MASTER'S THESIS | LUND UNIVERSITY 2016

Merging customer relationship management data

Fredrik Gustafsson

Department of Computer Science
Faculty of Engineering LTH

ISSN 1650-2884
LU-CS-EX 2016-29

Merging customer relationship management data

Fredrik Gustafsson
iveqy@iveqy.com

August 6, 2016

Version 1.0

Keywords: CRM, merge, supporting distributed workflow, merging structural data in CRM systems, distributed, distributed CRM, customer relationship management, customer relationship data, merging CRM data

Dept. of Computer Science, Lund University
Supervisor: Lars Bendix
Examiner: Ulf Asklund

Abstract

Working distributed is increasingly important today with technology being more and more portable while connectivity is still lacking in some areas. Cellphones and laptops are increasingly in use outside the office with connectivity to internal office services being unreliable, either due to lacking speed or inability to have a connection.

Different solutions exist. One is a client-server setup where the data traffic can be minimized. Another is an internet service where the client is in the web browser. A third is to use a distributed system where each system is has full functionality by its own but needs synchronizing with other copies of the system. Distributed work requires duplication of data and therefore, also a need to merge data.

Great efforts have been made to be able to merge program code in text files and this kind of merge solutions are pretty mature. However, for merging other types of data the methods are still young and unproved.

For a CRM system, the data is highly structured and different data fields are known in advance. This gives good opportunities to merge diverged data. However the current systems on the market are lacking support for merge algorithm knowing about the data format.

Even if the data is structured it doesn't mean that a merge is trivial. Structured data might have dependencies between different data fields, so that one field depends on the value of another field. That means that a merge tool needs to be aware of these dependencies. If also taking the history of changes since the diverging point into account the possibilities for detecting merge problems is greatly increased.

Different CRM systems have used different approaches to merging data, however, all found approaches have been without actually merging any data, but instead either identify a difference or guessing the most correct version without merging. That means that a result will be either one of two conflicting changes but not a combination of the two.

This paper has studied different use cases for altering CRM data and how three different types of CRM programs solve these cases. A custom merge algorithm that solves all studied cases has been designed and implemented as a proof of concept.

The proof of concept implementation works satisfying. It uses the history and the structure of the data, combined with information about dependencies between different data fields to identify conflicts and perform a merge. Merge is only done when it can be done without any risk of loosing information.

The conclusion is that current CRM software handles these problems poorly and that it is possible to merge this data and/or detect conflicts in a good way.

Contents

1 Introduction

Computers and cellphones are more and more capable and portable the need for working everywhere has increased. Being forced to be at the office or connected to a certain network to be able to use different computer programs are an obstacle for working efficient. This is especially true for traveling salesmen who are working on the move.

Mobile networks and Virtual Private Network, VPN, connections are two solutions used to let a computer work outside the office site, both has however, their weaknesses. Mobile networks are often slow and unreliable. The quality depends on the strength of the connection, which varies between different areas and are dependent of the position of the device using them.

VPN connections isn't really a type of connection but a way to connect securely from one untrusted network to a trusted one. This type of solution is in itself not bad, but it is often used to make the computer using it to think that it is the same network as it uses VPN to connect to. However even if it is the same network security wise, it isn't the same network speed wise. Connecting over a VPN tunnel is often much slower that sitting direct on the network. This means that network heavy programs won't work in a good way over network, even if VPN is used.

A third common way is to use a web application so that the program is used as a website. This has the advantage that the client doesn't have to install any additional software and that it can be reached from everywhere there is an internet connection. However, web applications are often slow and hard to work with due to loading times. A loading time of 1 seconds is perceived as fast, but studies show that a human will notice loading times over 100 ms and change her behavior when using the application with slower than this loading times, [14].

The problem with sharing data has been seen in program development and source code collaboration for quite a few years. This has led to distributed systems where each client is fully fledged and fully featured by itself and then it's synchronized with other clients. This is possible thanks to advances in techniques for merging two different versions of text files, which is what source code most often is written in.

Since the computer industry is considered to be early adopters of technology, it can be assumed that this techniques could be interesting to examine for use in different areas, such as Customer Relationship Management, CRM, systems.

Instead of trying to achieve good connectivity and instant data for all users, a distributed system could be used where users can work offline with access to the whole database.

A users' database is then allowed to diverge against other users' databases, resulting in the double maintenance problem, [4]. To reduce the different versions down to one version again, the databases needs to be synchronized

in some way. This requires good support for conflict detection and merge capabilities.

There's a vast number of different CRM solutions on the market. Looking at how different types of software handle this issue today, categorizing these systems to different architectures are leaves just a couple of different implementations. Two typical and one more exotic distributed architectures has been chosen to study how they allow distributed working.

It might be possible to merge CRM data more efficiently taking the knowledge about the data format into account. CRM data is highly structured and it's known beforehand which fields exists and what their relations between each other are.

A custom algorithm will be implemented as a proof of concept to show that it is possible to merge or at least be notified about merge conflicts for all use cases studied.

To answer these questions, a collection of common use cases has to been defined and studied. The use cases show conflicts, merges and how even non conflicting changes could be a problem for parallel work. An important part here being the unit of versioning [8].

It's easy to imagine that interesting use cases should contain difficult merge problems. However, it's equally important to show cases of easy merges, cases that just need to be found to be conflicts, and cases that doesn't conflict at all. All these needs to be handled in a correct manner to efficiently and securely be able to work distributed.

This paper presents a number of use cases, describe how different types of CRM software handle distributed workflows and show an implementation of a custom merge algorithm that successfully will solve all use cases presented.

2 Background

In this chapter a background of the problem will be presented and then the different CRM applications that will be examined will be presented.

Using inspiration from *Diff and merge support for feature oriented development* [7] and remembering the choices needed to be made between consistency, availability and partition tolerance [1] and the performance impact on the user experience when working with thin clients [14] an interest in study merging of customer relationship data was born.

The CAP-theorem [1] says that there's a tradeoff between consistency, availability and partitioning. Increasing availability and partitioning (and therefore performance) will lower the consistency and result in the double maintenance problem [4].

On a conceptual level the question is if it is possible to mitigate the consistency issue enough to enjoy the improved availability and partitioning. Most CRM application today seems to rate availability above consistency and trying to mitigate the availability issue instead of the consistency issues.

This chapter will start with a section describing different CRM architectures from a data perspective. Then three CRM application are chosen to represent the different architectures, these will also be presented.

2.1 CRM architectures

There are several different CRM applications available and there are many different implementations which are using different architectures.

Three common architectures are found, client-server, web based and distributed. From these three CRM application were chosen to represent each category.

Different architectures does not imply different merge strategies but they do have different need of merges. A client-server architecture has such small timeframe where a conflict might arise that the need for a merge capability is lower since there're less conflicts than for a system with very long timeframes.

However in this paper, the common solution for each architecture is used as an example and the fact that a certain architecture can have different merge soltions is ignored.

The common problem with a multiuser CRM system is to have the whole database shared with all users. That is, all users of the system need to have access to the same data. Being a CRM system, it's not needed to be updated instantly, but in a reasonable timeframe.

2.1.1 Client-server

Technically, all types of systems presented here can be called a client-server system. A traditional client-server system is a system where all data is

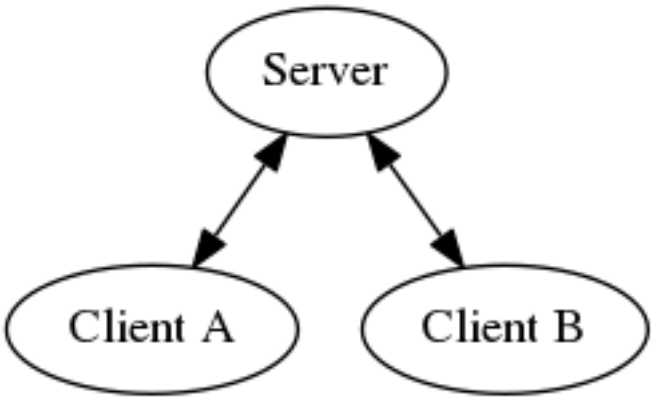

Figure 1: Client-server

present on the server and where the clients asks for information from the server each time information is needed. There can be different thicknesses [1] of the clients, depending on how much information is stored locally at the client.

2.1.2 Web-based

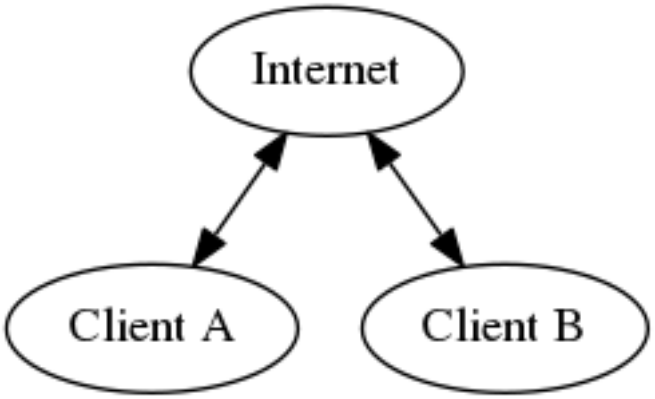

Figure 2: Web-based

A web-based system is a client-server system with all information and the program stored on the server. The system is using web technologies such as HTML and http, to connect to the server. This has the advantage that every computer that has a web browser can connect to the server, but

[1] A thin client is a client without any data stored, depending completely on information from the server to operate. A thick client has all data local and communicates with the server when necessary. There's unlimited levels of thicknesses between these two extremes

the disadvantage that web technologies is limited by the choices of protocol available, since only web protocols such as http, https, etc. can be used.

The web-based system is actually a restricted client-server system. The reason for separating it into its own category is that web-based system often do similar tradeoffs. They often rely on transactions being fast and overwrite data if another user has saved data between a page load and a page save from another user.

There's also a very limited possibility to alert clients that the data they are looking at has changed and needs to be reloaded.

2.1.3 Distributed

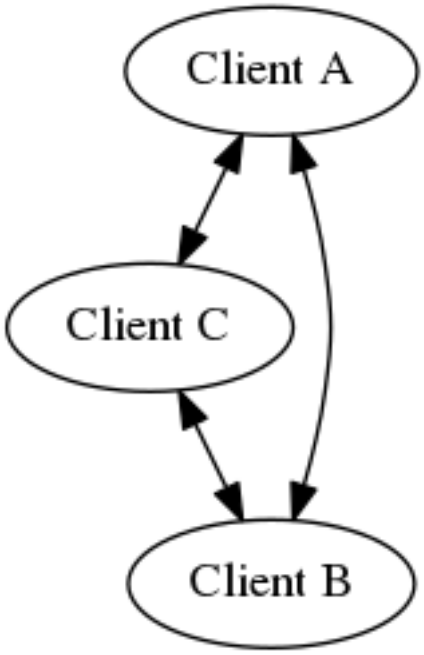

Figure 3: Distributed system

A distributed system has several equal nodes[2] where no one is more important than the other. All nodes can share information with any other node. The advantage that each system is self sufficient, but with the disadvantage that data may diverge and that there's no clear point that is guaranteed to have the most up to date data.

A distributed system can be used with a centralized server so that it actually acts as a client-server system, with the exception that offline work is possible and that the server isn't more important in any way than the

[2] a node is an instance of the program. Most often a separate computer with the program running on it.

client. Except that it's used as a communication center for the clients. If the server disappears any of the clients contains all the information needed to replace the server.

2.2 Different CRM systems

In this section, three different CRM system will be introduced. Their origin and technical differences as well as why they were chosen will be presented.

The CRM systems were chosen based on their different technical implementation approaches:

- Lime Easy, use a local database that is be shared over a network file system such as samba.
- Deko the CRM, use a document database with built-in support for conflicting versions of a document.
- Fat Free CRM, is a web application that will have one common data source for all clients.

2.2.1 Lime Easy

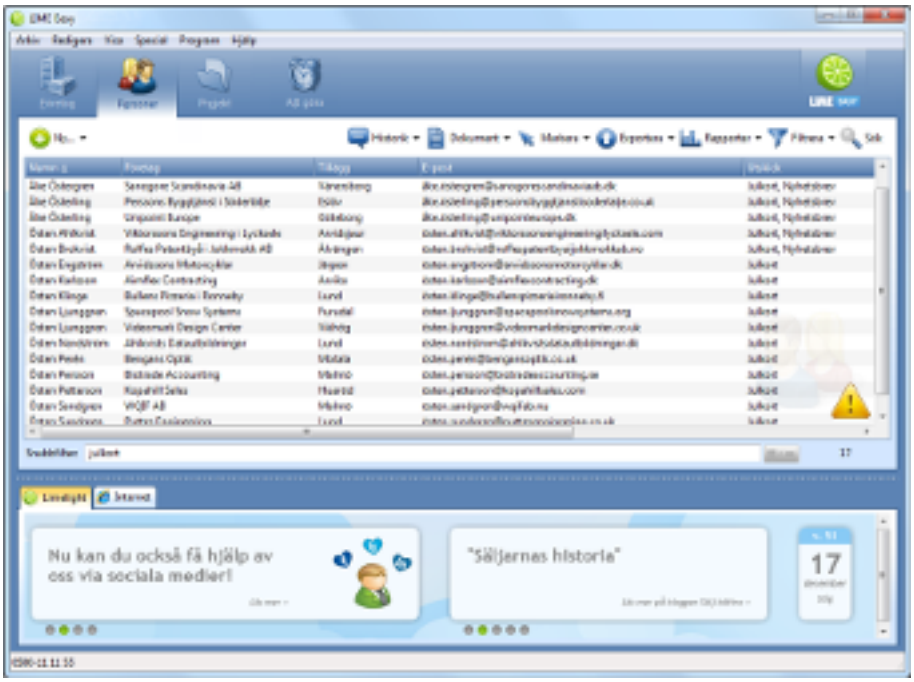

Figure 4: Lime Easy

Lime easy is a CRM system developed by Lundalogik [?]. It follows a very common model for programs sharing a database on a Microsoft system.

The client program is written as a desktop application and it uses a Microsoft Access database as data store.

This database could be put on a network drive, to allow multiple clients to use the same database. This is a simple method of that quickly let the developer of the software implement a client/server like application. The negative parts is that it often requires a fast network and hence won't scale to many clients. This is because all data that is needed for a calculation must travel over the network instead of just the data that the user is requesting to see, which is the case for most web-based systems.

For using Lime easy in a distributed way without being online with a central server, Lime easy is using a "check in"/"check out" model. A workflow where the network database is copied in whole to the client and when the client is put in offline mode it's using its local database.

Lime Easy is chosen to represent the old style client-server type of CRM systems and how they can cope with the modern requirements of distributed workflows.

2.2.2 Deko the CRM

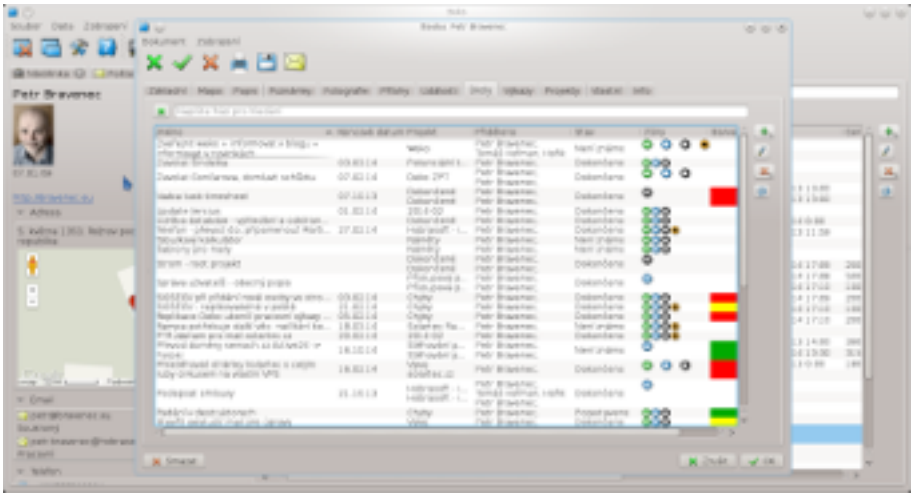

Figure 5: Deko the CRM

Deko the CRM is a CRM system developed by Hobrasoft [2]. After requests from customers about the possibility to work offline, they build this system on top of Apache CouchDB. A screenshot of Deko the CRM can be seen in figure 5.

The Apache CouchDB has a built in replication and synchronization mechanism. Being a document database, it doesn't have any knowledge about the data inside the documents, but can resolve conflicts on a document level only.

A user has a complete database locally and can specify remote databases to sync against. If the remote databases are available, the synchronization can start and conflicts will be detected by Deko the CRM, or more correctly by Apache CouchDB. All conflicts have to be manually solved and are listed in a list of conflicts for the user to easy see which records are conflicting.

Deko the CRM is chosen to investigate one of few distributed CRM systems that has an interesting technical implementation, since CRM systems building upon a distributed database is very rare.

2.2.3 Fat Free CRM

Figure 6: Fat Free CRM

Fat Free CRM, figure 6, is an open source CRM system founded by Michael Dvorkin [?]. It's an application written in the programming environment Ruby on Rails [?], with a SQL-database as data store. It's installed on one server and then connected to by visiting a webpage, usually over the internet. The system is then interacted with through the web browser.

The thought about conflicts here is that each local copy of data is so short lived that conflicts are rare and hence do not need discovery. Data is saved when the user press the a *Save* button, so the assumption is that the user is quick to press that button after an edit is done.

Instead the latest version is always used. This means that data can be lost when it's overwritten and that the user won't even get a notice about it.

Fat Free CRM is chosen to investigate the rapidly, in popularity, growing web-based CRM systems.

3 Analysis

In this chapter, there is a short overview of current CRM implementations. Then an analysis of different use cases and what their merge result should be. followed by an analysis of Lime Easy, Deko the CRM and Fat Free CRM will be done.

After reading this chapter the reader will be familiar with common software architectures of common CRM application and have a good overview of a couple of interesting use cases for distributed work with CRM application.

The reader will also have a toolbox that make it possible to solve the different uses cases and an understanding on which tools are useful in which case as well as which tools are not useful.

3.1 Use cases

In this section a number of use cases are presented. The use cases are chosen to explore cases that could be common use cases when working distributed with CRM. They will then function as a requirement base and test environment for a custom implemented tool that will explore the possibilities of solving the use cases.

There's a huge number of different use cases that could be described. However just a few of them are chosen as they represent the same problem as many other use cases.

Each use case have a prefered result that is the correct behaviour for that situation. A good result would be an implementation that produces that result. A less good result but still an acceptable result is an implementation that does not destroy data or might produce wrong results and a bad implementation is an implementation that might destroy data.

In this paper the different use cases will be categorized as:

- Only one change
- Unrelated changes
- Related changes
- Conflicting change
- Unknown relation

An edge case that seldom occur is not of great importance as long as the program assures that it results in a merge conflict, losing data is a problem, edge case or not.

In this section a collection of use cases is presented. They are chosen to be representative both from a view of different common use case scenarios but also to demonstrate the different merge techniques that may be available.

Some merge techniques presented are not implemented in the custom merge tool for the reason that they simply aren't good enough for this particular problem. They are still presented as dead ends in the analysis phase of the paper.

First some general techniques and related topics are presented to give the reader a good ground to stand on before the different types of changes and conflicts will be presented. This is followed by a few techniques that could be used to solve use cases.

3.1.1 Figure explanation

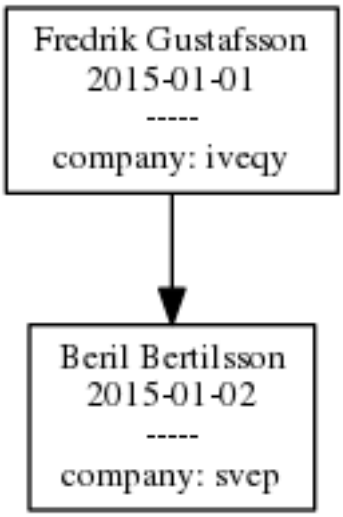

Figure 7: Example figure

In figure 7 a change can be seen. Each square represents a node. The first line in the node is the name of the person that has created that node. Below is the date when the node was created. In a real implementation this would be a timestamp instead of a date, but in this paper dates are used for clarity.

Below the line a list of changed fields can be seen. If there's nothing below the line, no changes is made.

Even if each node contains of multiple fields only the fields that are changed somewhere in the figure are shown.

An arrow means that one node is the parent to an other node.

In figure 7 the author *Fredrik Gustafsson* did a change at *2015-01-01*. It's not showed all fields that Fredrik changed but it's showed that Fredrik gave the field *Company* the value *iveqy*.

Then the new author *Beril Bertilsson* did a change to Fredriks node at *2015-01-02* where he changed the field *Company* to have the new value *svep*.

This page is intentionally left blank.

3.1.2 Use case 1

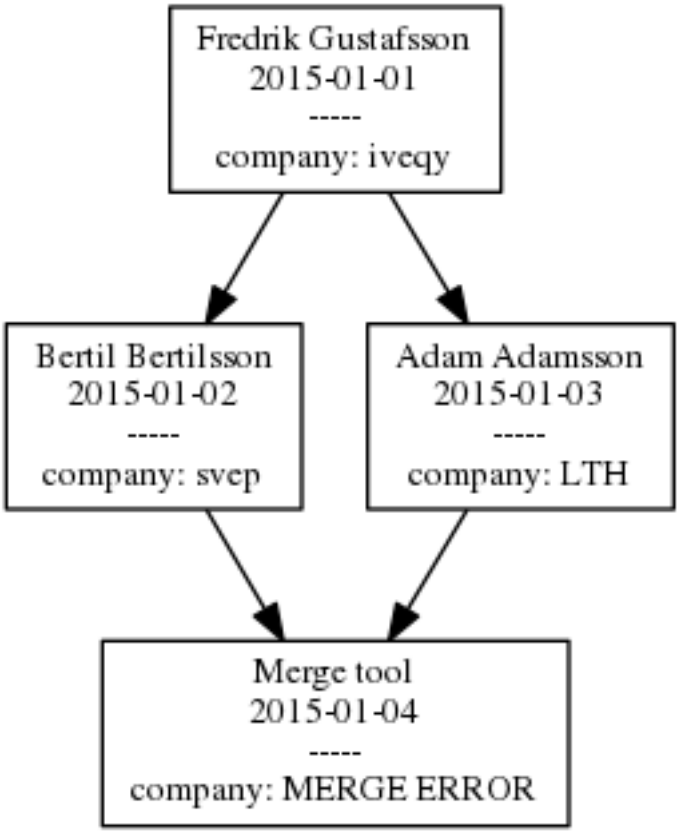

Figure 8: Use case 1

This test case will show a conflict detection solution for two conflicting one field changes.

