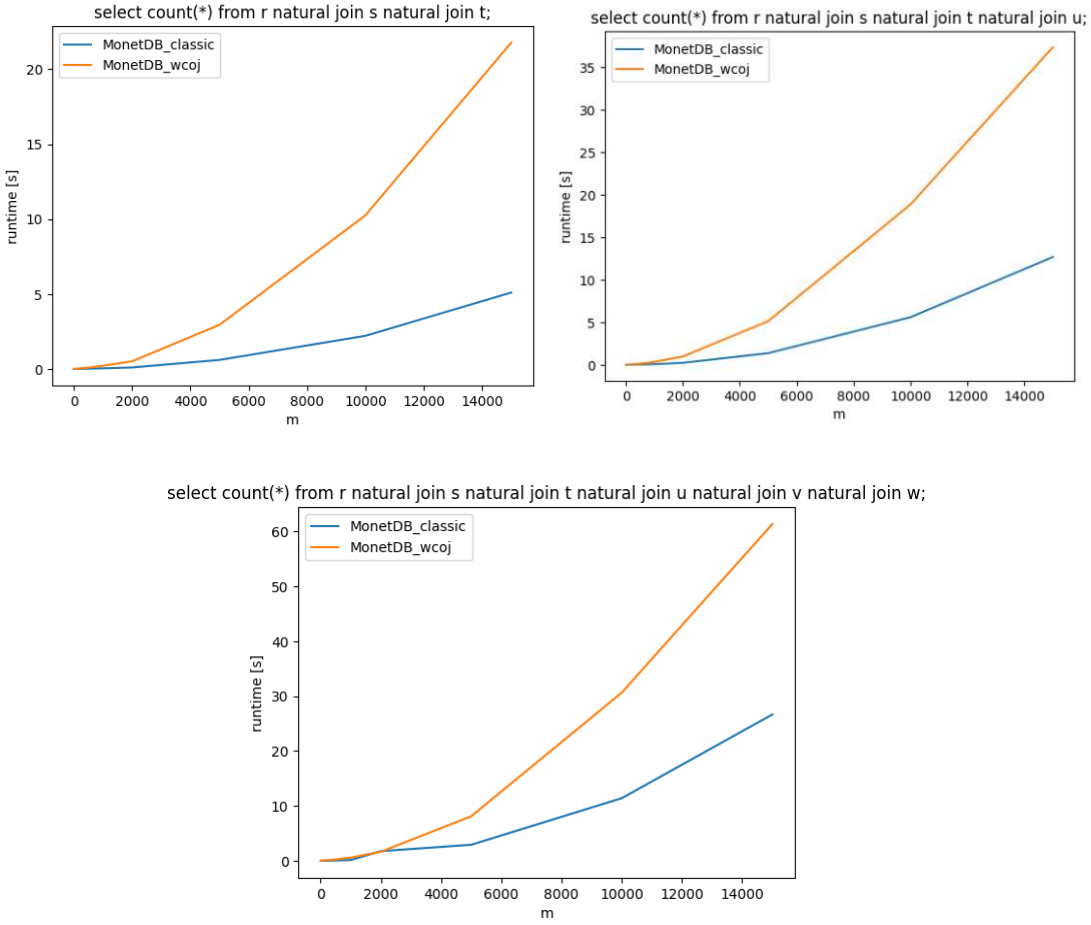


Figure 5.2: Visualization of benchmark results of the three queries go3, go4 and go6 for the original MonetDB vs. the direct translation approach. Note that these plots correspond to the values depicted in Table 5.1.

idea behind this approach is to reduce variation between independent runs. One specific situation that benefits from this approach is, when the original MonetDB system allocates almost all available memory for BATs. It has been observed that measured running times greatly vary in such situations. Table 5.2 shows a comparison of running times of the original MonetDB system and MonetDB with optimized translation integrated. The table depicts results for answering queries `go3`, `go4` and `go6` across a range of input relation sizes given by column `m`. Note that the run with $m = 90000$ and query `go4` on the original MonetDB system led to a situation, in which MonetDB consumed almost all available memory, which results in big deviations between independent runs. Furthermore, more memory intense runs failed with memory errors, which is represented in Table 5.2 by `MEM-ERR` entries. The original MonetDB system fails in such settings

before trying to evaluate the given query, because there is not enough memory for allocating the required BATs. Moreover, it is quite notable that the MonetDB with optimized translation can successfully answer queries for which the original system fails with a memory error. Therefore, the first achievement of integrating a class of WCOJ algorithms into MonetDB is a system that handles memory intense queries in a more memory friendly and efficient way. This conclusion has been made based on the benchmark results from Table 5.2. To gain deeper insights, future work could monitor the memory footprint during benchmarking the different MonetDB instances.

Table 5.2 further underlines that the MonetDB with optimized translation clearly outperforms the original system for large input relations. Both systems have achieved very similar runtime for input relations of size $m \leq 15000$ and the original MonetDB system even outperforms the WCOJ approach a bit. Moreover, for input relation sizes of $m \geq 30000$ the MonetDB with optimized translation starts to benefit from the `Multi-Way-Join` processing approach and significantly outperforms the baseline. For instance, the original MonetDB requires 26 minutes and 23 seconds to answer query `go6` for $m = 90000$ whereas the MonetDB with optimized translation only needs 10 minutes and 12 seconds. The results depicted in Table 5.2 are plotted in Figure 5.3. The subplots correspond to the queries `go3`, `go4` and `go6`. These plots reveal, that the MonetDB with optimized translation (`monetdb_wcoj_optimized`) performs asymptotically better than the baseline (`monetdb_classic`). The plots in Figure 5.3 represent the expected result when starting to integrate the `Generic-Join` algorithm 3.1 into MonetDB. The main achievement of this work is the establishment of an adapted MonetDB system that integrates a class of WCOJ algorithms and asymptotically outperforms the original system that traditionally evaluates join in a pairwise manner. Note that the increasing steepness at the end of curve `monetdb_classic` for query `go4` is caused by the original MonetDB allocating almost all available memory. Operating at the edge of available memory leads to strong runtime deviations as observed during the benchmarks.

