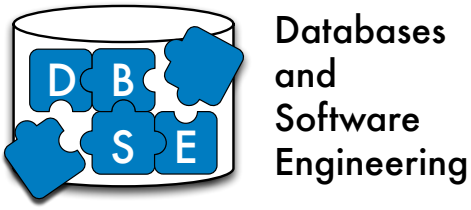
University of Magdeburg

School of Computer Science



Master's Thesis

Elf meets MonetDB: Integrating a multi-column structure into a column store

Author:

Florian Bethe

May 11, 2018

Advisors:

Prof. Gunter Saake

Department for Technical and Business Information Systems

M.Sc. David Broneske

Department for Technical and Business Information Systems

Bethe, Florian:
Elf meets MonetDB: Integrating a multi-column structure into a column store
Master's Thesis, University of Magdeburg, 2018.

Abstract

The emergence of main-memory DBMS brought about the need of cache-conscious structures and algorithms. For the workload of OLAP scenarios, column stores like *MonetDB* have a favourable memory layout, allowing sequential scans over contiguous memory. When facing selection predicates for multiple columns, however, they offer little to accelerate them. Multi-dimensional index structures such as *kd-trees* attempt to improve on plain scans, but face the curse of dimensionality when many columns are queried.

In this work, we integrate the multi-dimensional main-memory index structure *Elf*, which does not suffer from said curse, into the DBMS *MonetDB*. Since Elf only supports *select* queries, we provide interoperability with MonetDB's query engine and show various improvements of the naive approach. To enable real-world use, we propose two competing approaches of querying string-typed columns with Elf. As modern CPUs feature lengthy pipelines and out-of-order execution, we also explore the possible trade-off between branching complexity and early termination for Elf traversal.

Our experiments show that Elf can outperform the accelerated main-memory scans of MonetDB, which improves further for larger datasets. However, the overhead of string queries is not negligible and has to be considered when choosing between the two. Additionally, the set of columns on which Elf is built should be kept to a minimum; the performance worsens significantly if unaffected columns are included.

Inhaltsangabe

Das Aufkommen von Hauptspeicher-Datenbanksystemen brachte das Bedürfnis nach Cache-bewussten Strukturen und Algorithmen mit sich. "Column stores" wie *MonetDB* haben eine für OLAP-Szenarien vorteilhafte Speicheranordnung, welche sequenzielle Speicherscans ermöglicht. Allerdings bieten diese nur wenige Beschleunigungsmöglichkeiten für mehrspaltige Selektionsprädikate. Um dies zu verbessern wurden zahlreiche mehrdimensionale Indexstrukturen wie *kd-Bäume* vorgeschlagen. Leider kämpfen diese aber mit dem Fluch der Dimensionalität, wenn viele Spalten abgefragt werden.

In dieser Arbeit integrieren wir die mehrdimensionale Hauptspeicher-Indexstruktur *Elf*, welche nicht von dem genannten Fluch betroffen ist, in das Datenbanksystem *MonetDB*. Da Elf nur Selektionsabfragen unterstützt, stellen wir Methoden zur Interoperabilität mit MonetDB's Query Engine bereit und zeigen verschiedene Verbesserungen dieser einfachen Ansätze. Um den realen Einsatz zu ermöglichen, bringen wir zwei konkurrierende Ansätze zur Abfrage von Spalten mit String-Typ ein. Da moderne CPUs lange Pipelines und Out-of-order Ausführung aufweisen, erkunden wir auch den Konflikt zwischen Branching-Komplexität und frühem Abbrechen bei der Traversierung des Elf.

Unsere Experimente zeigen, dass Elf schneller als die beschleunigten Hauptspeicher-Scans von MonetDB sein kann, was sich für größere Datensätze noch verbessert. Allerdings sind die zusätzlichen Kosten von String-Abfragen nicht vernachlässigbar und müssen bei der Wahl zwischen den beiden bedacht werden. Zusätzlich sollte die Menge an Spalten, aus welchen die Elf-Struktur gebaut wird, auf ein Minimum reduziert werden, da nicht betroffene Spalten die Leistung erheblich reduzieren können.

Acknowledgements

I would like to thank my supervisor, David Broneske, for providing me both insight into the topic as well as invaluable support throughout the thesis, for which I am very grateful.

I would also like to thank Penny and Dennis for them supporting me and listening to my complaints.

Contents

List of Figures

List of Tables

List of Code Listings

1. Introduction

The increasing amount of main memory in modern Database Systems brought about a shift in focus for these systems. Formerly limited by their disk speed, the performance bottleneck shifted upwards in the storage hierarchy towards *main-memory* access [MBK00a]. Following this, utilizing the CPU cache became more important to mitigate the limited bandwidth and access times of RAM, leading to the development of cache-concious algorithms.

In analytical databases, one often encounters full table scans as an operation. While data was traditionally split into pages to save disk bandwidth, main-memory database systems do not need to perform such splits. However, the question of how table data should be partitioned remains: a primarily transactional workload as found in OLTP scenarios benefits from storing tuples compactly, while OLAP applications often scan over individual attributes. Although wasting bandwidth is not a concern for a main-memory DBMS, using a *decomposed storage model* increases its cache friendliness for this use case by reducing the amount of irrelevant data per cache line [MBK00a].

Neither of the two traditional storage models address the increasingly frequent workload of evaluating selection predicates on multiple columns. To accelerate attribute scans, techniques such as *column imprints* [SK13] and *BitWeaving* [LP13] have been proposed. While they are designed to be cache-friendly, both can only evaluate one column at a time. Other approaches such as tree-based index structures do evaluate multi-column predicates in one scan, but usually suffer from the *curse of dimensionality*; their effectiveness decreases with increasing column count, eventually making plain scans the preferable option.

The recently proposed *Elf*, a main-memory index structure, does not suffer from this particular shortcoming. It achieves this by reducing the amount of data stored, grouping together tuples with equal prefixes. Its design takes into account the CPU cache, densely packing column values and stores them consecutively in memory. So far, however, Elf has only been tested as a stand-alone structure in limited test scenarios. To enable real-world usage and evaluate its performance, we integrate Elf into the column-store DBMS *MonetDB*.

Goal of this Thesis

Since MonetDB uses its own data structure to store and algorithms to process data, swapping in Elf as a substitute is not a trivial task. The goal of this work is divided into the following tasks:

- We integrate Elf into MonetDB as a secondary storage structure. This involves the loading of data into Elf as well as executing queries on it. Since Elf does not support *joins* and other operations, we also need to ensure that MonetDB can continue to work with the results.
- Since the issue of storing strings in Elf has so far been circumvented by applying dictionary compression, we explore different ways to natively support strings.
- We derive several improvements for the interoperability of Elf with the rest of MonetDB. In this light we also theorize the influence of some of Elf's features on modern CPUs.
- We evaluate the impact of our improvements as well as how the Elf performs in a real DBMS compared to memory scans.

Structure of this Thesis

To present our work, we structure the thesis as follows. In Chapter 2, we first present different storage models and query types. We then introduce both *MonetDB* and Elf as well as their relation to the issue of multi-column predicates.

We explain our base integration of Elf into MonetDB in Chapter 3. We show how we interface Elf with their internal query language and showcase the different parts of the system which need to be modified. We also discuss two different approaches of dealing with strings in Elf. Following up, Chapter 4 introduces different options of improving the interoperability between Elf and MonetDB with regards to query execution.

In Chapter 5, we show the methods and results of our evaluation of the optimizations compared to our base implementation. We then compare the query performance to MonetDB's own engine and discuss our findings. We finally conclude our work by presenting related work in Chapter 6 as well as summarizing and proposing options to continue future work in Chapter 7.

2. Background

The goal of this thesis is to integrate the main-memory structure *Elf* into the DBMS *MonetDB*. In this chapter we first introduce the concept of different DBMS storage and query types. Then we explain what issues *Elf* and *MonetDB* attempt to solve and how they do that.

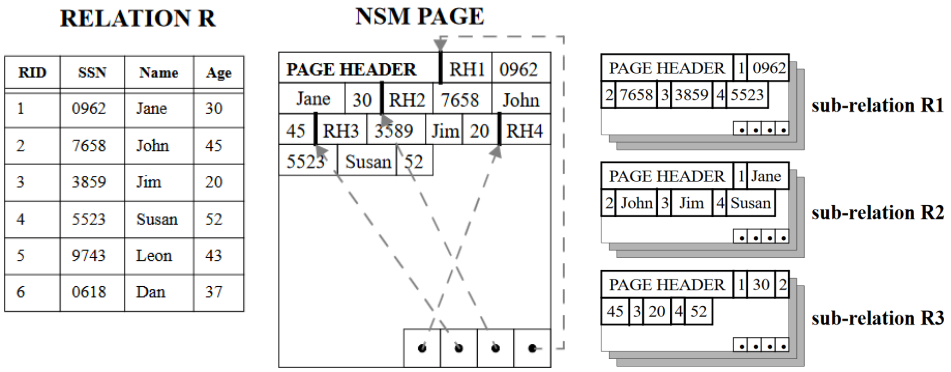
2.1 DBMS storage models

This section introduces a selection of *storage models* for relational DBMSs, meaning how they organize their memory layout to store relations. Important in this context are the two workload types *OLAP* (Online analytical processing) and *OLTP* (Online transactional processing). *OLAP* represents common applications like data mining where little to no updates on the data happen and columns are usually analyzed separately [Pla09], whereas *OLTP* is mostly concerned with many changes across multiple columns.

2.1.1 N-ary storage model (NSM)

In traditional relational DBMSs, tuples are naturally stored consecutively in a page. In Figure 2.1a we show how each tuple is paired with a surrogate or *tuple ID* and concatenated with its surrounding tuples [ADHS01], forming a *row store*. To accommodate for variable tuple lengths, a typical implementation are *slotted pages* [AMH08]. At the end of each page is a list with pointers for each tuple, which avoids having to iterate through all preceding tuples

Row stores are well suited for *OLTP* queries [Pla09]. Slotted pages make inserting tuples into not-full pages simple, while accessing a certain tuple only requires a scan, possibly sped up with indices. Modifying individual attribute values of a found tuple only needs access to a single page. However, row stores always load entire tuples, even though that might not be necessary; computing an aggregate on one attribute thus has to fetch the entire relation. This wastes I/O bandwidth and pollutes the CPU cache, which usually relies on spatial and temporal locality to pre-fetch data [SBKZ08].



(a) Relation and page layout with NSM

(b) The same relation stored with DSM

Figure 2.1: Page-layout differences between NSM and DSM, taken from [ADH02]

2.1.2 Decomposed storage model (DSM)

To better accommodate *OLAP* access patterns Copeland and Khoshafian proposed *column stores* [CK85]. Here, each relation is decomposed into its attributes. The values of each attribute are then stored together as a column, alongside a surrogate as its *tuple or record ID*. 2.1b shows the different memory layout compared to *row stores*. To avoid the overhead of storing this ID for each column entry, densely packed columns may omit them. Instead they are implicitly given by an entry's index [ABH09].

This storage model reduces the amount of disk I/O when only accessing a subset of attributes. Since all values are consecutive in memory, CPU pollution during sequential column scans is also not an issue [SBKZ08]. However, accessing individual tuples or materializing intermediate query results needs to reconstruct tuples from multiple columns, which should be delayed as long as possible [Aba08]. This requires a join on the tuple IDs if a column is not dense anymore, for example due to a select. Insert operations also need to touch multiple pages instead of only one, since all columns need to be modified. This brings another issue: whereas inserting a tuple into a *row store* does not have to move tuples around within pages, *column stores* may need to insert attribute values at a certain index or lose a column's density property [ABH09].

Independent of the use case, columns make easier use of compression like *run-length* or *dictionary encoding* due to their high data locality [Pla09] [Aba08]. Due to these properties *column stores* are ideal for scenarios where updates on tuples are rare and many operations operate on entire columns, like *OLAP* features them [SBKZ08].

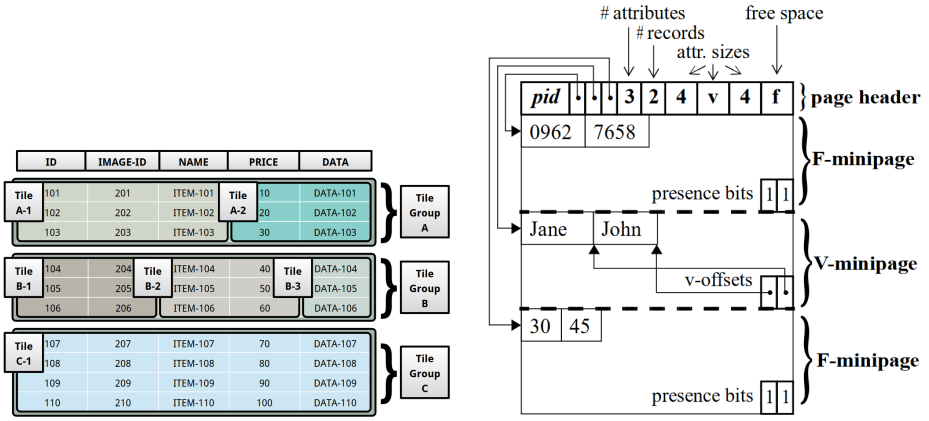
2.1.3 Alternative storage models

Under the premise that neither *column* or *row stores* are suited for workloads combining *OLTP* and *OLAP* known as *hybrid transaction-analytical processing (HTAP)*, Arulraj et al. propose a combination of both to mitigate the respective other's weaknesses [APM16]. When tuples are first inserted into a database, it is more likely to be accessed briefly after insertion than later on. They propose a tile-based storage model illustrated by 2.2a, grouping data on both tuples and attributes. As the

likelihood of individual tuple access decreases, the corresponding tiles shift towards more narrow, *OLAP*-friendly storage.

Partitioned Attributes Across (PAX) is another hybrid model proposed by [ADHS01]. Here the base layout is that of a *row store*. However, as 2.2b shows, each tuple holds multiple attribute values. This reduces cache pollution when scanning over individual attributes.

PAX pages are restricted in that attributes stay grouped, which may result in unneeded data to be loaded. A generalized version called *Data Morphing* allows arbitrary grouping of attributes within a page [HP03]. The *HYRISE* storage engine uses a similar layout; however, its algorithm to find partitions differs [GKP+10].



(a) A physical FSM tile [APM16] (b) Multiple attributes in a *PAX* tuple [ADH02]

Figure 2.2: Alternative storage models

2.2 MonetDB - A column-store DBMS

MonetDB is an open-source DBMS developed at the CWI (Centrum Wiskunde & Informatica) and University of Amsterdam [BK94]. It was primarily designed to handle data warehouse workloads and is based on a column-store architecture [IGN+12]. To enable different applications *MonetDB* employs its own extensible algebra, operating on basic primitives. This section provides an overview of *MonetDB*'s architecture and design.

2.2.1 Data model

MonetDB uses a decomposed storage model with the *Gremlin Database Kernel* at its core. Each column of a relation is represented by a *Binary Association Table (BAT)* [BK99]. Each BAT is made up of two columns labeled *head* and *tail*, storing value pairs called *binary unit (BUN)*. To hold differing values the tail column can be parameterized on either built-in or user-specified types. The head column stores the record's *object identifier*, linking together BUNs belonging to the same original tuple. Figure 2.3 shows an exemplary mapping of two tables, *order* and *item*, onto six distinct BATs. The OIDs are assigned densely and in ascending order upon

creation. Thus base BATs do not need to actually store the head column, instead only providing an offset for the first record [IGN+12]. In such cases BATs consist merely of their header and a traditional array. This lends BATs the typical cache friendliness of column stores.

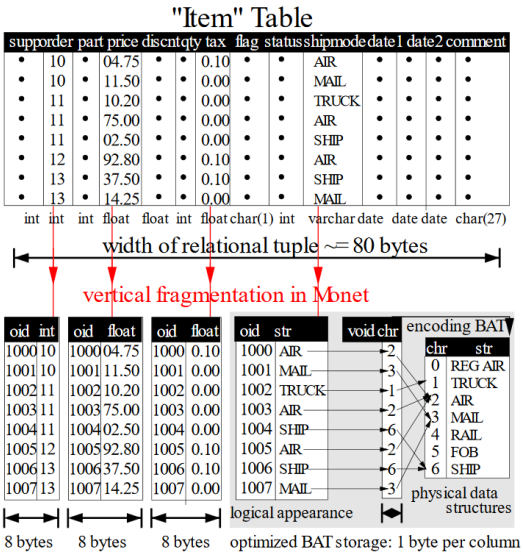


Figure 2.3: Mapping relational tables onto *BAT*s with materialized OIDs

To extend the built-in type system, *MonetDB* allows users to define custom *atomic types* via an ADT system. These can be derived from existing types to simply provide semantics or newly defined, in which case functions for conversion and internal operations need to be defined [BK99]. These as well as other custom operators are to be implemented in C and announced to the DBMS via bindings in its internal query language (see Section 2.2.3). Fixed-length types are stored directly in the tail column, whereas variable-length types such as strings are located in a heap instead, with the tail holding an offset into it.

The necessary reconstruction of tuples from individual BATs is delayed as long as possible [IKM09]. If the density property has not been violated the join only has to perform an index lookup instead of a more costly alternative like hash-joining. *MonetDB* can also use the cheaper lookup even when the participating BATs are no longer dense, but the order of their BUNs is still aligned. This happens when the executed operators are tuple-order preserving, like the relational *select*-operator.

2.2.2 Architecture

MonetDB architecture, as Figure 2.4 shows, consists of two layers separating query processing and execution [BK94]. The top layer may be composed of multiple front-ends responsible for communicating with application programs via TCP/IP and mapping requests onto Monet's internal query representation [BK99]. Front-ends for, amongst others, SQL and XQuery exist [BGVK+06]. The bottom layer contains components related to query execution. This includes the MAL interpreter and optimizer (see Section 2.2.3), the database kernel, and the storage manager. It also harbors a parallelization unit [GKS16]. [HSP+13] extends this to add hardware-oblivious parallelism with the help of *OpenCL*.

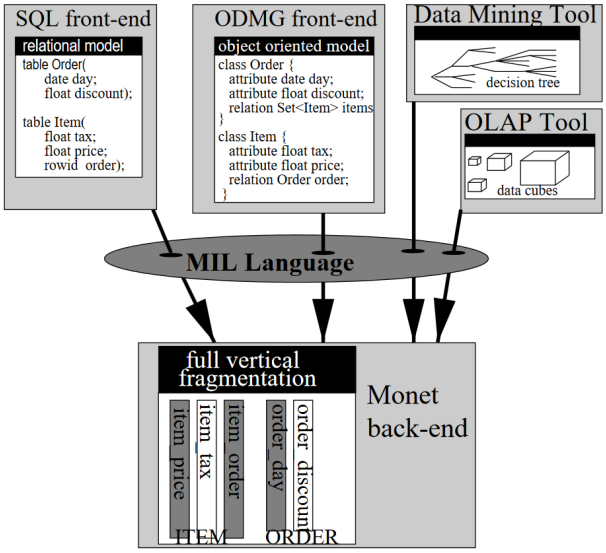


Figure 2.4: *MonetDB*'s two-layer architecture [BK99]

The storage manager coordinates memory at BAT level with *BAT buffer pools* [MBK00b]. Intended as a main-memory DBMS, BATs are always loaded completely into memory or not at all if not needed. To still be able to operate on larger BATs and relations beyond the main-memory capacity they are stored persistently using virtual memory [BQK96]. Large heaps may be mapped as *memory-mapped files*, which are automatically loaded into main memory when an address mapped to them is accessed; this is enabled with the page fault mechanism of paged operating systems. This allows dataset using most of the CPU-architecture's address space. Since certain access patterns may be predictable, mapped heaps can be marked to buffer with different strategies such as *pre-fetch* or *sequential*. Both load succeeding pages upon fault, but *sequential* also marks already accessed pages for swap-out to free main-memory. For datasets just barely not fitting into main-memory, *MonetDB* also relies on the operating system's automatic *page-swapping*, if available [GVK+14].

2.2.3 MonetDB assembly language (MAL)

The glue between front- and back-end of *MonetDB* is the *MonetDB assembly language*, in previous versions known as *MonetDB interpreter language*. It provides every front-end a means to map its operations onto Monet's logical model. To achieve this it can be extended with user-defined *modules* containing custom types and operators [BK99]. These may be loaded on startup or when needed.

The language structure can be split into the four categories *atomic types*, *operators*, *iterators*, and *accelerators* which we address in the next sections.

2.2.3.1 MAL type system

MAL's type system is close to that of its implementation language *C* [BK99]. The types set up on the base types of *MonetDB*'s *Goblin Database Kernel (GDK)*. As a base it offers the standard integer and floating-point types, complemented by a 128-bit integer type if the target architecture supports it. An arbitrary-pointer is also present, although it is forbidden to store hard pointers to data in BATs to

allow Monet to move them in memory (see [BK94]). These types are fixed in size and as such are stored directly in a BAT's tail column. The build-in string type is not, instead it needs to be stored on the associated heap. This enables them to be dictionary compressed by default, storing only the string's index in the column.

All the mentioned base types of Monet directly map to types in C. To allow additional types with semantic meaning, users may also define types which map to a different MAL type. This helps communicating the purpose of variables and parameters (see Section 2.2.3.2). The kernel itsElf uses this to refer to BAT and object identifiers, which are represented by the C types *int* and *long int* respectively. The BAT type is specially treated: since a BAT is parameterized with two types for its head and tail column, every parameterized BAT is a valid type as long as the parameters themselves are valid types. The BAT identifier does not carry any of this information; it is merely a cache ID, which can be used to retrieve the actual BAT if needed.