In figure 8, a simple change can be seen. A version changes the value of the company field from "iveqy" to "svep". The other version changes the company field from "iveqy" to "LTH".

There's no good way, from this information alone, to know if "LTH" or "svep" is the correct company. The result of a merge between these two versions should therefore be a merge error.

A tool picking the latest version would choose "LTH" and just ignore "svep", since "LTH" was edited after "svep". Adam didn't have access to the edit done by Bertil and that's why information will be lost if only the latest version is picked.

A merge error should be shown to the user to be able to do a manual merge. This can be done in a number of different ways, the most common being that the last person trying to synchronize with a master database,

that being Adam and Bertil, the date of the change doesn't matter, just the synchronization date, should do the merge resolution.

It's not necessary that the last person has enough information to do the merge manually. Another way to manually solve a merge conflict is that any person could suggest a solution and that all persons then vote on the best solution and strive for something everybody has deemed correct. This works well in a centralized system where all users always can communicate with all other users. In a distributed system this is not always the case and the team effort of solving merge conflicts is much harder to implement.

The easiest way to solve this is that if the person doing the merge resolution doesn't have enough information to do a good resolution, he or she should make sure to get access to enough information to be able to solve the merge. For example, by talking to the person that had done the conflicting change.

So it can conclude that this use case tests if a tool have conflict detection.

3.1.3 Use case 2

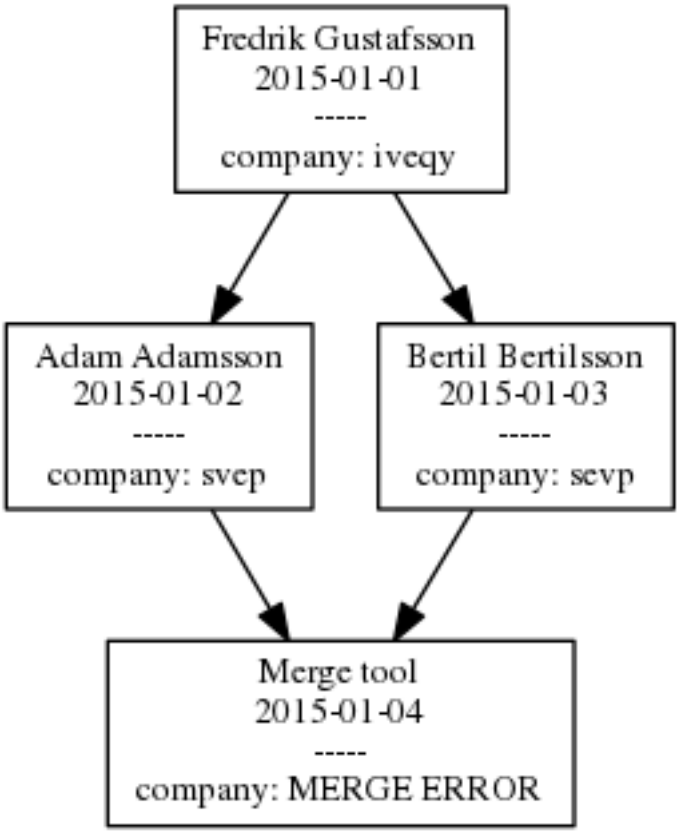

Figure 9: Use case 2

In this use case, the handling of conflicts due to spelling errors will be examined.

In figure 9, the initial version is diverging into two different versions. With company "svep" and "sevp". One version is misspelled. The most important thing is to notice that this is a merge error.

This would trigger the same merge error mechanism as use case 1, section 3.1.2.

Trying to propose a merge can be done with noticing that it's a spelling error. There's a number of algorithms to determine the edit distance between two words. The most famous might be the Levenshtein distance, however there are algorithms that will provide better results for names.

A possible solution would be to see if the edit distance between two words in a merge conflict is small and then use an external resource, for example, a wordlist, or even better a list of names or company names to determine if one of the words, but not the other, is correctly spelled.

If two words have a small edit distance and one of them is misspelled while the other one is correctly spelled, it can be assumed that the correctly spelled word is the correct answer to the merge conflict. However, this would still just be an assumption. How good this assumption could be is outside of this paper, a very rough estimation would be over 85 % [5].

Since there's no way to verify if a merge has produced a correct answer or not, like it is with source code, an erroneous merge shouldn't be able to compile or run the test suite, it's very important that a merge never introduces false positives. That is, solves a merge case without a guaranteed correctness of the result. Instead it should be marked as a merge conflict to be solved manually.

A helpful program could of course try to hint this to the human trying to solve the merge, that this is probably a spelling error.

So it can be concluded that there aren't good enough tools to solve this merge on CRM data, but this should result in a merge error.

3.1.4 Use case 3

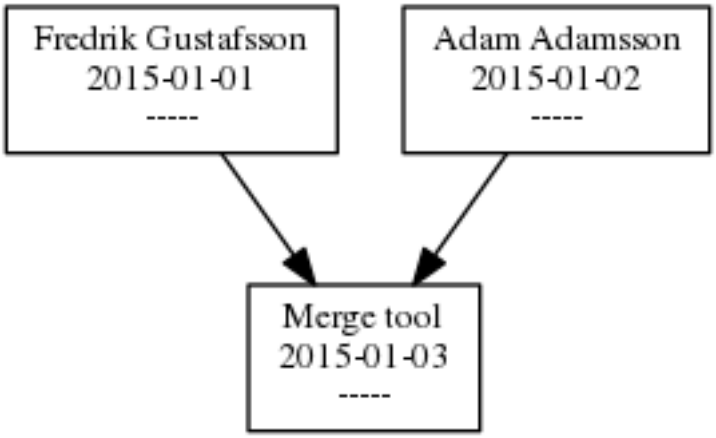

Figure 10: Use case 3

In this use case, two exact same leaf nodes are present. For example if two salesmen met the same customer and add him/her to the system with the exact same data.

One orphan[3] node exists with the same data as another orphan, as in figure 10, or a not orphan leaf node. In the case that all data is filled in, it's a clear duplicate and the two nodes should be merged to one.

However, if not all data is filled in, there's a chance that it's not the same node. For example, if only first name and company are entered it's not unreasonable to assume that there can be two people with the same first name working at the same company.

If there isn't a full match, the safest approach here is to not flag this at all. The risk here is duplicated data, which is a serious problem in databases, but to throw away data that a faulty merge would do isn't very good either.

If one of the nodes isn't a leaf node. The leaf node can be advanced to the same state as the non-leaf node and merge, if and only if, it can be determined that they are the same entry (that is a full match).

A possible middle way solution is to flag a merge conflict if there's a partial data match and let the user decide if this could be a duplicate or not.

With only a few fields containing data, the likelihood if two versions being a duplicate is lower even if the data match. A good merge algorithm should be able to only flag possible duplicates if it has a high enough reason to suspect that a duplicate might be possible.

For example, just adding someone named "Fredrik" without any other data added should probably not be flagged as a duplicate of another entry

[3] An orphan node is a node with no parent

with the name “Fredrik”. Names are an example of a poor comparison field, since two people could have the same name. A phone number is a much stronger indicator of a duplicate and should render a higher score for determination if it’s something to alert the user about or not, but of course even a phone number doesn’t need to be personal and more than one person could share the same phone number.

It can be concluded that great care must be taken to ensure that two physical different persons don’t end up as one entry in the system. But avoiding duplicates greatly improves the data quality in the CRM system.

3.1.5 Use case 4

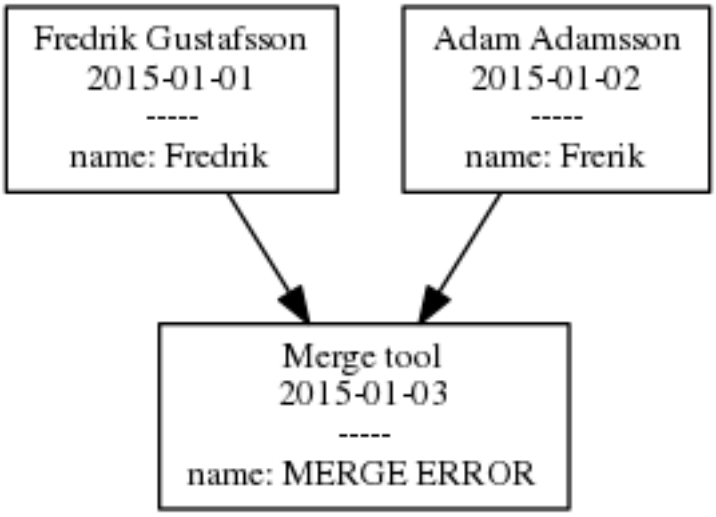

Figure 11: Use case 4

This use case will examine the possibilities for finding duplicates due to spelling errors.

An edit distance search could be made to find potential duplicate data. This would be a combination of the spelling error in use case 2, in section 3.1.3, and the duplicate data problem in use case 3, in section 3.1.4. The correction of spelling mistakes has the same consequences as in use case 2, in section 3.1.3 and those has to be weight against the duplicate data problem. However, since duplicated data is better than thrown away data, this case shouldn't be flagged.

If the spelling error detection can be good enough, there might still be a real world usage for a merge algorithm that can succeed in merging this case, but the result can never be assured to be correct. A safe approach would be to flag this as a possible duplicate and let the user make a decision.

This depends on which data field is being examined. In this example the name field is the field examined and as discussed in use case 3, section 3.1.4 it's rather poor indicator of a duplicate. In use case 3, section 3.1.4, a phone number is an example of a field with a high score for determining a duplicate. However, it's not necessarily true for a misspelled phone number since people at the same company tend to have similar phone numbers that have a low edit distance between them. In that case it's also important at what position the phone number differs.

For two entries where only the phone number differs slightly and this is not the last number of the phone number that differs, a duplicate is less likely to be the case than if the phone number differs at an earlier number.

Since all possible duplicates are flagged for the user anyway, a false positive isn't as bad as a faulty merge. The extra work done by solving false duplicates are to be related to the cost of having a database with a higher number of duplicates. This is probably something that different users/companies will value differently depending on their business case.

It can be concluded that even if the results will contain false positives it can be worth doing since the user will check if the merge should be concluded or not.

3.1.6 Use case 5

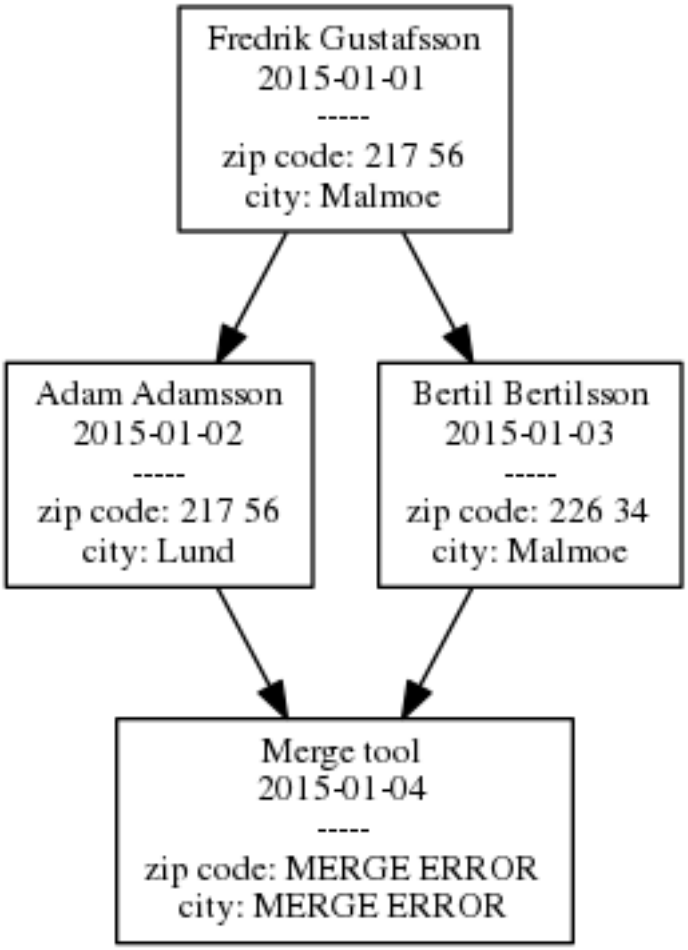

Figure 12: Use case 5

In this use case it will be examined if two dependent fields [4] are changed in a way that each fields don't conflict but a logical conflict due to the dependency between the fields exists.

In figure 12, two values are changed. The values are dependent on each other and that dependency must be taken into account when merging.

A tool doing a merge by looking at each row would simple merge this without any conflict ending up with incorrect data. In the worst case, both previous versions would have correct data that combined would have an invalid combination of values. This would be what is often called an evil

[4] By definition in this paper, zip code and city are dependent.

merge [5].

A tool doing an item based merge, where the change of zip code also marks the city as a change would mark this as a merge conflict, a much safer approach that will result in more merge conflicts but guaranteed correct data.

A way to attack this problem is to have a dependency list of which field is dependent on which other field(s) and then do the merge if the fields are independent and mark a merge conflict if they are dependent. This is related to the problem of choosing the best unit of comparison and is an implementation of a variable unit of comparison suggested in [8]. The idea is to have a very small unit of comparison that can be combined with other units when needed to form bigger units of comparison. This might sound a bit tricky but is easy to implement for the CRM system case where all data fields are known beforehand. For other systems where the format of the data can't be defined or known beforehand in the same way this would be harder to implement.

It can be concluded that it's important to have a flexible unit of comparison to be able to have an as small as possible unit of comparison, but never a so small unit of comparison that inconsistent merge results can occur.

[5] An evil merge is defined in this paper as a merge resulting in a value that isn't present in any of the versions that are being merged.

3.1.7 Use case 6

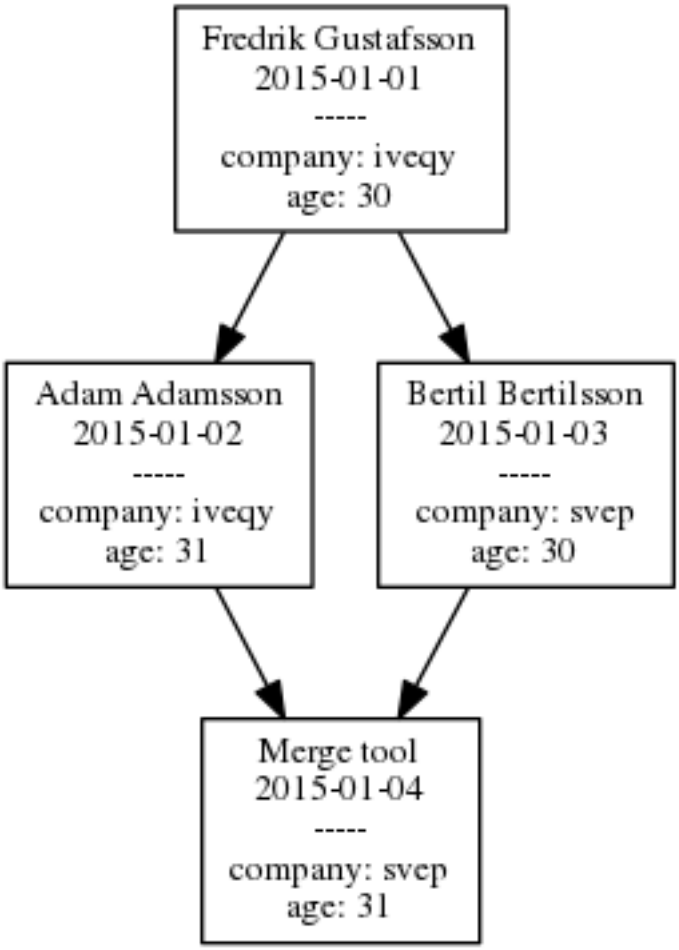

Figure 13: Use case 6

In this use case it will be examined if two independent fields are changed in a way that each field don't conflict but no logical conflict due to the dependency between the fields exists.

In figure 13, two values are changed. The values are independent and no dependency must be taken into account when merging. This is the same problem as in use case 5, section 3.1.6, but with independent fields. This merge should succeed in a good tool.

This use case will show if a tool is doing a change-based merge where any conflict will result in a merge conflict. This instead of a item-based or row-based merge where each value in the change will be merged against the corresponding value in the other change.

The unit of comparison is important for this merge case. A small unit of

comparison will merge this without problem while a bigger unit of comparison will result in a merge conflict.

A merge conflict would still be better than to just choose Bertils change because that's the newest. The result then would be that Adams age change would get lost.

A small unit of comparison, but without a merge algorithm more advanced then taking the latest entry, would also succeed with this merge case.

It can be concluded that it's important to have a flexible unit of comparison to be able to have an as small as possible unit of comparison and minimize the number of conflicts.

3.1.8 Use case 7

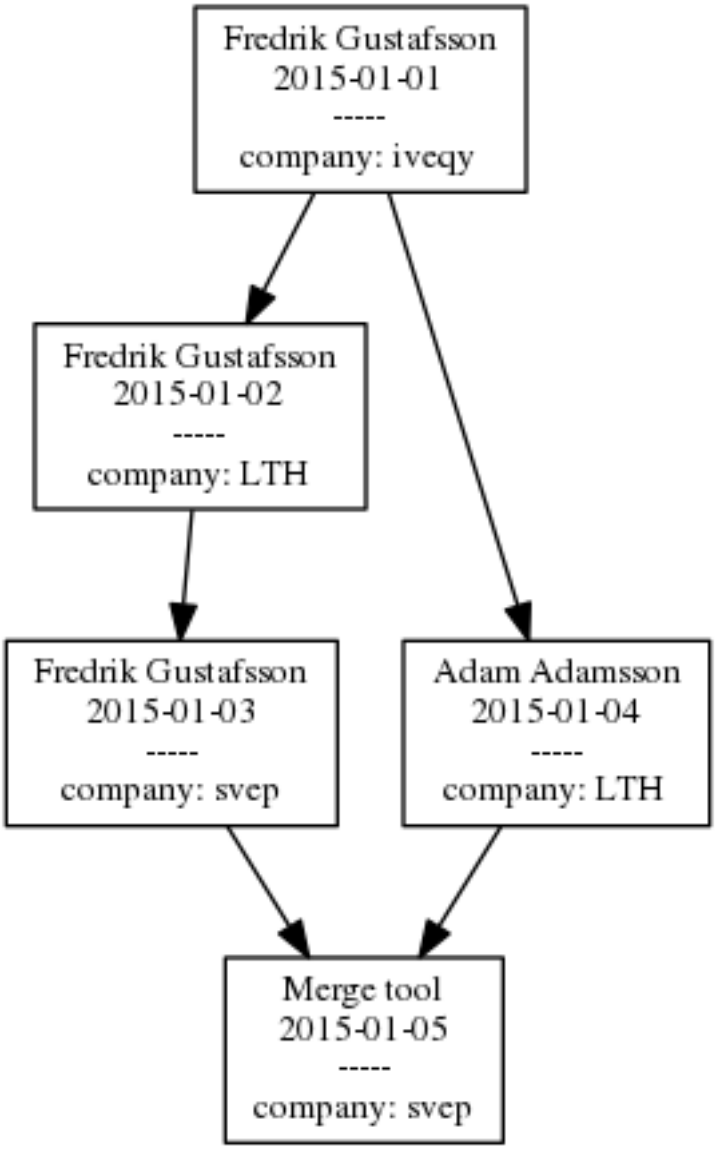

Figure 14: Use case 7

In this use case it will be shown how information about the history can help solve a simple merge conflict.

In figure 14, looking only on the changes with the dates 2015-01-03 and 2015-01-04 a clear conflict can be seen. However, when also looking at the history this use case could be solved.

One way to solve this would be to look at the dates of the two conflicting

changes and choose the latest one. However, this could be wrong since each change builds on the data from the previous change. Choosing the latest change will result in choosing a change that is done without all data. In this case the change from 2015-01-04 would be chosen and it won't know about 2015-01-02 at all (in this particular case 2015-01-04 and 2015-01-02 is identical and it doesn't matter).

Another way to solve this use case would be to use the branch with most changes in. This is for example the standard behavior of merges done by Couch DB [3]. This way would work in this particular case but could result in errors for example if one change was a simple spelling correction, that branch would still have a heavier weight.

The way of doing this would be to find a common sequence. Since one branch contains the change "LTH" $\rightarrow$ "svep", that change could be applied to the other branch as well, since "LTH" exists in that branch as well. The result would be two branches with leafs that both contain "svep" and the merge would be without conflicts.

This way could be achieved by keeping a change list from each branch and try to apply that to the other branch(es) until a trivial merge is found.

Looking at not the data but the changes and merge changes instead of data is closely related to operation-based merging and conflict detection [10].

It can be concluded that saving the history of each branch can give the merge algorithm valuable information for being able to solve a merge conflict.

