Study of bitwise operations on non-scarce attribute based data structures in PostgreSQL

MARCEL ESCHMANN
Study of bitwise operations on non-scarce attribute based data structures in PostgreSQL

Marcel Eschmann
Abstract

This report investigates the viability of bitwise operations on non-scarce attribute based data structures in PostgreSQL. For applications where computation can’t be avoided, it most probably can be optimized. In an attempt of bringing the computation closer to hardware and the underlying data, operations directly on the database system are explored, taking inspiration from the research field of comparative genomics. With the case-study of an online job platform in mind, where possible matchings between candidate and job descriptions are calculated by a matching engine, a binary encoding is proposed and the computational components identified. The ultimate goal was to evaluate the scalability of the bitwise strategy with respect to the current matching engine. Through an iterative approach, this report conducts quantitative experiments on the presented components. Most notably, an implementation of the population count in the form of a C extension was introduced. It was found, that even for large sequence lengths, the operation is highly efficient. Among the chosen algorithms Lookup Table, Hamming Weight, Intrinsic functions and Unrolled Inline Assembly, the 64 bit intrinsic function displayed the best performance. Benchmarks determined, that the proposed bitwise approach is an excellent strategy for the outlined use-case. Despite the tradeoff of additional complexity in the encoding and decoding of data, the speedup is so significant, that the targeted user base of 100000 can easily be managed and allows for the deprecation of caching mechanisms.

Keywords: matching engine; scaling; binary encoding; bitwise; attribute data; PostgreSQL; population count
Sammanfattning


Nyckelord: matchningsmotor; skalning; binär kodning; bitwise; attributdata; PostgreSQL; population count
# Table of contents

Abstract ii

Sammanfattning iii

1 Introduction 1
  1.1 Background .................................................. 1
  1.2 Problem ..................................................... 2
  1.3 Purpose ....................................................... 2
  1.4 Goal ........................................................... 2
  1.5 Methodology .................................................. 2
  1.6 Delimitations ................................................ 2
  1.7 Social benefits, ethics and sustainable development ............... 3
  1.8 Disposition ................................................... 3

2 Background 4
  2.1 Genome comparison .......................................... 5
  2.2 SQL database systems ....................................... 6
  2.3 Varlena ......................................................... 6
  2.4 Hamming weight ............................................. 6
  2.5 Great-circle distance ........................................ 6
  2.6 Binary encoding ............................................. 7

3 Methodology 8
  3.1 Data collection ............................................... 8
  3.2 Measurements ............................................... 8

4 Test bench 10
  4.1 Configuration ............................................... 10
  4.2 PHP clock .................................................... 10
  4.3 Database clock ............................................... 11

5 Experimental Evaluation 12
  5.1 Initial benchmarks ........................................... 12
  5.2 Attribute encoding ......................................... 13
    5.2.1 Grouped boolean ....................................... 13
    5.2.2 Grouped scalar ......................................... 14
    5.2.3 At least one of boolean group ......................... 14
  5.3 Additional attributes ....................................... 15
    5.3.1 Range .................................................... 15
    5.3.2 Geolocation ............................................. 15
5.4 Database requirements ........................................... 15
5.5 MySQL constraints .................................................. 15
5.6 Population count ..................................................... 16
  5.6.1 PostgreSQL extension ............................................ 16
  5.6.2 Implementation .................................................. 17
  5.6.3 POPCNT instruction support .................................... 21
  5.6.4 Benchmarks ....................................................... 22
5.7 Geolocation .......................................................... 25
  5.7.1 Implementation .................................................... 26
  5.7.2 Benchmarks ....................................................... 26
5.8 Data set generation ................................................... 26
  5.8.1 Seed extraction .................................................... 26
  5.8.2 Properties ........................................................ 28
5.9 Matching engine benchmarks ....................................... 28

6 Conclusion ............................................................. 30
  6.1 Discussion .......................................................... 30

References ............................................................... 31

A Appendix ............................................................... 34
  A.1 Source code ......................................................... 34
1 Introduction

Binary encoding and bit-wise operations do not enjoy the same exposure like more recent approaches utilizing machine learning do. Within the problem area of recommendation and personalization, companies such as Google rely on vast data sets to predict user’s preferences [1]. Deep neural networks keep visitor’s attention on the platform by always displaying information that is relevant and engaging [2]. For emerging services, where data is limited and highly specific, this approach may be less viable. On the other hand, the computational complexity often increases swiftly with a growing user base. If the computation can not be avoided, it most probably can be optimized. Binary encoding and bit-wise operations present an opportunity to bring said computation closer to hardware and therefore increase performance considerably [3]. In bioinformatics, the field of sequence alignment continuously studies different tactics of computation on encoded data [4]. Genome databases are vast and individual entries contain hidden evolutionary information in an encoded format [5]. This complex research area has been widely studied and serves as an excellent inspiration for this report.

1.1 Background

Uska [6] is a job-platform within the health care sector. The platform currently approaches 10’000 users and promises to bring employers and candidates together through intelligent matching. For that purpose, both parties register and create a matching profile containing attributes like spoken languages and strengths in areas specific to health care. The matching engine determines a score for a variety of weighted factors between candidate and job.

Database indexes allow to quickly locate data without visiting each row at the expense of memory. In some use-cases however, it is necessary to execute calculations on all data points available. In the case of Uska, a low matching score is just as valuable as a higher one, because the platform also promises to support candidates in growth and ultimately pursuing a path leading to their dream job. Hence for every job, the matching score needs to be calculated for every candidate on the platform. For platforms with framework abstractions, parsing and execution of these calculations requires a significant amount of computing power and ultimately time. With a growing user base on a service relying on these calculations, waiting times can get out of hand. At that point, it might be valuable to explore different approaches, where calculations are moved back from the backend to the database as much as possible, by encoding the data differently.