In order to visualize the asymptotic behaviour of the original MonetDB and the MonetDB with optimized translation, Figure 5.4 and Figure 5.5 represent log-log plots per query that enable to graphically compare the results to $m^{\frac{3}{2}}$ and m^2. The earlier figure represents the results over the full range of values `m` whereas the later figure cuts off values $m < 10000$ in order to focus on the asymptotic behaviour of big input relations. This representation uses the natural logarithm. Figure 5.4 clearly shows that both systems perform quite well on smaller sizes of input relations and that their asymptotic behaviour can be estimated somewhere around m^2 or higher. Figure 5.5 just plots logarithmic values for $m \geq 10000$ in order to get better insights in this upper region of the plots in Figure 5.4. This area of big input relation sizes is very interesting with respect to asymptotic behaviour of the MonetDB with optimized translation (`monetdb_wcoj_optimzed`), since it aims for meeting the worst-case optimal runtime bound discussed by Ngo et al. [NRR13]. Figure 5.5 shows that `monetdb_wcoj_optimized` grows steadily with increasing m and that its angle can be estimated to be between $m^{\frac{3}{2}}$ and m^2. Therefore, the curve corresponding to `monetdb_wcoj_optimized` depends on the size of its input relations

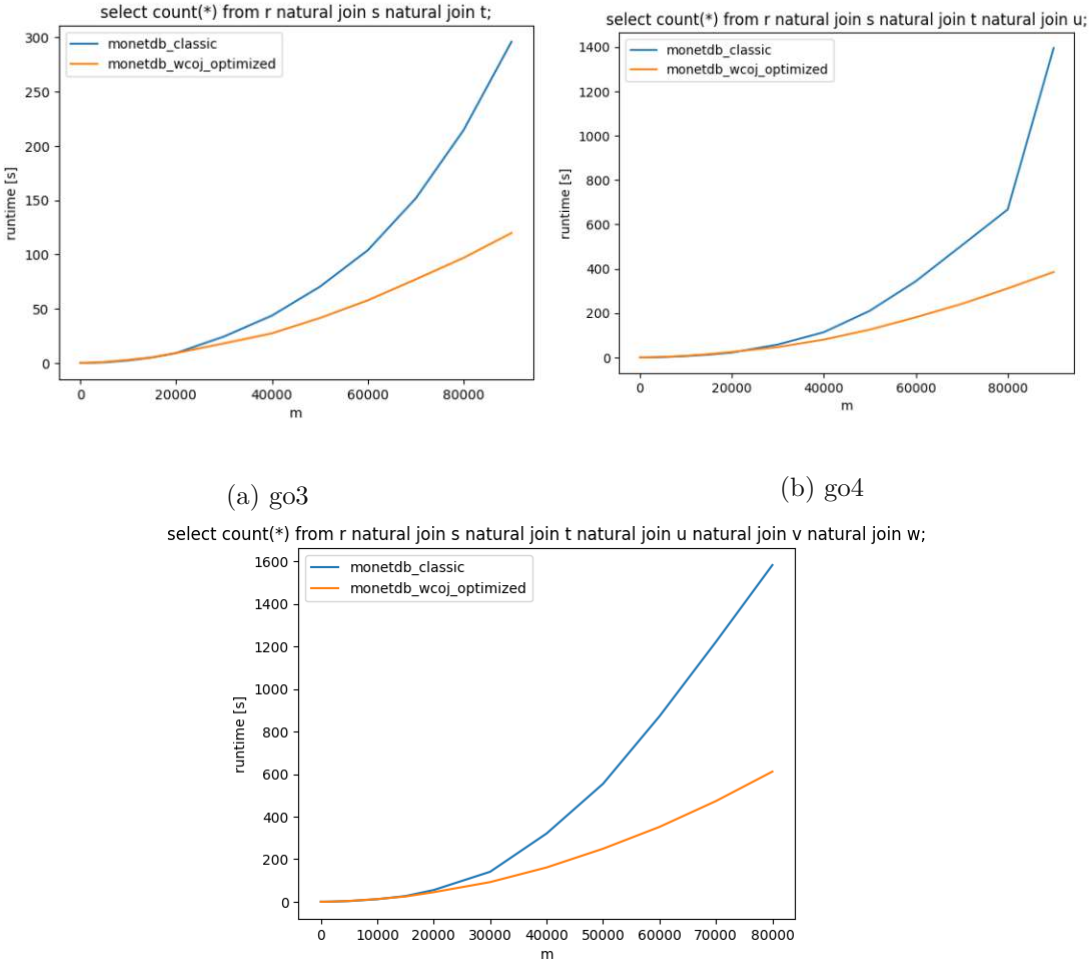
with a constant exponential factor between 1,5 and 2, which can be estimated to be closer to 2 when looking closely at the plots in Figure 5.5. Thus, the results meet the theoretical bound discussed by Ngo et al. [NRR13] with some potential overhead introduced by MonetDB. Furthermore, the curve `monetdb_classic` is way steeper, which underlines the worse asymptotic behaviour of the original MonetDB system for the given queries.

m	**Classic**			**WCOJ Optimized**		
	go3 (s)	go4 (s)	go6 (s)	go3 (s)	go4 (s)	go6 (s)
10	0,121	0,552	0,437	0,012	0,013	0,017
100	0,153	0,208	0,260	0,019	0,024	0,035
500	0,022	0,036	0,072	0,037	0,066	0,128
1000	0,040	0,081	0,361	0,071	0,148	0,288
2000	0,221	0,257	0,831	0,170	0,438	0,815
5000	0,595	1,675	3,268	0,974	2,496	3,928
10000	2,323	5,955	11,957	2,943	7,315	12,761
15000	5,125	12,658	26,821	5,240	15,266	24,751
20000	9,157	21,988	55,168	9,207	25,005	44,995
30000	24,449	58,296	141,043	18,077	46,814	92,416
40000	43,734	113,796	320,903	27,370	80,538	161,249
50000	70,318	210,338	554,513	41,463	125,379	249,312
60000	103,933	343,542	871,868	57,769	181,054	351,862
70000	151,664	504,957	1221,621	77,092	241,355	473,263
80000	214,647	667,121	1582,506	96,933	311,276	611,765
90000	296,099	1394,951	MEM-ERR	119,839	385,316	790,587
100000	MEM-ERR	MEM-ERR	MEM-ERR	149,555	488,995	976,385

Table 5.2: Table representation of the comparison of running times of the original MonetDB system (Classic) and the MonetDB optimized translation (WCOJ Optimized). The displayed numbers results from averaging three independent benchmark runs.

5.2.3 Impact of Fan-Out in Data

The following section will briefly discuss the impact of different fan-out values on the input data and will shed a light on skewed data. Moreover, the importance of skewed data and input relation size for the superiority of the WCOJ approach over the baseline system will be pointed out. All benchmarks discussed until now were performed on data generated with a fixed fan-out of 1 and a varying input relation size `m`. Thus, the performance of different MonetDB systems could be compared over increasing size of input relations. The evaluation discussed in this section compares the original MonetDB system to the MonetDB with optimized translation on a fixed input relation size $m = 50000$ and varying fan-out value between 1 and 5. The `benchmark.sh` script has been adapted in