3.1.9 Use case 8

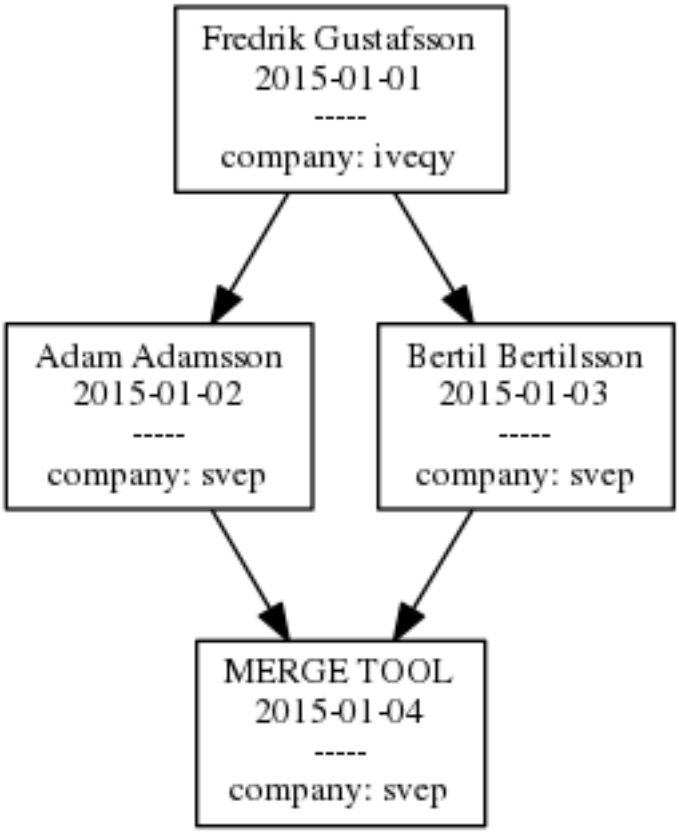

Figure 15: Use case 8

In this use case, two identical changes should be merge into one.

In figure 15, two people change the same field with the exact same value. This shouldn't be a problem for a merge tool and it's not a conflict so it shouldn't be flagged as a conflict. However, this is an easy test to see if an application is using actually trying to do a trivial merge or just flags a conflict if an edit has been made.

A tool doing a trivial merge attempt will probably solve this with ease while a tool not doing a merge will fail and mark this as a merge conflict, hopefully. A tool could also choose to ignore this at all and just perform a merge, picking one of the versions. This wouldn't show up in this use case since a merge will result in the same result as if just one version would be chosen at random, however such approach will be revealed in use case 1 8.

It can be concluded that even a simple case like this will fail if the tool doesn't try to do a merge, but only conflict detection.

This page is intentionally left blank.

3.1.10 Use case 9

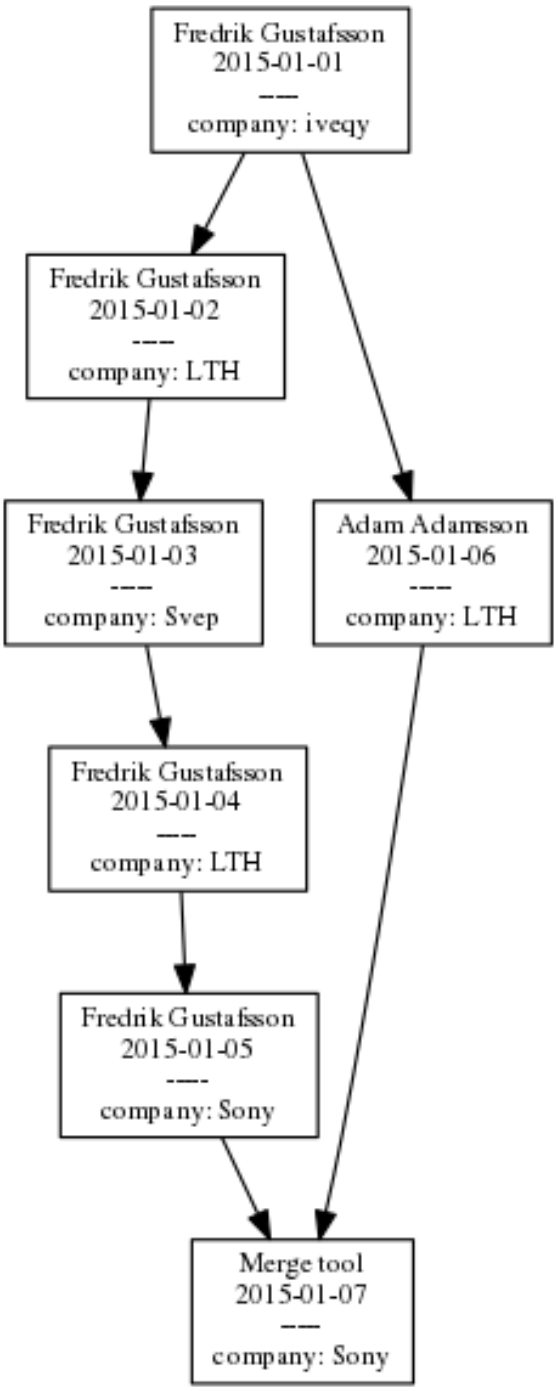

Figure 16: Use case 9

This use case will study the use of many one step translations instead of one multi-step translation.

The case showed in figure 16 is almost identical with use case 7, section 3.1.8, however here there is two translations from "LTH". That is:

- "LTH"→"Svep"
- "LTH"→"Sony"

Looking only for one step translations won't work here. Either a multi-step translation could be done, that is first:

- "LTH"→"Svep"
- "Svep"→"LTH"
- "LTH"→"Sony"

should be applied to the right hand branch, or translations with more steps should be allowed, that is:

- "LTH"→"Svep"
- "LTH"→"Svep"→"LTH"
- "LTH"→"Svep"→"LTH"→"Sony"

should be applied one at the time until a trivial merge is found.

Note that in the multi-step solution, the case that starts and ends with the same value ("LTH"→"Svep"→"LTH") this could be ignored since it doesn't change the value (and is probably an "undo" operation from the user).

It can be concluded that support for multi-step translations or translations with more steps are crucial for cases with longer history.

3.1.11 Use case 10

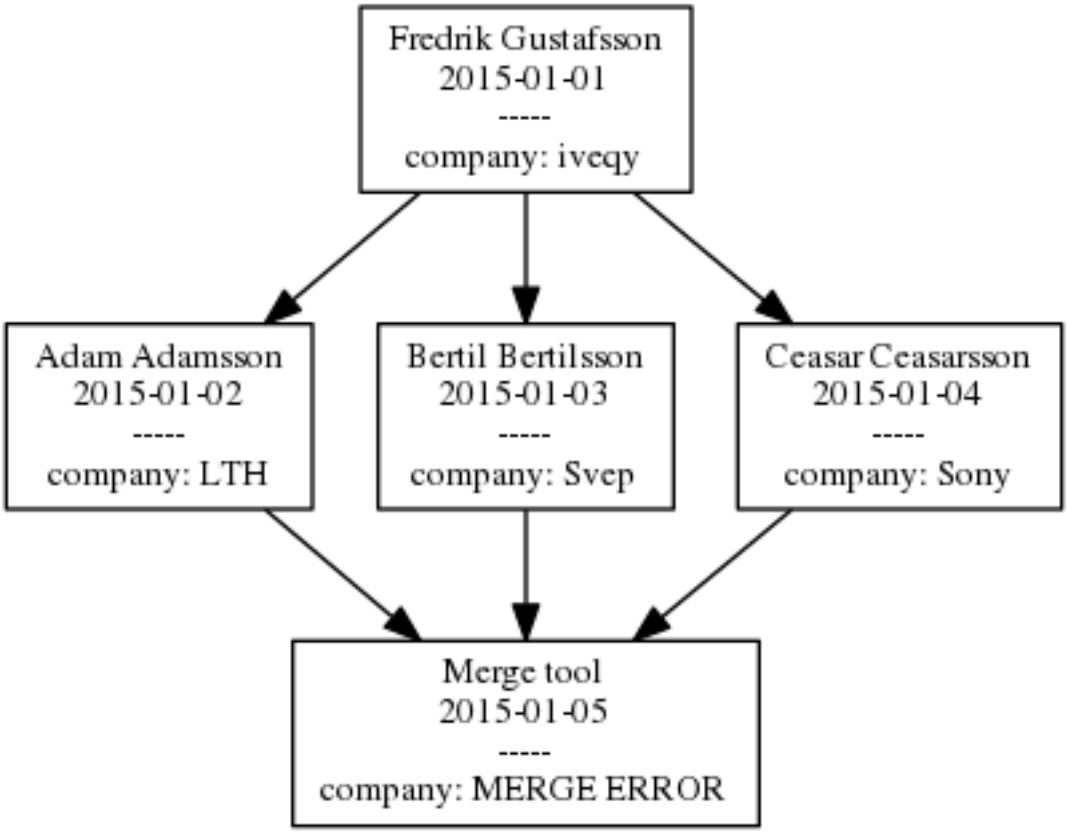

Figure 17: Use case 10

This use case will show how to handle multiple parents for an easy conflict detection case.

In figure 17 a multi parent merge is shown. A simple way to attack this problem is to merge changes two and two until all changes are merged. For example, first merge Adams and Bertils changes into one and then merge that result with Ceasars change. This might have implications, that will be shown in use case 11, section 3.1.12, however for this use case it will work just fine.

Since all changes that is been merged have different values, this is clearly a conflict that should result in a merge error.

It can be concluded that if all leafs are different, they can be handled two and two using the previous discussed techniques.

This page is intentionally left blank.

3.1.12 Use case 11

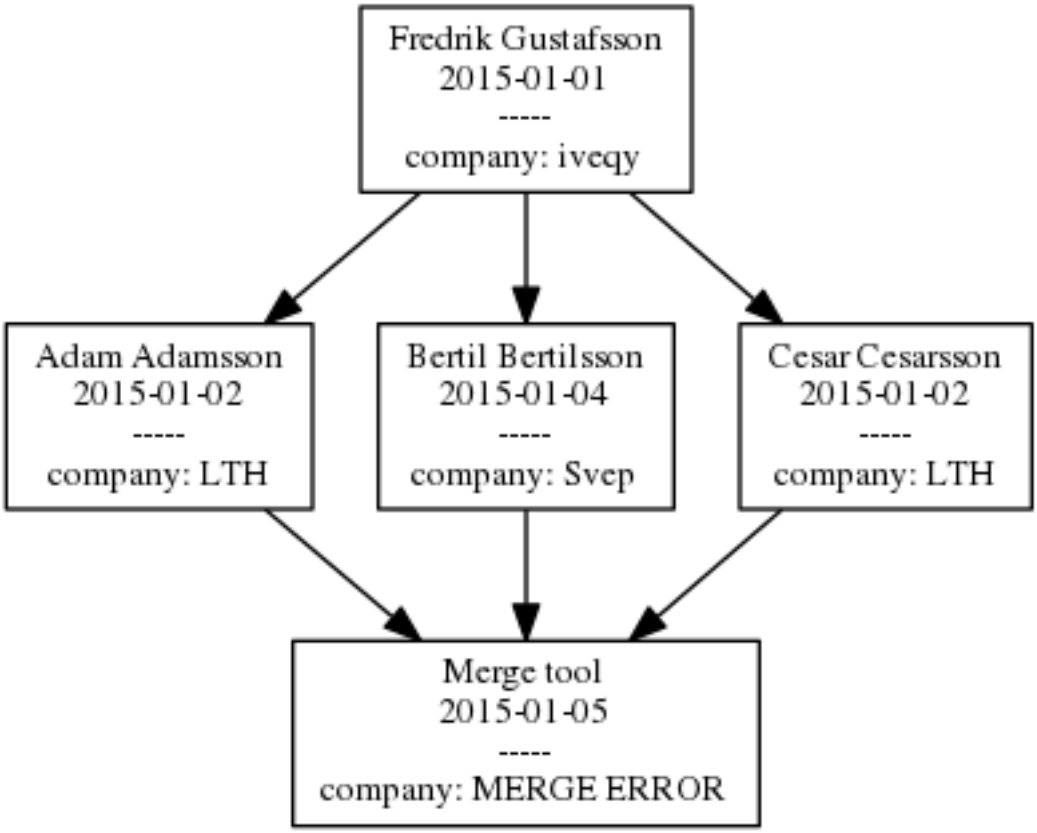

Figure 18: Use case 11

This use case will examine the case where a merge has three parents but two of the parents doesn't conflict.

This problem is just like use case 10, section 3.1.11 except that two of the changes have the same value.

One way to solve this would be by vote, since two values are the same, that value might be more correct than the single one. This is also a good example on when the merge two and two changes, described in use case 10, section 3.1.11, wouldn't work. That is, it would always give *MERGE ERROR* as a result, but the correct result for this use case when using the vote method would be *LTH* (although this paper argues that the vote method is giving the wrong result in this case and shouldn't be used).

Consider three salesmen at a multi-day long conference with no internet connection. They all work at the same company and the all meet the client David. The first two salesmen meet David on the first day, noting that the company information about David is wrong, so they both update their

system stating that David works at "LTH". However, during the conference David is recruited by "Svep" and when the last salesmen meet David on the last day of the conference, he enters "Svep".

When all three salesmen get access to internet again and are able to sync their copies of the database, the vote method wouldn't give a correct result.

The result for the use case in figure 18 should therefore be a merge error.

If the time stamp of the changes is taken into account. Let's examine if the vote method works if it also is a requirement for the majority value to also be the value of the latest change.

That would work for the example with David, leaving that to be a merge error as well. However, one problem with human entered information is that there's a delay from the point the information is received to it is entered into the system. This could possibly be a quite long delay. For example, that one of the salesmen that meets David the first day, just record this information into his notebook and then enter it in the computer on his way home. That would make the time stamps of the changes out of order with the time stamps of when the information was received by any of the salesmen. Hence this solution wouldn't work either.

So it can be concluded that the voting method shouldn't be used in this case since it has clear drawbacks.

3.1.13 Use case 12

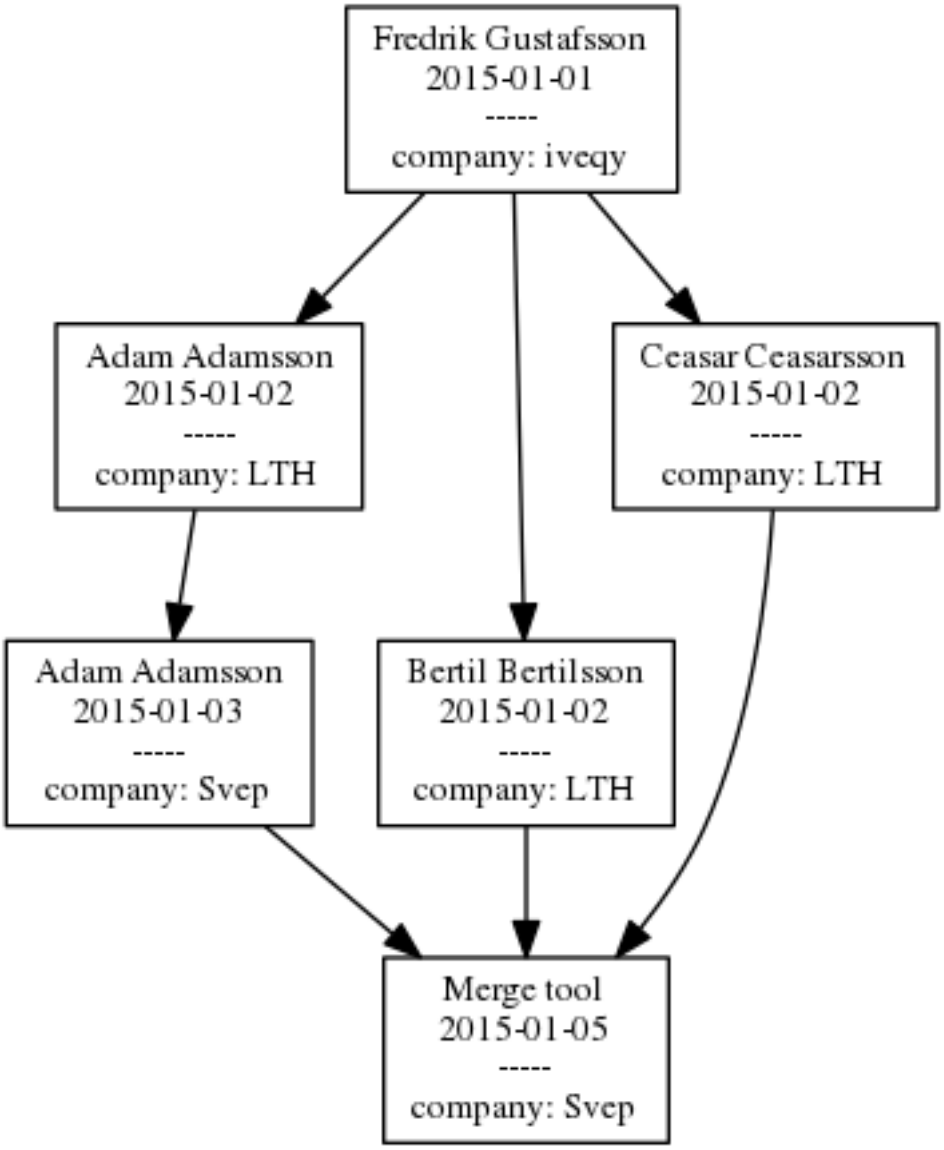

Figure 19: Use case 12

This use case is a combination of translation and multi-parent merges.

The use case in figure 19 is similar to the one in use case 11, section 3.1.12. If the voting approach discussed in use case 11 should be used, the result here would be "LTH".

However, the difference between figure 18 and figure 19 is that figure 19 has more history to take into account.

Using the translation techniques from use case 7, section 3.1.8, here would translate "LTH" to "Svep" and three leaves with the same value would be found. That would make a trivial merge with "Svep" as a correct result.

The voting method is delivering a less safe result and should therefore be deemed less successful than the translation method. This is because information doesn't have a weight and therefore two identical data isn't more correct than one data that differs.

It can be concluded that for the multi-parent case, voting would give a different result than a merge done with the translation technique.

3.1.14 Use case 13

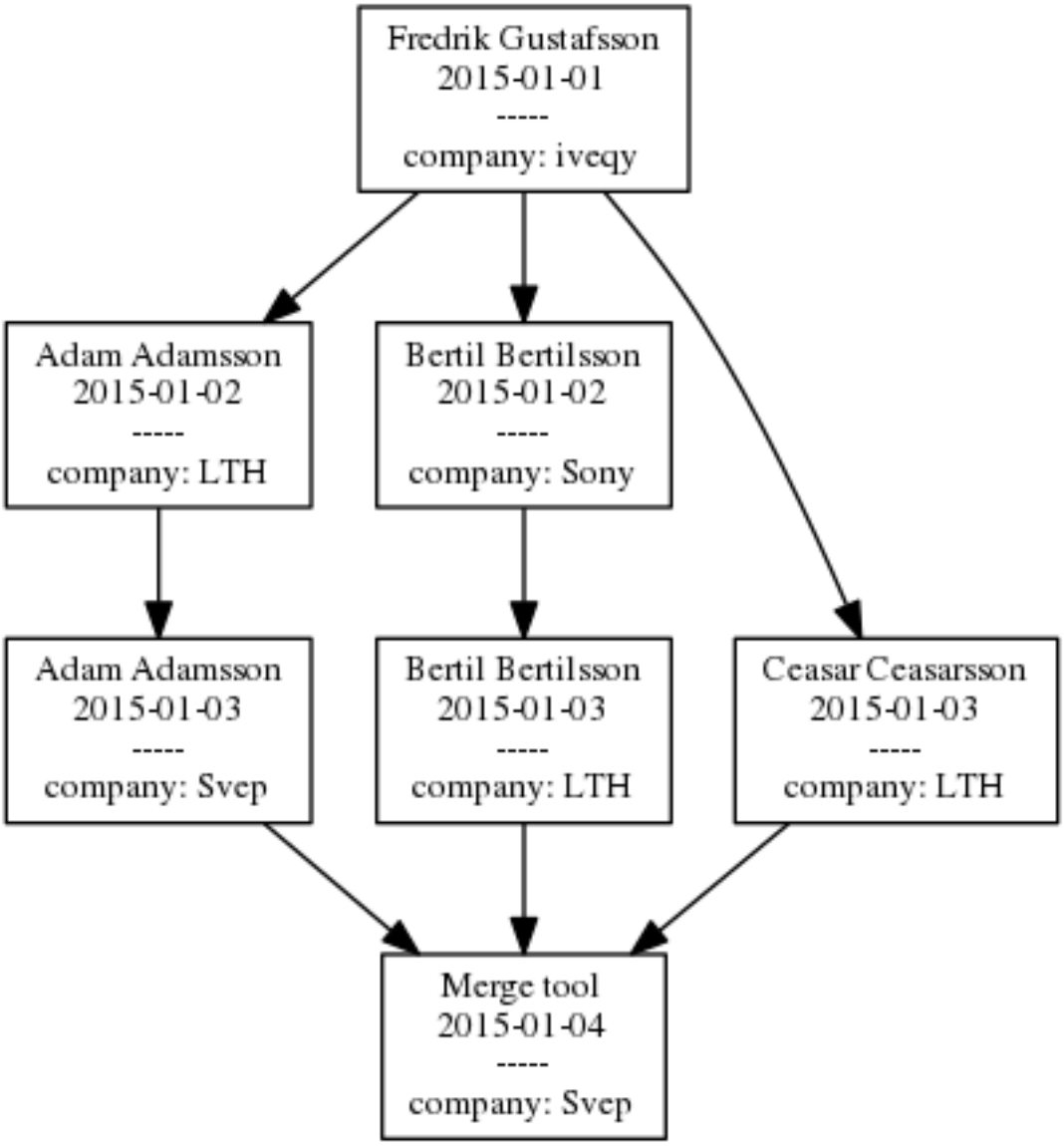

Figure 20: Use case 13

In this use case, ignoring (some) history is examined.

The use case in figure 20 is similar to the one in use case 12, section 3.1.13.

Let's look at the change with value "Sony". If the only thing interesting is the current value, it can probably be safetly ignored. However, in the case of a CRM system, the history would probably be interesting. For example, knowing a contacts employment history good give good insights in how to

interact with that contact.

In use case 7, section 3.1.8, information is added. Here information will be ignored. That's probably not a good way to solve this.