In addition to the standard types *MonetDB* offers two meta-types: *VOID* and *any*. To avoid unnecessary overhead, head columns for base BATs are only materialized if necessary. To signal that a column is not (yet) materialized *MonetDB* uses the type *VOID* (virtual OID). When an operation needs to break the denseness of a BAT it creates a BAT containing OIDs in its tail column. Tail columns themselves may also carry virtual OIDs, resulting in an effectively empty BAT.

The second meta-type is *any*. Since operators may accept more than one type of data MAL allows the concrete type to be replaced by *any*, which also extends to BAT type parameters. To also allow scenarios where the actual type is irrelevant but two or more parameters must match in type, *any* may be numerated.

User-defined types are declared and made available to MAL in their respective module. Unless they inherit their implementation from an already existing type they must be accomplished by functions for converting them to and from string, reading and writing to client streams, comparing and hashing them as well as what represents an unknown value. Additional functions to determine length and interact with the heap are needed for variably sized types. Inclined users may also override the book-keeping that MAL performs during runtime.

2.2.3.2 MAL operators and language framework

MAL operators exist in three flavors: functions, commands, and patterns. *Functions* are written in MAL and may be implemented at runtime. *Commands* are language bindings for C functions compiled alongside *MonetDB*. The MAL runtime passes pointers to its stack which makes a call to them cheap. *Patterns* are also implemented in C but get direct access to the MAL stack and other information associated with the operator. This makes them more expensive to call but enables features such as variable-length parameter or return lists. All three types may be marked as *unsafe*, meaning that the operation performs changes to the database. The MAL optimizer will then guarantee their order of execution [Ams97].

In addition to types, operators are also grouped in modules. These need to be included by the runtime to enable access. Operators may be overloaded, meaning they share the same name but differ in parameter count or types. Resolution of

these overloads is performed at runtime [BK99]. They also support polymorphism, enabled by the *any* type described earlier. The build-in operators of *MonetDB* mostly wrap the functionality of *GDK* to interact with BATs and build-in types.

Iterators are provided in their own module as MAL hooks. They allow BATs to be broken into smaller read-only views, which may be preferential for efficient processing of large BATs. Accelerators for data structures need to be added manually to the DBMS if necessary.

2.2.4 Optimization Pipeline

To improve the performance of complex queries *MonetDB* has three optimization layers [IGN+12]:

- **Strategical:** This stage optimizes the query before it is transformed into MAL. In the case of SQL it consists of heuristics to reduce its estimated cost like executing select statements as early as possible to reduce the result size. *MonetDB* may also construct indexes if it thinks they may be beneficiary.
- **Tactical:** The tactical optimizer works on generated MAL code. It can be customized and is responsible for transforming the plan to suit the architecture. *Mitosis* may fragment the database if it helps parallel processing, and projections may be chained together to reduce the number of materializations.
- **Operational:** The GDK selects the actual algorithms used at runtime, such as a merge-join over a hash-join if the attributes are sorted.

Since the focus of this work is on integrating an alternative query structure, we give a brief overview of the concrete algorithms MonetDB uses for selects. The basic fall-back algorithm is a *memory scan*, iterating all values in the BAT and adding matching ones to the result BAT. There are a number of circumstances where MonetDB can improve on this, however:

- *Column imprints*: MonetDB offers *column imprints* as a secondary index structure [SK13]. Main-memory DBMSs are mostly concerned with utilizing the CPU instead of disk I/O. To alleviate the issue of relatively slow main-memory access, MonetDB creates bit vectors for each cache line of an indexed BAT. Each bit indicates the presence of either a value or, if the domain cardinality exceeds machine-word size, a range of values in its associated cache line. This allows a scan to fetch relevant cache lines, which most likely would have been done by the CPUs prefetcher anyway, and skip those without relevant entries. However, column imprints only work for integral-type columns.
- *Hash-select*: When a query is looking for an exact match, MonetDB may employ simple bucket-chained hashing. Theoretically, all column types would be eligible; however, MonetDB restricts this to types with a size larger than 2 bytes on 64-bit systems. The select then performs a lookup in the hash table and checks against all values in the respective chain.

- *Order-index*: MonetDB offers to create an *order index* for columns, which stores the indices of its elements as if they were sorted. This helps when determining the lower and upper limits of the scan: a binary search may quickly locate both. The same technique may obviously be applied if the column is sorted to begin with, which may be the case for ID's. If the entries furthermore are dense, i.e. form a continuous sequence, the bounds can be inferred without searching at all. Additionally, the result BAT is then not materialized at all, only storing the bounds, while sorted BATs still need to iterate the value range and at least preliminary store them in the result BAT.

MonetDB builds both column imprints and hash-index on the fly if not present, but only for BATs which are persistent, i.e. permanently stored on disk. The construction of an order-index has to be triggered manually due to its limited use cases. The algorithms also change slightly when a *candidate list* is present, which contains a set of acceptable OIDs: for sorted columns a list merge is necessary, while the regular scans iterate both BAT and candidate list at the same time.

2.3 Elf - A multi-dimensional query structure

The alternative storage models explained in Section 2.1.3 all aim to increase query performance for hybrid workloads by changing the layout. Additional techniques to accelerate scans are not considered, they have to be implemented on top of them. A common choice to speed up searches are indexes [SK13]. Primary indexes place the data directly into a navigational structure leading to higher spatial locality for relevant data but necessitating data duplication for each additional index, whereas secondary indexes only reference the original data. The latter may incur significant I/O overhead on disk-based DBMS to retrieve the actual data and cannot properly prefetch relevant data.

Column imprints are only formed for a single column. This means that a query with multiple predicates has to scan each column and combine them for the final result. Other index types avoid this recombination by allowing multi-dimensional queries directly. Both *kd-* and *r-trees* enable this by walking a tree-like structure and checking against the predicates to descend further or discard branches [SBKZ08]. They differ in what exactly they partition: while *kd-trees* split the data space recursively, *r-trees* group together data points with bounding boxes. This class of indexes however suffers from the *curse of dimensionality*, meaning that the sub-spaces in their leafs become increasingly sparse for higher dimensions and thus degrading their effectiveness [BBK98].

All current tree-based approaches to reduce the influence of said curse only increase the threshold for breaking even with a non-indexed scan. A common issue for them is that instances are grouped by enclosing primitives, which inadvertently also contain empty space. Broneske et al. proposed the *Elf* structure, which groups instances without including empty space [BKSS17].

2.3.1 Concept of Elf

Elf is a tree-like data structure devised for main-memory DBMS. Unlike other indexes like the *r-tree* it does not use geometric shapes to partition the indexed data

[BKSS17]. One of the core ideas of Elf is *prefix redundancy elimination*. This technique is also used by *dwarf*, a compressed structure for data cubes, in liaison with suffix redundancy elimination [SDRK02]. Elf does not use the latter, which decreases the storage size for sparse areas but increases query complexity.

Figure 2.5 contains an example relation transformed into an *Elf* to illustrate its key features. Starting from the first dimension D_1, Elf stores a sorted list of the dimension's unique values called *dimension list*. Each of them is accompanied by a pointer to the next dimension list, holding only those values possible when considering tuples with matching values in previous dimensions. This way any possible *path* in the Elf is unique and duplicate value prefixes of tuples are only stored once. At the end of each path is the tuple ID of the original tuple.

D_1	D_2	D_3	D_4	*Ref*
0	1	0	1	T_1
0	2	0	0	T_2
1	0	1	0	T_3

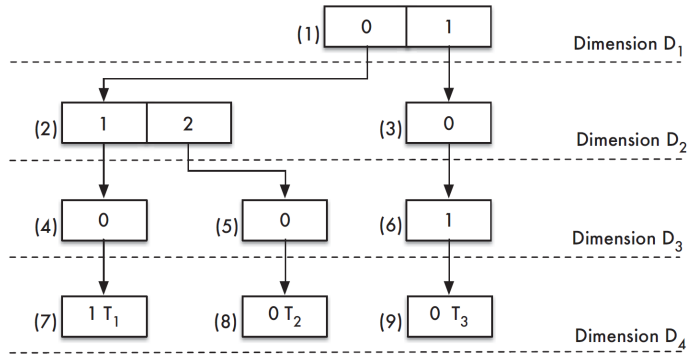


Figure 2.5: An exemplary relation and its representation in an Elf [BKSS17]

Eliminating the prefix redundancy has two effects. For one it compresses the data, reducing its memory footprint. In combination with the sortedness of dimension lists it also helps with efficiently pruning subtrees during a query. Since both dimension lists and window queries are sorted possible query paths do not need to be evaluated once a value larger or equal to the upper window bound has been seen in the dimension list. Additionally, each path in the Elf has equal length, making traversal predictable.

To avoid having data in the Elf that is not or only seldom used it may omit such dimensions. They do not help with narrowing down the search space when traversing the tree during a query and may even prevent relevant dimension lists from being loaded into the cache. The original data can then be accessed using the stored tuple ID.

2.3.2 Optimized memory layout

While the layout in Figure 2.5 reduces needed memory in lower dimensions it does not effectively do so for higher ones. With increasing dimensionality the dimension lists store less values. The worst case is only a single value per dimension list, which happens once no other tuple has the same prefix. In that case no prefix redundancy exists anymore. This actually increases the necessary memory compared to not indexing, considering that each value needs a pointer for the next dimension list. To address this issue Elf introduces *monolists*. Once a path is unique to one tuple it no longer creates new dimension list. Instead the remaining values are stored

consecutively in a list, complete with the tuple ID. This also partially addresses another issue: dimension lists belonging to one path are not necessarily in the same or neighboring cache lines, aggravating the problem of small dimension lists. Reading a consecutive list increases the chances of prefetching by the CPU, even when it is not aligned or confined to a single cache line.

Another possible bottleneck is the relatively large size of the very first dimension list. Since no prefix redundancy elimination has taken place yet its worst case size equals the number of tuples in the relation, which might all need to be scanned for a query. However, Elf uses the fact that it only stores integers; other data types must be transformed, possibly with dictionary encoding. Generally Elf assumes that the first dimension is both ordered and dense, allowing its use as a perfect hash map. This way query values act as a simple index into the *hash list*. Additionally it is no longer required to store the actual values since they are implicitly given by their indexes.

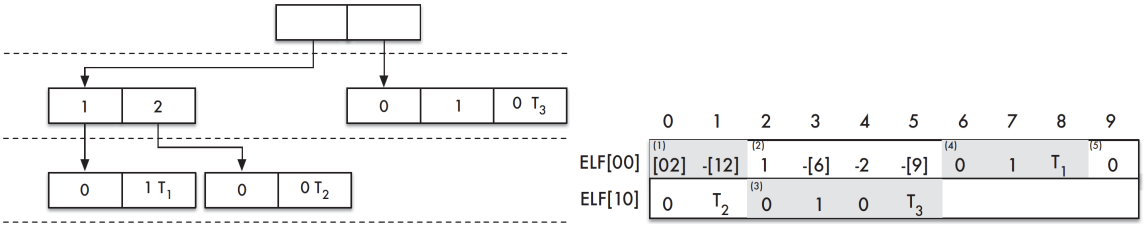


Figure 2.6: Elf with implemented mono- and hash list and its in-memory representation [BKSS17]

Another issue touched on before remains: while monolists are consecutive in memory, a naive tree implementation with pointers does not make optimal use of caching and results in repetitive cache misses for smaller dimension lists. The actual layout thus discards pointers in favor of relative offsets. This also makes modifying operations easier, since unaffected subtrees may stay completely untouched and can be copied without issue. Figure 2.6 shows both the previous example equipped with mono- and hash list as well as the in-memory layout. Offsets are marked with square brackets and list endings with a minus sign. Offsets paired with a list ending indicate that the following list is a mono- rather than a dimension list. Tuple IDs do not need to be marked if duplicates are out of the question; otherwise they form monolists of their own.

2.3.3 Construction

Construction of an Elf depends on whether the data is already present and complete, which may be the case in OLAP scenarios. Although the optimized layout is better suited for querying it is easier to insert new tuples into the original tree-like structure (see Section 2.3.5). Thus it may be preferential for OLTP scenarios to keep two Elfs: its *insert-optimized (IElf)* and query-optimized *linearized (OElf)* layouts. Tuples are then first accumulated in the IElf by following its prefix into the Elf until no match is found. Then a new entry is added to the first non-matching dimension list and more are added to create the full path. If the full path is already present only the tuple ID will be appended.

Following the insertion phase comes the linearization. If the data is already fully present the data may be linearized directly. First the hash list is created; offsets are left blank until the corresponding subtree is constructed. Then each dimension list is inserted right after the previous entry, repeating recursively. In case of a solitary path a monolist is created instead. When constructing the Elf directly the dimension list also needs to be sorted.

2.3.4 Querying

Both IElf and OElf share the basic query algorithms. How the query is performed depends on the type query:

- **Exact-match query:** First the hash list provides the offset by looking up the first query value. Then each following dimension list is iterated until the value is found; if it is not present the query is terminated, otherwise it descends further. When it hits a monolist it is iterated in the same manner, terminating on mismatch. The special cases for hash and monolist are not applicable for IElf.
- **Window query:** Instead of terminating the query upon mismatch or finding a tuple ID, each value falling into the window is followed up. Early termination is still possible once values cannot lie in the window anymore due to the sortedness of dimension lists.
- **Partial-match query:** Dimensions marked as irrelevant do not check their list values and simply act as if the window encompasses the entire domain.

Other query types require to keep track of more information. Checking against a list of values per dimension (*in-query*) necessitates iterating both lists, which may also terminate early if the values are sorted. To also handle queries comparing columns to each other (*column-column query*) a query needs to keep track of previously encountered values. Additionally the column comparison must be ordered so that later dimensions get matched against earlier once so that the values are both in the taken path segment.

2.3.5 Update operations

As already mentioned it may be wise for OLTP scenarios to keep both Elfs in parallel. Inserting new tuples is only performed on the IElf. Once it grows large enough to be a significant slowdown due to its worse cache layout and missing optimizations it may be merged into the existing OElf. This operation is expensive since it requires copying around subtrees to make room for new entries and dimension lists as well as possibly breaking up monolists.

Updating tuples is so far an open problem, as changing values in the Elf may break the sortedness of dimension lists or invalidate monolists. A similar problem exists for deleting tuples.

3. Integrating Elf into MonetDB

To benefit from the advantages of Elf, we present its integration into MonetDB in this chapter. There are several reasons why we cannot completely replace BATs and use Elf as the primary storage structure: first, Elf can only store integer types with less than 32 bits. Since it also reserves the highest bit as an end-of-list flag, even 32-bit integers may not use their full domain. Thus we would require a secondary storage structure for larger or non-integer types. Secondly, Elf so far only supports a small set of queries natively, namely *window* or *partial-match*, *in*, and *column-column*. It lacks algorithms for *or*- and any kind of cross-table queries such as *join*. To still support all SQL queries we defer these to MonetDB's original query engine, which requires us to keep BATs as primary storage and treat Elf as an acceleration structure only.

In this chapter we present the necessary modifications to MonetDB to accommodate the Elf. This requires changes in both the *storage* and *query* modules. The latter further breaks down into parsing, query transformation, and execution. Section 3.2 discusses the changes to these stages, while we consider how Elf gets stored in Section 3.1. We explain the additional challenge of storing non-integer types in Section 3.3 and consider optimizations to the base implementation in Chapter 4.

3.1 Storage of Elf

In this section we consider how and where to integrate the Elf within MonetDB's internals. Additionally, Elf needs to be constructed from data, which we cover in Section 3.1.2.

3.1.1 Extending MonetDB's SQL structures

The first design considerations for our implementation are what to index in an Elf and where to store it. MonetDB divides its storage between the GDK and the SQL layer. BATs reside inside the *BAT buffer pool (BBP)*, which indexes them with a *BAT cache ID*. To control if a BAT is in use, the BBP keeps track of *logical* and *physical* references, keeping a counter for each of them. Physical references indicate

whether a BAT is loaded and actively in use, whereas logical references determine whether a BAT still exists or can be removed entirely. The Elf itself does not need access to BATs during querying if we restrict ourselves to directly indexable values, but we need to retrieve them from the BATs at creation time (see Section 3.1.2). MonetDB itself does not directly store BAT cache IDs for columns, but rather uses *sql_delta*, which encapsulates the base BAT alongside BATs containing inserts and updates.

Since an Elf indexes a set of columns the natural assumption is to associate one Elf with one relation. This restriction is not necessary from the Elf's point of view and may be lifted in the future to easily enable cross-table queries, it allows us to make the Elf a part of MonetDB's *sql_table* structure, as shown in Figure Figure 3.1. As the name suggests it represents an SQL table and contains its name, reference counts, and internal information concerning the type of table. It also carries five *changesets* for columns, indexes, keys, triggers, and members, each containing both a list for present and deleted elements. The two relevant changesets are *columns* and *indexes*. Indexes contain their name, type, parent table, and a list of columns which are part of the index. Considering that we use Elf as an index we may utilize this existing structure to indicate the presence of an Elf; however, we still need to store a reference to the Elf structure. For other indexes such as the *order index* and *imprints* MonetDB stores that information directly with the affected BATs. Since the granularity of Elf structure is on a list of columns and as mentioned we restrict it to one table we store a pointer directly in the affected *sql_table*.

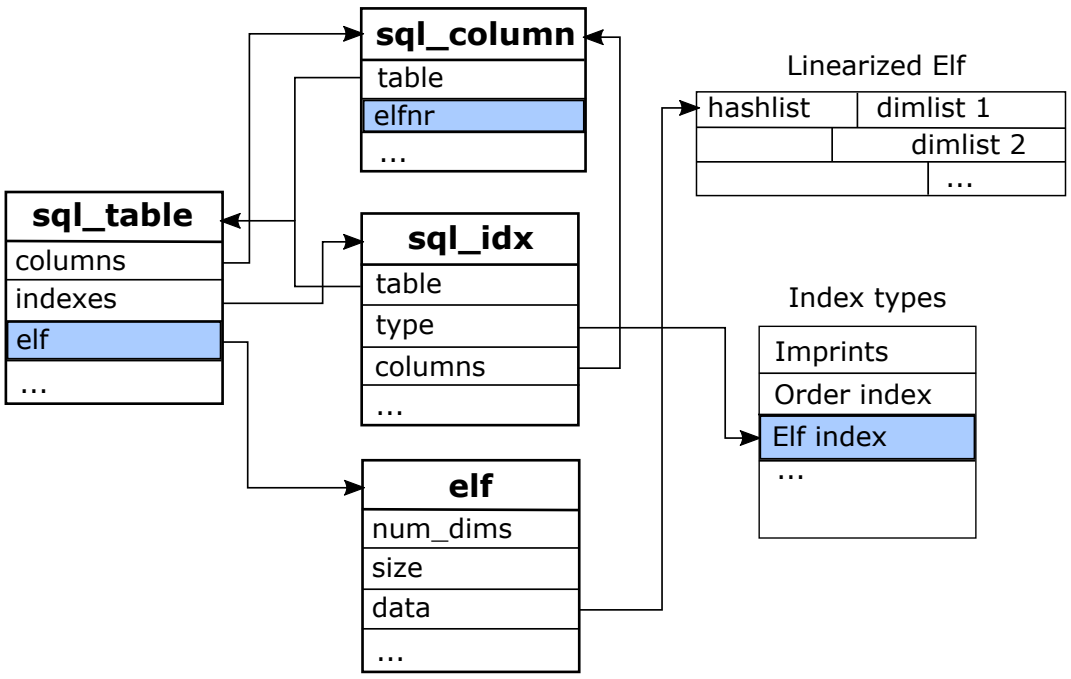


Figure 3.1: Changes to MonetDB's SQL structures to include Elf. Additions are marked in blue

An issue arises in the way MonetDB handles tables internally. The *sql_table* structure is not only used for persistent user tables, but also for system and transactional

ones. Thus allocating memory for Elf bookkeeping alongside table creation wastes resources. One solution would be to check the type of table and only allocate for non-system, persistent tables. Since one design goal is also selectively using Elf we defer the allocation to actually requesting Elf (see Section 3.2.1). For duplicating and destroying SQL tables we deep-copy or deallocate the Elf if it exists at all.

MonetDB stores information about a table's column in the structure *sql_column*. Alongside information about the SQL type it keeps a reference to its parent table and what its index is in the relation. Since we want Elf to index columns in a different order and possibly not all columns we add the column's index within the Elf as a member. We set this index upon creating the Elf index. Unlike the SQL table we do not have to worry about different column types, but we need to propagate the index upon duplication.

To align Elf with existing indexes, we also introduce a new index type indicating that an Elf index is present. This allows us to use the column list maintained by MonetDB instead of having to iterate every column of the table and checking if it has a non-negative index.

3.1.2 Loading data into Elf

Typical DBMS operations on a table include updating and deleting existing tuples and inserting new ones. However, we cannot support the former two due to the lack of (efficient) algorithms to change tuples present in a linearized Elf. Thus we focus on inserting new tuples into Elf in this section.

SQL allows users to insert new tuples via the `insert into` statement, which may contain multiple new rows. As an extension to this, MonetDB provides a `copy from` statement, which loads tuples from a file into the specified table. To avoid the issue of first accumulating tuples and later creating the linearized Elf, we move the linearization to two points: first upon loading data with *copy from*, and second when creating an Elf index when tuples are already present. Additionally, since the lifetime of the Elf ends at the latest with the lifetime of the table's object, we would also have to re-create the index upon starting the system. This behavior may not be desirable in all circumstances, primarily because Elf only resides in main memory. Thus the Elf index has to be re-created when restarting MonetDB and is thus not persistent.

