Anton Strand

Merging multiple hospital databases into a single shared database

SCHOOL OF TECHNOLOGY AND INNOVATIONS
Master of Science
Computer and Automation Technology

Vaasa 2022

UNIVERSITY OF VAASA
SCHOOL OF TECHNOLOGY AND INNOVATIONS

Tekijä:	Anton Strand
Tutkielman nimi:	Merging multiple hospital databases into a single shared database
Degree:	Master of Science
Degree Programme:	Computer and Automation Technology
Instructor:	Jouni Lampinen
Year of graduation:	2022 **Number of pages:** 44

ABSTRACT:

This work concerns the merging of eight hospital databases in the Finnish Ostrobothnia region. Each of them used their own database solution, making it hard for them to analyze data between hospitals. The purpose of this project is to make it possible for them to compare and analyze the data of all participating hospitals, making it possible to find new useful datapoints, and making it easier to find anomalies that previously would have been almost impossible to find when constrained to a single hospitals data. The work is needed because before this, no cross hospital analyzes were possible on a larger scale, and patient trends in the region were constrained to each hospital, making some of them more obfuscated than necessary, and the participating hospitals determined that this was something they all wanted to improve. Due to GDPR and patient rights, no patient identifying information was included in the scope of data, and restrictions had to be made on which people had right to see which data. The project was completed by first determining the scope of what database data that should be included on the table and column level for each hospital. Once that has been determined, the chosen data is gathered using SQL fetches, modified to a homogenous format using Python scripts and extracted to csv files. The resulting files are then sent to the shared server over SFTP and saved to the shared database using Python. After the data has been transferred to the database, then the Neotides reporting program is integrated to work with this merged data. Once the work was done, analysis of the participating hospitals patient data as a whole became possible, and patient flows between hospitals became possible to follow. This made it possible to analyze long term effects of procedures with greater accuracy, since the data was no longer constrained to patients revisiting the same hospital for follow up procedures and complications. The hospitals could also analyze their strengths and weaknesses compared to one another much easier and more accurately than before. These results among other things makes it possible for hospital management to improve their hospitals performance both for quality of care as well as long term cost per patient. To make even more precise analysis possible, the hospitals would have to work together to standardize their data collection practices in some way, making the starting point more homogenous. This would also make it easier for the hospitals to sometime in the future share visit data and other data with each other more easily for more than just management level reports and analyzes. Further work could also include increasing the scope of data to collect, and inviting more hospitals to be a part of it.

KEY WORDS: Merging databases, public healthcare, SQL

Abbreviations and explanations

SFTP	SSH File Transfer Protocol
SQL	Structured Query Language
SSH	Secure Shell
VSHP	Vaasan Sairaanhoitopiiri. (Vaasa Hospital District)

Table of contents

1 Introduction

In this project, the purpose is to gather data from hospital databases and merging that data in a shared database in a way that makes analysis and visualization of the data easy. The hospitals involved in this project are all located in the Finnish Ostrobothnia region. More specifically, participating hospital districts are as follows: Kaskinen, Kristiinankaupunki, Maalahti, Maksamaa, Mustasaari, Pietarsaari, Vaasa and VSHP. The company Neotide oversees this project, and my part in this was done while working fulltime under their payroll.

In order to not make this project scope too large, it was constrained to five different main tables and their associated lookup tables. These five are visits, ward periods, tooth clinic data, referrals, and emergency room data. Later more tables could be added if needed.

When sending the data to the shared server, no person identifying data of any patient is allowed, and GDPR compliancy is a given. Thus, a way to pinpoint visit data, ward period data, and other hospital data to a patient without making it possible to know who the patient in question is, needs to be implemented. Most other similar projects either don't take patient rights into account at all, or are old considerations that don't account for all things needed under GDPR. As an example of what a typical older paper on similar project contains regarding patient data and its usage, The University of Virginias Health System has a brief summary where multiple legacy clinical databases were merged. In it, it briefly mentions disguised internal identifiers, but offers no further information related to it afterwards (Scully, Pates, Desper, Connors, Harrell, Pieper, Hannan, Reynolds 1997: 32).

The hospitals in our project store their data in two SQL server types: Oracle sqldeveloper, and Microsoft SQL server. The data collecting systems also differ from hospital to hospital. VSHP, Kristiinankaupuni, Pietarsaari, and Vaasa toothcare use Effica(Tieto 2013) and Lifecare(Tieto 2013). Vaasa uses Pegasos, which is provided by CGI (CGI 2013). Mustasaari uses Oberon, which is provided by Logica. Kaskinen, Kristiinankaupunki and Maalahti use solutions provided by Abilita.

All these data storage solutions store their data in different ways. The key types of the different solutions are also different. Meanwhile the keys can overlap when multiple hospitals use the same database solution. The kind of data that is stored also differs from place to place. Something as simple as a column showing visit type is stored in many ways. Some places have tens of options for this, ranging from normal visit in special medical care, to group visit, to contact by telephone. Meanwhile others can have as few options as two: Special medical care, or primary health care. A way to modify these kinds of data groups to fit nicely together needs to be implemented.

In order to get valuable information from the data, it needs to be visualized and presented in a way that is easy to analyze. This will be done using the program Exreport provided by Neotide.

Once this has been implemented then reports can be done in a format where the users themselves can choose what data to include and what it should compare against. One possible analysis target would for example be the number of patients with same diagnose types in the different hospitals.

The reports should also make it possible to easily detect irregularities. For example, if visits per patient per year is remarkably higher in one hospital compared to the rest, or if patients are sent from one hospital to another and then sent back repeatedly. This means patient flows between hospitals should also be possible to analyze.

The whole data gathering, sending and merging will also have to be automated, so that the data can always be up to date without a large ongoing workload for maintainers just to get the data there.

2 Merging techniques

Relational databases consist of tables that have rows and columns. The columns tell what kind of data is in the table, the table is then populated by rows of data of these types. One or several of the columns must form the keys of the table, making it possible to pinpoint to the wanted row of data. When merging tables containing similar data, there are three types of structural conflicts that one needs to be wary of. Columns or keys can still be formed differently, the tables can use the same keys leading to duplicates when merged, and their dependencies can vary (Batini, Lenzerini & Navathe 1986: 346). In these cases, a new system needs to be formed that can hold the data from all tables while using the same format. Turning the combined database from a heterogenous system to a homogenous one (Batini, Lenzerini & Navathe 1986: 327]. In order to accomplish this, a number of mapping techniques need to be utilized:

2.1 Intermediate tables

In many cases, the columns in a main table utilizes codes and lookup tables as a data saving measure. If a diagnose name in the main table takes 15 characters per field on average, an alternative is to instead assign a code to each diagnose, and create a lookup table consisting of the code as its unique key and the text as another column containing the wanted data. Then whenever a row with a diagnose is created, it uses the code instead, and link to the lookup table to the main table using the code as a key can be used if you want to read the text the code is referring to.

But using lookup tables is not that straightforward when merging tables, there can for example be description conflicts, meaning there can be different names referring to the same data (Shamkanth & Suresh 1982: 150). A lack of object similarity can also be problematic (Shamkanth & Suresh 1982: 149). The tables can be using different codes to refer to the same data and can have different areas or information precision that the lookup table covers. For example, one of the lookup tables could be for diagnoses in a hospital's cancer department, and as such the diagnoses would only have to do with cancer, while the other lookup table is for diagnoses issued by a health clinic that doesn't do invasive

procedures and doesn't have a separate cancer department, and as such covers a lot more, but on more general detail level. As such the lookup tables are of different sizes, contain different keys, but sometimes refer to the same kind of data. But even when its referring to the same kind of data, the diagnoses could differ in the naming or precision given by the diagnose. Lets take a closer look using an example:

```
select * from table1 order by keyid
```

100 %

Results | Messages

	keyid	diagnose
1	1	a13
2	2	b2
3	3	a45
4	4	z87
5	5	a11
6	6	a45
7	7	a13
8	8	a13
9	9	z87
10	10	a11

Image 1 Patient visit data for the first table

Above in image 1 is an example of one table containing visit rows, with a key id and diagnose code, with a diagnose lookup table for diagnose names.

	dianoseid	diagnoseName
1	a13	fractured left humerus
2	b2	long finger flexor tendon rupture
3	a45	stress fracture to left scapula
4	z87	other
5	a11	broken left ulna

Image 2 lookup table for table 1

Image 2 shows the lookup data for the first table, and when using this data for reporting, you get the wanted information by joining the lookup tables diagnoseid column to table1s diagnose column, resulting in the data in image 3 below to be returned:

```
select keyid, diagnosename from table1
left outer join lookuptable
on diagnose=diagnoseid order by keyid
```

100 %

Results | Messages

	keyid	diagnosename
1	1	fractured left humerus
2	2	long finger flexor tendon rupture
3	3	stress fracture to left scapula
4	4	other
5	5	broken left ulna
6	6	stress fracture to left scapula
7	7	fractured left humerus
8	8	fractured left humerus
9	9	other
10	10	broken left ulna

Image 3 table 1 joined with its lookup table

This results in a data saving measure that is fast and easy to implement.