An argument could be had about adding "Sony" to all branches but that would be wrong since each change is building on its parent. Hence the two changes with value "LTH" that has parent "iveqy" also implicit holds the information that company changed from "iveqy" to "LTH". Inserting "Sony" in between would destroy that information.

History could be important in some cases for a CRM tool. But to be able to use the history, it would have to be linear. Since all merges combines two branches into a common branch and does not alter the history to make sure that the history agrees, the history would be useless. Therefore, the solution here is chosen to be "Svep". Other tools that values the history in another way, would find this to be the wrong approach and instead favor a "MERGE ERROR".

It can be concluded that sometimes not all history must be used, but some can be ignored.

3.1.15 Use case 14

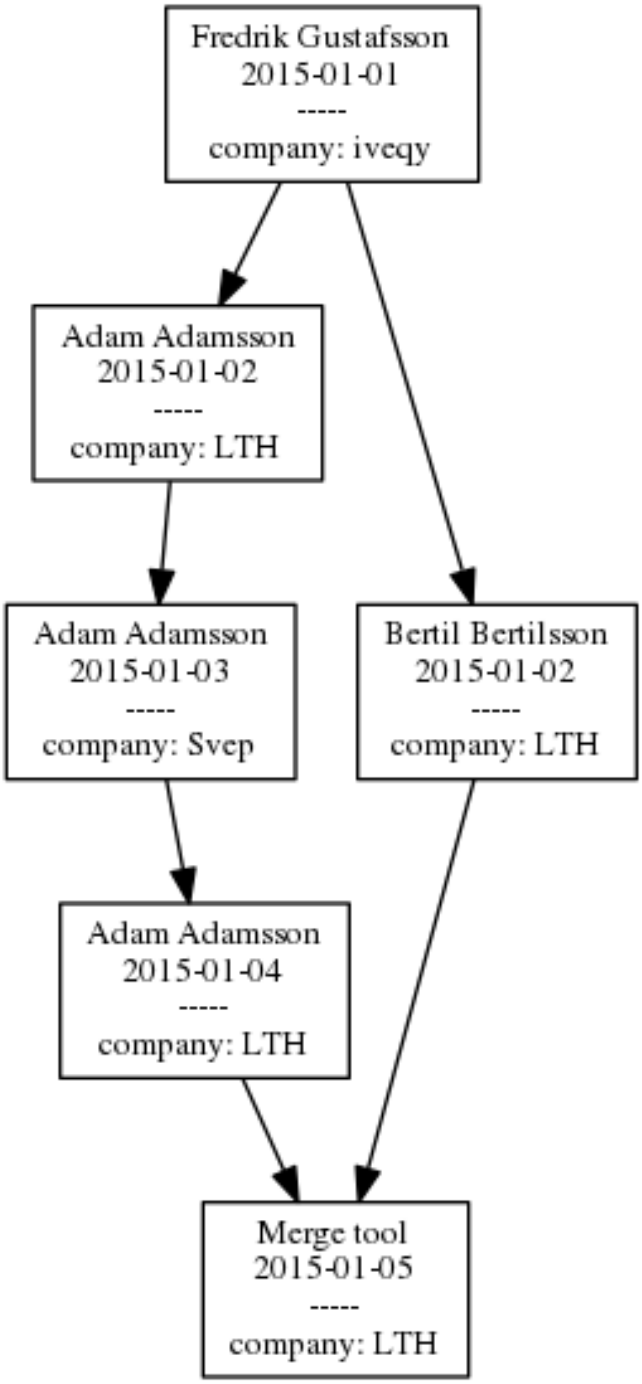

Figure 21: Use case 14

This use case will show how an erroneous implementation of translations would result in the wrong merge result for some values.

This case is very similar to use case 9, section 3.1.10. It would be possible to add a change with the value "Svep" and one with "LTH" to the right hand branch, then end up with two leaves both with the value "LTH" that could trivially be merged.

However, that is way too complex, the case starts with two leaves with value "LTH". That does however not take linear history into account as discussed in use case 13, section 3.1.14. If history is needed, here's actually a use case where the history can be merged!

Trying to implement a merge algorithm for this case could however be a bit tricky. The problem being to know when to stop translations, since the algorithm should both start and stop with "LTH".

In this particular case two stop conditions are possible, either stop when the right hand branch has equally or more changes than the left hand branch or stop when the right hand branch has reached "LTH" again. Neither those criteria will work for more complex cases, for example when both branches needs to translate [6] or when a translation contains several changes with the same value.

Since history merging isn't a top priority for this paper. No further exploration is done here.

Since both leaves are identical this use case would word fine with a simple merge algorithm, for example the one used in use case 1, see figure 8. Use case 14 can be solved without looking at the history. That means that for use case 14 to be a good test for a merge tool, it must tell the merge tool to specific use the translation merge algorithm or make sure that the tool chooses the merge algorithm for use case 1 above the merge algorithm for translation merge.

It can be concluded that translations must be limited and is not always the go to tool for all history enhanced merge algorithms.

[6] It could be argued that both branches never should be translated since it could result in an evil merge.

3.1.16 Use case 15

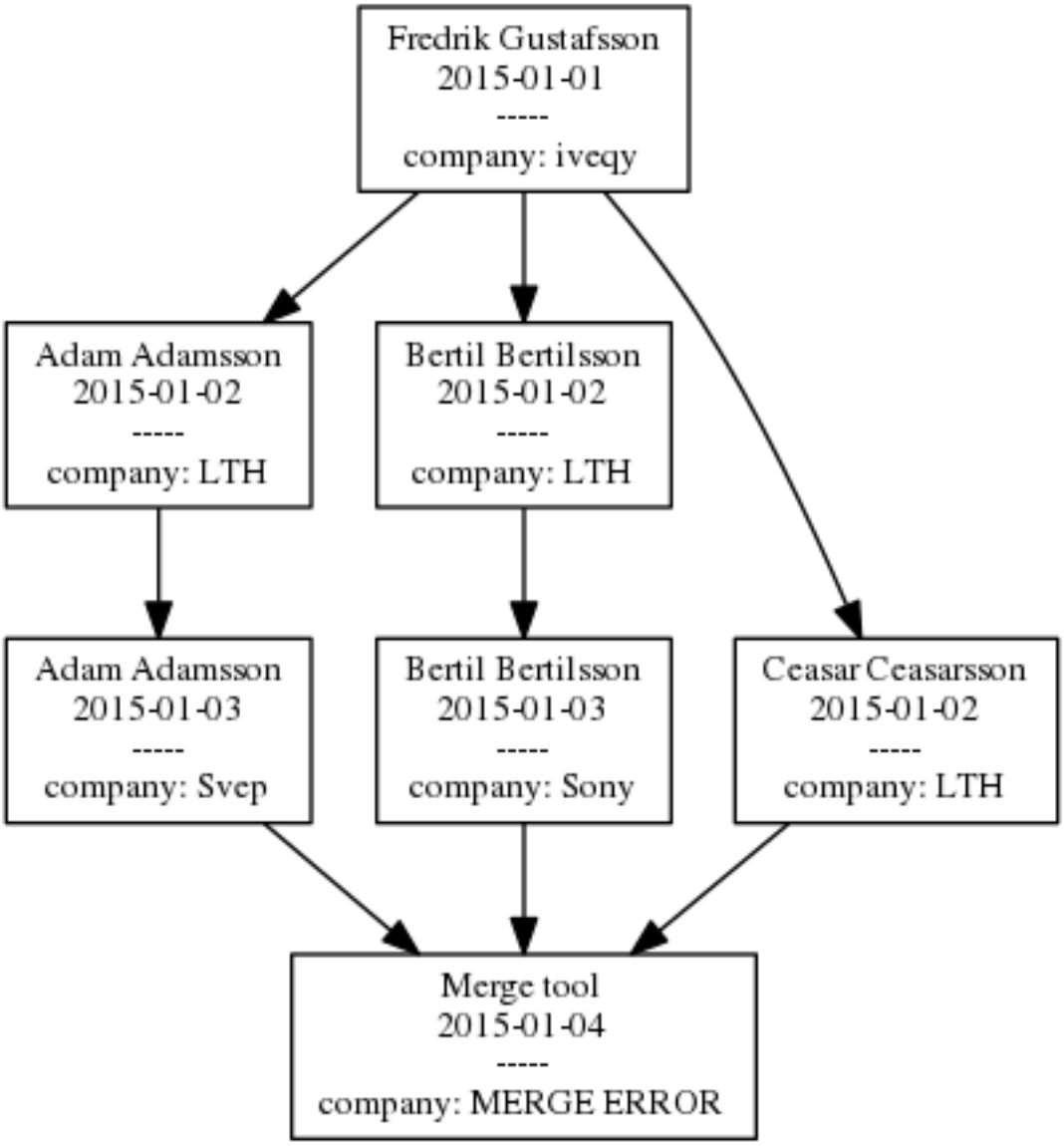

Figure 22: Use case 15

In this case it will be shown where translations used before voting with multiple parents can lead to undefined results.

In use case 11 in 3.1.12 it was concluded that voting is an approach that wasn't suitable. This case, seen in figure 22 would serve as a tricky voting case. "LTH" could be translated to both "Svep" and "Sony" and whichever is done, opens up for a vote result. Hence this case would be a good case to

show an implementation flaw in a voting implementation. The correct result here would be a merge error, because even if it's known that "LTH" isn't the correct result, it's impossible to know if "Svep" or "Sony" is the correct result. The tool doing the merge shouldn't throw away the information that "LTH" could be translated. This is important information that can ease a manual solution to this merge. Instead all the changes with "LTH" should be merged and after that merge two divergent branches will be seen. This is defined in this paper as a partial history merge, where a part of the history can be merged. It will still result in a conflict for the user, but that conflict will probably be easier to solve.

A merge tool doesn't need to have the two extremes that it either successfully do a merge or completely aborts. Sometimes, like this example shows, it's perfectly fine to partially succeed with a merge.

It can be concluded that, even if voting where a good method is used, it would have some troublesome cases.

3.1.17 Use case 16

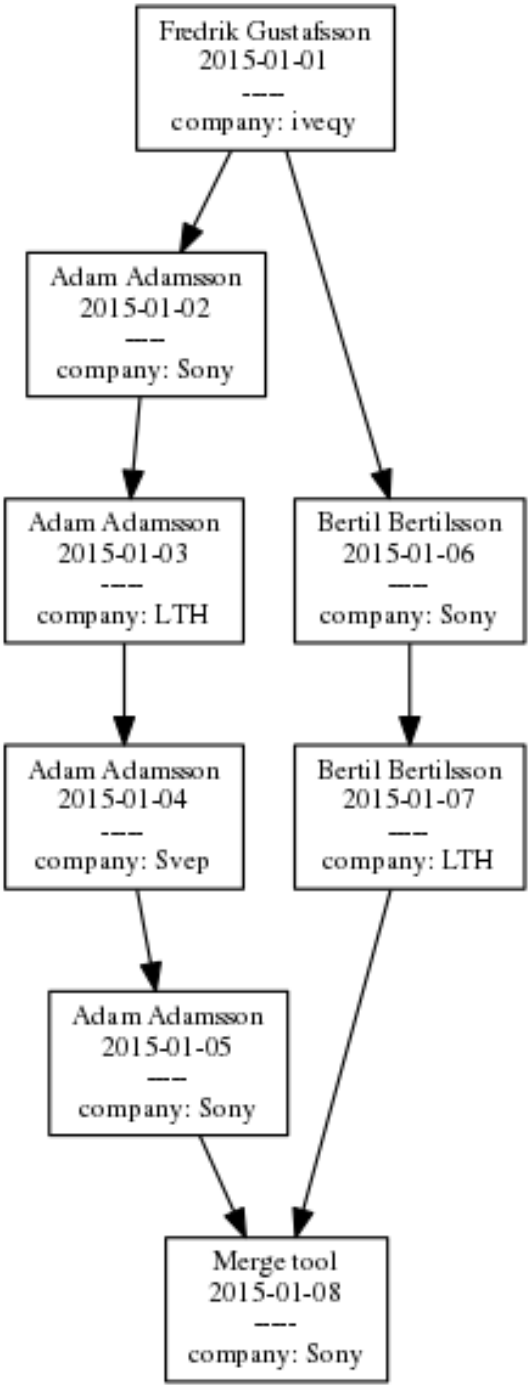

Figure 23: Use case 16

This use case will show that for some cases translations and partial history merges could lead to the same result.

The merge problem in figure 23 has a clear solution. Previous use cases has shown that both translation and partial history merges would reach the same conclusion.

Whenever partial history merges are possible, they are advised to be used. That would ease other merge approaches, which could be more expensive to calculate.

A partial history merge would simply result in a single branch history in this case.

Note the dates of the changes. They are irrelevant for the result. Not all tools agree with this.

It can be concluded that partial history merges can be good, but not always needed since translations gives the same resolution, and are already needed for other use cases.

3.1.18 Use case 17

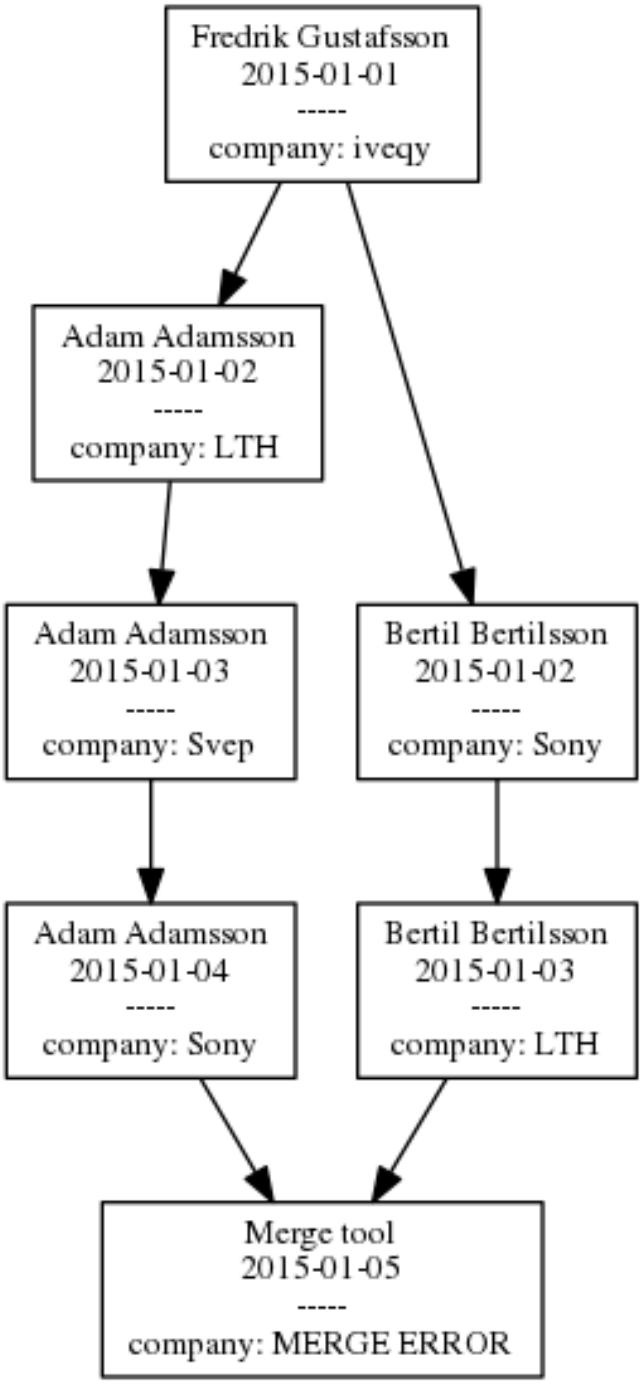

Figure 24: Use case 17

This use cases will show that translations don't always work.

The use case 17, figure 24, looks to be an easy solve for the translation approach at first. Trying to translate the left hand branch would result in "LTH". Trying to translate the right hand branch would result in "Sony".

It's obvious that this approach doesn't work here. One way to solve this case would be to look at the change dates and try to see if the both changes with "Sony" match better or worse than the two changes with "LTH".

Looking at change dates is however not safe as discussed in use case 11, section 3.1.12. The correct solution for this should therefore be a "MERGE ERROR".

This case is also interesting from an implementation point of view. It's not totally clear when to stop translations here. The rule here would be that a translation found from one branch cannot be applied to the same branch again, since that would result in an infinite loop.

It can be concluded that translations can be tricky when hitting circular dependencies.

3.2 Characteristics of data

To be able to do a good conflict detection and merge resolution, it's important to know as much as possible about the data being examined.

In this section some of the characteristics about CRM data will be analysed and presented. CRM data is different from many other types of data, like source code or UML data, but close to other types of data, most obvious different types of registers.

First a way of determining how different two words is will be presented, then a discussion about what, extra, information is needed to be able to do a good merge. The difference between CRM data and source code will be analysed in depth and then this section will finish with a discussion about how to format data.

3.2.1 Edit distance

The edit distance, often in the form of Levenshtein distance [6], is the number of operations needed to convert one word into another word. This is useful to see if one word is similar to another word. If two words are very similar it's possible that one of the words is a misspelled version of the other word. There are many other techniques for comparing words, specially names which can be spelled quite differently have phonetic representations that can be used to find similarity. This is useful for finding if two person entries in a database refer to the same physical person even if the spelling is different. Algorithms for doing these kind of detections often have higher than 90% accuracy [13].

Since 90% correctness isn't enough for what a CRM system need as accuracy, these methods should result in a merge conflict instead of trying to do an automatic merge. That is, techniques that cannot guarantee a 100% accuracy shouldn't be used for CRM data since there's no way to verify that the result is sane. It would be possible to give a merge proposal, which is good but not considered in this paper.

An example of this kind of merge conflict can be seen in Case 2, chapter 3.1.3.

It is important to notice that the Levenshtein merge is just unsuitable for CRM data. For other types of data where the result could be validated, for example with a compiler, this merge approach could still be a good candidate.

See section 3.2.4 for a discussion about why CRM data need higher accuracy than source code.

3.2.2 Unit of comparison

An important part of the paper *Requirements for Practical Model Merge - An Industrial Perspective* [8] is the discussion about unit of comparison. The unit of comparison is the unit of data that should be compared to other units of data. The extreme being that the whole database is seen as one unit of

data and hence any change would result in a conflict and the other extreme being that each data field is seen as a unit of comparison which might lead to invalid combinations of data as shown in chapter 3.1.6.

A too big unit of comparison will lead to many conflicts and made it harder to solve conflicts since they can be rather big.

In this paper a flexible unit of comparison will be used, where the size is scaled dependent on the data examined. This is possible since the data is highly structured and the structure of the data is known beforehand.

3.2.3 Useful information

Doing a merge is often easy if enough information is given. Often the problem is that enough information isn't available. It is probably not a good idea to demand enough information for each merge case. Enough information can require the use of external resources which could be unreliable but more possibly nonexistent.

For example, there's a conflict between two addresses for a company. The information about which address that is correct is probably available in some register somewhere or on the company website, etc. It's not realistic for the algorithm to have access to external data sources for all cases.

Instead a merge algorithm should do its best job with the data present in the system that is doing a merge. To know what data that is present, a decision must be made to which data to save.

If all data entered into the system is saved, the algorithm will have the best possibility to correct a merge. In this paper it will be shown that saving the edit history give extra information needed to solve complex merge cases.

This differs from the techniques used in *Requirements for Practical Model Merge - An Industrial Perspective* [8] where history is thrown away deemed no longer usefull.

3.2.4 Difference between source code and CRM data

The most common distributed data systems are code version control systems. There are a number of mature tools on the market to handle simultaneous updates to the code.

Common tools like CVS, Subversion and git uses row based merge algorithms where each row is its own entity to merge.

CRM data is not code and should not be treated that way. Code have strict rules on how it should look, its syntax. After a merge a syntax check could be performed with a compiler [7]. That is, code that compiles have probably had a somewhat good merge.

[7] A compiler is a program that converts a textual representation of computer code into binary represented instructions for the computer

Even if a merge error is introduced that won't get recognized by syntax checking, there are (should be) test cases that ensure that the code behaves as expected. For CRM data the options to check if a merge is correct or not is somewhat more limited.

Code uses syntax rules, but for a CRM system it has to use data, see section 3.2.3. That's more expensive and not always possible to even get hold of. For example, two CRM records are merged and ends up with a record involving a city and a zip code. There are no rules that tells if the city can have that zip code or vice versa. However, there are records to look up if this is a correct combination.

Finding records for zip codes and cities may be trivial, but what about phone numbers and people? Here it's not even hard but impossible since phone records don't have coverage or correct information for the CRM system.

A person that could be reached at a phone number that could be registered to the company that the person works at for example.

Given these difference between code and CRM data, different prioritizations has to be done. It's much more important for a CRM merge algorithm to be correct, hence it should give false merge errors rather than possibly false information.

3.2.5 Data formatting

Some errors are trivial, the information in the two different records being merged might be the same, but due to the computers inability to understand the information it still results in a conflict.

For example, the phone number "0733 608274" and "073 36 08 274" will show a conflict in a comparison. This can be solved by always having the data formatted in the same way, either at the time it's typed into the system or by the time it's compared.

Comparing data that first is transformed by (to the user) hidden rules are error prone and confusing. That's why data should be on the correct format already when entered into the system.

This paper won't go deeper into different technique to solve this.

3.3 Types of conflicts

Conflicts can be divided into five different categories, only one change, matching changes, unrelated changed and conflicting change. In this section they will all be analysed in detail and in case of conflicting change, different techniques on how to solve a conflict will be presented.

3.3.1 Only one change

Let's assume that two users are involved, Adam and Bertil and that these two needs to synchronize their work from time to time.

The simplest case is when Bertil has done a change, but Adam hasn't. When they now synchronize their changes, Adam will have all Bertil changes and no conflicts would have occurred since Adams changes are done following the same version as Bertil has.

Matching changes

When Adam and Bertil both have done the exact same change, a merge will be trivial since a merge change (a change with more than one parent [8]) will have the same information as any of its parents, since all parents have the same information.

