Learned Encodings in SageDB

by

Lujing Cen

B.S., Massachusetts Institute of Technology (2020)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 14, 2021

Certified by .....
Tim Kraska
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Learned Encodings in SageDB

by

Lujing Cen

Submitted to the Department of Electrical Engineering and Computer Science
on May 14, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As the demand for data outpaces diminishing improvements in the hardware used to store and query them, we must find intelligent ways to increase database performance on existing systems. This project is focused on integrating learned encodings into SageDB, a database capable of accelerating queries by analyzing and adapting to different workloads. Encodings improve query performance through lossless compression, thereby reducing I/O time during scans. Different encoding types exhibit different characteristics depending on properties of the underlying data and the hardware on which queries are executed. We implement a variety of common encodings in SageDB and propose a learning-based approach to select the optimal encoding for a given data block by combining block-level statistics with sampling. In addition, we demonstrate how to leverage properties of encoded data along with vectorized processing units in modern CPUs to more efficiently execute queries without the need to decode every value.

Thesis Supervisor: Tim Kraska
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgments

Thank you to Tim Kraska, Ryan Marcus, and Andreas Kipf for being great mentors. Your insights and feedback have really helped me grow as a researcher and an engineer. I am really glad to have been able to work with the SageDB team and contribute to the field of learned systems.

Thank you to Sarah Raines for keeping me on track, helping me maintain healthy work life habits, and editing this thesis. I am very grateful for all your support, care, and love.

Thank you to my parents for supporting me throughout my MIT journey and encouraging me to pursue my interests. You were always there for me when I was stressed or frustrated.

Finally, thank you to my grandpa for cultivating my interest in engineering and computer science. You showed me the beauty of knowledge and the importance of passing it on to others.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

In the last 10 years, the world has rapidly transitioned into a more data-driven economy [18]. Data warehouses containing petabytes of information are used to drive decisions like how successful a new feature is or whether a new store should be opened in a particular region. However, computational limits are quickly becoming a bottleneck for databases that have large amounts of data and complex queries. The end of Dennard scaling means that we cannot rely on hardware alone to achieve reasonable performance improvements to match the ever-increasing demand for more data [8]. Instead, we need smarter ways to exploit existing hardware and algorithms. For a long time, this has been done through manually tuning databases, designing data layouts, and choosing the appropriate indexes and encodings for table columns. The major downside to this approach is that it requires dedicated engineering effort for each system. New use cases and workloads often require re-tuning the database and re-engineering table layouts.

1.1 Instance-Optimized Databases

Recently, there has been a growing interest in instance-optimized databases. A database instance is a combination of query workloads and the hardware on which those queries are executed. To some extent, databases already perform some optimizations at runtime such as determining the best join order based on statistical

properties of the data and predetermined heuristics. However, production instances of these systems often require significant tuning or manual guidance in order to achieve optimal performance.

In theory, an instance-optimized database could completely remove the need for manual tuning. Such a system would be able to automatically specialize to a given application or workload through a combination of code synthesis, data analysis, and machine learning. This thesis contributes a few building blocks related to data encodings for a specific instance-optimized database known as SageDB. The ideas presented here can be generalized for other instance-optimized systems.

1.2 Thesis Contributions

The focus of this thesis is the design, implementation, and automatic selection of data encodings in SageDB. We lay out the main contributions below.

Efficient Encoding Implementations. We implement a few well-known data encodings in SageDB using C++17 that support query compilation, grouped columns, and efficient reads through an iterator interface.

Operations on Encoded Data. We show how to leverage properties of encodings to more efficiently execute filters, aggregations, and group by operations in queries without the need to fully decode data blocks. By taking advantage of vectorized execution through SIMD[1], some queries are able to achieve speeds of over one billion rows per second on a single core.

Automatic Encoding Selection. We present a learning-based approach that can automatically determine the best encoding for a given data block. Whereas existing systems rely on fixed cost models and heuristics to select encodings, this new technique is able to optimize for different objectives in an instance-optimized manner with minimal evaluation overhead and training time on the target hardware. Support for new encodings can be added without re-training existing models. Selections made by this approach are close to optimal for a variety of datasets and objectives.

[1]single instruction, multiple data; available in modern x86-64 and ARM CPUs

Learned Encoding Advisor. As an extension of the work done here, we published a workshop paper [2] which applies our learning-based encoding selection on a commercial column store. We demonstrated that our technique outperforms state of the art heuristics by a substantial margin on query latency and storage size. Integration was done using only the SQL interface provided by the database and no modifications were made to the models or parameters presented in this thesis.

1.3 Organization

In Chapter 2, we provide an overview of SageDB and how this work integrates with its existing components. In Chapter 3, we present the implementation of encodings in SageDB along with the design of operations on encoded data. In Chapter 4, we detail the architecture of automatic encoding selection and how it integrates into the existing system. A brief discussion of design decisions are presented in Chapter 5, followed by experimental results in Chapter 6. Finally, future work and conclusions are presented in Chapter 7 and Chapter 8 respectively.

Chapter 2

Background on SageDB

In this section, we provide some background information about SageDB. A majority of the work presented in this thesis are extensions on top of the existing system. At the time of writing, SageDB is an instance-optimized query accelerator for PostgreSQL [13]. It is capable of automatically adapting to different workloads and data distributions. Much of the design philosophy of this system relies on the idea that traditional database components like index structures and query cost estimators can be completely replaced by learned components. We begin with a high level overview of the system, followed by descriptions of relevant modules.

2.1 Architecture Overview

SageDB sits behind a deployment of PostgreSQL. The user specifies one or more tables that they would like SageDB to optimize. Through the usage of a foreign data wrapper, the PostgreSQL query optimizer is able to push down queries on supported tables to SageDB. Operations which cannot be handled by SageDB always fall back to the their default PostgreSQL implementation.

The architecture of SageDB consists of a frontend written in Rust, a data layout optimizer known as Hurricane, and an execution engine that relies on a collection of C++ templates. The interaction between different components can be seen in Figure 2-1. When a query is received by SageDB, it is placed into a query log for later

analysis. The frontend performs the necessary query optimizations and generates a C++ program, which is then compiled and executed.

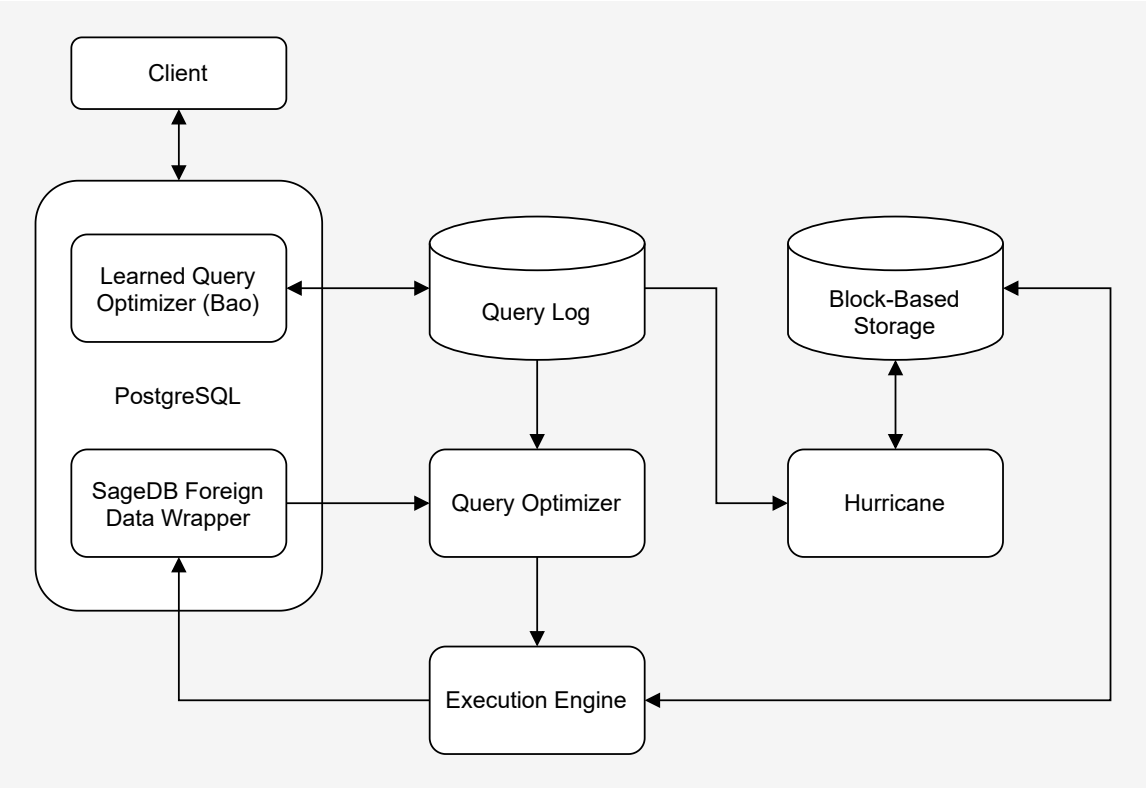

Figure 2-1: Architecture of SageDB.

A learned query optimizer known as Bao [15] is integrated into the PostgreSQL deployment. It uses reinforcement learning to provide query optimization hints based on the runtime characteristics of different queries that have previously been executed. Eventually, the goal is for Bao to be integrated into the SageDB query optimizer as well, but that work is beyond the scope of this thesis.

At the moment, SageDB is a single-node database and does not support distributed query processing. Data is persisted on disk or in a non-local storage bucket like Amazon S3. Inserts are first sent to PostgreSQL before being processed by SageDB. As such, the overall system generally holds two copies of the data, which is comparable to production-capable databases used in industry. SageDB does not support insertions or deletions of individual rows. Instead, it occasionally updates its optimized copies of tables through batch appends or re-writes.

2.2 Query Templates

A database that relies on query compilation has a drawback in that it must incur the cost of compilation for each new query that it sees. Generally, the performance improvements from query compilation outweigh the added overhead. However, many applications in practice use the same set of query templates. A query template has one or more parameters that can be changed, as seen in Listing 2-1.

```
SELECT
  site_name,
  COUNT(*)
FROM
  denorm_so
WHERE
  score < $1:INT64
  AND comment_count < $2:UINT64
GROUP BY
  site_name;
```

Listing 2-1: Example of a query template.

When compiling a query template, SageDB notes the parameters that can change, and generates a C++ program that accepts those parameters at runtime through standard input. This way, the compiled query can be cached for a given query template so that repeated executions of the same type of queries with slightly different parameters will not trigger re-compilation.

2.3 Block Storage

SageDB mainly operates as a column store. Data is stored in blocks[1] that each contain up to one million rows. Unlike conventional column stores where only a single column is stored per block, SageDB supports column groups. A column group is simply a subset of all columns in a table. Its data is stored row-by-row but contains only the specified columns. This is useful for taking advantage of locality for queries that

[1] sometimes known as segments or horizontal partitions

frequently access multiple columns together. The selection of these column groups is discussed in more detail in Section 2.5.

Prior to writing out a block, SageDB finds the minimum, maximum, and cardinality of all values in that block for each column. This allows for an operation known as block pruning during query execution. Given a set of query predicates, SageDB uses the Z3 SMT solver [7] to test whether each block will satisfy the query. Eliminated blocks are not scanned at all during query execution, which can significantly reduce I/O time. To support query templates, the blocks satisfying a query are passed as a bitmap to the compiled C++ program at runtime.

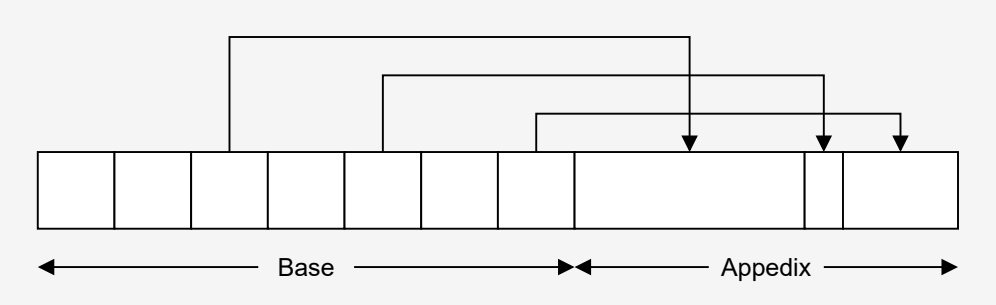

Figure 2-2: Block format on disk.

A single block consists of two sections, as seen in Figure 2-2. Each row in the base section has a fixed width that is known at compile time. This allows for quick random access into a block. Data types like integers are stored entirely in the base section. Variable width data types like strings may require the usage of the appendix section. Short strings are stored directly in the base section. If a string exceeds a certain length, the row corresponding to the string in the base section is replaced by an offset into the appendix section, which can be used to access the full string.