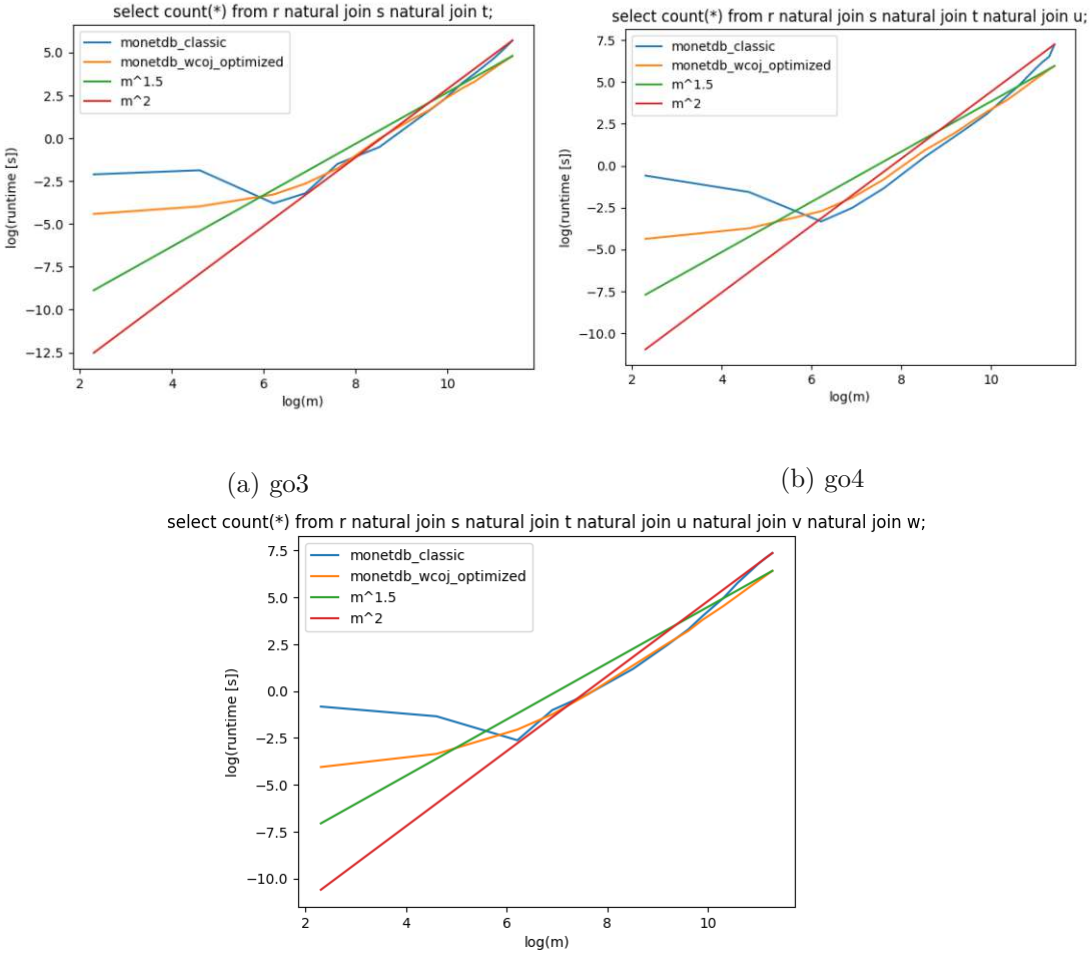
(a) go3

(b) go4

(c) go6

Figure 5.3: Visualization of benchmark results of the three queries go3, go4 and go6 for the original MonetDB vs. the optimized translation approach. Note that these plots correspond to the values depicted in Table 5.2 which are averaged across three independent runs.

order to enable such evaluation. Figure 5.6 shows the results of this benchmark with varying fan-out values for queries `go3`, `go4` and `go6`. In case of queries `go3` and `go6`, the MonetDB with optimized translation (`monetdb_wcoj_optimized`) outperforms the baseline up to a fan-out value of 4. The query `go4` is only superior up to a fan-out of 2 and shows similar runtime to the baseline for higher fan-out values. To be able to interpret the results from Figure 5.6, it is important to understand the meaning of an increasing fan-out while keeping the input relation size m constant. Given a fan-out of 1, the `gendb.py` utility will create relations where one value per attribute will occur



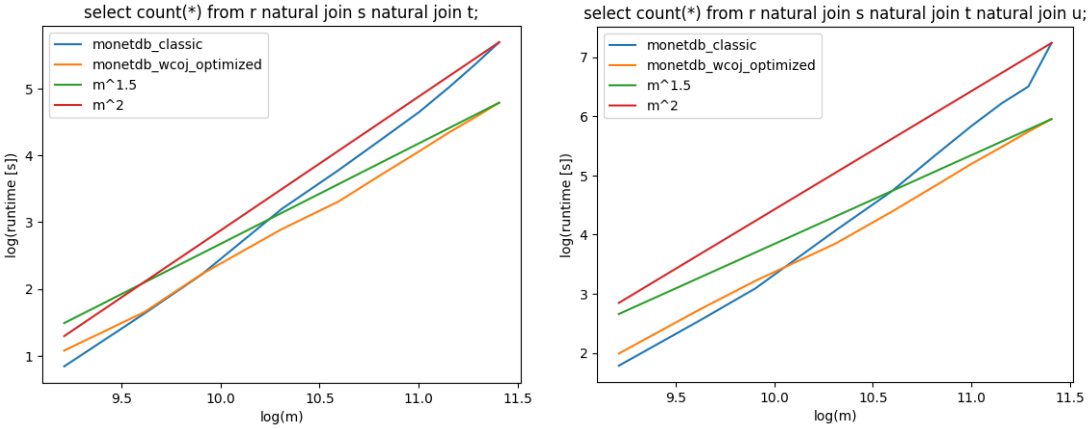
(a) go3

(b) go4

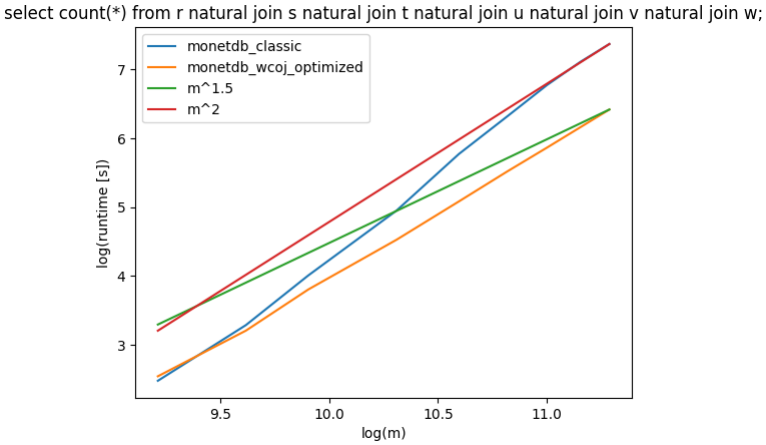
(c) go6

Figure 5.4: Visualization of the asymptotic behaviour of the benchmark results of the three queries go3, go4 and go6 for the original MonetDB vs. the optimized translation approach.

in a tuple with all other values. In case of the query `go3` the value $a0$ of attribute A in relation R will occur in a tuple together with all values $b0...49999$. Furthermore, for the same example a fan-out of two would create tuples where values $a0$ and $a1$ occur in tuples with all values $b0...b24999$ in order to keep the overall input relation size at $m = 50000$. This pattern can be followed to obtain the structure for higher fan-out values. Moreover, the value $a0$ would occur in a tuple with 50000 different values of attribute B for a fan-out of 1. Facing a fan-out of 2, this number would decrease to 25000, because value $a1$ will occur in tuples with 25000 different values of attribute B while the overall relation size remains constant at $m = 50000$. Following this pattern