Binary encoding of attributes could enable performant comparison operations and reduction of the database size by avoiding the requirement of separate attribute tables. Uska already uses a bitwise strategy with success, however can not utilize it to its full potential due to technical limitations. Hence the platform serves as an excellent use-case
1. INTRODUCTION

to explore the viability of such an approach, negating its current technical limitations.

1.2 Problem

Uska [6] as a platform is rapidly growing and the current matching engine is not prepared to withstand the increasing computational requirements. Aiming to become the market leader for jobs within the health sector of Sweden, the system should be able to accommodate the total of approximately 400'000 individuals pursuing a profession within that area. Hence, a scalable approach to the matching engine needs to be identified and verified. As a milestone, the scaling to 100'000 users is targeted.

1.3 Purpose

The purpose of this report is to investigate the performance implications of bitwise operations on large data sets for non-scarce attribute-based data structures. Uska [6] serves as a suitable case-study, requiring quick identification of the best matches within its data set. The experimental evaluation as part of this report aims to give answers to its limits and opportunities, but also to be a data point for future researchers and engineers interested in utilizing binary encoding and bitwise operations for their systems.

1.4 Goal

The goal of this report is to study the performance implications of encoding attribute data in binary form and performing bitwise operations directly on the database, similar to genome comparisons. To that end, the specific use-case of a job-platform in the health care sector is explored. Raw performance measurements of the online platform serves as a baseline to be improved upon. Ultimately, the goal is the compilation of a recommendation for the usage of the proposed approach at the example of the case-study.

1.5 Methodology

This report proposes a binary encoding for attribute-based data in the context of the online platform Uska and its application for bitwise operations on large non-scarce data sets. Through an iterative approach, the required moving parts are introduced and quantitatively evaluated. After the definition of all components, a suitable data set was generated for the final benchmarks, in order to answer the goal of compiling a recommendation for the case-study.

1.6 Delimitations

The specifics of the matching engine and user data sets can not be disclosed in this report, due to non disclosure agreements with Skillio AB, the company behind the Uska platform. This means that the baseline and final performance results can not
be reproduced directly. Special care is given to conserve the problem complexity and is motivated in the report.

1.7 Social benefits, ethics and sustainable development

While the experiments were based on real user data, the original data was first obfuscated and anonymized, such that no identification of individuals is possible. It served the sole purpose of generating a data set with realistic computational complexity, which is crucial in order to evaluate the effectiveness of the proposed approach. Uska as a platform is GDPR (General Data Protection Regulation) [7] compliant and the security of users data is of utmost importance.

Due to the nature of this report, tackling computational complexity, neither social benefit- nor sustainable development aspects other than the reduction of power consumption due to reduced computation are mentionable.

1.8 Disposition

Section two presents the background of this report. After introducing the online platform Uska serving as the case-study, relevant theory required for the remainder of the report is motivated. The methodology used for the data collection and measurements is stated in section 3. It is the basis for the following two sections. All measurement experiments were conducted on the same hardware, which is stated in section 4.

Section 5 serves as the main part of the thesis and iteratively introduces the moving parts for the proposed approach of binary encoding and bitwise operations. Based on experiments, the impact of individual components is studied before combining them all for the final benchmarks. The following and final section 6 summarizes the results and provides specific answers to the goals outlined. A discussion of the results concludes the report.
2 Background

The matching engine of the job-platform Uska within the health sector serves as the case-study of this report. Its implementation has gone through multiple revisions since its original release in 2016. Initially, the implementation consisted in the core of iceberg database queries [8] using a high number of joins with dynamic attributes. This approach was working for the prototype state of the project, but quickly turned out too slow, once the project got traction. Consequently, caching mechanisms were introduced.

The current implementation of the matching engine leans on genome comparison in bioinformatics, meaning that attributes are encoded into bit-sequences. This allows both for quick but coarse and slower but accurate matching, such that the full matching is only executed on pairings that make sense. Matching profiles are stored in a MySQL database due to legacy constraints. Logic was moved from the database to the back-end, which enabled easier extension, maintenance and testability. In the mean time, the number of attributes and user base have grown significantly, requiring once again a new revision of the system.

The matching engine in its current state is facing multiple challenges. Due to limitations of MySQL, bitwise operations as well as the BIT-type column are restricted to a length of 64. Hence, the sequence representation of the matching profile had to be split into multiple columns. Attributes like languages may also have a numerical range attached, rather than a binary value, in order to express knowledge levels. Those fields had to be split even more, such that a BIT-type column was used for existence, and a varchar type for holding scalar values. Consequently, additional computing is introduced and outsourced from the database to the application server. With growing numbers of users and complexity of the product, the matching engine will hit its limit. Other areas that are important to the product, such as geolocation, are affected as well. For example, MySQL does not offer great-circle distance calculations for coordinates and this feature had to be implemented.

This increase in complexity is noticeable to the user. When an employer changes attributes of a job, the updated list of matching candidates is expected almost instantly. However, this requires the comparison to every candidate in the system, resulting in delays and 100% cpu usage on the server. Growth of the user base increases this delay and cpu usage constantly. Uska aims to become the market leader for jobs within the health sector in Sweden. It is approximated, that 400’000 individuals pursue jobs within that area in Sweden. For that reason and the possibility to explore different areas, scaling of the system is required.
2. BACKGROUND

2.1 Genome comparison

As already mentioned, genome comparison from the field of bioinformatics has inspired the implementation of the matching engine. This section introduces the terminology. Comparative genomics [9] refers to the research of species comparison and is constantly evolving as an area. Ultimately, the goal is to understand evolution. A genome contains all genetic information of an organism and is a string of nucleotids. Each nucleotide denotes a letter or attribute of DNA. To that end, genome sequences are compared in their encoded format in order to infer information regarding their most likely shared history. Whole-genome comparison is a fundamental computational problem [10].

