Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Lars-Olav Vågene

Using the ICU library for collations in MySQL

Master's thesis in Informatics
Supervisor: Norvald H. Ryeng
June 2023

NTNU
Norwegian University of
Science and Technology

Lars-Olav Vågene

Using the ICU library for collations in MySQL

Master's thesis in Informatics
Supervisor: Norvald H. Ryeng
June 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

Collations are rules defining the order of characters in a given language that applications use to sort text in a locale-aware fashion. The International Components for Unicode (ICU) library is a widely used text-handling toolkit with collation support. It has extensive support for collation in various languages and is a popular choice for applications that require advanced collation support.

However, collations in ICU are not semantically stable, which means the underlying rules can change between versions and cause the same rule set to produce different results. Lack of semantic stability presents a problem for some applications, such as database management systems (DBMS). The indexes these maintain on disk can be invalidated by changes in collation order.

This report presents a proof-of-concept prototype designed to enforce semantic stability in collations derived from ICU. We use MySQL in the implementation, but the general principle applies to other applications with similar requirements. Several benchmarking experiments measure the effect of this modification on performance. These show that our prototype has comparable performance to the existing implementation in MySQL and that ICU offers significant performance improvements in some cases.

We also demonstrate a utility for verifying that two collations are semantically equivalent. This utility is notable for its fast run time, which makes it practical to use as an automated test verifying semantic stability. It is not exclusive to MySQL and is adaptable to any collation interface.

We use this utility to verify that our prototype is semantically equivalent to the original ICU collations. From this, we conclude that it is possible to enforce semantic stability in the ICU library and that this is a practical solution for applications that require semantic stability.

Sammendrag

Kollasjoner er regler som definerer rekkefølgen av tegn, noe som applikasjoner bruker for å sortere tekst for et gitt språk. International Components for Unicode (ICU) er et mye brukt bibliotek som blant annet har støtte for kollasjoner basert på Unicode-standarden. Det har omfattende støtte for kollasjoner for ulike språk og er et populært valg for applikasjoner som krever avansert kollasjonsstøtte.

Kollasjoner i ICU er imidlertid ikke semantisk stabile, noe som betyr at de underliggende reglene kan endre seg fra versjon til versjon og dermed gi forskjellige resultater. Mangel på semantisk stabilitet er et problem for noen applikasjoner, som for eksempel databasesystemer (DBMS). Disse opprettholder indekser lagret på disk og disse kan bli gjort ugyldige av endringer i kollasjonsrekkefølge.

Denne rapporten presenterer en prototype som oppnår semantisk stabilitet i kollasjoner basert på ICU. Implementasjonen er gjort i MySQL, men det generelle prinsippet bak gjelder for andre applikasjoner med lignende krav. Flere ytelsestester er gjort for å måle effekten av denne modifikasjonen. Disse viser at prototypen har sammenlignbar ytelse med den eksisterende implementasjonen i MySQL og at ICU i noen tilfeller gir betydelige ytelsesforbedringer.

Vi demonstrerer også et verktøy for å verifisere at to kollasjoner er semantisk like. Dette verktøyet er interessant fordi det har kort kjøretid, noe som gjør det mulig å bruke som del av automatisert testing for å verifisere semantisk stabilitet. Verktøyet kan tilpasses til hvilket som helst kollasjonsgrensesnitt og er ikke spesifikt for MySQL.

Vi bruker dette verktøyet for å verifisere at vår prototype er semantisk ekvivalent med de originale ICU-kollasjonene. Ut fra dette konkluderer vi med at det er mulig å oppnå semantisk stabilitet med ICU-biblioteket og at dette er en mulig løsning for applikasjoner som krever semantisk stabilitet.

Acknowledgements

I would like to thank my supervisor, Dr. Norvald Ryeng, for his guidance and support throughout this project. His enthusiasm for the field and willingness to help have been invaluable.

I would also like to thank the MySQL team in Trondheim for their help and for welcoming me into their office. In particular, I appreciate the guidance from Bernt Johnsen, Magnus Brevik, and Tor Didriksen in navigating the MySQL codebase.

CONTENTS

CHAPTER 1

INTRODUCTION

This thesis is a study of the ICU [1] library and its use for collation in applications that require semantic stability. This is investigated in the context of database management systems (DBMSs), and a prototype implementation is developed for MySQL, but the general approach should also apply to other applications.

The primary goal of the thesis is to investigate the feasibility of using ICU for collation in applications that require both semantic stability and the ability to upgrade to newer versions of the library easily. The scope of the investigation is limited to a single prototype implementation in order to demonstrate the feasibility of the approach, as well as a series of experiments to test the performance and correctness of the prototype.

In this chapter, we will discuss the context, background, and motivation for the study, the problem statement, the objectives and scope, the contribution and significance, as well as the structure of the thesis.

[1] International Components for Unicode

1.1 Background and motivation

The term *collation* commonly refers to the process of gathering and comparing data [2]. In the context of databases, collation refers more specifically to the process of sorting or comparing text according to a set of predefined rules. This set of rules is often referred to as a collation order or simply a collation, and hence the word can refer to either the act of sorting, the set of rules used for sorting, or a concrete implementation of such a rule set. These rules can be complex and depend on the language, region, and conventions of the intended audience, also known as the *locale*.

One example of collation is the ordering of names in a phone book, where the names are sorted alphabetically to allow a reader to find the phone number associated with a given name quickly. For example, in a Norwegian phone book, the name *Aasmund Aasmundsson* would be placed towards the end of the list [3]. On the other hand, it would be placed at the very beginning of a British phone book. Because the collation order depends on the locale, it is important for applications to use the correct collation order for the intended audience. While most users do not think about collation when using an application, it is an important part of the user experience and can be a source of frustration if it is done incorrectly. Collation is a fundamental operation in many applications, particularly in the case of DBMSs, where it is used for sorting and comparing text in queries, as well as for creating indexes on text columns. It is, therefore, important to get collation right and to ensure that it is done consistently.

ICU is a set of free and open-source libraries for Java, C, and C++, initially developed in the late 1990s by IBM and currently maintained by the Unicode Consortium [2]. Through the Unicode Consortium, ICU is supported by a broad alliance of companies and organizations [3] and is likely to be well maintained for the foreseeable future. ICU provides a wide range of support related to Unicode, text handling, and internationalization, including collation support for a large number of locales.

There are other libraries available that could be used for collation support, such as the GNU C library (commonly known as *glibc*). However, ICU is a common choice for applications that require support for multiple lo-

[2] From the Latin *collatiō*, meaning *to bring together* [1].

[3] Because in Norwegian *'Aa'* ≈ *'Å'*, the last letter of this alphabet. See section 2.3.2.

cales, and many different DBMSs already use ICU in some way for collation support [4]

While it is certainly possible for an application developer to implement collation support themselves, it is a complex task, and there are many advantages to using a third-party library. One of the main advantages is that the work of development and maintenance is offloaded to the maintainers of the library, which allows the application developers to focus on their core business. This can save a significant amount of time and money in terms of time spent developing new features. More importantly, the developers are freed from time-intensive maintenance tasks, such as fixing bugs and keeping up with changes in the Unicode standard. A third-party library is likely to have a larger user base and garner more interest from the community, which in turn means that bugs are more likely to be discovered and fixed. Such bugs could even represent security vulnerabilities, making it even more important to have them fixed as soon as possible.

However, a problem with using third-party libraries to provide collation support is that of *semantic stability*. In the context of databases, the term semantic stability refers to whether queries retain their original meaning while the implementation changes [4]. If the implementation is semantically stable, the same query should produce the same result every time. On the other hand, if a collation implementation changes and no longer produces the same result, then it is not semantically stable. A lack of semantic stability does not mean that the new result is wrong, merely that it differs from the previous result. This instability is acceptable or preferable for some applications, as the change is likely an improvement, and the main concern is that the result is correct. For example, the change could be a bug fix correcting an error in the previous implementation or adding a new symbol that did not exist before. In such cases, most users would prefer the new result over the old one as they likely value correctness over stability.

However, the lack of semantic stability can be a serious problem for other applications. One example of this problem is the use of indexes in DBMSs. An index is a secondary data structure that is used to speed up queries by maintaining a collection of sorted keys pointing to the primary

[4] See section 3.3 for a list of DBMSs using ICU.

data [5]. Indexes exist outside of the realm of databases as well, and a familiar example is the index at the end of a book. Imagine, for example, looking up the name *Aasmundsson* in the alphabetically sorted index of a Norwegian book. An international reader might expect to find this term at the start of the index, or at least before any terms starting with *B*. However, this relies on the assumption that the index was created with the same collation rules as the reader is used to. If the index is sorted using Norwegian rules, the name is instead moved toward the end.

Another possible scenario is the introduction (or removal) of a bug which causes the collation order to change and be different from what is expected. In either case, the result is that the user cannot find the term they are looking for. The same problem applies to an index in a DBMS, where the DBMS expects that the index was created using the same rules it uses when reading the index. If the rules differ and items are not in their expected order, the system will fail to find the data at the expected location. Changes to the collation order render the existing index invalid and make using it impossible or error-prone until the issue is resolved. The simplest solution when this happens is to rebuild the index. When changes are introduced to the collation order in a DBMS, the existing index is rendered invalid, and using it is likely to cause errors or other issues. Whether this represents a minor inconvenience or a significant problem depends on the size of the index, the time required to build it, and the guarantees provided by the DBMS in terms of downtime during upgrades, among other things.

Given these issues, one might expect ICU to be semantically stable, but this is not the case. While changes to collation order are rare, and changes to the default collation order are even rarer, they can happen, and ICU does not guarantee semantic stability when upgrading from one version to another. Upgrading to a new version of ICU can therefore break existing indexes, which is a serious problem for DBMSs. However, upgrading can also be essential, as the new version may fix security vulnerabilities or other serious bugs. Applications using ICU for collation are therefore left with a dilemma: Accept the lack of semantic stability, or forego regular patching of ICU. If they choose the former, they risk having their indexes break when upgrading to a new version of ICU. The latter option means

[5] For more information on indexes, see section 2.5.1.

requiring a static version of ICU to be used for the lifetime of the application, with no possibility of upgrading to newer versions. For applications where neither of these options is acceptable, such as MySQL, the only remaining choice has been to implement and maintain collation support on their own.

1.2 Problem statement

For security reasons, most applications require the ability to upgrade third-party libraries. MySQL is no exception to this, and keeping its dependencies up to date is essential to ensure that security vulnerabilities are fixed quickly. There are also applications that either require or would greatly benefit from having guaranteed semantic stability. This also applies to MySQL because it would be problematic to require users to rebuild their indexes due to changes in dependencies. The problem is that for collation, these two requirements conflict. Collations change slowly, but by their nature, they inevitably change over time. New characters are created regularly, and people change their minds about how to sort them. Such changes are eventually reflected in updates to the ICU library, alongside bug fixes to unintended errors in collation order. This makes it challenging to both require stable behavior and the ability to upgrade because the two requirements are fundamentally contradictory.

However, the architecture of ICU is such that it may be possible to solve this problem. The library exposes an interface for adding custom rules that modify its behavior, which could be exploited to provide semantic stability. This would allow the library to be upgraded while still providing the same collation order as before. Given the potential benefits of using ICU for collation, it is worth investigating whether this is a viable solution. These benefits primarily include extensive collation support, covering a wide range of languages and locales, but also the possibility of improved performance. Given that ICU is a specialized collation library, it may be able to provide better performance [6] than the current implementation in MySQL.

The main research question for this thesis is as follows:

[6] I.e., the time required to perform collation. See section 5.2 for details.

RQ1 How can applications that require semantic stability use ICU for collation without requiring a static library version?

Secondary research questions which support the main research question are:

RQ2 Given that ICU is a specialized library, does it perform better than the current collation implementation in MySQL?

RQ3 Does it harm performance to enforce semantic stability in ICU?

RQ4 How can we detect changes to a collation order?

While some available academic literature is related to the topics discussed in this thesis, very little is directly relevant. The topic is specific to a particular use case and library, and most applications do not require both semantic stability and regular patching. Because of this, the pre-study report for this thesis failed to find any academic literature addressing the issue of semantic stability [5]. There is, however, some discussion of the issue in articles and blog posts, which acknowledge and discuss the problem [7]. Therefore, we believe there is a gap in the literature that this thesis can help fill, however slim that gap may be.

1.3 Contribution and significance

The primary contribution of this thesis is a demonstration of how ICU can be used for collation in applications that require semantic stability across arbitrary versions of the library. This is achieved by implementing a proof-of-concept patch for MySQL which adds collations based on ICU, and then demonstrating that the patch works as intended.

As a secondary contribution, we demonstrate a utility for quickly comparing two collation implementations to determine whether they are equivalent. This utility was developed specifically to validate the results of the prototype implementation but could potentially be useful in other contexts as well. It is DBMS agnostic and can easily be adapted to compare different implementations, given that they both expose an interface for string comparison.

[7] See discussion in section 3.1.3 and section 3.2.

In addition, the thesis discusses the benefits and drawbacks of using third-party libraries and the specific challenges of refactoring an existing codebase like MySQL to use such libraries.

The original problem which spawned this thesis was the difficulty of using ICU for collation without breaking indexes in DBMSs. An alternative approach to solving this problem would be to investigate fast algorithms for repairing indexes, as that would alleviate this specific problem for DBMSs. While this is a possible future research direction, it is outside the scope of this thesis.

The main significance of the study lies in demonstrating that it is viable for applications like MySQL to use ICU for collation. This is important because it allows such applications to offload the work of implementing and maintaining collations to a third-party library and to benefit from the specialized knowledge and experience of the developers of that library. This, in turn, benefits the users of these applications in that they gain access to a broader set of collations that cover more languages and locales. It also benefits the developers of these applications in that they can focus on their core functionality and do not need to spend time on implementing and maintaining collations.

1.4 Objectives and scope

As stated by the research questions and desired contributions outlined above, the main objective of this thesis is to investigate the feasibility of using ICU for collation in applications that require both semantic stability and the ability to easily upgrade to newer versions of the library. MySQL is an example of such an application due to the issue of changes in collation order affecting indexes. The topic should, however, be relevant to other DBMSs and other applications which require semantic stability. The scope of the thesis is limited to the specific case of MySQL, but the results should also apply to other applications.

In order to achieve the stated objective, we will be using a research design consisting of a proof-of-concept implementation and several experiments. As the prototype implementation is a proof-of-concept, it is not expected to be production-ready, but it should be able to demonstrate the approach's feasibility.

The performance experiments are intended to show that the prototype implementation does not incur a significant performance penalty. However, it falls outside the scope of the thesis to provide a thorough and comparative analysis of this performance. While we plan to conduct a preliminary analysis of any observed performance disparities, we will not delve deeply enough to pinpoint the precise causes with certainty. For example, as the prototype is a proof-of-concept implementation, it is at a disadvantage in terms of performance when compared to a battle-tested and optimized implementation like the one currently used in MySQL. Also, the prototype must be adapted to work with the current MySQL code base, with minimal changes to the existing code [8]. Because of this, it is expected that the prototype will be sub-optimal in terms of performance, but the experiments should still be able to indicate whether or not there is a meaningful performance difference between the implementations being tested.

The results from the experiments will be presented in chapter 5, and the code used to conduct them will be made available to allow others to reproduce the results. The code for the prototype implementation will also be made available to allow others to build on the work potentially. Short excerpts or code snippets may be included in the thesis itself, but the code will otherwise be available on GitHub.

Data collection and analysis from the experiments will be performed using a combination of shell scripts and Python scripts. These will log the results of the experiments to an SQLite database, from which the graphs and tables in chapter 5 will be generated. The experiments will provide a quantitative basis for evaluating the prototype implementation, but the advantages and disadvantages of the approach will also be discussed qualitatively. For example, certain aspects of the prototype implementation may be challenging to quantify, such as the maintainability of the code or the benefit of having access to ICU collations.

1.5 Structure of the thesis

Chapter 2 presents relevant background information and underlying concepts required for understanding the problem and the proposed solution,

[8] See discussion in section 4.4.

which may serve as a helpful reference for readers unfamiliar with the topic. This is followed by a review of related work in chapter 3, where we discuss the current state of the art and the challenges faced in the field. In particular, we will discuss how other DBMSs deal with the problem and how they compare to the approach taken in this thesis, as well as the current implementation in MySQL.

In chapter 4, we describe the basic principles of our proposed solution, followed by a more detailed explanation of its implementation. The prototype implementation consists of a modification of the MySQL codebase that adds collations based on ICU.

Chapter 5 outlines the experiments conducted to verify the correctness and performance of the prototype implementation. The performance experiments aim to measure any performance gains or penalties incurred by the prototype implementation. In contrast, the correctness experiments aim to verify that the prototype produces the same collation order as the original ICU collations.

Finally, in chapter 6, we summarize the experiments' results and what has been discussed in the thesis, and what that means for the feasibility of the approach.

CHAPTER 2

BACKGROUND AND MOTIVATION

This chapter gives a brief introduction to some of the concepts and technologies that are relevant to the thesis. It is intended to provide the reader with enough background material to understand the rest of the thesis. For readers who are already familiar with the topics discussed, it can be skipped and referred to as needed. We will also give a more detailed explanation of the technical challenges forming the motivation behind the thesis, which were discussed briefly in the previous chapter. This includes a discussion of the problems that the thesis aims to solve and why they are important.

In section 2.1, we discuss some general concepts relevant to the thesis, such as code reuse, third-party libraries, and dependency management. Next, we cover the concept of collation in greater detail and define relevant terms in section 2.2. We also discuss when to consider two collations equivalent and how this relates to a semantically stable collation implementation. We then cover some relevant concepts from the Unicode standard in section 2.3 and introduce the ICU library in section 2.4. Finally, in section 2.5, we explain through practical examples how indexes in databases are affected by collation.

2.1 Code reuse

Code reuse refers to using existing code in new projects rather than writing new code from scratch. Writing code that is versatile enough to be reused has a high upfront cost, but it can save time and effort in the long run because less code needs to be maintained [6]. It can also lead to higher-quality software because the code has already been tested and used elsewhere. The high upfront cost of developing reusable code is often justified because it can be reused in many different projects, which means the cost is spread over many projects. From the point of view of individual developers and organizations, this cost is often irrelevant because someone else "pays" for it, and the code is used as-is. This typically takes the form of using a third-party library, where the code is written by someone else and made available for others to use. Much like in a physical library, a developer can browse and borrow freely from this collection of resources and use it in their projects. When a project relies on a third-party library, this is referred to as a *dependency*. Code reuse is a common practice in software development [7], and it can significantly reduce the work required to develop software.

A key advantage of using third-party libraries is that it allows developers to focus on the core functionality of their software rather than spending time on implementing features that are not directly related to the purpose of the software. It also allows developers to use code written by experts in the field, which can lead to higher-quality software. In practice, the reused code is also often more mature and better tested than newly written code, which means there is a greater chance that security vulnerabilities and bugs have been found and fixed. This is particularly true for widely used open-source libraries, which are often used by many projects and have a large community of developers and users.

There are, however, some disadvantages to using third-party libraries. For example, the library is not likely to be tailored to the project's specific needs, so some extra work or inefficiencies could be involved. Also, if the library is not well maintained, it could become outdated and force the users to either update it themselves or find a replacement. This could happen, for instance, if a critical security vulnerability is found in an abandoned library or if the library needs to be updated to work with a newer

version of the programming language or operating system.

However, the advantages of using third-party libraries typically outweigh the disadvantages, and few choose to write custom implementations if a suitable library exists.

2.1.1 Versioning

Because third-party libraries are often updated, it is important to keep track of which version of the library is being used. This is necessary because the library might change in ways that affect the application using it, such as by removing or changing features or by introducing bugs. While library developers are not likely to be intentionally breaking their software, it is not uncommon for an update to introduce a bug or incompatibility that was not caught during testing. For these reasons, many organizations and developers have a policy of not updating their dependencies immediately (unless critical security vulnerabilities are discovered) but instead waiting for a while to see if any issues are reported by other users.

One common approach to describing software versions, be it for a library or an application, is to use *semantic versioning* (also known as *semver*) [8]. This is a versioning scheme that, in its simplest form, consists of three numbers separated by dots, such as 1.2.3. The first number is the *major version* and is incremented when the update contains breaking changes. The second number is the *minor version* and is incremented when the update contains new features that are backward compatible. The third number is the *patch version* and is incremented when the update contains backward-compatible bug fixes. This is meant to give users of the software an indication of the level of difficulty in upgrading to a new version. For example, upgrading from 1.2.3 to 1.2.4 should be a trivial upgrade and is not likely to cause issues. On the other hand, upgrading from 1.2.3 to 2.0.0 could be a major undertaking, and users should be prepared for breaking changes. Developers can use version numbers like these to specify a range of acceptable versions for their dependencies. For example, a developer could specify that their software requires version 1.2.3 or newer but not version 2.0.0 or newer. This is typically done by specifying a range of versions, such as [>=1.2.3, <2.0.0].

Not all projects follow semantic versioning, but it is a common practice and is used by many popular libraries and applications. Even for projects

following semantic versioning, mistakes can still happen, and breaking changes can be accidentally introduced in a minor or patch version. However, it is still a useful tool for developers to communicate the level of risk involved in upgrading to a new version.