The *copy from* statement is backed by the MAL binding *copy_from*. To efficiently load the tuples, MonetDB employs a bulk loader utilizing multiple threads. The values are copied into the respective BATs, whose size is extended to accommodate the new values. Figure 3.2 shows how we obtain the Elf from the loaded BATs. To create the linearized Elf we need to sort the values repeatedly in different dimensions. The GDK offers a built-in function to sort BATs, which optionally returns the original order of elements. While this would be sufficient for fully sorting a number of columns, we need to reorder a subset of the tuples when creating dimension lists in deeper levels. Thus we copy the values from the BATs into a buffer and fall back to *qsort* from the C standard libray for sorting. To copy the values in proper order we obtain the order of columns from the index stored in *sql_column*. We then iterate the column's BAT, copying integer values into the buffer; more complex types require

a different approach, which we detail in Section 3.3. The tight loop is subject to compiler optimization and may be substituted by a regular *memcpy* (see [IGN+12]). After filling the buffer, we linearize the Elf as described in [BKSS17].

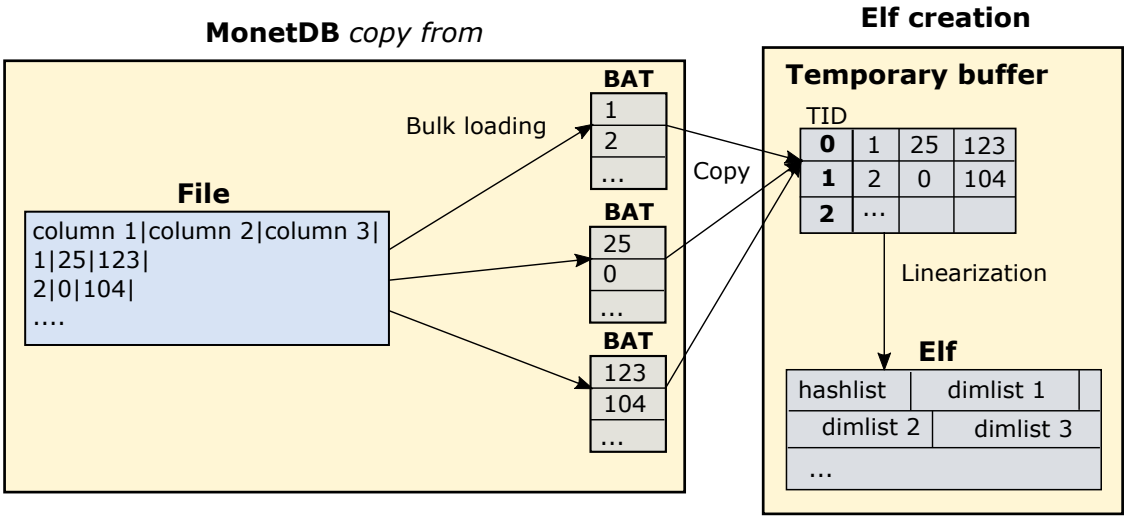


Figure 3.2: Building on MonetDB's bulk loader, we create an Elf via a temporal buffer

The deallocation of Elf occurs upon either destruction of the index or its targeted table, whichever occurs first. Since we do not provide persistency, we only deallocate the memory and reset the indexes in the SQL columns.

3.2 Modifying MonetDB's query execution pipeline

Implementing a different data structure within MonetDB requires modifications to both its query and storage modules. The former consists of three stages: query parsing, transformation, and execution. Since Elf is supposed to coexist with MonetDB's regular column stores we need to somehow express that a relation is supposed to be stored with an Elf index. Considering its performance implications it also must be able to specify which columns and in what order are to be indexed. Query transformation splits in two parts: building MonetDB's internal structure from the parsed query and translating that to executable MAL code. The actual execution of queries does not need to interfere with the rest of the code base as the MAL bindings may reside in their own module. The following sections explain the necessary modifications for each part of MonetDB.

3.2.1 Parser extensions

We need to communicate to MonetDB which table and which columns of that table should make up an Elf. For this we need the support of MonetDB's parser. Since MonetDB supports different types of indexes already, each with its own keyword, we make use of the `create index` statement. Listing 3.1 on the next page demonstrates how to create a new table with Elf indexing. The injection of *ELF* specifies that Elf is to be used instead of imprints or another index type. Following the table name

```
CREATE TABLE table_name (column1 type, column2 type, column3 type);
CREATE ELF INDEX idxname ON table_name(column2, column1);
```

Listing 3.1: Keyword to specify Elf indexing

comes a column list which specifies both what columns are indexed and in which order.

The parser backend then saves the columns position in the *sql_column* and allocates the Elf structure. Since we do not have any data available yet we cannot allocate the memory block for the index but rather initialize the bookkeeping structures.

3.2.2 Relation and expression tree

To translate the parsed SQL query into MAL, MonetDB uses several intermediate representations. The first step is translating the operations into an *expression tree*. It is composed of two structures, *relations* and *expressions*. A relation stores the type of SQL statement as well as pointers to possible sub-statements. It also holds information about whether it requires further processing to finish. To fully evaluate the SQL statement, possible expressions in the `where` clause must be stored too. This task is fulfilled by the *expression* type. It distinguishes between comparisons, atomics, and other operations like aggregates. All expressions of a relation are stored as a list.

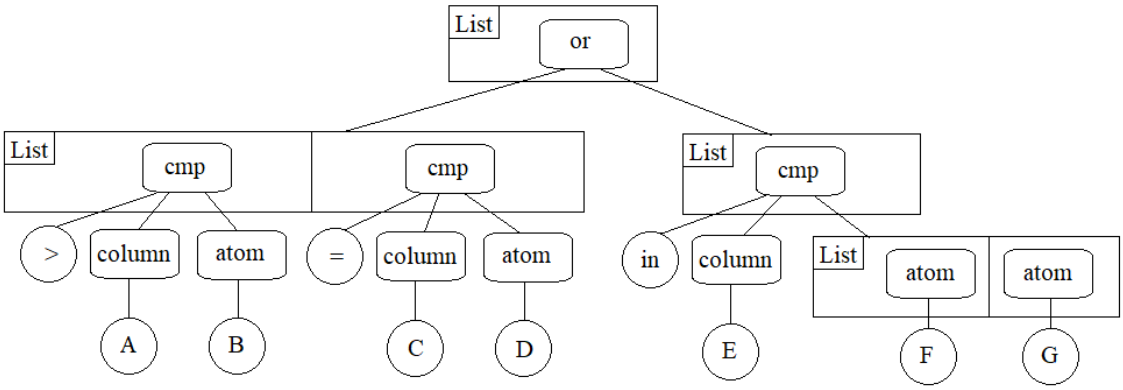


Figure 3.3: Expression tree of the `where`-clause `((A > B) and (C = D)) or (E in (F, G)`

While not of direct interest for Elf, the generation of expressions is subject to MonetDB's optimization pipeline. An example of this are comparison expressions of the form `(A >= X) AND (A <= Y)`. MonetDB generally assumes the logical *and* to be the normal operation between expressions within a list, while the logical *or* is seen as a subexpression. Figure 3.3 shows the expression tree for an example query. However, MonetDB's *select* algorithm offers built-in support for range queries. This is more efficient than the naive execution of two selects and an intersection of the resulting TID lists. Thus MonetDB replaces such an expression with SQLs *between* operator: `A between X and Y`. Elf can utilize this merging of individual expressions, too; we go into detail about such optimizations in Chapter 4.

3.2.3 MAL generation

The core of MonetDB's query engine is the MAL language. Any query will be compiled into MAL statements before execution. These statements are not the same as MAL bindings, rather they *capture* the intended binding and any dependencies the call may have. Since we want to support both queries using Elf and BATs we must consequently provide MonetDB with MAL bindings ourselves to interface with its queries. In this section we present the MAL bindings and generation at the example of a representative query shown in Listing 3.2.

```
SELECT sum(l_discount)
FROM lineitem
WHERE (l_linenumber in(2, 5, 9))
  and (l_shipdate <= date '1993-10-01')
  and (l_shipdate <> l_commitdate);
```

Listing 3.2: Example SQL query to demonstrate MAL generation

3.2.3.1 MonetDB's MAL statements

Neither relations nor expressions translate directly into MAL bindings which can be executed by MonetDB's interpreter. Instead, queries in that form are translated into *MAL statements* first. Each statement is equivalent to one or more MAL instructions. It stores the statement type, organizational data such as the result variable ID, and pointers to the statements which it is directly dependent on: manually adding a value to a BAT is directly dependent on the previous operation performed on said BAT, down to its original creation. It also stores an *instruction pointer*.

The instruction pointer is the bookkeeping structure for MAL. It stores the function and module names of the instruction as well as flags for control flow, garbage collection, and type analysis. It also keeps track of data relevant to the profiler, including time spent on the instruction, and argument counts. In turn it keeps a reference to a *MAL block*, which is the direct equivalent to a MAL binding. It stores the related C function to call, the variables to pass, and additional data related to the optimizer.

As a baseline for comparison Listing 3.3 on the next page shows an excerpt from an optimized MAL plan, generated by MonetDB. The *sql* statements are generally performing catalog operations: *sql.bind* is responsible for binding the cache ID of a column's BAT to a MAL variable. On principal MonetDB binds all columns which are part of the query. *sql.tid* creates a BAT containing the valid tuple identifiers of the table. If it is not partitioned or tuples have been deleted this returns a not-materialized BAT, since the valid TIDs are continuous.

The actual pieces of the query are performed by operators in the *algebra* and *bat* module, which we aim to replace with Elf-specific ones:

- *thetaselect* takes a BAT and optionally a candidate list as well as a value and a comparison operator. A candidate list is a BAT with possible OIDs. It returns a BAT with the OIDs for which the comparison holds and is part

```
1  X_12:bat[:int] := sql.bind(X_8:int, "sys":str, "lineitem":
       str, "l_linenumber":str, 0:int);
2  X_29:bat[:date] := sql.bind(X_8:int, "sys":str, "lineitem":
       str, "l_shipdate":str, 0:int);
3  X_36:bat[:date] := sql.bind(X_8:int, "sys":str, "lineitem":
       str, "l_commitdate":str, 0:int);
4  X_43:bat[:bit] := batcalc.!=(X_29:bat[:date], X_36:bat[:date
       ]);
5  C_9:bat[:oid] := sql.tid(X_8:int, "sys":str, "lineitem":str)
       ;
6  C_46:bat[:oid] := algebra.select(X_43:bat[:bit], C_9:bat[:
       oid], true:bit, true:bit, true:bit, true:bit, false:bit);
7  C_51:bat[:oid] := algebra.thetaselect(X_29:bat[:date], C_46:
       bat[:oid], "1993-10-01":date, "<=":str);
8  C_54:bat[:oid] := algebra.thetaselect(X_12:bat[:int], C_51:
       bat[:oid], 2:int, "==":str);
9  C_57:bat[:oid] := algebra.thetaselect(X_12:bat[:int], C_51:
       bat[:oid], 5:int, "==":str);
10 X_58:bat[:oid] := bat.mergecand(C_54:bat[:oid], C_57:bat[:
       oid]);
11 C_60:bat[:oid] := algebra.thetaselect(X_12:bat[:int], C_51:
       bat[:oid], 9:int, "==":str);
12 X_61:bat[:oid] := bat.mergecand(X_58:bat[:oid], C_60:bat[:
       oid]);
13 X_22:bat[:lng] := sql.bind(X_8:int, "sys":str, "lineitem":
       str, "l_discount":str, 0:int);
14 X_63:bat[:lng] := algebra.projection(X_61:bat[:oid], X_22:
       bat[:lng]);
15 X_66:hge := aggr.sum(X_63:bat[:lng]);
```

Listing 3.3: MAL code generated by MonetDB for the SQL query in Listing 3.2 on the facing page

of the candidate list. The *thetaselect* in line 7 performs *l_shipdate <= date '1993-10-01'*, while those in lines 8, 9, and 10 perform *l_linenumber in (2, 5, 9)*.

- *select* also takes a BAT and a candidate list. Instead of a comparison, however, it takes two values, forming a window. In this case it is coupled with the *batcalc.!=* statement, which performs the column-column query *l_shipdate <> l_commitdate* and returns a bit mask, indicating unequal values. The select then simply creates a BAT listing all OIDs for which the value is unequal.

- *mergecand* merges together two candidate lists. The generated *thetaselect*s each only check for one value; thus the resulting OIDs must be merged to get the final result.

- *projection* in line 14 takes a BAT and a candidate list and returns a BAT with only the values with matching OID. This effectively gives us the query result.

Since the query asks for the sum of *l_discount* MonetDB aggregates the resulting BAT in line 15. The result is then processed and sent to the client, which we leave out since we do not touch this part.

3.2.3.2 MAL bindings for Elf

To illustrate the MAL bindings for Elf, Listing 3.4 shows the same SQL query generated to use the Elf as much as possible. In this section we will walk through the newly added bindings.

```
1  X_43:elfquery := elf.create_query(0x3:ptr);
2  X_47:elfquery := elf.add_col_col_query(X_43:elfquery, 0:int,
       2:int, 5:int);
3  X_49:bat[:oid] := elf.select(X_47:elfquery);
4  elf.destroy_query(X_47:elfquery);
5  X_51:elfquery := elf.create_query_idx(0x3:ptr);
6  X_55:elfval := elf.num2elfval("1993-10-01":date, 21:int);
7  X_57:elfquery := elf.add_window_query(X_51:elfquery, 0:int,
       2:int, X_55:elfval);
8  X_58:bat[:oid] := elf.select(X_57:elfquery);
9  elf.destroy_query(X_57:elfquery);
10 C_76:bat[:oid] := bat.intersectcand(X_49:bat[:oid], X_58:bat
       [:oid]);
11 X_59:elfquery := elf.create_query_idx(0x3:ptr);
12 X_64:elfval := elf.num2elfval(2:int, 8:int);
13 X_65:elfquery := elf.add_in_query(X_59:elfquery, 1:int, X_64
       :elfval);
14 X_69:elfval := elf.num2elfval(5:int, 8:int);
15 X_70:elfquery := elf.add_in_query(X_65:elfquery, 1:int, X_69
       :elfval);
16 X_73:elfval := elf.num2elfval(9:int, 8:int);
17 X_74:elfquery := elf.add_in_query(X_70:elfquery, 1:int, X_73
       :elfval);
18 X_75:bat[:oid] := elf.select(X_74:elfquery);
19 elf.destroy_query(X_74:elfquery);
20 C_81:bat[:oid] := bat.intersectcand(C_76:bat[:oid], X_75:bat
       [:oid]);
21 ...
22 X_83:bat[:lng] := algebra.projectionpath(C_81:bat[:oid], C_9
       :bat[:oid], X_22:bat[:lng]);
```

Listing 3.4: MAL code for Elf for the SQL query in Listing 3.2 on page 20

In the first line we create a new structure *elfquery*. Since MonetDB does not resolve query values until runtime, we need to communicate not only the type of query but

also its possible values to MAL. However, fully specifying the type as an atom would be overkill; we do not need to compute its hash or compare it, as it simply holds the current query. Instead, *elfquery* inherits from *pointer* and gets defined in C, containing the following:

- **Elf pointer**: Direct memory reference to the table's Elf object.
- **Window boundaries**: For a window query we need the upper and lower limit as well as the column it is targeting. This is implemented as a simple array, marking affected columns with *true*.
- **In-list**: For `in` expressions we store a vector with the query values alongside an indication of the column and whether it is `in` or `not in`.
- **Column-column comparison**: This part stores both columns to query as well as the comparator.

The query is initialized by the binding *elf.create_query*, which takes a pointer to an Elf structure. In this we allocate arrays indicating the comparison type for each dimension, but not the members which hold the values themselves. This part is left to *elf.add_window_query*, *elf.add_in_query*, and *elf.add_col_col_query*.

- *elf.add_window_query* (line 7) takes a single value, a dimension, and a comparison operator and initializes the query to form a window matching the inputs.
- *elf.add_in_query* (line 13) adds a new value for an *in*-query. Unlike the other two it may be called multiple times, each time adding a value to the query list. However, this may only happen for the same dimension.
- *elf.add_col_col_query* (line 2) takes two dimensions and a comparison operator for a column-column query.

After adding the desired query, the call to *elf.select* executes the query and returns, similar to MonetDB's equivalents, a candidate list. After execution the query gets destroyed in *elf.query_destroy*. Since each *select* only computes part of the query, we also need *intersectcand* (lines 10 and 20) to obtain the proper OID list.

Beside *elfquery* we additionally introduce the type *elfval*. We use it to convert between SQL values and what we store in an Elf, which are 32-bit unsigned integers. While we could pass query values as untyped values into the query operators, we would need to also pass its original type or risk violating C's strict aliasing rules, which forbid accessing values through a pointer of a different type ([HER15]). Instead we introduce a polymorphic MAL binding *num2elfval*, which calls the appropriate C function to perform the cast. It also comes in a variant taking any value plus its type (line 6). We require this for values which are directly of integral type but rather derived from one, such as *date*, which has its own atomic type but is stored as a simple integer. In these cases the MAL interpreter does not call the binding with the integral type and sees the program as ill-formed. To avoid issues

with strict aliasing when using *elfval*, we also implement the functions MonetDB expects for atomic types to be present, such as comparison and hashing as well as string parsing.

To obtain the result BAT, MonetDB can no longer use *algebra.projection*, as it does in Listing 3.3 on page 21. The reason for this is that *algebra.select* takes the possible TIDs as a candidate list. As such it no longer needs to be projected onto the result BAT. Since the TIDs in Elf are not ordered it is not as efficient to take a candidate list into consideration as it is for BATs, which inherently are ordered and densely headed. Considering that the additional projection in *algebra.projectionpath* for the likely not-materialized TID BAT is relatively cheap, we decided to not include candidate lists into Elf queries.

3.2.3.3 Statement generation for SELECT statements

Since Elf only supports selection we focus on the translation of `select` statements into MonetDB statements. The primary function of interest is *rel2bin_select*. It takes a relation and returns a list of statements which, once executed, perform selects on the underlying BATs. To enable nested queries it may call the entry function recursively. It then iterates over the expressions given in the relation and evaluates them in the function *exp_bin*, which returns a single statement and takes both the optional sub-statements from nested queries as well as the statement which was generated in the previous iteration.

To generate an Elf query instead we first check whether the SQL table on which the `select` operates has any columns indexed. `Select`s with more than one table in the `where` clause are split into independent relations by MonetDB, so we evaluate them one at a time. We then iterate over the expression list, too, whilst calling a modified version of *exp_bin*. Additionally, we need to merge the OID lists which will result from an Elf query. MonetDB deals with this by using previous *candidate lists*. Each select operation on a BAT may be accompanied by another BAT, which shrinks the search space for possible OIDs. This does not make sense for Elf; since we do not operate on BATs we cannot abuse possible density properties. Instead we resort to simply intersecting the resulting OID BATs. To still enable operations that are not carried out on an Elf, such as aggregates, we carry along the generated statement in the same manner as before.

To interpret the expressions is *rel_bin*s task. It distinguishes between seven expression types: *PSM*, *atom*, *conversion*, *function*, *aggregate*, *column*, and *comparison*. *PSM* handles execution of stored SQL procedures, while *function* and *aggregate* deals with SQL functions and built-in aggregates. Of interest for Elf are the remaining four. *Column* resolves the SQL column name to MonetDB's internal structure and the associated BAT. It also emits statements to update the BAT to reflect any changes in the current transaction. Along with *atom*/*conversion*, which bind query values or function results in need of type conversion to MAL variables, it enables us to access the BATs to lookup query values and the column which the column has in the Elf.

Expressions of type *comparison* are furthermore divided by the comparison type. We limit ourselves to those of type *or*, *in*, and regular comparisons. While MonetDB

also supports SQLs *like*, it does not have a direct translation to an Elf query due to its wildcards. Along with *joins* we leave the statements which MonetDB generates. This leaves us with four cases we do handle:

- **Or:** Since expressions in one list are implicitly connected by *and*, parts of the `where` clause connected via *or* are implemented as a list of sub-expressions. `(A and B) or C` thus results in a list containing a single expression of type *or*, which in turn contains two expression lists, one for each side of it. Thus MonetDB performs roughly the same task as for the top-level expression list, which we replace with Elf query creations in the same manner as before. We then form the union of the resulting BATs from the two sub-expression lists for both Elf and MonetDB's statements.

- **In:** *In* and *not in* expressions come with a list of expressions containing the values for the value set. MonetDB iteratively selects matching entries from the BAT, unifying the results. Since Elf natively supports *in*-queries we do not need to create multiple queries. Instead we issue a MAL statement to add the resolved query value to a list, which we will then use to find matching OIDs in one traversal.

- **Window query:** To deal with queries of the form `A` θ `B` where $\theta \in \{=, <, >, <=, >=\}$, we keep tabs of a lower and upper boundary for each query column. For equality, both boundaries have the same value. In the case of unbounded comparisons we choose the minimum/maximum values the Elf can hold as the boundary. We also handle `between` in the same way, since it naturally gives us both limits. A special comparator is $\neq$: there exists no way to encode it in a window query of this type. To solve this we change it into a single-element `not in`.

- **Column-column query:** Elf provides support for inter-Elf comparisons between columns. Thus we only add the comparison type and the queried columns to the query.