For this approach, the number of similar or identical amino acid residues are explored and compared to the total number thereof. Here we can draw the first parallels to what a matching engine seeks to accomplish. The number of similar or identical attributes of a matching profile can be compared to the total number of attributes existent within the sequence. This case is simpler, since the structure of each genome is predefined and does not contain unknown gaps. As an important research topic, sequence analysis has attracted a wide range of research effort. Known methods take different approaches, such as alignment, word frequency and geometric representation.

Local sequence alignment is one approach, where sequences have gaps and matrix back-tracking is used. Smith and Waterman [11] described a general algorithm to identify the optimal local sequence alignment score. It requires $O(n^3)$ time and $O(n^2)$ memory for a substitution score matrix and a gap penalty function [12]. Since the structure of matching profiles is known and our sequences are gapless, this approach is not suitable.

Full sequence alignment such as the Smith-Waterman algorithm are very time consuming. BLAST [13] takes a different route, using a heuristic algorithm without calculating the optimal alignment. Since genome databases usually are vast, the emphasis on speed is crucial and has allowed for BLAST to become one of the most widely used programs. Ideally, the proposed approach in this report should allow for efficient computation of all matchings, since the underlying data does not reach the scale of genome databases. Hence heuristics are not relevant yet, could however be very interesting for future research.

To summarize, only the ideas of encoding and comparison are extracted. Sequence alignment is not necessary as of yet, however it might become interesting once other business areas are explored and inter-professional comparisons emerge as a feature.
2. BACKGROUND

2.2 SQL database systems

SQL stands for Structured Query Language and is used for the handling of structured data in a database system. It is a domain-specific language and exists in several flavours for the specific database implementations. For this report, MySQL and PostgreSQL are of particular interest. Both are open source object-relational database management systems that were initially released in the 90s. Due to their long life span, each system has developed their own specialities and opinions, however because both rely on SQL, generally cover approximately the same features. Such specifics are exactly the research field of this report. The investigated case-study currently relies on MySQL, while PostgreSQL offers a potential performance gain.

2.3 Varlena

Varlena stands for variable length array and is a key data structure used by PostgreSQL [14]. The elegant data structure simply consists of an integer length and a starting pointer of byte length. Several data types such as text, bytea, bit, json, array, and more are based on verlena. It can hold any data of up to 1GB of size and is one of the reasons for the extensibility of the database system. For example, when writing an extension for a binary data type, access to the varlena data structure allows for easy manipulation. It plays an integral role for the implementation of the Hamming Weight in the context of this report.

2.4 Hamming weight

The Hamming Weight, or population count, is a variant of the Hamming Distance, first introduced in 1950 by Richard Hamming [15]. For two binary sequences of equal length, the Hamming Distance computes the number of positions, for which the bit values differ. As a variant of that algorithm, the Hamming Weight computes the number of bits not equal to zero. The implementation of the Hamming Weight has been widely studied and the best known software-based algorithm for non-scarce sequences consists of only 12 processor operations [16]. Modern processors include a processor instruction for this sole purpose.

2.5 Great-circle distance

The great-circle distance is the shortest distance between two points along the surface of an approximated sphere. Modelling Earth as a sphere, the distance between two coordinates can be computed to sufficient accuracy [17] for most applications. PostgreSQL includes the earthdistance module, it is just not enabled by default. For this module, the earth is modelled as a perfect sphere. The popular PostGIS project alternatively offers more accurate calculations. MySQL on the other hand does not offer comparable functionality and often projects are required to write their own extensions. In the case
of Uska, a software abstraction generates the corresponding query for two geographic coordinate points, utilizing available trigonometric functions. PostgreSQL promises a speedup for the calculation of distances.

2.6 Binary encoding

Binary encoding is the process of taking regular data and representing it in a binary way, meaning a sequence of ones and zeros. The predominant example is a boolean value, true or false, that is simply described by a zero (false) or one (true). It is an interesting approach, because it allows for the usage of operations very close to hardware. Arithmetic Logic Units (ALUs) of mainstream consumer processors operate on binary sequences after all. Operations are of bitwise and arithmetic nature. Consequently, the computation is very efficient. In contrast, the encoding requires additional engineering effort for parsing, rendering it a trade-off.
3 Methodology

This report proposes a binary encoding for attribute data and conducts an experimental evaluation of bitwise operations on it. Since the goal is to compile a recommendation for a specific use-case and scale, raw performance measurements are first gathered. Afterwards, the individual computational components for the new encoding are introduced and evaluated through quantitative inductive methodology. This allows for the detection of potential bottlenecks or speedups in comparison to the baseline performance. In order to reach significant results for the case-study of an online job-platform within the health-care sector, the quantitative experiments need to be conducted on realistic data sets. Therefore, anonymised statistical distributions of real production data will be extracted and used as a seed of new artificial data sets.

3.1 Data collection

Privacy is important and relevant to this report, since the case-study of a job-platform brings sensitive data along. However, for the final experiments, where the new approach is to be evaluated, such data is very valuable. Specifically, it is crucial to ensure, that the problem complexity is not reduced or increased. It would render the comparison to the baseline performance distorted. In the case of attribute data, only the distribution within the database is of importance for the experiments. Specific labels of columns on the other hand are irrelevant. In order to both ensure privacy and not to reveal company secrets, all data gathered from the online platform is obfuscated before usage. That is to ensure that no individuals are identifiable and that the particularities of the matching engine remain secret.

The components necessary for the proposed encoding to work, will be introduced and studied individually before commencing to the final benchmarks. In order to avoid overfitting, general test data specific to the individual component will be used. That is also to ensure that the experiments are reproducible and of value for future research.