This might seem to be a strange case, but it's highly practical when dealing with merge conflicts later on because it requires that conflict detection is present, which is one of the requirements for a good merge tool.

If a conflict somehow could be translated into this case, a merge would be trivial. Therefore, this is an important scenario. This is called a trivial merge.

Unrelated changes

An unrelated change is a change of one value, that isn't related to a change of another value. The values have no logical dependency and a merge between two changes should be trivial. An example of this can be seen in use case 6 in section 3.1.7.

In use case 6, see section 3.1.7, Adam changes the age and Bertil changes the company. Since the age isn't related to the company in any way, a merge would be trivial.

This is only if an item based merge is done. If the merge would be document based, there would be a conflict. Since the documents are bigger than items since a document contains multiple items. The key issue here is if a single item is considered to be the unit of comparison or if a document is considered to be the unit of comparison.

An example from the source code world. If Adam change a row in a text file and Bertil change another row in the same text file, a merge could be successfully done.

However if the merge tool had used a whole file as unit of comparison, there would be a conflict.

[8] A parent is an entry which precedes and other entry

3.3.2 Related changes

A problem with unrelated changes is that there needs to be a way to know if they are unrelated or not, see section above on *Unrelated changes*. For example, a city is related to the zip code. Change of just the zip code and not the city could, but doesn't have to, result in invalid data. Great care has to be taken doing merges with dependent data since the merge algorithm doesn't know which relations are valid or invalid. A way to prevent this issue is to not do item based merges but only document merges, see section 3.3.1. Another way is to identify the dependent relations that are present in the data and provide that information to the merge algorithm.

Since CRM data is a highly structured data with known data types it is possible to identify these dependencies. This is a manual process that only needs to be done once.

In use case 5, see section 3.1.6 a merge with two changes can be seen. Adam who has changed the city to "Lund" and Bertil who has changed the zip code to "226 34". Since the merge algorithm knows that city and zip code are dependent a merge error is the result instead of a successful merge that it would be at first sight.

If an unrelated merge strategy had been used, the result would have been "Lund" and "226 34". That could be a perfectly fine answer, but there's no way for the merge algorithm to know that.

As discussed in section 3.2.4 there is no way to verify if it's a valid result or not, hence great care must be taken to prevent invalid information being the result of a merge.

3.3.3 Conflicting change

Changes that have values that conflict with each other. The most simple case is a value that is totally different from a value in the other change. See use case 1, section 3.1.2, where the company for one record is changed by Bertil and Adam to two different values.

Note that Adam is doing his change one day later than Bertil. This shouldn't have an effect on the resulting merge. Since the time information is entered isn't the same as the time the information is received by the user.

Since a CRM system is dependent of when the data typed into it, (in this case) the information always has a certain delay before being entered into the system. This delay will vary depending on a huge number of different factors. For example, Ceasar is talking with a customer, he gets information that he needs to change about the customer in the CRM system. But before he does that, he takes a Swedish fika, and the entry timestamp is delayed by 30 minutes.

Adam on the other hand, doesn't have fika at the same time. Adam is working at the same company as Ceasar. The customer remember that he

gave Ceasar wrong information and call the company again, now speaking with Adam. Adam enter the correct information in the system, and later when Ceasar is done with his fika he has no idea that his information now is deprecated.

The locking approach [9] does not work here since a distributed system is used, hence there no way to communicate the lock between different parts of the system. A centralized system would be able to solve this with locking, but then other problems with mobility and connectivity will arise.

There are a number of different scenarios with the same consequences. The result being that listening to the timestamp when the data is entered into the system is a bad indicator on how correct the data is, see use case 1 in section 3.1.2.

Another trivial error is misspelled words. The treatment of spelling was presented in 3.2.1.

When two or more fields are conflicting, it's not more difficult to merge them than if one field is in conflict. See use case 6, section 3.1.7 for an example of this. To solve this, just threat each field separately.

This is not the case if there's a dependency between the two different fields conflicting. In that case the unit of comparison should increase to be big enough to fit all fields dependent on each other. In use case 6 however, the fields are independent. For a discussion about how to handle the dependent case, see use case 5 in section 3.1.6.

More information should lead to better merge results. Each change is information, so instead of throwing that information away keep that information stored until the merge is done. This can be seen in use case 7, section 3.1.8.

Now the question is, how can this extra information be used to result in a better merge?

Translations

This paper propose the solution of translation lists. A translation list is a list of changes from one value to the next that are stored. All those translations can then be used on any other branch that are to be merged until they go in a circle. A good example of this can be seen in use case 9 in section 3.1.10. Where a translation (or two following translations) can result in the same value ("LTH") as started with, resulting in a loop.

Sometimes more than one translation step needs to be taken. This can be implemented in two different ways. Either each translation is one step long and multiple steps can be applied or one translation can in itself be

[9] Locking means that a user locks a record from editing by any other user, until the user holding the lock releases it again. This make sure there's never two conflicting update since the versions never diverge.

multiple steps long. Again use case 9 in section 3.1.10, is a good example of this.

Let's say that more than two changes has been done in parallel as can be seen in use case 10, in section 3.1.11. An easy approach here would be to compare two changes at the time, merging them down to one change and then continue with the next one.

Voting

If two changes are equal as in use case 11, section 3.1.12, a voting method can be used. Let each change have one vote and use the change with the most votes.

This is unfortunately a bad merge strategy that shouldn't be used. Since multiple equal changes do not need to add weight to a change. If two changes are done that are equal and then the information changes resulting in a third change with the correct value, then a merge would have two versions with the old incorrect data and one version with the new correct data. Still, using a voting method, the old data would win since there are two changes for it. An example can be found in 3.1.12.

Use case 12 in section 3.1.13 explore this even further, when translations and voting methods are combined and different results occurs depending on which method to start with. This makes it clear that voting is a bad method for a multi parent merge strategy.

This isn't a method used by the custom implementation by reasons discussed in 3.1.12.

History recording

Most systems today are just interested in the current information. That gives a lot more freedom when designing a merge algorithm. Consider use case 13 in section 3.1.14. If "Sony" can be ignored, this merge is easily solved using previously discussed techniques, however if the information that "Sony" has been a previous company for the person should be stored, this should result in a merge error. Chances are however that "Sony" isn't a correct value at all, but an error by the person doing the data entry. The value of recording this is hard to measure, but it's probably very low. The question is not if the history should be kept, but if the history is important for the user. For example, if the history of a contact's different companies should be recorded, the history needs to be correct as well and there need to be a history merge to result in a correct history as well as make sure that the latest version is correct.

Starting from the oldest ancestor, the history should be tried to be merged step wise. This would solve some otherwise tricky situations. Using translations to solve use case 14, section 3.1.15, could be tricky. Even easier

would be to merge first generation children first, it would be a trivial merge, and then only a linear history is left.

Adding multi parent merges into this, as in use case 15, section 3.1.16, shows that translations isn't the most efficient way to solve this problem.

There are times when a translation approach could fail even for a merge case with two branches. This can be seen in use case 17 in 3.1.18 where applying translations to the two different branches result in a different result. This could lead to data loss and is hard to detect in an efficient way. Since data correctness is very important for CRM data merges, this should be flagged as a conflict.

3.3.4 Unknown relation

As seen in use case 3, in section 3.1.4, two changes could need a merge resulting in a non-conflicting merge, see section *Matching changes* in section3.1. To detect this case avoids duplicates in the database.

Knowing which changes that are to be merge is crucial, otherwise the merge problem isn't even known to exist! If two or more changes has a common ancestor, it's pretty clear in this case that they should be merged. Assume that all changes with a common ancestor should be merge or flagged as merge errors. However, it's not clear that changes that don't have a common ancestor should be merged. They probably shouldn't, but sometimes they should. If two records are meant to represent the same physical person, but are entered in the system as two different accounts, they should be merged to one.

It's hard to identify which ones should be merged or not, but since this isn't a merge, but a suggestion for merge. The need for a correct result isn't that important and many of the techniques deemed too unsafe in chapter 3.3.3 can be used to flag possible merges.

The trivial case would be two changes, or rather entries, that are identical with all data fields given a value.

For example, if one record is for Bertil working at Tetra Pak and another record is for Bertil working at Tetra Pak, they should be flagged as a possible duplicate but they do not need to be a duplicate, since there is a chance that there are two people named Bertil that work on Tetra Pak.

However, if all fields are given a value, they may still be two different persons, but the systems resolution isn't enough to keep them apart anyway[10], so it is safe to assume that they can be merged into one entry.

Records with partial matching fields should be considered for a merge candidate. Partial matching can be that some fields match and some fields don't, that some fields match and some other fields are compared to an

[10] Let's assume that a system records only first and last names of a person. There might be two different persons with the same first and last name in the real world, but not in the system. The systems resolution isn't enough to keep them apart.

empty field. This would possibly be a pretty expensive operation if all records should be compared to all other records and when two records are compared, all fields in those records should be compared as well.

For merging two entities without a common ancestor, a 100 % match for all fields are required to be sure that data isn't lost. However since this isn't a merge, but a suggestion for a merge candidate, there's no need for a 100 % match. Instead the edit distance can be used to find fields that almost matches. For example, the Levenshtein distance could be used to compare different fields to each other.

Levenshtein distance is only one method of comparing the distance between two words. There are different algorithms that are more suitable for names. These algorithms and their abilities are outside the scope for this document. An example of this scenario can be seen in use case 4 in section 3.1.5.

To figure out if two entities should be merge candidates or not, a given is that if many fields are equal or almost equal, the higher chance of the two entities are good merge candidates. However not all fields should have the same weight.

When comparing fields, they should be weight against each other. For example, two records with the same person name is more likely to be a good merge candidate than two records with the same company name. This is because person names tend to be, at least to some point, unique while many people can work at one company. Even more people live in the same city, so finding two records with the name Bertil and the city Lund should probably not result in a merge candidate. However, two records with the name Bertil and the company Arimans Café might. Two records with the name Bertil and the company Tetra Pak might however not, since Tetra Pak has more employees that Arimans Café.

The model considered in this paper lacks information about the company size. The weight for different fields probably has to be determined by trial and error with real data.

Two changes that are resolved as a non-merge, would possible be detected again, next time a scan is done for finding a potential merge. To prevent this, even merge conflicts resulting in a non-merge result need to be recorded.

3.4 CRM applications

There are many CRM applications on the market today. In this report the leading applications have not been chosen but instead three applications that differs from each other at an architectural standpoint.

Most of the leading CRM tools on the market today resemble at least one of the chosen tools. First a tool named Lime Easy will be evaluated. Lime Easy is a client-server style type of tool. Then there's Deko the CRM that is one of few distributed CRM systems on the market today and last

Table 1: Lime Easy results

Use case	Expected result	Result	Verdict
1.	MERGE ERROR	svep and LTH	incorrect merge
5.	MERGE ERROR	226 34 and Malmoe	incorrect merge
6.	svep and 31	svep and 31	correct
8.	svep	svep	correct

there's Fat Free CRM that is to represent the web based systems that are getting increasingly popular in recent days.

3.4.1 Lime Easy

Lime easy does distribution by "checking in" and "checking out". A workflow where the database on the network is copied in whole to the client and when the client is put in offline mode it's using its local database.

When reconnecting to the network with the master database again a check in is performed, that is a synchronization between the both databases. There could probably be conflicts here, when a conflict is discovered, the latest entry is chosen.

The latest entry is not necessarily the correct one, especially not since the timestamp is set when the data is entered into the system and not when the data is acquired [11]

Lime Easy is doing a poor job of solving merge conflicts, table 1. The time of entry is not the same time as the time the data is correct, if the frequency of data change is much lower than the frequency of the data being entered into the system, the entered timestamp would probably mostly be correct enough.

If the number of customer entry points to a company is small, for example one account manager handles one customer, the risk of collisions will also be small. It could be argued that since Lime Easy is targeting small businesses the need for merge solving isn't that huge since conflicts will be rare. Several persons will seldom work on the same customer. Each customer will seldom be updated and if several persons do work on the same customer their work will be far apart in time. Also if the salesmen are touching base with the master version of the database daily instead of monthly the conflicts will probably be rather few.

Although this could be a correct guess, the problem here is that data

[11] Let's assume that there's something like correct data come from one single source and should be registered in the CRM system. Since the CRM system could be updated in different ways, due to it having multiple users, the data could arrive to the CRM system out of order. Therefore, the timestamps where when the data arrive to the CRM system is not guaranteed to be in the same order as the data is produced from this imaginary single source.

can be destroyed. To prevent this, conflict detection is needed, although an automatic merge algorithm might not be needed.

When a conflict occurs, as in use case 1 in chapter 3.1.2, Lime Easy would simply choose the latest entry and therefore possibly loose data. Duplicate data isn't found either but are both inserted into the system, this problem will be analyzed in use case 3 in chapter 3.1.4.

The huge problem with Lime Easy isn't the lack of merging but the total lack of conflict detection. Even if conflict detection actually is done, it's not communicated to the user and therefore in practice Lime Easy does lack conflict detection.

To summarize, a good merge algorithm needs first of all to have good conflict detection. If no conflict detection is present, it's impossible to even know that a merge is needed.

3.4.2 Deko the CRM

The Apache CouchDB has a built in replication and synchronization mechanism, that is used by Deko. CouchDB is a document database, meaning that it stores documents and it also find conflicts of different documents. Being a document database, it doesn't have any knowledge about the data inside the documents, but can detect conflicts on a document level.

If there's a conflict, CouchDB will choose the version with most edits first and in case of a draw it will choose the latest version.

However, CouchDB will always notify the application about the conflict and will provide both conflicting documents for access if the application wants to.

Deko the CRM is using the capabilites of CouchDB for doing conflict detection. If a document is conflicting with another version of that same document, Deko will show a conflict and let the user merge the two versions.

Merging on a document level is hard, because there's no diffing functionality so there's no way of knowing what differs between two versions of the document. Also a document can be quite big, usually containing for example all information about a person.

Deko the CRM is finding a lot of conflicts, table 2. This is good since the risc for data loss from a automatic merge like in Lime Easy isn't possible. However, since the merge conflicts are both big and hard to know which items actually conflicts. Each conflict resolution is pretty expensive since it requires manual work.

As seen in table 2, use case 1 is resulting in a conflict as it should but use case 6 is also resulting in a conflict even if it shouldn't.

If the number of conflicts are big, it can be imagined that the human way

Table 2: Deko the CRM results

Use case	Expected result	Result	Verdict
1.	MERGE ERROR	conflict but LTH as winner	incorrect merge
5.	MERGE ERROR	conflict but 226 34 and Malmoe as winner	incorrect merge
6.	svep and 31	conflict but svep and 30 as winner	incorrect merge
8.	svep	conflict but svep as winner	correct

to resolve each of them is error prone [12] and this can lead to invalid data being stored.

To conclude, Deko the CRM is doing conflict detection and are presenting this to the user. However, Deko is doing a poor job of helping the user solve the merge and is also giving a lot of false positives due to a too big unit of comparison.

Using the same arguments as for Lime Easy, assuming that Deko is used by a small company with conflicts seldom to occur, this is still probably a better approach than Lime Easy since Deko won't risk losing any data. However in a bigger setting, the false positives and the expensive merges would make Deko hard to work with.

In the case where conflicts are rare and the Lime Easy solution already is deemed good enough, it's not sure that the increased complexity of Deko the CRM is worth having. That is, in Lime Easy the users don't have to think about merging, but in Deko they have. In Deko the CRM the data sharing problems are visible, that might be a first step, but it's also an extra load on the users.

Two different versions need to be presented to the user with the differences clearly visible so that a manual conflict resolution can be done.

[12] Humans make errors, with a huge number of conflicts the human solving them are probably reducing his/hers attention after a while and gets sloppy. This of course doesn't need to be the case but in that case it would probably be time consuming

3.4.3 Fat Free CRM

Fat Free CRM isn't the most popular web based system, but many of the available web based systems share the same problems and solutions. Although the use of web technologies does not imply that a conflict detection and merge behavior needs to be implemented in a certain way, they often are.

The solution for offline work in these web based applications is that it is nonexistent. They instead rely on the widely spread internet connectivity and as long as there is an internet connection it's possible to use the application.

Since each client has such direct communication with the server, the time for data to diverge is often very small. Since this time is so small, it's often thought that conflict detection isn't needed because of the low frequency of conflicts.

Table 3: Fat Free CRM results

Use case	Expected result	Result	Verdict
1.	MERGE ERROR	LTH	incorrect merge
5.	MERGE ERROR	226 34 and Malmoe	incorrect merge
6.	svep and 31	svep and 30	incorrect merge
8.	svep	svep	correct

The results from the tests can be seen in table 3. The problem here is that conflict detection is nonexistent and that the unit of comparison is a whole page load. Since not only the changes but the current content of a view is sent, it's impossible for Fat Free CRM to know what has changed.

Technically there could be done a server side diff [13] to know this, but then the version to diff against must be known and such history support isn't existent in most web based tools.

Both a simple use case like use case 1 that should conflict and a simple use case 6 that should result in a merge fails.

Fat Free CRM is suffering from both a big unit of comparison and its lack of conflict detection. The version saved is always the latest version no matter if that version has the current version as an ancestor or not.

The risk of data loss is present, however it is mitigated by the short times each client holds its data to itself.

[13] A diff is the difference between two versions

4 Design and results

In this chapter it will be shown how to obtain the desired results for the different use cases previously analyzed. The merge algorithm used will be analyzed in detail and the efficacy of it will be discussed.

From the use cases, five important requirements have been collected.

- Conflict detection
- Merge capability
- Flexible unit of comparison
- Access to change history
- Access to dependencies between fields

Not all use cases require all five requirements to be used at the same time. But together they form a very good base for being able to solve all the use cases.

For being able to do a merge, it must first be known if there's a conflict between two versions. For this to be useful it's not only of interest to know that two versions differ but also what differs between them.

When a conflict is found, a merge should be conducted. A merge is a quite complex operation. It's known from software development that it is possible to merge text files with a pretty good result. However, merging CRM data is different. The data types and the data structure is much more defined and won't change between different versions and there's no compiler to verify the result meaning that the importance of a correct merge is higher.

A key to being able to do conflict detection and merges is to have a flexible unit of comparison. The algorithm should always use the smallest possible unit of comparison, but not any smaller. Since the smallest possible unit of comparison differs between different data fields, them being dependent on other data fields or not, the unit of comparison must be flexible.

This chapter will start with a discussion about the different merge approaches needed to solve the use cases from chapter 3.1 and then discuss the custom implementation that is done as a proof of concept to show that the techniques actually work.

4.1 Merge approaches

In this section, the different merge approaches that are implemented by the custom merge tool will be analyzed. Not all approaches are used to solve the use cases analyzed in the paper and not all merge approaches previously discussed are implemented.

For example the voting method isn't implemented since it was deemed to be a bad fit already in the analysis of the use cases.

The Levenstein merge is implemented but not used since the false positives are too many. This was seen already in the analysis part, but a test implementation was done either way. The results were correct in the test run but the test data too small for that to have any importance.

First the case of a simple merge will be discussed, followed by how to handle a merge where the leafs to be merged have a common ancestor. Then merging with the translation method described in 3.3.3 will be discussed. The case with doing a merge with more than two parents, that is merge three of more leafs together will then be discussed, followed by the same case, but including translations. Finally a discussion about partial history merge will be held.

4.1.1 Simple merge

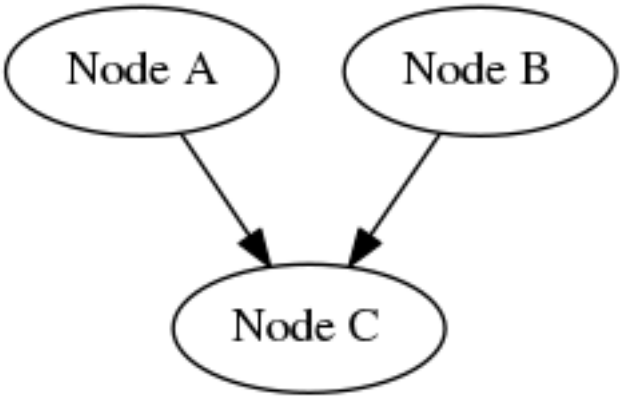

Figure 25: Simple merge

A simple merge is the merge of two nodes, see figure 25. In the figure node **a** is merged with node **b** to form node **c**.

One way to solve this is to look at the timestamps of the nodes, and just pick the latest node. This is what Lime Easy and Fat Free CRM does. A better approach would be to flag it as a conflict, that's what Deko the CRM does. Best would be to try to merge the two nodes with an as fine unit of comparison as possible and only flag it as a merge conflict if needed.

To pick just the latest node isn't the same as picking the most correct node and data might be lost. Always requires human merge resolution is cumbersome and work intense. Hence an automatic merge should be tried, since that's the best solution if it works.

In detail the custom algorithm work as described below.

The simple merge will take two nodes. It will first create a third node to hold the result of the merge. It is this node that will be populated with data and returned as the result node.

The meta data of the newly created third node will be populated with the author, which would be the merge tool itself and the current time. The parents of the node will be set to be the first and second node, the nodes supplied to the simple merge algorithm. Each node has an id, the supplied two nodes will have id 1 and 2 respectively and the third resulting node will get an id that is one bigger than the biggest id supplied, in this case 3. Each field of the supplied nodes will be traversed and if they match, the value of that field will be put as the value at the corresponding field on the third node that will hold the result. If the values differ, a merge error marker will be inserted instead of the data from any of the supplied nodes. The result node will then be returned as a result.