Now lets see what another clinics lookup table that we want to merge into the first one looks like:

	diagnoseId	diagnoseName
1	a10	laceration, left thigh
2	a11	avulsion, left thigh
3	a12	small cut, left thigh
4	a13	tetanus, left thigh
5	a14	torn muscle, left thigh
6	a15	fractured femur, left
7	a16	broken left ulna
8	a17	fractured left humerus
9	a18	stress fracture, left scapula
10	a19	long finger flexor tendon rupture
11	a20	other

Image 4 the second tables lookup table

As can be seen in Image 4, the diagnose IDs overlap, and the same ids don't refer to the same kind of diagnose.

When merging these tables an intermediate table for the lookups can then be used. You first take all the rows from both lookup tables into a combined list, and then remove duplicates that refer to the same data with a different code. A new diagnose key is

created so no overlaps can occur. Then you create an intermediate table that contains the hospital and diagnose code as keys, and the new key as the value to link to the new lookup table with. The resulting intermediate table from our two example tables can be seen in image 5 below.

	source	sourceDiagnose	mergedDiagnose
1	1	a1	a15
2	1	a11	a16
3	1	a13	a17
4	1	a45	a18
5	1	b2	a19
6	1	z87	a20
7	2	a10	a10
8	2	a11	a11
9	2	a12	a12
10	2	a13	a13
11	2	a14	a14
12	2	a15	a15

Image 5 What the resuting intermediate table looks like

This way, the data from the hospitals don't have to be fiddled with during data importing, and when you want to find rows for a cancer diagnosis from both tables, you can get data from both hospitals using the lookup table when you need to.

2.2 Key changing

Oftentimes, the diagnose names and other columns can have identical field names, with the only difference being the wording choice, or the key being different. In these cases, one can use the same intermediate table that is seen above, or simply change the key ID:s so all sources have the same key ID for the same field after migration, and the better worded field of the alternatives can be the field that all sources would use afterwards.

2.3 Tree hierarchies

Another mapping technique used in this project was concerning tables utilizing columns as tree hierarchies. A tree hierarchy consists of a number of nodes of data. Each node is

linked to a maximum of one parent node, and any number of child nodes (Knuth 1997: 308).

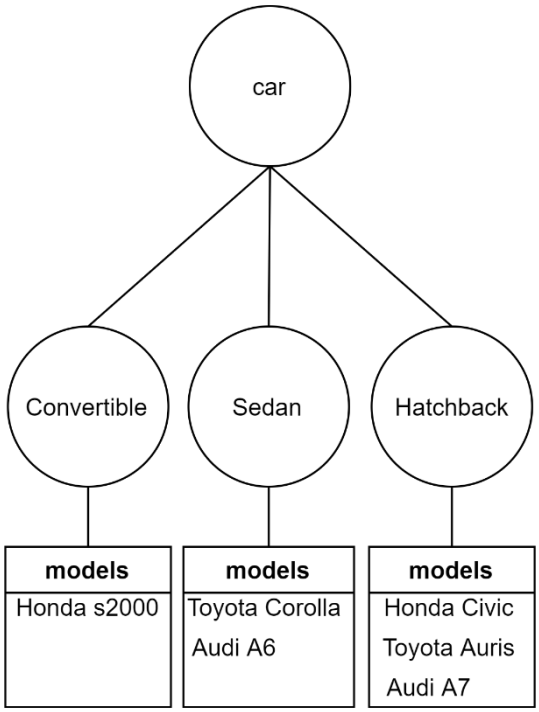

Image 6 Tree diagram with 3 levels

As the image above shows, Data can be organized nicely with tree hierarchies when the data can be organized in multiple levels of categories and subcategories.

But How can multiple hierarches be merged with one another? Let’s take the Image below as example of a tree hierarchy that needs to be merged with the previous one.

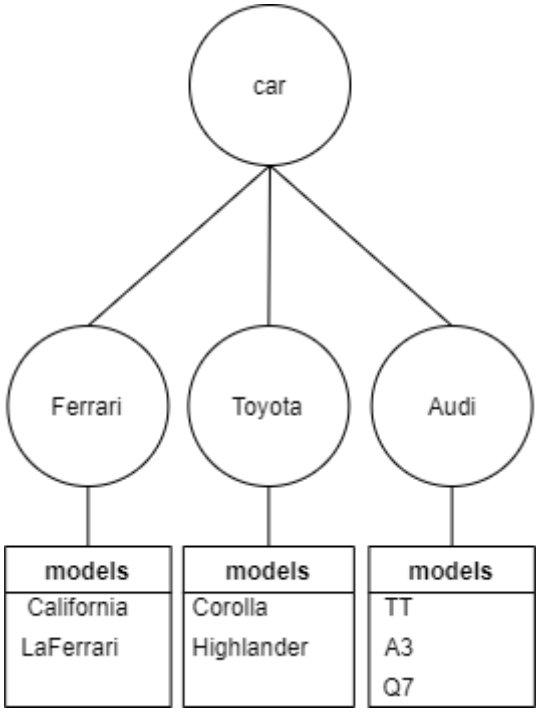

Image 7 Tree hierarchy 2

There are a number of differences between these two hierarchies in image 6 and image 7. Some of the models are the same, while others are different. The hierarchy methodology is also different. The first hierarchy's car section had it categorized as Car → body type → Maker+model. The second hierarchy has it categorized as Car → Maker → Models. In instances like these some manual work is required to merge them. Lets say that the end result needs to retain the info on car type from the hierarchy in image 1, then the relevant info needs to be figured out for the models in image 2.

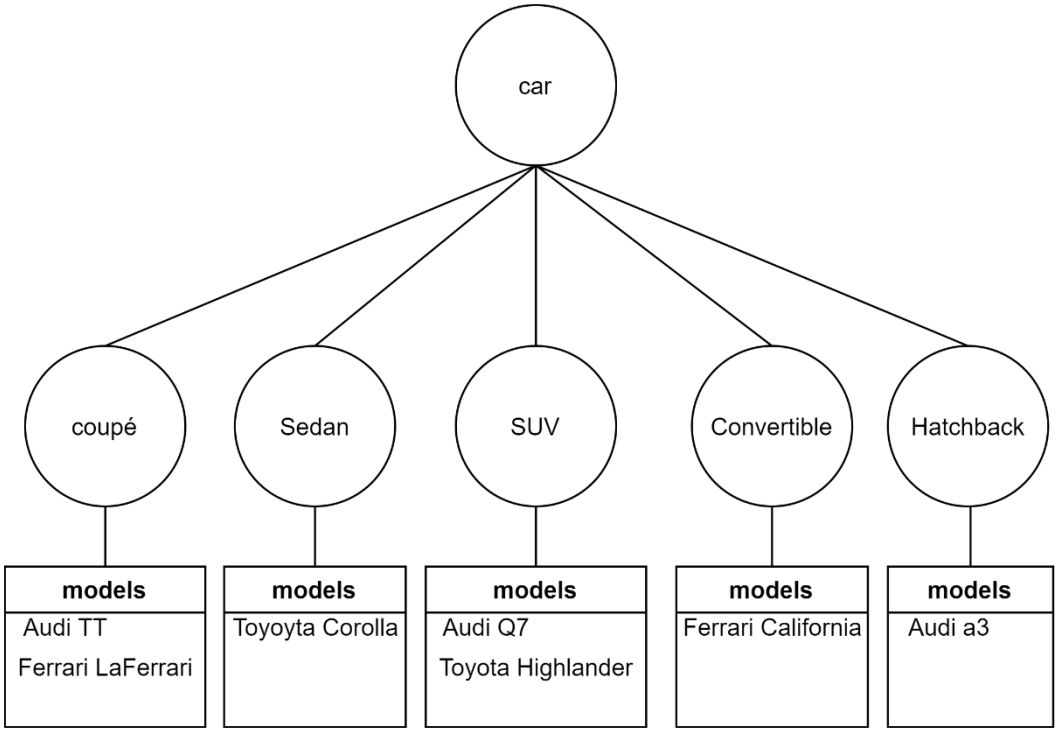

Image 8, hierarchy 2 changed for merging

Above in image 8 is the end result of the second hierarchy when its changed to the same type as hierarchy 1. Now as both hierarchies are of the same type and merging becomes easier. Simply take each model level node and add it under the same car type node in the other tree. If there are no fitting car type nodes in the first tree, then there are two options.

1) Import the car type node from tree 2 and link it to the “car” root.

2) Create an extra node in tree 1 before merging called “Others”, link all cars with body types that don’t fit in tree 1 into it.

Depending on the use case for the data, either one of these options could be more suitable than the other.