2.1.2 Dependency management

Once the choice to use a third-party library in a project has been made, the next step is to decide how to manage this dependency. Dependency management refers to handling, including, and updating dependencies in an application. The two main approaches are *bundling* and *system libraries*. The exact details of how it is done depend on the programming language, build system, and operating systems used, but the general ideas are the same.

When bundling a library, the library is somehow included with the project and provided alongside the source code or binaries of the project. However, this means that the library is not shared with other projects and that each project has a separate copy of the library. This duplication can lead to inefficiencies, as the user might end up with multiple copies of the same library on their system. More significantly, it can be a security issue because bundled libraries are not updated automatically. Instead, it falls on the developer to keep track of updates and manually release a new version of their application with an updated version of the library included. This is a problem because it relies on manual action from both the application developer and the user. Even if both are actively working to keep the software updated, it could significantly delay the release of security updates, leaving users vulnerable to attacks.

Because of these issues, it is generally considered to be a better practice to use system libraries instead whenever possible. This means the application relies on the library being available on its system rather than including it with the project. Using shared system libraries allows each library to be shared between multiple projects, and it also means that the library can be updated independently of the project. Installing and updating these shared libraries is typically done by the operating system (or a package manager included with the operating system). However, it can also be done manually by the user. The operating system or package manager can also be configured to automatically install updates, so the user

does not have to worry about keeping the libraries updated. Even if a system is not set up for automatic updates, it is still easier and faster to update a shared library than a bundled one.

While bundling libraries is problematic, it does guarantee that the library is available on the system and that the expected version is used. When relying on system libraries, the application cannot control which libraries are installed or their versions. Therefore bundling is sometimes still used, but it is often discouraged and considered a last resort. For example, the Fedora Project (a common Linux distribution) has a policy recommending that shared libraries should be used if possible [9]. Shared libraries are preferred because they allow the system to update the library independently of the application using it, which means that security vulnerabilities and bugs can be fixed without having to update the software. It also reduces the amount of disk space the system uses because the library only needs to be stored once and can be shared by all applications. However, there are cases where it is necessary to bundle dependencies, such as when the target system does not provide the required libraries. For example, MySQL currently bundles ICU because it was previously unavailable on some systems they supported. However, this was acceptable for them because they did not use it for collation and could update the bundled version easily.

In practical terms, there is no difference between a system dependency pinned to a specific version and a bundled dependency because the software will only work with the required library version. In either case, the software will be locked to a specific library version, so the only difference is where the library is stored. However, if the software developers trusted the versioning scheme of the library sufficiently, they could use a system dependency and specify a broader range of accepted future versions. For instance, if the current version of the library is 1.2.3 when the application is released, the application could still be configured to accept system libraries in the range [>=1.2.3, <1.3]. In this scenario, it would be possible for users to upgrade to newer *patch* library versions without requiring any action from the application developers.

2.2 Collation

In this section, we will discuss the concept of collation in greater detail than in the introduction. We will also attempt to provide clear definitions of relevant terms and concepts related to collation which will be necessary for the rest of the thesis. We start with a brief overview of relevant concepts from set theory in section 2.2.1 to be able to define more precisely what a collation is in section 2.2.2. These concepts are necessary to discuss in section 2.2.3 what it means for a collation implementation to be *semantically stable* and to be *semantically equivalent* to another collation implementation.

2.2.1 Set theory

As stated by Cormen et al., a binary relation R on sets A and B is a subset of the Cartesian product $A \times B$ [10]. Similarly, a binary relation R on a set A is a subset of the Cartesian product $A \times A$. In other words, a binary relation R on A is the set of all possible pairs of items from A related by R. Thus, the claim *"a is related to b by R"* can be written as $(a, b) \in R$ or aRb. For example, the relation *"greater than"* is defined on the set of integers $\mathbb{Z}$ as the set of all pairs (a, b) where $a > b$.

The following properties of a relation R on a set A are relevant here:

Reflexivity: aRa for all $a \in A$.

Symmetry: $aRb \implies bRa$ for all $a, b \in A$.

Antisymmetry: $aRb \wedge bRa \implies a = b$ for all $a, b \in A$.

Transitivity: $aRb \wedge bRc \implies aRc$ for all $a, b, c \in A$.

In other words, if the relation is reflexive, then every item is related to itself. If it is symmetrical, then a being related to b means that b is also related to a. Conversely, if it is antisymmetrical, then the relation is never symmetrical for different elements. Finally, if it is transitive, then if a is related to b and b is related to c, then a is also related to c.

If a relation R on A is reflexive, antisymmetrical, and transitive, then it is called a *partial order* and the set it defines a *partially ordered set* [10]. When this relation holds for all elements in A, it is called a *total order*, and

the set is called a *totally ordered set*. For example, the relation *"greater than or equal to"* is a total order on the set of integers $\mathbb{Z}$, because for every possible pair of integers, at least one of them is greater than or equal to the other. On the other hand, the relation *"greater than"* is a partial order on $\mathbb{Z}$ because it is not reflexive.

Two relations are equivalent if they relate the same pairs of items, i.e., they produce the same result for all possible inputs. Formally, this can be defined as follows:

Domain For a relation R on a set A, the domain of R is the set of all $a \in A$ such that $(a, b) \in R$ for some $b \in A$ [11].

Range For a relation R on a set A, the range of R is the set of all $b \in A$ such that $(a, b) \in R$ for some $a \in A$ [11].

Equivalence of relations Two relations R and S on a set A are equivalent if they have the same domain and range, and for all $a, b \in A$, $aRb \iff aSb$ [10].

2.2.2 What is collation?

As mentioned in the introduction, the term *collation* has several related meanings. The intended meaning can usually be inferred from the context, but it is important to be aware of the different concepts. According to the *Merriam-Webster* dictionary, the word *collation* refers to the "act, process or result of collating". In turn, *collating* can mean either "to compare critically" or "to assemble in proper order" [1]. Other definitions, in the context of computer science, include "the process and function of determining the sorting order of strings of characters" [12] and "language-sensitive string comparison" [13]. Collation is defined more precisely in the SQL standard [14], which states that:

> A collation is defined by [ISO14651] as "a process by which two strings are determined to be in exactly one of the relationships of less than, greater than, or equal to one another".

However, in practical usage, the term is used to refer to both the sorting order itself, i.e., the abstract rules that define the order, as well as the implementation of these rules. When necessary to avoid confusion, it is

possible to differentiate between the process, the rules, and the implementation by referring to them as *collation process*, *collation order*, and *collation implementation*, respectively. By extension, something performing collation is called a *collator*.

Key phrases in the definitions above include *proper order* and *language-sensitive*. The proper order depends on the context and the language, which is usually called a *locale* in this context. A locale identifies a specific user community, i.e., a group of users who share a similar language and culture [15]. User communities range from entire countries or regions to smaller groups and subsets of other communities. For example, we can consider the US military a separate locale from the US civilian population, as they have different conventions for writing dates and times. This report will generally refer to locales by their locale ID in the format used by ICU. This is a string consisting of one or more pieces of ordered information, including language code, country code, script code, and variant code. Variant codes can represent things such as currency formats, dialects, and collation orders. The language and country codes are defined by the ISO standards ISO-639 and ISO-3166 [15]. For example, these are valid locale IDs and their meanings:

en_US English as used in the United States

sr_RS@latin Serbian as used in Serbia with a Latin script

es_TRADITIONAL Spanish with traditional collation rules

The definitions above differ somewhat, but they refer to the same concept and generally describe the same thing. However, in this thesis, it will prove useful to be able to determine whether two collation implementations are equivalent. While two collations might be implemented differently, they should still be considered equivalent if they produce the same result for all possible inputs. Implementation details of collations can still affect performance, but they should not affect the correctness of the result. We will therefore attempt to define a collation order more precisely, using the definition in the SQL standard as a starting point. This definition does not describe three separate relationships but rather the three possible outcomes of a single relationship. We can define this relationship as a binary relation R on a set of strings S. For any two strings $a, b \in S$, aRb if

and only if a is less than, greater than, or equal to b. Because the relationship places all strings in precisely one of the three categories, it is a total order on the set of strings.

An important caveat here is that a total order does not define the order of items that are equal to each other. For instance, a case-insensitive collation order might define *"aardvark"* and *"Aardvark"* to be equal, but it does not define which one should come first in a sorted list. The order of two equal items is, therefore, not defined by the collation order but by the sorting algorithm used [12], which may or may not implement a stable sort that will produce the same order on every repetition of a query.

2.2.3 Semantic stability and equivalence

This thesis aims to preserve semantic stability across different versions of the same collation implementation. However, for this discussion, it does not matter whether these collation implementations are different versions of the same implementation with minor changes footnoteI.e., ICU 70.1 versus ICU 70.2, or completely different implementations. The key point is whether or not two implementations produce the same collation order. If they do, we will consider them to be *semantically equivalent*. Semantic equivalence refers to the concept of two functions having the same meaning or producing the same result for all possible inputs, regardless of how they are implemented. Using the fact that a collation order can be defined as a relation imposing a total order on the set of all strings, we can define this more formally as follows:

Semantic equivalence Two collation implementations are semantically equivalent if their respective collation orders are equivalent relations.

Semantic stability A collation library is semantically stable if every version of its collation implementation is semantically equivalent to its predecessor.

2.3 Unicode

The *Universal Coded Character Set* (UCS or Unicode) is a standard method for representing text. It seeks to specify a set of characters covering all human languages, using a shared encoding scheme [13]. As of version 15.0.0, the Unicode standard defines 149,186 characters and 161 scripts [16]. This specification provides a unique, unambiguous value for every character, regardless of language or platform. A universal specification for characters is necessary to store and transmit text consistently and safely, preserving the meaning of the content. Historically, conflicting standards for representing text have made exchanging text between different systems challenging [1]. When a system attempts to read text encoded in a different format, the characters can be misinterpreted, resulting in corrupted data. This is often seen by characters being replaced with question marks or other unintended characters, such as *MÃ¼nchen* instead of *München*.

For historical reasons, Unicode and ISO 10646 are separate and distinct means, which stems from the fact that different organizations proposed these standards separately in the 1980s and later unified. They are effectively identical, however, due to efforts to keep them in sync. The *Unicode Consortium* and *ISO/IEC JTC1/SC2/WG2* are the two separate organizations maintaining these standards. However, when the text refers to *Unicode*, it generally means either the Unicode standard or the consortium which manages it, as the details of how this relates to ISO 10646 are irrelevant to the discussion.

However, some key terms relating to Unicode are worth considering when discussing text. For example, the Unicode standard distinguishes between characters and glyphs. Unicode is concerned primarily with the semantic meaning of characters and not their visual representation. A character is a semantic concept, such as *the Latin letter A*, while a glyph is a visual representation of a character [2]. Unicode further separates these concerns with a five-layer model [13]:

Abstract character repertoire A set of characters, such as the Latin alphabet.

[1] This is why we sometimes see a question mark instead of the intended character.

[2] Visual representations are the domain of fonts, which are luckily beyond the scope of this thesis.

Coded character set A mapping from the abstract character repertoire to a set of code points, e.g., the mapping from the Latin alphabet to the code points U+0041 to U+005A. Code points are the positions of characters in a table.

Character encoding form The storage format, mapping code point values to integers, also known as code units.

Character encoding scheme Serialization format, mapping code units to bytes. Here code units more than a byte long are converted to a sequence of bytes. UTF-8 and UTF-16 are examples of character encoding schemes.

Transfer encoding syntax Optional transformation to encode the bytes in a more constrained syntax, such as Base64.

2.3.1 CLDR and DUCET

Common Locale Data Repository (CLDR) is a Unicode project to provide locale-specific data for use in applications. Among other things, it contains number formats, date formats, and time formats. More importantly for this thesis, CLDR also contains collation orders. Default Unicode Collation Element Table (DUCET) is the default collation order for all Unicode characters and is defined in the Unicode standard [3]. It is impossible for a single collation order to satisfy all languages, but DUCET is designed to be a reasonable default for most languages. This is then used as a starting point for other collation orders, which modify or override the order of specific characters or ranges of characters in DUCET. The ICU library uses a root collation order which is based on DUCET but which may contain additional modifications or extensions. For instance, almost all currency characters are grouped in DUCET except for two characters. These are the characters ₨ (RUPEE SIGN) and ﷼ (RIAL SIGN), which the ICU root collation moves into the currency group [17].

However, collation orders derive from human language and regional conventions, which makes them inherently unstable and subject to change. To quote the Unicode Consortium [12]:

[3] The table itself is available at https://www.unicode.org/Public/UCA/latest/allkeys.txt.

> Over time, collation order will vary: there may be fixes needed as more information becomes available about languages; there may be new government or industry standards for the language that require changes; and finally, new characters added to the Unicode Standard will interleave with the previously-defined ones. This means that collations must be carefully versioned.

One example of this is the addition of the character ㋿ (U+32FF) in 2019, which was added to the Unicode standard in version 12.1 [18]. On the ascension of the new Japanese emperor, a new era was declared in the Japanese calendar, and this character was added to the standard to represent this era. It is a combination of two existing kanji, 令和, which can be romanized as *Reiwa*. This character was then assigned to a previously unused code point and inserted into the collation order at the appropriate position. This is an example of how the collation order is subject to change and why it is important to keep it up to date. The addition of new symbols is not a rare occurrence, with 4,489 characters added in version 15.0 of the Unicode standard alone. While the addition of new characters is one reason for changes to the collation order, it is not the only one. For instance, it can and does happen that the collation order of existing characters is changed. This can happen for a variety of reasons, such as changing standards for the language or an error in the previous collation order.

2.3.2 Practical considerations: An example

In the introduction, we mentioned some practical differences between collations for different locales, using an example familiar to the author. We will now explain this example in more detail and use it to illustrate some of the challenges of enforcing rules derived from the conventions of natural language. The Norwegian alphabet extends the Latin alphabet with three additional letters: *æ/Æ*, *ø/Ø*, and *å/Å*. Because they are more recent additions, they are placed last in the alphabet, after *z/Z*. However, historically the sounds represented by these letters were written using other symbols, such as the digraph *aa* instead of *å* [4]. When used to repre-

[4] The circle in *Å* represents a tiny *A* stacked on top of the main *A* [19].

sent this sound, this digraph is considered equivalent to the letter *å* and should be sorted as such [20]. This means that the collation order for Norwegian is not as simple as just adding three letters to the end of the alphabet.

Note that the rule is based on how the word is pronounced, not how it is written. This cannot easily be generalized to a clear and universal rule digestible by a collation system. A human might understand that *afrikaans* should sort before *afrikaner*, while *Aasmund* is equal to *Åsmund*, but this cannot be expressed in a simple rule. It is possible to handle the most common cases using a dictionary of exceptions or to manually sort a list of words, but this is not a suitable solution for a general-purpose collation library. This is a common problem with natural language, and it is not limited to Norwegian. In this specific case, the CLDR rules for Norwegian handle it by treating the digraph *aa* as an accented version of *Å* [21]. A collation based on these rules will therefore sort *afrikaner* before *afrikaans*, which is incorrect. This is an example of how the rules for a collation order can be difficult to implement in practice and that the end result is not always consistent with the conventions of natural language.

2.4 ICU

As the introduction mentions, the ICU project is a widely used toolkit for handling text. It comprises many different modules for handling different aspects of text, such as collation, date formatting, character set conversions, and more. The project was initially released in 1999 under the name *IBM Classes for Unicode*. It was later renamed to *International Components for Unicode* to reflect the fact that it was no longer an IBM-only project [22][5]. There are separate sub-libraries for Java (ICU4J) and C/C++ (ICU4C), and the C/C++ library is the one we are concerned with in this thesis. The C/C++ library has separate interfaces for C and C++, but the functionality is essentially the same. Whether we refer to ICU as a library or a set of libraries is a matter of perspective and the level of detail required. However, for simplicity, we will generally refer to it as a library. In this thesis, we are primarily concerned with the collation module of

[5] With the new name chosen specifically to avoid changing the abbreviation.

ICU4C, and references to ICU will refer to this unless otherwise specified.

2.4.1 Tailoring and comparison levels

Collation orders in ICU are specified as modifications, or *tailoring rules*, of the root collation order. This modification uses tailoring rules applied on top of the root collation order. These rules specify the relationship between characters and strings, which is used to determine the collation order. This relationship is either a *equal* relationship, specifying that the two strings are equal, or one of four levels of *difference*. ICU provides a complex syntax for specifying tailoring rules [6], which is beyond the scope of this thesis. By specifying the different levels of difference in the rules, the same rule set can be used to generate different collation orders. Collators can be sensitive to either *primary* (base letter), *secondary* (accent), *tertiary* (case), or *quaternary* (kana) differences. Generating different collation orders from the same set of rules is done by simply changing the collator's comparison level or *strength*. For instance, the only difference between a case-sensitive and case-insensitive collation for the same locale in ICU is the strength level set on the collator.

2.4.2 Usage

Two main ways to perform collation with ICU are relevant to this thesis. The first is directly comparing two strings, i.e., a comparison operation. Comparison means using a Collator object to compare the two strings, which returns the integers -1, 0, or 1, depending on whether the first string is considered less than, equal to, or greater than the other. The second way is to generate a sort key for a string. Sort keys are also generated with a Collator object, but instead of comparing the strings directly, the Collator transforms the string into a sequence of bytes, which is the sort key. This sort key can then be compared to the sort key of another string to determine the collation order. While this is a significantly more expensive operation upfront than a direct comparison, it is much faster to compare two sort keys than two strings [23]. This is because the sort keys are just a sequence of bytes and can be compared byte by byte, whereas

[6] https://unicode-org.github.io/icu/userguide/collation/customization/

the strings need to be parsed and compared according to the collation rules. For this reason, sort keys are often used in databases that need to sort large amounts of data. However, ICU recommends against using sort keys unless absolutely necessary, as direct comparisons are generally much faster [24]. Another difference between the two methods is that ICU offers both UTF-8 and UTF-16 interfaces for the comparison operator but only a UTF-16 interface for sort key generation. ICU generally uses UTF-16 internally, and only some of the collation methods have UTF-8 interfaces. In this case, the strings need to be converted to UTF-16 before generating the sort keys, which adds additional overhead to the operation.

2.4.3 Changes across versions

Collations in ICU are not guaranteed to be stable across versions. They are instead constantly updated to reflect changes in the Unicode standard, as well as to fix bugs and improve performance. This means that the collation order of a string might change between versions of ICU. ICU offers a versioning scheme for collators, which indicates whether or not the collation order has changed between versions. This can be used to quickly determine if there have been breaking changes, but it does not provide any information about what has changed. One important caveat with sort keys is that they are not guaranteed to be stable across different versions of ICU. Their generation is an implementation detail of ICU, and it is recommended to regenerate them when upgrading to a new version of ICU [23]. This means that it is unwise to store sort keys in a database unless regenerating them on every upgrade is acceptable. However, the actual order of the strings should be stable unless deliberate changes have been made to the collation rules.

ICU does not follow semantic versioning explicitly, but the basic principle is similar [25]. The versioning scheme used by ICU is more complex than semantic versioning, as it is used for many different components, and each component has a separate version number. For example, the ICU library itself has a version number, but so does the ICU data, ICU4C and ICU4J libraries, and the collation code. Because of this, there are interfaces for checking the version of each component at runtime, which can be helpful for applications that need to support multiple versions of ICU. As ICU still follows the general principles behind semantic versioning, it

```
#include <unicode/uvernum.h>

if (U_ICU_VERSION_MAJOR_NUM > 75) {
  // Handle logic for ICU version > 75
} else if (U_ICU_VERSION_MAJOR_NUM > 70) {
  // Handle logic for ICU version > 70
} else if (U_ICU_VERSION_MAJOR_NUM > 66) {
  if (U_ICU_VERSION_MINOR_NUM > 1) {
    // Handle logic for ICU version > 66.1
  } else {
    // Handle logic for ICU version > 66
  }
} else {
  // Handle logic for ICU version <= 66
}

```

Example 2.1: Example demonstrating handling different versions of ICU in C++

is still possible to use their version numbers to indicate compatibility. For example, an application could specify that it supports ICU versions 66 to 75 but internally handle ranges of versions differently (as shown in figure 2.1). Such a range might then be specified as "`>=65.0.0, <76`" in the application's configuration file [7], which would indicate that any ICU version from 65.0.0 up to (but not including) 76.0.0 is supported. Specifying a range of versions would allow users to use the application with any version of ICU in the given range. However, the application could still take advantage of improvements in newer versions if they are available.

A caveat here, as mentioned in the ICU documentation [25], is that only the C interface is compatible across versions. Because of technical limitations in C++, an application using the C++ interface needs to be compiled for a specific version of ICU and cannot be used with other versions. An application that accepts a range of ICU versions should instead use the C interface. Because C++ is a superset of C, this is not a significant problem, as the C interface can be used from C++ code.

[7] The syntax will vary, but the principles are generally the same.