3.2.4 Query execution

As a preparation we first consider the type of query. In case of an *in*-query we need to sort the values, because it allows Elf to terminate earlier when matching dimension lists. Next the query creates a BAT to store the OIDs in. We then choose the appropriate query function depending on what has been added to the query prior. During the query we add OIDs matching the query to the BAT. Unfortunately we need to grow it dynamically, since Elf does not feature a suitable heuristic for an upper limit of matching OIDs, whereas MonetDB uses the sizes of the BAT and candidate list. To avoid too many small allocations while keeping we allocate a BAT with one million entries or the number of entries in the Elf, whichever is smaller; MonetDB uses this heuristic when it cannot obtain a more accurate estimate (see function *BATselect* in *gdk_select.c*).

As we mentioned, Elf does not impose any order on the stored OIDs. However, MonetDB expects candidate lists to be sorted, since its own functions mostly pre-

serve tuple order. Thus we need to sort the OIDs before passing it back to the MAL runtime as a result.

3.3 Storing non-integral values in Elf

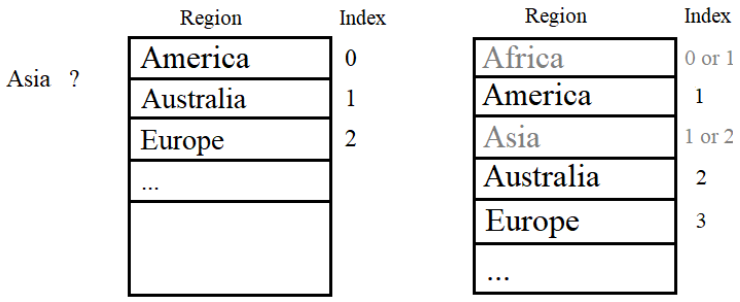
In the original implementation in [BKSS17], Elf only stores 32-bit unsigned integers. However, a DBMS generally wants to store other types such as strings or big integers as well. Thus we need to provide some form of mediation to allow these types to also be indexed.

A naive approach is to simply use the index of a value in its BAT. This works fine when only considering queries checking for equality. However, this does not work for window queries in general and, upon closer inspection, does not function with Elf's early termination approach when querying. The reason for this is that MonetDB does not necessarily store values sorted in BATs. Thus the order of indexes does not necessarily reflect the order of the values they represent. We propose two solutions to this problem: *index-map* and *resolve-comparison*.

3.3.1 Index-mapping to obtain ordering

Index-map keeps the Elf querying the same. To get an integral value for non-integral types we create a mapping of continuous indexes to the values' actual indexes in the BAT. To achieve this we extend the *sql_column* structure by a BAT cache ID, which points to the BAT containing the mapping. Its creation happens before filling the temporal buffer used for linearizing the Elf: when a column's type is non-integral or too large to be stored in the Elf directly, we create a sorted BAT and the corresponding order BAT. For each unique entry we then store the original index, obtained from the order BAT, in the mapping.

However, this alone does not account for values which are not present in the original BAT at all. Figure 3.4a demonstrates the problem of string sequences which are not dense. They cannot be resolved to such an index during query time. In fact there cannot exist an fixed-size index-like string representation without this issue: assuming a cardinality of X we may only map X strings without duplicate index assignment.



(a) The issue of storing strings by index in the Elf

(b) Clamping absent values to nearest neighbor for window queries

Figure 3.4: Mapping strings onto Elf values

Figure 3.4b shows a solution to this problem. Instead of obtaining an index for the value itself, we combine the index resolution with the comparison operator in question. In a less-than query, there is no difference between using the upper bound's index minus one and the index of the next lower value. Consquently, for the queries $A > X$ and $A >= X$ the next lexicographically smaller string serves as the index, and the next greater one equivalently for $A < X$ and $A <= X$. An edge case occurs when the query value is smaller than all stored BATs, in which case we may need an index smaller than 0. To avoid this indices are incremented by one before being stored in the Elf, hence the entry 'America' now starts at index 1. Unfortunately this requires iterating the entire BAT to find such a suitable replacement.

To resolve query values at runtime we introduce two new MAL bindings as well as the *idxquery* type to distinguish it from the second solution in Section 3.3.2.

- *bind_index_map* properly binds the BAT containing the index mapping.
- *resolve_bat_index_map* takes both a storage and an index-map BAT as well as a comparison operator and the value to look up. It then returns an Elf value following the schema in Figure 3.4b.

This solution has a large performance penalty for looking up query values. However, since queries are normally executed there is no additional overhead for *elf.select*.

3.3.2 Resolving indexes at runtime

The second option requires that indices need to be resolved when being compared during Elf traversal. This is a relatively inexpensive operation since the indices grant direct access to MonetDB's heap. Instead of comparing only integers, however, Elf now needs to compare strings. Not only is this more costly itself, accessing the BAT also pollutes the cache. Additionally, it adds more branching to the Elf traversal, since we need to distinguish between string and non-string columns, putting pressure on the CPUs branch predictor. However, this approach does not have the upfront cost of a BAT traversal for each query value.

Similarly to the first solution, we add the type *resquery* to differentiate between the two. We do not require new MAL bindings; however, we need to change both the way Elfs are created and executed. Instead of integers the query and the temporal buffer now store *ValRecord*s, which is the structure MonetDB uses for values whose type it cannot establish at compile time. However, we still store the value's indexes in the Elf. As Figure Figure 3.5 shows, we now perform a a lookup in the BAT during Elf traversal to perform comparisons if the type is non-integral.

This approach has some downsides. For one, we now need to bring the BAT of every queried column into the traversal. More importantly, we can no longer treat the first Elf dimension as a hashlist since the indexes are not guaranteed to be in order. According to [BKSS17] this removes a large advantage of Elf.

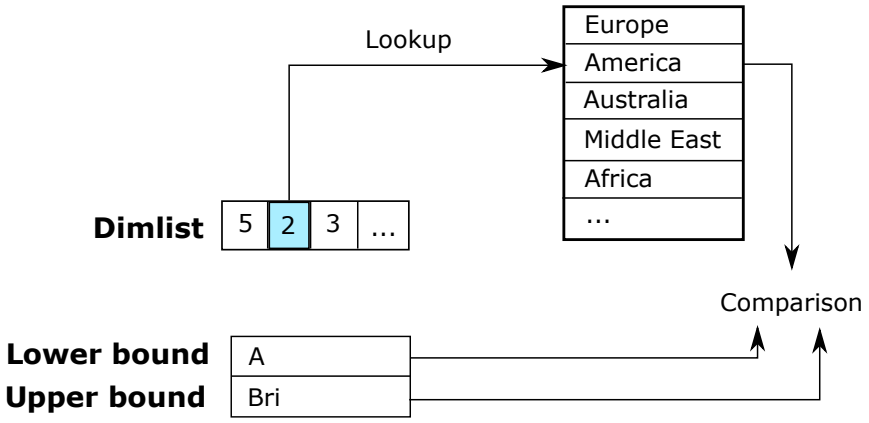


Figure 3.5: Indirection of BAT lookup is needed when storing strings without mapping

4. Query optimization for Elf

It is generally expected of DBMSes to perform their given task as quickly as possible. Thus it is of utmost importance that their implementation employs fast and optimized algorithms and minimizes overhead. The Elf structure already offers optimized traversal. Consequently, the largest optimization possibilities lie in the layers joining MonetDB and Elf together. In this chapter we take a look at how to improve our baseline implementation in Chapter 3.

4.1 Merging different query types for traversal

In the previous chapter, we presented MAL code generated from the example query in Listing 3.2 on page 20. Looking at the MAL code using Elf in Listing 3.4 on page 22 we can see that every condition from the *where*-clause corresponds to one Elf query (see lines 1, 5, and 11). This does not really cater to the main strengths of Elf, which is cache-concious traversal. After every traversal the cache will no longer contain the initial dimension lists, assuming that the Elf in its entirety does not fit into the CPUs cache. In [BKSS17], partial-match queries are already used and their construction trivial: assuming two initial boundaries, successive narrowing for the respective comparison operators results in a window for each column. Since this was used in the original paper's benchmarks, we consider this our baseline implementation. However, merging the traversal of different query *types* is not as straightforward; we discuss the concrete implementations in the following sections.

4.1.1 Merging window queries

Before we take a look at how to traverse a query with multiple types, we first need to look at how to merge them together. While it is trivial to merge a window with an *in* query, ignoring the need to pass along the values, merging queries of the same type is not as easy.

For *window queries*, we may simply adjust the boundaries to the newly added minimum/maximum. However, this only works when storing integers in the Elf. The solution for storing strings and larger integers from Section 3.3.2 does store integers,

but they do not need to be ordered as they get resolved at runtime. Hence we store the actual query values instead of integers. This does not cooperate with the current way Elf checks if a dimension-list value falls into a window; it assumes that the boundaries are inclusive. For comparisons of type $A \leq B$ however, we cannot simply change it into an inclusive boundary. For positive integers, the comparison is equivalent to $A \leq B - 1$ as long as $B \neq 0$, for which the comparison is always false. While a similar adaptation would work for strings, considering that lexicographical ordering maps them onto continuous integers, this becomes cumbersome for floating-point numbers. Taking into consideration that changing a string to its next higher or lower value depends on the encoding and may even require reallocation, a different approach is to change the way we perform a window query. Instead of only storing if a column has a window query, we store what type of window query to expect. Thus, when evaluating against a window query at runtime, we have 9 possibilities: one for each combination of $\geq$ or $\geq$ and $\leq$ or $\leq$, respectively, as well as non-existing boundaries. This also alleviates us from having to specify a *nil* element.

This differentiation into different window types necessarily leads to an increase in branching as a trade-off for increased flexibility. Inspecting the generated assembly in Listing 4.1 on the facing page reveals that GCC 7.1 implements the *switch*-statement as a jump table: in line 4 the case of no window query is handled by jumping past the table, while the actual jump into the table occurs in line 13. This means that before the actual comparison takes place we perform an indirect jump, hoping that the branch prediction masks most of the performance penalty. We investigate whether this has a significant impact on the query performance in Chapter 5. For readability purposes we compiled the presented assembly without optimizations enabled; while the index computation gets improved, the concept of a jump table is unchanged.

4.1.2 Merging *in*-queries

To support queries asking for specific values to be present rather than a range of values, we provide an Elf query with a list of values for each column which is then traversed alongside the respective dimension lists. Merging together two concurring *in*-queries is simple: if the column already has a list, we intersect it with the new one. However, their complement in the form of *not in* queries requires more precaution, as shown in Figure 4.1: if a list is already present, but for an *in*-query, we need to remove all values which are present in both. This procedure is also necessary in the inverted case.

We now illustrate the general algorithm for traversing a dimension list for which both an *in*- and partial-match query is specified. To check as little values as possible during traversal, we run along the value lists of the query in lockstep with the dimension lists. There are now three termination conditions: the dimension list ends, the list of *in*-values ends or we exit the window. Since both lists are sorted, we may compare the current values for each one and check for equality. There is a slight difference when we have *not in* values: finishing the list is not a termination criterion, but we only need to check if the dimension lists lie within the window after it is over.

```
1  .window_match_res:
2          ...
3          movl    (%rax), %eax
4          cmpl    $31, %eax
5          ja      .L34
6          movl    %eax, %eax
7          leaq    0(,%rax,4), %rdx
8          leaq    .L36(%rip), %rax
9          movl    (%rdx,%rax), %eax
10         movslq  %eax, %rdx
11         leaq    .L36(%rip), %rax
12         addq    %rdx, %rax
13         jmp     *%rax
14         .section        .rodata
15         .align 4
16         .align 4
17 .L36:
18         .long   .L35-.L36
19         .long   .L34-.L36
20         .long   .L34-.L36
21         .long   .L37-.L36
22         ...
```

Listing 4.1: Jump table implementation of the window type distiction

4.1.3 Merging column-column queries

Column-column queries in Elf are realized by associating a column with both a comparison operator and the index of the column to compare it with. As such we limit ourselves to supporting column queries for different dimensions; supporting multiple comparisons per column would increase both complexity of the traversal and memory consumption per query. Seeing as column-column queries within one table are relatively rare, we decided this to be out of scope and it thus requires merging OID lists instead.

We support the six common comparison operators, each with its own unique termination criterion: for *less*, *equal*, and *less-or-equal* we may stop traversal once reaching a value larger or equal than the target value. *Greater* and *greater-or-equal* cannot be terminated early, but the check may be dropped once the threshold is reached, removing branching from loop.

Combining this with both *window* and *in*-queries increases the complexity quite a bit. While the number of termination criteria does not change, we need different traversal loops depending on whether the column comparison has been met or not, the value list is over or the dimension list values passed the lower window boundary.

Listing 4.2 on page 33 shows the MAL code when merging together the different queries. Not only is there only one traversal in line 11, but there is also less overhead in the form of query creation. We also no longer need to intersect OID lists; in this

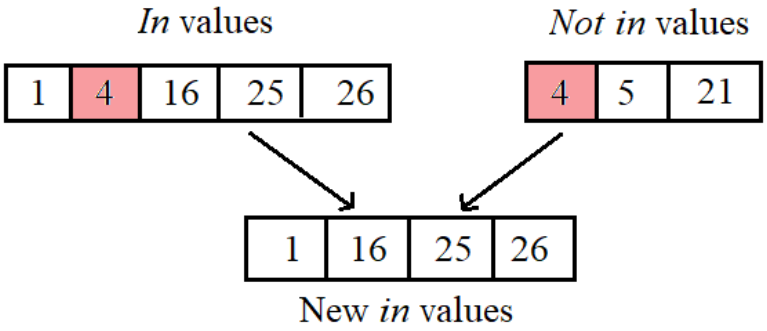


Figure 4.1: Joining together *in* and *not in* values for joined traversal

concrete example we do not intersect at all, since there is no part of the query not handled with the Elf.

4.1.4 Optimizing interop with MonetDB query execution

A big part of integrating Elf into MonetDB is the coexistence with it already-present query engine. The main part of this is fusing the OID list resulting from an Elf query with the results of possible other queries. In Listing 3.4 on page 22 this happens in lines 10 and 20, although in that specific case we fuse OID lists from individual Elf queries. MonetDB uses the function *bat.intersectcand* to intersect *candidate lists*. This does, however, come with the requirement that the input lists are sorted. Unfortunately, Elf does not store TIDs as such. Thus, we have to sort the OID list before passing it into the intersection function, as shown in Figure 4.2.

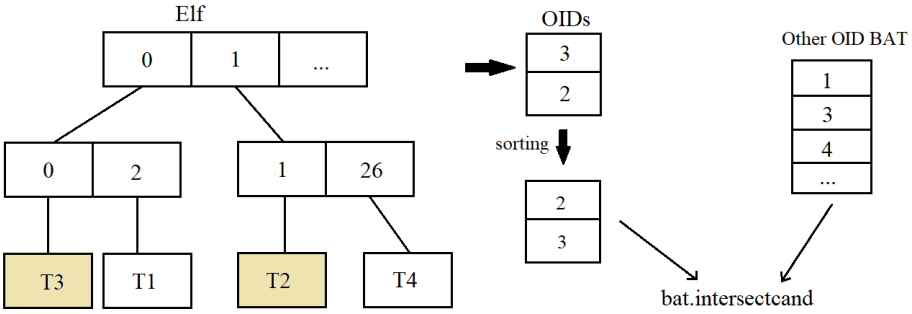


Figure 4.2: OID lists resulting from Elf queries need to be sorted before further processing

While it is sensible for MonetDB to operate with candidate lists, since their operations for the most part retain tuple order, the additional cost of sorting the OIDs makes Elf infeasible for queries with larger results. While Elf shines in scenarios with low selectivity, this may still result in large numbers of OIDs. To alleviate some of that cost, we instead stop seeing the query result as a candidate list and instead as a regular BAT. MonetDB then allows us to use *algebra.intersect*, which determines the best algorithm to use at runtime. This does not directly produce a usable OID BAT; instead it returns the OIDs of the original OID BAT which have a

```
1  X_43:idxquery := elf.create_query_idx(0x3:ptr);
2  X_47:idxquery := elf.add_col_col_query(X_43:idxquery, 0:int,
       2:int, 5:int);
3  X_51:elfval := elf.num2elfval("1993-10-01":date, 2:int, 21:
       int);
4  X_53:idxquery := elf.add_window_query(X_47:idxquery, 0:int,
       2:int, X_51:elfval);
5  X_57:elfval := elf.num2elfval(2:int, 8:int);
6  X_58:idxquery := elf.add_in_query(X_53:idxquery, 1:int, 8:
       int, X_57:elfval);
7  X_62:elfval := elf.num2elfval(5:int, 8:int);
8  X_63:idxquery := elf.add_in_query(X_58:idxquery, 1:int, 8:
       int, X_62:elfval);
9  X_66:elfval := elf.num2elfval(9:int, 8:int);
10 X_67:idxquery := elf.add_in_query(X_63:idxquery, 1:int, 8:
       int, X_66:elfval);
11 X_68:bat[:oid] := elf.select(X_67:idxquery);
12 X_8:int := sql.mvc();
13 C_9:bat[:oid] := sql.tid(X_8:int, "sys":str, "lineitem":str)
       ;
14 X_22:bat[:lng] := sql.bind(X_8:int, "sys":str, "lineitem":
       str, "l_discount":str, 0:int);
15 X_70:bat[:lng] := algebra.projectionpath(X_68:bat[:oid], C_9
       :bat[:oid], X_22:bat[:lng]);
16 X_73:hge := aggr.sum(X_70:bat[:lng]);
```

Listing 4.2: MAL code with merged query types for Listing 3.2 on page 20

match. Thus we need an additional projection to obtain the final result. Listing 4.3 on the next page shows the new MAL code for intersecting result BATs. However, this approach may not prove to be beneficial if the result set is sufficiently small, in which case the overhead of generating extra MAL statements and MonetDB's checks for deciding on a concrete algorithm may dominate. We will thus consider both approaches in the evaluation (see Chapter 5).

The *resolve*-variant to enable string storage in so far requires the presence of all BATs during traversal to resolve the stored indices to their actual values. This is somewhat suboptimal; MonetDB's data-flow analyzer has to assume that we use the BATs and thus may be prevented from reordering MAL instructions more favourably. This would be especially beneficial in a multi-core setting, since independent instructions may be executed in parallel. Since we can track which columns actually take part in a query, we simply withhold those that are not required from the selection.

4.1.5 Distributively reordering *where* clauses

Assuming that we merge together queries, our goal is to traverse the Elf as little as possible. However, certain queries prevent us from merging altogether: *where* clauses combined with *or*. Similar to unmerged *and* clauses, we have to form the

```
1 ...
2 X_75:bat[:oid] := elf.select(X_74:elfquery);
3 elf.destroy_query(X_74:elfquery);
4 C_81:bat[:oid] := algebra.intersect(C_76:bat[:oid], X_75:bat
      [:oid], nil:bat, nil:bat, false:bit, nil:lng);
5 C_82:bat[:oid] := algebra.projection(C_81:bat[:oid], C_75:
      bat[:oid]);
6 ...
7 X_83:bat[:lng] := algebra.projectionpath(C_82:bat[:oid], C_9
      :bat[:oid], X_22:bat[:lng]);
```

Listing 4.3: lst:sort-projection

union out of the two resulting OID BATs, and we also have the choice between treating them as candidate lists or not.

To alleviate some of the issues with *or* clauses, MonetDB performs some transformations in its SQL optimizer. This includes changing $A = B_1$ *or* $A = B_2$ *or...* into A *in* $(B_1, B_2, ...)$. This is also favourable for Elf, since it reduces the number of needed traversals from n to 1. This concept can be extended to other clause combinations. Clauses of the form $(A = X)$ *and* $(B = Y$ *or* $C = Z)$ would require a total of three traversals; however, using the distributive law we may reshape it as $(A = X$ *and* $B = Y)$ *or* $(A = X$ *and* $C = Z)$. This reduces the number of traversals down to two, at the expense of having to check for A an additional time.

4.2 Optimizing Elf traversal

So far, each of the query types *partial-match*, *in*, and *column-column* existed in isolation. That means a traversal could focus on checking the respective conditions efficiently by skipping obviously unfit elements of a dimension list by utilizing their sortedness. This results in nine different versions for the traversal of a dimension list: one for *partial-match*, *in*, and *not in* queries respectively and six for the possible column-column comparisons. To still keep the number of checked elements as low as possible, queries containing multiple types require a unique function to handle each of the possible combinations, resulting in 41 functions.

Although it seems beneficial having to load the least amount of dimension list elements, there are some possible downsides to this. First, the comparatively large number of functions necessarily increases the code size. Our implementation of these functions results in sizes around 1KB, which may put pressure on the instruction cache. Secondly, and probably more severely, deciding which function is the correct one to use leads to quite a bit of branching. Depending on the quality of the CPU's branch predictor this may have more or less branch mispredictions as a consequence, which may eat up the possible gains from the reduction in element checking. An aggravating circumstance is that the average length of dimension lists gets shorter with every column. Thus the usefulness of early termination gets greatly diminished the lower one traverses into an Elf. To benefit from both the reduced checking as well as less branching, we propose two heuristics shown in Figure 4.3 as a means to decide whether to use early termination or not.

4.2.1 Determining cut-off column for early termination

The first heuristic looks at the *average dimension list size*. During construction of the Elf we keep track of the average number of elements per dimension list. We then store the index of the first column for which this average falls below a certain threshold. At traversal time, we switch over from early termination to simply iterating all elements once we reach that column. This relies on the observation that, while the number of dimension lists goes up, their size goes down in later columns. This does not necessarily hold for all cases; when the columns are ordered in a way such that low-cardinality ones occur earlier, the Elf's prefix redundancy elimination may lead to later dimension lists having a larger-on-average size.