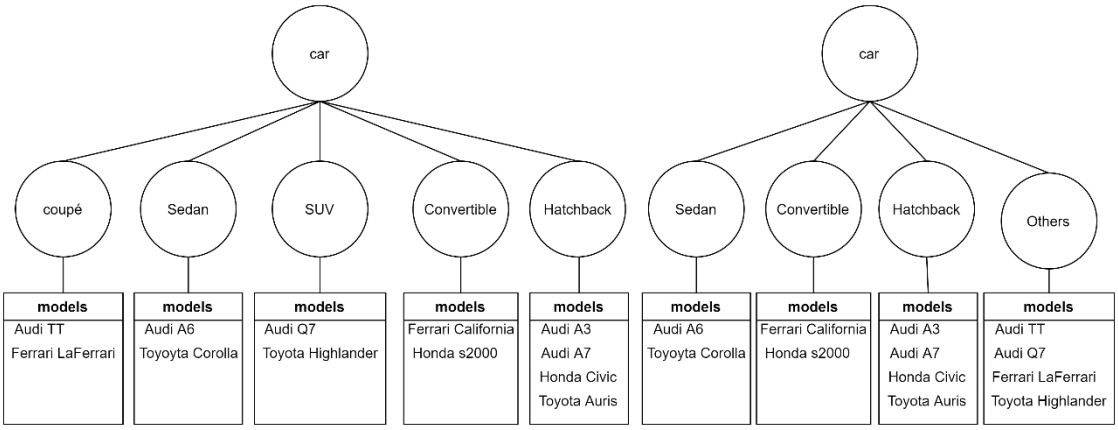

Image 9 the two end results

Above are the two end results. As can be seen, they both work fine, but what would happen if the tree that is supposed to be merged differs much more from tree 1 than tree 2 did?

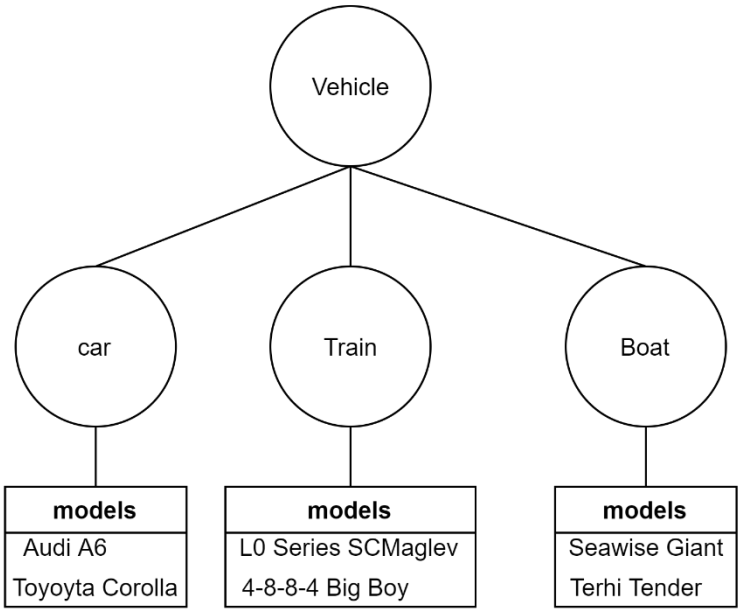

Image 10 More difficult tree to merge (Tree hierarchy 3)

Above in image 10, another type of tree hierarchy is shown. This one deals not only with cars, but other vehicle types as well. Each vehicle type is also categorized much simpler compared to tree 1. When merging this tree with tree 1, a decision must be made before starting. Does the end result need info on vehicle types other than cars or not? If the

part of the hierarchy with trains and boats needs to be merged as well, is it okay for the categorization to be simpler than the car part of tree 1, or should the trains and boats also be split up into what body type they have?

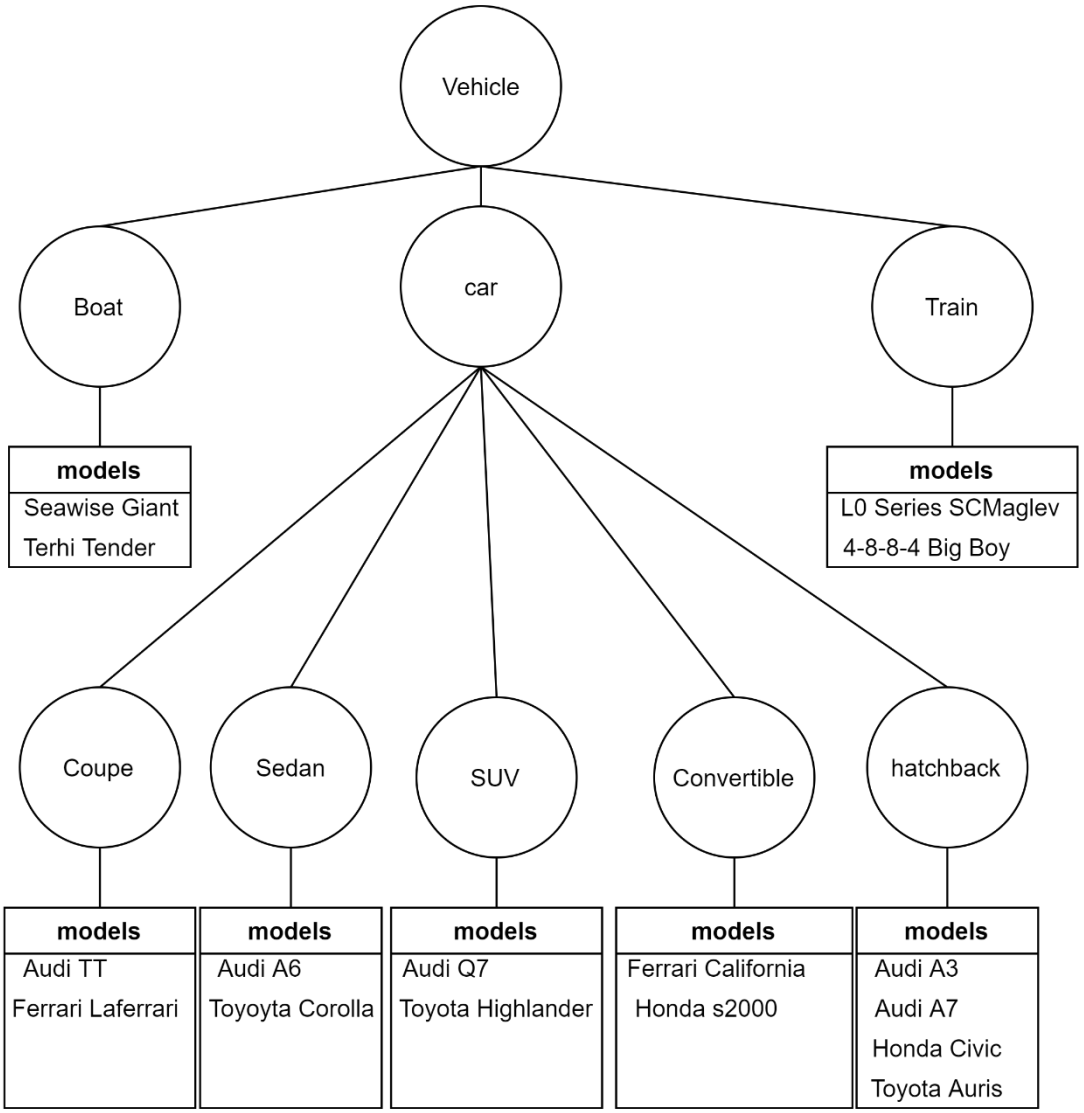

Image 11 Tree 3 merged with tree 1

Image 11 shows one of many possible merge results. The resulting hierarchy now has different amounts of nodes depending on vehicle type, so that the hierarchy for cars from tree 1 stayed the same, and the hierarchy for boats and trains in tree 3 stayed the same.

2.4 Value swapping

If two tables that need to be merged contain nearly identical data in a column, just with different names for the values, then one can swap one of the columns values to match the other in the merged product. In the 2001 paper titled Merging databases: Problems and examples by Cholwy and Moral, they go deep into the details on how to make efficient automatic versions of this in the chapter Numerical models for matching values. Fortunately for us, our use case is much simpler and smaller scale, so such extensive and complicated value swapping techniques aren't necessary. Instead, we can simplify their complex automatic process by manually changing the values in one table to match the other for values that mean the same or refer to the same thing. Simply go through distinct values in the primary table one by one, check if the other table has the same data with the same name, same data with different name, or if the data is absent. When the value names are different, put them in a list that can be used during merging to change the values of one of the tables to match the other. No other steps necessary. This can be applied both to text columns, and columns with lookup values. In the second case, simply change the lookup key instead of the value in the lookup table.

The scale of the data to apply this method to in this project is so small that doing this manually saves great time and effort compared to doing an automated method. In this project none of the columns in question used manual text input that can have typing errors or different ways to express the same thing depending on the person and their word choice, instead we just have to look through each hospitals list of options for said field and merge them. As such, this simplistic method works for our project, if this method was to be done in a more complex case, or if the scope of this project some day grows to include a hospital where manual typing is used for a needed column, then something like Cholwys and Morals technique would most likely have to be implemented.