4.1.2 Merge with common ancestor

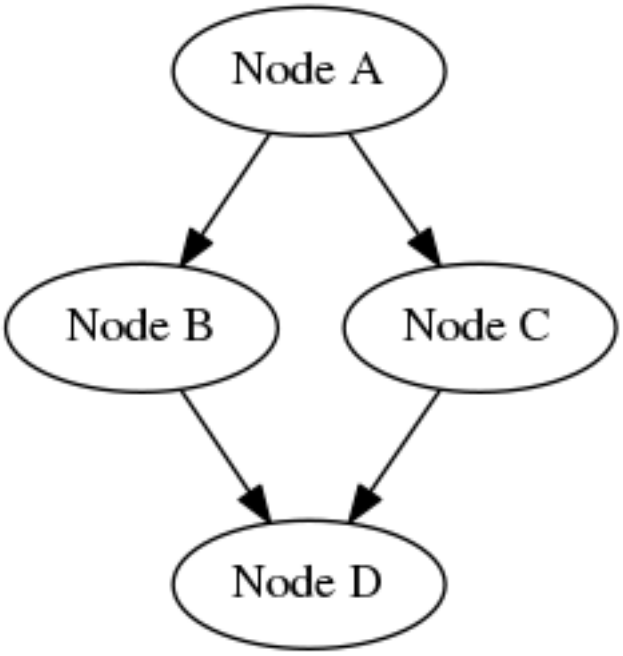

Figure 26: Merge with common ancestor

In section 4.1.1 there's no way to know if the changes done to node **a** and the changes done to **b** are conflicting with each other or not.

One way to solve this is to use an algorithm called the three way merge [12].

The three-way merge algorithm looks at what changes between node **a** and **b** and compare that with what changed between node **a** and **c**. This

way the changes, not the content, will conflict.

The same problem as in section 4.1.1 can be solved with the additional information about the common ancestor of the two nodes that are going to be merged.

In figure 26 the merge case can be seen, where two nodes have the same ancestor and then are merged together to form a fourth node combining the information from both node **b** and **c**. What this actually does is decreasing the unit of comparision from being a node to be a change between two nodes. This might have implications, since one change may alter one field that depends on another field. That's why a flexible unit of comparision is implemented. This will detect if a change alters a value that has a dependency and in that case it will mark both values as changed. More about how this is implemented can be found in section 4.2.

In detail the custom algorithm work as described below.

The merge with parent will take three nodes, node **a**, node **b** and **c** and a list of dependencies between different data fields. The first node will be the common parent of node **b** and **c**.

A new node, node **d** will be created as the result. The meta data of the newly created fourth node will be populated with the author, which would be the merge tool itself and the current time. The parents of the node will be set to be the second and third node. Each node has an id, the supplied three nodes will have id 1, 2 and 3 respectively and the fourth resulting node will get an id that is one bigger than the biggest id supplied, in this case 4. Then each field of node **b** and **c** will be traversed and if they match, the value of that field will be put as the value at the corresponding field on the fourth node that will hold the result.

If the values differ, a check will be conducted to see if a value is changed between the parent and the node. If that's the case, that data field is dependent on any other data field. If it is a dependency and the field has changed between the parent or any of the children a merge error marker will be inserted instead of the data from any of the supplied nodes. The result node will then be returned as a result.

4.1.3 Merge with translations

In previous section, 4.1.2, a conflict is solved by additional information, that is the common ancestor node. More information about the history might help to create an even better merge.

This isn't obvious. Previous versions could contain errors, and this method would make it possible to translate a value to an erroneous value, for example a misspelled word. However, assume that that error also have been corrected, then that information on how to correct such error could be useful in a merge.

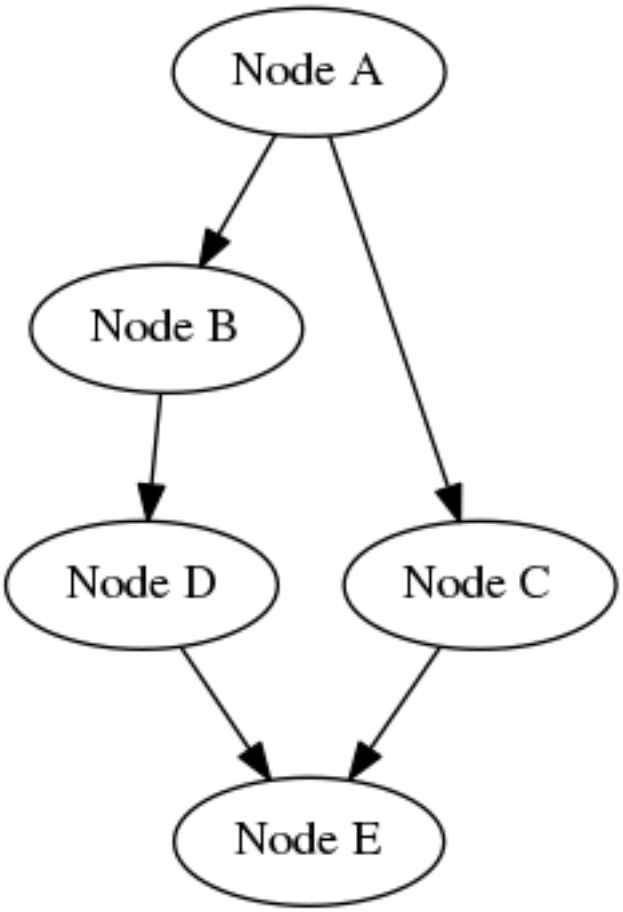

Figure 27: Merge with translations

In figure 27. The approach is to use information from the changes between node **a** and node **b**, and node **b** and node **d** instead of just the change between node **a** and **d** as is done in the merge with common ancestor in the previous section.

In detail the custom algorithm work as described below.

The merge with translations will take an object containing all nodes, one of the nodes being merged called node **d** and the other node being merged called node **c**. A new node, node **e** will be created as the result. The meta data of the newly created node **e** will be populated with the author, which would be the merge tool itself and the current time. The parents of the node will be set to node **d** and node **c**. Each node has an id, the resulting node will get an id that is one bigger than the biggest id supplied. Each field of node **d** and **d** will be traversed and if they match, the value of that field will be put as the values at the corresponding field of node **e**, the node that will hold the result.

If a field doesn't match, an effort to find translations for that field is done. A translation is a change to a field, a rule that says that one value of a particular field can be translated to another value of that field. This was analyzed in section 3.3.1.

Finding translations will traverse the history and record all changes to that particular field. This is called translations and all these translations will be collected. Next translations will be applied to first **d** to find a match with **c** and then on **c** to find a match on **d.** The application of translations is a recursive function that will try all collected translations on field in order to find a field that will be equal to the target field. This will be done for all translations, knowing that multiple translations could be applied to after each other.

If no translations succeed to result in a merge, a merge error marker will be inserted in node **e** instead. The result node, node **e**, will then be returned as a result.

4.1.4 Merge with more than two parents

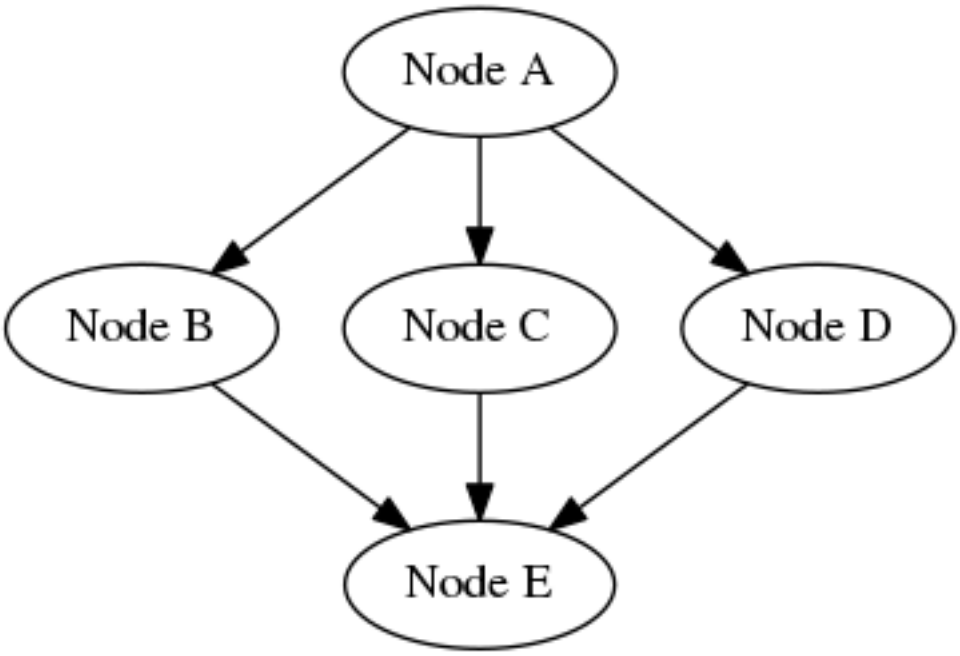

Figure 28: Merge with more than two parents

In figure 28 a merge of three nodes is shown. This case could be treated the same way at a two parent merge, done multiple times. However, then the information that this is a multiple parent merge is lost. It's seen from section 4.1.3 that extra information can be important. For using the extra

information, the merge algorithm called voting, see section 3.3.3 can be used. However as discussed in use case 12, section 3.1.13, this isn't a method that is safe to use in the CRM setting and hence there's no need for the extra information needed.

The case with more than two parents could therefore be handled as a generalization of the two parent case.

In detail the custom algorithm work as described below.

The merge with multiple parents will take a list of the nodes that are to be merged. For example in use case 10, section 3.1.11 a list of three different nodes.

A new node, node **d** will be created as the result. The meta data of the newly created node **a** will be populated with the author, which would be the merge tool itself and the current time. The parents of the node will be the second and third node. Each node has an id, the resulting node will get an id that is one bigger than the biggest id supplied.

For each field in the nodes, the value will be compared with all other nodes in the list of nodes supplied to the merge with multiple parents function. If they match, the value will be applied to the corresponding field in node **d.** If not a merge error marker will be inserted in node **d** instead.

The result node, node **d**, will then be returned as a result.

4.1.5 Merge with more than two parents and translations

Let's consider the most complex case where history and more than two parents are combined in one merge case. A simple example can be seen in figure 29. This must be handled by a merge tool.

A combination of the merge with more than two parent in section 4.1.4 and the translation merge in section 4.1.3 are combined in order to solve this merge.

A case to consider here is when node **b**, node **c** and node **d** are equal. In that case node **f** will be equal to node **e**. This might from a first look seem a bit strange, but that's the result of ruling out voting, section 3.3.3, as a method.

The partial history merge could be seen as a purely optimization tool since the merge result would be the same with or without it.

In detail the custom algorithm work as described below.

The merge with multiple parents and translations function will take an object with all the nodes as input data as well as a list of the leave nodes, the (soon to be) parents to merge. A new node, node **f** will be created as the result. The meta data of the newly created node **a** will be populated with the author, which would be the merge tool itself and the current time. The parents of the node will be set to be the second and third node. Each node has an id, the resulting node will get an id that is one bigger than the biggest id supplied.

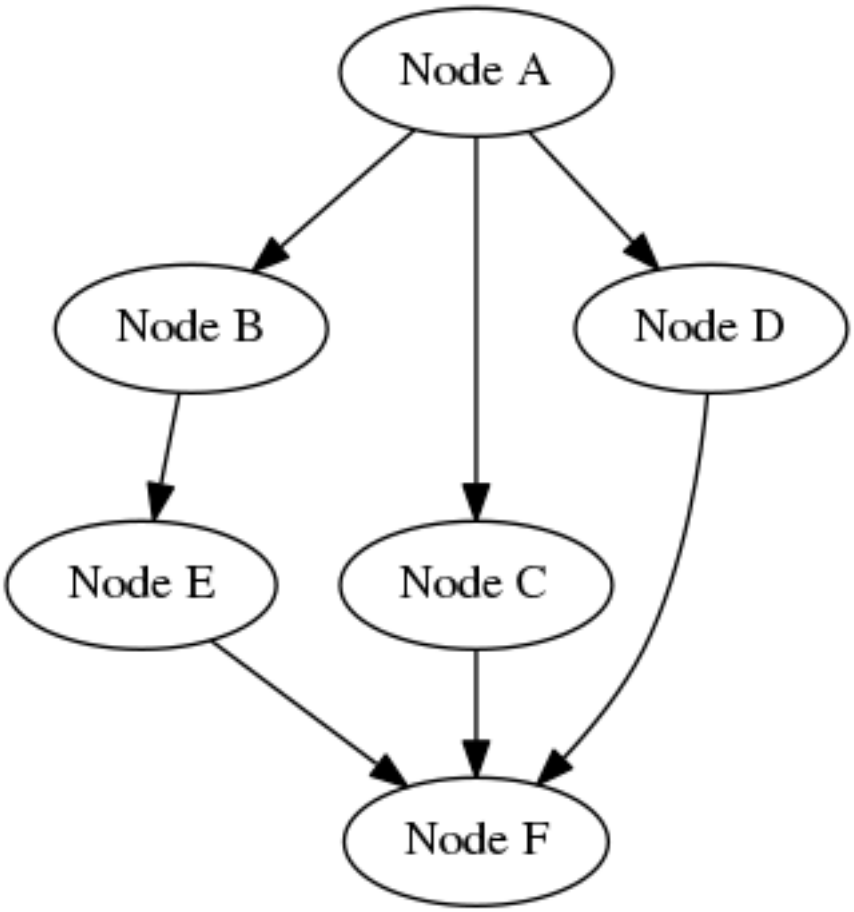

Figure 29: Merge with more than two parents and translations

For each field in the nodes, the value will be compared with all other nodes in the list of nodes supplied to the merge with multiple parent function. If they match, the value will be applied to the corresponding field in node **a**. If they don't match, the nodes will be traversed to find possible translations that can be used.

The translations will be appended to each of the leave nodes and try to get two leaves to match. If that's possible the result will be tried to even get the third node to match. This is implemented as a recursive function that could possibly be pretty expensive to run. Here the algorithm could be optimized by make use of dynamic programming and score boarding techniques to reduce the cost of running this function.

If still no merge could be done a merge error marker will be inserted in node **a** instead. The result node, node **a**, will then be returned as a result.

4.1.6 Partial history merge

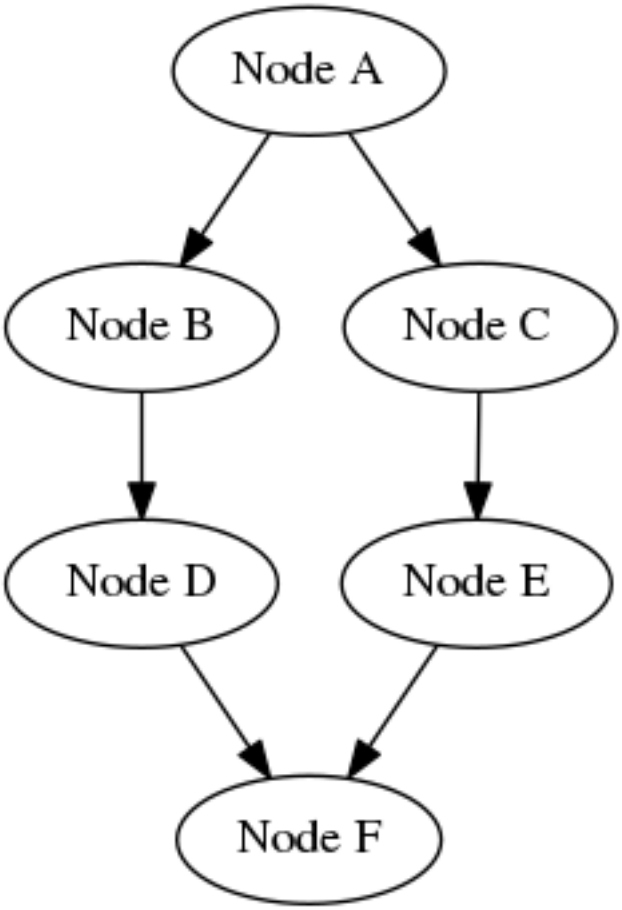

Figure 30: Partial history merge

Consider figure 30. If node **b** and node **c** are equal, they could be merged with the simple merge, section 4.1.1, to form a new history, hence a new merge case, as shown in figure 31. This is a method of simplifying the history to ease the use of other merge algorithms.

The partial history merge doesn't solve any problem that the other merge algorithms haven't already solved, but can greatly improve the performance of the other merge algorithms since partial history merge is much cheaper to do than for example merge with more than two parents and translations from section 4.1.5. The example from section 4.1.5 would never happened if partial history merge was done first.

The need for partial history merge isn't obvious but reducing the merge problem as much as possible is efficient for dealing with longer histories.

In detail the custom algorithm work as described below.

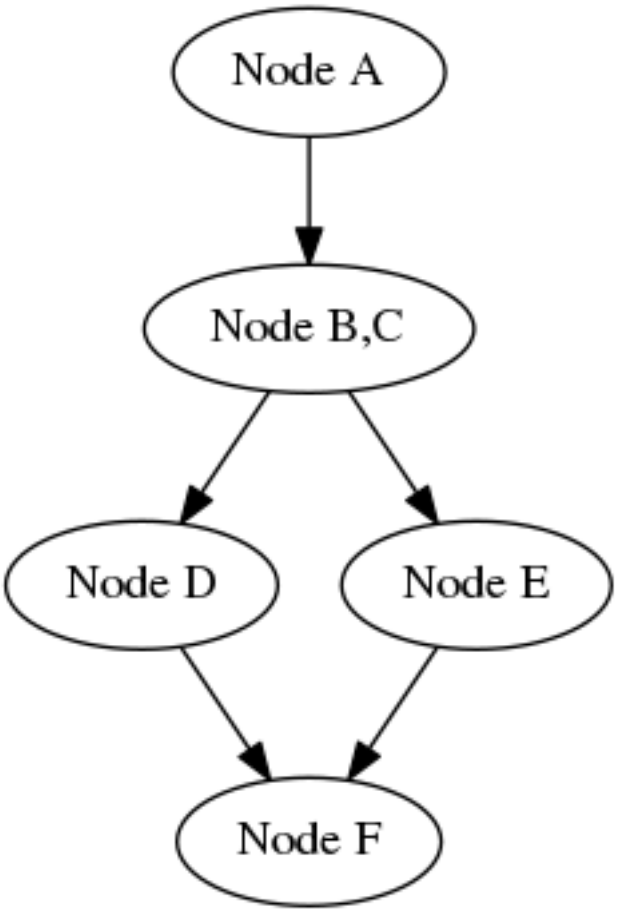

Figure 31: Result after applying partial history merge

The partial history merge will take an object containing all nodes. It will then create three lists, new, drop and adopt. The new list will contain the new collection of nodes that another merge algorithm will consider its input data. The drop list will make sure that a node won't be processed more than once and the adopt list will contain information about which nodes that can be said to adopt any other nodes in order to reduce the history.

For each node, all children will be found. If any of the children are present in the drop list, they are already processed and nothing will be done. The adopt list will be traversed to see if the item's parent could be changed with the information gathered in the adopt list or not. Then if more than two children exists they will be compared. If they contain the same values for each field, they will be merged using the simple merge algorithm described above. Note that they are guaranteed not to have any merge conflicts since they are compared first.

The new node is added the new list and data is added to the adopt

list with information about each node that can be translated to the newly created node. Also each merged node is added to the drop list so that is won't be examined again. If they do differ in the comparison, they will be added to the new list.

This function will be called recursively with the new list as its argument as many times as it takes not to have any newly create merged items added to the new list. Then the new list will be returned.

4.2 Custom implementation

In this section, the custom implementation will be analyzed together with its abilities and limitations.

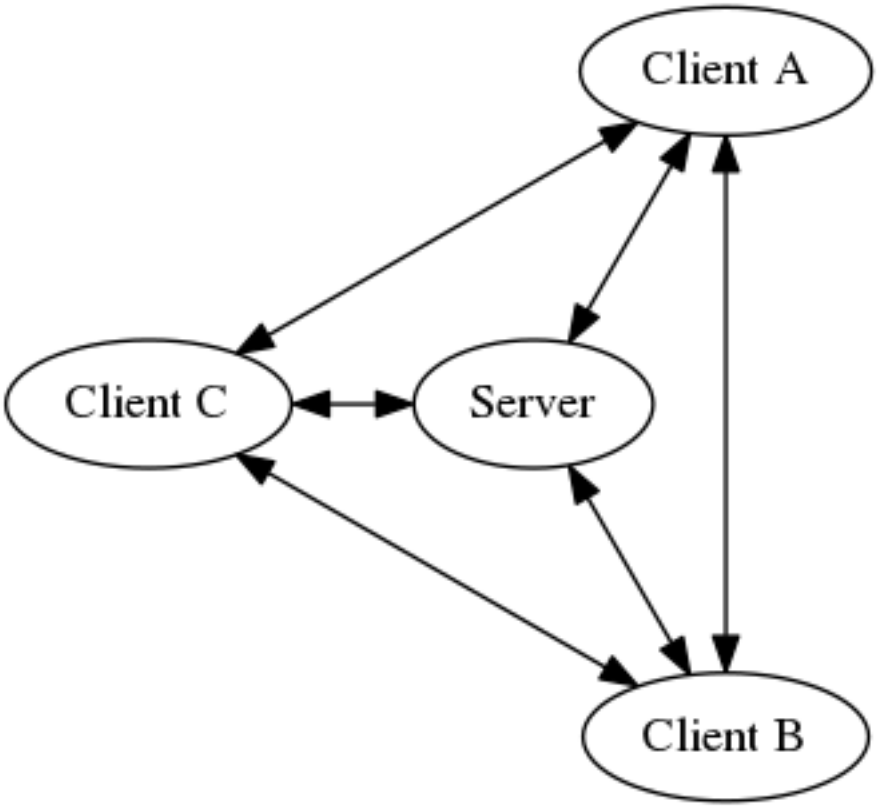

Figure 32: Distributed and centralized workflow

To prove that the techniques from chapter 4, can be used to solve the use cases from chapter 3.1, an implementation is done as a proof of concept. The implementation is pure conflict detection and merge program and does not include any other CRM functionality nor a user interface. With these techniques a distributed workflow as well as a centralized workflow is possible. In figure 32 three salesmen can be seen working centralized towards a server at

the same time as they share information with each other directly. Working towards a server could be great if the salesmen rarely see each other. It can also be great to be able to share information between each other if meeting without being able to communicate with the server. Both cases should work when the salesman is offline from time to time. A good conflict detection and merge tool makes this possible.