Note that SageDB does not rely on an internal buffer manager. Instead, blocks are read in using `mmap`[2], which handles caching of data pages and simplifies the design of the system. However, encodings must take into account the presence of a memory mapped appendix section when dealing with variable width data to avoid materializing long strings unless they are explicitly accessed.

[2]https://man7.org/linux/man-pages/man2/mmap.2.html

2.4 Pipelined Execution

SageDB uses a pipelined execution model consisting of sources and sinks. It share some elements with the volcano iterator model [10]. A source produces tuples while a sink consumes tuples. Sinks generally perform some transformation or operation on the tuples. Once a sink has received all of the tuples, it can produce another source for further processing. All queries end with a sink that outputs the result to the Rust frontend. The interface for sources and sinks containing relevant methods can be seen in Listing 2-2.

```
class Source {
  // Returns whether the source has more rows.
  bool has_next();
  // Advances the source to the next row.
  void next();
  // Gets the current tuple element at a column index.
  template <size_t idx>
  auto get();
  // Marks the source as finished and performs cleanup.
  void finish();
};

class Sink {
  // Sinks all columns of the current row from the source.
  template <class Source>
  void sink_all(const Source& source);
  // Marks the sink as finished, possibly returning a source.
  auto finish();
};
```

Listing 2-2: Interface for sources and sinks.

Consider the aggregation query presented in Listing 2-1. An example of the generated code can be seen in Listing 2-3. Note that various initialization and finalization steps have been omitted for simplicity. The code assumes that each block contains only one column. A `ChunkSource` takes the blocks for a given column and materializes them one row at a time. A `ZipSource` combines multiple other sources into a single source containing multiple columns. We've left out the implementation of

`filter`, which is used to test whether a row satisfies the query predicates. Tuples first flow into the `HashAggregationSink`, which performs the grouping and aggregations in memory. When it is finished, a new source is produced containing the results of the group by. Finally, a `JSONSink` is used to output the results.

```
static ChunkSource<...> score(...);
static ChunkSource<...> comment_count(...);
static ChunkSource<...> site_name(...);
static ZipSource<&score, &comment_count, &site_name> source0;
static HashAggregationSink<..., CountAggregator> sink0;

while (source0.has_next()) {
  if (filter.check(source0)) {
    sink0.sink_all(source0);
  }
  source0.next();
}

static JSONSink<...> sink1;
auto source1 = sink0.finish();

while (source1.has_next()) {
    sink1.sink_all(source1);
    source1.next();
}
```

Listing 2-3: Generated code for query template.

Queries of arbitrary complexity can be handled using pipelined execution. One major benefit is that the compiler can perform all of the necessary optimizations for the given hardware, which means that the Rust frontend only needs to generate high level C++ code without worrying about low level details.

2.5 Layout Optimization

Over time, SageDB will gain an understanding of the type of queries that will be frequently executed based on the query log. It will asynchronously invoke a module known as Hurricane to perform layout optimization. The task of layout optimization

is to figure out how to best organize blocks such that the overall cost of running queries present in the workload is minimized subject to certain storage constraints. This involves selecting column groups, sort orders, and partitions. Hurricane tries to solve the same problem as Qd-trees [32], but differs in that its eventual goal is to perform global optimizations that take into account all levels of the memory hierarchy.

The work in this thesis relies on the assumption that the layout has already been determined. This is reasonable in the presence of a layout optimizer, but also because many existing databases store blocks that contain a single column sorted based on the primary key. Learned encodings will allow the layout optimizer or database to efficiently determine the best encoding for blocks by providing accurate cost predictions. It is not highly concerned with determining the right objective to use for the layout, although we will present a few to evaluate the performance of encoding selection.

Chapter 3

Encoding Design

In this chapter, we describe the design of different encodings and how they are integrated into SageDB. We will also look at how queries can take advantage of the properties of encoded data by switching from a tuple-by-tuple execution format to a vectorized execution format. An overarching design philosophy is that abstractions should introduce no runtime overhead while requiring minimal changes to the code generation phase.

3.1 Shared Components

Encodings share a common base interface and various subroutines used to handle nested compression schemes. We first look at bit packing and varint, which are used to compress integral data types. Then, we describe the base sink and source classes that are used by all encoding implementations.

3.1.1 Bit Packing

One way to compress a list of non-negative integers is to take advantage of the fact that there are often leading zero bits. Computers are byte addressable, which means that the smallest unsigned integral type is `uint8_t`, followed by types that are powers of two up to `uint64_t` for most 64-bit processors. Storing a list of n uncompressed

`uint64_t` values requires at least $8n$ bytes. However, if all the integers in the list are small, then we do not need a full 8 bytes for each value.

Bit packing stores each unsigned integer using as many bits as necessary to represent the largest value without leading zeros. The values are packed tightly together, which means that one byte could contain bits from multiple values. In the worst case, this method will still require 8 bytes for each `uint64_t`.

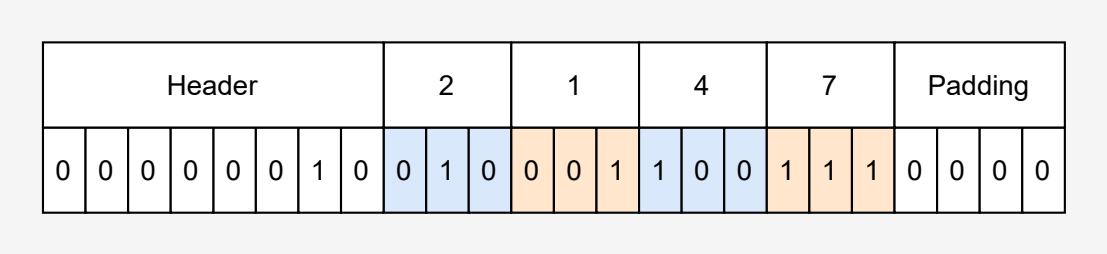

Figure 3-1: Example of bit packing on 4 integers.

An example of bit packing can be seen in Figure 3-1. A one byte header is used to store the number of bits in each value. The entire bit pack is zero padded to the next full byte. Note that this compression scheme supports random access, since each value has a fixed width. Special care must be taken when decoding packed values that are 58 bits or wider. Usually, it is possible to compute the starting byte for a given index and load in a `uint64_t` containing the entire packed value. A packed value can start at the last bit of its first byte, which means that large packed values sometimes require an additional byte to be read.

3.1.2 Prefix Varint

A variable-width integer (varint) is another compression scheme for integers that complements bit packing. One advantage of varint is that it encodes each value independently, so a single outlier would not greatly affect the compressed size of a block. However, varint does not support random access, since we must perform a sequential scan of the packed values in order to determine the width of each value.

In SageDB, we implemented a type of varint compression that uses a prefix. It was invented independently by various systems [17] and relies on the fact that modern

CPUs have an efficient instruction to count the number of trailing zeros. The compression scheme works by using trailing zeros in the least significant byte to represent the size of the value. See Table 3-1 for more information. Decompression works by using the prefix byte to determine the size of the encoding, loading a `uint64_t`, and zeroing out unused bits. Note that values are assumed to be stored in little-endian format on disk.

Prefix	**Capacity (bits)**	**Size (bytes)**
xxxxxxx1	7	1
xxxxxx10	14	2
xxxxx100	21	3
xxxx1000	28	4
xxx10000	35	5
xx100000	42	6
x1000000	49	7
10000000	56	8
00000000	64	9

Table 3-1: Relation between prefix, capacity, and size for prefix varint.

Varint compression works poorly with negative integers since they contain no leading zeros. In order to efficiently compress negative values, signed integer types are first encoded using ZigZag [21] before the varint compression stage. If an integer x is non-negative, ZigZag encodes it as $2x$. Otherwise, ZigZag encodes it as $2\,|x| - 1$. The encoding and decoding phases can both be represented using only a few bitwise operations. Applying ZigZag allows signed integral data with small absolute value to achieve a good compression ratio, which is necessary for some data distributions.

3.1.3 Encoding Sink

The encoding sink is a base class that is shared by all encoding types. It provides convenience methods for derived classes to write different compressed and uncompressed data types. The encoding sink does not assume that writes are append-only, and exposes methods for derived classes to obtain and change the write offset arbitrarily.

All writes to the base section are cached in a buffer that is flushed when it is full or when a seek occurs. To maximize disk throughput, writes to storage are handled

through the POSIX asynchronous I/O interface. When `finish` is called on the sink, a header containing the size of the base section is written out. This is necessary because decoding variable width data types requires knowledge of where the appendix section starts. At this point, all data for the appendix section is also written out directly after the end of the base section.

3.1.4 Encoding Source

The encoding source is also shared by all encoding types. It is used to manage memory for read operations and provides convenience methods to decode and decompress different data types. A block is memory mapped to a virtual address using `mmap`, which offloads the task of caching and reading data to the operating system. This design allows frequently read blocks to be available in memory across different query executions without a persistent buffer manager.

Bit packing and prefix varint both read unaligned `uint64_t` values without regard to the end of the base section. This improves performance because the decoding phase does not need to check if it will perform an out-of-bounds read for each value. Due to the design of `mmap`, it is not valid to simply memory map a region larger than the underlying file, because attempting to access a page not backed by the file triggers a `SIGBUS`. Instead, we check if adding an 8 byte padding to the size of the file would result in a new page. If so, the encoding source will `mmap` an additional slop page directly after the end of the last page backed by the file.

3.2 Supported Encodings

We have implemented five existing encodings for SageDB. These encodings all support column groups and cluster the data for each row to leverage benefits that arise from locality. Note that this is different from concatenating the encoded data for each column in the column group, which would cause values from the same row to be placed on different pages in memory. For each of the encodings, we will discuss how they are implemented, what data distributions they should be used on, and any

subtleties in their operation.

A diagram of the layout of all encodings is shown in Figure 3-2, but more details will be provided in the individual sections. In general, encodings do not store information about the number of rows. That information is assumed to be provided at compile-time or runtime by an internal manifest used by SageDB.

More encodings schemes and implementations that are highly optimized for SIMD-capable CPUs have been studied [14, 24], but we did not incorporate them here because they are not necessary to achieve significant query speedups. Furthermore, the encodings that we have chosen to integrate into SageDB are more useful for studying automatic encoding selection in the context of instance-optimized systems as they do not depend on a particular set of CPU instructions.

3.2.1 Delta Encoding

Delta encoding compresses data by representing rows as the difference between the current row and the previous row. The first row of a block is stored in uncompressed format. The remaining rows are stored using delta values that are varint encoded. In the case of multiple columns, the delta values for each row are stored next to one another.

Delta encoding works great when a column is sequential or sorted because the differences between adjacent values tend to be small. This encoding scheme can only handle fixed width data types and does not support random access. Reading a value at a given index requires summing over all previous values in the column. Therefore, it is not suitable for certain optimizations, as we will discuss in Section 3.4.

3.2.2 Dictionary Encoding

Dictionary encoding assigns an integer key to each distinct value in the block, effectively creating a dictionary. Each row in the block is represented by an integer key that can be used to look up the value in the dictionary. In terms of implementation, it is enough to store the number of distinct values followed by the values themselves.

The index of the values implicitly defines their key. Decoding a value from an index is therefore as efficient as an array lookup. We chose to apply bit packing over the key for each row, which improves the compression ratio without sacrificing too much performance. This is especially necessary for integral types, since the uncompressed key itself could be larger than the size of one row.

This encoding scheme supports random access and variable width data. If there is an appendix section, it is stored directly after the last bit packed value. Dictionary encoding works well when the cardinality of the block is small. Column types involving the group names of categorical data can often see a lot of performance benefits from using dictionary encoding.

3.2.3 Frame of Reference Encoding

Frame of reference encoding works by finding the minimum value for each column. Those minimums (the reference row) are stored first, followed by the delta of each row from those minimums. This is similar to the scheme of delta encoding, except all of the deltas are relative to the minimum of each column rather than the previous row. The delta values are bit packed rather than varint encoded because all of the deltas are non-negative and because bit packing supports random access.

This encoding scheme only supports fixed width data. Random access can be achieved by decoding the delta values for a given row offset and then adding them to the reference row. Frame of reference encoding works well on a variety of real world column types. There are many instances where small numbers are stored using a larger integral type. In other cases, the range of the numbers in a block could be small compared to their magnitude.

3.2.4 Run Length Encoding

Run length encoding compresses data by organizing them into contiguous runs of the same row. For each run, a varint encoded count along with the uncompressed row is stored. Decoding works by storing a row and the number of tuples remaining in the

current run. The same row is returned until the run is exhausted. The properties of run length encoding make it an ideal candidate for vectorized execution, which we examine in more detail in Section 3.4.

Run length encoding supports all data types that have a well-defined equality operator. If an appendix section exists, it is stored directly after the last run. This encoding scheme works well for data that has very low cardinality or contains reasonably long runs of the same row. In many cases, run length encoding can outperform dictionary encoding in both decoding speed and compression ratio because it requires very little space to represent a contiguous run.