3 Project implementation steps/methodology

This projects goal is to merge data from several hospitals and integrate them into a preexisting reporting program so it becomes possible to analyze data from all participating hospitals as a whole instead of being restricted to one. In order to make this reality we arrived at a plan consisting of three steps.

1)

Go through all concerning tables/databases, find out their similarities and differences, and pick out the columns/tables that seem suitable in the merged product and were included in the original scope. Leave out any data if its importance isn't obvious or the customers don't want them included to make the final data structure less chaotic. Once the first implementation has been done, more data columns can be added postmortem, and the users will get a clearer picture of what they still want to add once they have gotten a chance to use the reporting program. This way there are less guesses that need to be made concerning data to include in the starting phase, resulting in a cleaner and more easily maintanable product.

2)

Create SQL queries to gather the data selected in step 1 for a shorter time period. This means a sql query for each main table in each system concerning visits, ward period, referrals, emergency room tables and tooth clinic tables, as well as queries to get the lookup table data used by them. Once a more finalized structure has been made and any tests have been passed, we can add data from a longer time period to increase the analyzing capabilities of the program.

3)

Create scripts that take the data from the queries from all hospitals databases, modifies them to a homogenous format for the merged product, and sends the converted data to the shared server.

Once all data has been imported on this merged server, the reporting program can utilize this merged data for new and improved reports. The steps used resemble a simplified version of Generalized entity manipulator merging model first presented by Mannino

and Effelsberg (Batini et al 1986:357], and the first step was also critical in another similar project involving clinical databases done in the USA (Abrams, Kahn, Marrs & Steib 1994: 647).

It was necessary to go through each table in the databases by hand and by asking hospital staff for extra info when needed, since the db system providers for each hospital obviously wouldn't be happy to share info on specifics of their database schemas to a competitor. The only info we could get directly from them is publicly available information, such as older project plans for the initial implementations (Tieto.com 2013).

4 Determining what data to include

4.1 The tables to include

For this project, the focus was on five different data groups:

- Visits
- Ward periods
- Referrals
- Tooth clinic data
- Emergency room data

Visits, ward periods and referrals were the most obvious choices to include. Each hospital has a table containing visit data. It's usually the biggest data point used in analyzes internally, so visit data will obviously be included in the merged version as well. Most hospitals also have a table for ward periods, and substantial amounts of data within them. They also need a table for referrals to keep a track on where a patient came from or where its going. Most hospitals also have an emergency room, and the data from that table is often connected to the others in one way or another. Emergency room data is thus also an obvious choice to include. The odd one out is the Tooth clinic data, which usually is not connected to the rest in any way other than patient identifiers. Most hospitals also don't have tooth clinic data, which makes comparing tooth clinic data between hospitals not that useful. Generally, this kind of data specific to one schemata/database should be avoided when possible, since it adds complexity that can easily snowball the project scope and complexity when not used sparingly (Batini & Lenzerini 1983:653). But the amount of tooth clinic data available still meant that it could provide enough added value to analyzes in the merged product to be included, for example if a person from a small municipality goes to a neighboring one for their oral healthcare, we can check which municipalty the majority prefer to go to, as well as if there are some anomalies in visits after some oral healthcare procedures that may lead to increases in other visits.

Each of these also have some lookup tables associated with them, which must be included to get the data in a form that is more useful to the clients.

Most of these tables can also be connected to each other in some way. The ward period table often has a column with a visit identifier, which will then have the visit code if a long-term patient has gotten a procedure done on them. Likewise, both visits and ward periods can have referrals connected to them, which shows the patient flow both within the hospital, and from one hospital to another. But as we now can also have the ward and visit data for both hospitals involved, even more data should be useable in analyzes. Patients from the emergency room also often continue from there for a long term stay or a procedure, which logs a data point to the ward periods table or visits table.

4.2 What data to include from the tables in question

What columns should be included? First of all, all the essential columns, like timestamps, patient identifiers, diagnoses and such must be included. If we can't pinpoint to whom a procedure has been done, when the patient arrived, and why he/she was there in the first place, most other comparisons won't be useful.

Lets take the visit tables as an example. All hospitals visit tables contain timestamp data for when the patient arrived, and some also contain timestamp data for when a procedure started/ended, and when the patient finally left. All the visit tables also contain data such as diagnoses, patient identifiers, place of service, and specialty. Other than that, the data included starts to diverge between the hospitals. Some can have a list of all people involved, while others can have just the doctor/nurse giving the diagnose. The amount of time and timestamp columns and what they refer to also differs, from just visit/procedure length to timestamp when patient entered the room, doctor entered the room, anesthetic administration start time etc. For this project, it was determined that only the most general timestamps that most hospitals have would be included. The timestamps that are specific to just one hospital would be excluded, since it would be left out in comparisons and analyzes for the hospitals as a group anyway. But for the ones included, extra care has to be used, since its easy to accidentally group the wrong timestamps together.

But when looking past the absolutely necessary data columns, then the focus should be shifted to columns that most of the hospitals use, so that the data can be compared. This then enables each hospital to see what they can improve on and makes it possible to find anomalies that otherwise potentially wouldn't be detected.

In some cases, the hospitals have almost the same data stored in a column, just worded differently from hospital to hospital, or grouped in a different way, or with only part of the data matching. In these cases, merging them could still be possible in a way to gather some useful data.

Data that can only be found in one or two places but that is deemed important enough should still be included even direct comparison with other hospitals won't be possible for it, at least not directly. Including the data means that it can then be used to possibly figure out why the hospital in question has some metric that differs from the rest in some way.

Other than these kinds of columns, all data should be left out to improve the reportin programs responsiveness, lower the implementation work time and minimize disk space usage.

5 Sending the data to the shared server

First an SQL query is run to get the data we want, and using python, the results of this query are then modified so small differences between hospital data types are made homogenous using the techniques in chapter 2.1-2.4. Some other measures also must be done to the data before it can be sent. Due to GDPR, user identifying data cannot be sent without their permission. This means data such as birthdates, social security numbers, and first/last names cannot be included. But we still want to make it possible to observe patient flows between hospitals. To make this possible, we used a one-way hash on each social security number with no other highly personal data included, which makes it possible to pinpoint each data row down to an individual, without knowing who the individual in question is. The hashed column would also be visible only the people that the group of hospitals agreed have a need to see such low level data.

After the data has been modified to its final form, the data is then saved to a csv file. After all queries have been run and all files have been made, the csv files are sent to the shared server using a secure sftp link between the shared server and the hospital servers. Each hospital had to make sure the connection was allowed. Both legally by getting the required permits, and in practice by opening the connection in their respective firewalls. Although new connections to places outside the hospitals aren't preferable, making them one-way connections is still the best option in this case.

6 The implementation

Now that we have gone through some merging techniques and the steps needed, we still have to go through how the data from the hospital databases were gathered, merged and ultimately used in a reporting program, and what steps were taken in order to make it work practically.

6.1 The code to make this all happen:

The code below contains part of the basic python batch script that all sites will run variations of. It uses a base batch class that Neotide has worked on over the years, on top of which we add the wanted functionality for our specific usecase. It starts off with a couple of the basic things that are needed for the transfer scripts. First, it lists of several variables and their values, leading to clearer code, and an easier time for the coder if some of the variables need to be changed. For example, if the file directory should be changed in the future.

```
class Batch(base.Batch):
    bulktable_name = 'datapool'
    description = 'Exporting data to shared datapool'
    gdpr_title = 'Datapool'
    gdpr_group = 'export'
    file_directory = os.path.join(base.Batch.source_path,
'files', 'datapool')
    secret = "identifier for some secret code"
    sourcedb = None

    def compute_sha(self, s):
        return hashlib.sha1(s.encode('utf-8') + self.se-
cret).hexdigest()

    def get_file_path(self, name, start_date):
        now = datetime.datetime.now()
        filename = config.site + '_' + name +
start_date.strftime("%Y%m%d") + '_' +
now.strftime("%Y%m%d") + '.csv'
        path = os.path.join(self.file_directory, name)
        os.makedirs(path, exist_ok=True)
        return os.path.join(path, filename)
```

The method compute_sha is used to anonymize some data that we still want to be able to analyze, but that not all users of the software doing the analyzing would need to see for it to work. Specifics will appear later when this method is actually called upon.
The method get_file_path, as the name suggests, figures out the file path that the file should be saved to.
The batch code continues below with a method called get_values. With it, we take in the parameter columns, containing a list of tuples, with the values for the column name in the database, as well as the name the column should have in the written file. Another input parameter that we will see often is cursor, which is used to connect to the database and run SQL scripts to gather the data that we want to be transferred.The third parameter is a string containing of the SQL we want to run, and the three last input parameters that make optional modifications to the script. This type of code used in Neotide for almost all cases involving interacting with a database, just modified to fit each usecase.

```
def get_values(self, columns, cursor, sql, start_date,
sql_args=None, timestamp_sql_args=None):
    headers, columns = zip(*columns)
    if sql_args is None:
        sql_args = {
            'columns': cursor.escaped_sql(', '.join(col-
umns)),
        }
    if timestamp_sql_args is None:
        timestamp_sql_args = {
            'start_date': start_date,
        }
    sql_args = {**sql_args, **timestamp_sql_args}
    cursor.execute(sql, sql_args)
    for row in cursor:
        values = {}
        zip_row(headers, row, values)
        yield values
```

The contents of the get_values method itself is pretty simple. It takes the inputs, makes the basic needed modifications if none are given, and then runs the sql script. Afterwards it goes through the rows returned by said script line by line, creating a tuple containing the column names, row number, and values using Python 3:s builtin zip_row method, and lastly yields the result for further manipulation.