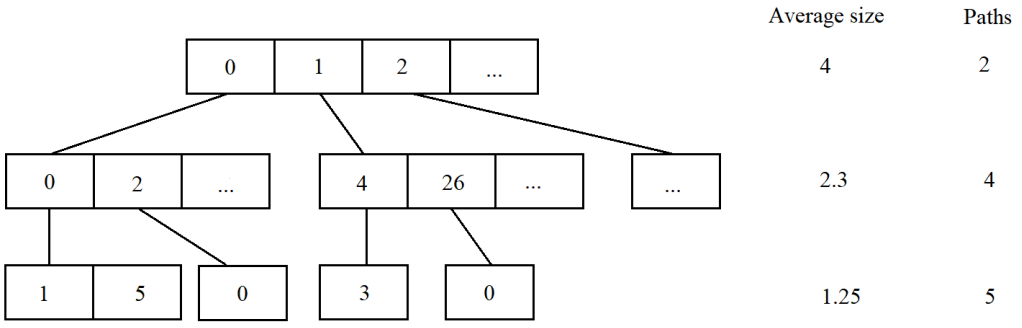


Figure 4.3: The two heuristics for ending early termination: dimension list size and compactness of paths (in yellow)

The second heuristic is also a threshold, but instead of tracking the *size* of dimension lists, we observe how "unravelled" the Elf becomes. We look at the ratio between the number of TIDs and the number of paths for a given column in the Elf as an indicator for this. This is necessarily monotone, since paths may split up but cannot recombine. This value is highest for the hashlist and lowest for the last column, with a minimum of 1, assuming no duplicate entries, and the number of TIDs as its maximum, assuming a hashlist size of 1. Thus we define the *compactness* of an Elf column as

$$c_{Elf,d} = \frac{\frac{|TID|}{|Paths_d|} - 1}{|TID| - 1}, \tag{4.1}$$

yielding a range of $[0, 1]$. We then specify a threshold in that range after which we no longer use early termination in the same manner as for the first heuristic.

None of the two heuristics is perfect; while the *compactness* attempts to remedy the lack of guaranteed monotony, it lacks the concrete relation to dimension list size, as it is only a surrogate. The threshold needs to be chosen with respect to the total number of TIDs in the Elf. However, the compactness is less prone to changes in column order.

4.2.2 Small string optimization

In Section 3.3, we discussed how to efficiently query non-integral columns with Elf. We made the general assumption that strings are of variable length and larger than a machine word. In some relations, however, it may occur that strings are short and

fixed-sized, possibly being used as flags or abbreviations. If the length is smaller than four (i.e. the number of bytes per Elf entry) then we may store them directly in the Elf without having to resolve them at runtime as in Section 3.3.2.

When creating the Elf from parsed BATs, we check the type of column; eligible types are *char(n)*, where $n \leq 4$. To avoid collision with Elf's end-of-list bit, we require the 4th character, if present, to be of ordinal value less than 128, which are all standard ASCII characters. Then, instead of storing an index, we convert the string such that the first character corresponds to the highest byte as the Elf value: the string *abc* turns into the number $97 * 2^16 + 98 * 2^8 + 99 = 6382179$, assuming standard ASCII encoding. This ensures that the lexicographical ordering is upheld.

At query time, we perform the same conversion for strings occurring in queries instead of an index lookup, avoiding most of the overhead otherwise present to prepare a query. Similarly, these strings do not need to be resolved during Elf traversal either.

5. Evaluation

In this chapter we discuss the evaluation of our work. This encompasses our setup, the performance criteria, and findings of our work. To do so we analyse the query runtime of MonetDB and compare it to both the baseline integration of Elf as well as the different variants described in the previous chapters.

5.1 Evaluation setup

In order to properly judge the adequacy of our implementation, we need to compare it with MonetDB in its original state. Since the primary focus of Elf lies on query performance, our evaluation consists of measuring the completion time of queries. This will allow us to not only gauge the quality of our different optimizations over the Elf's base implementation, but also compare Elf directly with sequential BAT scans, either accelerated or not, in terms of speed. While certainly interesting for a DBMS, we decided to not evaluate the additional memory consumption of Elf beyond cursory observations.

The following sections detail our choice of benchmark and evaluation procedure. We also attempt to predict the effect of our optimizations.

5.1.1 Dataset and selected queries

To have consistent and comparable findings, we decide to use the TPC-H benchmark [Cou14], which the original Elf paper also uses [BKSS17]. In their evaluation, they operate on *compressed* data, which allows them to stay clear of strings entirely. Since a large part of our integration revolves around the issue of strings, however, we do not perform dictionary compression beforehand and instead load the data directly from file.

To judge the effects of different dataset sizes and queries, we selected th following scale factors and queries:

- Scale factors(SF): 0.01, 0.1, 1, and 10. This gives us a good estimate of when the Elf offers benefits compared to MonetDB's implementation.

- Queries: Q6, Q12, Q16, Q19, Q22. We selected these queries based on their selectivity as well as their potential for Elf utilization. Except for Q6, the original Elf paper evaluated on different queries; however, these mostly utilize only a single indexed dimension per table and are thus ill-suited to evaluate our optimizations.

The query Q19 poses a difficulty for us: its sub-conditions operate on the same columns. As a consequence MonetDB optimizes it in a way that does not allow for easy conversion to Elf-based querying. Thus, we use a variant of Q19 which does not have different sub-conditions and instead uses only the first of three (see Listing 5.1).

```
1  select sum(l_extendedprice* (1 - l_discount)) as revenue
2  from lineitem, part
3  where (
4  p_partkey = l_partkey
5  and p_brand = 'Brand#12'
6  and p_container in ('SM␣CASE', 'SM␣BOX', 'SM␣PACK', 'SM␣PKG'
       )
7  and l_quantity >= 1 and l_quantity <= 1 + 10
8  and p_size between 1 and 5
9  and l_shipmode in ('AIR', 'AIR␣REG')
10 and l_shipinstruct = 'TAKE␣BACK␣RETURN'
11 );
```

Listing 5.1: Reduced query Q19

We also include a custom, synthetic query to test our small-string optimization, as TPC-H does not feature a query suitable for this; while Q10 does have a predicate on *lineitem*'s *l_returnflag*, the resulting minimal Elf for the table would only consist of a single column. Since it is unrealistic to deploy a multi-dimensional index structure for single-column data, we instead use the query depicted in Listing 5.2 for our comparison as a modification of Q10.

```
1  select count(*) from lineitem
2  where l_shipdate >= date '1993-10-01'
3  and l_shipdate < date '1993-10-01' + interval '3' month
4  and l_returnflag = 'R';
```

Listing 5.2: Synthetic query for SSO evaluation

A large part of the Elf's performance is in what order columns are indexed: columns with lower cardinality feature larger prefix redundancy and thus larger possible savings in memory and traversal time. Since the Elf's main target is data without many changes, we assume that it is realistic to know the cardinalities prior to creating our Elf. Thus we order the columns ascending with regards to their cardinality.

Another aspect of this are columns which are not used by queries. Since they do not offer any selectivity but still need to be traversed, such columns have the potential to

significantly slow down traversals. While it may be less realistic, we also build Elfs which only feature those columns partaking in a query. Furthermore, building Elfs specific to a query offers the opportunity to reorder the columns more accurately, using the query's selectivity instead of cardinality. We thus end up with the following configuration:

- Q6: lineitem(l_quantity, l_discount, l_shipdate)
- Q12: lineitem(l_shipmode, l_receiptdate, l_commitdate, l_shipdate)
- Q16: part(p_type, p_brand, p_size)
- Q19: part(p_size, p_container, p_brand) and lineitem(l_quantity, l_shipinstruct, l_shipmode)
- Q22: customer(c_acctbal, c_phone)
- Q23: lineitem(l_returnflag, l_shipdate)

Note that we omit columns used in a query but not usable by Elf, e.g. for joins or *like*-queries, too. The same holds for Elfs which would only hold a singular column, as would be the case in Q16 for the table *partsupp*. To estimate Elf's performance for the worst case in which nothing is known about future queries or queries may use all columns, we evaluate our test set on fully built Elfs. As a heuristic we order the columns such that those with lowest cardinality appear first, since that utilizes prefix redundancy the most.

To better understand how MonetDB's performance relates to Elf, we list which query columns use column imprints: all columns of Q6, l_receiptdate of Q12, l_quantity and p_size of Q19, as well as c_acctbal of Q22. The full MAL code generated for each query along with annotations of the concrete algorithm for each select can be found in Chapter 8.

5.1.2 Testing variants and expected results

Since most of the TPC-H queries operate on one or more string columns, we divide the variants in two major categories: index-based as *idx* (see Section 3.3.1) and resolve-based as *res* (see Section 3.3.2). When a query contains strings, *idx* determines their index in the BAT or, if it is not present, a suitable replacement matching the comparator before the actual traversal. Hand in hand, the Elf indexes strings by storing their *ordered* index, which allows it to compare them during traversal without having to use the strings directly. On the other side, *ref* stores unordered indices and instead retrieves the strings during traversal for comparison; this removes the need for looking up string indices prior to the query.

To denote whether an Elf contains the entire dataset or only the columns relevant to the query, we append the suffix *min* in the latter case. As mentioned, *min* Elfs have their columns ordered by their selectivity for each respective query, while full Elfs use their cardinality as a heuristic.

Since most of the additions or optimizations we described in the previous chapters may be combined with each other, we have to select a subset of them. In the following we list them and their abbreviation which we will use:

- *base*: This is the base variant, not using any optimization at all.
- *combine (c)*: Instead of creating a new query and subsequent traversal for every predicate, they are combined into one query (see Section 4.1). This is only possible when predicates are combined via *and*, e.g. $a >= b$ *AND* $a <= c$. This also removes the need for merging together multiple TID/OID lists from the traversals.
- *sort (s)*: Since Elf does not support all query types such as *join*, the results need to be processed by MonetDB. Since MonetDB requires OID lists to be sorted, two options exist: either we treat the OID list as a regular BAT and perform an unsorted merge with other OID lists, or we *sort* the TID list we obtain from Elf traversal (see Section 4.1.4). The *base* variant lets MonetDB handle the issue and thus merges OID lists with a merge-join.
- *early termination (et)*: When querying Elf, the presence of an upper limit allows us to stop iterating a dimension list once we reach said limit. However, these lists can get small enough to fit in a cache line, which is 64 bytes for the CPU we used in the experiments. In this case, the additional branching needed for the early termination may outweight the cost of the additional memory access and comparison: a L1 cache hit takes four cycles for the Intel Haswell architecture, while its pipeline has at least 14 stages [Int16]. The *base* variant fully iterates all dimension list, while *et* aborts as soon as possible. To allow for a dynamic switch between terminating early and fully iterating dimension lists, there are two ways to determine a cut-off column: the *dimlist* and *path* heuristic.
  - *dimlist heuristic (d)*: To strike a compromise between never and always terminating early, we compute the average dimension list length and stop *et* after a threshold as described in Section 4.2.1. We set the threshold to three; this was chosen to have a high chance of having the entire list in the L1 cache and a pipeline stall being larger than the accumulated access cost.
  - *path heuristic (p)*: Similarly to *d*, we compute a cutoff dimension until which we terminate early. Instead of the average dimlist size, this variant uses the ratio between the number of unique tuples and paths in the Elf. We chose to set the cutoff ratio to 0.9.
- *index-only (i)*: This variant is only sensible for the *res* Elf. Generally we store integers directly in both *idx* and *res* Elfs, while strings have their indices stored. To get an impression of the performance penalty involved, we test the *res* Elf having to look up all value types in their BATs, not just strings.
- *Small-string optimization (sso)*: To avoid the special treatment of strings altogether, we transform strings with a length less or equal to 4 into 32-bit integers

as described in Section 4.2.2. This allows us to treat them as regular integers both during query preparation and traversal.

Both *i* and *sso* will be looked at separately from the other variants: the former is only applicable for *res*-type Elfs and is only supposed to visualize the cost of indirect access, while the latter is a strict improvement. We also cannot evaluate *sso* with the queries Q6 to Q22 due to their lack of small strings.

A reasonable assumption about the general performance is that the minimal Elfs are generally faster than the fully-built ones: they require strictly less dimension list traversals per retrieved TID. Not quite as clear is the question how the resolve-based Elf matches up against the regular, index-based one. The only advantage it offers is the absence of expensive pre-computing for strings. Thus, one might expect an edge for queries with many string comparisons such as Q19. However, if the cardinality of the string columns is not excessive, larger datasets should favour the index-based Elf due to the lower cost of traversal. If a query does not use string columns, there should be a disadvantage for the resolve-based approach stemming from some extra branching when determining if a dimension list needs resolving. This should be mostly negligible due to branch prediction.

Out of the optimizations, we expect *combine* to improve the base implementation: combined traversals mean less main-memory access and operations at the expense of increased code complexity and some additional branching, which is even mitigated by branch prediction. In addition, it should also reduce the overhead of merging together multiple TID lists, as otherwise would be necessary. We also expect *early termination* to improve the runtime, considering that this was used in the Elf's original implementation. Since these two optimizations do not affect each other, combining the two should yield the best results.

The impact of sorting TID lists before handing them over to MonetDB is somewhat uncertain. On the one hand, merging two sorted lists is faster than two non-sorted lists. On the other hand, MonetDB's sorting function allocates a new BAT for the result. In addition, if we merge with a TID list handled by MonetDB, one of the two lists is already sorted, enabling MonetDB to use a merge join.

We estimate that the heuristics to determine when to stop early termination should have a small positive effect on the query runtime, considering that it removes some of the branching. This should be most noticeable when a query combines partial-match, in, and column-column queries such as Q12. Since we sort the columns by selectivity/cardinality, there should be little difference between the *dimlist* and *path heuristic*. Small-string optimization also should have a net positive effect, split between query preparation for index-based and traversal time for resolve-based Elf.

5.1.3 Evaluation procedure

To limit side-effects, we structured the evaluation in separate steps. For each scale factor we load each table from a text file on disk. This happens via MonetDB's SQL extension *copy into*, which simply deposits the parsed data in BATs. Next, we test each of the optimizations described in Section 5.1.2 paired with the four Elf variants

idx, *idx-min*, *res*, and *res-min*. For the minimal Elfs, we first create the Elf for a query, execute the query ten times, and then discard the Elf again. For the full Elfs, as well as MonetDB's regular query engine, we instead create all Elfs beforehand and then execute all queries in succession, repeated again ten times. After that, we drop the Elfs again. To compare the performance against MonetDB we also execute all queries in the same manner without Elf enabled.

To avoid a series of complications regarding an inconsistent environment, we pin the process *mserver5*, which parses and executes the queries, to the first CPU core. Between each query execution we attempt to flush the data cache by creating a 400MB sized array and looping over it twice, writing the loop index and summing the array contents respectively.

Each query execution tracks three metrics: execution time of select statements, execution time of the MAL routine, and execution time of the entire query. For Elf, select statements are equivalent to *elf.select*. For MonetDB, consequently, we count all MAL statements which compute or are part of the computation we replace with *elf.select*; this excludes any projection or result aggregation. Measuring the MAL execution time in addition to this allows us to evaluate the overhead of creating Elf queries, particularly resolving strings and combining OID lists. Since we also modified the MAL generation layer we can use the difference between MAL and query execution time to evaluate possible overhead in the plan creation, too.

We perform the tracking with the built-in utilities of MonetDB, namely the *trace* statement to time the MAL execution as well as the *querylog* routines for the entire query duration, including parsing. It is important to note that the query times may be influenced by the performance measurement; however, since they are present for every query it is a systemic bias. Additionally, we cannot separate the overhead of column imprints construction from the actual query. Since this only happens once per BAT and scale factor, we instead discard outliers as addressed in Section 5.4.1.

The evaluation happens on a PC with two Intel Xeon E5-2630 v3 with 8 cores and 2.4 GHz each, alongside 1024GB of DDR4-2133 RAM running at 1866 MHz. The operating system is CentOS 7.3 with kernel version 3.10 and the compiler is the system compiler GCC 4.8.5. We use MonetDB's release configuration, which uses optimization level O3, omits frame pointers for smaller functions, and disables debug code like asserts.

5.2 Experiments

To evaluate how well each variant or combination thereof performs, we will first look at the difference between index- and resolve-based Elf. In the following we go through each variant and discuss how they influence the query times.

5.2.1 Index-based vs. resolve-based

The first choice for Elf is whether to use *idx* or *res* as a way to deal with strings. Figure 5.1 shows the query execution times of both variants each for full as well as minimal Elf. It is apparent that *res* is strictly worse than *idx* for all queries and sizes, worsening for larger scale factors. This corresponds to our expectations, seeing

as more data means more resolutions during traversal. The same graph also shows that the minimal Elfs always perform better, once again scaling with size. However, the difference depends on the query, with Q6, Q12, and Q19 showing the largest improvement.

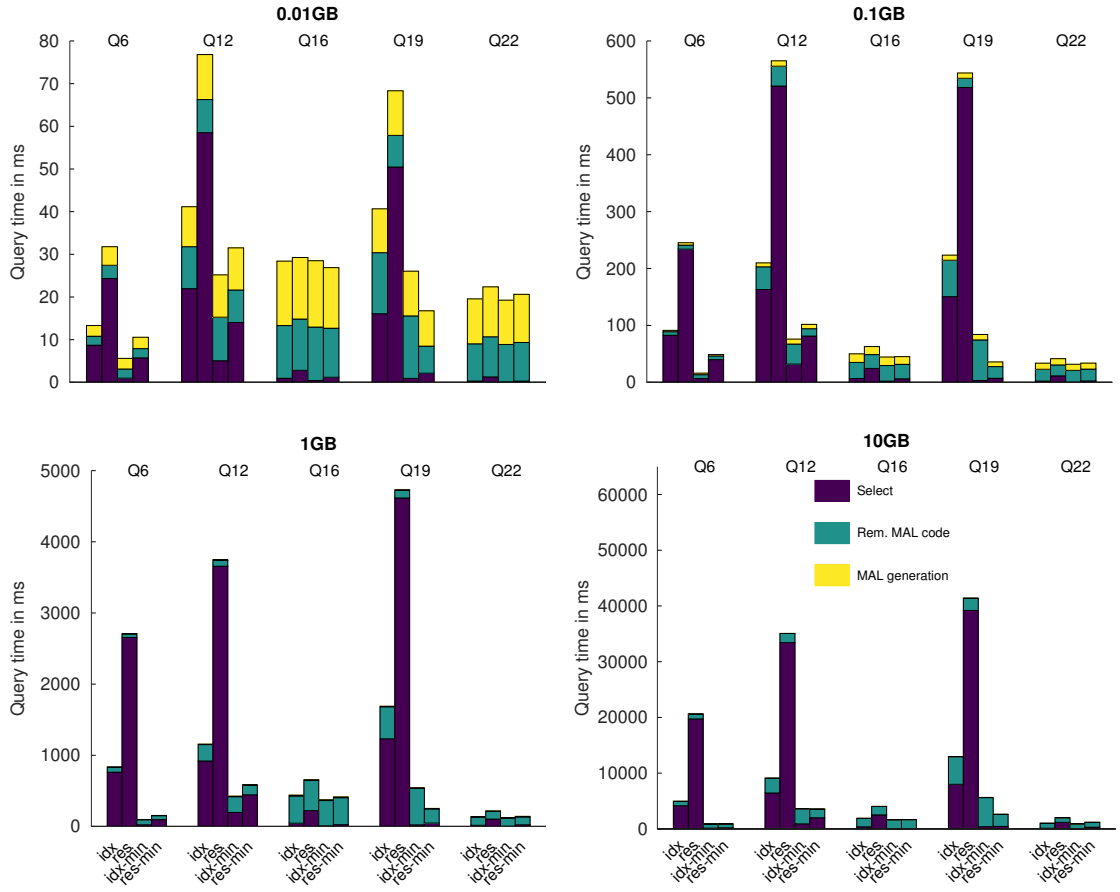


Figure 5.1: Index- vs. resolve-based Elf query times

When considering the minimal Elf, *res-min* appears to perform better than *idx-min* for Q16 and Q19, whereas the two draw even in Q6, which does not have any string predicates. For both Q16 and Q19, the advantage of *res-min* comes entirely from the query preparation, which is generally faster for *res*, while the traversal is slower than for its counterpart. This is most pronounced in Q19, which has many string predicates: here the preparation takes three times as long as the traversal for *idx-min*.

5.2.2 Comparison of optimizations

In this section we will walk through all optimizations, comparing them to our baseline implementation.