2.5 Databases and collation

A database management system (DBMS) is a sophisticated piece of software that is used to store and manage data. Many popular DBMSs have been around for decades and have been continuously updated and improved over the years. MySQL is no exception, as it was released in 1995 [26] and is still widely used today. Many design choices made in the early days of DBMSs are still present in modern systems, even if their original reasoning is no longer valid. For example, when MySQL was first developed, ICU had not yet been released. Given that ICU is now the de facto standard for Unicode support, it is likely that a new DBMS developed today would use it for collation. However, as we have discussed previously, there are disadvantages to using ICU for collation in a DBMS. Chiefly, the fact that ICU collations are not guaranteed stable across versions is problematic for indexes relying on the collation order.

2.5.1 What is an index?

Indexes are secondary data structures that in databases are used primarily to speed up queries. They provide a short and direct path to relevant data, often much faster than scanning the entire table. Because indexes are separate data structures, they can be optimized for the specific use case. Indexes are also much smaller than the table they are derived from and can often be kept entirely in memory. Reading data from disk is a slow operation, so minimizing the amount of data that needs to be read is essential for performance. Even reading from memory has a cost, so an index typically aims to minimize the number of distinct read operations required.

There are many types of indexes, but a common type we will discuss here is the B-tree. B-trees are self-balancing, sorted tree data structures that allow for efficient search, insertion, and deletion of data [27]. This a type of search tree where each node contains one or more keys and a pointer to either another node in the tree or a data record. The keys in each node are sorted and guide us to the next relevant node. Following the pointers from the root node to the leaf node, we restrict the search space at every level to a subtree that contains the desired data. Most implementations of B-tree indexing use a variation called a **B+-tree**, which

we will use for our examples. Unless otherwise specified, references to B-trees here refer to this variation.

2.5.2 Practical examples

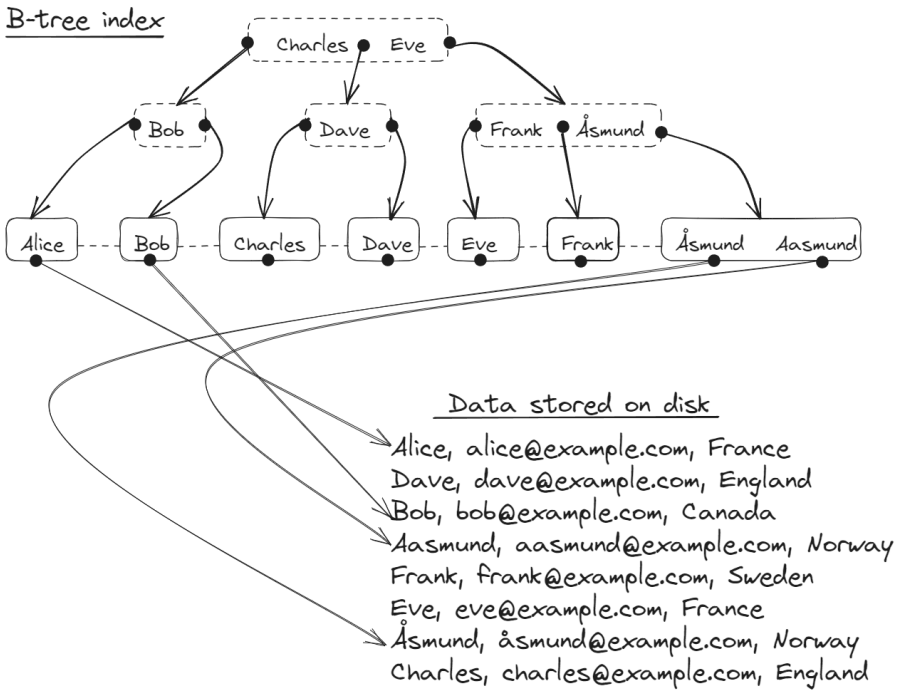


Figure 2.1: Simplified B-tree example. Keys are sorted by their Norwegian collation order.

In order to illustrate the issue, we will use an example of a B-tree index. Assume a table named users, with the three columns Name, E-mail, and Country. We create an index on the Name column to allow us to quickly find the associated E-mail and Country for a given name. We further specify that this index should be sorted using the Norwegian collation order, as we know that the users of our system will be Norwegian. Recall that in Norwegian, a name starting with *Aa* is considered the same as starting with *Å*, and both are sorted after *Z*. Figure 2.1 shows a simplified example

of this scenario. Here we see two levels of internal nodes (dotted lines) with one level of leaf nodes at the bottom. Each node can contain multiple keys, and each key is associated with a pointer to either another node or a data record. Note that we have left out some of the pointers from the leaf nodes to disk for readability. The dotted lines between the leaf nodes indicate they can be traversed like a linked list, allowing efficient range queries. A range query is a query that returns all data within a given range, for example, all names starting with *D* to *E*. In this example, such a query would need to scan at most two intermediate nodes and three leaf nodes because it would only need to find the start of the range and traverse the leaf nodes until the end of the desired range.

It would be equally possible to create a separate index on one of the other columns or a combination of multiple columns. In a database, having multiple indexes on the same table is common, as this allows for more flexibility when querying the data. While the issue of how to store data on disk is an essential consideration in a DBMS, it is not relevant to this thesis. Assume for the sake of argument that the data is stored in a heap file, which means that it is stored as a sequence of bytes in the order it was inserted. The index contains pointers to the exact position of the row associated with each key, allowing the DBMS to look up the desired data directly. Without an index, the data would have to be stored and sorted in a specific way (perhaps on a single column) to allow for efficient lookup. Relying on how the data is stored in sorted order would mean that queries on other columns would be much slower, as the entire table would have to be scanned to find the relevant data.

Let us examine what happens when the DBMS executes this query:

```
1   SELECT 'email' FROM 'users' WHERE 'name' = 'Aasmund';
```

Assuming that the DBMS chooses to use the index on the Name column [8], it will start at the root node and follow the pointers to the leaf node containing the key *Aasmund* as illustrated by fig. 2.2. For instance, after reading the first node containing the keys [Charles, Eve], it will compare each of them to the search key *Aasmund*. Since *Aasmund* is sorted after *Eve*, it will

[8] In a real scenario with such a small table, the DBMS would realize a table scan is faster and do that instead.

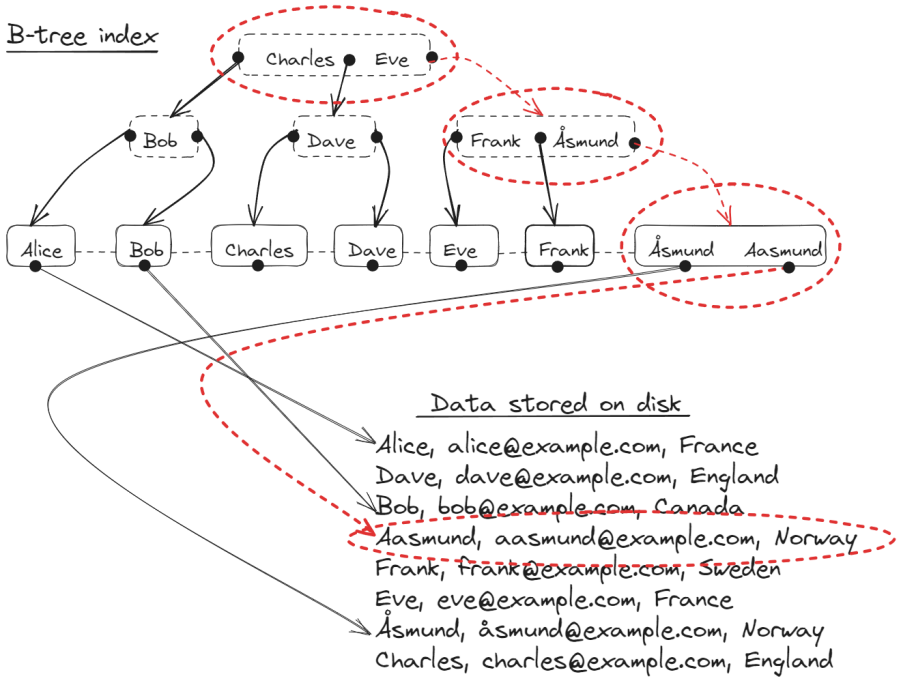


Figure 2.2: Searching for *Aasmund* in B-tree. Red circles and arrows indicate data read from disk or memory.

follow the pointer to the next node after *Eve*. This process continues until the DBMS finds the leaf node containing the key *Aasmund*, or it has determined that the key does not exist in the index. By checking in each intermediate node whether the search key is less than or greater than the keys in the node, the DBMS can determine which subtree to follow. It does not know whether or not the key exists in the index until it reaches the leaf node, only that it cannot exist in any of the other subtrees. If the DBMS traverses the index down to the leaf nodes without finding the key, it assumes the key does not exist in the index and returns an empty result set.

Note the fact that we are performing a comparison operation on strings here. As we have discussed, the collation determines the result of a comparison operation on a pair of strings. This means that we must use the same collation when we search through the index, as was used in creating and updating the index. Here, a Norwegian collation [9] was used both for creating the index and for searching through it. According to this, Aasmund is sorted after all the other keys in the index and will be found in the last leaf node.

However, what would happen if we used a different collation when searching through the index? Let us repeat the same operation, but this time using an English collation [10]. This collation has no special rules for *Aa* and will simply consider them as two separate letters which precede *B*. Again, the DBMS will read the root note and compare the search key *Aasmund* to the keys in the node. However, this time the DBMS will find that *Aasmund* is sorted before *Charles* and will follow the pointer to the left subtree. In the next node, it will find that *Aasmund* is sorted before *Bob* and will follow the pointer to the left, which points to the leaf node containing *Alice*. Upon scanning this, it determines that all the keys in this node are sorted after *Aasmund*, which should mean that the key does not exist in the index. As we can see illustrated in fig. 2.3, the DBMS will not find the key *Aasmund* in the index, even though it exists. Instead, it will return an empty result set.

This example illustrates the importance of using the same collation when searching through an index as was used when creating it. The actual scenario is contrived, as it is unlikely that a DBMS would use a different

[9] For example, *utf8mb4_nb_0900_ai_ci* in MySQL.
[10] For example *utf8mb4_0900_ai_ci* in MySQL.

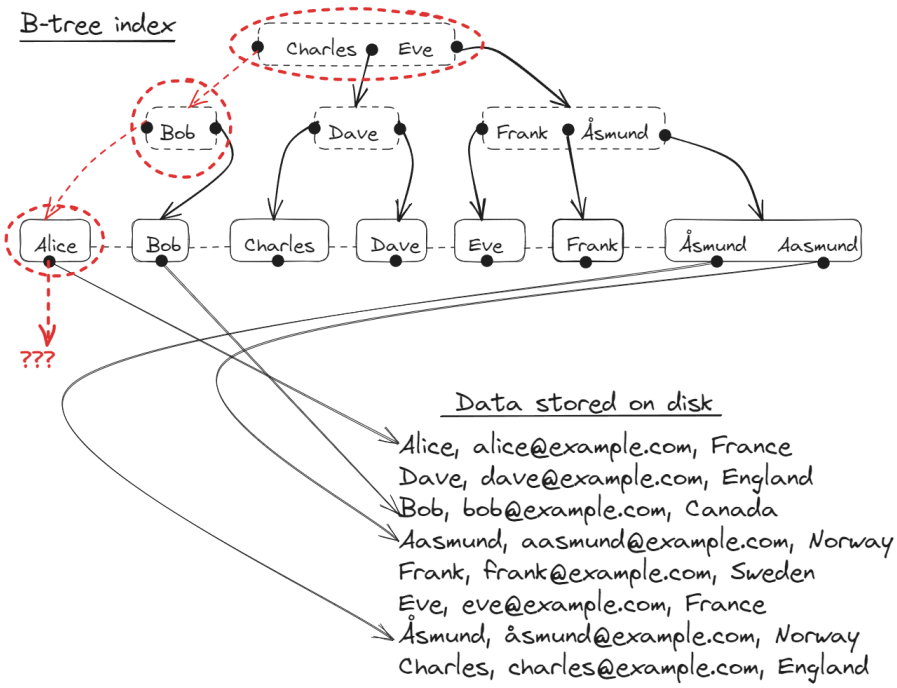


Figure 2.3: Searching for *Aasmund* in B-tree using the wrong collation. Red dotted lines indicate data read from disk or memory.

collation when searching through an index than the one used when creating it. However, it does apply directly to the scenario we are considering in this thesis. As we have discussed previously, a collation implementation with any kind of change applied to the collation order is effectively a different collation. If this new collation changes the relative order of any two strings that are currently stored in the index, the DBMS will most likely be unable to find them. This is because the DBMS will follow the pointers to the next node based on the result of the comparison operation. Whether this causes the DBMS to return an empty result set, an invalid result set, or an error depends on the implementation of the DBMS and the specific scenario.

If the collation changes in such a way that only applies to strings not currently stored in the index, the DBMS should still be able to use it without any issues. For example, this could be due to adding a new character to the collation order, as discussed in section 2.3.1. However, it is hypothetically possible that the database could somehow contain a row with a previously unassigned *code point* inside it and that this could still cause issues. Collation changes affecting the relative order of strings currently stored in the index are the most likely to cause issues and are the most interesting to consider. If such changes occur, the index should be considered corrupted or invalid, as it is no longer possible to be sure that it is retrieving the correct data. It should then be repaired or rebuilt to ensure consistency with the new collation. While this is perhaps a minor problem for a DBMS aware of the collation changes, it is a more significant problem if the DBMS does not detect the changes. Rebuilding an index can also be time-consuming, especially for large indexes. In the worst-case scenario, such changes can lead to significant downtime for the users of the database or even data loss due to invalid results. We can conclude that semantic stability is advantageous for collation implementation, as it avoids these issues.

CHAPTER 3
RELATED WORK AND STATE OF THE ART

In this chapter, we cover relevant related work in the area of collation and collation support in DBMSs. We first cover how collation is currently implemented in MySQL, which is the DBMS we will be using in our prototype. This serves as a useful comparison to our prototype and will also give the necessary context to understand the changes we make in our prototype. We then discuss recent changes in PostgreSQL, which is a similar DBMS to MySQL, and how they have added support for collation using ICU. This gives us an idea of how ICU is typically used and will also explain why this solution is not suitable for all applications. Finally, we briefly discuss other DBMSs which use ICU for collation and how they use it. This is meant to give an overview of how common it is to use ICU for collation and to list applications that could potentially benefit from the solution we will be proposing.

3.1 Collation in MySQL

As MySQL was released in the mid-1990s, it predates the release of ICU and the current maturity of the Unicode standard. Collation support for various locales is a basic requirement for a DBMS, which would have been implemented early on. Given that this predates the release of ICU, it is not

surprising that MySQL does not currently use ICU for collation. However, as ICU has steadily become more mature, stable, and popular in recent years, there have been calls for MySQL to migrate to ICU for collation support [28]. Many of the original reasons for not choosing ICU are no longer valid, and the current implementation of collation in MySQL is not without its issues.

3.1.1 Background and motivation

Currently, the main obstacle to migrating to ICU lies in the fact that changes to collation order can break indexes, which can be a serious issue for production systems. This is not a hypothetical scenario, as it has happened to MySQL in the past. In 2007, MySQL changed the collation order of the German *sharp S* character (ß) in the utf8_general_ci collation, which caused indexes to be invalidated and searches to fail [29, 30]. After this, one MySQL developer made the following remark to explain why they were unwilling to make other changes to collation order [31]:

> Why don't we just change the rules for utf8_general_ci, instead of introducing new collations with new rules? Well, as a matter of fact, that happened for another rule affecting German Sharp S in our version 5.1. The results were catastrophic, because collation affects order of keys in an index, and when index keys are out of the expected order then searches fail (Bug#37046 etc.). The only solution is to rebuild the index and when we have customers with multiple billion rows that's hugely inconvenient. This change was a stupid error, we have sworn not to repeat it.

With this in mind, it is understandable that MySQL is reluctant to risk making changes to collation orders. They wish to maintain semantic stability for all their collation implementations because the consequences for their customers would be unacceptable. Requiring users to rebuild indexes during an upgrade process would be problematic, as it could cause unpredictable downtime. Deliberate changes to collation order are, therefore, not introduced to existing collation implementations. Instead, they create new collation implementations with the desired changes and then slowly deprecate the old ones. While this does not remove the issue of

rebuilding indexes, it makes it a choice for individual users which can be planned for rather than an unexpected consequence of an upgrade.

For reasons of security and established practice, it is also not viable for MySQL to pin a specific version of ICU. Version pinning would solve the problem of semantic stability, but it would make it impossible to update ICU to fix security issues, which is unacceptable. Regardless of whether this version is bundled with MySQL or not, the problem would be the same. There is no practical way to pick and choose which updates to ICU to include either, as this would require forking ICU and maintaining a separate version. If MySQL were to use ICU for collation, they would need to find a way to solve this issue. They need to preserve semantic stability while at the same time allowing for internal changes to how ICU is implemented that could affect the collation order.

3.1.2 Implementation details

Given its age and wide use, MySQL has a large number of collation implementations covering a range of locales and use cases. These are implemented in a variety of ways, and the details of each implementation are beyond the scope of this thesis. We will primarily focus on their most modern collations, which are based on UTF-8, the Unicode Collation Algorithm (UCA), and Unicode 9.0. However, there are some commonalities between the implementations which are worth discussing.

A quirk of how collations are implemented in MySQL is the tight coupling between collation and character set. By character set, we here refer to a set of valid characters and the encoding used to represent them, such as UTF-8, UTF-16, or cp1252 [1]. Each supported character set in MySQL has a set of compatible collations. These are not shared between character sets but are implemented separately for each character set. The consequence is that users must specify both the character set and collation implementation if the default options are not suitable. There is also no guarantee that they will find a suitable collation for their character set, as not all character sets support the same collation orders.

This coupling is not an inherent requirement of collation but rather a design choice and implementation detail in MySQL. Any character set

[1] Also known as latin1.

compatible with Unicode could use the same collation implementation, with a conversion layer applied as necessary. We will do this in our prototype, as ICU uses UTF-16 internally while MySQL uses UTF-8.

Another implementation detail in MySQL worth explaining here is the naming scheme for their collation implementations. This scheme is covered in greater detail in the MySQL documentation [32], but we will provide a brief overview here. The naming scheme is relevant because we, for practical reasons, have kept to the same naming scheme in our collation implementations. Collation names in MySQL contain the name of the character set used, the locale (if applicable), and other collation properties. For instance, *utf8mb4_0900_ai_ci* is a collation for the utf8mb4 character set, based on Unicode 9.0, which is accent- and case-insensitive. This is the current default collation and is suitable for most use cases, having a similar role as the *root collation order* in ICU. As an aside, the default character set in MySQL is currently utf8mb4, and the associated collations are prefixed with utf8mb4 rather than utf8 or utf8mb3. This is due to issues in the original UTF-8 implementation in MySQL, which only supported 3 bytes per character rather than the full 4 bytes specified by the UTF-8 standard.

3.1.3 Previous debate

The debate over whether MySQL should migrate to ICU has been ongoing for several years now. In 2017, a blog post was published by a former MySQL developer discussing the possibility of migrating to ICU [28]. His conclusion was that the reasons against doing so were either no longer valid or did not outweigh the benefits of migrating. The main point left against using ICU was still the risk of changes to collation order breaking indexes [2]. This issue is still unresolved, but the author points out that ICU has been more stable in recent years, and changes are likely to occur less often. Another issue is that some Linux distributions exercise more control over which versions of ICU are available, and this has caused problems for other users in the past. However, as ICU is now more stable, this is less likely to be a concern.

Other issues discussed in the blog post are less relevant today than in the past. These include the size of the ICU library, the license required to

[2] Coincidentally, the blog author is also the one quoted in section 3.1.1.

use it, the difference in collation order between *glibc* and ICU, and whether ICU had acceptable performance. As the author points out, the size of the ICU library is no longer an issue with modern hardware, and the license has changed to one that is more permissive. The difference in collation order between *glibc* and ICU is also unlikely to be a concern, and it is only an issue of users getting different results than what they are used to from a completely different implementation. It requires benchmarking to determine how the performance of ICU compares to the current implementation in MySQL. However, as the author also points out, it need only be comparable to be acceptable.

3.2 Collation in PostgreSQL

PostgreSQL is another popular open-source relational DBMS, which serves a similar role to MySQL and is often mentioned in the same context due to their similarities. Beginning in 2017, with version 10, PostgreSQL has gradually introduced support for ICU collations. With the release of version 15 in 2022, ICU collations can now be set as the default collation for a database. Several articles have been published by the PostgreSQL team discussing the migration to ICU [33, 34], and the reasons for doing so are similar to those we have discussed for MySQL.

There are some differences in how PostgreSQL and MySQL handle collation, however, which made the transition less challenging for PostgreSQL. Chief among these is that PostgreSQL previously used *libc* as their collation implementation rather than maintaining their own. This means that the main problem we are concerned with in this thesis is one they had already lived with for many years and which had been a constant source of problems for their users [35]. By using ICU instead, they have a more stable implementation than they previously had, although it is still not guaranteed to be completely stable.