First a discussion about the data format used and how the result validation is implemented, then a discussion about merge algorithm selection will follow. Last the limitations of the implementation will be discussed.

4.2.1 Data format and result validation

To be able to specify the use cases in a clear way and to be able to verify that the result of the custom merge algorithm was successful, a special data format for describing use cases was constructed.

Each use case is described in a text file with each node, including the correct result that can be seen in the figures in chapter, 3.1.

For processing a use case, the program need to read the file and separately store the final node. The merge algorithm is then run with all the other nodes as input data. A result in the form of a node will be returned from the merge algorithm.

The returned node is then compared to the final node stored earlier. If they are equal, the merge algorithm is said to successfully solve the use case.

A success could be in the form of a merge error, showing that the merge algorithm is capable of conflict detection in those cases where a merge can't or shouldn't be done.

There's certain meta data that should be excluded from the comparison between the merge node and the final node, for example the date meta data that is changed each day the merge algorithm is run.

Since the meta data isn't important as a result (and isn't important for solving any use case either) the complete meta data is ignored in this comparison, even if it could be included or not.

The meta data contains information about when the change was done and who did the change, as well as which parents a change has. Since a merge can be run at any time and still should produce the same result for the exact same input data, the timestamp of the result shouldn't be compared in the validation. The same is true for a few other fields as well, but this should be pretty self-explanatory.

4.2.2 Merge algorithm selection

Previously in this chapter six different merge approaches where listed:

- Simple merge

- Merge with common ancestor
- Merge with translations
- Merge with multiple parents
- Merge with multiple parents and translations
- Partial history merge

An important part of a merge algorithm is to know which one to use. This is done manually by the custom implementation. That is, since the use cases are predefined, which merge strategy that should be used to which use case is predefined at the implementation time. The reason for this is that merge algorithm selection in its own is an interesting problem and deemed too big to fit in this paper. This wouldn't work in a real world scenario where it's also important to determine which approach is the best to take.

An easy way of solving this would be to run each merge algorithm until a merge error isn't the result or until all merge algorithms are tested. This requires that a merge algorithm never will give a faulty answer.

For the above merge algorithms that's probably true. A merge algorithm that fails in one case, won't succeed in another case, that another merge algorithm should solve, but fail there too. To prove this is outside the scope of this paper and is left as a possibility for future work.

Running a merge algorithm could possibly be pretty expensive compute wise. One way to mitigate this is to run the merge algorithm with the most likelihood to succeed or to reduce the problem first and then order the rest of the merge algorithm in the same way.

The first merge approach should probably be partial history merge. Although it not a very likely case, it has the ability to reduce the problem for the other merge algorithms making them cheaper. Next algorithm to choose should be the simplest that will work for the most cases and not be that expensive.

An even better way would be to somewhat analyze the input data and then predict the best algorithm to use. This could be done for example with machine learning. This is however not in the scope of this paper.

4.2.3 Limitations

Apart from the lack of merge algorithm selection, there are a couple of other known limitations of the implementation.

The data format is hard defined and changing the number of input fields would make the program stop working correctly. This would be simple to fix but it's not needed for the custom implementation. The implementation is done to prove a concept for the above use cases and not for being used in a real world settings. With just small adjustments it would be possible to

feed the implementation with different sized nodes. This is important since it would be possible for the customer to choose which data to track of each customer and the merge algorithms would still work as long as the customer is also specifying the possible dependencies that exist between different items.

All multi-parent cases are limited to three parents. This is enough for evaluating how to handle multiple parent situations good enough for this paper, and the same algorithms and techniques that is used for a three parent merge could most likely be used for a n parent merge, where n is an integer bigger that 3. This is however not supported in the custom implementation that is limited to max three parents. This is because each algorithm is done from a two conflicting nodes case and then needs to be generalized. A generalization would be done already for three nodes, so the conceptual difference between a merge algorithm that can handle three nodes and n nodes would be nonexistent. However, there might still be differences between n nodes and three nodes in the implementation.

Data formats and sub-algorithms used are not optimized at all and will possibly scale badly for a real world scenario. For example, in some cases lists are used where a hash map should be more efficient. The data formats would be deemed too heavy for real world usage. For example the JSON data format is using a lot of overhead since each item contains a full description. Since the data format is static and won't change, this is not needed for this application.

Example of sub-algorithms used could be finding an item in a collection, or iterate over all nodes in the history. This is not optimized and it's possible that other data collections or iteration patterns would be more efficient.

Error handling is very rudimentary, each use case only presented with a "Success" or "Failure" label in the output, but never what is failing. Since the use cases processed is all known and know to succeed, a more elaborate error handling isn't implemented. For trying this program on other use cases than the above, a better error handling with more information to the user should be implemented. For example, information about which items failed.

However, for the user of a CRM program it is often not important to know why a merge approach fails, just that the failure is and what fails so that he or she could do a manual merge. A next step for aiding the user would be to leave a merge proposal where the merge algorithm presents the most likely case and/or simplify the merge problem as much as possible before presenting it to the user, for example with partial history merge and translations.

If both conflicting leaves will need to have translations done to them in order to find a match, then this algorithm will fail since it's not supported to translate both nodes. The reasoning behind this is that if it's allowed, a merge could result in a value that isn't present in any of the leaves that are being merged. This should never be allowed to happen from a data integrity state of view. So even if this could be seen as a limitation, it's actually a

safeguard against evil merges [14].

[14] An evil merge is defined in this paper as a merge resulting in a value that isn't present in any of the versions that are being merged.

5 Discussion and related work

In the previous chapters the current support for distributed work has been studied in some different CRM systems on the market today, then requirements have been collected for what is needed of a good merge tool for CRM data. Different tools and techniques for solving these requirements have been studied and then implemented.

This chapter will start with a discussion on the current CRM implementations that where analyzed earlier in this paper. A discussion about the use cases presented and their relevance will follow. Then there will be a discussion about the custom merge algorithm implementation and the design choices made when writing it. This will be followed by a part about related work and last a few suggestions for future work.

5.1 Current CRM implementations

Current CRM software shows poor or nonexistent conflict detection as well as merge proposals. Different tools are using different sizes of unit of comparison and conflict detection was only present in one of three systems studied. The case where conflict detection was present had however a lacking merge support as well as good tools for showing the difference between two versions, meaning that conflicts was hard to solve. The two other tools that didn't had conflict detection had very different unit of comparison. This lead to very different behavior in the use cases presented earlier in this paper.

During talks with different representatives for CRM vendors, it's clear that much often the problem of diverging data is poorly understood by the sales representatives. This would probably be a hint about that the problem is even less understood by the customers, the implementers of CRM software. If there's no demand for handling data collaboration problems, there won't be any solutions given either. This is probably because a lack of awareness of the problem and the different solutions possible and the lack of serious consequences if something actually goes wrong.

An exception to this is Deko the CRM who had listen to some of its customers that actually understood these problems and made a solution [2]. However, the solution is clearly lacking compared to the tools for software development that are available today, even if it is market lead in the CRM industry.

The lack of support for distributed workflow in CRM systems isn't necessarily a bad thing. Experience shows that even a good tool can be bad if not understood correctly. Very rudimentary configuration management systems is performing better than more advanced once if the users do understand them. That is, if the users don't understand a distributed workflow and the need for conflict detection and merging, it might be better to just loose data since that can be understood easier.

It's not a clear if educating a sales force enough to use distributed systems is economical beneficent or not. That's also not part of the scope for this paper.

There should however be a market place for a system that not only supports conflict detection (which is hard to get user friendly) but also merge resolution. If the merge resolution is good enough it will be easy for users to handle the system. A CRM system does not need to show complex configuration management tools such as branching to the users but only conflicts in the few cases when a merge can't be done manually.

A good merge algorithm is the foundation towards the next generation of CRM systems.

5.2 Use cases

Most of the use cases are relevant for everyday use, some are only edge cases. The goal was to choose use cases where to study what was common scenarios when working with distributed and diverging datasets and also cover some edge cases to get a feel for the depth of edge cases that might occur. If the number of edge cases would have turned out to be numerous and they also had been hard to solve, then the cost of solving the distributed workflow model would have been clearly much more expensive than if the edge cases was few and easy to solve.

As it turned out, most edge cases are still very easy to detect and it's often enough to just detect edge cases since they are so rare (by definition since they are edge cases) that a manual merge procedure is acceptable.

An important lesson from working with the use cases is that even a simple use case can be poorly handled by a naive implementation. Therefore even the very simple looking use cases are often the most important ones.

The most important use cases are the one that results in a merge conflict. Conflict detection is extremely important and is often the first thing that shows if a CRM system is lacking or not. If a system cannot detect a merge conflict, it doesn't matter how good it can merge data. Detecting the need for a merge is fundamental before a merge should be done.

Other important merge conflicts are the ones that handle different unit of comparisons. Both where a merge conflict should be detected and when a merge should be made without any conflict. The technique with a flexible unit of comparison couldn't be found in any of the systems examined and should therefore be a unique selling point for any system that implements it.

Duplicate detection is maybe not that critical for an everyday workflow, since missing to find a duplicate won't result in any immediate problems. However, they are as important as any other use case in the long run and is often overlooked. Unfortunately, it's a rather hard case to solve that almost always will need human verification. This is because an entry seldom has a unique identifier. Names and even phone numbers and

e-mail addresses are not guaranteed to be unique to a single person. For example, the e-mail address "info@lth.se" is probably not personal while "fredrik.gustafsson@svep.se" are. This is hard to verify for a computer but a human can probably do a better job in weighting all available information and resolve the situation.

Duplicate detection is needed both in a centralized system but is more important in the case of a distributed/off-line system. In a centralized system it is always possible to get all the current data in the system and to be able to detect duplicates in some way (even if that way is very cumbersome sometimes). However, in an off-line system where each node doesn't have access to all data at any given moment it's impossible to do duplicate detection at creation time of a record and it must therefore be considered to be a merge problem and not just a fringe benefit.

The case with miss spelled names or other fields are interesting but not that important for a merge tool. Since the accuracy isn't high enough there's no use for any special techniques in this case but a simple merge error should be the result. This is however an area where increase in the accuracy for misspelled words could change the importance of these use cases.

Use cases with more than two parents is interesting and opens up for more possibilities for solving a merge case, for example by voting. However, the extra information that the algorithm get access to by treating a merge case with all parents at once instead of solving the cases as multiple two parent merges seem to be superfluous. Therefore, it's of no use to handle merges with more than one parent as a special case, but instead merge the leafs two and two. This would not only result in the same merge result, but also probably be easier to implement since code can be reused from the case with a two parent merge. See use case 11, section 3.1.12, for a more detailed discussion about this.

5.3 Design and implementation

The custom merge algorithm was satisfying. It succeeded in solving all use cases as intended. A few general techniques could be used that solved all use cases. The results can be seen in table 4.

Most surprisingly was the implementation of flexible unit of comparison that was very easy to implement. It's implemented in the way that the merge function takes a dictionary, for example:

```
    city -> [zip code]
zip code -> [city]
```

This list explains that *city* depends on *zip code* and that *zip code* depends on *city*. This system is flexible enough to even specify single direction de-

Table 4: Results from testing the custom merge algorithm implementation

Use case	Verdict
1.	Success
2.	Success
3.	Success
4.	Success
5.	Success
6.	Success
7.	Success
8.	Success
9.	Success
10.	Success
11.	Success
12.	Success
13.	Success
14.	Success
15.	Success
16.	Success
17.	Success

pendencies, something that wasn't explored at all in this paper. It's possible that it would be a feature wanted if the data fields where chosen different.

The algorithm is a proof of concept algorithm that has been run on limited datasets which means that the efficiency of the calculations done hasn't been important. No optimization has been done and in a few places the data structures and (sub)algorithms used are not optimal. Since the datasets where quite limited in size, this was not a problem and the algorithms was fast to run. On the authors computer, solving all 17 use cases took about 60 ms. The computer being a 2015 high end desktop computer, with 8 virtual cpu cores and 16 gb in RAM. If this scale is not of interest for this paper and the test implementation would probably not scale very well due to inefficient use data structures.

In a real life scenario, not only is the time to do the merge of importance but the complete time to sync one client with another. That involves to find out what to merge. This could possibly be expensive. See future work for a discussion about this.

Even if the results are satisfying for as a proof of concept, there's still work to be done for using this in production. There would need to be a merge selection algorithm and some parts of the code might scale badly. The parts that probably would scale badly is excessive use of lists instead of arrays or hash maps where appropriate. Therefore, longer histories before the common ancestor could become a problem.

To conclude, the implementation was a success and it proves that it is possible to successfully solve the problem with conflict detection and merge resolution presented in chapter 3.1.

Translations

The translation method is the most questionable method. The idea behind is the assumption that one of the leaf nodes has the correct data for a field and that all changes handle a progress and progress is correct.

The assumption that one leaf node has correct data is not very hard to do. It's logical to believe that if two versions are both said to represent a common truth and they differ, one of them is wrong. It could of course also be so that both are wrong, but in that case it doesn't matter which one that are chosen. However, it's not possible that the truth is a mixture between the two versions, since a mixture should require each version to know about the other version when reading the truth in order to be able to divided the truth between two nodes.

The second assumption is that progress is correct. This one is not completely true. For example, a mistake could be made that makes a field change from being correct to be incorrect. If the user notice this and change back to a correct value, no harm is done. The translations would result in a no-op and won't impact the result. If the user doesn't notice the error made the translation technique could be dangerous since such an error could spread between different clients, since it would be deemed to be a progress of the data.

Merge algorithm selection

Since different use cases needs different techniques to be used to solve them, different merge algorithms need to be used. A first task for solving a use case is to determine which merge algorithm to choose. Some thoughts about how to implement this is presented in 5.5.

The custom algorithm did not implement merge algorithm selection since it was deemed out of scope for this paper. However, with manual algorithm selection it successfully solves all merge cases. That is, all merge cases get the desired results stated in the use case descriptions in chapter 3.1.

One way of doing merge algorithm selection is to make sure that each algorithm always will fail if it's used on an unsuitable merge case and no previously run merge algorithm has succeeded. In that case it would be possible to order the merge algorithms and run them one by one until a merge algorithm succeeds. It would be advisable to run the less expensive merge algorithms just after the merge problem reducing algorithms, such as partial history merge, that should be run first.

Another way would be to study the merge problem and then guess a

merge algorithm. This would possibly be more efficient but needs more study to know how to match a merge problem with a merge algorithm correctly (and efficient).

5.4 Related work

There is a few related papers on this subject.

The most closely related text about this problem is the manual for Deko the CRM, [2] that isn't a scientific paper but presents it's solution to the problem with distributed CRM systems. The underlying techniques for Deko is described in the manual for Couch DB, [3].

Deko the CRM has successfully identified the problem that a user might work off-line and that there's a need for synchronizing and detecting conflicts. With that, this paper agrees. The solution chosen in Deko the CRM is however not optimal. The document database Couch DB is used as a data store and also its merge and conflict detection algorithms. The conflict detection in Couch DB is not specialized for CRM data but are general algorithms that has no knowledge about the data it handles. This makes it to be a blunt tool that need much support from the program using it, support that Deko the CRM doesn't deliver. Although being the best found CRM software on the market today, in the area of distributed work, it could still improve.

A more general text about working together with data and the fundamental problems that arise from sharing data is described by Babich in his book Software configuration management : coordination for team productivity, [4]. Babich identifies three fundamental problems for data sharing between different users. His double maintenance problem is the most obvious for the CRM cases where two copies of the same data should be maintained and be identical.

A similar problem is the problem of merging models instead of source code. There has been quite a lot of work in that area, for example by Bendix, Emanuelsson [8], where different requirements for model based merges are examined. A model based merge has similarities with CRM data merge that it's not row based information that is to be merge but data that has another structure. Bendix and Emanuelsson are however working with the assumption that the models should be represented as text and then merge to be able to use a conventional version control system. This is a limitation that this paper hasn't. The difference here being that a CRM system is its own data store and has full control over it, while the models in Bendix and Emanuelsson's paper is generated by a program and then needs to be stored in another program together with other data.

An alternative approach is to use operation based merge instead of data based merge. This is studied by Koegel et. al. in [10]. Here it's not the data that is stored as version information but all operations stored on that data.

This paper suggests not merging the data but the changes done to that data. A change and an operation isn't quite the same thing, but related at least in concept. Koegel has a problem with versioning a model and not about merging data. To solve this Koegel merge operations instead. The CRM data is just data and there's no need to record operations, operations are quite simple in the use cases presented above. If however a more full data model of a CRM system would be used, that has multiple relations between different objects, let's say instead of the CRM system being a single database table as in this paper, it would be a relation database with multiple tables, then the techniques learned by Koegel would be highly relevant because the problems would be very similar.

CRM data is however different from data that should describe a model, for example XMI, see [9] for a good explanation of XMI. It is therefore more complex than needed for CRM data, which has a rigid data structure with known data.

Lide, [9], studied XMI merging as well and built his work on a paper by Martini, [11].

The main difference between XMI data and the CRM data that are presented in this paper is that XMI data describes which types of data can occur, but not that they occur or which relation they have between each other. All data fields in the CRM data case are always present, although they may be empty, and the relations between different fields are known beforehand. This means that the unit of comparison is simple to calculate when the system is designed. It can be flexible (as described in this paper) but it doesn't have to be dynamic. For the case with XMI represented data this is not true. The data structure can be changed and those changed should be able to merge, leading to a much more complicate problem.

For deeper study in name matching techniques, a good start would be Christen [5]. This paper presents more effective algorithms for matching names than the more general Levenshtein approach. For example, has really good results been delivered in matching patient data from hospital records. This is probably not that far from CRM data, and many of the techniques here could be useful for displaying a merge proposal to the user. Although since the techniques didn't give a 100% secure result they could result in incorrect merges if the merges should be performed automatically and not just be presented as a proposal to the user.

The name matching techniques used are highly coupled to the English language and would probably need to be adjusted for being used in an other context. The sound matching techniques presented in [5] are using the first character of a name followed by a representation of the sound of that name. This is an interesting approach to identify misspelled names that would often sound the same but be spelled differently. The problem here being that the person writing the name is not the owner of the name and has only a vocal record of the name.

However, many names differs in the first character as well. For example "Filip" and "Philip" would be hard to be identified in these algorithms. For using these algorithms, some additional data editing beforehand would probably give a better result.

5.5 Future work

In this section, different parts that could be interesting for future work will be presented.

If the techniques described in this paper are implemented in a real world application it would be interesting to see real world use statistics that shows which algorithms is used and how much. Maybe some of the algorithms here is so rarely used that they actually isn't worth having? Maybe there's a use case not covered here that is frequent enough to demand the invention of a new merge algorithm?

5.5.1 Merge algorithm selection

There is multiple parts that would be interesting for future work. The most obvious one is merge algorithm selection. An easy way would be to design all algorithms so that they will result in a merge error if they don't produce correct result, that is, they won't be dependent on in which order to be run. This would make it possible to start to run one algorithm and if that fail continue to try the next one and so on until one algorithm solves the merge case or until all algorithms fails. However, this isn't very efficient. A case for future work would be a predictor that could predict which algorithm to test based on the input data.

5.5.2 Uses for editing distance techniques

The Levenshtein merge that earlier was deemed too insecure for use on CRM data is still interesting to look at. Would it be possible to combine different word comparison algorithms to be able to get a good enough result? Is there any kind of data that could use this since a good validation tool is available?

5.5.3 Optimal information needs

In section 3.2.3 the amount of information available to the merge algorithm is explored. More information would probably give a better chance to design a good merge algorithm. Which information is needed, which information isn't needed and what additional information would have a high impact on the quality of the merge algorithms results would be an other area for future work.

In section 3.3.4 in partial matching records are discussed. Finding partial matching records is a possibly expensive to do, in terms of cpu cycles. Here

it would be possible to do some optimizations to improve the speed on how to find partial matching records.

5.5.4 Finding what to merge

The real world use case would be that two identical databases are separately updated until they diverge with different data. Then the two databases are to be merge into being identical again.

Except the merge problem itself, an important problem to solve is to know what to merge. This could be done by saving all changes done to the database in chronological order and then just look for merge candidates for the data that has been updated in any of the databases from the time since they where merged last time.

If **a** and **b** are both changed since last merge, the next step is to figure out if they are related at all and should be merged or if they are two completely unrelated items.

One way to do this is to check if they have a common parent and if they do try to merge them. Another way of doing that is to let each tree have an unique identifier and aim for that each tree only should have one leaf after a merge.

6 Conclusions

Today's CRM software has no standard practice for solving the problems with a distributed workflow, the approach differs a lot between different products. Trying to merge data to ease distributed work is nonexistent. The best programs at least warns when a possible data loss is possible due to update conflicts. While most programs won't even tell that data is overwritten.

Since the data is structured and the relationship between different data fields can be known in advance, the possibilities of doing a merge of the data is good. It's even better if the history is preserved.

This paper examines three different CRM systems and their solution to the distributed workflow problem. The paper also presents 17 use cases that can be used to analys a CRM system as well as work as a requirement specification for an implementation of a CRM system that supports distributed workflows. Last this paper explore techniques that can be used to solve all 17 use cases and implements a software that successfully solves all 17 cases as a proof of concept.

The three CRM systems that was analysed in this paper has to solve the simultaneously update problem in some way. The most common is to make the diverging time as small as possible and then use the latest version, even if another version would be overwritten. Since people usually not work on the same record, this approach seems to work fairly well as long as the diverging time is small. When starting to work offline the time two versions can diverge increases and conflicts will happen more often.

The conclusion is that this isn't an area given too much attention by today's CRM software, or that the knowledge of merge algorithms is low by designers of CRM software.

This paper delivers 17 use cases that could be used to assess the conflict detection and merge capabilities of a CRM software. It also delivers multiple techniques that can be used to successfully solve all 17 use cases.

It is possible to manage all studied examples with the techniques discussed in this paper. The most important result is however to not throw away data due to a merge conflict without the user noticing this. Achieving this is simple, but it's hard for the user to manually merge data, especially with many data fields.