3.2 Measurements

The virtualization of the cloud offers shelter to many web applications. Server resources are partially shared among instances and the scheduling of the operating system is not within the control of the user. Furthermore, processor usage is not constant, since it relies on the number of users currently using the service. When serving requests, these external influences may affect the performance and lead to spikes, outliers from the expected performance. For the measurement of time, it is important to use a method that is accurate and offers the necessary resolution. In order to mitigate the influence of such factors, all measurements should be executed multiple times. Specifically, 50 trials will be used for all experiments. Since the aforementioned spikes can be intrusive and
should not affect the performance studies, the median was used for the presentation of the results, rather than an average. This alleviates the effects of various measurement anomalies. The steady-state performance [18] is more valuable to the report, since the web server is generally running for extended periods of time. Effects such as the cold start, one-time costs on startup, would affect the average in a more substantial manner. The clocks used will be introduced in the next chapter.
4 Test bench

The nature of this report required quantitative experiments conducted on a consistent test bench. In order to avoid additional variables such as the remaining computational load of the system, a virtual machine with guaranteed exclusive access was set up on a cloud hosting provider. The short lifespan of the virtual machines allows for comfortable and low-cost experiments. Digitalocean [19] offers a variety of server configurations. The category optimized droplets promises "compute optimized" virtual machines with dedicated Intel hyper-threads. From personal experience the platform is quick on setup, flexible and offers modern hardware. All droplets are equipped with SSDs and the processors are of recent generations.

4.1 Configuration

The following environment was used for all experiments.

- **Operating system**: Ubuntu 18.04
- **CPU** Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz, 4 Hyper-Threads
- **C Compiler version**:
  - gcc Ubuntu 7.3.0-16ubuntu3
  - clang version 6.0.0-1ubuntu2
- **Python version**: Python 3.6.5
- **PHP version**: PHP 7.2.5-0ubuntu0.18.04.1
- **PostgreSQL version**: PostgreSQL 10.3 (Ubuntu 10.3-1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 7.3.0-5ubuntu1) 7.3.0, 64-bit
- **MySQL server version**: 5.7.22-0ubuntu18.04.1 (Ubuntu)

Unfortunately, the specifications of the memory are hidden by the virtualization of the hosting. It is simply stated as 8GB of ram, without the possibility to read the speed and type from the shell. Intel’s product specifications page [20] lists the CPU as a 24 core and 48 threads processor with 33MB of L3 cache. This last level cache is shared by all cores of the processors. This means that the exclusive access is understandably only conditionally accurate.

4.2 PHP clock

This report revolves around the case-study of the Uska online job-platform, which is running on a PHP stack. For the retrieval of base line performance results, the PHP function *microtime* was wrapped around the corresponding code segments. It returns the current Unix timestamp with microseconds resolution.
4.3 Database clock

Both MySQL and PostgreSQL offer internal profilers. Using these profilers, we can avoid the problem of measuring computation outside of the database system. However, database queries invoke not only the execution of the actual query, but also wrapper processes. As such, empty queries were used to determine the minimum time elapsed for querying the two database systems. Figure 4.1 displays the histogram of 100000 empty queries.

A significantly higher empty query duration can be observed for PostgreSQL. Outliers take as long as 0.334 ms, whereas MySQL reached 0.188 ms as a maximum. The median time is 0.130 and 0.027 ms for PostgreSQL and MySQL, respectively. For low sample sizes, PostgreSQL queries should be assumed to take longer, compared to MySQL. This also confirms the argumentation in section 3.2, that the median is more applicable than the average. These spikes would affect the average too much.
5 Experimental Evaluation

This chapter first discusses the performance of the current online platform. After proposing the binary encoding, the individual components necessary are presented and evaluated iteratively, until ultimately all components are combined to perform the final measurements.

5.1 Initial benchmarks

The performance of the matching engine currently in use on the Uska online platform is provided as is, in order not to reveal company secrets (see 1.6). It serves as the baseline to be improved upon. Approaching 10000 users, the median number of candidates matched against a job description is 1029. Figure 5.1 shows the frequency plots of 50 trials, performing the current median matching computation. The maximum of 3742 candidates per job description can be seen in figure 5.2. Both experiments indicate that the first computation requires significantly more time, represented by the outliers marked orange. These data points can be ascribed to the cold start discussed in section 3.2.

![Figure 5.1: Frequency plot of 50 trials for the matching engine computation using the current median of 1029 candidates. The median runtime is 1.99 seconds, outliers are marked orange.](image-url)

In order to reach comparable numbers to future experiments, the runtime is limited to the matching engine computation. For a user within the service, additional parsing,
rendering and latency is added to these numbers. Contrary to the statement, that the matching should be calculated for every candidate on each job description, only a subset is currently computed, to avoid loosing users due to the waiting time.

Evidently, this approach is not suitable for computing the fully connected graph on-the-fly and not scalable. The following chapters introduce a strategy to potentially improve on these numbers, moving the computation from the backend closer to the database and hardware.

5.2 Attribute encoding

In order to take advantage of efficient bitwise operations close to hardware, the corresponding data needs to be encoded in an applicable binary format. The following sections propose an encoding for the attributes relevant to this report and build the basis for the forthcoming sections building up the final benchmarks.

5.2.1 Grouped boolean

The first attribute type is called \textit{grouped boolean}, where a set of attributes are simply checked for existence. Instead of adding one boolean column for each attribute to the database, a bit column is introduced, grouping together said attributes. Figure 5.2: Frequency plot of 50 trials for the matching engine computation using the current maximum of 3742 candidates. The median runtime is 6.66 seconds, outliers are marked orange.
5. EXPERIMENTAL EVALUATION

illustrates a possible scenario for 8 grouped attributes.

\[
\begin{align*}
0111'0011 & \quad & 0101'0010 & \rightarrow & \quad 3 \\
0101'1010 & & & & & \\
\end{align*}
\]  

(1)

The owning side (above) is compared to the opposing attribute group using a bitwise AND operation. The resulting bit sequence has 3 bits set to value 1. Hence for the required attributes of the owning side, 3 attributes are satisfied by the opposing side. This is significant, because an important aspect of the genome comparison approach is the comparison of the sum of available attributes to the total amount of attributes (see 2.1).