(a) go3 (b) go4



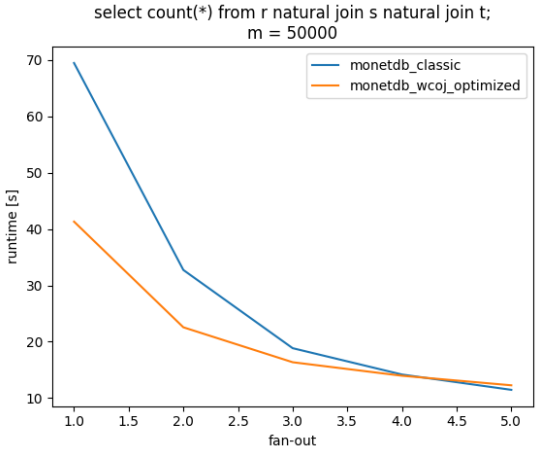
(c) go6

Figure 5.5: Visualization of the asymptotic behaviour of the benchmark results of the three queries go3, go4 and go6 for the original MonetDB vs. the optimized translation approach. The x-Axis takes m values from 10000 to 90000 into account and therefore depicts the asymptomatic runtime behaviour.

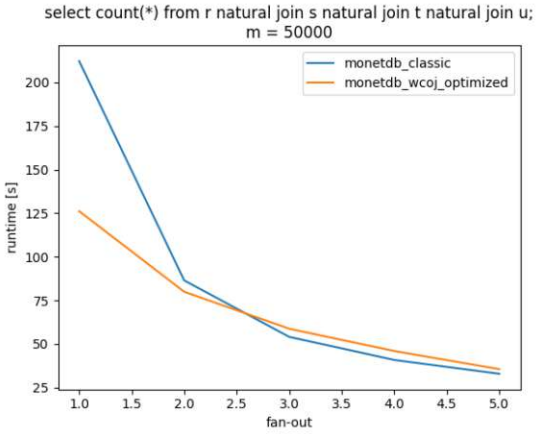
for an increasing fan-out value will gradually reduce the skew in input data. Thus, the current benchmark setting evaluates a given MonetDB variant's performance on data with different degrees of skew, which can be controlled by the fan-out parameter. Reducing skew by increasing the fan-out value leads to similar performance of the MonetDB with optimized translation in comparison to the baseline. Furthermore, the baseline sometimes performs even slightly better than the WCOJ approach for reduced skew. However, the MonetDB with optimized translation significantly outperforms the original MonetDB system for highly skewed data. This is the expected behaviour that has been discussed

by Ngo et al. [NRR13] from a database theoretical point of view.

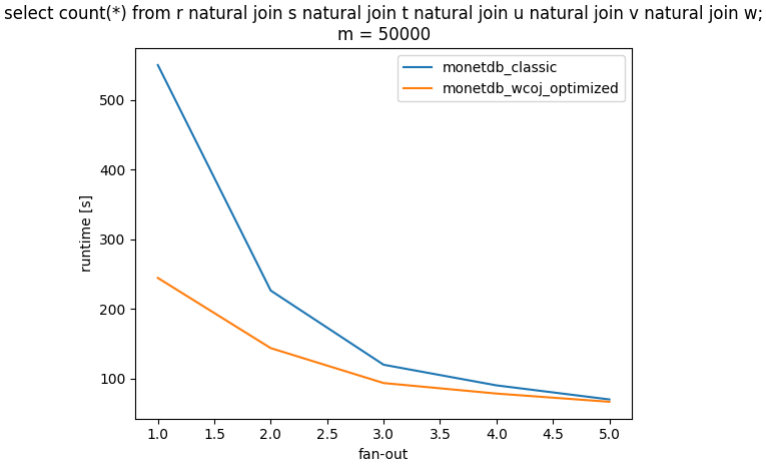
To sum up, the degree of skew within the input data directly impacts the superiority of the `Multi-Way-Join` approach introduced by WCOJ algorithms.



(a) go3



(b) go4



(c) go6

Figure 5.6: Visualization of runtime behaviour of MonetDB with optimized translation and the original MonetDB system with a fixed input relation size $m = 50000$ and varying fan-out value.

CHAPTER 6

Conclusion

This work has been the first to show that integrating WCOJ algorithms into column-oriented database systems is possible, and it has been evaluated that this integration can significantly increase performance for natural join queries on data-intensive workloads. The second important outcome shown is that the integration of WCOJ algorithms improved efficiency of the column-store on workloads with highly skewed data. An increasing degree of skew in the input data results in an increasing performance benefit of the column-store with integrated WCOJ algorithm over the baseline system. Furthermore, this work showed that the system with WCOJ algorithm integrated is more memory-efficient than the original MonetDB, since it successfully evaluates queries which cause memory allocation errors for the baseline system. Moreover, it is quite notable that the original MonetDB system can achieve similar performance to the system with WCOJ algorithm integrated for skewed input relations of sizes smaller than 20000 rows due to concepts from several decades of database research.

This work aims for providing an answer to the first open question in Chapter 5 of the fundamental theoretical research paper by Ngo et al. [NRR13] about WCOJs which this work is built upon. As shown by this work, the algorithmic ideas presented by Ngo et al. [NRR13] can gain runtime efficiency when integrated in column-oriented database systems. Moreover, the set of queries used for experimental evaluation of the resulting system were crafted according to Ngo et al. [NRR13] to underline the system's power on skewed input data. Figure 3.3 illustrates such skew on the example of the triangle query $\mathcal{Q}_{\triangle}$. Even tough, this work has been evaluated using synthetic data, one can imagine a real word setting with highly skewed data, where certain attributes provide low entropy values. Due to the power of WCOJ algorithms when evaluating joins on relations with skewed data, a tactical query optimizer could decide to make use of the WCOJ approach based on the given query and input relations. Thus, natural join queries could be computed way more runtime efficient in settings with highly skewed data.

Besides these very interesting evaluation results, this work has further analyzed, and discussed the MonetDB's query compiler and its MAL code in great detail. This can be of certain interest, since the official documentation does not provide such deep insights into the compiler. Furthermore, a manual translation of a class of WCOJ algorithms given by the `Generic-Join` algorithm 3.1 into a MAL program has been introduced. Based on the overall knowledge of the specific column-store and the `Generic-Join` algorithm 3.1, an optimized translation has been proposed and discussed. This translation is used by the final system resulting from this work.

6.1 Open Questions

Finally, some open topics were identified that could be visited in the future in order to further improve the integration of a class of WCOJ algorithms into column-stores.

The integrated `Generic-Join` algorithm 3.1 divides the query hypergraph based on a partition of vertices $\mathcal{V}$ into two sets I and J. The partition strategy chosen in this work is relatively simple and need not be an optimal strategy to the best of our knowledge. An open problem is to find the optimal partitioning strategy in order to further improve the efficiency of the column-store with WCOJ algorithm integrated. Furthermore, the performance benefit of an optimal strategy compared to the straightforward one discussed in this work has to be evaluated.