3.2.5 Zstandard Encoding

Zstandard encoding utilizes the `zstd` lossless compression algorithm developed at Facebook. `zstd` outperforms `zlib` in almost all regards [3], which makes it an ideal candidate for a block encoding scheme. In SageDB, Zstandard encoding is applied on the block without any additional transformations. Note that both the base and appendix sections in the block are compressed using `zstd`. For simplicity, we store the size of the base section of the uncompressed block before the compressed data. This makes it possible to recover the offset of the uncompressed appendix section without relying on information about the number of rows in the block.

Because `zstd` is a general purpose compression algorithm, it achieves great compression ratios on all data types, especially long strings. However, it is not always the case that `zstd` outperforms all other encoding methods in terms of compressed size (see Chapter 4). Furthermore, `zstd` is rarely the fastest compression method because the entire block must be decompressed before rows can be read. These trade-offs make the task of encoding selection a lot more meaningful in real world applications.

3.3 Queries on Encoded Data

Running queries efficiently on encoded data presents some unique challenges. For one, we do not make the assumption that all blocks in the same column group have

Figure 3-2: Layout of all encodings on a example input consisting of six `uint32_t` values. Note that the number of bytes occupied by each encoding is not represented by the size of the squares.

the same encoding. This means that the compiled query must be able to determine the right decoding method to use at runtime. We will first present how we solve this problem efficiently in SageDB and then describe the changes necessary to support queries over encoded blocks.

3.3.1 Efficient Variant

All of the encoding types conform to the same encoding sink and source interfaces shown in Listing 2-2. Throughout the rest of this section, we will only be concerned with source operations because SageDB is optimized for reads. The same principles can be applied to sink operations, but they do not warrant a separate discussion.

When executing a query over encoded data, SageDB uses a manifest to determine the type of each block at runtime. Each encoding type has a different implementation of the source interface. Given a call to a source method, the compiler needs to know what implementation to use. This is simple when there is only one encoding type because there is only one implementation of each method. With multiple encoding types, additional state needs to be kept about what the current block's encoding type is so that an implementation can be chosen for each source method invocation.

A common way to solve this problem is through dynamic dispatch. It is achieved in C++ by declaring an abstract class containing pure virtual methods along with subclasses that implement those virtual methods. In order to know which implementation a virtual method corresponds to, the compiler adds an additional layer of indirection known as a virtual method table (VMT). We will not discuss VMTs in detail here, but it is important to note that they add some overhead to each function call [1]. Furthermore, virtual methods usually cannot be inlined because the compiler does not know the exact implementation at compile-time.

In SageDB, we chose to implement all interfaces without the usage of any virtual methods. Instead, we rely on C++17's `std::variant`, which is a class template representing a type-safe union. The variant stores which type is currently active, and supports the visitor pattern using `std::visit`. Unfortunately, the standard library's implementation of `std::visit` effectively compiles to a VMT (see Listings A-1 and

Listing A-2. Because many source operations like `has_next` and `next` take only a few cycles in some encoding implementations, the overhead of a VMT is unacceptable.

```
template <class Function, class Variant>
decltype(auto) optimized_visit(Function&& f, Variant&& v) {
  constexpr size_t v_size =
      std::variant_size_v<std::decay_t<decltype(v)>>;

  switch (v.index()) {
    case 0: {
      if constexpr (v_size > 0) {
        return f(*std::get_if<0>(&std::forward<Variant>(v)));
      }
    }
    case 1: {
      if constexpr (v_size > 1) {
        return f(*std::get_if<1>(&std::forward<Variant>(v)));
      }
    }
  }
}
```

Listing 3-1: Optimized visit method for variants with up two alternative types.

We implemented our own visit method for `std::variant` that draws inspiration from [19]. It permits inlining and therefore allows the compiler to generate near optimal assembly. See Listing 3-1 for an example implementation. One additional benefit of `optimized_visit` is that the compiler can automatically determine the best way to handle the dynamic dispatch. For variants containing only a single type, this will be as efficient as using the underlying type directly. An example program and generated assembly using the optimized visit method can be seen in Listing A-3 and Listing A-4.

3.3.2 SageDB Changes

The query parser of SageDB was modified to accept encodings during the creation of column groups. An example query can be seen in Listing 3-2, which creates a projection consisting of two encoded column groups that each contain a single column.

The same encoding will be applied to all blocks of the column group. In the case that no encoding is specified for a column group, SageDB will default to automatic encoding selection (Chapter 4).

```
CREATE PROJECTION [
  (id) ENCODE "delta",
  (poster_reputation) ENCODE "frame-of-reference"
] FROM denorm_so;
```

Listing 3-2: Query that creates a projection with two encoded column groups.

An additional entry was added for each manifest file to indicate its encoding type. This manifest is compiled into the query executable for simplicity, although it could also be supplied at runtime. Two new classes — `EncodingChunkSink` and `EncodingChunkSource` — were created to handle writing to and reading from encoded blocks respectively. They accept a manifest list that contains the encoding type for each file and utilize the optimized variant described in the previous section to handle moving from one block to the next.

An optimization that the Rust frontend makes when compiling the query is that only the encoding types which are used in the manifest are included in the variant alternatives. This not only reduces the binary size but also ensures that columns using a single encoding don't have to incur the cost of dynamic dispatch.

3.4 Operator Specialization

Applying encodings over columns usually increases the overall query latency if many blocks are already cached in memory. However, the performance of encoded columns can be significantly improved by leveraging an execution model known as vector-at-a-time (vectorized) processing [12]. This allows each encoding to have specialized implementations of different operators, which was not previously possible in SageDB under the iterator execution model. We assume that the query optimizer will decide the best execution pathway for a given query. We will mostly examine the design of

the new interface and the specializations for different encodings.

```
template <class T>
class TileOperator {
  // Loads the next `n` values into an array. Returns a pointer to
  // the start of the array. Using the buffer is optional.
  const T* load(size_t n, T* buffer);
  // Evaluates the filter on each row of the source into the mask.
  template <class Function>
  void filter(const Function& filter, uint8_t* mask, size_t n);
  // Updates the aggregator using the provided mask.
  template <class class Aggregator>
  void aggregate(Aggregator* aggregator, const uint8_t* mask,
                 size_t n);
  // Assigns group IDs to each row where the mask is true. Updates
  // the map with the assignments.
  void group(std::unordered_map<T, uint32_t>* map,
             uint32_t* group_ids, const uint8_t* mask, size_t n);
};
```

Listing 3-3: Interface of operator specialization for vectorized query execution.

There are four operations that different encodings can specialize on. The interface is shown in Listing 3-3. For simplicity, we assume that each encoding type contains only one column. In the actual implementation, SageDB supports specifying a template column index for each of these operations. The base `TileOperator` class uses curiously recurring template pattern (CRTP) to supply default implementations to derived classes. This significantly reduces code duplication, since each encoding type only needs to override a subset of supported operators. More details about these specialized operations are in the next few sections.

Given these specializations, it is possible to execute different parts of a query using tiles — small contiguous sections of a single column. Each tile is generally small enough to fit in L1 cache and permits efficient vectorized execution using SIMD. Most operations involve the usage of a mask. Rather than conditionally skipping a row if it does not satisfy the filter, the result of applying the filter over a tile is stored in a mask that can be passed to other operations. This allows some specializations to avoid branching altogether when processing a tile.

```
static EncodingChunkSource<...> score(...);
static EncodingChunkSource<...> comment_count(...);
static EncodingChunkSource<...> site_name(...);
static HashAggregationSink<..., CountAggregator> sink0;

tiled(total_rows, [&](const Tile& tile) {
  tile.store(tile_group(score, comment_count, site_name));

  auto mask0 = tile.filter(score, filter0);
  auto mask1 = tile.filter(comment_count, filter1);
  auto mask2 = tile.transform(tile_group(mask0, mask1),
      [](auto a, auto b) { return a && b; });

  tile.update(tile_group(site_name), sink0, mask2);
});
```

Listing 3-4: Generated code for vectorized query execution.

Listing 3-4 shows a compiled version of the query previously introduced in Listing 2-1 but using vectorized execution instead of row-by-row execution. We omitted code for outputting the aggregation result, which is the same as that found in Listing 2-3. The total number of rows in a query is not always a multiple of the tile size (1024 in the case of SageDB). Therefore, a helper function is used to automatically invoke the lambda function for each tile group. The `Tile` class contains useful abstractions for operating on individual tiles and the tile group as a whole.

One complexity associated with vectorized execution in combination with operator specialization is that it is sometimes more efficient to operate on encoded data multiple times than it is to materialize a decoded tile first. One main example of this can be seen in Section 3.4.4. Because all sources conform to an iterator-like interface, tile operations cause sources to advance. To allow sources to rewind to the beginning of a tile, we added two new methods to the source interface – `store` and `recall`. Invoking `recall` restores the state of a source to the last call of `store`. This recall mechanism can be efficiently implemented by storing any non-constant state of a source in additional class members.

We can now describe all of the steps in Listing 3-4. First, `store` is called on all

columns. This is necessary so that other operations can freely rewind any source. Filters are applied to the individual columns and combined using the `transform` function. Finally, the aggregation sink is updated using `update`, which in turn uses the specialized `group` operator to efficiently aggregate on `site_name`.

3.4.1 Load

In some cases, the same tile participates in more than one operation. For example, a column could be used in multiple aggregations and also appear in filter clauses. For some encoding types, this could be more expensive than first decoding the data into a buffer. Once the buffer has been materialized, many operators will be able to take advantage of SIMD instructions. It is up to the optimizer to determine whether to materialize a column when using vectorized processing.

Dictionary and frame of reference encodings have a specialization of `load` that can leverage SIMD to decode bit packed data. For bit packs whose element widths are less than 58 bits, decoding uses only arithmetic and bitwise operators, which permits vectorization. For run length encoded data, the value of each run is copied into the output multiple times. This is easily vectorized by the compiler. Note that none of the implementations we describe here involve manually writing assembly for different processor types. Instead, we rely on the auto-vectorization capabilities found in modern compilers, which are fairly good at handling most tile operations.

For Zstandard encodings with only one column, there is no need to copy the data into the output buffer. Instead, `load` will simply return a pointer to the appropriate offset in the base section of the decompressed data.

3.4.2 Filter

The filter operation for dictionary encoding can be specialized because columns which use this encoding generally have low cardinality. The predicate function is evaluated once for each distinct value in the encoding and cached so that additional calls to `filter` on the same source do not need to re-evaluate the predicate function. Writing

out the mask involves looking up the predicate result corresponding to each bit packed value. This specialization can reduce the number of predicate evaluations by a few orders of magnitude, which provides significant performance benefits, especially when the filter is complex or involves strings.

Run length encoding supports a similar kind of specialization for the filter operation. The predicate function only needs to be evaluated once per run, and the result can be placed into the mask using `memcpy`. This will almost always outperform evaluating the predicate function once per row provided that run length encoding is appropriate for the column.

3.4.3 Ungrouped Aggregation

Ungrouped aggregations (the `aggregate` function) achieve speedups through vectorization of branch-free code using a mask. In general, aggregations take a variable amount of time depending on the selectivity of the predicates. Predicates with extremely low selectivities do not benefit much from this execution model, since very few values are decoded in the first place. However, it is generally safe to assume that aggregations will be run on queries with predicates that are not very selective. This is even more true in the case of SageDB, since block pruning skips blocks that are certain to contain no values satisfying the predicates.

Encoding	**Specialized Aggregators**
None	`COUNT`, `MAX`, `MIN`, `SUM`
Delta	`COUNT`
Dictionary	`COUNT`
Frame of Reference	`COUNT`, `MAX`, `MIN`
Run Length	`COUNT (+DISTINCT)`, `MAX`, `MIN`, `SUM`
Zstandard	`COUNT`

Table 3-2: Aggregator specialization support for different encoding types.

A summary of the supported aggregator specializations can be seen in Table 3-2. The `COUNT` aggregator specialization is shared among all encoding types because it relies only on the mask. Updating the aggregator simply involves counting the number of true values in the mask.

The `MAX` and `MIN` aggregators can be updated in a branch-free manner for decoded values using bit hacks. See Listing A-5 for an example implementation. The basic idea is to use the mask byte to alter the current value to be either the smallest or largest value of the type, respectively, for `MAX` and `MIN`. Thus, locations where the mask is false will effectively be excluded. A similar technique can be used to update the `SUM` aggregator by setting the value to zero where the mask is false.

For frame of reference encoding, it is not necessary to fully unpack the data first when updating `MAX` and `MIN` aggregators. The maximum and minimum of the bit packed delta values are also the maximum and minimum for the block when the reference value is added.

For run length encoding, `MAX` and `MIN` are specialized by performing at most one update on the aggregator per run. It is efficient to check if the bit mask is true anywhere in a run. If that is the case, the value for the run is used to update the aggregator. A similar idea can be applied for the `COUNT DISTINCT` aggregator. The hash map only needs to be accessed at most once per run. Run length encoding also supports a specialization for the `SUM` aggregator. For each run, we can update the aggregator with the product of the run value and the number of locations where the mask is true for that run.