Another disadvantage of using *libc* is that it is a large and complex library, which is not designed specifically for collation. This means that keeping it updated is even more crucial than in the case of ICU because it is likely used by many other applications on the system. There was also no easy way to detect changes in collation order with *libc*, which meant that users could not be warned about potential issues with indexes. This is al-

leviated by using ICU, which provides a version number for the collation which can be checked against the version number of the collation used to create the index. Example 3.1 shows an example of this in action, where the user is warned that the collation used to create the index is older than the current version of the collation [3]. This is not a perfect solution, but still a clear improvement over the previous situation.

```
postgres=# select * from city where name between 'Aarhus' and
    'Antioch';
WARNING:  index "city_pkey" depends on collation "default"
    version "34.0", but the current version is "36.0"
DETAIL:  The index may be corrupted due to changes in sort
    order.
HINT:  REINDEX to avoid the risk of corruption.

```

Example 3.1: Error from PostgreSQL indicating changes in collation order may have occurred [34].

3.3 Other DBMSs using ICU

There are a number of other DBMSs that use ICU for collation now, either as their main collation implementation or as an alternative. Because not all of these are open-source, it is not always possible to determine precisely how they use ICU. In table 3.1, we have attempted to summarize the available information about ICU usage in these DBMSs [4]. We have attempted to indicate in the table where the information comes from and whether the DBMS uses a bundled version of ICU or the system version. This information is relevant because it indicates that there are a number of other applications which could benefit from the solution proposed in this thesis. Indexes being corrupted by changes in collation order is not a problem unique to MySQL, as the general principles of indexing are the same across DBMSs.

[3] Example reused verbatim from [34].
[4] Table reused from pre-study [5].

DBMS	Uses ICU	Bundles ICU	Source of information	Source code available
ArangoDB	Yes	Yes	Source code [36]	Yes
BigTable	Unknown	N/A	N/A	No
Cassandra	No	N/A	Issue tracker [37]	Yes
CouchDB	Yes	No	Documentation [38]	Yes
DB2	Yes	Unknown	Documentation [39]	No
EnterpriseDB	Yes	Unknown	Documentation [40]	Yes
Firebird	Yes	Partially	Unofficial [28]	Yes
MongoDB	Yes	Yes	Source code [41]	Yes
MS SQL	No	N/A	Documentation [42]	No
Oracle	Unknown	N/A	N/A	No
PostgreSQL	Yes (opt-in)	No	Documentation [33][43]	Yes
SQLite	Yes (opt-in)	No	Documentation [44]	Yes
SAP SQL Anywhere	Yes	No	Documentation [45]	No

Table 3.1: Overview of DBMSs investigated and their respective usage of ICU [5].

CHAPTER 4
IMPLEMENTATION

In this chapter, we will describe how we have implemented a semantically stable collation system using ICU. This implementation is based on adjusting the collation order used by ICU using tailoring rules, as described in section 2.4.1. We give a brief overview of what that entails and why MySQL was a suitable starting point for the prototype. Next, we describe relevant implementation details and how the prototype interacts with MySQL's existing architecture. Finally, we discuss some of the trade-offs and limitations of the prototype and how these might affect performance and usability.

4.1 Overview of the prototype

The prototype consists of a modified version of the MySQL server codebase, where the existing collation implementation has been augmented with several special collations using ICU. MySQL collations are essentially implemented as a set of structs containing a set of function pointers and other metadata about the collation, which form the pool of available collations for the server. Each new collation is based on a specific ICU collation with a given comparison level [1] and is implemented as a wrapper around

[1] See section 2.4.1.

an ICU collator object. These wrappers allow the server to call ICU functions using the same interface as the existing collation implementation. The prototype is based on the MySQL 8.0.32 source code and is available on GitHub [46].

It is not intended as a complete solution but rather as a proof-of-concept of how ICU can be used while avoiding the issues described in chapter 2. However, it should serve as a starting point for further discussion and development and as a basis for further experiments. The primary goal is to demonstrate that ICU can be used in applications that require semantic stability. A secondary goal is to show that replacing the existing collation implementation in MySQL with ICU is feasible without requiring major modification or incurring significant performance penalties.

4.2 Why use MySQL?

The underlying principles of this prototype are not specific to MySQL, and it should be possible to adapt it to other applications. Implementing the prototype as a standalone application or in a different DBMS would also have been possible. There were several reasons for choosing MySQL as a starting point for this prototype. Firstly, MySQL is a widely used, open-source DBMS written in C++. This makes it relatively easy to modify and research. Because it is written in C++, it can also use either the C or C++ interface in ICU. Secondly, MySQL currently uses a custom collation implementation. This custom implementation provided both a motivation for this thesis and a suitable comparison for the prototype. Because each locale must be implemented separately, MySQL currently only supports a limited number of locales compared to ICU. Based on this and the other reasons we discussed in section 3.1.3, it may be desirable to replace the existing implementation with ICU if possible. Finally, the collation system in MySQL is written in such a way that swapping out the existing implementation with ICU is relatively straightforward.

4.3 Collators and how they are made

In this section, we will present some simplified pseudo-code examples of how ICU might be used for collation. We will first present a simple example of how ICU is normally used for collation and then an example of how this is adapted for the prototype. The examples use the C++ API for simplicity and readability, but the same principles apply to the C API. This is meant to give the reader an idea of how ICU is used in the prototype and how it differs from how ICU is intended to be used.

Example 4.1 shows a highly simplified example of how ICU is normally used for collation. In this example, a Collator object is created for the specified locale (*nb_NO*) and then used to compare two strings. This relies on first creating a Locale object from the locale string, which contains the data needed to perform the collation operation. The key point here is that the collation data is taken directly from the ICU library rather than being stored in the application. This means that the collation data is outside of the application's control.

Example 4.2 shows a simplified example of how ICU is used in the prototype. In this example, a RuleBasedCollator object is created from a string containing the collation rules for the specified locale (*nb_NO*). Note that for brevity, only an excerpt of these rules is shown.

We first generate a prefix for our rules, compensating for changes in the root collation order. This example shows a fabricated scenario where version 67 of ICU introduced a change in the root collation order, which was to sort the cat emoji (*U+1F431*) after the dog emoji (*U+1F436*). We fix this "bug" by adding a prefix to the collation rules, which instead sorts the cat emoji before the dog emoji [2].

Note also that the hex codes, i.e., code points, for the cat and dog emojis are used here rather than the actual emojis. ICU accepts both [3], albeit in a different format. The code point *U+1F431* (or *U0001F431*) represents the character named *Cat Face*, and one possible glyph representing that is the emoji 🐱.

We then retrieve the "frozen" collation rules for the specified locale, which we have stored as a static string. Next, we concatenate it with our

[2] Because cats would insist on going first.

[3] This is because the font used does not support the emojis.

prefix and instantiate a new RuleBasedCollator object with the resulting string.

4.4 Collation operations

Our prototype essentially implements support for two collation operations, namely comparison and sort key generation. This is done by cre-

```
#include "icu_header.h" // Dummy header file to hide includes

// Each thread (connection) has its own STATUS variable.
thread_local icu::ErrorCode STATUS = icu::ErrorCode();

icu::Collator *get_collator(char *locale_string) {
  icu::Locale locale = icu::Locale(locale_string);
  icu::Collator *collator = icu::Collator::createInstance(
    locale, STATUS);

  return collator;
}

int compare_with_icu(char *s1, char *s2) {
  // Returns 0 if s1 == s2
  // Returns -1 if s1 < s2
  // Returns 1 if s1 > s2

  icu::Collator *collator = get_collator("nb_NO");

  return collator->compare(icu::UnicodeString(s1), icu::
    UnicodeString(s2), STATUS);
}

int main() {
  char *s1 = "hello";
  char *s2 = "world";
  int result = compare_with_icu(s1, s2);
  printf("result: %d\n", result); // result: -1
  return 0;
}
```

Example 4.1: Using ICU as intended by the library

ating a separate function for each operation which wraps a call to an ICU collator, instantiated as described above. When MySQL attempts to perform one of these collation operations, it references the corresponding struct containing related functions for that collation and calls the wrapper function for the operation it wishes to perform. These wrapper functions can be seen in mysql-server/strings/ctype-icu.cc.

As mentioned in section 2.4.2, ICU offers a UTF-8 interface for some operations. This interface includes comparison operations but not sort key generation. We use this interface for comparison operations and the UTF-16 interface for sort key generation. Our wrapper function for sort key generation must first convert the input string from UTF-8 to UTF-16 before calling the corresponding ICU function in the UTF-16 interface.

4.5 Implemented collations

Table 4.1 shows the collations that have been implemented in the prototype. As we will discuss in more detail in section 5.2.1, these were chosen because they were well-suited for the comparative experiments we wished to conduct. Each of them had a similar collation in MySQL, which we could use as a comparison. The similar MySQL collation was one with the same comparison strength and the same locale. The naming scheme used here is an artifact of how MySQL couples collation to character sets and is irrelevant to the prototype. The prefix *utf8mb4* is used to make the collation available for the *utf8mb4* character set, which is the character set used throughout our experiments. The suffix *icu* indicates that the collation is implemented using ICU, while the rest of the name indicates the locale and the strength of the collation.

4.6 Development flow

As we discussed in section 2.1.1, it is hypothetically possible for a developer to configure their application to support future versions of a library. However, this is not a viable option for an application like a DBMS as it would require them to support something without testing it first. This testing is essential for a DBMS like MySQL, which is used in a wide variety

Name	Locale	Strength
utf8mb4_icu_en_US_ai_ci	en_US	Primary
utf8mb4_icu_en_US_as_cs	en_US	Tertiary
utf8mb4_icu_nb_NO_ai_ci	nb_NO	Primary
utf8mb4_icu_fr_FR_ai_ci	fr_FR	Primary
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	Tertiary
utf8mb4_icu_ja_JP_as_cs	ja_JP	Tertiary
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	Quaternary

Table 4.1: ICU collations implemented, with locale and comparison strength.

of applications and environments. Because of this, when any new version of ICU is released, it would be necessary to test it and then release a new version of MySQL that officially supports it.

We, therefore, propose the development flow shown in fig. 4.1. This figure is meant to illustrate that users would not be able to upgrade to new versions of ICU without also upgrading to a new version of MySQL. However, when they upgrade, they can do so without rebuilding indexes.

If the application instead used ICU as it was intended to be used, it would still be necessary to go through a similar process. The difference would be that the check to verify index integrity would be moved to the users. Moving this work to the users means that after they upgrade to the new version of the application, they would then need to verify that the new ICU version is compatible with their existing indexes. Such a process could be automated, but it would still cause unexpected downtime for the users if a change in collation order were detected.

4.7 Limitations and simplifications

Because the implementation here is intended only as a proof-of-concept, several limitations and simplifications should be noted. We will discuss these briefly here and explain how they might be addressed in a more complete implementation. They do not affect the validity of the prototype or our results, but they should be considered if the prototype is to be

used as a basis for further development.

Firstly, the prototype is made to fit into the existing architecture of MySQL. MySQL implements collation in a way that is very different from how ICU is intended to be used, and we have made as few changes as possible to the existing code. This means that the prototype is not a *pure* implementation of ICU and that some of the benefits of ICU are not fully realized. One example is that we create an entirely separate collation struct for each combination of case-, accent- and kana-sensitivity for a given locale. ICU is designed to allow for a single collator to be used for all of these, with these combinations set by the *comparison level* of the collator [4]. Duplicating the collator for each tested combination is a simplification that does not affect performance, but it does make for a less elegant and less easily extendable solution.

Secondly, the prototype uses a simplified method for storing the tailoring data. This is provided indirectly in the form of XML files in the CLDR [5], but the prototype instead defines these as constant strings in a header file. These strings were generated by instantiating a RuleBasedCollator object with the desired Locale and then calling the getRules method to extract the tailoring data. See appendix B.1 for an example of this. The reason for using this method is that parsing the XML files is complicated and that this simplification could be made without sacrificing the validity of the prototype. In a complete implementation, it would be preferable to parse the XML files directly or store the tailoring data in another appropriate format. This additional work would allow storing the data in a structured way, similar to how it is stored in the CLDR. It would be more complex to implement, but it would make working with the data more manageable in the future.

As we discussed in section 2.4.3, the C++ API is not suited for applications that need to handle a range of ICU versions. The choice to use C++ was made before realizing this limitation, but it was a deliberate decision to keep it for the prototype. The reasons for this include time constraints, the fact that the C++ API is easier to use, and that the prototype did not need to be compatible with multiple versions of ICU. The two interfaces are otherwise equivalent, however, and the same principles apply

[4] See section section 2.4.1.

[5] As discussed in section 2.3.1, ICU customizes this data slightly.

to both. Converting the prototype or implementing similar ideas in the C API should be relatively straightforward and does not require any major changes to the architecture. For a more detailed description of the ICU C/C++ APIs, see the ICU documentation [6].

One minor issue in terms of performance is how we choose to instantiate collator objects. This is done once for each collation implementation used in thread, i.e., per client connection, and only done when the first query using that collation is executed. The effect of this is that the first query using a given collation may be slightly slower than subsequent queries. However, the effect of this was not measurable in our tests, which is why we chose to keep this implementation. If this is found to be a problem in the future, it would be possible to instantiate the collator objects when the server starts and then reuse them for each client connection. Profiling to ensure that this affects performance would be advisable before making this change, however.

[6] https://unicode-org.github.io/icu-docs/apidoc/released/icu4c/

```cpp
#include "icu_header.h" // Dummy header file to hide includes

// Each thread (connection) has its own STATUS variable.
thread_local icu::ErrorCode STATUS = icu::ErrorCode();

// Tailoring rules for "nb_NO"
static const char *ICU_NB_NO = "&å<<<Å<<aa<<<Aa<<<AA";

icu::UnicodeString get_rules(char *locale_string) {
  if (strcmp(locale_string, "nb_NO") == 0) {
    return icu::UnicodeString(ICU_NB_NO);
  } else {
    return icu::UnicodeString("");
  }
}

icu::UnicodeString get_rule_prefix() {
  if (U_ICU_VERSION_MAJOR_NUM > 67) {
    return icu::UnicodeString("&\U0001F431 < \U0001F436")
  } else {
      return icu::UnicodeString()
  }
}

icu::RuleBasedCollator *get_collator(char *locale_string) {
  auto prefix = get_rule_prefix();
  auto rules = get_rules(cs);
  auto tailoring = prefix + rules;
  return new icu::RuleBasedCollator(tailoring, STATUS);
}

int compare_with_icu(char *s1, char *s2) {
  icu::Collator *collator = get_collator("nb_NO");
  return collator->compare(icu::UnicodeString(s1), icu::
    UnicodeString(s2), STATUS);
}

int main() {
  char *s1 = "hello";
  char *s2 = "world";
  int result = compare_with_icu(s1, s2);
  printf("result: %d\n", result); // result: -1
  return 0;
}
```

Example 4.2: Using ICU with frozen collations applied as tailoring

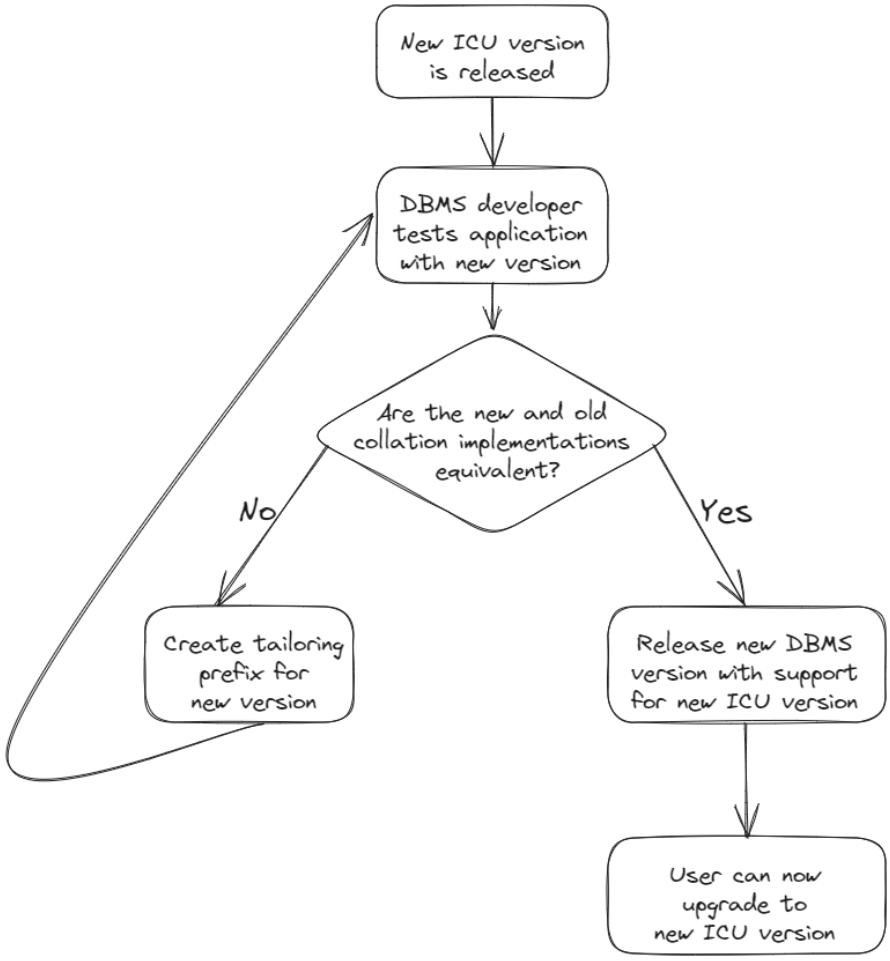


Figure 4.1: Proposed development flow with ICU

CHAPTER 5
EXPERIMENTS AND RESULTS

In this chapter, we will present the experiments we have performed to evaluate our prototype. The experiments are intended to show whether the prototype is a viable alternative to the intended way of using ICU and whether it is a viable alternative to the current implementation in MySQL. First, we will describe the experimental setup, i.e., the hardware and software used in the experiments. We will then present three separate experiments, each with their own goal. For each experiment, we will describe any setup specific to that experiment, present the results of the experiment, and then discuss the results and their implications. When appropriate, we will present relevant parts of the experiments as simplified pseudocode to make it easier to understand what was done. The code itself is available on GitHub, which should also make it possible to reproduce the results gathered here. Finally, we will discuss the general limitations of the experiments carried out here, followed by a summary of the chapter and the conclusions we can draw from the results.

5.1 Experimental setup

For the results used in this report, the two performance experiments are carried out on a dedicated computer to minimize the risk of other pro-

cesses and setup quirks interfering with the results. This computer was a Lenovo ThinkCentre M920t, with an Intel i7-8700 CPU[1] and 32 GiB of RAM. The operating system used is Ubuntu 22.04.2 with kernel version 5.19.0–43-generic. We use the default version of ICU available from the package manager, which in this case is version 70.1-2 of *libicu-dev*. As discussed in section 4.1, the builds of MySQL used in our tests are based on MySQL 8.0.32 source code. The source code was compiled using version 11.3.0 of gcc/g++, with the -O3 optimization flag enabled.

A small suite of Python tools was written to orchestrate the experiments [47]. This suite includes a command-line interface (CLI) to connect to the running MySQL server, prepare the database, run the queries, and collect the results. Collected results are written to a local SQLite database, which is used to generate graphs and tables.

5.1.1 Building MySQL

A custom bash script was created to build MySQL [2] to simplify the process of re-running the experiments. With this script, we ensured the same configuration was used for each run. This script sets up the correct folder structure and generates the necessary CMake command to build MySQL from the source code. While this is not overly complex, having a script to ensure that the same build configuration was used for all experiments was helpful. This was especially useful when developing and testing the project with multiple computers.

Some of the experiments carried out here are designed to test different configurations of the prototype. Three configurations are being evaluated here for both the performance benchmarks and the validity tests. For the sake of simplicity, these configurations are hard coded in the prototype as boolean constants, and the prototype is built three times with different configurations. The values of these variables do not affect the existing MySQL collation code, only the ICU-based collations added by the prototype. These configurations are set in the file mysql-server/strings/ctype-icu.h, and the values of the constants are listed in table 5.1.

The first configuration is the *default* configuration, where the proto-

[1] 6 cores, 12 threads, and 3.2 GHz base clock speed.

[2] https://github.com/LarsV123/master-util/blob/master/build.sh

type uses the default ICU Collator class, and collation data is loaded from a Locale object. This means the prototype is performing collation more or less the same way as ICU is intended to be used. Notably, this means that the collation implementation is not semantically stable across versions of ICU, as discussed in section 2.4.3. It is included to compare performance and collation order with the other configurations.

The second configuration is the *frozen* configuration, where the prototype uses a RuleBasedCollator, and the collation data is loaded from static strings. These strings are generated by extracting the collation rules from a Locale-based collator instantiated with the desired locale. As discussed in section 4.3, this should result in a collator with the same behavior as a Locale-based collator. The difference is that this version can be made to be semantically stable across versions of ICU by adjusting the collation rules to compensate for changes in ICU behavior.

The final configuration is the *tailored* configuration. This is the same as the frozen configuration, except that the collation rules are modified to move 100 characters to the end of the collation order. The purpose of these is to simulate the effect of adding additional rules, such as those discussed in section 2.4.3. This configuration is included to test the performance of the prototype with additional rules applied.