5.2.2 Grouped scalar

As an extension to the previous attribute type, each individual attribute now may have a numerical range attached to it. This is close to the common use-case of restaurant star ratings, where a value between 0 and 5 is chosen to describe the offering of an establishment. In this case, a number of attributes with a range are grouped together in a bit sequence. For the encoding and consistency, two constraints need to be defined. First, each attribute in the group is encoded as a scalar in binary form from the same side, either left or right side. Second, the range is constant. The rating is manifested by the amount of bits set to the value 1. Two attributes with a range of 4 from the right side (least significant) are encoded in figure 2, illustrating the computation.

\[
\begin{align*}
0000'0111 & \quad & 0001'0111 & \rightarrow & \quad 3 \\
0111'0111 & & & & & \\
\end{align*}
\]  

(2)

While the calculations are exactly the same as for the grouped boolean type, the difference in encoding enables additional querying capabilities. In the example, the upper bit sequence could be interpreted as the query, looking for all entities with at least rating 3 for the second attribute.

5.2.3 At least one of boolean group

In some cases, it might suffice to have one attribute of a set. However, following the approach above, to have multiple groups within the same column, would yield invalid (too high) results. If we look again at figure 1, for each section of 4 bits at least one attribute was satisfied. The result was 3, however in this case a value of 2 would be admired, since for each section at least one attribute was satisfied. The first section was counted twice. Unfortunately, this can not be overcome with bitwise operations over grouped sets of this attribute type. Either each group needs to be encoded in their own database column, or a new operation is required. Such operation could iterate over the bit sequence with a predefined group length and execute the bitwise calculations locally. For the individual column of each group, any value above 0 is accepted.
5. **Experimental Evaluation**

5.3 **Additional attributes**

There are other attributes, for which binary encoding does not offer potential speedups, however are relevant to the use-case of the Uska platform. In order to draw comparison between the initial benchmarks of the platform and the newly proposed strategy, two more attributes are introduced.

5.3.1 **Range**

The range attribute comprises of two integers, symbolizing a numerical range between the two. As a convention, the first integer needs to be smaller or equal to the other one. This allows for the comparison between two ranges for overlap.

5.3.2 **Geolocation**

Distance between a candidate and a potential job is a crucial attribute. Using the great-circle distance (see section 2.5) algorithm, the geographical distance between two coordinate points can be approximated.

5.4 **Database requirements**

To summarize, the following features are required by the database system to hold the attribute encoding introduced above.

- Bitwise AND operation
- Counting the number of set bits, known as popcount (population count) operation
- Range column types
- Spatial calculations on coordinates

These requirements will be discussed in the following sections thoroughly, ultimately being combined to generate the final benchmark. Since the range attribute does not have any equivalent MySQL implementation to compare to, it will not be discussed further.

5.5 **MySQL constraints**

MySQL exhibits distinct limitations with regards to bit-wise operations. Bit functions and operators require all arguments to be of type `BIGINT`, which is a 64-bit integer [21]. Any other input type will be implicitly cast to 64 bit integers before the operation and longer sequences will be truncated. Especially for the grouped scalar attribute encoding, this imposes limits on the amount of attributes in a single column significantly and renders true binary encoding impractical.
The upcoming release of MySQL 8 [22] promises to eliminate the 64bit limitations mentioned above. Column type support for bit functions and operators is extended to [VAR]BINARY/ [TINY— MEDIUM— LONG]BLOB. While the column values still will be casted, it is implemented in a way such that no bits are lost. Non-binary string arguments continue to have the limitations described for the previous versions. While the introduced encoding surely could be adapted for these binary data types, it would introduce additional encoding. Additionally, length of these types is determined by a number of bytes, such that arbitrary lengths are not supported.

PostgreSQL offers the bit(n) type without such limitations on its length and also includes support for range and spatial columns, missing in MySQL. Furthermore, since most database abstraction software in PHP offers drivers for both MySQL and PostgreSQL, a migration while costly, would not be as costly as a complete rewrite. Consequently, the remainder of this chapter will explore PostgreSQL’s capabilities for the outlined approach, while relating its performance to MySQL.

5.6 Population count

The population count operation returns the number of bits set to 1 for a given binary sequence. Surprisingly, PostgreSQL does not offer built-in functionality for this operation. However, due to the available data structure varlena (variable length array, see section 2.3), the functionality can be implemented through a C extension. This section is devoted to the proposed implementation and benchmarking in comparison to the version given in MySQL.

5.6.1 PostgreSQL extension

The computation of the population count for a given binary sequence has been thoroughly researched and therefore efficient algorithms are available and well-documented. Reducing the problem to only 12 processor instructions for 32 bit integers, the Hamming Weight algorithm (see section 2.4) is the most widely used implementation. A version for 64 bit integers is known as well. MySQL uses the 32 bit version for its implementation of the \(BIT\_COUN\)T function. The PostgreSQL data type bit(n) can be of arbitrary integer length \(2^{30} - 1\) bytes (1GB), to be precise) and is based on the varlena data structure, which is an 8 bit arbitrary length array. Utilizing the Hamming Weight algorithm therefore introduces additional alignment logic for the least significant bits for configurations which are not 32 bit aligned.

One different approach is precalculating a lookup table for all 8 bit integers. Instead of actually inspecting the sequence, the popcount is retrieved from the lookup table by integer cast and array lookup. Especially the aspect of caching renders this approach promising.
5. EXPERIMENTAL EVALUATION

The aim is to achieve an implementation of similar or better performance to the MySQL variant in PostgreSQL, accepting input lengths beyond the 64 bit limitation. For that purpose, the following three algorithms were implemented as a C extension:

- 32 bit Hamming Weight
- 64 bit Hamming Weight
- 8 bit lookup table

It would be possible to utilize the `bytea` data structure in PostgreSQL to reduce the storage requirements, similar to the BINARY and BLOB types of MySQL. Acting as a compressed form of the `bit(n)` data type, storage is saved in favour of readability. Additionally, bitwise operators are only supported for `bit(n)` and `varying bit(n)` in PostgreSQL and would have to be implemented. Since storage is not a concern at this point and bit length is always known, `bit(n)` was chosen.