Even though not all encodings support all aggregator specializations, they can still benefit from vectorized processing because once they are materialized into a buffer using `load`, they can be treated as having no encoding. This means that all specialized aggregators for raw data can be applied.

3.4.4 Grouped Aggregation

In order to support `GROUP BY` specialization (the `group` function), we made a change to the way SageDB handles grouped aggregations. Initially, the `GROUP BY` aggregator maintained an internal hash map from keys to tuples of aggregators. Unfortunately, this does not integrate with vectorized processing. We added a layer of indirection by first mapping keys to group IDs, and then using the group IDs to index into a vector of aggregator tuples. This introduces minimal runtime overhead and permits

specializing the group ID assignment phase, which dominates in terms of CPU time when the total number of aggregators is small.

Dictionary encodings are especially efficient at assigning group IDs because each distinct value has already been assigned a unique ID. When `group` is called on a dictionary encoding source for the first time, a cached vector that maps bit packed indices to group IDs is constructed by iterating through all values in the dictionary in order. For each distinct value, a group ID can be assigned by performing a lookup or insertion into the group map. The output group IDs for the current tile can be obtained by unpacking each bit packed value and using it to index into the cached vector. Note that we cannot directly assign the dictionary indices as group IDs because the group map may already contain assignments from other sources.

Run length encoding also supports fast group ID assignments. Only one lookup or insert needs to be performed on the group map for each run. The group ID, which is the same for the entire run, can then be placed into the output using `memcpy`.

3.5 Related Work

Many modern column-oriented databases support a variety of data encodings [5, 6, 28, 31], but do so at the granularity of a column. The only system that supports different encodings for each column within a block is SingleStore [4]. However, none of these systems support encodings on column groups.

Apache Druid and SingleStore both indicate that they are able to perform operations on encoded data [4, 31]. Druid supports this mostly for dictionary encodings, whereas SingleStore has specializations for more encodings. There has also been work by the SingleStore team on leveraging SIMD to operate on encoded data in aggregations, which can be seen in the BIPie paper [16]. The work presented here is comparable to state-of-the-art operator specializations supported by SingleStore for a single table.

There have also been studies comparing different models of compiled query execution that leverage SIMD [12, 25], although they do not consider operations on

encoded data. We are mainly concerned with exposing an additional execution pathway which permits operator specialization. The task of choosing the best query plan is left to the optimizer and considered out of scope for this thesis.

Chapter 4

Automatic Encoding Selection

The goal of automatic encoding selection is to determine the best encoding type for each slice (a column within a block) in an instance-optimized manner based on the workload and hardware. Unlike previous approaches, we do not make assumptions about the configuration of the machines, the expected queries, or the encoding types that are implemented by the system. Instead, we present a generalized approach for encoding selection that uses estimates of properties from different encoding types to optimize for an arbitrary objective. We have implemented the selection strategy in SageDB and as a learned encoding advisor (see related work in [2]).

We first present some intuition about how automatic encoding selection can be achieved through feature estimation and why it is useful compared to other techniques. Next, we look at different models that are used to perform those estimations along with how they can be efficiently trained through synthetic distributions. Finally, we look at how automatic encoding selection is integrated with SageDB.

4.1 Intuition

Encoding selection can be seen in the context of a global optimization problem. If a database knows about the expected queries that it will receive and the slices that those queries will access, then it can try to organize and encode data in a way such that the overall query processing time is minimized. For example, this might involve

placing slices into different parts of the memory hierarchy based on access frequency.

The primary reason for encoding slices is to reduce their overall size on storage. Various systems make the assumption that smaller slices are better for query execution [22, 31]. With decreasing storage costs and the prevalence of solid-state drives, this assumption is no longer true (see Chapter 6). In addition, some encoding implementations rely on specific vector instructions [14, 16, 24] in order to achieve speedups. This means that the task of choosing an encoding is more complex than simply picking the one with the smallest size on storage.

Given these observations, we decided to take a learned approach to solving the problem of automatic encoding selection. We rely on the fact that it is possible to test the encodings of different slices on the target system. Given an encoding property (e.g., sequential read speed) and a slice, we can see how each encoding type performs on that slice. By looking at many slices containing different data distributions, we can train a model for each encoding that predicts the desired property from features of a block. An arbitrary optimization criteria (e.g., smallest size on storage) can be applied over the desired properties to select the best encoding type.

This learned approach has a few distinct advantages. First, it takes into account the runtime properties of the encodings which depend on the exact hardware configuration of the underlying system. Second, each model can be trained independently of other models, which means that adding an encoding type will not require re-training existing models. Last, the models perform regression rather than classification, which means that it is easy to incorporate an objective that uses the predicted properties.

4.2 Learned Models

There are three model types that we use to predict encoding properties — encoded size, memory speed, and storage speed. They are described in more depth in the next few sections. Note that the memory speed and storage speed model types encompass multiple models that predict different encoding properties. For example, the storage speed model type could contain models for predicting the cost of reading data from

local SSD and the cost of retrieving data from a network storage device. For a given encoding property, we train one model for each encoding type (delta, dictionary, etc.) and data type (fixed width or variable width) so that adding support for new encodings is possible without modifying existing models.

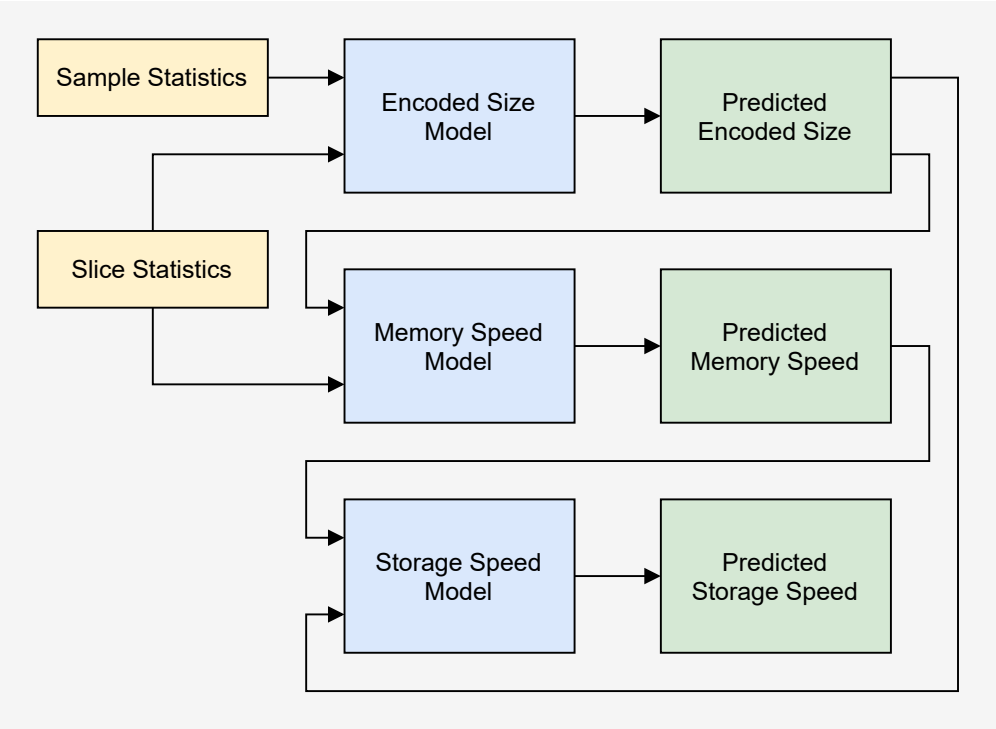

Figure 4-1: Pipeline of models used in automatic encoding selection.

A diagram of the model pipeline can be seen in Figure 4-1. To improve prediction accuracy, two categories of statistics are used as inputs to the models. Sample statistics are obtained by choosing a 1% contiguous sample of values uniformly at random from a slice. Encoding properties are extracted by applying the desired encoding on the slice. It is not feasible to test the speed of encodings on a sample of the slice at runtime because an accurate measurement would require multiple iterations and significantly degrade the performance of the overall system. Instead, we chose to only collect the encoded size of the sample, which can be done in a single pass in memory.

Slice statistics describe data in the entire slice as opposed to a contiguous sample. Some of these statistics like minimum and maximum are already used by other parts of the system for block pruning. All slice statistics should be cheap to compute in

one pass over the data. Although sample statistics provide meaningful information most of the time, they can lead to a biased view of the slice if a non-representative sample is chosen. For example, if a slice contains mostly small values but a few really large outliers, a sample would likely favor frame of reference encoding, which in reality achieves a poor compression ratio due to the large range of values in the slice. Empirically, slice statistics help correct poorly chosen samples and lead to more accurate predictions.

We use different slice statistics depending on the data type. For fixed width data, we compute the range (maximum minus minimum), cardinality, type width (bytes for a single value), and the first three moments (mean, variance, and skewness) of the distance between adjacent values. For variable width data, we compute the cardinality and mean length. The reason for this distinction is that range-like statistics are generally not useful in predicting the properties of data with unbounded width.

4.2.1 Encoded Size

The encoded size model type contains only one model for each encoding that predicts the size of the encoded slice given the slice statistics and sample statistics. This model type depends only on the implementation of the encodings and not on the target hardware. Therefore, the models can be pre-trained and distributed with the database. Having predictions about the encoded size of a slice is not only useful for optimizing storage costs, but also for predicting more complex and dependent encoding properties (see Section 4.2.2 and Section 4.2.3).

For fixed width data, we use random forest regression. In some cases, encoded size is neither a linear nor continuous function of the inputs. Other regression techniques like neural networks and support-vector machines perform poorly with a modest amount of training data. Given the constraint on the slice size, it is possible to construct training examples that explore the input space efficiently. For variable width data, we also use random forest regression for slices with small mean length (at most 64 in our implementation), but fall back to linear regression in other cases. Random forest regression is accurate when the range of inputs being evaluated fall

within the range of the training inputs. However, it performs poorly when the mean length is unbounded. For data containing longer values, linear regression works well because the encoded size of the sample is a good predictor for the encoded size of the overall slice.

4.2.2 Memory Speed

The memory speed model type includes models that predict how fast different operations take on a slice when it is already in memory. These models must be trained on the target machine because processor and memory configurations can significantly affect the performance of different encodings. Not all operations on an encoded slice take the same amount of time in memory. It is usually sufficient to use sequential scan speed as a proxy for the speed of other operations. However, additional models can be trained if a database supports operator specialization.

Similar to the models for encoded size, we use random forest regression for fixed width data and a combination of random forest regression and linear regression for variable width data. Memory speed models use the predicted encoded size of the slice as an input instead of sample statistics. In general, we would expect larger blocks to take more time to process for a given encoding. Additional inputs from the slice statistics are useful in correcting bad samples that lead to poor predictions about encoded size.

4.2.3 Storage Speed

The storage speed model type predicts how long it takes to operate on an uncached slice given the predicted memory speed and the predicted encoded size. If a slice is located on a low bandwidth storage device, it is beneficial to choose an encoding with a high compression ratio. However, decoding speed becomes more relevant on newer storage technologies like SSD and NVMe since the cost of retrieving entire slices can be on the order of milliseconds. With the prevalence of network block storage like Amazon EBS, it is usually sufficient to train models for a single storage backend.

We assume that all encodings for a given data type share the same access pattern and can therefore use the same model. The reason we chose to differentiate data types is because decoding variable width values can result in non-sequential reads due to the appendix section. To model a storage device, we use constrained linear regression with the Huber loss function to fit latency and throughput. We indirectly measure how long it takes to read data from storage by using the time difference between cached and uncached operations on the same slice. By incorporating the Huber loss function, we are able to reduce the impact of outliers, which show up frequently when timing accesses to a storage device.

4.3 Training

All models are built and trained using algorithms found in Scikit-learn [20] and SciPy [29]. Additional data processing is done using NumPy [11] and Pandas [30]. A Python process communicates with a compiled binary that contains all implemented encodings using JSON. To perform an experiment that will be used as part of the training data, the Python process first writes out a slice to a temporary file. It then sends metadata to the compiled binary, which will load the slice, test all encodings, and send back the resulting encoding properties. To improve training speed, experiments on encoded size are performed in parallel. However, memory speed and storage speed experiments still need to be executed sequentially to obtain accurate measurements.

One of the main challenges in training these models is obtaining enough slices to experiment on. From empirical evaluation, we need on the order of one thousand slices (or 1 billion rows) per data type. Obtaining this much data from existing datasets is difficult. Although a single column can contain many slices, all of the values are usually drawn from the same underlying distribution. This means that we can only get one meaningful input per column. Furthermore, it is not efficient for users of the database system to download all of the training data for each deployment. To solve this problem, we developed a way to train the models using only synthetic data that is randomly generated on the target machine. Not only does this eliminate the need

to download datasets, it also improves the performance of the models through better exploration of the input space.