Looking at Figure 5.2, it becomes immediately apparent that *sort* does not actually improve the query time. Instead, the time spent on the selection increases drastically, becoming worse with increased dataset size. Contrary, the query preparation time gets reduced, which results from the sorting being part of the Elf traversal. The

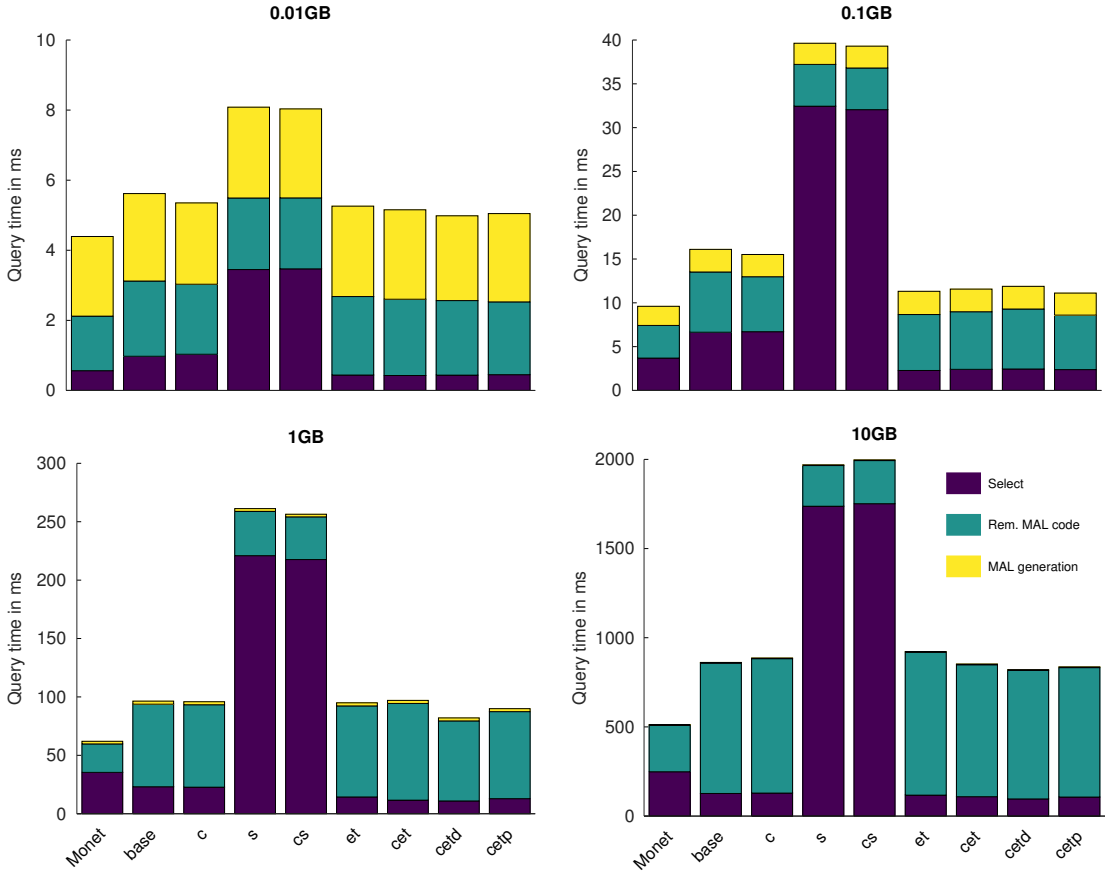


Figure 5.2: Query times for *idx-min* Elf on Q6

reason for this performance drop is two-fold: for one, MonetDB does not offer a routine for in-place sorting, which requires it to allocate a new BAT for the sorted TIDs. Secondly, the merge-join performed when we do not sort the results operates on one already-sorted list. This improves its complexity significantly over the general unsorted case. In addition, after sorting the result, it still needs a merge with other OID lists, resulting in yet another allocation.

The *combine* variant improves both select and query preparation time significantly. The positive effect seems to increase with dataset size, too. This is a direct consequence of the reduced number of traversals, which also means that less result lists have to be merged together. It also reduces the overhead for creating Elf queries, which is a fixed cost for each query. The combination of both *combine* and *sort* better shows the cost of sorting since it only sorts one TID list; while it is significantly lower than having to sort many lists, it does not beat MonetDB's merge-join. The *res* variants profit even more from *combine* due to their more expensive traversal.

Early termination by itself improves the baseline select performance especially for the lower scale factors. This shows that the memory access of Elf and the possible cache miss have a larger impact on the traversal time than possible branch mispredictions and the increased code size due to all the corner cases involved. The lower gains for the larger dataset stems from a non-linear increase in cardinality for the columns, which in turn reduces the number of dimlist entries omitted by early

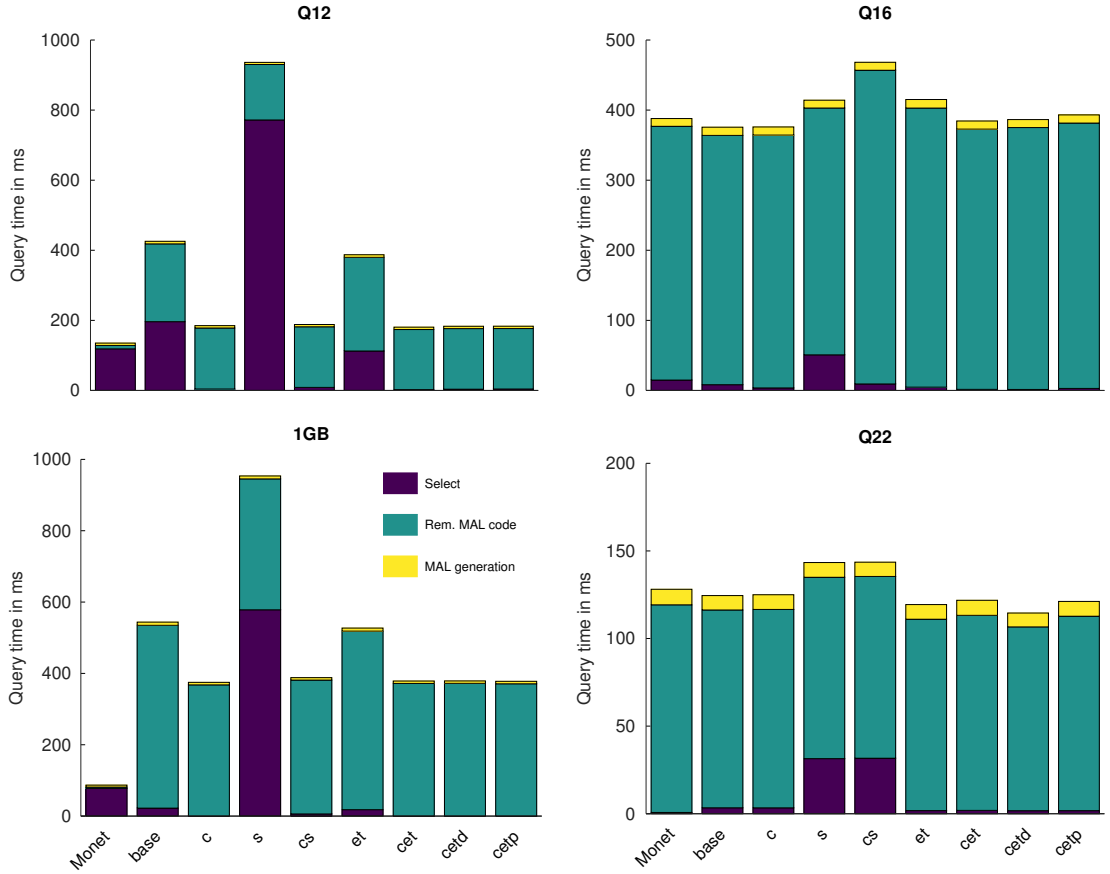


Figure 5.3: Query times of the index-based approach for Q12, Q6, Q19, and Q22 for scale factor 1

termination. Since there are no changes to the selection prelude, the query overhead stays the same compared to the baseline. For *res* and *res-min* early termination does offer an improvement, but it is less pronounced. This is likely due to the fact that a large portion of *resolve*'s cost is caused by fetching BAT parts into the cache. This will always happen when entering a dimension list supposedly storing strings, but subsequent lookups for the same dimension list may already be in the cache due to prefetching, which means that later dimension list entries have a lower comparison cost. A mitigated form of this effect might also occur for index-based dimension lists when a dimension list contains the end of a cache line and the CPU does not prefetch further lines, but it is not guaranteed to occur every dimension list.

Combining both *early termination* and *combine* yields the best performance across all sizes and Elf types. The performance difference when using either the *dimlist* or *path heuristic* is negligible, as well as the difference between the two. This may be explained by the potency of modern CPU's branch prediction, reducing the frequency and thus impact of pipeline stalls, as well as the fact that fetching dimension lists from memory has a far greater cost associated with it, dominating the traversal time.

As figure Figure 5.3 shows, the behaviour of the optimizations is mostly uniform across all queries, with the exception of *combine* and *sort* together for Q22; this

happens since there is only one Elf select, which means no improvement from query combination.

In summary, the fastest variant in most scenarios is *idx-min* with *early termination* and *combine*, while *sort* for any Elf type adds a large penalty. The *res* variants can only really compete in Q19, which has a lot of query strings and thus expensive query preparation.

5.2.3 Small String Optimization

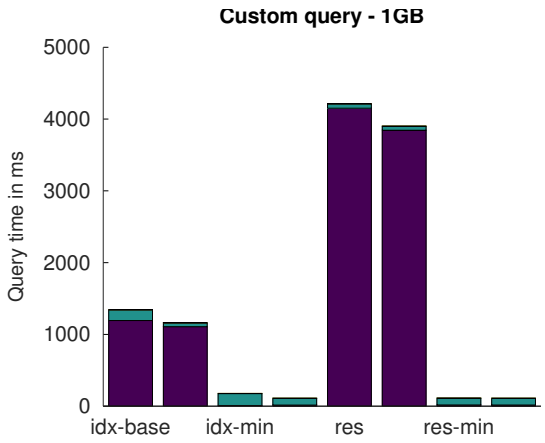


Figure 5.4: Comparison between base variant and SSO for custom query

As Figure 5.4 shows, replacing small fixed-size strings with integers does improve query performance for our artificial query. The effect is most pronounced for *res* Elfs. The reason for this is that the fully-built Elf does not have the string column as its first, which leads to more BAT look-ups since there are more dimension lists in lower dimensions.

The *idx* variant also profits from *sso*. Instead of speeding up traversal, however, the prelude of finding the string's index is omitted. This leads to a noticeable improvements, even for the minimal Elf, whereas there is no visible improvement for resolve-based minimal Elf due to the overall low query time.

5.2.4 Size scaling

We conducted our experiments with data limited in size up to 10GB. However, to draw conclusions about possible performance for larger datasets, Figure 5.5 shows how well the different Elf variants scale with size compared to MonetDB. The *normalized time ratio* computes from the ratio of query times between two dataset sizes normalized by the ratio of the sizes, with a ratio below one indicating a less-than-linear increase in query time. In our case the normalization factor is 10. We split the results in two parts: the left side of each sub-plot shows the ratios for the entire query duration, whereas the right side only takes the time spend on the selection into account.

The optimization chosen for Elf was *concat* combined with *early termination*. Overall, how well any approach scales depends on the query in question. While the

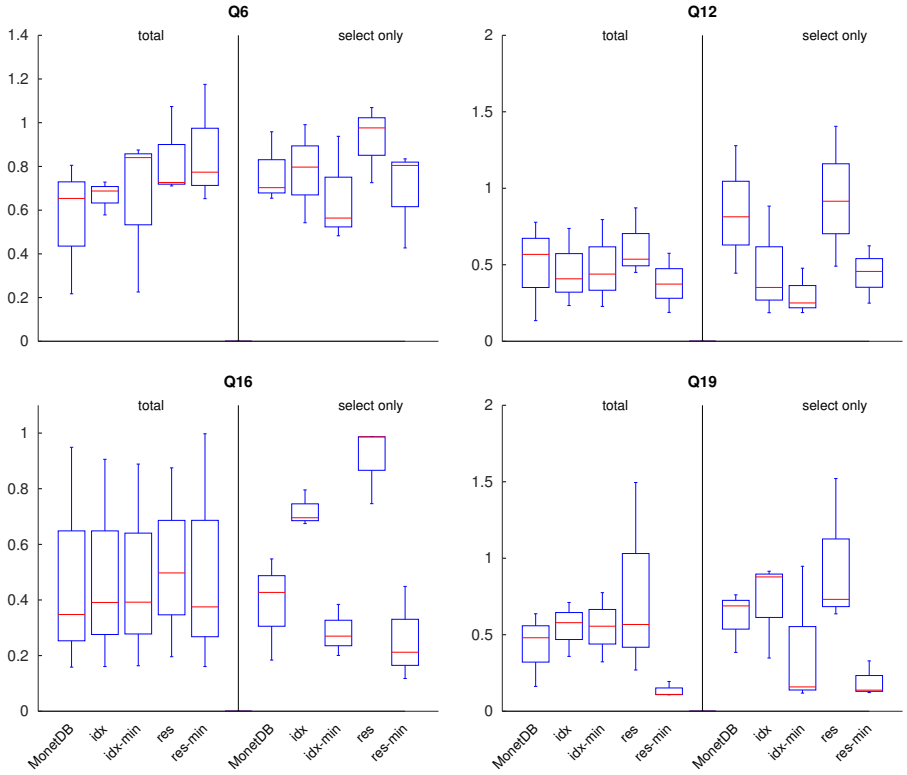


Figure 5.5: Scaling of MonetDB and Elf approaches when increasing size

minimal Elf generally scales better, the overhead of query creation balances the scaling when looking at the entire query duration. The selection part for minimal Elfs always scales better than MonetDB, regardless of query and Elf type. There does not seem to be a difference between resolve- and index-based minimal Elf, while the full resolve-based Elf is generally worse. This is likely due to the extra columns being worse for the latter, since they pollute the cache and may push out BAT values.

5.3 Discussion

Looking at Figure 5.6, whether Elf is the better choice over scanning BATs depends on the concrete type of Elf as wells as the query in question. The *res* Elf does best for Q19, where *idx* Elf incurs a large overhead to find indexes for the query strings. The time spend on the actual select is usually lower for Elf; unfortunately, in most cases the necessary inter-op to MonetDB costs more than the gain from the fast traversal. Exceptions to this are Q22, for which Elf can only query a single column, as well as Q12 when combining *res* Elf and low scale factor. The latter is a result of the additional BAT lookup, which are not cached at the beginning of the traversal and thus incur a larger initial penalty.

A big strength of Elf lies in its ability to combine multiple query predicates into one traversal, which requires support from the DBMS. Taking that away makes Elf a non-favourable choice, regardless of how it handles strings. Similar effects, but to a lesser extend, can be observed when omitting its early termination. Both of

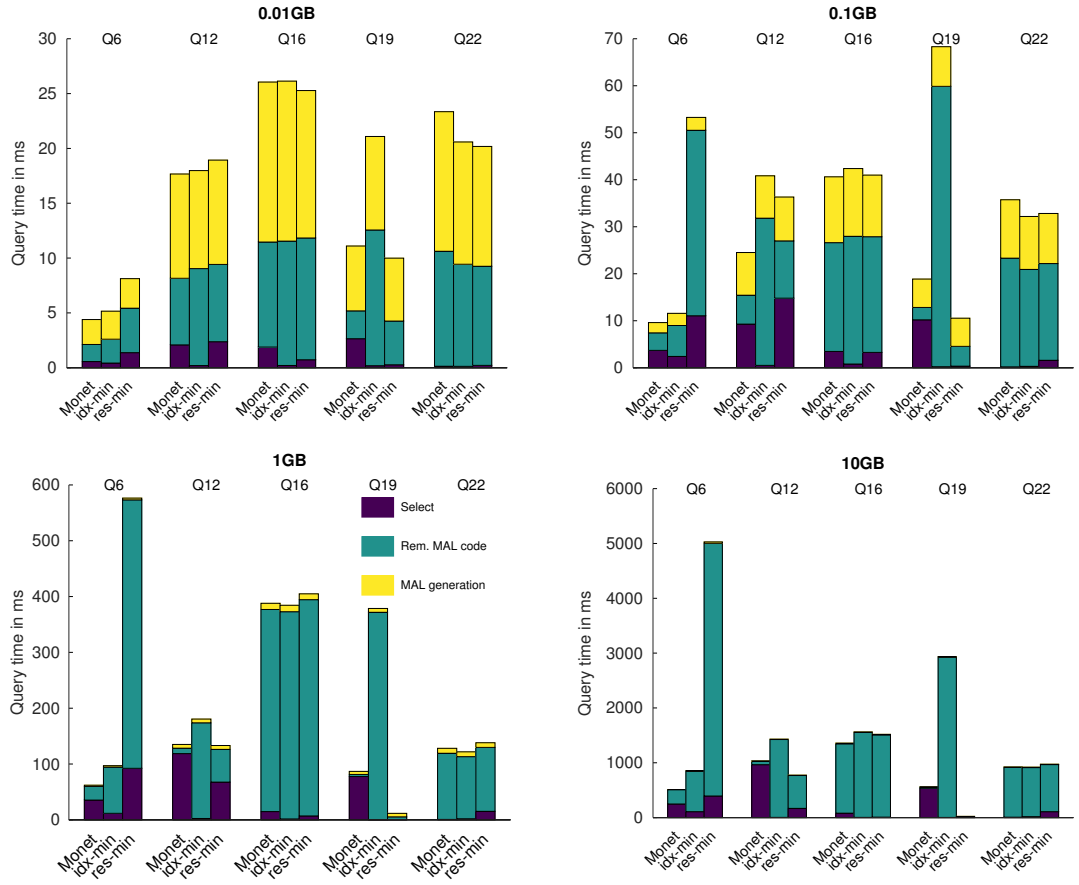


Figure 5.6: Query times for MonetDB and both minimal Elf variants with *combine* and *early termination* (in order: MonetDB, index-based, resolve-based)

these techniques reduce the number of memory accesses, which is what gives it an advantage over plain memory scans.

Generally, both MonetDB and Elf scale better-than-linear. While this is expected of Elf due to its prefix redundancy elimination, a memory scan should not accelerate given a sufficiently large dataset. However, MonetDB does not just invoke such a scan. Instead, it performs a bunch of checks to figure out what algorithm to choose and how large the result set may be. As such, there is some static overhead skewing the raw select duration. In combination with the hash lookup performed for strings, which naturally does not scale linearly, the less-than-linear growth is explained.

This does not hold for Q6, however, since it does not use strings and uses column imprints to accelerate the scans. Unfortunately, we do not have an explanation for this behaviour aside from CPU cache phenomenons: the biggest improvement happens when increasing the scale factor from 1 to 10, at which point the columns no longer fit into the L2 cache.

5.4 Threats to validity

To draw valid conclusions from the evaluation, we considered a number of possible caveats beforehand. In this section we discuss both *internal* and *external* threats to validity as well as our measures to limit their influence.

5.4.1 Internal validity

In the design of our experiments we addressed the following threats: sampling variance, cache reusage, and measurement inaccuracy induced by MonetDB's profiler.

- *Sampling variance*: Since our experiments cannot run in complete isolation, there is bound to be some variance in the query time. This could be caused by a number of issues: scheduling operations by the OS, sleep states of the CPU, or state of the branch predictor at the start of a query are mostly out of our control. To mitigate these we run each query ten times and averaged the run times after discarding the three furthest outliers. Additionally, we pin the process responsible for query execution to one core, minimizing the influence of the task scheduler. Addressing the branch predictor is more difficult; aside from averaging we do not run the same query twice in a row without some other operation in-between, which is another query being executed or instructing MonetDB to create an Elf.

- *Cache reusage*: One of the biggest influences for index structures and memory scans is the CPU's cache state. While not running the same query in succession helps mitigating the issue, it cannot guarantee that some parts of the previous data remain. Thus we attempt to reset the cache into a well-known state before each query execution by loading a 50MB array, which exceeds the L3 cache size of 20MB. To also flush the L1 and L2 data caches, we execute this on the same core as the query execution. Since we do not directly invalidate cache lines, it might happen that some query data remains cached if the CPU uses an unusual caching strategy, i.e. it does not evict cache lines in favor of sequentially read data.

- *Measurement inaccuracy*: Considering that we need to measure the query execution time, we introduce a systemic bias by using MonetDB's *trace* facilities, slightly increasing query duration for every query. Differences should happen due to longer MAL code and thus more measuring, but only be on the scale of microseconds and thus the same scale as *trace*'s accuracy. To avoid measuring the profiler itself, we do not count the completion time of MAL statements related to trace itself or the query log. However, the total query time still includes these, adding a slight bias to it.

5.4.2 External validity

The main threat to generalizing our evaluation is the lack of realism. Since the Elf is not applicable to a large range of query types, we had to limit ourselves to a narrow selection of queries. To avoid over-specializing, we chose queries which elicit different parts of our implementation. Choosing different scale factors helps determining the system's capacity to deal with different dataset sizes. This does not only affect the actual select, but also the interop for result set creation or further processing, as the result size grows with over dataset size.

6. Related Work

In this chapter, we list some research related to our own work. Due to MonetDB's modularity and research character, there are a number of extensions building on it. We first discuss some alternative acceleration approaches and then give an overview of existing extensions.

Research on improving multi-dimensional queries as used in OLAP/OLTP may be divided in two categories: indexing and accelerating main-memory scans. The former splits again into *data-* and *space-partitioning* approaches, including tree-based techniques like *kd* and *r-trees* (see Section 2.3). An alternative to the *column imprints* used by MonetDB for the acceleration of main-memory scans is *BitWeaving*. For columns which use less bits than a full processor word offers, it packs multiple values into one word, evaluating a query predicate on all of them in parallel [LP13]. Depending on the how the packing occurs, BitWeaving differentiates between horizontal and vertical bit-parallelism. Like column imprints, it can only speed up queries for a single column at a time.

The separation of front- and back-end as well as the flexibility of *MAL* allow for customizations of *MonetDB* without needing to change the entire system. The large datasets of GIS applications brought about the need for efficient bulk loading and filtering of point cloud data, leading to the implementation of the *geom*-submodule of MonetDB [BQK96]. Importantly, it makes use of column imprints (see Section 2.3) for the filtering. To allow processing of genome sequence data, the *BAM/SAM*-module extends MonetDB with SQL functionality to load and export sequence data [CMK+15]. To work with the data, it also introduces functions operating on sequence alignment data to the front-end and automatic construction of read pairs.