The batch file continues with the code below, where the method write_csv takes in data that has been retrieved previously with get_values and potentially done some further modifications to, and writes the row to the path specified in get_file_path.

```
@contextmanager
def write_csv(self, columns, name, start_date):
    def to_str(v):
        if isinstance(v, datetime.datetime):
            return db.datetime(v)
        return v

    def write(values):
        csv_writer.writerow(to_str(values.get(c)) for c in
headers)

    headers, columns = zip(*columns)
    path = self.get_file_path(name, start_date)
    with open(path, 'w', newline='') as f:
        csv_writer = csv.writer(f, delimiter=';')
        csv_writer.writerow(headers)
        yield write
```

Below is one of the simplest examples showing what the previous code parts all lead to. In this case, products from a source using effica:s database architecture are gathered and written to a file.

```
def read_effica_tuote(self, start_date):
    self.running_timestamp()
    columns = [
        ('source', self.source),
        ('tuote', 'tuote'),
        ('tuoteryhma', 'tuoteryhma'),
        ('name', "name"),
    ]
    sql = "select %(columns)s from tuote"
    with self.write_csv(columns, 'tuote', start_date) as
write:
        for values in self.get_values(columns, self.cursor,
sql, start_date):
            write(values)
```

The read_effica_tuote method starts off with listing the columns it will gather from the database and write to a file. The column, being a list of tuples, has four pairs in this case: source (meaning which database this data originated from), product, product group, and

name. The right part of the tuple showing what will be inputted to the SQL to get these values, and the left showing what the column names will be in the written file.

The first column, source is a slight anomaly when compared to the rest. Since source is specified within the file itself and will always return the same string containing the data that has been specified there.

The next part of the code shows the SQL that will be run. In this case it fits nicely on a single line, but in some cases, these SQLs can be quite hard to read, containing hundreds of lines of code with dozens of joined tables, subqueries, and other restrictions and modifications to get the data needed. Showing those SQLs could reveal so much of the database structure that a rivaling product could potentially be made using it, which is why I will only show these simple sqls that don't reveal nearly enough to do something like that.

The last thing the read_effica_tuote method does, is start a loop where write_csv and get_values are called, the data is retrieved, and then written to the specified file. This concludes the retrieval and writing part of the batch, since all that is left is now a dozen or so methods like read_effica_tuote where different parts of the database are retrieved and written to their own files. But the batch still contains one more thing, the part taking care of the transfer of the resulting files, which is seen below in the method transfer_files.

```
def transfer_files(self, start_date):
    self.running_timestamp()
    directories = collections.defaultdict(list)
    for base_dir, dirs, files in os.walk(self.file_direc-
tory):
        folder = base_dir.split('\\')[-1]
        for file in files:
            modified_datetime =
datetime.datetime.fromtimestamp(os.path.getmtime(os.path.jo
in(base_dir, file)))
            if modified_datetime >= start_date:
                directories[folder].append(file)
    with paramiko.Transport(('101.101.101.10', 55)) as t:
        t.connect(username=r'vsvd\exampleuser', pass-
word='123!”#jkl)(/78')
        with paramiko.SFTPClient.from_transport(t) as sftp:
            for folder, files in directories.items():
                directory = os.path.join(self.file_direc-
tory, folder)
                sftp.chdir(None)
                try:
                    sftp.chdir(directory)
```

```
                except IOError:
                    sftp.mkdir(directory)
                    sftp.chdir(directory)
                for file in files:
                    file_path = os.path.join(directory,
file)
                    sftp.put(file_path, file)
                    os.remove(file_path)
```

The transfer_files method goes through all files that exist in the specified file directory, selects the files that are newer than the last batch run time, and then uses the paramiko library to easily connect to the wanted IP and sending the files to the wanted directory using SFTP. Once the file has been successfully copied to target location, then the original file is removed since it is no longer needed. The choice of paramiko as the SFTP library to use was chosen by others in Neotide that had more knowhow on the subject.

One of the slightly more complicated things to implement is the organization hierarchy modifications. Since the organization structure can vary wildly between the source systems, a two-part code implementation was implemented with the help of one of the founders of the company, Patrik Simons. Lets first take a look at the method get_organisation_hierarchy_values. Most of the source and target column values can not be determined in the beginning, so only year and source are set, since they are the same for all instances in the different hospitals. After that base values for the intended leaf-node implementation are set. After that, the input arguments are looped through, hierarchy level and other column values are calculated and set, and the resulting column values are then returned.

```
def get_organisation_hierarchy_values(self, namefi, namesv,
year, *args, max_level=5):
    values = {
        'source': self.source,
        'year': year
    }
    prefix = self.source + '_'
    hierarchy_level = self.source + '^'
    org_code = self.source
    parent = None
```

```
        for value in args:
            if value is not None and value.strip():
                parent = org_code
                org_code = prefix + value.strip()
                hierarchy_level += org_code + '^'
        level = hierarchy_level.count('^')
        values.update({
            'org_code': org_code,
            'hierarchy_level': hierarchy_level,
            'leaf': 1 if level == max_level else 0,
            'level': level,
            'namefi': namefi,
            'namesv': namesv,
            'parent': parent,
        })
        return values
```

In order to visualize this slightly better, Image 7 below shows the result of running this method for one organization row.

	source	org_code	year	hierarchy_level	leaf	level	namefi	namesv	parent
1	1	0101	2016	1^0101^	1	2	Oravaisten neuvola	Rådgivningen i Oravais	1

Image 12 The resulting column values for one instance of the previous method

Source, org_code and year makes it possible to pinpoint the original row in the source system. Hierarchy_level modified the input data into a form readable in target system regardless of source structure, leaf and level makes it easy to group the row with other wanted ones, for example if we wanted to know all the child levels of a target hierarchy level.

But since get_organisation_hierarchy only takes input arguments that have already been modified to the needed level for creating the mergeable product, there is still some prep work needed for the data on each source organizational structure to extract the data in this form and in the end save all the modified rows to the csv file. In the read_effica_organisation method, we see what one of these source specific organization methods look like. But since this method takes up about two pages, I have split it up so we can more easily analyze each part of it.

```
    def read_effica_organisation(self, start_date):
        self.running_timestamp()
        columns = [
            ('source', self.source),
            ('org_code', 'suorituspaikka'),
```

```
            ('year', 'sp.year'),
            ('hierarchy_level', 'suorituspaikka'),
            ('leaf', '1'),
            ('level', '1'),
            ('namefi', 'name'),
            ('namesv', 'name'),
            ('parent', self.source),
        ]
        top_values = {
            'source': self.source,
            'org_code': self.source,
            'year': start_date.year,
            'hierarchy_level': self.source + '^',
            'leaf': 0,
            'level': 1,
            'namefi': datapooldb.DatapoolSource.table_con-
    tent[int(self.source) - 1][1],
            'namesv': datapooldb.DatapoolSource.table_con-
    tent[int(self.source) - 1][2],
        }
```

Here, the column initialization lists all source and target column names. Source, leaf, level and parent always have the same starting values, so we just initialize them with the values we want when modifying the first row after the SQL has been executed.