The second open question is related to the memory footprints of the respective column-store implementations during benchmark runs. This work provides evidence that the MonetDB with WCOJ algorithm integrated is more memory-efficient than the original system. However, the corresponding memory footprints have not been measured during the experimental evaluation. Future work could investigate and evaluate the overall memory allocation of the respective implementations. The gained insights could potentially lead to ideas for further improvements of this work.

APPENDIX A

MAL Programs

This chapter shows various MAL programs that are referenced throughout this work. These MAL programs include example programs that result from the modified query compiler and cover the final results for different queries, as well as different intermediate results that have been achieved during earlier integration attempts. MAL code generated by the query compiler can be obtained using the `EXPLAIN` statement modifier. Moreover, this statement modifier prints all actions that have been applied in order to generate the MAL program (e.g. it lists the utilized optimizer and the number of times specific modules have been used respectively). Furthermore, the `mclient` program can be configured to run MAL code directly instead of evaluating SQL queries. This can be achieved by starting the `mclient` program using the following command: *mclient -d <dbname> -l msql*. It is important to note that running MAL programs obtained using the `EXPLAIN` statement modifier can lead to problems with generic variable names. Renaming them solves this issue and it remains unclear why running MAL code using generic variable names can cause problems for some instructions. Listing A.3 shows a MAL program that computes the triangle query on given relations $R(A,B), S(B,C), T(A,C)$ following the `Generic-Join` algorithm 3.1 and which can be successfully run using the above configuration of the `mclient`.

Moreover, Listing A.1 shows another MAL program that computes the output of the triangle query $\mathcal{Q}_{\triangle}$ for relations $R(A,B), S(B,C), T(A,C)$. This program has been generated by the unmodified MonetDB using the `EXPLAIN` statement modifier and represents the optimized version of the program. For instance, optimizer introduced the hash operators on BATs and dataflow optimizations. Note that this program cannot be run by the MonetDB's MAL interpreter when passed to it using the `mclient` program, since it somehow struggles with the dataflow barrier, log output and the `language.pass` operator. Removing lines 1-3, 44-60 and 62 as well as renaming generic variable identifiers that cause interpretation errors will make the program shown in Listing A.1 interpretable

by MonetDB using the `mclient`.

```
1  function user.main():void;
2    X_1:void := querylog.define("explain select * from r natural join s
         natural join t;":str, "default_pipe":str, 70:int);
3    barrier X_166:bit := language.dataflow();
4      X_4:int := sql.mvc();
5      C_5:bat[:oid] := sql.tid(X_4:int, "sys":str, "s":str);
6      X_8:bat[:str] := sql.bind(X_4:int, "sys":str, "s":str, "b":str, 0:int);
7      X_15:bat[:str] := sql.bind(X_4:int, "sys":str, "s":str, "c":str,
           0:int);
8      C_20:bat[:oid] := sql.tid(X_4:int, "sys":str, "t":str);
9      X_22:bat[:str] := sql.bind(X_4:int, "sys":str, "t":str, "a":str,
           0:int);
10     X_27:bat[:str] := sql.bind(X_4:int, "sys":str, "t":str, "c":str,
           0:int);
11     X_32:bat[:str] := algebra.projection(C_5:bat[:oid], X_15:bat[:str]);
12     X_34:bat[:str] := algebra.projection(C_20:bat[:oid], X_27:bat[:str]);
13     (X_35:bat[:oid], X_36:bat[:oid]) := algebra.join(X_32:bat[:str],
           X_34:bat[:str], nil:BAT, nil:BAT, false:bit, nil:lng);
14     X_41:bat[:str] := algebra.projectionpath(X_35:bat[:oid],
           C_5:bat[:oid], X_8:bat[:str]);
15     X_43:bat[:str] := algebra.projectionpath(X_36:bat[:oid],
           C_20:bat[:oid], X_22:bat[:str]);
16     C_45:bat[:oid] := sql.tid(X_4:int, "sys":str, "r":str);
17     X_48:bat[:str] := sql.bind(X_4:int, "sys":str, "r":str, "a":str,
           0:int);
18     X_54:bat[:str] := sql.bind(X_4:int, "sys":str, "r":str, "b":str,
           0:int);
19     X_59:bat[:str] := algebra.projection(C_45:bat[:oid], X_48:bat[:str]);
20     X_60:bat[:str] := algebra.projection(C_45:bat[:oid], X_54:bat[:str]);
21     X_67:bat[:lng] := batmkey.hash(X_41:bat[:str]);
22     X_70:bat[:lng] := mkey.bulk_rotate_xor_hash(X_67:bat[:lng], 22:int,
           X_43:bat[:str]);
23     X_72:bat[:lng] := batmkey.hash(X_60:bat[:str]);
24     X_73:bat[:lng] := mkey.bulk_rotate_xor_hash(X_72:bat[:lng], 22:int,
           X_59:bat[:str]);
25     (X_74:bat[:oid], X_75:bat[:oid]) := algebra.join(X_70:bat[:lng],
           X_73:bat[:lng], nil:BAT, nil:BAT, true:bit, nil:lng);
26     X_80:bat[:str] := algebra.projection(X_74:bat[:oid], X_41:bat[:str]);
27     X_81:bat[:str] := algebra.projection(X_75:bat[:oid], X_60:bat[:str]);
28     X_82:bat[:bit] := batcalc.==(X_80:bat[:str], X_81:bat[:str]);
29     C_86:bat[:oid] := algebra.thetaselect(X_82:bat[:bit], nil:BAT,
           true:bit, "==":str);
30     X_87:bat[:oid] := algebra.projection(C_86:bat[:oid], X_74:bat[:oid]);
31     X_88:bat[:oid] := algebra.projection(C_86:bat[:oid], X_75:bat[:oid]);
32     X_89:bat[:str] := algebra.projection(X_87:bat[:oid], X_43:bat[:str]);
33     X_90:bat[:str] := algebra.projection(X_88:bat[:oid], X_59:bat[:str]);
34     X_91:bat[:bit] := batcalc.==(X_89:bat[:str], X_90:bat[:str]);
35     C_93:bat[:oid] := algebra.thetaselect(X_91:bat[:bit], nil:BAT,
           true:bit, "==":str);
36     X_95:bat[:oid] := algebra.projection(C_93:bat[:oid], X_88:bat[:oid]);
37     X_97:bat[:str] := algebra.projectionpath(C_93:bat[:oid],
```

```
          X_87:bat[:oid], X_35:bat[:oid], X_32:bat[:str]);
38      X_100:bat[:str] := algebra.projection(X_95:bat[:oid], X_59:bat[:str]);
39      X_101:bat[:str] := algebra.projection(X_95:bat[:oid], X_60:bat[:str]);
40      X_103:bat[:str] := bat.pack("sys.r":str, "sys.r":str, "sys.s":str);
41      X_104:bat[:str] := bat.pack("a":str, "b":str, "c":str);
42      X_105:bat[:str] := bat.pack("clob":str, "clob":str, "clob":str);
43      X_106:bat[:int] := bat.pack(0:int, 0:int, 0:int);
44      X_168:void := language.pass(C_5:bat[:oid]);
45      X_169:void := language.pass(C_20:bat[:oid]);
46      X_170:void := language.pass(C_45:bat[:oid]);
47      X_171:void := language.pass(X_41:bat[:str]);
48      X_172:void := language.pass(X_74:bat[:oid]);
49      X_173:void := language.pass(C_86:bat[:oid]);
50      X_174:void := language.pass(X_75:bat[:oid]);
51      X_175:void := language.pass(X_43:bat[:str]);
52      X_176:void := language.pass(X_88:bat[:oid]);
53      X_177:void := language.pass(C_93:bat[:oid]);
54      X_178:void := language.pass(X_87:bat[:oid]);
55      X_179:void := language.pass(X_35:bat[:oid]);
56      X_180:void := language.pass(X_32:bat[:str]);
57      X_181:void := language.pass(X_59:bat[:str]);
58      X_182:void := language.pass(X_95:bat[:oid]);
59      X_183:void := language.pass(X_60:bat[:str]);
60    exit X_166:bit;
61    X_102:int := sql.resultSet(X_103:bat[:str], X_104:bat[:str],
          X_105:bat[:str], X_106:bat[:int], X_106:bat[:int], X_100:bat[:str],
          X_101:bat[:str], X_97:bat[:str]);
62  end user.main;
```