A custom merge algorithm was implemented as a proof of concept with the techniques presented in this paper. It successfully detects when a conflict can be merged successfully and when to ask for assistance from the user.

It is possible to detect merge conflicts and do merging in the cases that are possible to solve. Current software has limited support for this, but the potential for improvement in this area is great. It's not just merging that is the challenge, but also conflict detection which is a big part of the challenges for distributed work.

References

[1] Multiple authors. Cap theorem - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/CAP_theorem. (Visited on 06/08/2015).

[2] Multiple authors. Hobrasoft: Deko the crm – distributed crm system. http://www.hobrasoft.cz/en/deko/description. (Visited on 07/03/2015).

[3] Multiple authors. Overview — apache couchdb 1.6 documentation. http://docs.couchdb.org/en/1.6.1/. (Visited on 06/08/2015).

[4] Wayne Babich. *Software configuration management : coordination for team productivity.* Addison-Wesley, Reading, Mass, 1986.

[5] Peter Christen. A comparison of personal name matching: Techniques and practical issues, 2006.

[6] Wilbert Jan Heeringa. Measuring dialect pronunciation differences using levenshtein distance, 2004.

[7] Bendix L and Emanuelsson P. Diff and merge support for feature oriented development, 2008.

[8] Bendix L and Emanuelsson P. Requirements for practical model merge - an industrial perspective, 2009.

[9] Aron Lide. A state-based 3-way batch merge algoritm for models serialized in xmi, 2011.

[10] Koegel M, Herrmannsdoerfer M, von Wesendonk O, and Helming J. Operation-based conflict dtection, 2010.

[11] Antonio Martini. Merge of models: an xmi approach.

[12] Pablo Santos. Three-way merging: A look under the hood. http://www.drdobbs.com/tools/three-way-merging-a-look-under-the-hood/240164902, 12 2013. (Visited on 15/03/2016).

[13] Clement McDonald Shaun J. Grannis, J. Marc Overhage. Performance of approzimate string comparators for use in patient matching, 2004.

[14] Niraj Tolia, David G. Andersen, and Mahadev Satyanarayanan. Quantifying interactive user experience on thin clients, 2006.

A Implementation of merge algorithms

```
#!/usr/bin/python3
import json
import os
import time
import datetime
import sys
from pprint import pprint
from subprocess import call

def read_case(filename):
    with open(filename) as df:
        data = json.load(df)
    return data

# Remove the result from the data
def remove_result(data):
    dag = list()
    size = len(data)
    i = 0
    for r in data:
        i += 1
        result = r
        if (i != size):
            dag.append(r)
    return dag, result

# Print debug data
def print_error(calc, desi):
    print("==========================")
    print(calc)
    print("--------------------------")
    print(desi)
    print("==========================")

# Compare with desired result
def compare_result(calc, desi):
    res = set(calc["data"]) & set(desi["data"])
    for c in calc["data"]:
        if calc["data"][c] != desi["data"][c]:
            print_error(calc, desi);
            return "Error"
    return "Success"

def process_case(filename, function):
    data = read_case(filename)
    data, result = remove_result(data)
    calc = function(data)
    res = compare_result(calc, result)
    print(filename + ": " + res)

def empty_object():
```

```
52      jsonstr = ('{ "id": 0, "parent": [], "author": "",'
53                 '"timestamp": "", "data": { "name": "",'
54                 '"age": 0, "company": "", "address": "",'
55                 '"zip code": "", "city": "", "phone": "",'
56                 '"email": "" }}')
57      return json.loads(jsonstr)
58
59  # Return the minimum of three values
60  def minimum(a, b, c):
61      if (a <= b and a <= c):
62          return a
63      elif (b <= a and b <= c):
64          return b
65      else:
66          return c
67
68  # Return the maximum of two value
69  def maximum(a, b):
70      if (a < b):
71          return b
72      return a
73
74  # Find the levenstein distance between two strings
75  def levenstein(a, b):
76      a = a.strip()
77      b = b.strip()
78
79      if (len(a) == 0):
80          return len(b)
81      if (len(b) == 0):
82          return len(a)
83
84      cost = 1;
85      if (a[-1] == b[-1]):
86          cost = 0
87
88      return minimum(levenstein(a[0:-1], b) + 1,
89                     levenstein(a, b[0:-1]) + 1,
90                     levenstein(a[0:-1], b[0:-1]) + cost);
91
92  # Return the levenstein distance as a normalized
93  # value between 0 and 1
94  def norm_levenstein(a, b):
95      return levenstein(a, b) / maximum(len(a), len(b))
96
97  def merge_levenstein(a, b, wordlist):
98      # The distance between a and b is small.
99      # Assume spelling error and look
100     # for correct spelling in a word list.
101     if (norm_levenstein(a, b) < 0.6):
102         if (wordlist(a)):
103             if (wordlist(b)):
104                 return "MERGE_ERROR: " + a + "|===|" + b
105             else:
```

```
106                 return a
107         elif (wordlist(b)):
108             if (wordlist(a)):
109                 return "MERGE_ERROR: " + a + "|===|" + b
110             else:
111                 return b
112     return "MERGE_ERROR: " + a + "|===|" + b
113
114 def merge(a, b):
115     c = empty_object()
116     c["author"] = "Merge_tool"
117     ts = time.time()
118     unformated_datetime = datetime.datetime.fromtimestamp(ts)
119     c["timestamp"] = unformated_datetime.strftime('%Y-%m-%d')
120     parents = list()
121     parents.append(a["id"])
122     parents.append(b["id"])
123     c["parent"] = parents
124     cid = 0
125     if (a["id"] > b["id"]):
126         cid = a["id"] + 1
127     else:
128         cid = b["id"] + 1
129     c["id"] = cid
130
131     for i in c["data"]:
132         if (a["data"][i] == b["data"][i]):
133             c["data"][i] = a["data"][i]
134         else:
135             c["data"][i] = "MERGE_ERROR"
136     return c
137
138 # Check if there's a dependency between p, a and b
139 # on position i, in the dependency list deps
140 def check_dependencies(i, p, a, b, deps):
141     for item in deps:
142         if item == i:
143             return False
144     return True
145
146 # p = parent of both a and b
147 # a one object
148 # b on other object
149 def merge_with_parent(p, a, b, dependencies):
150     c = empty_object()
151     c["author"] = "Merge_tool"
152     ts = time.time()
153     unformated_datetime = datetime.datetime.fromtimestamp(ts)
154     c["timestamp"] = unformated_datetime.strftime('%Y-%m-%d')
155     parents = list()
156     parents.append(a["id"])
157     parents.append(b["id"])
158     c["parent"] = parents
159     cid = 0
```

```
160     if (a["id"] > b["id"]):
161         cid = a["id"] + 1
162     else:
163         cid = b["id"] + 1
164     c["id"] = cid
165 
166     for i in c["data"]:
167         if (a["data"][i] == b["data"][i]):
168             c["data"][i] = a["data"][i]
169         else:
170             # Check if this value depends on an other
171             # and in that case, if that also can be
172             # merged the same way
173             if (a["data"][i] == p["data"][i] and
174                 check_dependencies(i, p, a, b, dependencies)):
175                 c["data"][i] = b["data"][i];
176             elif (b["data"][i] == p["data"][i] and
177                 check_dependencies(i, p, a, b, dependencies)):
178                 c["data"][i] = a["data"][i];
179             else:
180                 c["data"][i] = "MERGE_ERROR"
181     return c
182 
183 # Function to ease debug
184 def print_list(l):
185     print("- list -")
186     for i in l:
187         print("ID: " + str(i["id"]) + " PARENT: " + str(i["
                parent"]))
188     print("- end -")
189 
190 def get_item_by_id(data, i):
191     for j in data:
192         if j['id'] == i:
193             return j
194     return False
195 
196 def get_changes(a, b, index):
197     changes = {}
198 
199     if a['data'][index] != b['data'][index]:
200         changes[a['data'][index]] = b['data'][index]
201     return changes
202 
203 def find_translations(data, index):
204     translations = {}
205     for i in data:
206         if (len(i['parent']) > 0 and i['parent'][0] != 0):
207             # Compare with each parent and get change
208             # Add change to translation if not already in the
                    translation
209             for j in i['parent']:
210                 parent = get_item_by_id(data, j)
211                 diff   = get_changes(parent, i, index)
```

```
212                 translations.update(diff)
213     return translations
214
215 # Applies a translation to a node, returns false if this would
216 # result in a loop.
217 def apply_translation(trans, target, item):
218     if target == item:
219         return True
220     for t in trans:
221         if (item == t):
222             return apply_translation(trans, target, trans[t])
223     return False
224
225 # data the original data to find translations from
226 # a one object
227 # b one other object
228 def merge_translations(data, a, b):
229     c = empty_object()
230     c["author"] = "Merge_tool"
231     ts = time.time()
232     unformated_datetime = datetime.datetime.fromtimestamp(ts)
233     c["timestamp"] = unformated_datetime.strftime('%Y-%m-%d')
234     parents = list()
235     parents.append(a["id"])
236     parents.append(b["id"])
237     c["parent"] = parents
238     cid = 0
239     if (a["id"] > b["id"]):
240         cid = a["id"] + 1
241     else:
242         cid = b["id"] + 1
243     c["id"] = cid
244
245     for i in c["data"]:
246         if (a["data"][i] == b["data"][i]):
247             c["data"][i] = a["data"][i]
248         else:
249             trans = find_translations(data, i)
250             success = 0
251             if (apply_translation(trans, a["data"][i], b["data"
                    ][i])):
252                 c["data"][i] = a["data"][i]
253                 success += 1
254             if (apply_translation(trans, b["data"][i], a["data"
                    ][i])):
255                 c["data"][i] = b["data"][i]
256                 success += 1
257
258             if success != 1:
259                 c["data"][i] = "MERGE_ERROR"
260     return c
261
262 def compare_value_at(l, i):
263     first = l[0]["data"][i]
```

```
264     for item in l:
265         if (item["data"][i] != first):
266             return False
267     return True
268 
269 def compare_value(val, l):
270     print("compare value")
271     print(val)
272     print(l)
273     toret = False
274     for a in l:
275         if a == val:
276             toret = True
277         else:
278             toret = False
279             break
280     print("toret: " + str(toret))
281     return toret
282 
283 def apply_multiparent_translations(trans, l):
284     for i in l:
285         for j in l:
286             if apply_translation(trans, i, j) and i != j:
287                 tmplist = list()
288                 once = True
289                 for k in l:
290                     if (k == j and once):
291                         once = False
292                         continue
293                     tmplist.append(k)
294                 if len(tmplist) == 2:
295                     if (apply_translation(trans, tmplist[0],
                            tmplist[1])):
296                         return tmplist[0]
297                     elif (apply_translation(trans, tmplist[1],
                            tmplist[0])):
298                         return tmplist[1]
299                     else:
300                         return False
301                 else:
302                     return apply_multiparent_translations(trans,
                            tmplist)
303     return False
304 
305 def merge_multiparent(l):
306     c = empty_object()
307     c["author"] = "Merge tool"
308     ts = time.time()
309     unformated_datetime = datetime.datetime.fromtimestamp(ts)
310     c["timestamp"] = unformated_datetime.strftime('%Y-%m-%d')
311     parents = list()
312     cid = 0
313     for p in l:
314         parents.append(p["id"])
```

```
315         if (p["id"] > cid):
316             cid = p["id"]
317     c["parent"] = parents
318     c["id"] = cid
319 
320     for i in c["data"]:
321         if (compare_value_at(l, i)):
322             c["data"][i] = l[0]["data"][i]
323         else:
324             c["data"][i] = "MERGE_ERROR"
325     return c
326 
327 def merge_multiparent_translations(data, l):
328     c = empty_object()
329     c["author"] = "Merge_tool"
330     ts = time.time()
331     unformated_datetime = datetime.datetime.fromtimestamp(ts)
332     c["timestamp"] = unformated_datetime.strftime('%Y-%m-%d')
333     parents = list()
334     cid = 0
335     for p in l:
336         parents.append(p["id"])
337         if (p["id"] > cid):
338             cid = p["id"]
339     c["parent"] = parents
340     c["id"] = cid
341 
342     for i in c["data"]:
343         if (compare_value_at(l, i)):
344             c["data"][i] = l[0]["data"][i]
345         else:
346             trans = find_translations(data, i)
347             tmplist = list()
348             for j in l:
349                 tmplist.append(j["data"][i])
350             res = apply_multiparent_translations(trans, tmplist)
351             if res != False:
352                 c["data"][i] = res
353             else:
354                 c["data"][i] = "MERGE_ERROR"
355     return c
356 
357 def next_id(data):
358     max = 0
359     for i in data:
360         if i["id"] > max:
361             max = i["id"]
362     return max + 1
363 
364 def compare(a, b):
365     for i in a["data"]:
366         if b["data"][i] != a["data"][i]:
367             return False
368     return True
```

```
369
370 def find_children(data, parent):
371     children = list()
372     for item in data:
373         for p in item["parent"]:
374             if p == parent["id"]:
375                 children.append(item)
376     return children
377
378 def adopt(data, new_parent, adoptees):
379     for d in data:
380         if d["parent"][0] in adoptees:
381             d["parent"][0] = new_parent
382
383 def partial_history_merge(data):
384     newlist = list()
385     droplist = list()
386     adoptlist = {}
387     has_new = False
388     for item in data:
389         children = find_children(data, item)
390         cont = False
391         for k in droplist:
392             if k == item["id"]:
393                 cont = True
394         if cont:
395             continue
396         for k,v in adoptlist.items():
397             if item["parent"][0] == k:
398                 item["parent"][0] = v
399         if len(children) == 2:
400             if compare(children[0], children[1]):
401                 new = merge(children[0], children[1])
402                 parents = list()
403                 parents.append(item["id"])
404                 new["parent"] = parents
405                 new["id"] = next_id(data)
406                 newlist.append(new)
407                 adoptlist[children[0]["id"]] = new["id"]
408                 adoptlist[children[1]["id"]] = new["id"]
409                 droplist.append(children[0]["id"])
410                 droplist.append(children[1]["id"])
411                 has_new = True
412             else:
413                 newlist.append(item)
414         else:
415             newlist.append(item)
416
417     if (has_new):
418         return partial_history_merge(newlist)
419     else:
420         return newlist
421
422 # Normalization of data
```

```
423 # ------------------------
424 # To prevent merge conflicts it's important to have
425 # normalized data. That is, data that follows a
426 # certain convention. For example to write post
427 # number as "217 56" and not "21 756". In all merge cases
428 # here we assume that the data is normalized. This can be
429 # done before saving the data, or before comparing the
430 # divergent branches. In the case when the data is normalized
431 # just for the diff, a random algorithm is used to determine
432 # which version has won.
433
434 # Detecting dublictes
435 # ------------------------
436 # Assuming two people are adding the same contact with maybe
437 # spelling errors and maybe a different number of fields
438 # entered. How can we merge this to one record? Are certain
439 # fields more important than others when detecting a match?
440 # Maybe two people from the same company shouldn't be
441 # considered as a duplicate but maybe two people with the
442 # same company and almost the same name should.
443
444 # Record linkage
445 # ----------------
446 # How to detect that different records are the same even if
447 # they appear to be different. See wikipedia.
448
449 # For detecting nicknames use phonetic algorithm such as soundex
    ,
450 # NYSIIS or metaphone.
451
452 # Case 1 is a conflict.
453 def case1(data):
454     return merge(data[1], data[2])
455
456 # Case 2 is a spelling error.
457 #
458 # We can calculate the Levenshtein distance between
459 # the mismatching fields. If the distance is low we
460 # probably have a spelling mistake.
461 #
462 # Since we know which field that has the conflict
463 # we can use a good list to find a correct spelling.
464 # For example if the field is a name field, we use a
465 # name register, if the field is an address field we
466 # use a address register.
467 def case2(data):
468     return merge(data[1], data[2])
469
470 # Case 3 is two newly entered values with a spelling
471 # error in one of them.
472 def case3(data):
473     return merge(data[0], data[1])
474
475 # Case 4 is two new entries with a misspelled name.
```

```
476 # This shouldn't be detected
477 def case4(data):
478     return merge(data[0], data[1])
479
480 # Case 5 is two fields that shouldn't be able to merge
481 # but result in a conflict since they are dependent on eachother
    .
482 def case5(data):
483     dependencies = { 'city':'zip_code', 'zip_code':'city' }
484     return merge_with_parent(data[0], data[1], data[2],
        dependencies)
485
486 # Case 6 is two fields that should be able to merge
487 # since they don't have a dependency.
488 def case6(data):
489     dependencies = { 'city':'zip_code', 'zip_code':'city' }
490     return merge_with_parent(data[0], data[1], data[2],
        dependencies)
491
492 # Case 7 is a case with history that should merge using the
493 # tranlation technique.
494 def case7(data):
495     return merge_translations(data, data[2], data[3])
496
497 # Case 8, todo, not implemented yet.
498 def case8(data):
499     return merge(data[1], data[2])
500
501 # Case 9, multistep translations
502 def case9(data):
503     return merge_translations(data, data[4], data[5])
504
505 # Case 10, multiparent
506 def case10(data):
507     items = list()
508     items.append(data[1])
509     items.append(data[2])
510     items.append(data[3])
511     return merge_multiparent(items)
512
513 # Case 11, voting should not be used
514 def case11(data):
515     items = list()
516     items.append(data[1])
517     items.append(data[2])
518     items.append(data[3])
519     return merge_multiparent(items)
520
521 # Case 12, multiparent with translations
522 def case12(data):
523     items = list()
524     items.append(data[1])
525     items.append(data[2])
526     items.append(data[3])
```

```
527     return merge_multiparent_translations(data, items)
528
529 # Case 13, multiparent with translations and different history
530 def case13(data):
531     items = list()
532     items.append(data[2])
533     items.append(data[4])
534     items.append(data[5])
535     return merge_multiparent_translations(data, items)
536
537 # Case 14, translations and mergeable history
538 def case14(data):
539     return merge_translations(data, data[3], data[4])
540
541 # Case 15, multiparent with translations and dubious if using
        voting
542 def case15(data):
543     items = list()
544     items.append(data[2])
545     items.append(data[3])
546     items.append(data[4])
547     return merge_multiparent_translations(data, items)
548
549 # Case 16, multistep translation and mergeable history
550 def case16(data):
551     data = partial_history_merge(data)
552     return data[-1]
553
554 # Case 17, translations that fails
555 def case17(data):
556     return merge_translations(data, data[3], data[5])
557
558 process_case("t/t1.json", case1)
559 process_case("t/t2.json", case2)
560 process_case("t/t3.json", case3)
561 process_case("t/t4.json", case4)
562 process_case("t/t5.json", case5)
563 process_case("t/t6.json", case6)
564 process_case("t/t7.json", case7)
565 process_case("t/t8.json", case8)
566 process_case("t/t9.json", case9)
567 process_case("t/t10.json", case10)
568 process_case("t/t11.json", case11)
569 process_case("t/t12.json", case12)
570 process_case("t/t13.json", case13)
571 process_case("t/t14.json", case14)
572 process_case("t/t15.json", case15)
573 process_case("t/t16.json", case16)
574 process_case("t/t17.json", case17)
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTATIONSDAG 2016-06-03

EXAMENSARBETE Merging customer relationship management data
STUDENT Fredrik Gustafsson
HANDLEDARE Lars Bendix (LTH)
EXAMINATOR Ulf Asklund (LTH)

Hitta konflikter och sammanfoga kunddata

POPULÄRVETENSKAPLIG SAMMANFATTNING **Fredrik Gustafsson**

Många företag använder ett kundhanteringssystem för att registrera interaktioner med sina kunder. I en allt mer mobil värld vill användarna av dessa system kunna använda systemen var de än befinner sig men oförändrad prestanda och funktionalitet.

De flesta kundhanteringssystem använder sig av någon form av server och client lösning. Det vill säga att en användare hämtar data från en gemensam server som alla användare delar och där all data finns. Detta gör att det alltid endast finns en version av datan och att alla användare alltid har tillgång till den senaste versionen av data.

För att kunna arbeta distribuerat med dåliga internet uppkopplingar eller kanske till och med utan internet över huvudtaget krävs det att varje användare själv har all data i systemet hos sig. Detta innebär att användarens version så småningom kommer skilja sig från andra användares versioner av datan när dessa ändrar i datan.

Olika kundhanteringssystem löser detta på olika sätt. Dessa har olika för och nackdelar. Tre olika kundhanteringssystem har analyserats för att se skilnaderna mellan dem och hur de på olika sätt hanterar distribuerat arbete. Resultaten har visat sig vara väldigt varierande.

De tre kundhanteringssystemen valdes inte efter deras populäritet utan efter deras olika angreppsätt mot distribuerat arbete. Här valdes ett webb-baserat systemm, ett system som dumt kopierar all data och sedan håller reda på vilka ändringar som är gjorda och ett system som använder ett lagringssystem med inbyggt stöd för distribuerad lagring.

För att kunna göra ovanstående analys har ett antal test fall tagits fram. Ett test fall är ett scenario där olika användare gör olika operationer på datan, exempelvis byter namn på en kontakt, och sedan när alla användare synkroniserar sina databaser med varandra ska den resulterande versionen innehålla rätt data. Vad rätt data är kan diskuteras men grundläggande principer är att data aldrig ska kastas bort och att användaren ska tillfrågas om systemet är osäkert på vilket resultat som är rätt.

För att kunna lösa alla test fall togs även ett antal lösningsmetodiker fram som beskriver hur man kan lösa olika typer av test fall. Här utforskas vilken annan data som kan sparas för att ge bättre lösningsförslag. Till exempel om det ger ett mervärde att inte bara spara senaste versionen av datan utan även historik för hur datan tidigare sett ut. Det framgår det att vetskapen om att det är just kunddata och inte någon annan typ av data (exempelvis fritext, bilder eller videor) ger extra kraftfulla verktyg.

Som bevis på att ovanstående metoder fungerar har dessa även implemeterats och test körts på samtliga framtagna test fall med gott resultat.