The top_values dictionary, as the name suggests, sets up what the top level will look like in this instance. If we for example had Pietarsaari using the effica system, then source, ord code and hierarchy level would have the source tables Pietarsaari key as value. nameefi and namesv would then be Pietarsaari and Jakobstad, and the reporting program would just show the one matching with the user language as the top level name for this organization hierarchy.

```
    missing_values = [
        {
            'source': self.source,
            'org_code': '5_tulal_ZZ',
            'year': start_date.year,
            'hierarchy_level': '{}^{}^'.format(self.source,
'5_tulal_ZZ'),
            'leaf': 0,
            'level': 2,
            'namefi': 'Okänd - Tuntematon',
            'namesv': 'Okänd - Tuntematon',
            'parent': self.source,
        },
        {
            'source': self.source,
            'org_code': '5_tulyks_ZZ',
```

```
            'year': start_date.year,
            'hierarchy_level': '{}^{}^{}^'.format(self.source,
'5_tulal_ZZ', '5_tulyks_ZZ'),
            'leaf': 0,
            'level': 3,
            'namefi': 'Okänd - Tuntematon',
            'namesv': 'Okänd - Tuntematon',
            'parent': '5_tulal_ZZ',
        },
    ]
```

The effica organization has four levels in its structure. Tulosalue at top level, tulosyksikkö at second level, kustannuspaikka at third level, and suorituspaikka at bottom level. But Pietarsaari has one part of their organization that doesn't have this many levels, so we need to add the missing levels in order to make the target system know what level these anomalous ones actually belong at The code snippet above shows the placeholder data and names for the missing organizational levels. If Pietarsaari changes the structure for this part of their organization to match the rest in the future, then it can be easily changed in the code as well by removing the missing_values list.

```
        with self.write_csv(columns, 'organisation',
    start_date) as write:
            write(top_values)
            for value in missing_values:
                write(value)
            self.cursor.execute(
    'select suorituspaikka, kp.kp, tulyks, tulal, name, year
    from suorituspaikka sp'
    ' join kp on sp.kp = kp.kp and kp.vuosi = %(year)s'
    ' where sp.year = %(year)s', {'year': start_date.year})
```

Now The csv_writing is initialized by calling write_csv. For the first row, the top_values dictionary is written, after which the missing values are gone through. When that is done, we start going through the meat of the code that we want to be written. Now, we run the SQL query that gathers all organization data from the source systems suorituspaikka table.

The rest of this code snippet is all inside the "with self.write_csv" part. Two tabs have been removed from the continued code below to make it more readable.

```
      for sp, kp, tulyks, tulal, name, year in self.cursor:
        # Prefix tulal, tulyks and kp. They can have the same
    value and the hierarchy can't handle it.
```

```
    tulal = 'tulal_' + tulal if tulal.strip() else None
    tulyks = 'tulyks_' + tulyks if tulyks.strip() else None
    kp = 'kp_' + kp if kp.strip() else None
    hierarchy_values = self.get_organisation_hierarchy_val-
ues(name, name, year, tulal, tulyks, kp, sp)
    write(hierarchy_values)
    self.cursor.execute(
'select kp, tulyks, tulal, yksnimi, vuosi from kp'
' where vuosi = %(year)s', {'year': start_date.year})
    for kp, tulyks, tulal, name, year in self.cursor:
        tulal = 'tulal_' + tulal if tulal.strip() else None
        tulyks = 'tulyks_' + tulyks if tulyks.strip() else
None
        kp = 'kp_' + kp if kp.strip() else None
        hierarchy_values = self.get_organisation_hierar-
chy_values(name, name, year, tulal, tulyks, kp)
        write(hierarchy_values)
```

The SQL query that was run returns rows that each contain the suorituspaikka, kustannuspaikka, tulosyksikkö, tulosalue, name, and the year it belongs to. With this, we know the hierarchy that even the lowest level units belong to. The parameters are then modified, after which the method get_organisation_hierarchy_values is run with the parameters in question. Once this is done. A similar query as the one before is run, this time gathering data from the kustannuspaikka table instead of the suorituspaikka table. The returned rows are then modified to the same format as before, after which the get:organization_hierarchy_values is run again with these new values. Once this is done, the whole organization hierarchy will have been read and modified to the form that the merged organisation hierarchies will use.

Some of the hospitals have separated the service departments from the rest of the organization hierarchies into its own column. The below code gets the values from the connected lookup table.

```
def read_abilita_laitososasto(self, start_date):
    self.running_timestamp()
    columns = [
        ('laitos', "'MUUT'"),
        ('osasto', 'serviceproducent'),
        ('namefi', 'text'),
        ('namesv', 'text'),
    ]
    sql = "select %(columns)s from abilita_serviceavdelning
where isnumeric(serviceproducent) = 0"
```

```
        with self.write_csv(columns, 'laitososasto',
    start_date) as write:
            for values in self.get_values(columns, self.cursor,
    sql, start_date):
                write(values)
```

The data for ones that have a connected wards can be found in another table with data on all departments, which is why the SQL query filters out all the ones with a ward. In this abilita table, rows with missing ward columns have a 0 instead, which is the query looks as it does.

The gathered data is then saved to a laitososasto csv file for merging.

This about covers the methods that can be shown without showing too much of the source systems table structures.

The rest of the code shown will instead focus on other parts needed in this project.

```
class Batch(datapool_export.Batch):
    gdpr_detail = (
        'Potilas- ja käyntitietoja',
        'Patient- och besöksuppgifter'
    )
    source_defaults = config.pegasos_defaults
    thread = 'pegasos'
    source = '4'

    def read_files(self, start_date):
        self.read_diagnoosit(start_date)
        self.read_hammashuolto_visits(start_date)
        self.read_lahete(start_date)
        self.read_laitososasto(start_date)
        self.read_organisation(start_date)
        self.read_osastojaksot(start_date)
        self.read_osastojaksot_shell(start_date)
        self.read_visits(start_date)
        self.read_palvelu(start_date)
        self.read_palvelutila(start_date)
        self.read_palvelusuunnitelma(start_date)
        self.read_tuottaja(start_date)
        self.transfer_files(start_date)
```

The above code shows part of the batch run in each source system. Diagnoses, ward periods, visits, and other wanted data is gathered from the source system, in this case using the pegasos database structure. After all data has been gathered in the csv form, then each csv file is sent to the shared server for merging.

```
modules = [
    ('site.datapool.datapool_lookup', 'datapool_lookup',
False),
    ('site.datapool.datapool_diagnoosit', 'datapool_di-
agnoosit', False),
    ('site.datapool.datapool_osastojaksot', 'data-
pool_osastojaksot', False),
    ('site.datapool.datapool_vaesto', 'datapool_vaesto',
False),
    ('site.datapool.datapool_visits', 'datapool_visits',
False),
    ('site.datapool.datapool_lahete', 'datapool_lahete',
False),
    ('site.datapool.datapool_palvelusuunnitelmat', 'data-
pool_sosiaali', False),
    ('site.datapool.datapool_organisation', 'datapool_or-
ganisation', False),
    ('site.datapool.datapool_raisoft', 'datapool_raisoft',
False),
    'deletestalesessions',
    'tunedb',
    'refreshcache',
   'backup',
]
```

The above code shows parts of what the runbatch looks like in the shared database. It shows which python scripts to run, in the order top to bottom. This takes care of the last step in the automation process, making sure all the data that has been sent to the server is imported to the database My work only includes to rows from datapool_diagnoosit' to datapool_organisation. The tunedb and other rows here involve processes to among other things improve the performance of the database that Exreport will use.

6.2 Integrating the reporting program to the new DB

```
/****** Script for SelectTopNRows command from SSMS  ******/
SELECT TOP (1000) [source]
      ,[org_code]
      ,[year]
      ,[hierarchy_level]
      ,[leaf]
      ,[level]
      ,[namefi]
      ,[namesv]
      ,[parent]
  FROM [exreport].[dbo].[datapool_organisation]
```

100 %

Results | Messages

	source	org_code	year	hierarchy_level	leaf	level	namefi	namesv	parent
1	1	0101	2016	1^0101^	1	2	Oravaisten neuvola	Rådgivningen i Oravais	1
2	1	0101	2019	1^0101^	1	2	Oravaisten neuvola	Rådgivningen i Oravais	1
3	1	0101	2020	1^0101^	1	2	Oravaisten neuvola	Rådgivningen i Oravais	1
4	1	0102	2016	1^0102^	1	2	Vöyrin neuvola	Rådgivningen i Vörå	1
5	1	0102	2019	1^0102^	1	2	Vöyrin neuvola	Rådgivningen i Vörå	1
6	1	0102	2020	1^0102^	1	2	Vöyrin neuvola	Rådgivningen i Vörå	1
7	1	0103	2016	1^0103^	1	2	Maksamaan neuvola	Maxmo rådgivning	1
8	1	0103	2019	1^0103^	1	2	Maksamaan neuvola	Maxmo rådgivning	1
9	1	0103	2020	1^0103^	1	2	Maksamaan neuvola	Maxmo rådgivning	1
10	1	0104	2016	1^0104^	1	2	Särkimo rådgivning	Särkimo rådgivning	1
11	1	0104	2019	1^0104^	1	2	Särkimo rådgivning	Särkimo rådgivning	1
12	1	0104	2020	1^0104^	1	2	Särkimo rådgivning	Särkimo rådgivning	1
13	1	0105	2016	1^0105^	1	2	Työterveyshuolto Vöyri	Företagshälsovården Vörå	1
14	1	0105	2019	1^0105^	1	2	Työterveyshuolto Vöyri	Företagshälsovården Vörå	1
15	1	0105	2020	1^0105^	1	2	Työterveyshuolto Vöyri	Företagshälsovården Vörå	1
16	1	0106	2016	1^0106^	1	2	Työterveyshuolto Oravainen	Företagshälsovården Oravais	1
17	1	0106	2019	1^0106^	1	2	Työterveyshuolto Oravainen	Företagshälsovården Oravais	1

Image 13 View of one table in the migrated db.