Configuration	**ICU_FROZEN**	**ICU_EXTRA_TAILORING**
Default	false	false
Frozen	true	false
Tailored	true	true

Table 5.1: Three different configurations of MySQL built for the experiments

5.2 Experiment 1: Performance benchmarks

This experiment is intended to look at performance differences, if any, between our prototype implementations and the current implementation in MySQL. The benchmark is designed to test performance in a *worst-case*

scenario, i.e. in a situation where the effect of collation performance is maximized. By performance, we here mean the time it takes to perform the collation operations, without regard for memory usage or other resources. This is here measured by the total time taken from the start of a query until the result is returned to the client. The scenarios being tested are queries that force the use of collation on a large number of unindexed rows. This should be the worst-case scenario for collation performance, i.e. the case where the effect of collation performance is maximized.

The intention is to test whether our prototype is significantly slower than the current implementation in MySQL and whether it is a viable alternative. The prototype will be tested in the three different configurations listed above, as well as the current implementation in MySQL, and the results compared to each other. This should indicate whether the use of the ICU library itself or the alternative way it is being used in the prototype is the cause of any performance differences.

5.2.1 Setup

The experiments done in the pre-study indicated that there could be significant performance differences in collation, depending on the locale of both the collator being used and the data being collated. Because of this, it was decided to test each non-English collation on data in both its native language and in English. Also, a set of collations were chosen to represent a variety of different languages and scripts, each with unique features which could potentially affect performance.

One complicating factor was that each ICU-based collation should be tested against a semantically equivalent MySQL collation to have comparable results. Because the performance of a collation is affected by the complexity of the rules it uses, and there is no guarantee that the rules used by ICU and MySQL are identical, differences here could potentially affect the results. However, we found many cases where both an ICU and a MySQL collation purported to be for the same locale and to have the same case- and accent sensitivity. A set of such pairs were chosen for this experiment, as these should have similar performance characteristics, while collations where we could find no such pair were excluded. For example, Thai (th_TH) is not supported by any MySQL collation, so we were unable to test it. In other cases, there were either case- and accent-sensitive or

case- and accent-insensitive collations for a locale in MySQL, while the ICU collations support both. Lack of collation support in MySQL is part of the motivation for wanting to use ICU in the first place, and this problem is therefore not unexpected.

In the end, a set of 7 ICU collations were chosen for this experiment, as presented in table 5.2. Which language each locale ID represents is shown in table 5.3, but for brevity, this report will generally refer to the locales by their IDs. Reasons for including this specific set of collations include both suspected performance differences and the desire to test a variety of different languages and scripts. For instance, Chinese and Japanese were chosen in part because they require more bytes per character than most other languages, and this could potentially affect performance. Japanese is also unusual in that it has a *quaternary* level of collation, i.e. a level of collation which is *kana sensitive* [3]. This is supported by MySQL, but the additional complexity could also potentially affect performance. Also, several of the tested collations are included as controls, because there is no reason to expect any significant performance differences between them. For *en_US*, two versions are tested, one case- and accent-sensitive and one case- and accent-insensitive, to test the effect of the case- and accent-sensitivity. Also, a collation for *fr_FR* is included as a control, because it should in theory be identical to *en_US*. Both of these should be identical to the root collation in the ICU library, which is expected to have the best performance of all the collations tested as other collations build on this. Finally, *nb_NO* is included because it is similar to *en_US*, but with minor changes to the alphabet and collation rules.

The data set forming the basis for this experiment is a collection of 249 location names in various languages [48]. This was chosen to have a set of real-world data with equivalents in each language we were interested in testing. For example, when testing Japanese collation, it is more realistic to have a set of Japanese location names than a set of random strings. It was also important to have a large enough data set to make the benchmark meaningful, so the data set was synthetically expanded by adding a number to each name. By doing this, three separate data sets were formed for each locale, containing 500K, 1000K, and 2500K names respectively. The size of each generated data set is shown in table 5.4. It

[3] See section 2.4.1.

ICU collation	MySQL collation	Locale
utf8mb4_icu_en_US_ai_ci	utf8mb4_0900_ai_ci	en_US
utf8mb4_icu_en_US_as_cs	utf8mb4_0900_as_cs	en_US
utf8mb4_icu_nb_NO_ai_ci	utf8mb4_nb_0900_ai_ci	en_US
utf8mb4_icu_nb_NO_ai_ci	utf8mb4_nb_0900_ai_ci	nb_NO
utf8mb4_icu_fr_FR_ai_ci	utf8mb4_0900_ai_ci	en_US
utf8mb4_icu_fr_FR_ai_ci	utf8mb4_0900_ai_ci	fr_FR
utf8mb4_icu_zh_Hans_as_cs	utf8mb4_zh_0900_as_cs	en_US
utf8mb4_icu_zh_Hans_as_cs	utf8mb4_zh_0900_as_cs	zh_Hans
utf8mb4_icu_ja_JP_as_cs	utf8mb4_ja_0900_as_cs	en_US
utf8mb4_icu_ja_JP_as_cs	utf8mb4_ja_0900_as_cs	ja_JP
utf8mb4_icu_ja_JP_as_cs_ks	utf8mb4_ja_0900_as_cs_ks	en_US
utf8mb4_icu_ja_JP_as_cs_ks	utf8mb4_ja_0900_as_cs_ks	ja_JP

Table 5.2: Combinations of ICU collation, MySQL collation, and data locale tested.

is worth noting that these differ significantly from language to language, with the Japanese data set being the largest by far. This should not affect the results, as the collations being compared against each other will be tested on the same data set.

Each test case runs three queries, testing different aspects of collation performance. These test both the comparison operation (equality), as well as the ORDER BY clause in both ASCENDING and DESCENDING order. There is no obvious reason why there should be a meaningful difference in performance between the two orderings, but it was included because preliminary tests indicated that there might be. The benchmark tests both comparison and sorting because these are fundamentally different operations, which are implemented differently [4]. Example 5.1 shows an example of the queries tested. The queries are fairly simple, but they are designed with the single purpose of forcing the MySQL server to perform a full table scan, iterating over every row in the table, and performing a collation operation for each row. Because there is no index created on the column being sorted, the only way to execute the queries is to iterate over

[4] See section 4.4.

Locale	Language
en_US	English (US)
nb_NO	Norwegian (Bokmål)
fr_FR	French
zh_Hans	Chinese (simplified)
ja_JP	Japanese

Table 5.3: Locale IDs with the corresponding language.

the entire table.

```
1  -- Test case: ORDER BY, ascending order, 1M rows, en_US locale
2  SELECT *
3  FROM test_en_US_1000000
4  ORDER BY `value`
5  COLLATE utf8mb4_icu_en_US_ai_ci ASC
6  LIMIT 1;
7
8  -- Test case: Comparison, 2.5M rows, nb_NO locale
9  SELECT * FROM test_nb_NO_25000000
10 WHERE `value` = 'Norge123'
11 COLLATE utf8mb4_icu_nb_NO_ai_ci;
```

Example 5.1: Example of queries used in performance benchmark

To summarize, we are testing 7 different ICU collations under 3 different build configurations, as listed in table 5.1. Each of these is compared with their closest equivalent in MySQL, for a further 6 collations [5]. The non-English collations are also tested on both data from their native locale and from the *en_US* locale. Each combination is tested on each of the 3 data sets described above. This gives us a total of $((7+5)*3+6+5) = 47$ different test cases, which are generated programmatically [6].

[5] The same MySQL collation covers both en_US and fr_FR.

[6] https://github.com/LarsV123/master-util/blob/master/src/benchmarks.py

Table	Size in MiB
test_ja_JP_2500000	246.81
test_fr_FR_2500000	222.77
test_zh_Hans_2500000	220.8
test_en_US_2500000	208.77
test_nb_NO_2500000	200.77
test_ja_JP_1000000	107.64
test_zh_Hans_1000000	102.64
test_fr_FR_1000000	98.64
test_nb_NO_1000000	96.64
test_en_US_1000000	94.64
test_ja_JP_500000	68.58
test_fr_FR_500000	64.58
test_zh_Hans_500000	61.59
test_en_US_500000	58.58
test_nb_NO_500000	58.58

Table 5.4: Data sets used in experiment 1, sorted by size.

5.2.2 Data collection and processing

Each test case in our benchmark suite is 11 times in sequence. The first run is used as a *warm-up* and the results are discarded. This is done to ensure consistent effects from caching and to avoid the first run being an outlier. Therefore, we get 10 recorded results for each test case from each run of the benchmark. In total, this produces $47 * 10 * 3 * 3 = 4230$ rows of data, which are then processed and analyzed using a Python script [7]. The raw results from this experiment are also available on GitHub [8].

The actual metric used for these results is the median execution time across all runs. In this experiment we repeat a deterministic operation multiple times, where the DBMS is forced to execute a table scan and perform the same collation operations every time. It is therefore reasonable to assume that any variation in the results is caused by external factors, such as other processes running on the machine, or the operating system scheduling threads differently. For this reason, we have chosen to use the median as the metric for the results, as this is less affected by outliers than the mean. We also calculate the standard deviation, to get an idea of the variance in the results. Both the median and standard deviation are presented in seconds.

Finally, we have selected one of the MySQL test cases as a baseline, to have a basis for comparison for the other combinations. This difference is calculated as the difference between the medians of the current measurement and the baseline, divided by the median of the baseline. The baseline collation, *utf8mb4_0900_ai_ci*, was selected because it is the default collation in MySQL. This was tested using the *en_US* locale data, i.e. the list of location names in English. The difference is presented in the tables as Δ *baseline*, as a percentage relative to the baseline. For all three measures, lower numbers are better and indicate improved performance.

5.2.3 Results

The results of this experiment have been processed and aggregated to make them easier to interpret. Given the number of different test cases, it is not feasible to present all the results in this section. We will therefore

[7] https://github.com/LarsV123/master-util/blob/master/src/experiment2.py
[8] https://github.com/LarsV123/master-util/blob/master/experiments.db

present an overview of the results, and refer to appendix A.1 for the full results. Instead, we will present excerpts from the results, showing a set of tables that are representative of the results as a whole. We will also present a set of graphs that show the results visually, before discussing the results in more detail.

In fig. 5.1 we see the median execution time per 100 000 rows for the ORDER BY ASC operation across all data set sizes. The median is calculated across all test cases (collation, data locale and configuration), and only split on the size of the data set used. This shows that the size of the data set has no meaningful effect on the performance of the operation. Because of this, all further results presented here are based on the medium-sized data set.

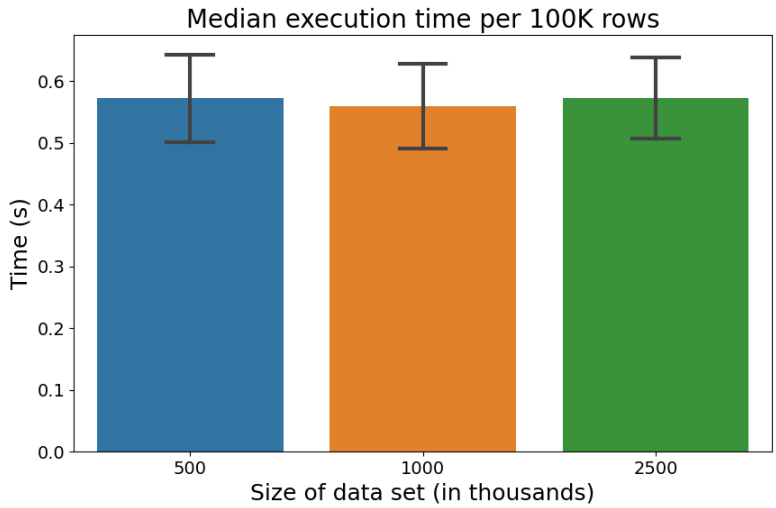


Figure 5.1: Execution time for the ORDER BY ASC operation across all measurements split by data set size. Lower execution time is better. Error bars show standard deviation.

The tables we have chosen to present here show how our main ICU implementation compares to the current MySQL implementation. Performance for the ORDER BY clause in ASCENDING order are shown in table 5.6 and table 5.5, for the ICU and MySQL implementations respectively. As

discussed in section 4.4, this is implemented by generating a sort key for each row, and then sorting the rows based on the sort key. Sorting tables based on a string column is a common use case for collation, and is therefore important to optimize. The performance effect of collation operations can make a significant impact when used on large data sets, and the results show that there is a meaningful difference between the implementations. These results indicate that the MySQL and ICU implementations perform differently based on the complexity of the collation. For more complex collations, such as those for Japanese and Chinese, the ICU implementation is consistently faster than the equivalent MySQL implementation. For the simpler collations, such as *nb_NO_ai_ci*, the difference is negligible.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_0900_ai_ci	en_US	4.76	0.05	0
utf8mb4_0900_as_cs	en_US	5.36	0.04	12.67
utf8mb4_ja_0900_as_cs	en_US	7.16	0.05	50.43
utf8mb4_ja_0900_as_cs_ks	en_US	7.31	0.27	53.6
utf8mb4_nb_0900_ai_ci	en_US	5.44	0.07	14.23
utf8mb4_zh_0900_as_cs	en_US	7.66	0.17	61
utf8mb4_0900_ai_ci	fr_FR	5.33	0.15	11.97
utf8mb4_ja_0900_as_cs	ja_JP	7.25	0.21	52.27
utf8mb4_ja_0900_as_cs_ks	ja_JP	7.69	0.17	61.67
utf8mb4_nb_0900_ai_ci	nb_NO	5.44	0.1	14.38
utf8mb4_zh_0900_as_cs	zh_Hans	6.34	0.19	33.28

Table 5.5: ORDER BY ASC for all MySQL collations.

In table 5.8 and table 5.7 we show the performance for the comparison operation, i.e. the scenario where two values are compared directly for equality. While there is some variation in the results, the performance is generally very similar between equivalent collations in the two implementations. As we mentioned in section 2.4.2, this operation is implemented with the UTF-8 API in ICU. This means that the ICU implementation is not at the disadvantage of having to convert to UTF-16 like it must

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.06	0.08	6.32
utf8mb4_icu_en_US_as_cs	en_US	5.03	0.09	5.64
utf8mb4_icu_fr_FR_ai_ci	en_US	5.5	0.16	15.54
utf8mb4_icu_ja_JP_as_cs	en_US	5.71	0.09	20.08
utf8mb4_icu_ja_JP_as_cs_ks	en_US	6.26	0.08	31.48
utf8mb4_icu_nb_NO_ai_ci	en_US	5.24	0.14	10.11
utf8mb4_icu_zh_Hans_as_cs	en_US	5.58	0.1	17.18
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.36	0.13	12.53
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.09	0.21	28.06
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.32	0.24	32.89
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.21	0.12	9.5
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.65	0.2	18.63

Table 5.6: ORDER BY ASC for all ICU_frozen collations.

for the ordering operation. Before the experiments were run, this was hypothesized to potentially be a significant factor in the performance difference between the two implementations. However, as we will see in section 5.3, this does not appear to be the case.

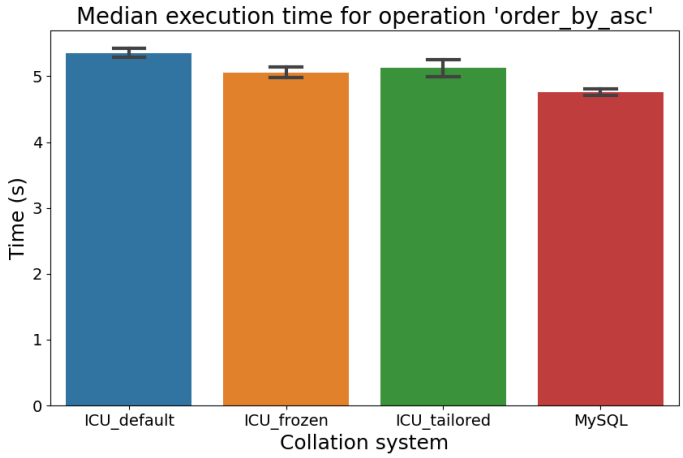
While the tables we have discussed so far show execution time for all test cases for the ICU_frozen and MySQL configurations, we have not yet discussed the results for the ICU_default and ICU_tailored configurations. As discussed in section section 4.3, these represent both the intended way of using the ICU library, as well as the way our prototype is likely to be used in practice if there are major changes to ICU collation. Figure 5.2 and fig. 5.2 show a subset of the tested collation and data locales for all four configurations. Here we present the results for the ordering operation in ascending order. Similarly, in fig. 5.4 and fig. 5.5 we see the same subset of collation and data locales, but for the equality comparison. The subset of locales shown here is selected to represent both the simplest case (English), a slightly more complex case (Norwegian), and the two most complex cases (Japanese and Chinese). These collations are all accent-, case- and (where applicable) kana-insensitive.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_0900_ai_ci	en_US	4.64	0.04	0
utf8mb4_0900_as_cs	en_US	4.51	0.04	-2.87
utf8mb4_ja_0900_as_cs	en_US	5.74	0.04	23.58
utf8mb4_ja_0900_as_cs_ks	en_US	5.73	0.27	23.4
utf8mb4_nb_0900_ai_ci	en_US	4.81	0.03	3.52
utf8mb4_zh_0900_as_cs	en_US	5.69	0.04	22.54
utf8mb4_0900_ai_ci	fr_FR	5.05	0.13	8.62
utf8mb4_ja_0900_as_cs	ja_JP	5.77	0.17	24.32
utf8mb4_ja_0900_as_cs_ks	ja_JP	5.76	0.22	24.04
utf8mb4_nb_0900_ai_ci	nb_NO	4.77	0.11	2.74
utf8mb4_zh_0900_as_cs	zh_Hans	5.64	0.19	21.35

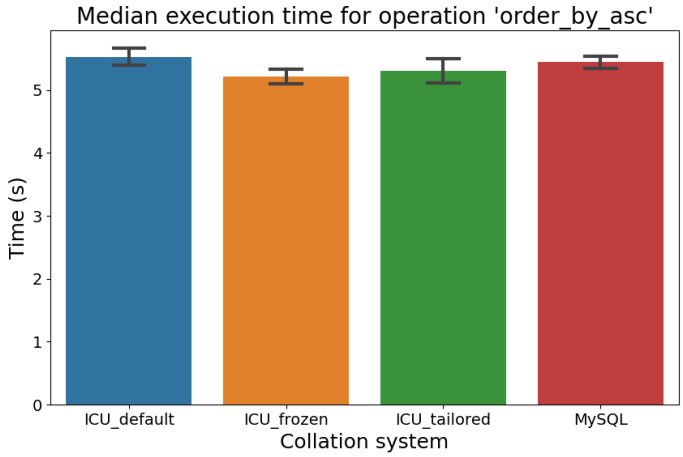
Table 5.7: Equality comparison for all MySQL collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	4.4	0.02	-5.29
utf8mb4_icu_en_US_as_cs	en_US	4.62	0.02	-0.55
utf8mb4_icu_fr_FR_ai_ci	en_US	5.01	0.03	7.86
utf8mb4_icu_ja_JP_as_cs	en_US	5.65	0.02	21.61
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.66	0.02	21.75
utf8mb4_icu_nb_NO_ai_ci	en_US	4.73	0.02	1.78
utf8mb4_icu_zh_Hans_as_cs	en_US	5.32	0.02	14.6
utf8mb4_icu_fr_FR_ai_ci	fr_FR	4.82	0.12	3.79
utf8mb4_icu_ja_JP_as_cs	ja_JP	5.51	0.18	18.58
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	5.71	0.16	22.87
utf8mb4_icu_nb_NO_ai_ci	nb_NO	4.66	0.02	0.29
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.17	0.11	11.4

Table 5.8: Equality comparison for all ICU_frozen collations.

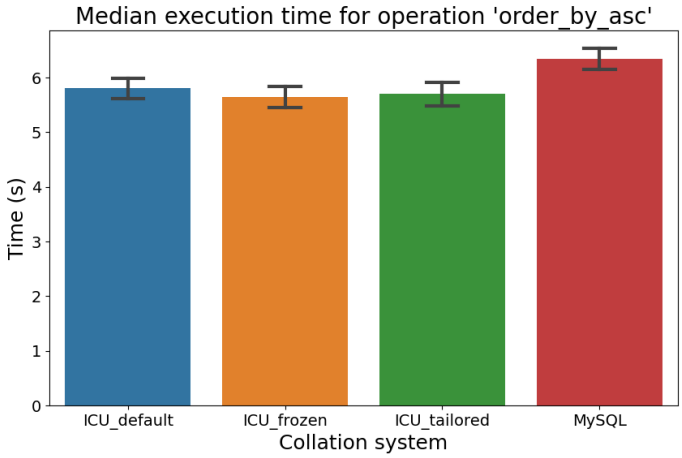


(a) English (en_US)

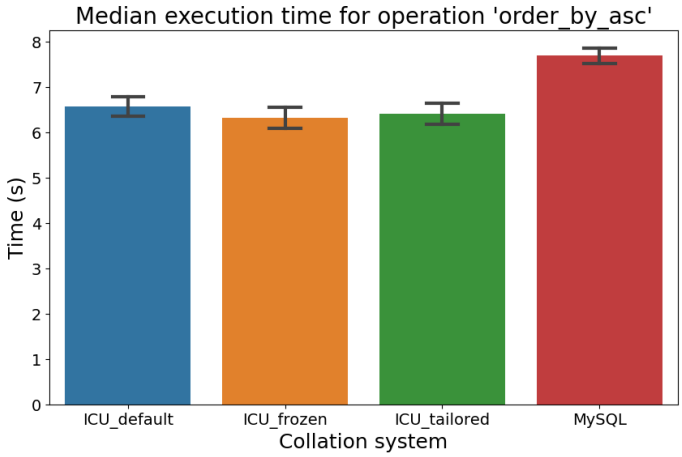


(b) Norwegian Bokmål (nb_NO)

Figure 5.2: Execution time for the ORDER BY ASC operation across selected collations. Lower execution time is better. Error bars show standard deviation.