Listing A.1: MAL program generated by the unmodified MonetDB using the `EXPLAIN` modifier that evaluates the triangle query.

```
1  sqlmvc:int := sql.mvc();
2  tidS:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "s":str);
3  bindSB:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "s":str, "b":str,
       0:int);
4  bindSC:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "s":str, "c":str,
       0:int);
5  projSB:bat[:str] := algebra.projection(tidS:bat[:oid], bindSB:bat[:str]);
6  tidT:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "t":str);
7  bindTA:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "t":str, "a":str,
       0:int);
8  bindTC:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "t":str, "c":str,
       0:int);
9  projTC:bat[:str] := algebra.projection(tidT:bat[:oid], bindTC:bat[:str]);
10 tidR:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "r":str);
11 bindRA:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "r":str, "a":str,
       0:int);
12 bindRB:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "r":str, "b":str,
       0:int);
13 projRB:bat[:str] := algebra.projection(tidR:bat[:oid], bindRB:bat[:str]);
14
15 a:bat[:str] := bat.new(nil:str);
```

```
16  b:bat[:str] := bat.new(nil:str);
17  c:bat[:str] := bat.new(nil:str);
18
19  interB:bat[:oid] := algebra.intersect(projSB:bat[:str], projRB:bat[:str],
        nil:bat[:oid], nil:bat[:oid], false:bit, false:bit, nil:lng);
20  projInterB:bat[:str] := algebra.projection(interB:bat[:oid],
        projSB:bat[:str]);
21  (X_62:bat[:oid], C_63:bat[:oid]) := group.groupdone(projInterB:bat[:str]);
22  distinctB:bat[:str] := algebra.projection(C_63:bat[:oid],
        projInterB:bat[:str]);
23
24  barrier (X_68:oid, X_69:str) := iterator.new(distinctB:bat[:str]);
25    X_70:bat[:str] := bat.new(nil:str);
26    tupleB:bat[:str] := bat.append(X_70:bat[:str], X_69:str, true:bit);
27
28    X_73:bat[:oid] := algebra.intersect(projSB:bat[:str], tupleB:bat[:str],
          nil:bat[:oid], nil:bat[:oid], false:bit, true:bit, nil:lng);
29    semiSC:bat[:str] := algebra.projectionpath(X_73:bat[:oid],
          tidS:bat[:oid], bindSC:bat[:str]);
30    X_79:bat[:oid] := algebra.intersect(projRB:bat[:str], tupleB:bat[:str],
          nil:bat[:oid], nil:bat[:oid], false:bit, true:bit, nil:lng);
31    semiRA:bat[:str] := algebra.projectionpath(X_79:bat[:oid],
          tidR:bat[:oid], bindRA:bat[:str]);
32
33    interC:bat[:oid] := algebra.intersect(semiSC:bat[:str],
          projTC:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
          false:bit, nil:lng);
34    projInterC:bat[:str] := algebra.projection(interC:bat[:oid],
          semiSC:bat[:str]);
35    (X_87:bat[:oid], C_88:bat[:oid]) :=
          group.groupdone(projInterC:bat[:str]);
36    distinctC:bat[:str] := algebra.projection(C_88:bat[:oid],
          projInterC:bat[:str]);
37
38    atmp:bat[:str] := bat.new(nil:str);
39    barrier (X_94:oid, X_95:str) := iterator.new(distinctC:bat[:str]);
40      X_96:bat[:str] := bat.new(nil:str);
41      tupleC:bat[:str] := bat.append(X_96:bat[:str], X_95:str, true:bit);
42
43      X_98:bat[:oid] := algebra.intersect(projTC:bat[:str],
            tupleC:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
            true:bit, nil:lng);
44      semiTA:bat[:str] := algebra.projectionpath(X_98:bat[:oid],
            tidT:bat[:oid], bindTA:bat[:str]);
45
46      C_106:bat[:oid] := algebra.intersect(semiTA:bat[:str],
            semiRA:bat[:str], nil:bat[:oid], nil:bat[:oid], false:bit,
            false:bit, nil:lng);
47      X_107:bat[:str] := algebra.projection(C_106:bat[:oid],
            semiTA:bat[:str]);
48      (X_108:bat[:oid], C_109:bat[:oid]) := group.groupdone(X_107:bat[:str]);
49      X_111:bat[:str] := algebra.projection(C_109:bat[:oid],
            X_107:bat[:str]);
```

```
50
51      (X_112:bat[:oid], X_113:bat[:oid]) :=
            algebra.crossproduct(X_111:bat[:str], tupleC:bat[:str], false:bit);
52      X_114:bat[:str] := algebra.projection(X_112:bat[:oid],
            X_111:bat[:str]);
53      X_115:bat[:str] := algebra.project(X_113:bat[:oid], X_95:str);
54      atmp:bat[:str] := bat.append(atmp:bat[:str], X_114:bat[:str],
            true:bit);
55      c:bat[:str] := bat.append(c:bat[:str], X_115:bat[:str], true:bit);
56
57      redo (X_94:oid, X_95:str) := iterator.next(distinctC:bat[:str]);
58    exit (X_94:oid, X_95:str);
59
60    X_123:bat[:oid] := algebra.crossproduct(tupleB:bat[:str],
          atmp:bat[:str], false:bit);
61    X_124:bat[:str] := algebra.project(X_123:bat[:oid], X_69:str);
62    b:bat[:str] := bat.append(b:bat[:str], X_124:bat[:str], true:bit);
63    a:bat[:str] := bat.append(a:bat[:str], atmp:bat[:str], true:bit);
64
65    redo (X_68:oid, X_69:str) := iterator.next(distinctB:bat[:str]);
66  exit (X_68:oid, X_69:str);
```