Once all data has been migrated over, starts the integration phase into the reporting program. Image 13 above shows what one of the tables to integrate looks like, the table containing the combined organization levels. The reporting program can be used for data spanning several years back, and since the hospitals organization hierarchies can change over the years, the year column was added. Using it, we can pinpoint to what the hierarchy looked like at a certain point in time. The hierarchy level is the column containing the data that pinpoints each data row to their corresponding position in the hierarchy, the data rows seen in this image only contains data on the top level for organizations in

the mustasaari district, but if we were to scroll and look at more rows, then we would soon get more organizations and levels within each organization.

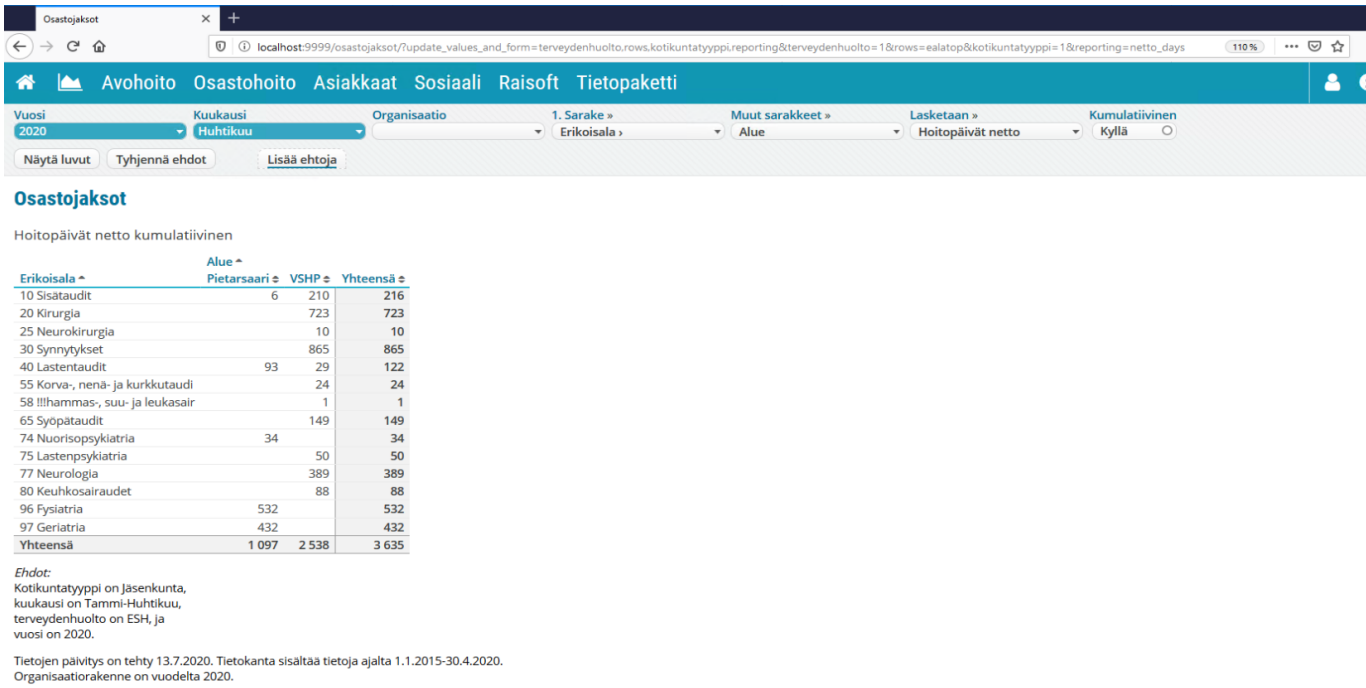

Image 14 One of the main reporting pages of the shared reporting program

Image 14 shows what the User interface for the reporting program looks like. At the top we have main reporting areas split up in Outpatient care(Avohoito),Ward care (Osastohoito), Clients(Asiakkaat), Social(Sosiaali), Raisoft and Tietopaketti. The original scope of the project and the part I was involved with included the first three of these, and I wont be going into detail on the last three because of this.

The currently opened reporting page is for ward periods, one of the reporting pages under Ward care. The reporting system consists of three main areas that make it possible to modify the shown data to the preferred form. First, there are year(Vuosi) and month(Kuukausi) selections, making it possible to pinpoint the wanted date range for the report. 1.Sarake, meaning first column, shows the main column that the report will be based on, in this case which specialty the ward period belongs to. The Other columns (Muut sarakkeet) option specifies what the other columns will be. In this case the chosen value was the district(Alue), slitting up the rows of found data into two columns for each specialty. More columns can be added if wanted, for example if we also wanted to see

age range for each field, then we would add the age-range option as a new column, which would for example split up each specialty rows district field into two if two age ranges were chose. The most important bit of this page is the Summing filter(Laksetaan), which specifies what the numbers in the fields actually mean. In this case, the option for ward days were chosen. Meaning if one field had data for two patients, one staying one day and one staying three days during the month in question, then the field would show four as the value.

The third menu option is for the organization, which Makes it possible to filter out data belonging to units/wards that we aren't interested in to focus on just the relevant units. The organization drop down can be seen in more detail in image 10 down below.

The first column in the report can be changed to almost any column in the ward table, in this case erikoisala(specialty) was chosen. This results in one row for each type of ward specialty, splitting up the data into those that have to do with internal medicine, surgery, neurology etc. And with this, we have everything the report needs to show the data. If we for example wanted to compare the neurology unit with the surgery unit, we could see that the neurology department had a load of 10 patient days during the selected period, while the more general surgery had 723.

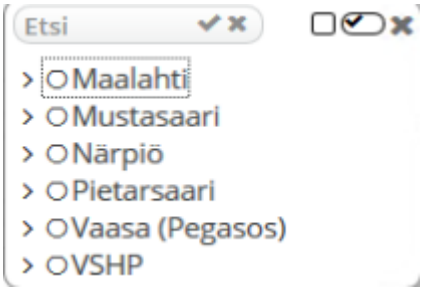

Image 15 organization hierarchy top level

Image 15 shows the top level of the organization hierarchy. Each municipality have their own top level, which can be opened up to select or deselect more specific units that the municipality in question have.

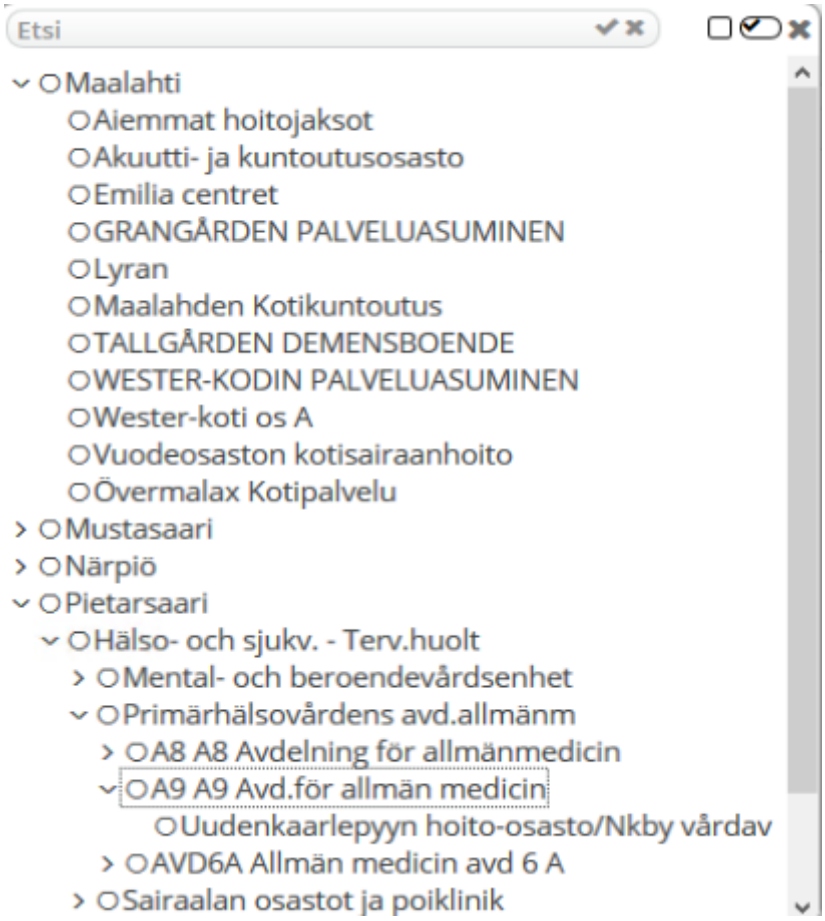

Image 16 Organization hierarchy in the reporting program

When the organization hierarchy in image 15 is opened up, you will get a view like Image 16 above. The top level of the hierarchy makes it easy to filter out by hospital if needed, and each sublevel can either be chosen, included, or further opened up for selection of its own sub-levels. The example report in image 14 had two top levels in its selection, Pietarsaari and VSHP. Selecting one or several of the sublevels instead would have further reduced the numbers in each field to only concern the selected organizations. It is also possible to select a higher level organization but leave out a lower level node from the calculations. This is useful if the user for example wants to see the total load of all units that aren't under the mental health ward.