(a) Simplified Chinese (zh_Hans)



(b) Japanese (ja_JP)

Figure 5.3: Execution time for the ORDER BY ASC operation across selected collations. Lower execution time is better. Error bars show standard deviation.

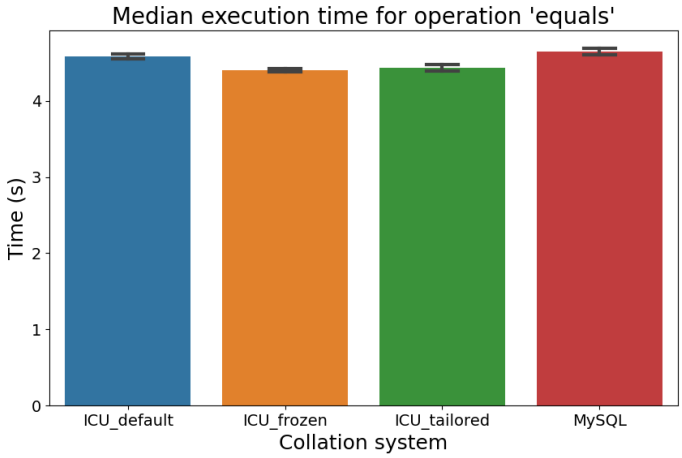
These results show that all three variations of the ICU implementation have similar performance for both the equality comparison and the ordering operation. The ICU_default configuration performs slightly worse than the other two variations, which is unexpected and difficult to explain based on the implementation. However, the difference is small and it is difficult to draw any conclusions from this.

5.2.4 Summary

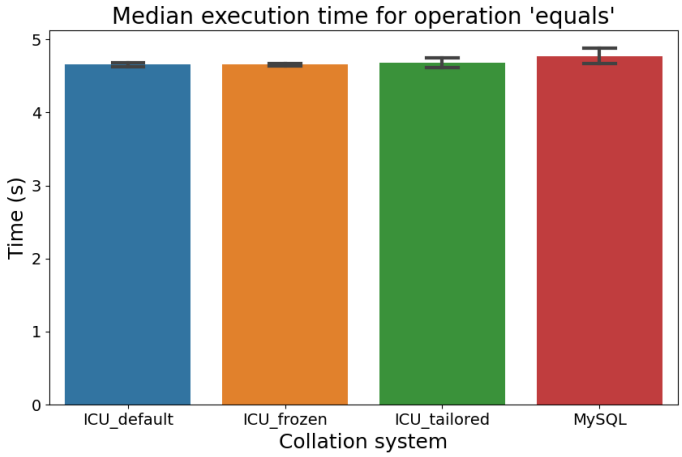
The main goals of this experiment were to determine whether the prototype implementation was significantly slower than the current implementation in MySQL and whether making our implementation semantically stable incurred a performance penalty. Before running this experiment, we had several assumptions and expectations about the results. On the one hand, there was reason to think ICU could perform better than MySQL because it is a specialized library for collation. On the other hand, using ICU required an extra conversion step, which might be costly [9]. Our prototype implementation might also be flawed in some way, which could lead to worse performance. There was also reason to think that the ORDER BY operations could be slower because the ICU documentation specifically recommends against using sort keys if performance is a concern.

However, our results do not indicate a significant performance penalty for our semantically stable prototype implementations. They have similar performance to the default ICU implementations they are based on and, in some cases, perform significantly better than the current MySQL implementations. The results indicate that ICU-based collations have an even performance profile and are less affected by the complexity of the collation rules than those in MySQL. For the most complex collations, such as Japanese, the performance of the ICU-based collations is significantly better than the MySQL-based collations. Adding additional tailoring rules to ICU collations also does not seem to impact performance.

[9] More on this in section 5.3

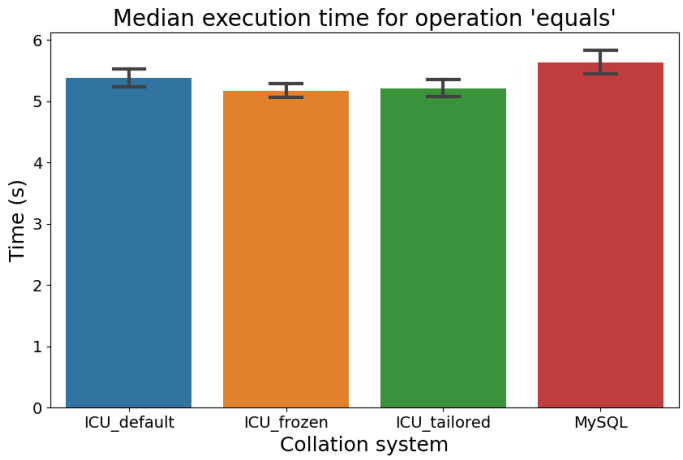


(a) English (en_US)

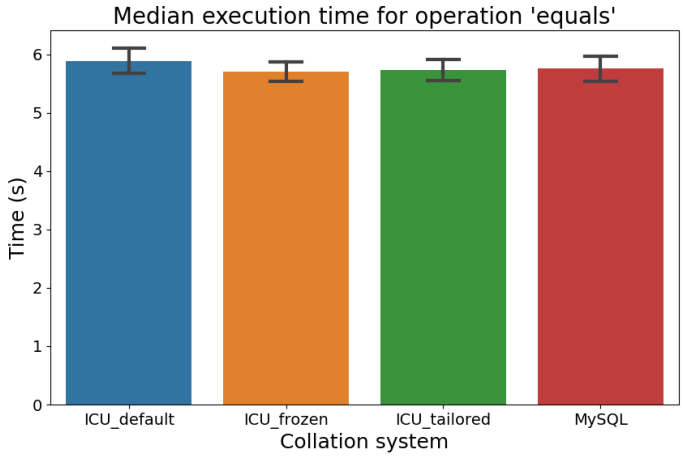


(b) Norwegian Bokmål (nb_NO)

Figure 5.4: Execution time for the equality comparison operation across selected collations. Lower execution time is better. Error bars show standard deviation.



(a) Simplified Chinese (zh_Hans)



(b) Japanese (ja_JP)

Figure 5.5: Execution time for the equality comparison operation across selected collations. Lower execution time is better. Error bars show standard deviation.

5.3 Experiment 2: Flame graph comparison

The goal of this experiment is to compare the performance of the ICU and MySQL implementations in a more fine-grained way than the benchmark above. The benchmark above only measures the time it takes to perform the collation operations, but does not provide any insight into what is happening during that time. In particular, it does not provide any insight into how much time is spent in the ICU library itself, and how much time is spent in the MySQL code. This experiment is intended to provide that insight, by recording activity during execution with the *perf* utility and visualizing it. This is done by generating so-called flame graphs, which are visualizations of the activity of the CPU during execution. The flame graphs are then compared to each other, to determine whether there are any significant differences in the execution of the ICU and MySQL implementations. Rather than a comparison of total execution time, this will allow us to see how the time is spent, and whether there are any significant differences here.

One particular concern raised in the pre-study was that MySQL stores and processes data in UTF-8, while the ICU library uses UTF-16 internally and only has UTF-8 support for certain operations. Notably, the generation of sort keys requires conversion from UTF-8 to UTF-16, and this conversion was feared to be expensive. To measure the impact of this, the actual conversion is split into a separate function in the prototype, which is then called in the parent collation function. The flame graphs therefore, should allow us to see whether conversion is an expensive operation and whether it is a significant part of the execution time.

5.3.1 Setup

The basic idea behind flame graphs is to record the activity of the CPU during execution, and then visualize this activity as a graph where each block represents the portion of CPU time spent within a given function. This is achieved here by using the *perf* utility to record the activity of the CPU and then using the *FlameGraph* tool to visualize the results. The *perf* utility is a Linux tool for performance analysis, which can be used to record a variety of different events.

Example 5.2 shows the actual query executed, which is a simplified

version of the ORDER BY query used in the benchmark in the previous section. This is run three times in a row for each configuration (with the appropriate collation inserted), which with this hardware and setup causes the MySQL server to work for roughly 15-20 seconds. The reason for running the query three times is to ensure that the results are consistent, to limit the effect of caching, and to ensure that the query is run for a long enough time to be able to record a sufficient amount of activity. Also, the same query is run once before the actual test and start of the recording, to ensure that the collation data is loaded into memory and that the collator is instantiated.

```
SELECT *
FROM test_en_US_1000000
ORDER BY value COLLATE %s
LIMIT 1;
```

Example 5.2: Example of query used in flame graph experiment

This process is orchestrated using a bash script, to simplify the process and make it easy to reproduce consistent results. First, it calls on *perf* to record the activity of the CPU for a specified number of seconds. This listens only to events related to the MySQL process and records the activity in a file. Then, in a separate thread, it calls on a Python script to execute a specific SQL query against the MySQL server. This script executes a short series of SQL queries, which are intended to be representative of the type of real-world query which would heavily depend on collation performance. While this is a contrived scenario to focus on collation performance, it is also not entirely unrealistic, as queries on unindexed text columns are not uncommon in real-world applications.

Given the number of different collations and data sets, only a subset of the configurations from the previous experiment are used here. Because the goal is to compare the performance of our prototype to MySQL, the ICU configuration used here is the same as the one used in the benchmark in the previous section, i.e. ICU_frozen. Also, each collation is only tested with the data set native to that locale and only with the medium-sized versions of the data sets (1000K rows). Given the results from experiment 1, this should be sufficient to provide a representative comparison of the

performance of the two implementations.

Collation	Data set
utf8mb4_icu_en_US_ai_ci	en_US (1000K)
utf8mb4_0900_ai_ci	en_US (1000K)
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP (1000K)
utf8mb4_ja_0900_as_cs_ks	ja_JP (1000K)

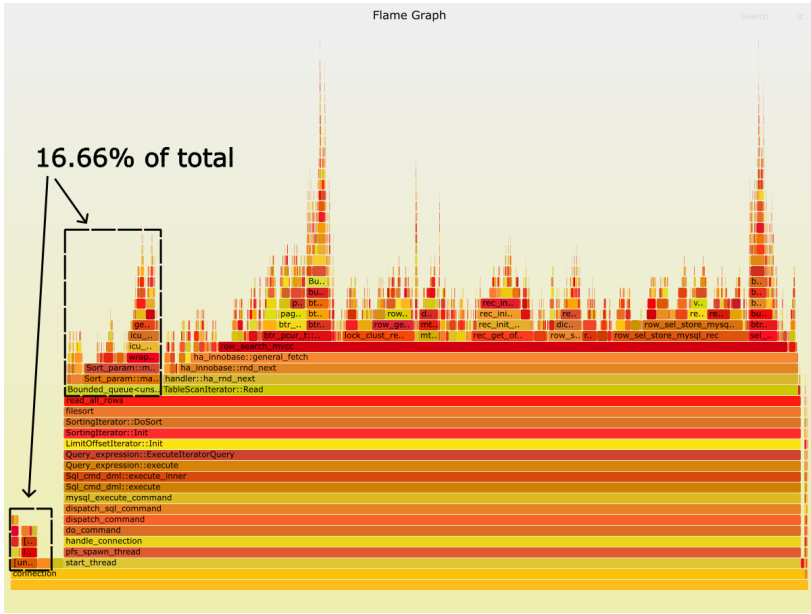
Table 5.9: Collations and data sets used for generating flame graphs.

The raw results from this experiment consist of 14 different flame graphs, in the Scalable Vector Graphics (SVG) format. These are interactive and can be viewed in a web browser where reading them is significantly easier. Here a search feature is also available, which can be used to highlight specific functions by name. Because of this, the full set of flame graphs are attached to the report and are also available on GitHub [10]. A subset of these will be presented here, to illustrate the results of the experiment. This subset is chosen to show the most interesting results and to show the difference between the ICU and MySQL implementations. The subset consists of the two pairs of collation and data locale shown in table 5.9, which are respectively the simplest and most complex test cases available. Other configurations are omitted because they are either very similar to one of these or fall in between them in terms of time spent on collation. For the sake of readability in printed format, the included flame graphs are annotated with boxes, arrows, and text to indicate which portion of the graph is collation-specific code. The recorded data includes all CPU activity in the server during the execution of the queries, which is not all specific to collation performance.

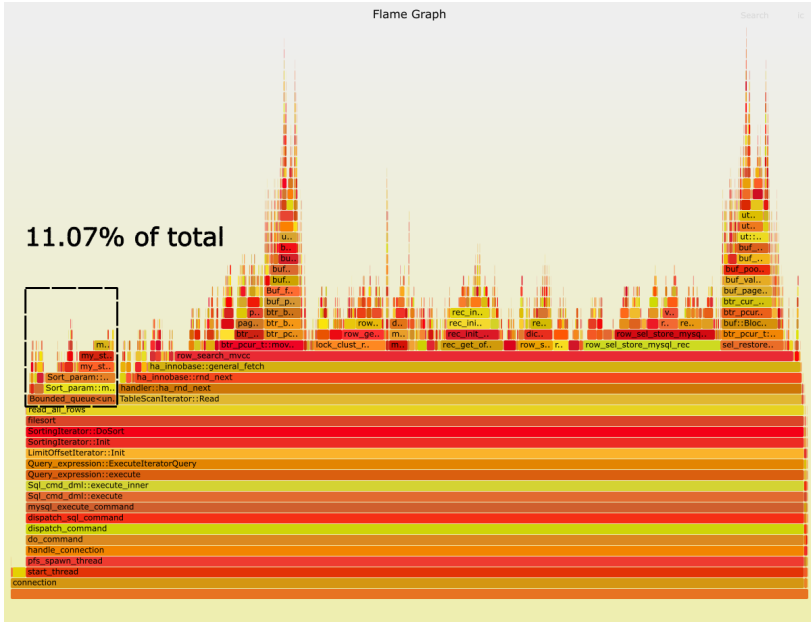
5.3.2 Results

The results of this experiment are shown in fig. 5.6 and fig. 5.7. Here we can see that the ICU implementation spends significantly more time on collation functions than the MySQL implementation using the default collation, indicating that the ICU implementation is less performant in this

[10] https://github.com/LarsV123/master-util/tree/master/results/experiment2

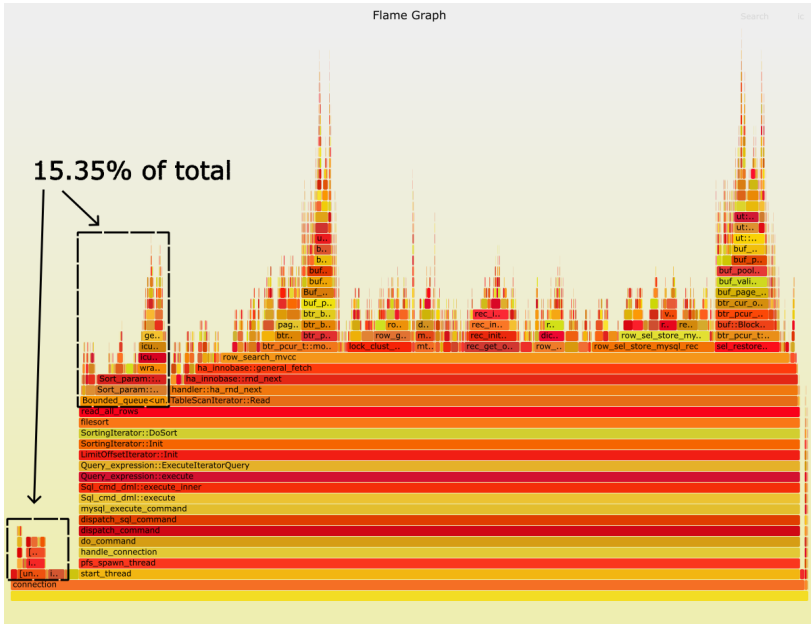


(a) ICU (utf8mb4_icu_en_US_ai_ci)

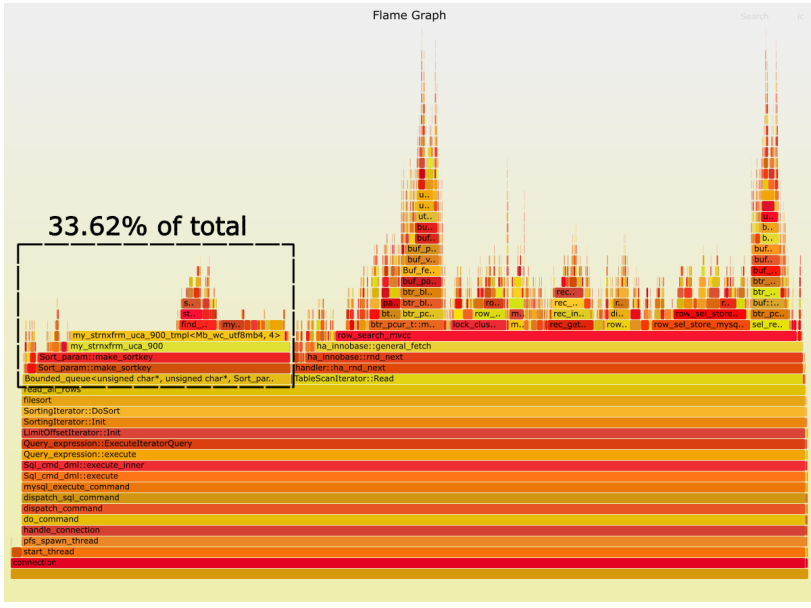


(b) MySQL (utf8mb4_0900_ai_ci)

Figure 5.6: Flame graphs comparing execution of ORDER BY queries for *en_US* collations. Annotations indicate collation-specific functions.



(a) ICU (utf8mb4_icu_ja_JP_as_cs_ks)



(b) MySQL - Japanese (ja_JP)

Figure 5.7: Flame graphs comparing execution of ORDER BY queries for *ja_JP* collations. Annotations indicate collation-specific functions.

case. This contradicts the results from the previous experiment, where the ICU collation was only slightly slower in the same scenario. It is difficult to speculate as to the exact reason for this difference, as these experiments measure slightly different performance metrics. While the previous experiment measure the total execution time for a query (what is sometimes called *wall-clock time*), this experiment measures time spent on computation by the CPU. Also, while the experiments were done on the same computer and with the same data, the measurements were done at different times, and there may have been other processes running on the computer which could have affected the results. Some degree of variation here is therefore to be expected.

However, the MySQL implementation using the Japanese collation spends significantly more time on collation functions than the ICU implementation using the same collation. Here we can see more clearly that while the ICU implementation is consistent in its total use of CPU time, the MySQL implementation spends significantly more computational resources on Japanese collation. In this specific scenario, the slowest MySQL collation spent roughly three times as much CPU time on collation as the fastest MySQL collation. This could indicate that the ICU implementation is more generic and will generally have similar performance regardless of the specific collation used, while the MySQL collations vary more in their performance characteristics. This is consistent with the results from experiment 1, where the ICU implementation was generally faster than the MySQL implementation for more complex collations.

As we can see in the flamegraphs for the ICU collations, or rather what we cannot see, is that the conversion from UTF-8 to UTF-16 is not a significant part of the execution time. It is such a small portion of total execution time that is not visible in the graph, generally taking up less than 0.2% of the total [11]. Regardless of other results, this at least shows that having to convert between these formats should not cause any significant performance issues. This resolves one of the questions we had at the start of the project, namely whether the need to convert to UTF-16 would be problematic in an application based on UTF-8.

[11] Search for *convert_utf8_to_utf16* in the SVG file to find it.

5.3.3 Summary

The key takeaways from this experiment are that the ICU implementation is less affected by the specific collation used than the MySQL implementation and that the conversion from UTF-8 to UTF-16 is not a significant part of the execution time. For complex collations, the difference in performance can be significant, while for simpler collations the difference is less pronounced and not always in favor of ICU. This is consistent with the results from the previous experiment and indicates that the ICU implementation is generally preferable to the MySQL implementation for complex collations. Another interesting point is that collation-specific work can take up a significant amount of CPU resources for certain queries. Queries that require ordering a large number of unindexed string values according to a complex collation can therefore benefit significantly from using a more performant collation implementation [12].

5.4 Experiment 3: Validity checks

This experiment has two main goals, which partially overlap in their aim and which we hope to solve with a single experimental setup. First, we need a method to detect changes in collation order, which we can use to verify that the ICU root collation has not changed between different versions of the ICU library. As discussed in section 2.4.3, such changes are one of the main concerns with this project, and we need a way to detect them. Such changes primarily occur when new characters are assigned to previously unassigned code points and inserted into the collation order. These changes should only affect individual characters, which would simplify our task significantly.