### 5.6.2 Implementation

This section introduces the implementations of the 8 bit lookup table and 32 bit Hamming Weight algorithms. The 64 bit variant of the Hamming Weight follows analogously to the 32 bit implementation.

#### Lookup table

Under the premise, that a table of the population counts for all 8 bit numbers is available, this algorithm simply iterates over byte chunks of the data and sums up the local counts. This plays very well along with the underlying varlena data structure of the `bit(n)` type. It is strictly a byte length of the attached data and a byte pointer to the first data point.

The following code extract shows the PostgreSQL C extension implementation.

```c
/**
 * Bit count for every 8bit decimal number
 **/
static const int bitcount[256] = {
  0, 1, 1, ..., 7, 7, 8 // abbreviated
};

/**
 * Cache lookup algorithm for counting bits set.
 **/
Datum popcount(PG_FUNCTION_ARGS) {
  VarBit *a = PG_GETARG_VARBIT_P(0);
```
Since the underlying data is byte aligned, this approach works even for sequences where the length is not a multiple of 8. The remaining bits will be set to zero.

**Hamming Weight**

Moving on to the Hamming Weight algorithm, the implementation is not as simple. This time, the population count actually needs to be computed. Furthermore, the Hamming Weight algorithm expects 32 bit aligned inputs, whereas the underlying data structure only ensures byte alignment. For example, a sequence of length 22 will be aligned to 24 bits by the varlena data structure. While the remaining byte in my tests always was zeroed, it is undefined behaviour, depends on the processor and might contain random information. The implementation of the algorithm therefore requires special care for the last chunk of the sequence, should it not be 32 bit aligned.

An implementation as a C extension to PostgreSQL could look as follows. In similar fashion to the lookup table, the data body is iterated using pointer increments. An additional cast to 32 bit length ensures that the pointer dereferencing returns suitable input arguments for the Hamming Weight algorithm. Behaviour for non 32 bit aligned data remains undefined for now.

```c
/**
 * 32bit Hamming weight / popcount algorithm for counting bits set.
 * Requires additional aligning logic for the last 32bit trunk.
 **/
Datum popcount32(PG_FUNCTION_ARGS) {
    VarBit *a = PG_GETARG_VARBIT_P(0);
    unsigned char *byte_pointer = VARBITS(a);
    int length = VARBITBYTES(a);
    int count = 0;
    uint32_t *position = (uint32_t *) byte_pointer;
    for (; length >= 4; length -= 4) {
        count += bitcount[(int) *pointer++];
    }
    PG_RETURN_INT32(count);
}
```
The Hamming Weight algorithm itself has been thoroughly studied and as already mentioned can be reduced to 12 operations [16]. Through a divide-and-conquer strategy, it counts occurrences of 1-bits in each nibble in parallel, converts these sums to byte sums and finally computes the sum of sums through clever bitwise operations.

Now we have a complete algorithm for 32 bit aligned sequences. In order to support any arbitrary sequence length, the remaining bytes need to be treated separately. The data body of the Varlenda data structure is saved as byte chunks using the big-endian format. However, our test bench for example is running on an Intel x86 architecture using little-endian format. Dereferencing should therefore be avoided, since the variable will contain a different value depending on the CPU architecture. This was already the case when feeding the Hamming Weight algorithm. Luckily, while the numerical value changes, the number of 1-bits remains constant, leaving the algorithm unaffected.

Since our extension contains the logic for the Lookup table algorithm, we could simply reuse that logic to count the remaining bytes.
In order to offer a self-contained solution, two additional approaches will be introduced. The first one copies the remaining bytes to a local 32 bit variable to be used for the Hamming Weight algorithm. The C library function `memcpy` is used, copying \( n \) characters from source to destination. Even though the resulting integer may be interpreted differently according to the processor architecture, only the proper parts of the memory are accessed and used for the population count.

The third approach is a custom implementation of what the `memcpy` function achieves. Since the official function includes sub-processes to work properly in a variety of circumstances, we can simplify it even further for our very specific use-case. Keeping track on two pointers for the source and destination, byte by byte is copied to a local variable.

In order to select the best alignment approach among the three, the different versions will be compared in the following benchmark section. References to the full source code can be found in the appendix A.1.
5. EXPERIMENTAL EVALUATION

5.6.3 POPCNT instruction support

POPCNT is the x86 instruction that calculates the number of bits set and potentially could improve the performance by orders of magnitude for compilers and processors that support it, known as hardware acceleration. Any modern CPU should support the instruction and for example the C compiler GCC provides a built-in function __builtin_popcount [23] that detects hardware support, falling back to a software implementation if that’s not the case. These functions are generally known as intrinsic functions and are compiler specific. However, inspecting the generated assembly code reveals, that an assembly implementation of the Hamming Weight is used, even if the CPUID flag for the POPCNT instruction was present. In order to benefit from the instruction, the compiler needs to be informed to target a platform supporting the instruction explicitly. For x86_64 architectures, the flag -mpopcnt [24] will ensure that the instruction is used. It is supported by both Clang and GCC compilers. This intrinsic function and its 64 bit variant will be used as additional possible approaches for the following benchmarks. Implementations are omitted from the report due to their similarity to the Hamming Weight variants, function calls are simply replaced.