4.3.1 Synthetic Data

For fixed width data, we draw from three compound distributions to obtain synthetic data. There are many other distributions which could be useful, but we have found these ones to be simple and effective. Note that the values obtained from these distributions are scaled and shifted in post-processing (see Section 4.3.2). Therefore, the mean and range of these distributions are fully determined in a later step.

Skewed Normal. The skewed normal distribution is useful for modeling a variety of real-world data. We choose a uniformly random variance and skew to better explore the input space of different encodings.

Discrete Uniform. The discrete uniform distribution is obtained by first choosing a cardinality from the log uniform distribution (covering 1 to the block size). The cardinality is then used to generate a set of distinct values, again chosen uniformly (from 0 to 1, although the exact range does not matter). Finally, a slice is generated by randomly sampling those distinct values. One observation is that the skewed normal distribution tends to generate data with high cardinality. This distribution allows the cardinality to be fixed before generating the values. The cardinality is chosen from the log uniform distribution instead of the uniform distribution to favor exploring slices with few distinct values.

Random Run. The random run distribution builds a slice by repeatedly generating runs of random lengths. Each run value is chosen uniformly (from 0 to 1) while each length is chosen from the log uniform distribution to make shorter runs more likely. Some real-world data with low cardinality tend to follow this distribution even when the column is unsorted.

For variable width data, we generate slices by first choosing a cardinality from the log uniform distribution. The cardinality is used to generate a vocabulary set consisting of random ASCII characters. The vocabulary set will have a mean length which is chosen uniformly at random. Finally, the slice is generated by repeatedly

sampling from the vocabulary with replacement. We capped the maximum mean length of the vocabulary set to 128 characters in order to bound training time. This is sufficient for the linear models to obtain good predictions on input slices with much higher mean lengths.

4.3.2 Post-processing

There are additional post-processing steps that need to be performed on a generated slice. For fixed width data, the slice is scaled and shifted according to the bounds of the data type. With 50% probability, null values are inserted into the slice at random locations. In SageDB, null integers are stored as the maximum value of their type, whereas null strings are equivalent to empty strings. With 50% probability, the slice is also sorted after null insertion.

We have found that there is no need to have explicit inputs for the number of null values in a slice or whether the slice is sorted. The slice statistics indirectly provide some of the same information without requiring additional overhead. However, these post-processing steps do improve the overall performance of the models since they allow the training data to be more representative of real-world inputs.

4.4 Selection

The selection process works by gathering the predicted values for all slices and then using an objective to determine the best encoding for each slice. For simplicity, we assume in this section that the memory model corresponds to in-memory scan speed and the storage model corresponds to from-storage scan speed. A more complex objective may require the use of additional memory and storage models, but the overall structure of selection remains the same.

The pseudocode for encoding selection can be seen in Listing 4-1. For each slice and encoding, all three models (size, memory, and storage) are evaluated using the appropriate inputs. The predictions for each slice and encoding are stored until all slices have been analyzed. At this point, the user-defined objective is called on the

```
def selection(slices, encodings, objective):
    """Returns the optimal encoding for each slice."""

    predictions = []

    for slice in slices:
        predictions.append({})

        for encoding in encodings:
            # Get the encoded size of a 1% contiguous sample.
            encoded_sample_size = get_encoded_size(sample(slice))

            # Get all slice statistics. Depends on the data type.
            slice_statistics = get_slice_statistics(slice)

            # Get all three models for the encoding.
            # The models may depend on properties of the slice.
            size_model = get_size_model(slice, encoding)
            memory_model = get_memory_model(slice, encoding)
            storage_model = get_storage_model(slice, encoding)

            # Predict the encoded size of the slice.
            size_inputs = [encoded_sample_size] + slice_statistics
            p_size = evaluate(size_model, size_inputs)

            # Predict the in-memory scan speed of the slice.
            memory_inputs = [p_size] + slice_statistics
            p_memory = evaluate(memory_model, memory_inputs)

            # Predict the from-storage scan speed of the slice.
            storage_inputs = [p_size, p_memory]
            p_storage = evaluate(storage_model, storage_inputs)

            # Add predicted values for encoding.
            predictions[-1][encoding] = \
                (p_size, p_memory, p_storage)

    # The objective chooses one encoding per list entry.
    return objective(predictions)
```

Listing 4-1: Pseudocode for encoding selection with an arbitrary objective.

predictions to determine the best encoding for each slice. It is not strictly necessary to evaluate the objective at the end. For simple objectives that depend only on the predictions for an individual slice (e.g., minimize encoded size), the objective can be evaluated per-slice. This is more efficient when writing out an encoded slice because the data for a single slice can generally be cached.

4.5 Integration

We integrated the learned encoding pipeline into SageDB. Users can apply automatic encoding selection for a column group by not specifying the encoding explicitly when creating a projection. At the moment, only column groups containing a single column are eligible for automatic encoding selection. Model training must run once prior to the start of the system for encoding selection to work. In our implementation, the training process only takes around 30 minutes for all model types on the target machine (without pre-training size models).

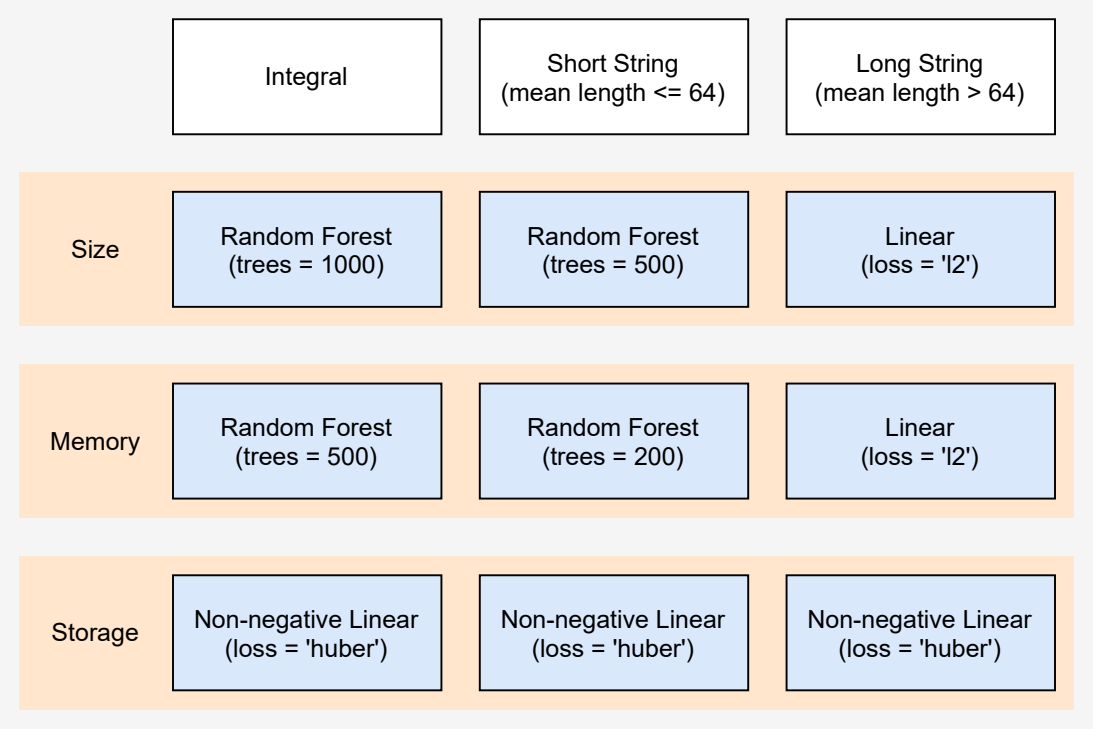

Figure 4-2: Summary of the specific model implementations used for each data type and model type.

The specific models used during training and evaluation are summarized in Figure 4-2. Random forest regression rely on Scikit-learn's `RandomForestRegressor` with the specified number of trees. Linear regression with L2 loss is trained using Scikit-learn's `LinearRegression` implementation. Non-negative linear regression is not fully support in Scikit-learn. Therefore, we implemented our own regressor using SciPy's non-linear `optimize.curve_fit` function with Huber loss. Although linear regression works, restricting the coefficients to be non-negative gives better results since storage latency and throughput are both non-negative. For strings, the mean length is extracted from the slice statistics to determine which set of models to use.

For simplicity, we implemented a two pass approach for determining and writing out encoded blocks. In the first pass, SageDB collect statistics about all slices that it will write out in the second pass. Sample statistics are obtained by taking a contiguous sample from each slice once it has been materialized in memory. All statistics are then passed to a Python process that hosts the trained models. Figure 4-3 shows an example of this interaction.

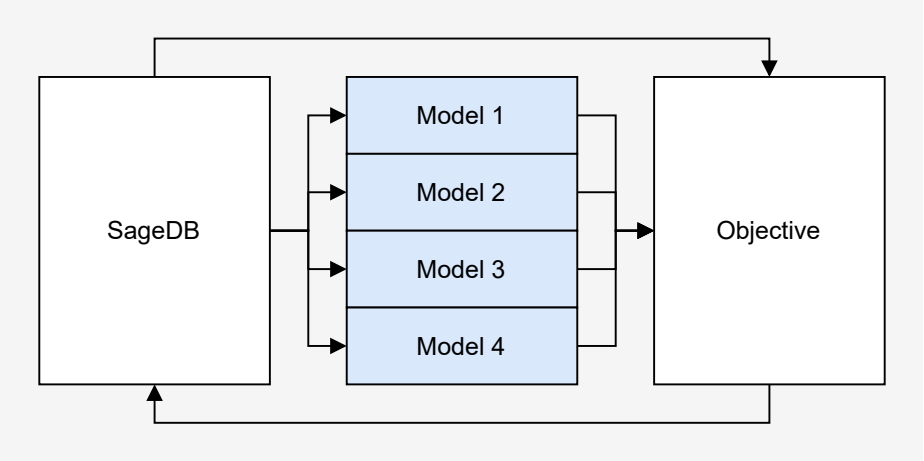

Figure 4-3: Interaction between four encoding models and SageDB.

SageDB also provides an objective that should be optimized. The Python process will evaluate all of the encoding models and feed the outputs into the objective to determine the optimal encoding for each slice. Those selections are then returned to SageDB, which will perform a second pass through the slices to apply the selected encodings and write them to storage.

There is some overhead associated with this approach. In particular, data has to be read from storage twice because the entire projection generally does not fit in main memory. An improved implementation could try to determine the optimal encoding for a given slice as soon as statistics are collected. However, this is not always possible, since certain objectives like minimizing query speed given an overall storage constraint require information about all slices before making a decision.

4.6 Related Work

There is limited work on using machine learning to predict the best combination of encodings for a given criteria. A project known as shrynk takes column-based data, computes certain features about the data, and uses those features in a classification model that then tries to predict the encodings with the best compressed size, read time, and write time [27]. The work demonstrated that it was possible to apply machine learning in the context of encoding selection.

A few existing database systems support some form of automatic encoding selection. Amazon Redshift and Vertica use sampling to suggest column encodings [5, 28] while SingleStore uses heuristics based on column statistics [4]. However, these encoding selection techniques are not instance-optimized and do not support different objectives. As we will see in later sections, differences in hardware and selection criteria can greatly impact which encodings are chosen.

Other approaches to improving database performance have been centered around automated database tuning, where different configuration parameters exposed by the underlying database are adjusted using heuristics or reinforcement learning [33, 34]. There is also extensive work in column advisors that automatically determine the best partitions, column clusters, indexes, and materialized views [23, 35]. Notably, the system presented in [23] tries to select the optimal column encodings, but does so using heuristics and a fixed cost model.

Chapter 5

Discussion

In this section, we discuss some of the trade-offs that exist in the current design of the system and the reasoning behind certain decisions.

5.1 Derivation of Encoded Size

It is possible to derive the exact encoded size for some encoding implementations given the sample statistics. For example, dictionary encoding only depends on the cardinality while frame of reference encoding only depends on the range. However, we chose not to manually derive the encoded size for any of the encodings to demonstrate the flexibility of encoding selection. There should be no need to hand tune the models even when new encoding types are added.

Another design would be to ignore the encoded size models altogether and simply encode each block in memory to determine the encoded size. We did consider this option, but found that some encodings take a fairly substantial amount of time to run and would significantly increase the write overhead.

5.2 Comparison with Cost Model

Learned encoding selection does not provide any bounds on its worst case selection performance. We found this to be a reasonable trade-off, since choosing the wrong en-

coding is usually not catastrophic. In the worst case, query performance will degrade but the results will still be correct. Many existing systems rely on a cost model along with heuristics to choose the best encodings. Those heuristics also have various edge cases that can cause them to fail. For example, a heuristic of choosing run length encoding when the cardinality is small would cause the encoded data to take up much more space compare to not using any encodings at all if the whole block consisted of alternating values.