However, the secondary goal of this experiment is to verify that the *default* ICU collations are semantically equivalent to the *frozen* collations we have implemented. If they are semantically equivalent, that means that they produce the same results in every case regardless of any implementation details which might differ [13]. These versions of our ICU-based collations are available in the first and second build configurations listed in

[12] This is admittedly less than ideal, and something to avoid in real-world applications.
[13] See section 2.2.3.

table 5.1 respectively. As we discussed in section 4.3, the frozen collations are created by exporting the rule set from a normal ICU Collator object (which is created based on a locale identifier) and then creating a new RuleBasedCollator object from this rule set. If this is done correctly, the two collators should be semantically equivalent, and should produce the same results when used for collation. However, these collations can contain special rules for multi-character strings, which complicates matters. For example, the collation rules for the nb_NO locale (Norwegian Bokmål) include special rules for the characters *å* and *Å*, which are considered to be the same base letter as the strings *aa*, *Aa* and *AA* [14]. Therefore these multi-character strings must be tested as well, in order to detect errors which could affect them.

While it does complicate the task somewhat, we have chosen to create a test suite which should be able to achieve both of our goals. It is designed to compare the results of two collators and fail if it detects any differences between them. The data set used contains all valid Unicode code points and a set of relevant multi-character strings, chosen to cover the most likely cases. While this is not guaranteed to catch all differences in collation order, it is also not feasible to test all possible multi-character strings. We therefore consider this a reasonable compromise which should suffice to verify that our implementation is both correct and semantically stable.

5.4.1 Defining and limiting scope

Validating that two collations are semantically equivalent is not a straightforward task, as the rules in a collation can be quite complex. A common way to find differences between two collations is to compare all possible strings against each other. Even with the reasonable limitation of only comparing single, valid Unicode characters [15], this is still a very large number of comparisons. The number of comparisons required would grow quadratically with the size of the input, i.e. with a time complexity of $O(n^2)$. Example 5.3 presents a simplified example of this algorithm.

While this algorithm is simple to implement, the number of comparisons required makes it impractical to use. Each test of a pair of collations

[14] They are considered to differ only on a case or accent level. A case- and accent-insensitive collation would consider them identical.

[15] As of Unicode 15.0, this amounts to 149,186 characters.

```python
# Create a connection to the DBMS
db = create_database_connection()

# Create a list of all Unicode characters
def get_all_unicode_characters() -> list[str]:
    query = "SELECT character FROM unicode;"
    return [row[0] for row in db.cursor.execute(query).
    fetchall()]
characters = get_all_unicode_characters()
count: int = len(characters) # ~100-150k characters, depending
     on Unicode version


def compare(collation: str, s1: str, s2: str) -> tuple[int,
    int]:
    query = f"""SELECT
    %s = %s COLLATE {collation} AS equal,
    %s < %s COLLATE {collation} AS less_than;
    """
    return db.cursor.execute(query, (s1, s2, s1, s2)).fetchone
    ()

# For each pair of characters, check if they are different
differences: list[tuple[str, str]] = []
for i in range(0, count - 1):
    s1 = characters[i]
    for j in range(i + 1, count):
        s2 = characters[j]
        result1 = compare("collation1", s1, s2)
        result2 = compare("collation2", s1, s2)
        if result1 != result2:
            differences.append((s1, s2))
```

Example 5.3: Example of n^2 test to find collation differences

would take hours or perhaps days to complete. However, the goal of this experiment is simply to determine if the two collations are semantically equivalent, not to find all differences between them. This allows us to limit the scope of the experiment significantly, as we only need to find a single difference between the two collations in order to determine that they are not semantically equivalent. Also, we do not need to find the actual difference, only confirm whether or not one exists.

With this in mind, we can use a simpler algorithm which only needs to determine if two collation implementations produce the same total order for a given set of strings. This can be done by taking advantage of the fact that when a set of items is sorted, any elements which are considered equal will be placed adjacent to each other in the sorted list. They are not guaranteed to be arranged in any particular order, but the n^{th} element in the list must be less than or equal to the n^{th} element in a list must be less than or equal to the $(n+1)^{th}$ item [16]. First of all, two collations which are semantically equivalent will agree on whether the two items are equal or not. If two collations disagree on whether two items are equal, they cannot be semantically equivalent. Also, if the second collation considers the n^{th} element to be greater than the $(n+1)^{th}$ element, it cannot be semantically equivalent to the first collation. Example 5.4 presents a simplified example of this algorithm, which we will refer to as the *fast test*. It is called the fast test because it is significantly faster than the n^2 test, as it only needs to compare each item in the list to its immediate successor. The algorithm is therefore linear in the size of the input, i.e. with a time complexity of $O(n)$ [17]. Because of the improved time complexity, it is viable to expand the data set being tested significantly with only minor penalties to total run time.

The purpose of this test is to confirm that two collations which are expected to be semantically equivalent, are in fact semantically equivalent. If the proposed prototype is to be used in a real-world scenario, this test should be run regularly whenever a new version of ICU is released. This would be necessary to confirm that no changes in collation order have been introduced. It is reasonable to assume that such changes will happen extremely rarely, but it is still important to be able to detect them when they do occur. This means that it is a significant advantage if the test can be run quickly, as it will be run frequently and on a large number of collations. In the rare cases where a difference is detected, the n^2 test can be used to find the actual difference, but this is not necessary for the purposes of this experiment.

Given the simplicity of the fast test, it is also viable to extend the pool

[16] Assuming the list is sorted in ascending order.

[17] While it also requires a sorting step, this can take advantage of an index and is not a significant factor.

```
db = create_database_connection()

# Create a sorted list of all Unicode characters
def get_all_unicode_characters(collation: str) -> list[str]:
    query = f"""SELECT character FROM unicode
    ORDER BY character COLLATE {collation};
    """
    return [row[0] for row in db.cursor.execute(query).
    fetchall()]

def compare(collation: str, s1: str, s2: str) -> tuple[int,
    int]:
    query = f"""SELECT
    %s = %s COLLATE {collation} AS equal,
    %s < %s COLLATE {collation} AS less_than;
    """
    return db.cursor.execute(query, (s1, s2, s1, s2)).fetchone
    ()

characters = get_all_unicode_characters("collation1")
count: int = len(characters)

for i in range(0, count):
    s1 = characters[i]
    s2 = characters[i + 1]
    result1 = compare("collation1", s1, s2)
    result2 = compare("collation2", s1, s2)

    # Check that the second collation agrees on the order of
    the characters
    less_than_or_equal = result2[0] or result2[1]
    if not less_than_or_equal:
        print("Difference found!")
        break

    # Check that the two collations agree on the equality of
    the characters
    if result1 != result2:
        print("Difference found!")
        break
```

Example 5.4: Simplified version of fast test to detect collation differences

of test data to include multi-character strings. This could potentially be useful if one has reason to suspect that a particular multi-character string might be affected by a change in collation rules.

It is also worth noting that the actual implementation of the fast test allows testing against two separate database connections, even though this experiment only tests MySQL. In this experiment, this is done in order to test against different builds of MySQL, but it would be trivial to rewrite it to support any other interface which supports comparison operations. The only requirement is that one interface needs to be able to sort the test data, while both interfaces need to support a comparison operation. This could be used to verify that the collation order is consistent across different DBMSs, which could be useful in some scenarios. For example, if one is migrating from one DBMS to another, it might be useful to be able to verify that the collation order is consistent across the two DBMSs.

5.4.2 Test data

The test data used in this experiment consist of two separate tables, each containing a selection of strings. The first table, unicode_characters, contains all code points in the Unicode range (U+0000 to U+10FFFF) [18]. The full Unicode range, mostly consisting of unassigned code points, is used here instead of only using valid Unicode characters. By checking the collation order of all code points, we can detect changes in the collation order of previously unassigned code points. This typically happens when new characters are added to Unicode.

In the second table, sample_strings, we have inserted all two-character permutations of the Latin alphabet and all multi-character strings used in the tailoring rules. The two-character strings are added as an example, to illustrate that the test can handle adding arbitrary strings to the test data. Two-character strings are also commonly used in tailoring rules (as they represent ligatures), so it is reasonable to assume that they are more likely to be affected by changes in the collation rules compared to more complex strings. The remaining strings are added to cover strings used in the applied tailoring rules. They are extracted from the tailoring rules

[18] 2048 surrogate code points are excluded, as these cannot represent characters on their own. This should not affect the validity of the test.

using a Python script, which is included in the source code for this project.

We cannot guarantee that this data set will be sufficient to find all possible differences in collation order, but it should be sufficient for the purposes of this experiment. The main concern with our implementation is that the root collation order in ICU is not guaranteed to be stable, and may change between different versions of ICU. However, even if this changes, there is no reason to believe that the changes will add special rules for multi-character strings. For example, if ICU added a special rule for the string Hello, this would not be detected by our test suite. However, it is unlikely that such a rule would be added and detecting such changes is beyond the scope of this project. In order to detect such changes, it would be necessary to manually add the strings to the test data and re-run this experiment. This is a cheap and simple operation, as the test suite is relatively fast to run, but it does require knowledge of which strings might potentially be affected by the change.

5.4.3 Test process

The test setup consists of two separate database connections, each connected to a MySQL server. For this test, we have built two versions of our prototype, in order to compare their collation orders. These are therefore run separately, running on ports 3306 and 3307 respectively. The first version uses the default ICU collation rules and the second version uses the frozen collation rules, as described in section 5.1.1. This means that we are comparing the collation order of the default ICU collation rules, taken from a Locale object, with our implementation using static tailoring strings. If these are equivalent, this would indicate that our implementation is correct and semantically equivalent to the default way of using ICU collations.

We use the fast test to compare the collation order of the two versions, running the test for every ICU-based collation. The data for the test is generated by running a query against the first connection, selecting all strings from the two tables described above, and ordering them with the collation we want to test. We then iterate through this sorted list with both connections, comparing each string with its preceding string in the list. In other words, both versions of the collation are given queries like *SELECT a = b, a < b;*, *SELECT b = c, b < c;* et cetera. The result itself does not matter, only

that both collations give the same result for each pair of strings tested. If they at any point disagree, the collations are not equivalent and the test fails. This process is automated using a Bash script and the results output directly to the terminal [19].

5.4.4 Results

Collation	Result
utf8mb4_icu_en_US_ai_ci	Success
utf8mb4_icu_en_US_as_cs	Success
utf8mb4_icu_fr_FR_ai_ci	Success
utf8mb4_icu_ja_JP_as_cs	Success
utf8mb4_icu_ja_JP_as_cs_ks	Success
utf8mb4_icu_nb_NO_ai_ci	Success
utf8mb4_icu_zh_Hans_as_cs	Success

Table 5.10: Collations validated by the fast test, comparing the frozen collations with their equivalent Collator+Locale versions.

5.4.5 Summary

In table 5.10 we see the results from running the fast validation test on all of the ICU based collations we have implemented. As we expected, the test was successful for every tested collation. This shows that we have detected no differences between our frozen collations, implemented with static strings and RuleBasedCollator, and the equivalent collations using the Collator class with a Locale object. This is a strong indication that the frozen collations are semantically equivalent to the current collations in ICU, which would mean that our implementation is correct. However, this is not a guarantee, as there may be other differences between the two implementations that are not detected by this test. Given the time constraints of this project, we have not been able to perform a more thorough analysis of the differences between the two implementations.

[19] https://github.com/LarsV123/master-util/blob/master/src/experiment3.sh

CHAPTER 6

CONCLUSION

This thesis started with four related research questions, which we will revisit to explain how we answered them:

RQ1 How can applications that require semantic stability use ICU for collation without requiring a static library version?

RQ2 Given that ICU is a specialized library, does it perform better than the current collation implementation in MySQL?

RQ3 Does it harm performance to enforce semantic stability in ICU?

RQ4 How can we detect changes to a collation order?

RQ1, our primary research question, concerned whether it was possible to enforce semantic stability in the ICU library without requiring a static version of the library. We show this is possible by implementing a proof-of-concept prototype that satisfies both requirements. The results in section 5.4 verify the validity of the approach. By integrating our prototype into MySQL, we also show that it is possible to use ICU as a drop-in replacement for existing collations. From this, we see that using the methods outlined here, MySQL and other applications with similar requirements can take advantage of the ICU library without sacrificing semantic stability.

The benchmarks in section 5.2 show that the performance of our prototype is at least comparable to the existing implementations and, in many cases, outperforms them. The results indicate that collations implemented using ICU have similar performance regardless of the collation rules used, while the different MySQL implementations vary in performance. This variance is also visible in the results of section 5.3, where we measured the amount of CPU time used by collation-specific functions rather than total execution time. While the impact on total execution time was less pronounced, these results show that the slowest MySQL collations use roughly three times as much CPU time as the fastest ones. These two experiments demonstrate the advantage of using ICU for collation, as it can provide consistent performance regardless of the collation rules. Therefore, the answer to RQ2 is inconclusive in the case of the default collation but positive for the other collations tested.

We also created a prototype variant with additional collation rules added, imitating the situation where we must compensate for unwanted changes in the underlying rules. This variant was tested in section 5.2, and the results show that the performance of this prototype is comparable to the original prototype. Therefore we conclude that the answer to RQ3 is negative, as the results indicate that enforcing semantic stability does not impact performance.

RQ4, our final research question, concerned the ability to detect such changes in the underlying rules and the practicality of doing so. We discuss a known method for finding differences in collation orders and a utility that implements a faster test to detect whether changes exist. This test confirms whether or not two collations (or versions of the same collation) are semantically equivalent. The utility is not dependent on ICU or MySQL and could be used independently of both with minor modifications. For this reason, it could be helpful to anyone who wishes to detect changes in collation rules. Because of the improved run time of this test, it could be practical to use it as part of automated testing, which would allow applications to detect changes in the underlying rules as soon as they occur.

6.1 Future work

The core problem which inspired this thesis was that the ICU library is not semantically stable and that this is a problem for applications that require semantic stability. Crucially, this is a problem for indexes in DBMSs, which become corrupted by changes in the underlying collation rules. Alternate solutions to this problem could be investigated, such as repairing indexes more efficiently after a collation change. The naive solution to this would entail traversing the leaf nodes of the index and checking whether they are in collation order, scaling linearly with the number of indexed records.

An alternative and more general approach would be to rewrite the ICU library in such a way as to guarantee semantic stability or at least warn the user when it is not possible. This approach would be a significant undertaking requiring much work. However, with the increasing use of ICU in applications such as PostgreSQL, it may be worth considering. It would be worthwhile investigating whether the ICU library could be versioned in such a way as to guarantee semantic stability and whether this would be a practical solution.

Finally, if MySQL or other applications wish to use the ideas discussed in this thesis, it is advisable to perform further testing. The performance benchmarks carried out in this thesis were limited in scope and only tested on a single platform. As MySQL is a cross-platform application, it would be advisable to conduct more extensive testing. This testing should include a broader set of collations and all platforms officially supported.

TABLE OF ABBREVIATIONS

Term	Definition
API	Application Programming Interface
CLDR	Common Locale Data Repository
DBMS	Database Management System
DUCET	Default Unicode Collation Element Table
ICU	International Components for Unicode
OSS	Open Source Software
SQL	Structured Query Language
UCA	Unicode Collation Algorithm
XML	Extensible Markup Language

Table 1: List of abbreviations used in the text.

APPENDIX A

APPENDIX A

This appendix contains the results of the experiments described in chapter 5. See that chapter for a more detailed explanation of the results.

A.1 Experiment 1 - Performance benchmark

This section presents the full, aggregated results from section 5.2 for the medium-sized data set. As discussed in section 5.2.3, the results for the other data sets are similar enough that they are not included here.

This experiment consists of three measurements (comparison, and ordering in both directions) for various combinations of collation and data locale. The ICU collations were tested in three different configurations, while the MySQL collations were tested once. These configurations are described in section 5.2. We will refer to section 5.2.3 for a more detailed explanation of what these tables show, as they are identical to those presented there.

The locale column specifies the locale of the data set being tested. For example, the data set for the en_US locale contains English names, while the data set for the ja_JP locale contains Japanese names. Because of the size of the tables, they are split on both configurations (ICU variation or MySQL) and the operation being measured.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_0900_ai_ci	en_US	4.76	0.05	0
utf8mb4_0900_as_cs	en_US	5.36	0.04	12.67
utf8mb4_ja_0900_as_cs	en_US	7.16	0.05	50.43
utf8mb4_ja_0900_as_cs_ks	en_US	7.31	0.27	53.6
utf8mb4_nb_0900_ai_ci	en_US	5.44	0.07	14.23
utf8mb4_zh_0900_as_cs	en_US	7.66	0.17	61
utf8mb4_0900_ai_ci	fr_FR	5.33	0.15	11.97
utf8mb4_ja_0900_as_cs	ja_JP	7.25	0.21	52.27
utf8mb4_ja_0900_as_cs_ks	ja_JP	7.69	0.17	61.67
utf8mb4_nb_0900_ai_ci	nb_NO	5.44	0.1	14.38
utf8mb4_zh_0900_as_cs	zh_Hans	6.34	0.19	33.28

Table A.1: ORDER BY ASC for all MySQL collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_0900_ai_ci	en_US	4.99	0.05	0
utf8mb4_0900_as_cs	en_US	5.45	0.05	9.31
utf8mb4_ja_0900_as_cs	en_US	6.75	0.05	35.35
utf8mb4_ja_0900_as_cs_ks	en_US	7.46	0.05	49.56
utf8mb4_nb_0900_ai_ci	en_US	5.68	0.07	13.81
utf8mb4_zh_0900_as_cs	en_US	8.16	0.17	63.56
utf8mb4_0900_ai_ci	fr_FR	5.42	0.13	8.72
utf8mb4_ja_0900_as_cs	ja_JP	7.33	0.16	46.96
utf8mb4_ja_0900_as_cs_ks	ja_JP	7.81	0.17	56.61
utf8mb4_nb_0900_ai_ci	nb_NO	5.68	0.04	13.78
utf8mb4_zh_0900_as_cs	zh_Hans	6.41	0.17	28.43

Table A.2: ORDER BY DESC for all MySQL collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_0900_ai_ci	en_US	4.64	0.04	0
utf8mb4_0900_as_cs	en_US	4.51	0.04	-2.87
utf8mb4_ja_0900_as_cs	en_US	5.74	0.04	23.58
utf8mb4_ja_0900_as_cs_ks	en_US	5.73	0.27	23.4
utf8mb4_nb_0900_ai_ci	en_US	4.81	0.03	3.52
utf8mb4_zh_0900_as_cs	en_US	5.69	0.04	22.54
utf8mb4_0900_ai_ci	fr_FR	5.05	0.13	8.62
utf8mb4_ja_0900_as_cs	ja_JP	5.77	0.17	24.32
utf8mb4_ja_0900_as_cs_ks	ja_JP	5.76	0.22	24.04
utf8mb4_nb_0900_ai_ci	nb_NO	4.77	0.11	2.74
utf8mb4_zh_0900_as_cs	zh_Hans	5.64	0.19	21.35

Table A.3: Equality comparison for all MySQL collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.35	0.07	12.51
utf8mb4_icu_en_US_as_cs	en_US	5.45	0.1	14.51
utf8mb4_icu_fr_FR_ai_ci	en_US	5.5	0.16	15.52
utf8mb4_icu_ja_JP_as_cs	en_US	6.51	0.07	36.72
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.98	0.07	25.72
utf8mb4_icu_nb_NO_ai_ci	en_US	5.59	0.06	17.57
utf8mb4_icu_zh_Hans_as_cs	en_US	6.2	0.04	30.35
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.6	0.12	17.72
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.33	0.15	32.99
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.58	0.22	38.16
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.53	0.14	16.17
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.81	0.19	22.03

Table A.4: ORDER BY ASC for all ICU_default collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.36	0.07	7.47
utf8mb4_icu_en_US_as_cs	en_US	5.48	0.06	9.94
utf8mb4_icu_fr_FR_ai_ci	en_US	5.7	0.16	14.31
utf8mb4_icu_ja_JP_as_cs	en_US	6.42	0.25	28.6
utf8mb4_icu_ja_JP_as_cs_ks	en_US	6.54	0.07	31.02
utf8mb4_icu_nb_NO_ai_ci	en_US	5.4	0.06	8.27
utf8mb4_icu_zh_Hans_as_cs	en_US	5.82	0.04	16.62
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.57	0.13	11.59
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.35	0.15	27.37
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.62	0.06	32.67
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.55	0.07	11.24
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.9	0.15	18.28

Table A.5: ORDER BY DESC for all ICU_default collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	4.58	0.03	-1.3
utf8mb4_icu_en_US_as_cs	en_US	4.67	0.09	0.54
utf8mb4_icu_fr_FR_ai_ci	en_US	5.2	0.03	11.84
utf8mb4_icu_ja_JP_as_cs	en_US	5.36	0.25	15.34
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.82	0.04	25.29
utf8mb4_icu_nb_NO_ai_ci	en_US	4.92	0.03	5.84
utf8mb4_icu_zh_Hans_as_cs	en_US	5.5	0.03	18.48
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5	0.12	7.7
utf8mb4_icu_ja_JP_as_cs	ja_JP	5.7	0.19	22.62
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	5.89	0.22	26.88
utf8mb4_icu_nb_NO_ai_ci	nb_NO	4.65	0.03	0.2
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.38	0.15	15.78