Unfortunately, the 64 bit assembly instruction popcntq suffers from a False Data Dependency issue, preventing it from surpassing the 32 bit variant in performance. The instruction has a 3 cycle latency, yet is pipelined such that it can execute each cycle [25]. Loop unrolling could circumvent the issue, by grouping together 4 popcnt instructions. Time for some inline assembly. First, the inline assembly was written in its own function, to avoid code duplication:

```c
static inline uint64_t popcntq(uint64_t val) {
    __asm__ ("popcnt %1, %0" :
    "=r" (val) :
    "0" (val));
    return val;
}
```

The inline keyword signals the compiler, that it might be worth it to substitute the function calls with the corresponding function body directly. This should ensure, that the instructions will be grouped together, circumventing the False Data Dependency. Afterwards, 256 bit chunks of the sequence are iterated over, until finally the remaining 64 bit chunks are treated individually:

```c
// ...  
// Unrolled popcntq to avoid False Data Dependency.  
for (; i < limit; i += 4) {
    count += popcntq(buffer[i]);
    count += popcntq(buffer[i+1]);
    count += popcntq(buffer[i+2]);
    count += popcntq(buffer[i+3]);
}
```
This strategy was used for a final implementation called `popcount256`. As was later discovered, the Clang compiler detected the loop within the 64 bit built-in function implementation and used exactly such an unrolled version:

```
## ...
popcntq (%rsi), %rcx
addl %ebx, %ecx
popcntq 8(%rsi), %rbx
addl %ecx, %ebx
popcntq 16(%rsi), %rcx
addl %ebx, %ecx
popcntq 24(%rsi), %rbx
addq $32, %rsi
addl %ecx, %ebx
addl $4, %edx
```

GCC takes a similar route. Consequently, the intrinsic implementation and the attempted 256 bit implementation should reach similar performance. Since the standard PostgreSQL distribution uses the GCC compiler, unless otherwise stated, all following benchmarks will be conducted using GCC compiled code.

### 5.6.4 Benchmarks

Both the Hamming Weight and table lookup algorithms utilize a fixed amount of instructions and are therefore not affected by the input distribution. For scarce sequences where only few positions are of value 1, the performance could be positively affected by introducing an additional check to skip the computation, should the current sequence chunk be of value zero. This would require two instructions, less than the 12 for the Hamming Weight, but only for zero-chunks. Since the data set available indicates the opposite, non-scarce sequences, this additional check was not introduced and therefore the actual numerical values of the test data set can be neglected. It is therefore sufficient to scale the bit length alignment.
To be able to include the MySQL `BIT_COUNT` performance for comparison, the first experiment was conducted on 64 bit aligned sequences. Since 64 is the least common multiple for the three algorithms, the additional alignment logic was avoided completely. For low sequence lengths such as 64, the algorithms should not produce significantly different results. Figure 5.3 presents the observed performance.

As expected, the algorithms achieve very similar performance. PostgreSQL takes the lead after overcoming the minimum required time seen from the observations in section 4.3. The benchmarks suggest linear time complexity when increasing the sample size. For the respectable size of 100000 samples, only around 12 ms were required. However, a length of 64 bits is not high enough to portray the performance difference among the outlined algorithms. Scaling the alignment size of the samples should reveal more interesting results.

Before doing that, the alignment logic needs to be addressed. To measure the impact of choosing one or the other approach, a database of 10 million records was generated, with a `bit(63)` type column. Each approach was implemented along the 32 bit Hamming Weight algorithm. The 63 bit length ensures, that 4 bytes need to be considered by the alignment logic. Figure 5.4 shows the 3 approaches discussed in the previous section.
compared to the lookup algorithm, which does not require additional alignment logic.

Figure 5.4: Speedup in percentage of the three alignment strategies compared to the lookup algorithm over a 63-bit aligned sample size of 10'000'000 records.

The results indicate, that the lookup table is the most efficient approach with only little overhead. However, with only 30.76 ms difference between the memcpy and lookup table approach over 10 million records (2.43% improvement), it can be argued that none of the approaches offer any significant gains over the others. In order to avoid dependencies between the algorithms, the lookup table approach was discarded. Since the custom copy approach only offers marginal improvements, the standard C library function was chosen for the final implementation.

Now, in order to get a better understanding of the proposed algorithms for PostgreSQL, the bit length will be scaled. Figure 5.5 displays the performance over a constant sample size of 500’000 while increasing the column length by the powers of 2. The increment strategy was chosen for its relation to common data storage alignments, since low level memory blocks often are of power of two sizes.

The 256 bit unrolled algorithm and 64 bit intrinsic approach match each other for best performance among the group. Linear time complexity can be observed, however around the column length of $2^{14}$, a significant increase in runtime occurs. After this bump, the trend continues to be linear, however the algorithms manifest in closer performance, reducing the previous spread. No particular explanation for this phenomenon could be found within the PostgreSQL documentation and source code. For the purpose
Figure 5.5: Log-log-plot of the six popcount algorithms over a constant sample size of 500000, while increasing the column length (bit alignment) by powers of two.

of this report, the matching engine will probably never reach such high values. Hence this phenomenon was not studied further, however could be crucial for other applications, requiring very long bit sequences.

The linear progression satisfies the need for further experiments for now. Despite of the marginal lead the 256 bit unrolled algorithm displays, the 64 bit intrinsic approach is chosen as the best alternative. Inline assembly usage renders the 256 bit unrolled version hardware specific, whereas the intrinsic versions are compiler specific and are therefore favourable. A final popcount version could be implemented, detecting support for the intrinsic function and falling back to the fully portable 64 bit Hamming Weight algorithm. For the purpose of this report, the PostgreSQL extension was elected to provide all algorithms to enable a conscious choice on runtime.

5.7 Geolocation

This section conducts a brief comparative study of the great-circle distance (see 2.5) computation between MySQL and PostgreSQL. The `earthdistance` module is used for PostgreSQL, while MySQL uses a custom implementation of the Haversine formula [26].
5. EXPERIMENTAL EVALUATION

5.7.1 Implementation

For the sake of simplicity and since the functionality won’t be used in further experiments, the MySQL implementation was hard-coded as a `SELECT` statement. The following code snippet presents such a statement under the assumption of two columns `p1` and `p2` of type `POINT`.

```sql
SELECT (6378.137 * acos(cos(radians(ST_X(p1))) * cos(radians(ST_X(p2))) *
        (cos(radians(ST_Y(p2)) - radians(ST_Y(p1))) + sin(radians(ST_X(p1))) *
        sin(radians(ST_X(p2))))) AS distance
```

Considering PostgreSQL includes a module for the computation of the distance between two coordinates, the syntax is comparatively simpler.

```sql
SELECT (p1 <@> p2) * 1.609344 AS distance
```

The two constants are necessary for the conversion to kilometer distance.