5.3 Workload Requirements

One of the assumptions about the learned encoding technique is that the workload is read-heavy and that a moderate increase in overhead when writing out blocks is acceptable. This is generally the case for analytical workloads but not transactional workloads. In particular, updating existing rows in a table incurs heavy performance penalties due to the fact that entire blocks have to be re-encoded or re-organized. Inserts are easier to handle, since only the last block needs to be updated. The block can remain in raw form until the block size is reached. At this point, the block can be encoded and persisted to the appropriate location in the storage hierarchy.

Chapter 6

Evaluation

We evaluated our implementation of encodings and encoding selection in SageDB. All experiments are performed on a system with an AWS c5d.2xlarge instance with a network-attached gp2 EBS that achieves a throughput of 250 MiB/s. The hardware used here is comparable to that of small to medium database deployments on cloud providers. Larger deployments usually involve additional storage layers like network-attached SSDs and S3 for cold storage, but these setups are not examined in this thesis. We first evaluate the encoding selection process independently from SageDB. Next, we present results from the end-to-end performance of SageDB with and without encodings. Lastly, we look at the impact of operators on encoded data in SageDB.

6.1 Encoding Selection

In this section, we evaluate the performance of encoding selection on three datasets: denormalized Stack Overflow, denormalized TPC-H at scale factor 1, and denormalized Bitcoin Transactions. The Stack Overflow dataset consists of posts made by users on various Stack Exchange sites until 2019. It was retrieved from [26] and contains 25 columns with roughly 12 million rows. The TPC-H dataset contains 68 columns with 7 million rows generated using [36]. The Bitcoin dataset was retrieved from Google Cloud Public Datasets [9] and contains 12 columns with 10 million rows. In total, they contain around 850 slices split between integral and string types.

6.1.1 Comparing Against Ground Truth

In Figure 6-1, we show the encoding and the number of slices for which the encoding is optimal under the given objective. Here, Memory Speed refers to the in-memory scan speed of the slice and Storage Speed refers to the from-storage scan speed of the slice. Note that we do not distinguish between different datasets and columns in the plot. Although small fixed width data types like `char(1)` can be treated as integral, we consider them to be strings for simplicity throughout this evaluation section.

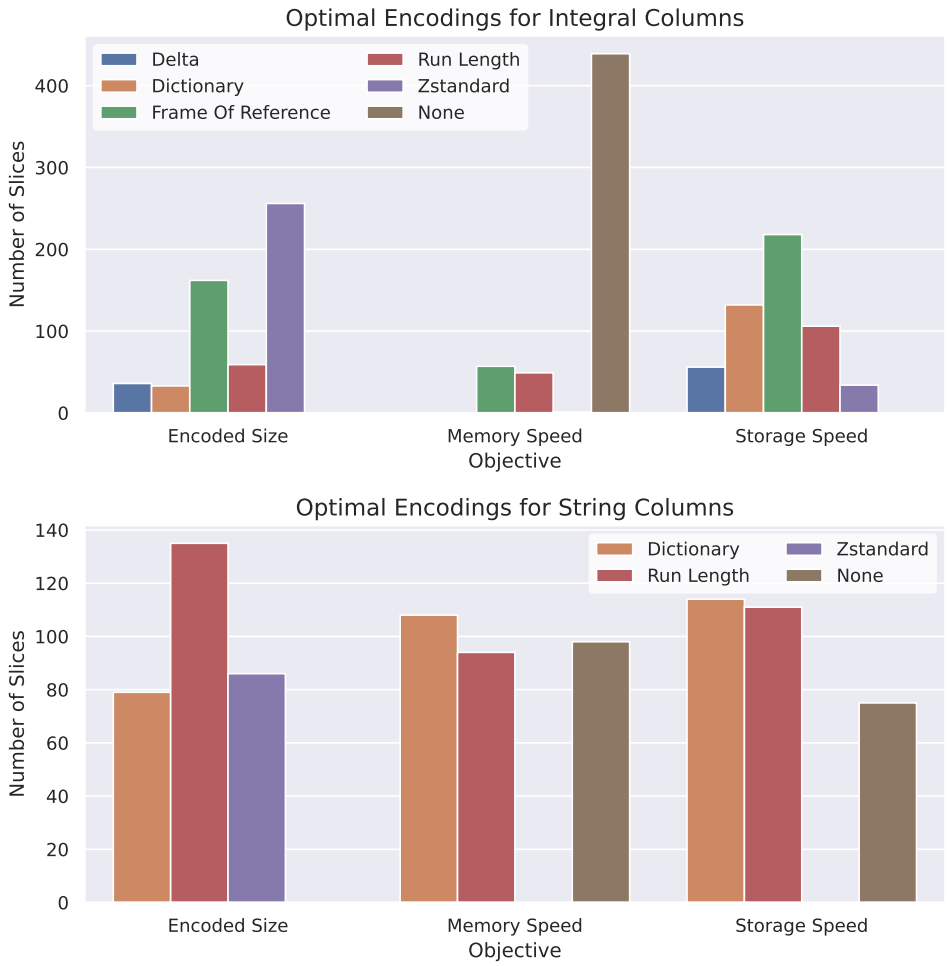

Figure 6-1: Number of slices for each encoding, data type, and objective.

We see that although Zstandard achieves great compression ratios, it is not the best encoding for more than 50% of the slices when optimizing for encoded size. Furthermore, it is rarely chosen when optimizing for scan speed because the decompres-

sion overhead is too high. This hints at the fact that purely optimizing for encoded size does not necessarily lead to improved query performance.

Another interesting point is that the optimal choice of encodings changes greatly depending on the objective. In the integral case, using no encodings at all is almost always the best when optimizing for in-memory scan speed, but never the best when optimizing for from-storage scan speed. This demonstrates the importance of instance-optimized encoding selection. When the underlying storage device is very fast, we would expect an encoding selection distribution similar to that of Memory Speed. However, when the storage device is very slow, we would expect a distribution closer to that of Encoded Size, since decoding time is negligible compared to the time it takes to read the encoded data.

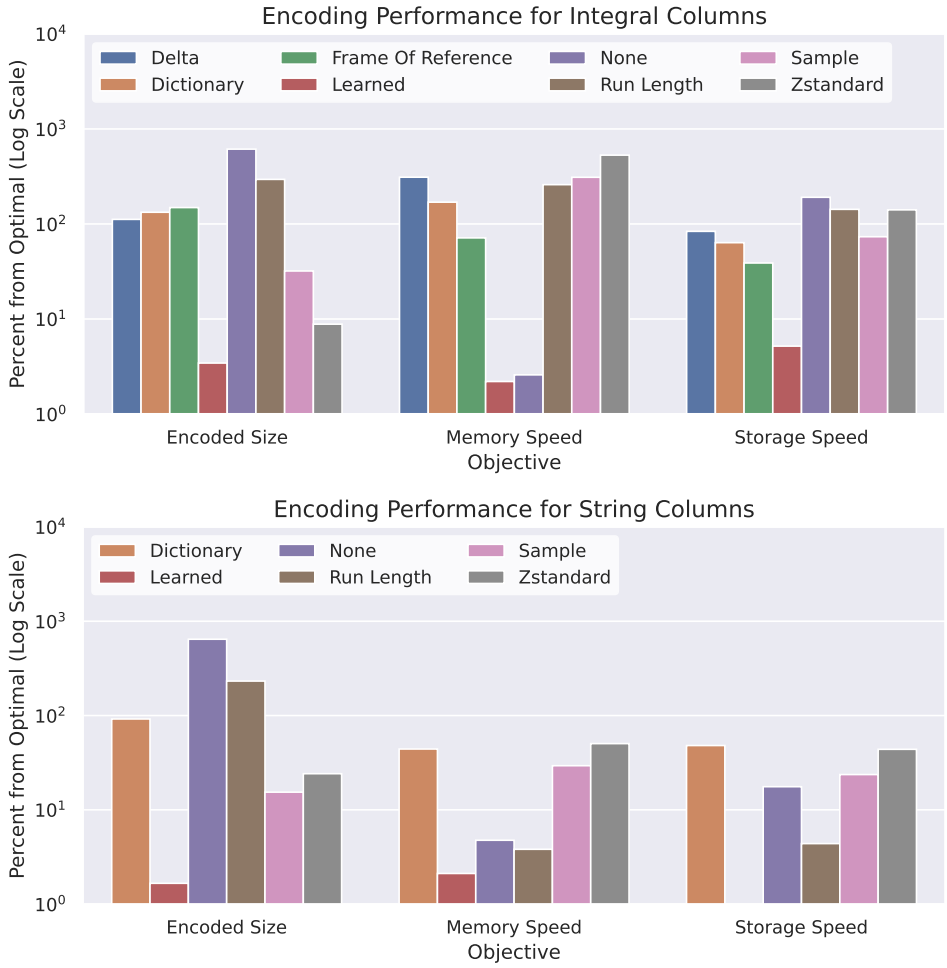

Figure 6-2: Performance of different encoding strategies against optimal.

We now look at the optimality of learned encoding selection compared to other encoding selection strategies. Figure 6-2 shows the percent from optimal for each strategy. The optimal encoding for each slice is determined by brute-forcing all encodings for each slice and objective. The percent deviation is computed relative to the sum of the corresponding encoding property (either encoded size, in-memory scan time, or from-storage scan time) for all slices. The Learned strategy corresponds to using automatic encoding selection on each slice. The Sample strategy picks the encoding that has the smallest sample encoded size (a 1% contiguous chunk) for each slice regardless of the objective. The remaining strategies correspond to using the specified encoding for all slices, similar to a default encoding for the data type.

Learned encoding selection is always within 10% of the optimal across all objectives and data types. In addition, it consistently outperforms other strategies. We would expect Sample to perform relatively well in terms of encoded size. However, the strategy of always choosing Zstandard is better for integral data. No single encoding works well across all three objectives because most encodings represent some trade-off between encoded size and decompression speed. In some cases, using a fixed default encoding can lead to poor performance when considering scan speed. For example, systems that use a general compression algorithm like Zstandard or LZ4 as the default encoding for columns might be better off not using any encoding at all for certain workloads that optimize for query latency.

	Integral			**String**		
	Size	**Memory**	**Storage**	**Size**	**Memory**	**Storage**
1st	423	467	409	257	160	212
2nd	97	50	98	38	109	86
3rd	21	23	28	5	30	1
4th	2	6	7	0	1	1
5th	3	0	4	-	-	-
6th	0	0	0	-	-	-

Table 6-1: Number of encodings of a given rank chosen by automatic selection for each slice.

Last, we analyze the performance of encoding selection on a more granular level. As before, the optimal encoding for each slice and objective is brute-forced. For each

slice, the encodings are ranked according to how well they perform on the objective, with the best as 1st. We count the number of times automatic selection chooses a given encoding rank over all slices. The results are presented in Table 6-1. Note that string columns only have 4 supported encodings, so the last two rows are blank.

We see that learned encoding selection chooses the best or second best encoding on most slices. However, there are some slices for which learned encoding performs poorly and chooses the worst possible encoding. There are a few factors that could contribute to poor selections. For one, performance timing is subject to a fair amount of variance. It is possible that there will be outliers for in-memory and from-storage scan speeds. The learned models may also need additional training input. Random forest regression does not perform well if it encounters edge cases that it has not been trained on. Due to randomly generated training slices, the performance of the models can differ depending on how similar the generated slices are to the evaluation dataset.

6.1.2 Ablations and Feature Importance

Our encoding selection technique relies on both slice statistics and sample statistics (sample encoded size). Here, we compare the accuracy of models using the full feature set (Sample + Slice) against ablated models that use only sample statistics (Sample Only) or slice statistics (Slice Only). Note that the experiment is only performed for integral encodings because string data types always rely on slice statistics to determine which model type (random forest versus linear) to use.

Figure 6-3 shows the performance of ablated models on five encodings against the vanilla models. There is no need to use a model for predicting the size of raw slices, since they can be determined from the data type and number of rows. We rely on symmetric mean absolute percentage error (SMAPE) to compare the predicted sizes with the actual sizes for each slice. Percentage error makes more sense than absolute error in this context due to the large size range of encoded blocks.