To enable XQuery as a query language, *MonetDB/XQuery* adds a parser front-end. However, since XML does not directly translate to relational schemata, it implements a relational encoding alongside mappings to relational algebra and algorithms for traversal and document updates. These changes are accompanied by MAL generation as well as an optimization pipeline to reorder the query plan and are aggregated in the *Pathfinder* compiler module. The corresponding runtime module adds *loop-lifting staircase join* to evaluate nested iteration scopes in XQuery.

MonetDB/DataCell is an extension retrofitting MonetDB with stream functionalities [LIMK12]. It interjects itself between the kernel and the SQL front-end, extending the parser with language constructs to recognize continuous queries and imposing a query rewrite on the generated SQL plan, storing it in possibly multiple *factories*. The query execution is handled by a scheduler, interacting with lightweight tables called *baskets*, which aggregate stream tuples for processing. These tuples enter the system via *receptors* and exit via *emitters* once the system finishes query execution on the aggregated data.

To further optimize cache usage on modern CPUs, *MonetDB/X100* implements a new query engine [BZN05]. It exchanges the compiled MAL statements in favor of vectorized data directly in the cache, processing multiple columns with relational operators [ZBNH05]. This helps both utilizing the CPU cache as well as pipelining. From this, *Vectorwise* developed, merging the execution engine of *MonetDB/X100* with the *Ingres RDBMS* [ZvdWB12].

Since GPUs have surpassed CPUs in raw computational power, *Ocelot* is a proof-of-concept extension for MonetDB, enabling it to take advantage of different hardware via hardware-oblivious parallelism [HSP+13]. It employs OpenCL to abstract away the actual hardware, compiling the operators at runtime. The device memory is seen as a cache, holding BATs and evicting them in favor of recently accessed ones akin to the CPU's own cache. A problem building from device-agnostic operators is the choice of the correct device to run a given operation on. The optimizer *HyPE* approaches this with a set of heuristics including device load and the estimated runtime of each operator [BBR+13]. Both systems were combined to form a system taking advantage of heterogeneous hardware [BKH+14].

7. Conclusion

To finalize this thesis, we summarize our findings and propose ideas for future work in this chapter.

In this work, we successfully integrated the main-memory index structure *Elf* into the column-store *MonetDB*. For this, we extended MonetDB in three areas: its parser, query execution engine, and data loading facilities. Extending the parser allows us to choose between different implementations. Via its internal language *MAL*, we provide facilities to create, populate, and execute partial-match, *in*, and column-column queries alongside MonetDB's regular query engine.

Furthermore, we introduced two variants to allow string-typed columns in Elf. They differ in where they place the additional overhead: we either sort strings during Elf creation and use their ordered indices as a replacement, or store their unordered indices and look them up during traversal, striking a trade-off between faster traversal and faster query preparation.

We then showed several improvements of our implementation. These aim for the interoperability with MonetDB as well as how to properly utilize Elf by aggregating multiple queries before execution. To improve the handling of strings, we proposed a way to quickly store small strings directly in Elf. Additionally, we suggested a way to dynamically switch between complex termination criteria and fully traversing small dimension lists as a trade-off between cache- and branching-friendly traversal.

Our evaluation on the TPC-H benchmark showed that the improvements to interoperability have a large impact on the query overhead. While the Elf with aggregated queries is mostly faster for the scale factors 1 and 10, the additional cost of string processing is significant compared to MonetDB's straight-forward query preparation. This is also reflected in the choice of string handling: many strings in a query largely favour resolving strings during traversal, albeit the increased actual query time. Determining a cut-off column for early termination proved to be largely insignificant for the traversal performance. Another significant factor for the query time is what columns the Elf indexes, which should be the minimal set necessary for the queries.

In summary, we enabled the optional use of Elf in a real-world DBMS and showed that it can improve the performance, although it can have a significant overhead. We showed solutions for storing strings and the significance of column selection for indexing.

7.1 Future work

In this section we present ideas for continued research. These revolve largely around improving the interoperability between Elf and MonetDB.

Elf in its current state is fairly limited in its possible operations: it only supports partial-match, in, and column-column queries, neither joins nor update operations of any kind are currently possible. To enable the use of Elf in scenarios with a largely static database, but small additions over time, we suggest research into extending Elf. One possible avenue for this could be the aggregation of tuples either in regular BATs or a separate, not linearized Elf. While this set of tuples remains small, the additional time spend on querying them is negligible. Once it grows to be noticeable, the DBMS may automatically trigger a rebuild or merge between the two sets.

Throughout this work we limited MonetDB and thus Elf to a single thread, which is fairly limiting for modern multi-core CPUs. A logical consequence would be to split the traversal of Elf into multiple threads, each aggregating TIDs from their respective sub-trees. However, care would have to be taken when combining the results, as the overhead may not be worth the effort. Similarly, Elf may be able to take advantage of SIMD. This would require proper alignment and thus an increase in size, which may also decrease performance since less dimension list entries would fit into a cache line.

Since our focus was on the selection performance of Elf, we neglected its creation time and memory footprint. For a real-world DBMS both matter, especially if the data is not totally static. To not lose the created Elf with every server shutdown, we propose to utilize MonetDB's storage facilities to make Elfs persistent alongside BATs, possibly utilizing memory-mapped files.

The downside of handling strings by ordering them is the large overhead upfront as well as additional memory consumption for the ordered indices. While the latter cannot be avoided entirely, it may be possible to speed up the index lookup. Currently, we need to iterate all indices if the comparator of the predicate is not *equals*. However, it may prove to be beneficial to use an acceleration structure for the lookup, provided it supports finding the next smaller and larger value. Alternatively, caching the lookup or looking up multiple strings per iteration may also reduce the overhead.

8. Appendix

```
1  function user.s6_1():void;
2  X_12:int := sql.mvc();
3  X_16:bat[:lng] := sql.bind(X_12:int, "sys":str, "lineitem":
       str, "l_quantity":str, 0:int);
4  X_33:bat[:lng] := sql.bind(X_12:int, "sys":str, "lineitem":
       str, "l_discount":str, 0:int);
5  X_40:bat[:date] := sql.bind(X_12:int, "sys":str, "lineitem":
       str, "l_shipdate":str, 0:int);
6  C_13:bat[:oid] := sql.tid(X_12:int, "sys":str, "lineitem":
       str);
7  // Column imprints
8  C_51:bat[:oid] := algebra.select(X_40:bat[:date], C_13:bat[:
       oid], "1994-01-01":date, "1995-01-01":date, true:bit,
       false:bit, false:bit);
9  // Column imprints with candidates
10 C_68:bat[:oid] := algebra.select(X_33:bat[:lng], C_51:bat[:
       oid], 5:lng, 7:lng, true:bit, true:bit, false:bit);
11 // Column imprints with candidates
12 C_71:bat[:oid] := algebra.thetaselect(X_16:bat[:lng], C_68:
       bat[:oid], 2400:lng, "<":str);
13 X_26:bat[:lng] := sql.bind(X_12:int, "sys":str, "lineitem":
       str, "l_extendedprice":str, 0:int);
14 X_74:bat[:lng] := algebra.projection(C_71:bat[:oid], X_26:
       bat[:lng]);
15 X_75:bat[:lng] := algebra.projection(C_71:bat[:oid], X_33:
       bat[:lng]);
16 X_81:bat[:hge] := batcalc.*(X_74:bat[:lng], X_75:bat[:lng]);
17 X_83:hge := aggr.sum(X_81:bat[:hge]);
18 end user.s2_1;
```

Listing 8.1: MAL code of Q6 by MonetDB

```
1  function user.s12_1():void;
2  X_17:int := sql.mvc();
3  X_45:bat[:str] := sql.bind(X_17:int, "sys":str, "lineitem":
       str, "l_shipmode":str, 0:int);
4  X_38:bat[:date] := sql.bind(X_17:int, "sys":str, "lineitem":
       str, "l_receiptdate":str, 0:int);
5  X_31:bat[:date] := sql.bind(X_17:int, "sys":str, "lineitem":
       str, "l_commitdate":str, 0:int);
6  X_21:bat[:date] := sql.bind(X_17:int, "sys":str, "lineitem":
       str, "l_shipdate":str, 0:int);
7  X_66:bat[:bit] := batcalc.>(X_31:bat[:date], X_21:bat[:date
       ]);
8  X_59:bat[:bit] := batcalc.<(X_31:bat[:date], X_38:bat[:date
       ]);
9  C_18:bat[:oid] := sql.tid(X_17:int, "sys":str, "lineitem":
       str);
10 // Full scan
11 C_62:bat[:oid] := algebra.select(X_59:bat[:bit], C_18:bat[:
       oid], true:bit, true:bit, true:bit, true:bit, false:bit);
12 // Candidate scan
13 C_68:bat[:oid] := algebra.select(X_66:bat[:bit], C_62:bat[:
       oid], true:bit, true:bit, true:bit, true:bit, false:bit);
14 // Column imprints with candidates
15 C_73:bat[:oid] := algebra.select(X_38:bat[:date], C_68:bat[:
       oid], "1994-01-01":date, "1995-01-01":date, true:bit,
       false:bit, false:bit);
16 // Full scan with hash lookup
17 C_75:bat[:oid] := algebra.thetaselect(X_45:bat[:str], C_73:
       bat[:oid], "MAIL":str, "==":str);
18 // Full scan with hash lookup
19 C_78:bat[:oid] := algebra.thetaselect(X_45:bat[:str], C_73:
       bat[:oid], "SHIP":str, "==":str);
20 X_79:bat[:oid] := bat.mergecand(C_75:bat[:oid], C_78:bat[:
       oid]);
21 X_52:bat[:oid] := sql.bind_idxbat(X_17:int, "sys":str, "
       lineitem":str, "lineitem_l_orderkey_fkey":str, 0:int);
22 (X_55:bat[:oid], X_56:bat[:oid]) := sql.bind_idxbat(X_17:int
       , "sys":str, "lineitem":str, "lineitem_l_orderkey_fkey":
       str, 2:int);
23 X_54:bat[:oid] := sql.bind_idxbat(X_17:int, "sys":str, "
       lineitem":str, "lineitem_l_orderkey_fkey":str, 1:int);
24 X_84:bat[:oid] := sql.projectdelta(X_79:bat[:oid], X_52:bat
       [:oid], X_55:bat[:oid], X_56:bat[:oid], X_54:bat[:oid]);
25 C_85:bat[:oid] := sql.tid(X_17:int, "sys":str, "orders":str)
       ;
```

Listing 8.2: MAL code of Q12 by MonetDB

```
(X_94:bat[:oid], X_95:bat[:oid]) := algebra.join(X_84:bat[:oid], C_85:bat[:oid], nil:BAT, nil:BAT, false:bit, nil:lng);
X_103:bat[:str] := algebra.projectionpath(X_94:bat[:oid], X_79:bat[:oid], X_45:bat[:str]);
(X_137:bat[:oid], C_138:bat[:oid], X_139:bat[:lng]) := group.groupdone(X_103:bat[:str]);
X_140:bat[:str] := algebra.projection(C_138:bat[:oid], X_103:bat[:str]);
X_87:bat[:str] := sql.bind(X_17:int, "sys":str, "orders":str, "o_orderpriority":str, 0:int);
X_105:bat[:str] := algebra.projectionpath(X_95:bat[:oid], C_85:bat[:oid], X_87:bat[:str]);
X_108:bat[:bit] := batcalc.==(X_105:bat[:str], "1-URGENT":str);
X_112:bat[:bit] := batcalc.==(X_105:bat[:str], "2-HIGH":str);
X_113:bat[:bit] := batcalc.or(X_108:bat[:bit], X_112:bat[:bit]);
X_115:bat[:bit] := batcalc.isnil(X_113:bat[:bit]);
X_118:bat[:bit] := batcalc.ifthenelse(X_115:bat[:bit], false:bit, X_113:bat[:bit]);
X_122:bat[:bte] := batcalc.ifthenelse(X_118:bat[:bit], 1:bte, 0:bte);
X_141:bat[:hge] := aggr.subsum(X_122:bat[:bte], X_137:bat[:oid], C_138:bat[:oid], true:bit, true:bit);
X_124:bat[:bit] := batcalc.!=(X_105:bat[:str], "1-URGENT":str);
X_127:bat[:bit] := batcalc.!=(X_105:bat[:str], "2-HIGH":str);
X_128:bat[:bit] := batcalc.and(X_124:bat[:bit], X_127:bat[:bit]);
X_130:bat[:bit] := batcalc.isnil(X_128:bat[:bit]);
X_133:bat[:bit] := batcalc.ifthenelse(X_130:bat[:bit], false:bit, X_128:bat[:bit]);
X_136:bat[:bte] := batcalc.ifthenelse(X_133:bat[:bit], 1:bte, 0:bte);
X_144:bat[:hge] := aggr.subsum(X_136:bat[:bte], X_137:bat[:oid], C_138:bat[:oid], true:bit, true:bit);
(X_145:bat[:str], X_146:bat[:oid], X_147:bat[:oid]) := algebra.sort(X_140:bat[:str], false:bit, false:bit);
X_150:bat[:hge] := algebra.projection(X_146:bat[:oid], X_144:bat[:hge]);
X_149:bat[:hge] := algebra.projection(X_146:bat[:oid], X_141:bat[:hge]);
X_148:bat[:str] := algebra.projection(X_146:bat[:oid], X_140:bat[:str]);
end user.s2_1;
```

Listing 8.3: MAL code of Q12 by MonetDB (cont.)

```
1  function user.s16_1():void;
2  X_15:int := sql.mvc();
3  C_16:bat[:oid] := sql.tid(X_15:int, "sys":str, "partsupp":
       str);
4  X_19:bat[:int] := sql.bind(X_15:int, "sys":str, "partsupp":
       str, "ps_suppkey":str, 0:int);
5  X_28:bat[:int] := algebra.projection(C_16:bat[:oid], X_19:
       bat[:int]);
6  X_63:bat[:oid] := bat.mirror(X_28:bat[:int]);
7  C_36:bat[:oid] := sql.tid(X_15:int, "sys":str, "supplier":
       str);
8  X_45:bat[:str] := sql.bind(X_15:int, "sys":str, "supplier":
       str, "s_comment":str, 0:int);
9  X_51:bat[:str] := algebra.projection(C_36:bat[:oid], X_45:
       bat[:str]);
10 C_55:bat[:oid] := algebra.likeselect(X_51:bat[:str], nil:BAT
       , "%Customer%Complaints%":str, "":str, false:bit);
11 X_38:bat[:int] := sql.bind(X_15:int, "sys":str, "supplier":
       str, "s_suppkey":str, 0:int);
12 X_58:bat[:int] := algebra.projectionpath(C_55:bat[:oid],
       C_36:bat[:oid], X_38:bat[:int]);
13 (X_60:bat[:oid], X_61:bat[:oid]) := algebra.join(X_28:bat[:
       int], X_58:bat[:int], nil:BAT, nil:BAT, false:bit, nil:
       lng);
14 X_64:bat[:oid] := algebra.difference(X_63:bat[:oid], X_60:
       bat[:oid], nil:BAT, nil:BAT, false:bit, nil:lng);
15 X_29:bat[:oid] := sql.bind_idxbat(X_15:int, "sys":str, "
       partsupp":str, "partsupp_ps_partkey_fkey":str, 0:int);
16 (X_32:bat[:oid], X_33:bat[:oid]) := sql.bind_idxbat(X_15:int
       , "sys":str, "partsupp":str, "partsupp_ps_partkey_fkey":
       str, 2:int);
17 X_31:bat[:oid] := sql.bind_idxbat(X_15:int, "sys":str, "
       partsupp":str, "partsupp_ps_partkey_fkey":str, 1:int);
18 X_34:bat[:oid] := sql.delta(X_29:bat[:oid], X_32:bat[:oid],
       X_33:bat[:oid], X_31:bat[:oid]);
19 X_66:bat[:oid] := algebra.projectionpath(X_64:bat[:oid],
       C_16:bat[:oid], X_34:bat[:oid]);
20 C_67:bat[:oid] := sql.tid(X_15:int, "sys":str, "part":str);
21 X_83:bat[:int] := sql.bind(X_15:int, "sys":str, "part":str,
       "p_size":str, 0:int);
22 X_89:bat[:int] := algebra.projection(C_67:bat[:oid], X_83:
       bat[:int]);
23 X_76:bat[:str] := sql.bind(X_15:int, "sys":str, "part":str,
       "p_type":str, 0:int);
24 X_82:bat[:str] := algebra.projection(C_67:bat[:oid], X_76:
       bat[:str]);
25 X_69:bat[:str] := sql.bind(X_15:int, "sys":str, "part":str,
       "p_brand":str, 0:int);
```

Listing 8.4: MAL code of Q16 by MonetDB

```
X_75:bat[:str] := algebra.projection(C_67:bat[:oid], X_69:
    bat[:str]);
// Full scan
C_91:bat[:oid] := algebra.thetaselect(X_75:bat[:str], "Brand
    #45":str, "!=":str);
C_96:bat[:oid] := algebra.likeselect(X_82:bat[:str], C_91:
    bat[:oid], "MEDIUM POLISHED%":str, "":str, true:bit);
// Candidate scan
C_99:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 49:int, "==":str);
// Candidate scan
C_102:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 14:int, "==":str);
X_103:bat[:oid] := bat.mergecand(C_99:bat[:oid], C_102:bat[:
    oid]);
// Candidate scan
C_105:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 23:int, "==":str);
X_106:bat[:oid] := bat.mergecand(X_103:bat[:oid], C_105:bat
    [:oid]);
// Candidate scan
C_108:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 45:int, "==":str);
X_109:bat[:oid] := bat.mergecand(X_106:bat[:oid], C_108:bat
    [:oid]);
// Candidate scan
C_111:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 19:int, "==":str);
X_112:bat[:oid] := bat.mergecand(X_109:bat[:oid], C_111:bat
    [:oid]);
// Candidate scan
C_114:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 3:int, "==":str);
X_115:bat[:oid] := bat.mergecand(X_112:bat[:oid], C_114:bat
    [:oid]);
// Candidate scan
C_117:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 36:int, "==":str);
X_118:bat[:oid] := bat.mergecand(X_115:bat[:oid], C_117:bat
    [:oid]);
// Candidate scan
C_120:bat[:oid] := algebra.thetaselect(X_89:bat[:int], C_96:
    bat[:oid], 9:int, "==":str);
X_121:bat[:oid] := bat.mergecand(X_118:bat[:oid], C_120:bat
    [:oid]);
X_125:bat[:oid] := algebra.projection(X_121:bat[:oid], C_67:
    bat[:oid]);
```

Listing 8.5: MAL code of Q16 by MonetDB (cont. 1)

```
1  (X_126:bat[:oid], X_127:bat[:oid]) := algebra.join(X_66:bat
       [:oid], X_125:bat[:oid], nil:BAT, nil:BAT, false:bit, nil
       :lng);
2  X_135:bat[:str] := algebra.projectionpath(X_127:bat[:oid],
       X_121:bat[:oid], X_82:bat[:str]);
3  X_136:bat[:int] := algebra.projectionpath(X_127:bat[:oid],
       X_121:bat[:oid], X_89:bat[:int]);
4  X_132:bat[:int] := algebra.projectionpath(X_126:bat[:oid],
       X_64:bat[:oid], X_28:bat[:int]);
5  X_134:bat[:str] := algebra.projectionpath(X_127:bat[:oid],
       X_121:bat[:oid], X_75:bat[:str]);
6  (X_138:bat[:oid], C_139:bat[:oid], X_140:bat[:lng]) := group
       .group(X_134:bat[:str]);
7  (X_141:bat[:oid], C_142:bat[:oid], X_143:bat[:lng]) := group
       .subgroup(X_132:bat[:int], X_138:bat[:oid]);
8  (X_144:bat[:oid], C_145:bat[:oid], X_146:bat[:lng]) := group
       .subgroup(X_136:bat[:int], X_141:bat[:oid]);
9  (X_147:bat[:oid], C_148:bat[:oid], X_149:bat[:lng]) := group
       .subgroupdone(X_135:bat[:str], X_144:bat[:oid]);
10 X_150:bat[:str] := algebra.projection(C_148:bat[:oid], X_134
       :bat[:str]);
11 X_151:bat[:str] := algebra.projection(C_148:bat[:oid], X_135
       :bat[:str]);
12 X_152:bat[:int] := algebra.projection(C_148:bat[:oid], X_136
       :bat[:int]);
13 X_153:bat[:int] := algebra.projection(C_148:bat[:oid], X_132
       :bat[:int]);
14 (X_154:bat[:oid], C_155:bat[:oid], X_156:bat[:lng]) := group
       .group(X_150:bat[:str]);
15 (X_157:bat[:oid], C_158:bat[:oid], X_159:bat[:lng]) := group
       .subgroup(X_152:bat[:int], X_154:bat[:oid]);
16 (X_160:bat[:oid], C_161:bat[:oid], X_162:bat[:lng]) := group
       .subgroupdone(X_151:bat[:str], X_157:bat[:oid]);
17 X_163:bat[:str] := algebra.projection(C_161:bat[:oid], X_150
       :bat[:str]);
18 X_164:bat[:str] := algebra.projection(C_161:bat[:oid], X_151
       :bat[:str]);
19 X_165:bat[:int] := algebra.projection(C_161:bat[:oid], X_152
       :bat[:int]);
20 X_166:bat[:lng] := aggr.subcount(X_153:bat[:int], X_160:bat
       [:oid], C_161:bat[:oid], true:bit);
21 (X_168:bat[:lng], X_169:bat[:oid], X_170:bat[:oid]) :=
       algebra.sort(X_166:bat[:lng], true:bit, false:bit);
22 (X_172:bat[:str], X_173:bat[:oid], X_174:bat[:oid]) :=
       algebra.sort(X_163:bat[:str], X_169:bat[:oid], X_170:bat
       [:oid], false:bit, false:bit);
```