5.7.2 Benchmarks

Our case-study, Uska, operates in Sweden. In order to generate a meaningful and easily scalable data set, a rectangular area was defined around the cities Stockholm, Göteborg, Malmö and Umeå using top left and bottom right coordinates. Each row of the data set contains two points randomly chosen from one of the cities, within the range of the spanned rectangle. Figure 5.6 compares the performance of the two database systems.

Similar to the popcount extension, PostgreSQL performs slightly better than MySQL, after the minimum time required is passed. Consequently, we can rule out spatial calculations as the reason of performance gains or drops for future experiments including such calculations.

5.8 Data set generation

In order to conduct experiments in comparison to the initial results from section 5.1, a suitable data set needs to be generated. As already discussed, the goal is to use anonymized production data as a seed. Hence this section first discusses the extraction of said seed and then the properties of the generated data set.

5.8.1 Seed extraction

For the extraction of production data from the job-platform Uska, an obfuscated copy of the production database was used. Operating on data that is not capable of identifying
individuals, the matching profile tables were extracted. Since this data set is limited to
the current state of the platform, the idea is to use its attributes to generate more data.
For that purpose, a tool was developed to extract a statistical distribution from values
within a specific table column. It introduces a data structure called TableDictionary,
allowing to easily generate the aforementioned distribution for database tables.

The tool performs a SQL query, grouping all unique values for a given column and
generating an occurrence count for each value. After sorting the result by occurrence
count, this distribution is incorporated into the TableDictionary instance, labeled with
the name corresponding to the table and column name. This process can be executed
for a list of columns in a batch operation. After gathering the required distributions,
the data structure can be asked to generate a new value for a given column. A strategy
similar to lottery scheduling in operating systems is employed. The probabilistic algo-

rithm generates a number between 0 and the maximum amount of tickets available. The
column distribution is then iterated and the occurrence count deducted from the tickets,
until the generated ticket amount is exhausted. Once the loop ends, the current column
value is chosen and returned.

Using that strategy, existing data sets can be scaled or rather stretched, conserving
their state. References to the source code can be found in appendix A.1. In the next
section, the properties of the generated data set using this approach are motivated.
5.8.2 Properties

As previously mentioned (see 1.6), neither the base data set nor the generated one can be shared directly. Therefore this section describes the data set generated for the final benchmarks in its properties.

Uska as a platform contains around 60 times more candidates than job descriptions as of the time of this writing. Since the number of 100000 users was targeted for this report, that number was interpreted as number of candidates. Accordingly, around 1600 job descriptions were generated. The matching table contains 98 encoded attributes, with the majority being of type *grouped scalar*. The final database query is supposed to generate the same result as the original matching engine. In the end, it contained 13 popcount- and 27 arithmetic operations, that were executed on each line. Benchmarks can be found in the next section.

5.9 Matching engine benchmarks

In similar fashion to the initial benchmarks, the final ones were conducted, using a PHP script and clock. The data set was continuously increased according to the previous section. Figure 5.7 displays the performance of the final matching engine query.

![Figure 5.7: Log-log-plot of the matching engine query performance over an increasing sample size of candidates.](image)

The new version is significantly faster, surpassing one second runtime only when approaching 100000 candidates. This makes on-the-fly calculations a viability for the Uska
platform, especially at the current stage. With these low run-time numbers, a majority of the time is required to transfer the data yielded by the query. In contrast, figure 5.8 shows a frequency plot of the same query over 100000 entries, however limiting the results to 50 rows. Each of the frequency plot entries is an average of 50 executions. This really outlines one of the advantages of moving computation towards the database. Previously, it was impossible to predict which rows would yield the best matching results. A large chunk of data had to be extracted, transferred and parsed in the backend, before it could be computed, now only the specific data required in order to serve a request leaves the database.

Figure 5.8: Frequency plot of 50 trials over 100000 candidates, limiting the query output to 50 results.
6 Conclusion

The goal of this report was to study the performance implications of encoding attribute data in binary form and performing bitwise operations directly on the database for the specific use-case of the online job-platform Uska. Through an iterative approach, the encoding and computational components were introduced and evaluated. Most notably, the population count was focused on especially. The conclusion is, that the approach is absolutely viable for the online platform and the targeted growth towards 100000 users can easily be handled by the outlined bitwise approach. It alleviates computational strain on the web server immensely and is definitely recommended to Uska. This report serves additionally as a data point for future researchers, that the binary encoding should be considered an option.

6.1 Discussion

The journey of implementing the bitwise approach produced a lot of interesting side results that could be collected. Future research regarding the implementation of the population count extension could potentially make operations on higher column lengths, or bit alignment, possible. On the other hand, putting more and more data in a single column reduces the readability and increases the complexity of the logic needed for parsing. This definitely is also a trade-off for the case-study of this report. The encoding introduces more complexity in the backend to extract the attribute data. However, the benefits in user experience of reducing the request times in such a substantial manner, should outweigh the additional engineering time involved.

In order to reduce the run-time even further, the population count for columns of high length could be calculated directly on each row update. PostgreSQL offers Trigger Procedures, that can be configured to run on certain conditions. One of which is, that it is triggered once a row is updated. It would be interesting to see if the improvements in performance would be worth the introduction of additional columns. Another approach could be automatic vectorization [27], in order to parallelize population count algorithms.
References


A Appendix

A.1 Source code

All source code that can be shared as part of this report is publicly available in the following git repositories.

- PostgreSQL extension: https://github.com/eschmar/postgres-bit-count
- Great-circle distance: https://github.com/eschmar/great-circle-distance
- TableDictionary: https://github.com/eschmar/laravel-table-dictionary