We see that the vanilla models outperform the ablated models in all cases. Slice statistics are particularly important for dictionary and frame of reference encodings. Dictionary encoding depends on cardinality and frame of reference encoding depends

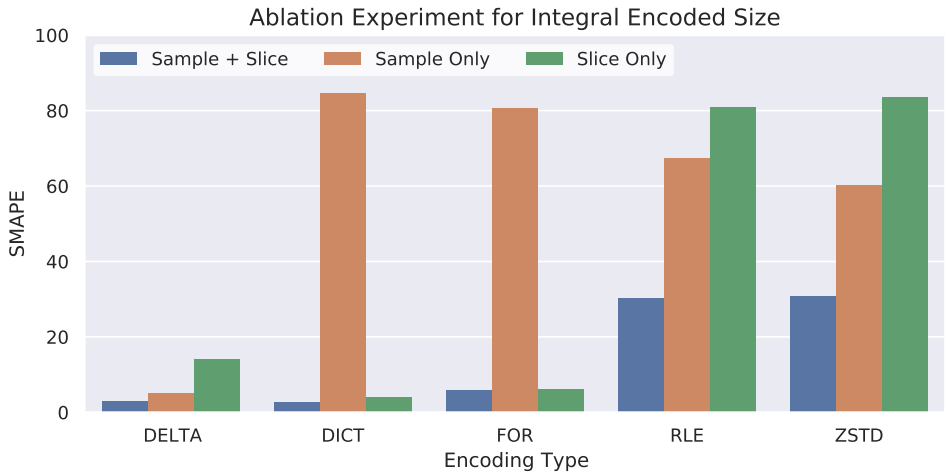

Figure 6-3: Ablation experiment on five encodings for integral data.

on domain size. However, both metrics are hard to estimate from a sample. The reverse is true for a general compression scheme like Zstandard. The sample encoded size is a better predictor of the overall encoded size compared to the slice statistics, which do not capture enough meaningful information on their own.

One benefit of using random forest regression is that it is easy to numerically compute the importance of each input feature. For each model and feature, we obtain the Gini importance using Scikit-learn's implementation after training the models. The results are shown in Figure 6-4. Note that we only performed this experiment for integral data because strings use different models depending on the average length.

We see that encoded sample size is important to most encodings with the exception of frame of reference. However, all features are useful for at least one encoding type. We know from the design of encodings that dictionary and frame of reference should depend heavily on the number of distinct values and the range respectively. This is confirmed by the Gini importance.

6.1.3 Convergence and Inference Time

Figure 6-5 shows the prediction accuracy of the models with an increasing number of training slices. Recall that each slice contains 1 million rows. We again use SMAPE

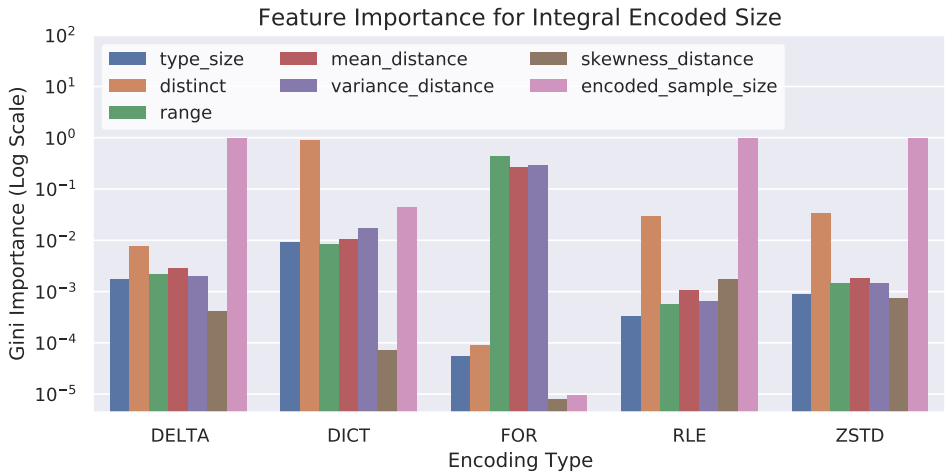

Figure 6-4: Gini importance of different features for each integral encoding type.

to measure the difference between the predicted sizes and actual sizes. However, we average the SMAPE values for all supported encodings of a data type so that each one can be represented using a single line.

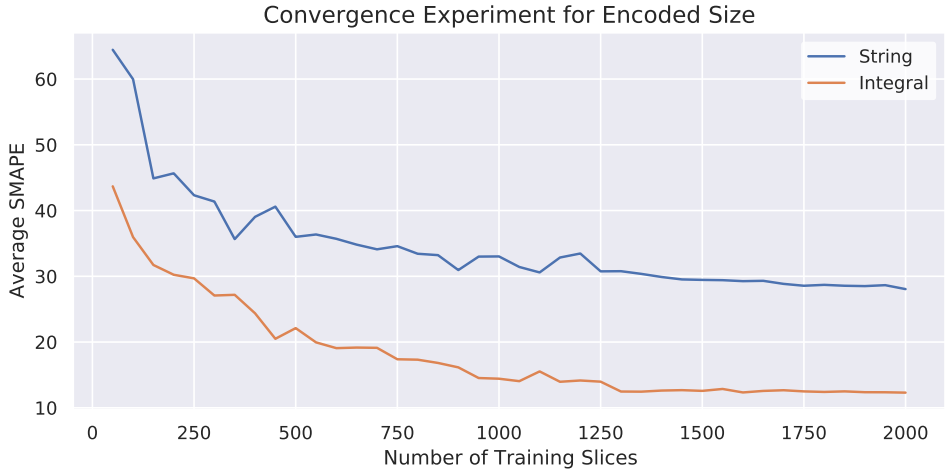

Figure 6-5: Prediction accuracy with an increasing number of training slices.

We see that for both integral and string data types, the models take around 1,500 slices to converge. In practice, we use around 2,000 slices for encoded size to ensure that edge cases for different encodings have a high likelihood of appearing as training inputs. Less blocks are necessary for in-memory scan speed and from-storage scan speed (around 500 and 10 respectively). Much of the time for training comes from generating the slices and performing experiments. The actual training time for all

string and integral models once statistics and encoding attributes have been gathered is less than 30 seconds.

Inference involves drawing slice and sample statistics from the data and feeding them to our different models. The most expensive step of the process is applying the different encodings to the samples. However, even this overhead is fairly small compared to the cost of reading from and writing to storage. For more information, see Table 6-2 in the next section. Once all statistics have been collected, the time it takes to evaluate the models and objective is fairly negligible. The optimal encodings for all 846 slices used in the testing data set can be determined in around 10 seconds.

6.2 Encodings in SageDB

In this section, we evaluate the performance of encoding selection in SageDB on the denormalized Stack Overflow and denormalized TPC-H datasets. There are 13 query templates for Stack Overflow and 8 query templates for TPC-H. Each query template is instantiated with around 25 parameter combinations to test different selectivities. Query latency is measured for the cold cache scenario (the file system cache is cleared before executing each query). The warm cache scenario is less interesting, since using no encoding is almost always optimal. Some systems take advantage of this by caching decoded blocks in memory, but SageDB does not yet support this feature. Note that the initial compilation overhead associated with each query template is not measured.

Query latency results are shown in Figure 6-6. For each query template, we compute the total time taken for all instantiations. The default encoding selection technique corresponds to using no encoding for all columns. We compare two learned selection techniques against the default — one that optimizes for the smallest encoded size (top) and one that optimizes for the lowest query latency (bottom). Values lower than 1 represent improvement over the default selection technique. The bars are sorted so that the query template with the greatest improvement comes first.

We see that optimizing for encoded size does not improve the query latency. In fact, there are significant regressions on around half of the query templates. Mean-

Figure 6-6: Performance of different encoding against optimal for latency.

while, learned selection that optimizes for query latency does indeed show significant improvement on more than half of the queries. There are some regressions, but they are all within a few percent of the default. One explanation for these regressions is that learned encoding selection only considers sequential scan speed. In the volcano iterator model, encodings that support random access perform better when the selectivity is low. However, encodings like delta have to decode values sequentially, thereby incurring performance penalties that are not accounted for in the training process. This can be fixed by incorporating random access time as a separate memory model and using an objective that integrates with the query plan.

Next, we look at the encoded sizes of each column within the two datasets under the same two learned encoding selection techniques. For each column, we compute

the total number of bytes occupied on storage over all slices. As before, the bars are sorted so that the column with the greatest improvement comes first. The results are presented in Figure 6-7.

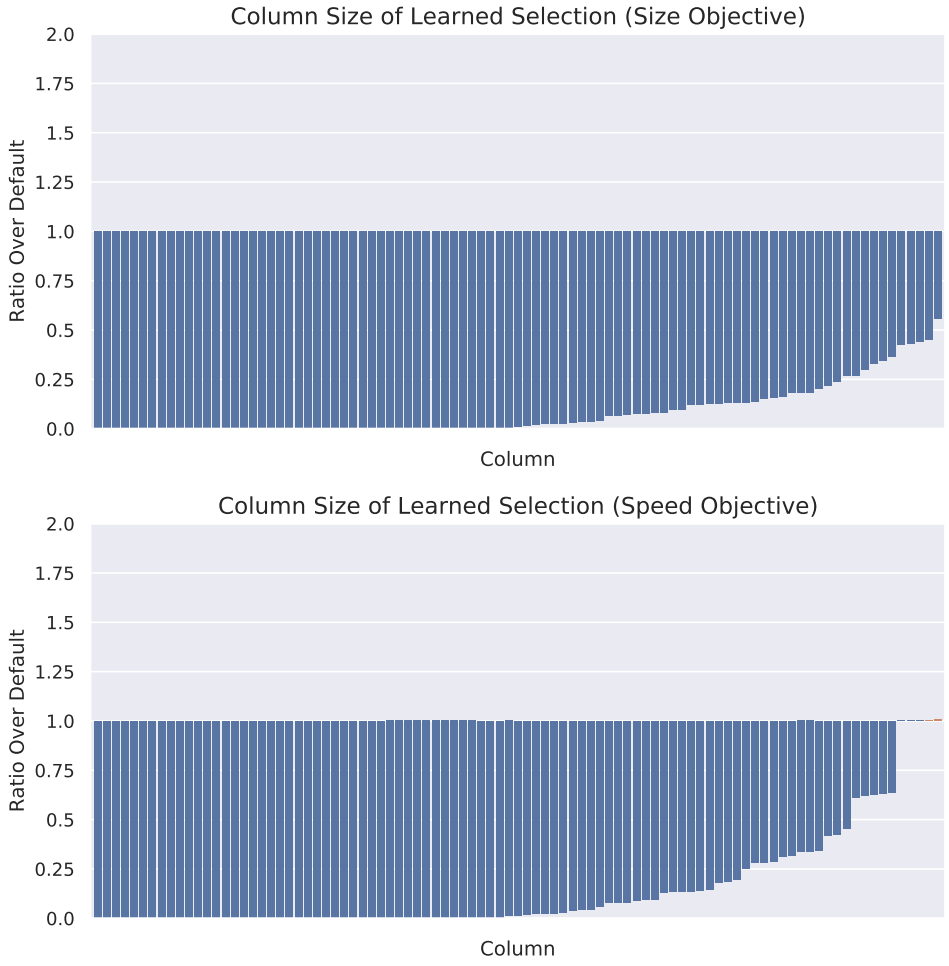

Figure 6-7: Performance of different encoding against optimal for size.

Both learned selection techniques significantly outperform the default encoding selection technique on almost all columns. This is expected, since general compression algorithms like Zstandard can reduce the size of most columns. Optimizing for the size objective does outperform the speed objective on around half of the columns. There are no regressions when using the size objective, whereas there is a slight regression on one column when using the speed objective. When two encodings are closely tied for a slice and objective, it is generally okay to choose either one. This is the behavior that we observe with learned encoding selection. In cases where one encoding is clearly

the best, learned encoding selection almost always picks that encoding.

	Stack Overflow			TPC-H		
	Default	**L-Size**	**L-Speed**	**Default**	**L-Size**	**L-Speed**
Table Size	22.4 GB	7.6 GB	21.5 GB	8.4 GB	0.52 GB	1.21 GB
Read Time	169 s	188 s	118 s	41.0 s	39.5 s	38.1 s
Write Time	215 s	356 s	313 s	111 s	128 s	114 s

Table 6-2: Summary statistics for two workloads with different encoding selection objectives compared to the default.

Summary statistics for each of the two workloads are shown in Table 6-2. Note that L-Size and L-Speed refer to learned encoding selection that optimizes for the size and speed objectives respectively. The table size metric refers to the total size of all encoded slices on storage. The read time metric refers to the total time it takes to run all query template instantiations for the table. The write time metric refers to the amount of time it takes to create the projection containing all of the columns for the table, including the time it takes to gather statistics and evaluate all models. Compilation time is omitted for both read time and write time metrics.

We see that learned encoding selection can significantly reduce space utilization when optimizing for size. However, this can come at the cost of degraded query performance for some workloads. Meanwhile, optimizing for speed not only reduces query latency, but also the table size. On TPC-H, L-Speed uses 86% less space and improves query latency by 7% compared to default. On Stack Overflow, L-Speed reduces space usage by 4% while improving query latency by 30%. Optimizing for query latency should be appropriate for most workloads and used as a default objective.

In terms of write time, learned encoding selection imposes some overhead, but it is always within a factor of 2 compared to creating the projection without any encodings. On Stack Overflow, a large string column causes the system to use swap space excessively during statistics collection and encoding, which could explain the large difference in write time. For TPC-H, we see that there is a negligible amount of overhead when optimizing for speed. A more optimized implementation of encoding selection that writes out blocks in a single pass could reduce the amount of time it

takes to create a projection compared to not using any encodings since there is less data to persist on storage.