Listing A.2: MAL program that evaluates the triangle query worst-case optimally following the `Generic-Join` algorithm 3.1. This is the directly translated MAL program for evaluating the triangle query without further improvements.

```
1  sqlmvc:int := sql.mvc();
2  tidS:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "s":str);
3  bindSB:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "s":str, "b":str,
       0:int);
4  bindSC:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "s":str, "c":str,
       0:int);
5  projSB:bat[:str] := algebra.projection(tidS:bat[:oid], bindSB:bat[:str]);
6  tidT:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "t":str);
7  bindTA:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "t":str, "a":str,
       0:int);
8  bindTC:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "t":str, "c":str,
       0:int);
9  projTC:bat[:str] := algebra.projection(tidT:bat[:oid], bindTC:bat[:str]);
10 tidR:bat[:oid] := sql.tid(sqlmvc:int, "sys":str, "r":str);
11 bindRA:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "r":str, "a":str,
       0:int);
12 bindRB:bat[:str] := sql.bind(sqlmvc:int, "sys":str, "r":str, "b":str,
       0:int);
13 projRB:bat[:str] := algebra.projection(tidR:bat[:oid], bindRB:bat[:str]);
14
15 b:bat[:str] := bat.new(nil:str);
16 c:bat[:str] := bat.new(nil:str);
17 tmpC:bat[:str] := bat.new(nil:str);
18 a:bat[:str] := bat.new(nil:str);
19
20 distA:bat[:str] := bat.new(nil:str);
21 distB:bat[:str] := bat.new(nil:str);
```

```
22  distC:bat[:str] := bat.new(nil:str);
23
24  C_58:bat[:oid] := algebra.intersect(projSB:bat[:str], projRB:bat[:str],
        nil:bat[:oid], nil:bat[:oid], false:bit, false:bit, nil:lng);
25  projInterB:bat[:str] := algebra.projection(C_58:bat[:oid],
        projSB:bat[:str]);
26  (X_65:bat[:oid], C_66:bat[:oid]) := group.groupdone(projInterB:bat[:str]);
27  distB:bat[:str] := algebra.projection(C_66:bat[:oid],
        projInterB:bat[:str]);
28
29  barrier (X_70:oid, X_71:str) := iterator.new(distB:bat[:str]);
30    X_75:bat[:oid] := algebra.thetaselect(bindSB:bat[:str], tidS:bat[:oid],
          X_71:str, "==":str);
31    X_77:bat[:str] := algebra.projectionpath(X_75:bat[:oid], tidS:bat[:oid],
          bindSC:bat[:str]);
32    X_78:bat[:oid] := algebra.thetaselect(bindRB:bat[:str], tidR:bat[:oid],
          X_71:str, "==":str);
33    X_80:bat[:str] := algebra.projectionpath(X_78:bat[:oid], tidR:bat[:oid],
          bindRA:bat[:str]);
34
35    C_81:bat[:oid] := algebra.intersect(X_77:bat[:str], projTC:bat[:str],
          nil:bat[:oid], nil:bat[:oid], false:bit, false:bit, nil:lng);
36    X_82:bat[:str] := algebra.projection(C_81:bat[:oid], X_77:bat[:str]);
37    (X_83:bat[:oid], C_84:bat[:oid]) := group.groupdone(X_82:bat[:str]);
38    distC:bat[:str] := algebra.projection(C_84:bat[:oid], X_82:bat[:str]);
39
40    tmpC:bat[:str] := bat.new(nil:str);
41    tmpBCand:bat[:oid] := bat.new(nil:oid);
42    barrier (X_90:oid, X_91:str) := iterator.new(distC:bat[:str]);
43      X_94:bat[:oid] := algebra.thetaselect(bindTC:bat[:str],
            tidT:bat[:oid], X_91:str, "==":str);
44      X_100:bat[:str] := algebra.projectionpath(X_94:bat[:oid],
            tidT:bat[:oid], bindTA:bat[:str]);
45
46      C_101:bat[:oid] := algebra.intersect(X_100:bat[:str], X_80:bat[:str],
            nil:bat[:oid], nil:bat[:oid], false:bit, false:bit, nil:lng);
47      X_102:bat[:str] := algebra.projection(C_101:bat[:oid],
            X_100:bat[:str]);
48      (X_103:bat[:oid], C_104:bat[:oid]) := group.groupdone(X_102:bat[:str]);
49      distA:bat[:str] := algebra.projection(C_104:bat[:oid],
            X_102:bat[:str]);
50
51      X_110:bat[:str] := algebra.project(C_104:bat[:oid], X_91:str);
52
53      tmpBCand:bat[:oid] := bat.append(tmpBCand:bat[:oid], C_104:bat[:oid]);
54      a:bat[:str] := bat.append(a:bat[:str], distA:bat[:str], true:bit);
55      tmpC:bat[:str] := bat.append(tmpC:bat[:str], X_110:bat[:str],
            true:bit);
56
57      redo (X_90:oid, X_91:str) := iterator.next(distC:bat[:str]);
58    exit (X_90:oid, X_91:str);
59
60    X_119:bat[:str] := algebra.project(tmpBCand:bat[:oid], X_71:str);
```

```
61
62    c:bat[:str] := bat.append(c:bat[:str], tmpC:bat[:str], true:bit);
63    b:bat[:str] := bat.append(b:bat[:str], X_119:bat[:str], true:bit);
64
65    redo (X_70:oid, X_71:str) := iterator.next(distB:bat[:str]);
66  exit (X_70:oid, X_71:str);
```

Listing A.3: MAL program that evaluates the triangle query worst-case optimally following the `Generic-Join` algorithm 3.1. This is the final improved MAL program for evaluating the triangle query.

Acronyms

ACID Atomicity, Consistency, Isolation and Durability. 18

API Application Programming Interface. 17

BAT Binary Association Table. 12, 16–18, 20–25, 53, 54, 57–59, 61–67, 69, 70, 72, 73, 80, 81, 89

BBP BAT Buffer Pool. 18

CPU Central Processing Unit. 4, 10, 12, 13, 16, 17, 19, 77

CSP Constraint Satisfaction Problem. 37

CWI Centrum Wiskunde & Informatica. 16

DBMS Database Management System. xv, 4, 7–9, 15–17, 27, 30, 32, 33, 47

DSM Decomposed Storage Model. 18

GDK Goblin Database Kernel. 18, 21

GHD Generalized Hypertree Decomposition. 46

MAL MonetDB Assembly Language. 5, 6, 10, 11, 14, 17–26, 45, 48–73, 75, 79, 88, 89, 91, 93, 95

MMU Memory Management Unit. 19

OID Object Identifier. 16, 23, 54, 59, 66, 67

OS Operating System. 18, 77

RLE Run Length Encoding. 10, 13

SIMD Single Instruction Multiple Data. 4, 12, 46

SQL Structured Query Language. 17, 19, 25, 46, 55–57, 69, 71, 89

WCOJ Worst Case Optimal Join. 3–6, 15, 22, 24, 27, 28, 33–37, 45–49, 55, 56, 60, 65, 67–69, 71, 72, 77–79, 81, 82, 85–88

Bibliography

[ABH+13] Daniel Abadi, Peter A. Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5(3):197–280, 2013.