Listing 8.6: MAL code of Q16 by MonetDB (cont. 2)

```
1 (X_175:bat[:str], X_176:bat[:oid], X_177:bat[:oid]) :=
    algebra.sort(X_164:bat[:str], X_173:bat[:oid], X_174:bat
    [:oid], false:bit, false:bit);
2 (X_178:bat[:int], X_179:bat[:oid], X_180:bat[:oid]) :=
    algebra.sort(X_165:bat[:int], X_176:bat[:oid], X_177:bat
    [:oid], false:bit, false:bit);
3 X_184:bat[:lng] := algebra.projection(X_179:bat[:oid], X_166
    :bat[:lng]);
4 X_183:bat[:int] := algebra.projection(X_179:bat[:oid], X_165
    :bat[:int]);
5 X_182:bat[:str] := algebra.projection(X_179:bat[:oid], X_164
    :bat[:str]);
6 X_181:bat[:str] := algebra.projection(X_179:bat[:oid], X_163
    :bat[:str]);
7 end user.s2_1;
```

Listing 8.7: MAL code of Q16 by MonetDB (cont. 3)

```
function user.s19_1():void;
X_18:int := sql.mvc();
X_46:bat[:str] := sql.bind(X_18:int, "sys":str, "part":str,
    "p_container":str, 0:int);
X_39:bat[:int] := sql.bind(X_18:int, "sys":str, "part":str,
    "p_size":str, 0:int);
X_32:bat[:str] := sql.bind(X_18:int, "sys":str, "part":str,
    "p_brand":str, 0:int);
C_19:bat[:oid] := sql.tid(X_18:int, "sys":str, "part":str);
// Full scan with hash lookup
C_54:bat[:oid] := algebra.thetaselect(X_32:bat[:str], C_19:
    bat[:oid], "Brand#12":str, "==":str);
// Column imprints with candidates
C_58:bat[:oid] := algebra.select(X_39:bat[:int], C_54:bat[:
    oid], 1:int, 5:int, true:bit, true:bit, false:bit);
// Candidate scan
C_63:bat[:oid] := algebra.thetaselect(X_46:bat[:str], C_58:
    bat[:oid], "SM CASE":str, "==":str);
// Candidate scan
C_65:bat[:oid] := algebra.thetaselect(X_46:bat[:str], C_58:
    bat[:oid], "SM BOX":str, "==":str);
X_66:bat[:oid] := bat.mergecand(C_63:bat[:oid], C_65:bat[:
    oid]);
// Candidate scan
C_68:bat[:oid] := algebra.thetaselect(X_46:bat[:str], C_58:
    bat[:oid], "SM PACK":str, "==":str);
X_69:bat[:oid] := bat.mergecand(X_66:bat[:oid], C_68:bat[:
    oid]);
// Candidate scan
C_71:bat[:oid] := algebra.thetaselect(X_46:bat[:str], C_58:
    bat[:oid], "SM PKG":str, "==":str);
X_72:bat[:oid] := bat.mergecand(X_69:bat[:oid], C_71:bat[:
    oid]);
X_22:bat[:int] := sql.bind(X_18:int, "sys":str, "part":str,
    "p_partkey":str, 0:int);
X_73:bat[:int] := algebra.projection(X_72:bat[:oid], X_22:
    bat[:int]);
X_114:bat[:str] := sql.bind(X_18:int, "sys":str, "lineitem":
    str, "l_shipmode":str, 0:int);
X_86:bat[:lng] := sql.bind(X_18:int, "sys":str, "lineitem":
    str, "l_quantity":str, 0:int);
X_107:bat[:str] := sql.bind(X_18:int, "sys":str, "lineitem":
    str, "l_shipinstruct":str, 0:int);
C_77:bat[:oid] := sql.tid(X_18:int, "sys":str, "lineitem":
    str);
```

Listing 8.8: MAL code of Q19 by MonetDB

```
1  // Candidate scan with hash lookup
2  C_122:bat[:oid] := algebra.thetaselect(X_107:bat[:str], C_77
       :bat[:oid], "TAKE BACK RETURN":str, "==":str);
3  // Column imprints with candidates
4  C_132:bat[:oid] := algebra.select(X_86:bat[:lng], C_122:bat
       [:oid], 100:lng, 1100:lng, true:bit, true:bit, false:bit)
       ;
5  // Candidate scan with hash lookup
6  C_137:bat[:oid] := algebra.thetaselect(X_114:bat[:str],
       C_132:bat[:oid], "AIR":str, "==":str);
7  // Candidate scan with hash lookup
8  C_139:bat[:oid] := algebra.thetaselect(X_114:bat[:str],
       C_132:bat[:oid], "AIR REG":str, "==":str);
9  X_140:bat[:oid] := bat.mergecand(C_137:bat[:oid], C_139:bat
       [:oid]);
10 X_79:bat[:int] := sql.bind(X_18:int, "sys":str, "lineitem":
       str, "l_partkey":str, 0:int);
11 X_141:bat[:int] := algebra.projection(X_140:bat[:oid], X_79:
       bat[:int]);
12 (X_147:bat[:oid], X_148:bat[:oid]) := algebra.join(X_73:bat
       [:int], X_141:bat[:int], nil:BAT, nil:BAT, false:bit, nil
       :lng);
13 X_93:bat[:lng] := sql.bind(X_18:int, "sys":str, "lineitem":
       str, "l_extendedprice":str, 0:int);
14 X_158:bat[:lng] := algebra.projectionpath(X_148:bat[:oid],
       X_140:bat[:oid], X_93:bat[:lng]);
15 X_100:bat[:lng] := sql.bind(X_18:int, "sys":str, "lineitem":
       str, "l_discount":str, 0:int);
16 X_159:bat[:lng] := algebra.projectionpath(X_148:bat[:oid],
       X_140:bat[:oid], X_100:bat[:lng]);
17 X_170:bat[:lng] := batcalc.-(100:lng, X_159:bat[:lng]);
18 X_172:bat[:hge] := batcalc.*(X_158:bat[:lng], X_170:bat[:lng
       ]);
19 X_174:hge := aggr.sum(X_172:bat[:hge]);
20 end user.s2_1;
```

Listing 8.9: MAL code of Q19 by MonetDB (cont.)

```
1  function user.s22_1():void;
2  X_25:int := sql.mvc();
3  C_26:bat[:oid] := sql.tid(X_25:int, "sys":str, "customer":
       str);
4  X_39:bat[:str] := sql.bind(X_25:int, "sys":str, "customer":
       str, "c_phone":str, 0:int);
5  X_45:bat[:str] := algebra.projection(C_26:bat[:oid], X_39:
       bat[:str]);
6  X_55:bat[:str] := batstr.substring(X_45:bat[:str], 1:int, 2:
       int);
7  // Full scan with hash lookup
8  C_59:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "13":
       str, "==":str);
9  // Candidate scan with hash lookup
10 C_62:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "31":
       str, "==":str);
11 X_63:bat[:oid] := bat.mergecand(C_59:bat[:oid], C_62:bat[:
       oid]);
12 // Candidate scan with hash lookup
13 C_65:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "23":
       str, "==":str);
14 X_66:bat[:oid] := bat.mergecand(X_63:bat[:oid], C_65:bat[:
       oid]);
15 // Candidate scan with hash lookup
16 C_68:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "29":
       str, "==":str);
17 X_69:bat[:oid] := bat.mergecand(X_66:bat[:oid], C_68:bat[:
       oid]);
18 // Candidate scan with hash lookup
19 C_71:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "30":
       str, "==":str);
20 X_72:bat[:oid] := bat.mergecand(X_69:bat[:oid], C_71:bat[:
       oid]);
21 // Candidate scan with hash lookup
22 C_74:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "18":
       str, "==":str);
23 X_75:bat[:oid] := bat.mergecand(X_72:bat[:oid], C_74:bat[:
       oid]);
24 // Candidate scan with hash lookup
25 C_77:bat[:oid] := algebra.thetaselect(X_55:bat[:str], "17":
       str, "==":str);
26 X_78:bat[:oid] := bat.mergecand(X_75:bat[:oid], C_77:bat[:
       oid]);
27 X_29:bat[:int] := sql.bind(X_25:int, "sys":str, "customer":
       str, "c_custkey":str, 0:int);
28 X_79:bat[:int] := algebra.projectionpath(X_78:bat[:oid],
       C_26:bat[:oid], X_29:bat[:int]);
```

Listing 8.10: MAL code of Q22 by MonetDB

```
X_145:bat[:oid] := bat.mirror(X_79:bat[:int]);
X_46:bat[:lng] := sql.bind(X_25:int, "sys":str, "customer":str, "c_acctbal":str, 0:int);
X_52:bat[:lng] := algebra.projection(C_26:bat[:oid], X_46:bat[:lng]);
X_81:bat[:lng] := algebra.projection(X_78:bat[:oid], X_52:bat[:lng]);
X_137:bat[:dbl] := batcalc.dbl(2:int, X_81:bat[:lng]);
// Column imprints
C_101:bat[:oid] := algebra.thetaselect(X_52:bat[:lng], 0:lng, ">":str);
// Candidate scan with hash lookup
C_109:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "13":str, "==":str);
// Candidate scan with hash lookup
C_112:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "31":str, "==":str);
X_113:bat[:oid] := bat.mergecand(C_109:bat[:oid], C_112:bat[:oid]);
// Candidate scan with hash lookup
C_115:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "23":str, "==":str);
X_116:bat[:oid] := bat.mergecand(X_113:bat[:oid], C_115:bat[:oid]);
// Candidate scan with hash lookup
C_118:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "29":str, "==":str);
X_119:bat[:oid] := bat.mergecand(X_116:bat[:oid], C_118:bat[:oid]);
// Candidate scan with hash lookup
C_121:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "30":str, "==":str);
X_122:bat[:oid] := bat.mergecand(X_119:bat[:oid], C_121:bat[:oid]);
// Candidate scan with hash lookup
C_124:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "18":str, "==":str);
X_125:bat[:oid] := bat.mergecand(X_122:bat[:oid], C_124:bat[:oid]);
// Candidate scan with hash lookup
C_127:bat[:oid] := algebra.thetaselect(X_55:bat[:str], C_101:bat[:oid], "17":str, "==":str);
X_128:bat[:oid] := bat.mergecand(X_125:bat[:oid], C_127:bat[:oid]);
X_130:bat[:lng] := algebra.projection(X_128:bat[:oid], X_52:bat[:lng]);
```

Listing 8.11: MAL code of Q22 by MonetDB (cont. 1)

```
1  X_131:bat[:dbl] := batcalc.dbl(2:int, X_130:bat[:lng]);
2  X_135:dbl := aggr.avg(X_131:bat[:dbl]);
3  X_136:bat[:dbl] := sql.single(X_135:dbl);
4  (X_138:bat[:oid], X_139:bat[:oid]) := algebra.thetajoin(
       X_137:bat[:dbl], X_136:bat[:dbl], nil:BAT, nil:BAT, 1:int
       , true:bit, nil:lng);
5  C_146:bat[:oid] := algebra.intersect(X_145:bat[:oid], X_138:
       bat[:oid], nil:BAT, nil:BAT, false:bit, nil:lng);
6  X_148:bat[:int] := algebra.projection(C_146:bat[:oid], X_79:
       bat[:int]);
7  X_169:bat[:oid] := bat.mirror(X_148:bat[:int]);
8  C_151:bat[:oid] := sql.tid(X_25:int, "sys":str, "orders":str
       );
9  X_153:bat[:int] := sql.bind(X_25:int, "sys":str, "orders":
       str, "o_custkey":str, 0:int);
10 X_159:bat[:int] := algebra.projection(C_151:bat[:oid], X_153
       :bat[:int]);
11 (X_167:bat[:oid], X_168:bat[:oid]) := algebra.join(X_148:bat
       [:int], X_159:bat[:int], nil:BAT, nil:BAT, false:bit, nil
       :lng);
12 X_170:bat[:oid] := algebra.difference(X_169:bat[:oid], X_167
       :bat[:oid], nil:BAT, nil:BAT, false:bit, nil:lng);
13 X_172:bat[:str] := algebra.projectionpath(X_170:bat[:oid],
       C_146:bat[:oid], X_78:bat[:oid], X_45:bat[:str]);
14 X_176:bat[:str] := batstr.substring(X_172:bat[:str], 1:int,
       2:int);
15 (X_179:bat[:oid], C_180:bat[:oid], X_181:bat[:lng]) := group
       .groupdone(X_176:bat[:str]);
16 X_182:bat[:str] := algebra.projection(C_180:bat[:oid], X_176
       :bat[:str]);
17 X_183:bat[:lng] := aggr.subcount(X_179:bat[:oid], X_179:bat
       [:oid], C_180:bat[:oid], false:bit);
18 X_173:bat[:lng] := algebra.projectionpath(X_170:bat[:oid],
       C_146:bat[:oid], X_81:bat[:lng]);
19 X_185:bat[:hge] := aggr.subsum(X_173:bat[:lng], X_179:bat[:
       oid], C_180:bat[:oid], true:bit, true:bit);
20 (X_188:bat[:str], X_189:bat[:oid], X_190:bat[:oid]) :=
       algebra.sort(X_182:bat[:str], false:bit, false:bit);
21 X_193:bat[:hge] := algebra.projection(X_189:bat[:oid], X_185
       :bat[:hge]);
22 X_192:bat[:lng] := algebra.projection(X_189:bat[:oid], X_183
       :bat[:lng]);
23 X_191:bat[:str] := algebra.projection(X_189:bat[:oid], X_182
       :bat[:str]);
24 end user.s2_1;
```

Listing 8.12: MAL code of Q22 by MonetDB (cont. 2)

Bibliography

[Aba08] Daniel J. Abadi. *Query execution in column-oriented database systems*. PhD thesis, Massachusetts Institute of Technology, 2008. (cited on Page 4)

[ABH09] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. Column-oriented database systems. *Proceedings of the VLDB Endowment*, 2(2):1664–1665, August 2009. (cited on Page 4)

[ADH02] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal—The International Journal on Very Large Data Bases*, 11(3):198–215, 2002. (cited on Page xi, 4, and 5)

[ADHS01] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and Marios Skounakis. Weaving relations for cache performance. In *VLDB*, volume 1, pages 169–180, 2001. (cited on Page 3 and 5)

[AMH08] Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the International Conference on Management of Data*, SIGMOD '08, pages 967–980. ACM, 2008. (cited on Page 3)

[Ams97] CWI Amsterdam. MonetDB. https://github.com/MonetDB/MonetDB, 1997. (cited on Page 8)

[APM16] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. Bridging the archipelago between row-stores and column-stores for hybrid workloads. In *Proceedings of the International Conference on Management of Data*, pages 583–598. ACM, 2016. (cited on Page 4 and 5)

[BBK98] Stefan Berchtold, Christian Böhm, and Hans-Peter Kriegal. The pyramid-technique: Towards breaking the curse of dimensionality. In *ACM SIGMOD Record*, volume 27, pages 142–153. ACM, 1998. (cited on Page 10)

[BBR+13] Sebastian Breß, Felix Beier, Hannes Rauhe, Kai-Uwe Sattler, Eike Schallehn, and Gunter Saake. Efficient co-processor utilization in database query processing. *Information Systems*, 38(8):1084–1096, 2013. (cited on Page 52)

[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 *Proceedings of the international conference on Management of data*, pages 479–490. ACM, 2006. (cited on Page 6)

[BK94] Peter A. Boncz and Martin L. Kersten. Monet. An impressionist sketch of an advanced database system. In *Proceedings of the IEEE BIWIT workshop*. Citeseer, 1994. (cited on Page 5, 6, and 8)

[BK99] Peter A. Boncz and Martin L. Kersten. MIL primitives for querying a fragmented world. *The VLDB Journal—The International Journal on Very Large Data Bases*, 8(2):101–119, 1999. (cited on Page xi, 5, 6, 7, and 9)

[BKH+14] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. Ocelot/HyPE: Optimized data processing on heterogeneous hardware. *Proceedings of the VLDB Endowment*, 7(13):1609–1612, 2014. (cited on Page 52)

[BKSS17] David Broneske, Veit Köppen, Gunter Saake, and Martin Schäler. Accelerating multi-column selection predicates in main-memory - the Elf approach. In *Data Engineering (ICDE), 2017 IEEE 33rd International Conference on*, pages 647–658. IEEE, 2017. (cited on Page xi, 10, 11, 12, 18, 26, 27, 29, and 37)

[BQK96] Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its geographical extensions: A novel approach to high performance GIS processing. In *International Conference on Extending Database Technology*, pages 145–166. Springer, 1996. (cited on Page 7 and 51)

[BZN05] Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-pipelining query execution. In *Proceedings of the Conference on Innovative Data Systems Research*, volume 5, pages 225–237, 2005. (cited on Page 52)

[CK85] George P. Copeland and Setrag N. Khoshafian. A decomposition storage model. In *Proceedings of the International Conference on Management of Data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM. (cited on Page 4)

[CMK+15] Robin Cijvat, Stefan Manegold, Martin L. Kersten, Gunnar W Klau, Alexander Schönhuth, Tobias Marschall, and Ying Zhang. Genome sequence analysis with MonetDB. *Datenbank-Spektrum*, 15(3):185–191, 2015. (cited on Page 51)

[Cou14] Transaction Processing Performance Council. TPC benchmark H-standard specification,, 2014. (cited on Page 37)

[GKP+10] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A main memory hybrid storage engine. *Proceedings of the VLDB Endowment*, 4(2):105–116, 2010. (cited on Page 5)

[GKS16] Mrunal Gawade, Martin L. Kersten, and Alkis Simitsis. Multi-core column-store parallelization under concurrent workload. In *Proceedings of the 12th International Workshop on Data Management on New Hardware*, page 1. ACM, 2016. (cited on Page 6)

[GVK+14] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi Kuno, Joseph Tucek, Mark Lillibridge, and Alistair Veitch. In-memory performance for big data. *Proceedings of the VLDB Endowment*, 8(1):37–48, 2014. (cited on Page 7)

[HER15] Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the undefinedness of C. In *ACM SIGPLAN Notices*, volume 50, pages 336–345. ACM, 2015. (cited on Page 23)

[HP03] Richard A. Hankins and Jignesh M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *Proceedings of the 29th VLDB conference*, pages 417–428. VLDB Endowment, 2003. (cited on Page 5)

[HSP+13] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proceedings of the VLDB Endowment*, 6(9):709–720, 2013. (cited on Page 6 and 52)

[IGN+12] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, Sjoerd Mullender, Martin Kersten, et al. MonetDB: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the IEEE Computer Society Technical Committee on Database Engineering*, 35(1):40–45, 2012. (cited on Page 5, 6, 9, and 18)

[IKM09] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. Self-organizing tuple reconstruction in column-stores. In *Proceedings of the International Conference on Management of data*, pages 297–308. ACM, 2009. (cited on Page 6)

[Int16] Intel. Intel 64 and IA-32 architectures optimization reference manual, 2016. (cited on Page 40)

[LIMK12] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. MonetDB/Datacell: Online analytics in a streaming column-store. *Proceedings of the VLDB Endowment*, 5(12):1910–1913, 2012. (cited on Page 52)

[LP13] Yinan Li and Jignesh M. Patel. BitWeaving: Fast scans for main memory data processing. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 289–300. ACM, 2013. (cited on Page 1 and 51)

[MBK00a] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal—The International Journal on Very Large Data Bases*, 9(3):231–246, 2000. (cited on Page 1)

[MBK00b] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. What happens during a join? Dissecting CPU and memory optimization effects. In *Proceedings of the 26th international conference on very large data bases*, pages 339–350. Morgan Kaufmann Publishers Inc., 2000. (cited on Page 7)

[Pla09] Hasso Plattner. A common database approach for OLTP and OLAP using an in-memory column database. In *Proceedings of the International Conference on Management of data*, pages 1–2. ACM, 2009. (cited on Page 3 and 4)

[SBKZ08] Jan Schaffner, Anja Bog, Jens Krüger, and Alexander Zeier. A hybrid row-column OLTP database architecture for operational reporting. In *International Workshop on Business Intelligence for the Real-Time Enterprise*, pages 61–74. Springer, 2008. (cited on Page 3, 4, and 10)

[SDRK02] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the International Conference on Management of Data*, pages 464–475, June 2002. (cited on Page 11)

[SK13] Lefteris Sidirourgos and Martin L. Kersten. Column imprints: A secondary index structure. In *Proceedings of the International Conference on Management of Data*, pages 893–904. ACM, 2013. (cited on Page 1, 9, and 10)

[ZBNH05] Marcin Zukowski, Peter A. Boncz, Niels Nes, and Sándor Héman. MonetDB/X100: A DBMS in the CPU cache. *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005. (cited on Page 52)

[ZvdWB12] Marcin Zukowski, Mark van de Wiel, and Peter A. Boncz. Vectorwise: A vectorized analytical DBMS. In *IEEE 28th International Conference on Data Engineering*, pages 1349–1350. IEEE, 2012. (cited on Page 52)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 11. Mai 2018