Table A.6: Equality comparison for all ICU_default collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.06	0.08	6.32
utf8mb4_icu_en_US_as_cs	en_US	5.03	0.09	5.64
utf8mb4_icu_fr_FR_ai_ci	en_US	5.5	0.16	15.54
utf8mb4_icu_ja_JP_as_cs	en_US	5.71	0.09	20.08
utf8mb4_icu_ja_JP_as_cs_ks	en_US	6.26	0.08	31.48
utf8mb4_icu_nb_NO_ai_ci	en_US	5.24	0.14	10.11
utf8mb4_icu_zh_Hans_as_cs	en_US	5.58	0.1	17.18
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.36	0.13	12.53
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.09	0.21	28.06
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.32	0.24	32.89
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.21	0.12	9.5
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.65	0.2	18.63

Table A.7: ORDER BY ASC for all ICU_frozen collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.09	0.07	1.98
utf8mb4_icu_en_US_as_cs	en_US	5.22	0.09	4.72
utf8mb4_icu_fr_FR_ai_ci	en_US	5.23	0.15	4.86
utf8mb4_icu_ja_JP_as_cs	en_US	6.26	0.08	25.45
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.76	0.08	15.39
utf8mb4_icu_nb_NO_ai_ci	en_US	5.33	0.12	6.84
utf8mb4_icu_zh_Hans_as_cs	en_US	6	0.09	20.27
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.38	0.17	7.77
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.13	0.09	22.85
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.35	0.23	27.22
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.27	0.12	5.68
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.67	0.19	13.58

Table A.8: ORDER BY DESC for all ICU_frozen collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	4.4	0.02	-5.29
utf8mb4_icu_en_US_as_cs	en_US	4.62	0.02	-0.55
utf8mb4_icu_fr_FR_ai_ci	en_US	5.01	0.03	7.86
utf8mb4_icu_ja_JP_as_cs	en_US	5.65	0.02	21.61
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.66	0.02	21.75
utf8mb4_icu_nb_NO_ai_ci	en_US	4.73	0.02	1.78
utf8mb4_icu_zh_Hans_as_cs	en_US	5.32	0.02	14.6
utf8mb4_icu_fr_FR_ai_ci	fr_FR	4.82	0.12	3.79
utf8mb4_icu_ja_JP_as_cs	ja_JP	5.51	0.18	18.58
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	5.71	0.16	22.87
utf8mb4_icu_nb_NO_ai_ci	nb_NO	4.66	0.02	0.29
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.17	0.11	11.4

Table A.9: Equality comparison for all ICU_frozen collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.12	0.13	7.69
utf8mb4_icu_en_US_as_cs	en_US	5.07	0.19	6.64
utf8mb4_icu_fr_FR_ai_ci	en_US	5.57	0.15	16.97
utf8mb4_icu_ja_JP_as_cs	en_US	5.76	0.36	21.13
utf8mb4_icu_ja_JP_as_cs_ks	en_US	6.31	0.15	32.69
utf8mb4_icu_nb_NO_ai_ci	en_US	5.37	0.2	12.8
utf8mb4_icu_zh_Hans_as_cs	en_US	5.63	0.31	18.3
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.42	0.14	13.89
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.15	0.21	29.19
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.42	0.24	34.86
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.31	0.19	11.57
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.7	0.22	19.79

Table A.10: ORDER BY ASC for all ICU_tailored collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	5.13	0.14	2.84
utf8mb4_icu_en_US_as_cs	en_US	5.28	0.09	5.82
utf8mb4_icu_fr_FR_ai_ci	en_US	5.28	0.27	5.85
utf8mb4_icu_ja_JP_as_cs	en_US	6.31	0.11	26.4
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.83	0.36	16.81
utf8mb4_icu_nb_NO_ai_ci	en_US	5.4	0.17	8.21
utf8mb4_icu_zh_Hans_as_cs	en_US	6.06	0.12	21.39
utf8mb4_icu_fr_FR_ai_ci	fr_FR	5.44	0.15	9.05
utf8mb4_icu_ja_JP_as_cs	ja_JP	6.18	0.11	23.84
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	6.43	0.24	28.94
utf8mb4_icu_nb_NO_ai_ci	nb_NO	5.33	0.15	6.89
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.73	0.2	14.89

Table A.11: ORDER BY DESC for all ICU_tailored collations.

Collation	Locale	Time (s)	Std. dev (s)	Δ baseline (%)
utf8mb4_icu_en_US_ai_ci	en_US	4.43	0.04	-4.6
utf8mb4_icu_en_US_as_cs	en_US	4.65	0.06	0.02
utf8mb4_icu_fr_FR_ai_ci	en_US	5.04	0.32	8.41
utf8mb4_icu_ja_JP_as_cs	en_US	5.66	0.23	21.89
utf8mb4_icu_ja_JP_as_cs_ks	en_US	5.68	0.17	22.24
utf8mb4_icu_nb_NO_ai_ci	en_US	4.74	0.08	2.11
utf8mb4_icu_zh_Hans_as_cs	en_US	5.34	0.17	14.92
utf8mb4_icu_fr_FR_ai_ci	fr_FR	4.88	0.12	4.99
utf8mb4_icu_ja_JP_as_cs	ja_JP	5.54	0.18	19.31
utf8mb4_icu_ja_JP_as_cs_ks	ja_JP	5.74	0.18	23.54
utf8mb4_icu_nb_NO_ai_ci	nb_NO	4.68	0.07	0.83
utf8mb4_icu_zh_Hans_as_cs	zh_Hans	5.21	0.13	12.27

Table A.12: Equality comparison for all ICU_tailored collations.

APPENDIX B

APPENDIX B

This appendix contains code examples that are either too long to fit in the main text or are not essential to the understanding of the thesis. The code examples are listed in the order of their appearance in the text.

B.1 Implementation

The following code snippet demonstrates how the frozen tailoring strings used in the implementation are generated.

```
#include <unicode/coll.h>
#include <unicode/errorcode.h>
#include <unicode/locid.h>
#include <unicode/regex.h>
#include <unicode/uchar.h>
#include <unicode/ucol.h>
#include <unicode/unistr.h>
#include <unicode/unorm2.h>
#include <unicode/usearch.h>
#include <unicode/ustream.h>
#include <unicode/ustring.h>
#include <unicode/utypes.h>

#include <iostream>
#include <vector>
```

```cpp

#include "unicode/coll.h"
#include "utils.hpp"

// Create a collator with a given locale ID and extract the tailoring rules
void get_tailoring(COLLATION collation) {
  // Create collator
  UErrorCode status = U_ZERO_ERROR;
  icu::Locale locale = icu::Locale::createFromName(collation.name.c_str());
  icu::Collator* collator = icu::Collator::createInstance(locale, status);

  // Get size of ruleset
  UCollator* ucollator = collator->toUCollator();
  int32_t rulesLength = ucol_getRulesEx(ucollator, UCOL_TAILORING_ONLY, NULL, 0);

  // Allocate buffer for ruleset
  UChar* rules = new UChar[rulesLength];
  ucol_getRulesEx(ucollator, UCOL_TAILORING_ONLY, rules, rulesLength);

  // Create string from ruleset
  UnicodeString rules_string(rules, rulesLength);

  // Write ruleset to a file with the given locale name
  string filename = "rules/" + collation.name + ".txt";
  cout << "Writing ruleset to " << filename << endl;
  ofstream file;
  file.open(filename, ios::out | ios::trunc);
  file << rules_string;
  file.close();
}

// Extract the tailoring rules from the root collation
void get_root_collation_tailoring() {
  // Create collator
  UErrorCode status = U_ZERO_ERROR;
  icu::Locale locale = icu::Locale::createFromName("en_US");
  icu::Collator* collator = icu::Collator::createInstance(locale, status);

  // Get size of ruleset
```

```
  UCollator* ucollator = collator->toUCollator();
  int32_t rulesLength = ucol_getRulesEx(ucollator,
    UCOL_FULL_RULES, NULL, 0);

  // Allocate buffer for ruleset
  UChar* rules = new UChar[rulesLength];
  ucol_getRulesEx(ucollator, UCOL_FULL_RULES, rules,
    rulesLength);

  // Create string from ruleset
  UnicodeString rules_string(rules, rulesLength);

  // Write ruleset to a file with the given locale name
  string filename = "rules/root.txt";
  cout << "Writing ruleset to " << filename << endl;
  ofstream file;
  file.open(filename, ios::out | ios::trunc);
  file << rules_string;
  file.close();
}

int main(int argc, char** argv) {
  // All locales used in the experiment
  vector<string> locales = {"nb_NO", "en_US", "uk_UA", "th_TH"
    , "zh_Hans"};

  // Get tailoring for all locales
  for (auto locale : locales) {
    cout << "Locale: " << locale.name << endl;
    get_tailoring(locale);
  }

  // Get tailoring for root collation
  get_root_collation_tailoring();
}
```

Example B.1: Method used for extracting tailoring rules from ICU Collator object.

BIBLIOGRAPHY

[1] Incorporated Merriam-Webster. *Merriam-Webster Dictionary*. 2023. URL: https://www.merriam-webster.com/ (visited on 2023-06-08).

[2] ICU Project. *ICU on GitHub*. 2022. URL: https://github.com/unicode-org/icu (visited on 2022-10-18).

[3] Inc. Unicode. *About Unicode*. 2023. URL: https://home.unicode.org/about-unicode/ (visited on 2023-06-13).

[4] Terry A. Halpin. *Conceptual Queries*. URL: http://www.orm.net/pdf/conceptqueries.pdf (visited on 2022-12-13).

[5] Lars-Olav Vågene. "Using the ICU library for collations in MySQL". Preparatory project. 2022.

[6] D Feitosa et al. "CODE reuse in practice: Benefiting or harming technical debt". In: *The Journal of Systems and Software* 167 (2020).

[7] S Haefliger, G von Krogh, and S Spaeth. "Code Reuse in Open Source Software". In: *Management science* 1 (2007), pp. 180–193. ISSN: 0025-1909.

[8] Semver.org. *Semantic Versioning 2.0.0*. 2023. URL: https://semver.org/ (visited on 2023-05-18).

[9] Fedora Project. *Fedora Packaging Guidelines*. 2022. URL: https://docs.fedoraproject.org/en-US/packaging-guidelines/ (visited on 2022-10-18).

[10] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd ed. MIT press, 2009. ISBN: 978-0262033848.

[11] Ralph Grimaldi. *Discrete and Combinatorial Mathematics: An Applied Introduction*. 5th ed. Pearson, 2014. ISBN: 978-1-292-02279-6.

[12] Unicode Consortium. *Unicode Collation Algorithm*. 2022. URL: https://unicode.org/reports/tr10/ (visited on 2022-11-21).

[13] Richard Gillam. *Unicode Demystified: A Practical Programmer's Guide to the Encoding Standard*. Addison Wesley, 2002. ISBN: 0-201-70052-2.

[14] *ISO/IEC 9075-2:2016: Information technology —Database languages —SQL —Part 2: Foundation (SQL/Foundation)*. Standard. International Organization for Standardization, 2016-12.

[15] Unicode Consortium. *ICU Documentation: Locale*. 2023. URL: https://unicode-org.github.io/icu/userguide/locale/ (visited on 2023-06-10).

[16] Unicode Consortium. *Unicode® 15.0.0*. 2022. URL: https://www.unicode.org/versions/Unicode15.0.0/ (visited on 2023-06-10).

[17] International Components for Unicode. *ICU - International Components for Unicode - ICU 4.6 Changes*. https://icu.unicode.org/download/46. 2010. (Visited on 2023-06-12).

[18] Unicode Consortium. *New Japanese Era*. 2018. URL: http://blog.unicode.org/2018/09/new-japanese-era.html (visited on 2023-06-10).

[19] Arne Torp. *Skandinaviske «særbokstaver»*. Språknytt. 2002. URL: https://www.sprakradet.no/Vi-og-vart/Publikasjoner/Spraaknytt/Arkivet/Spraaknytt_2002/Spraaknytt_2002_1/Skandinaviske_saerbokstaver/ (visited on 2023-06-12).

[20] Dag Gundersen. *aa*. Ed. by Erik Bolstad. Store norske leksikon, 2023. URL: https://snl.no/aa (visited on 2023-06-12).

[21] Unicode Consortium. *CLDR*. 2023. URL: https://github.com/unicode-org/cldr/blob/main/common/collation/no.xml (visited on 2023-06-12).

[22] Markus Scherer. *Unicode Mail List Archive: ICU library changes name*. 2000-01. URL: https://unicode.org/mail-arch/unicode-ml/Archives-Old/UML021/0409.html (visited on 2023-06-11).

[23] ICU Project. *Collation Service Architecture*. 2023. URL: https://unicode-org.github.io/icu/userguide/collation/architecture.html (visited on 2023-06-11).

[24] ICU Project. *Collation Concepts*. 2023. URL: https://unicode-org.github.io/icu/userguide/collation/concepts.html (visited on 2023-06-11).

[25] ICU Project. *ICU Architectural Design*. 2023. URL: https://unicode-org.github.io/icu/userguide/icu/design (visited on 2023-05-18).

[26] Tech Fry. *History and Versions of MySQL*. 2023. URL: https://www.techfry.com/programming-tutorial/history-and-versions-of-mysql (visited on 2023-06-11).

[27] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Pearson, 2015. ISBN: 0-13-397077-9.

[28] Peter Gulutzan. *MySQL, MariaDB, International Components for Unicode*. 2017. URL: https://ocelot.ca/blog/blog/2017/04/11/mysql-mariadb-international-components-for-unicode/ (visited on 2022-10-23).

[29] MySQL. *Bug #27877*. 2007. URL: https://bugs.mysql.com/bug.php?id=27877 (visited on 2023-06-13).

[30] MySQL. *Bug #40053*. 2008. URL: https://bugs.mysql.com/bug.php?id=40053 (visited on 2023-06-13).

[31] MySQL. *Bug #9604*. 2005. URL: https://bugs.mysql.com/bug.php?id=9604 (visited on 2023-06-13).

[32] *MySQL 8.0 Reference Manual :: 10.3.1 Collation Naming Conventions*. 2023. URL: https://dev.mysql.com/doc/refman/8.0/en/charset-collation-names.html (visited on 2023-06-13).

[33] Peter Eisentraut. *More robust collations with ICU support in PostgreSQL 10*. 2017. URL: https://www.2ndquadrant.com/en/blog/icu-support-postgresql-10/ (visited on 2022-10-06).

[34] Thomas Munro. *Don't Let Collation Versions Corrupt Your PostgreSQL Indexes*. 2022. URL: https://techcommunity.microsoft.com/t5/azure-database-for-postgresql/don-t-let-collation-versions-corrupt-your-postgresql-indexes/ba-p/1978394 (visited on 2023-06-13).

[35] PostgreSQL. *Collations*. 2021. URL: https://wiki.postgresql.org/wiki/Collations (visited on 2022-11-21).

[36] ArangoDB GmbH. *ArangoDB - GitHub*. 2020. URL: https://github.com/arangodb/arangodb/tree/devel/3rdParty/V8/v7.9.317/third_party/icu (visited on 2022-10-23).

[37] Apache Software Foundation. *Provide a locale/collation-aware text comparator*. 2012. URL: https://issues.apache.org/jira/browse/CASSANDRA-4245 (visited on 2022-10-23).

[38] Apache Software Foundation. *Installation*. 2022. URL: https://docs.couchdb.org/en/3.2.2-docs/install/unix.html (visited on 2022-10-23).

[39] IBM. *Collating sequence*. 2021. URL: https://www.ibm.com/docs/en/i/7.2?topic=concepts-collating-sequence (visited on 2022-10-23).

[40] EnterpriseDB. *Unicode collation algorithm*. 2022. URL: https://www.enterprisedb.com/docs/epas/latest/epas_guide/03_database_administration/06_unicode_collation_algorithm/ (visited on 2022-10-23).

[41] MongoDB Inc. *MongoDB - GitHub*. 2022. URL: https://github.com/mongodb/mongo/tree/master/src/third_party/icu4c-57.1 (visited on 2022-10-23).

[42] Microsoft Corporation. *Collation and Unicode Support*. 2022. URL: https://learn.microsoft.com/en-us/sql/relational-databases/collations/collation-and-unicode-support?view=sql-server-ver16 (visited on 2022-10-23).

[43] PostgreSQL. *PostgreSQL 15.1 Documentation - Appendix E. Release Notes*. 2022. URL: https://www.postgresql.org/docs/15/release-15.html (visited on 2022-11-21).

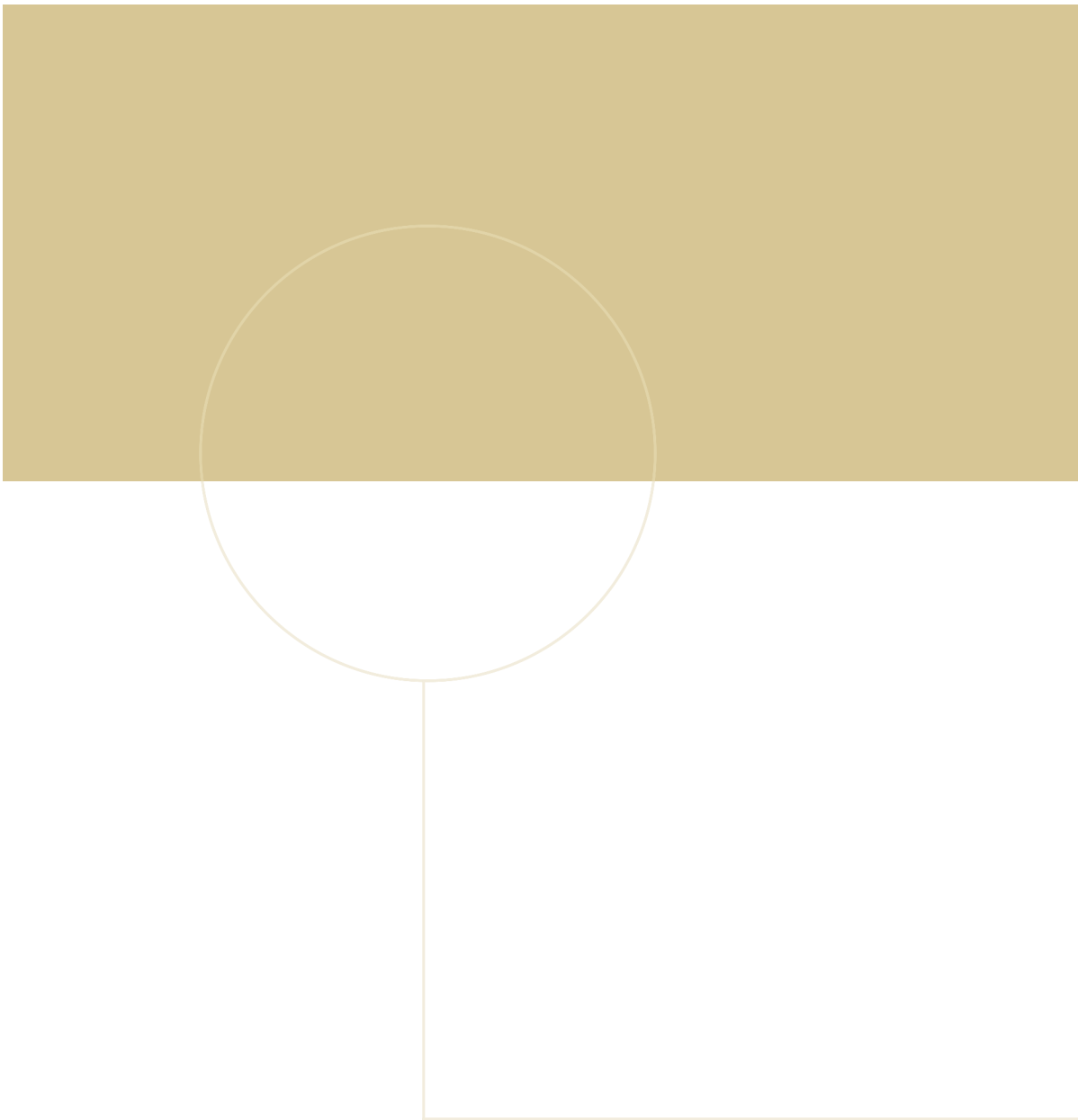
[44] D. Richard Hipp. *SQLite Documentation*. 2022. URL: https://www.sqlite.org/src/doc/trunk/ext/icu/README.txt (visited on 2022-10-23).

[45] SAP SE. *What is ICU, and when is it needed?* 2022. URL: https://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.help.sqlanywhere.12.0.0/dbadmin/natlang-s-7944836.html (visited on 2022-10-23).

[46] Lars-Olav Vågene. *Using the ICU library for collations in MySQL*. Prototype implementation. 2023. URL: https://github.com/LarsV123/mysql-server/pull/3/files (visited on 2023-06-13).

[47] Lars-Olav Vågene. *Using the ICU library for collations in MySQL*. Utility repository. 2023. URL: https://github.com/LarsV123/master-util (visited on 2023-06-13).

[48] Saša Stamenković. *Country List*. 2021. URL: https://github.com/umpirsky/country-list/ (visited on 2022-12-08).



NTNU

Norwegian University of
Science and Technology