[AGM08] Albert Atserias, Martin Grohe, and Dániel Marx. Size bounds and query plans for relational joins. In *49th Annual IEEE Symposium on Foundations of Computer Science, FOCS 2008, October 25-28, 2008, Philadelphia, PA, USA*, pages 739–748. IEEE Computer Society, 2008.

[Alo81] Noga Alon. On the number of subgraphs of prescribed type of graphs with a given number of edges. *Israel Journal of Mathematics*, 38(1-2):116–130, 1981.

[ALOR17] Christopher R. Aberger, Andrew Lamb, Kunle Olukotun, and Christopher Ré. Levelheaded: Making worst-case optimal joins work in the common case. *CoRR*, abs/1708.07859, 2017.

[ALT+17] Christopher R. Aberger, Andrew Lamb, Susan Tu, Andres Nötzli, Kunle Olukotun, and Christopher Ré. Emptyheaded: A relational engine for graph processing. *ACM Trans. Database Syst.*, 42(4):20:1–20:44, 2017.

[AMDM07] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. Materialization strategies in a column-oriented DBMS. In Rada Chirkova, Asuman Dogac, M. Tamer Özsu, and Timos K. Sellis, editors, *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel, Istanbul, Turkey, April 15-20, 2007*, pages 466–475. IEEE Computer Society, 2007.

[AMF06] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 671–682. ACM, 2006.

[AtCG+15] Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design

and implementation of the logicblox system. In Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives, editors, *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1371–1382. ACM, 2015.

[BGvK+06] Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. Monetdb/xquery: a fast xquery processor powered by a relational engine. In Surajit Chaudhuri, Vagelis Hristidis, and Neoklis Polyzotis, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*, pages 479–490. ACM, 2006.

[BKM08] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[BMK09] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. Database architecture evolution: Mammals flourished long before dinosaurs became extinct. *Proc. VLDB Endow.*, 2(2):1648–1653, 2009.

[BT95] Béla Bollobás and Andrew Thomason. Projections of bodies and hereditary properties of hypergraphs. *Bulletin of the London Mathematical Society*, 27(5):417–424, 1995.

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *Second Biennial Conference on Innovative Data Systems Research, CIDR 2005, Asilomar, CA, USA, January 4-7, 2005, Online Proceedings*, pages 225–237. www.cidrdb.org, 2005.

[FBS+20] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. Adopting worst-case optimal joins in relational database systems. *Proc. VLDB Endow.*, 13(11):1891–1904, 2020.

[GLVV12] Georg Gottlob, Stephanie Tien Lee, Gregory Valiant, and Paul Valiant. Size and treewidth bounds for conjunctive queries. *J. ACM*, 59(3):16:1–16:35, 2012.

[GM06] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. In *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, pages 289–298. ACM Press, 2006.

[GM14] Martin Grohe and Dániel Marx. Constraint solving via fractional edge covers. *ACM Trans. Algorithms*, 11(1):4:1–4:20, 2014.

[Gra93] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[Gro13] Martin Grohe. Bounds and algorithms for joins via fractional edge covers. In Val Tannen, Limsoon Wong, Leonid Libkin, Wenfei Fan, Wang-Chiew Tan, and Michael P. Fourman, editors, *In Search of Elegance in the Theory and Practice of Computation - Essays Dedicated to Peter Buneman*, volume 8000 of *Lecture Notes in Computer Science*, pages 321–338. Springer, 2013.

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[IKM07] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Database cracking. In *Third Biennial Conference on Innovative Data Systems Research, CIDR 2007, Asilomar, CA, USA, January 7-10, 2007, Online Proceedings*, pages 68–78. www.cidrdb.org, 2007.

[IKNG10] Milena Ivanova, Martin L. Kersten, Niels J. Nes, and Romulo Goncalves. An architecture for recycling intermediates in a column-store. *ACM Trans. Database Syst.*, 35(4):24:1–24:43, 2010.

[LW+49] Lynn H Loomis, Hassler Whitney, et al. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55(10):961–962, 1949.

[MBK02] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing main-memory join on modern hardware. *IEEE Trans. Knowl. Data Eng.*, 14(4):709–730, 2002.

[Mona] MonetDB. Mal optimizers. `https://www.monetdb.org/Documentation/Manuals/MonetDB/Optimizers`. Accessed: 2021-10-11.

[Monb] MonetDB. Monetdb. `https://www.monetdb.org/Home`. Accessed: 2021-09-18.

[Monc] MonetDB. Monetdb mal reference. `https://www.monetdb.org/Documentation/MonetDBInternals/MALReference`. Accessed: 2021-10-05.

[Mon21] MonetDB. Monetdb. `https://dev.monetdb.org/hg/MonetDB/`, 2021. Accessed: 2021-09-18.

[Ngo18] Hung Q. Ngo. Worst-case optimal join algorithms: Techniques, results, and open problems. In Jan Van den Bussche and Marcelo Arenas, editors, *Proceedings of the 37th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, Houston, TX, USA, June 10-15, 2018*, pages 111–124. ACM, 2018.

[NPRR12] Hung Q. Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms: [extended abstract]. In Michael Benedikt, Markus Krötzsch, and Maurizio Lenzerini, editors, *Proceedings of the 31st ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS 2012, Scottsdale, AZ, USA, May 20-24, 2012*, pages 37–48. ACM, 2012.

[NRR13] 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.

[NRR20] Gonzalo Navarro, Juan L. Reutter, and Javiel Rojas-Ledesma. Optimal joins using compact data structures. In Carsten Lutz and Jean Christoph Jung, editors, *23rd International Conference on Database Theory, ICDT 2020, March 30-April 2, 2020, Copenhagen, Denmark*, volume 155 of *LIPIcs*, pages 21:1–21:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[RG03] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3. ed.)*. McGraw-Hill, 2003.

[SAB+05] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. C-store: A column-oriented DBMS. In Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 553–564. ACM, 2005.

[Vel14] Todd L. Veldhuizen. Triejoin: A simple, worst-case optimal join algorithm. In Nicole Schweikardt, Vassilis Christophides, and Vincent Leroy, editors, *Proc. 17th International Conference on Database Theory (ICDT), Athens, Greece, March 24-28, 2014*, pages 96–106. OpenProceedings.org, 2014.

[Yan81] Mihalis Yannakakis. Algorithms for acyclic database schemes. In *Very Large Data Bases, 7th International Conference, September 9-11, 1981, Cannes, France, Proceedings*, pages 82–94. IEEE Computer Society, 1981.