6.3 Operator Specialization in SageDB

In this section, we examine the performance of operator specialization (operators on encoded data) in SageDB. We use three queries constructed from the Stack Overflow dataset. Q1 performs a max aggregation on a frame of reference encoded column with a filter on another frame of reference encoded column. Q2 performs a grouped aggregation on a dictionary encoded column to find the maximum of a second column that is not encoded. Q3 performs a sum aggregation over a run length encoded column with a filter on a dictionary encoded column.

Query	**Iterator Latency**	**Specialized Latency**
Q1	103 ms	23.4 ms
Q2	876 ms	120 ms
Q3	322 ms	15.3 ms

Table 6-3: Warm cache query latency comparison between iterator model and vectorized processing model with specialization.

The latencies for each query under two different execution models are reported in Table 6-3. The iterator model corresponds to the volcano iterator model currently used by SageDB while the specialized model corresponds to the vectorized processing model with operator specialization for different encodings (see Section 3.4). All latencies are for the warm cache scenario (blocks are already cached in memory) since operator specialization does not affect the time it takes to read blocks from storage and thus would not improve performance significantly for the cold cache case.

We see that operator specialization along with the new execution model significantly improves in-memory query performance. The amount of improvement for individual queries depend greatly on the selectivity and how useful the specialized operator is. For example, the max operator specialization for frame of reference encoded data does not yield as much performance benefits as the group operator specialization for dictionary encoded data when compared against the existing iterator

model. Nonetheless, we see significant improvements even on Q1 since the target system supports SIMD instructions that the compiled code is able to leverage.

Operator specialization can improve the performance of some queries by an order of magnitude when the blocks are cached in memory. This is particularly useful when there are some blocks that are frequently accessed by queries. An interesting comparison that has not been studied here is the impact of caching decoded blocks. We know experimentally that using no encoding is usually the best for query latency when the blocks are already in memory. However, operator specialization adds an additional layer of complication since decoding blocks lead to the loss of certain information that can be used to speed up operators (e.g., decoding a dictionary encoded block that will be used in a group by clause).

Chapter 7

Future Work

We list a few different extensions of this work that address some of the current limitations of the system and improve the effectiveness of learned encoding selection.

Additional Models. At the moment, we are only able to predict the size and sequential scan speed of a slice. Although this works well in improving query latency, a full-featured database system also performs random access into slices and supports various indexes on columns. Training and evaluating the performance of additional models can help learned encoding selection find better solutions for a given objective and workload.

Support for Multi-Column Blocks. Automatic encoding selection can be extended to support blocks with more than one column. In addition to column-level statistics (slice statistics), we would need information about the correlation between columns. Additional investigation is necessary to determine the best input features and how to efficiently train the models using synthetic data.

Operator Specialization with Bao. Operator specialization represents an alternative query execution path. The current system is not able to determine whether operator specialization would be beneficial for a given query. It should be possible to integrate operator specialization with a global query optimization system like Bao to generate better query plans that account for the cost of using specialized operators.

Evaluation on Different Hardware. We have presented evaluation results for learned encoding selection on a particular system. It would be good to evaluate

the technique on a variety of hardware configurations to demonstrate how learned encoding selection can adapt.

Workload Awareness. Learned encoding selection does not take into account the expected workload. At the moment, it assumes that columns are accessed with equal frequency by all queries. This assumption does not hold for a majority of workloads. When a column is accessed very infrequently, learned encoding selection should favor high compression, since this will reduce storage costs without sacrificing query performance. However, this type of optimization would require knowledge of the workload ahead of time and an objective that takes the workload into account.

Integration with Layout Optimizer. The predicted encoding attributes for a slice can be used by a layout optimizer to determine its placement in the storage hierarchy (memory, NVMe, cold storage, etc.). For example, frequently accessed blocks might be placed in memory with no encoding, whereas rarely accessed blocks might be placed in cold storage with maximum compression. The layout optimizer can also use learned encoding selection to determine the best encoding for a slice by incorporating predicted attributes in a global optimization problem that considers aspects of the system outside of storage costs and query latency.

Chapter 8

Conclusion

In this thesis, we presented a general and efficient approach to encoding selection that works in an instance-optimized manner. We implemented this technique in SageDB along with 5 different encoding schemes. Experimental results have shown that the encoding selection approach detailed here performs well on end-to-end queries and achieves very close to the optimal solution when considering individual data slices. Furthermore, the selection process can incorporate arbitrary user-defined objectives that are evaluated at runtime without the need to re-train any of the models. For a small increase in write time when first creating a projection along with a modest amount of training on the target system, automatic encoding selection can significantly improve query performance while reducing storage utilization.

In addition, we demonstrated the benefits of operators on encoded data by implementing a version of vectorized query processing in SageDB. By leveraging SIMD instructions along with specialized operators, we were able to speed up some types of queries by more than an order of magnitude for the warm cache scenario.

Appendix A

Additional Listings

This section contains various additional listings that provide more detail about the topics discussed in other sections. All assembly is generated using x86-64 clang 10.0.0 with the compiler flags `-std=c++17 -O3 -march=cascadelake`.

```
struct A {
  int f() { return 1; }
};

struct B {
  int f() { return 2; }
};

int do_visit(std::variant<A, B>& variant) {
  return std::visit(
      [](auto& variant) { return variant.f(); }, variant);
}
```

Listing A-1: Example of the visitor pattern on a variant with two types.

```
do_visit(std::variant<A, B>&):
  push    rax
  mov     rsi, rdi
  movzx   eax, byte ptr [rdi + 1]
  cmp     rax, 255
  mov     rcx, -1
  cmovne  rcx, rax
  mov     rdi, rsp
  call    qword ptr [8*rcx + vtable]
  pop     rcx
  ret
f1:
  mov     eax, 1
  ret
f2:
  mov     eax, 2
  ret
vtable:
  .quad   f1
  .quad   f2
```

Listing A-2: Compiled assembly for the code presented in Listing A-1.

```
struct A {
  int f() { return 1; }
};

struct B {
  int f() { return 2; }
};

int do_visit_optimized(std::variant<A, B>& variant) {
  return optimized_visit(
      [](auto& variant) { return variant.f(); }, variant);
}
```

Listing A-3: Same visitor pattern example as Listing A-1, but using the optimized visit method.

```
do_visit_optimized(std::variant<A, B>&):
  cmp     byte ptr [rdi + 1], 1
  mov     eax, 2
  sbb     eax, 0
  ret
```

Listing A-4: Compiled assembly for the code presented in Listing A-3.

```cpp
// Finds the maximum value in the data considering only locations
// where the mask is true.
template <class T>
T max(const T* data, const uint8_t* mask, size_t n) {
  // Keep track of the maximum value.
  auto max_value = std::numeric_limits<T>::min();

  // Go through all values in the data.
  for (size_t i = 0; i < n; ++i) {
    // Get the unsigned value.
    auto u_value = std::make_unsigned_t<T>(data[i]);

    // Specialized based on whether type is signed.
    if constexpr (!std::is_signed_v<T>) {
      // For unsigned types, set the value to 0 when the mask is
      // false. This effectively ignores the value.
      u_value &= ~(std::make_unsigned_t<T>(mask[i]) - 1U);
    } else {
      // For signed types, we need to do a bit more to set the
      // value to the minimum for the type when the mask is true,
      // but do nothing otherwise.
      auto ignore = std::make_unsigned_t<T>(
          std::numeric_limits<T>::min());
      u_value = ignore ^ ((u_value ^ ignore) & -(!!mask[i]));
    }

    // Update the maximum value.
    auto value = *((T*)(&u_value));
    max_value = value > max_value ? value : max_value;
  }

  return max_value;
}
```

Listing A-5: Subroutine for finding the maximum value of an array with a mask that supports vectorization.

Bibliography

[1] Eli Bendersky. The cost of dynamic (virtual calls) vs. static (crtp) dispatch in c++. https://eli.thegreenplace.net/2013/12/05/the-cost-of-dynamic-virtual-calls-vs-static-crtp-dispatch-in-c. Accessed: 2021-02-20.

[2] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. Lea: A learned encoding advisor for column stores. In *Proceedings of the Fourth International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '21, New York, NY, USA, 2021. Association for Computing Machinery.

[3] Yann Collet and Chip Turner. Smaller and faster data compression with zstandard. https://engineering.fb.com/2016/08/31/core-data/smaller-and-faster-data-compression-with-zstandard/. Accessed: 2021-02-19.

[4] Columnstore - singlestore documentation. https://docs.singlestore.com/v6.7/concepts/columnstore/#how-the-memsql-columnstore-works. Accessed: 2021-02-17.

[5] Compression encodings - amazon redshift. https://docs.aws.amazon.com/redshift/latest/dg/c_Compression_encodings.html. Accessed: 2021-02-17.

[6] Data compression | monetdb. https://www.monetdb.org/Documentation/Guide/Compression. Accessed: 2021-02-17.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[8] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. *SIGARCH Comput. Archit. News*, 39(3):365–376, June 2011.

[9] Google cloud public datasets. https://cloud.google.com/public-datasets. Accessed: 2021-03-27.

[10] G. Graefe. Volcano— an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[11] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.

[12] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, September 2018.

[13] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. Sagedb: A learned database system. *Conference on Innovative Data Systems Research (CIDR) 2019*, 2019.

[14] D. Lemire and L. Boytsov. Decoding billions of integers per second through vectorization. *Software: Practice and Experience*, 45(1):1–29, May 2013.

[15] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Learning to steer query optimizers, 2020.

[16] Michal Nowakiewicz, Eric Boutin, Eric Hanson, Robert Walzer, and Akash Katipally. Bipie: Fast selection and aggregation on encoded data using operator specialization. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1447–1459, New York, NY, USA, 2018. Association for Computing Machinery.

[17] Jakob Stoklund Olesen. Variable-length integers. `https://github.com/stoklund/varint`. Accessed: 2021-02-18.

[18] Albert Opher, Alex Chou, Andrew Onda, and Krishna Sounderrajan. The rise of the data economy: Driving value through internet of things data monetization. Technical report, IBM, 2016.

[19] Michael Park. Variant visitation v2. `https://mpark.github.io/programming/2019/01/22/variant-visitation-v2/`. Accessed: 2021-02-20.

[20] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[21] Protocol buffers: Encodings. `https://developers.google.com/protocol-buffers/docs/encoding`. Accessed: 2021-02-18.

[22] Alexander Rasin and Stan Zdonik. An automatic physical design tool for clustered column-stores. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 203–214, New York, NY, USA, 2013. Association for Computing Machinery.

[23] Alexander Rasin and Stan Zdonik. An automatic physical design tool for clustered column-stores. In *Proceedings of the 16th International Conference on Extending Database Technology*, EDBT '13, page 203–214, New York, NY, USA, 2013. Association for Computing Machinery.

[24] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10, page 34–40, New York, NY, USA, 2010. Association for Computing Machinery.

[25] Juliusz Sompolski, Marcin Zukowski, and Peter Boncz. Vectorization vs. compilation in query execution. In *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, DaMoN '11, page 33–40, New York, NY, USA, 2011. Association for Computing Machinery.

[26] Stack exchange data dump. `https://archive.org/details/stackexchange`. Accessed: 2021-03-27.

[27] Pascal van Kooten. shrynk - using machine learning to learn how to compress. `https://vks.ai/2019-12-05-shrynk-using-machine-learning-to-learn-how-to-compress`. Accessed: 2021-02-17.

[28] Vertica analytics platform - encoding types. `https://www.vertica.com/docs/9.2.x/HTML/Content/Authoring/SQLReferenceManual/Statements/encoding-type.htm`. Accessed: 2021-05-05.

[29] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[30] Wes McKinney. Data Structures for Statistical Computing in Python. In Stéfan van der Walt and Jarrod Millman, editors, *Proceedings of the 9th Python in Science Conference*, pages 56 – 61, 2010.

[31] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. Druid: A real-time analytical data store. In *Proceedings of the*

2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14, page 157–168, New York, NY, USA, 2014. Association for Computing Machinery.

[32] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per-Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-tree: Learning data layouts for big data analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 193–208, New York, NY, USA, 2020. Association for Computing Machinery.

[33] Bohan Zhang, Dana Van Aken, Justin Wang, Tao Dai, Shuli Jiang, Jacky Lao, Siyuan Sheng, Andrew Pavlo, and Geoffrey J. Gordon. A demonstration of the ottertune automatic database management system tuning service. *Proc. VLDB Endow.*, 11(12):1910–1913, August 2018.

[34] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, Minwei Ran, and Zekang Li. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, page 415–432, New York, NY, USA, 2019. Association for Computing Machinery.

[35] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, page 1087–1097. VLDB Endowment, 2004.

[36] Andreas Zimmerer. Denormalized tpc-h. `https://github.com/Jibbow/denormalized-tpch`. Accessed: 2021-03-27.