7 Results from the program using the merged data

Once all the data has been imported and integrated into the reporting program, we can start using it, and creating new reports taking advantage of this new data. The code for graphs shown in this paper were implemented by others in the company.

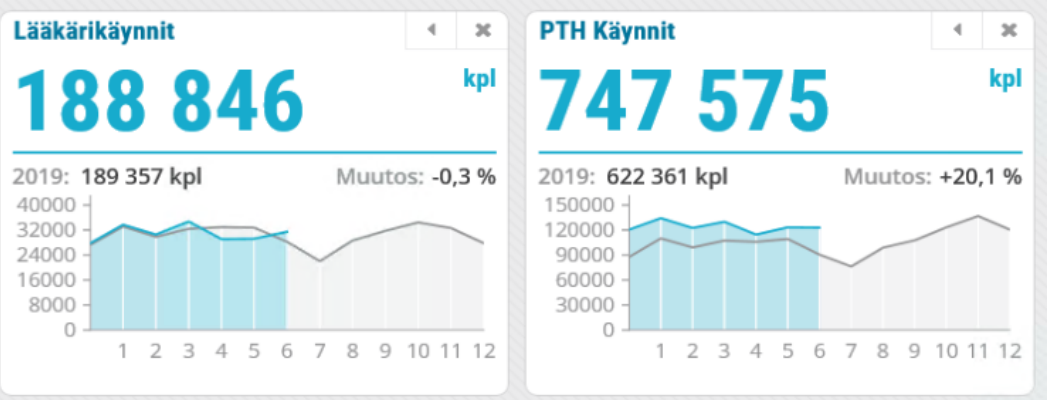

Image 17 One of the reporting pages containing general graphs

Image 17 shows one of the additions made to this program during this project. It shows more general data gotten from this in a variety of easily digested graphs. This simple view can make the hospital directors lives slightly easier when trying to figure out if there are changes in the amount of monthly patients that require changes or more closer inspections. The two graphs seen here can also be clicked to get a more detailed reporting view where we can filter out unwanted information and put on more restrictions for what to show and in what way.

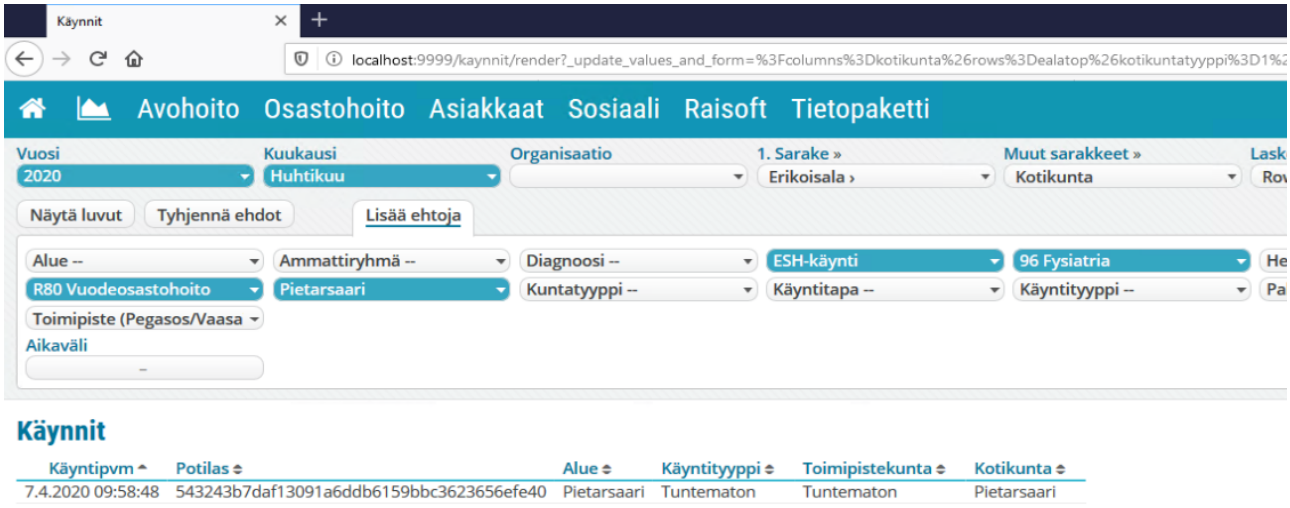

Image 18 Lowest level data possible in this program

Sometimes the user may want to see more more detailed information than what the upper level general reports can provide for a field. Fortunately, it is possible to click a field to drill down into the row level of the report for said field. Image 18 above shows what one row level can look like. It potentially containing more sensitive info also means that there are restrictions on which users has access to this data, only those who the hospitals have deemed suitable have access to it. Image 18 also shows that the number of restrictions placed for the filtering of results can be changed based on reports and user rights. This one for example restricted the data to people whose hometown was Pietarsaari at the moment of going to a hospital. With this information, it's possible to for example find out if there are some trends of people going to another city's hospital for certain procedures even if that procedure is available in a closer hospital, or to find out if certain areas have people visiting the hospital more often with a particular diagnose.

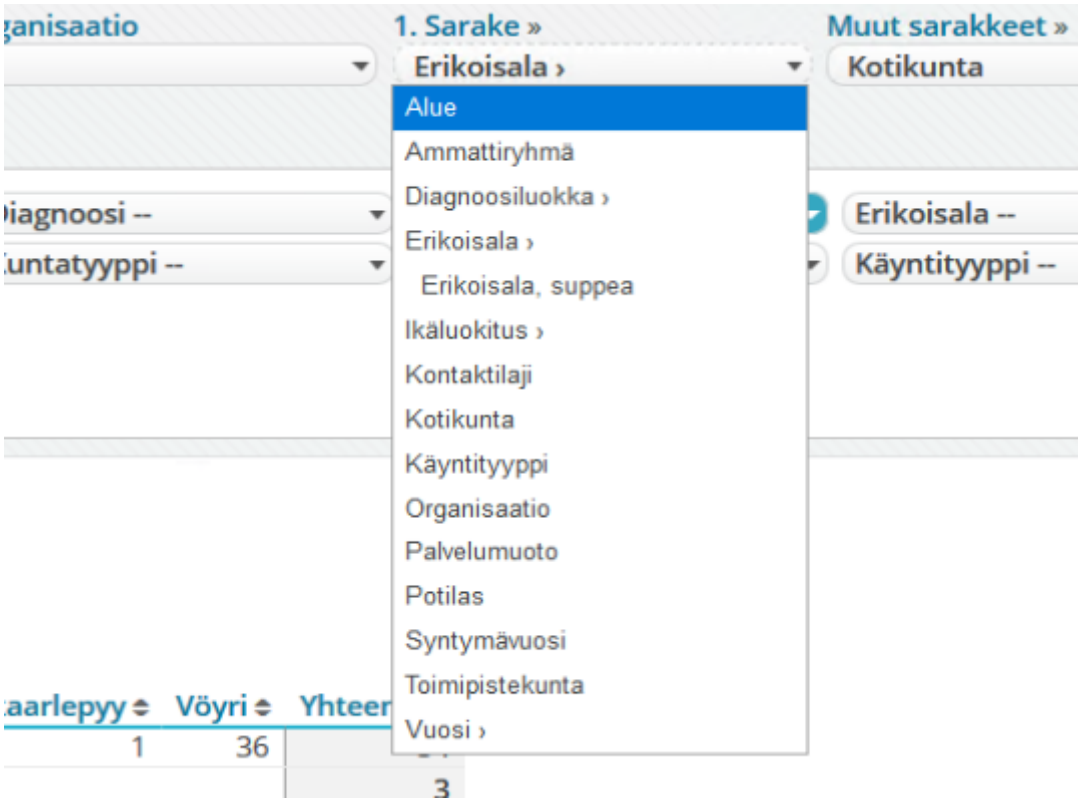

Image 19 Drop down for the first column in a report

Image 19 shows another part of how versatile the reports can be, this time focusing on the main option that configures what the rows will be split based on. For example, choosing the second option from the top, “ammattiryhmä” (meaning general doctor, nurse, surgeon, neurosurgeon, etc.) would split the visit data into rows based on which profession the person had that took care of the patient during the visit. Choosing “Syntymävuosi”, meaning birth year would instead split up the rows into one per birth year of the visiting patients. With this, its possible to easily spot patterns for this top level data, and if anomalies or interesting data has been found, then we can further restrict the data or drill down to get more information of the concerning data rows.

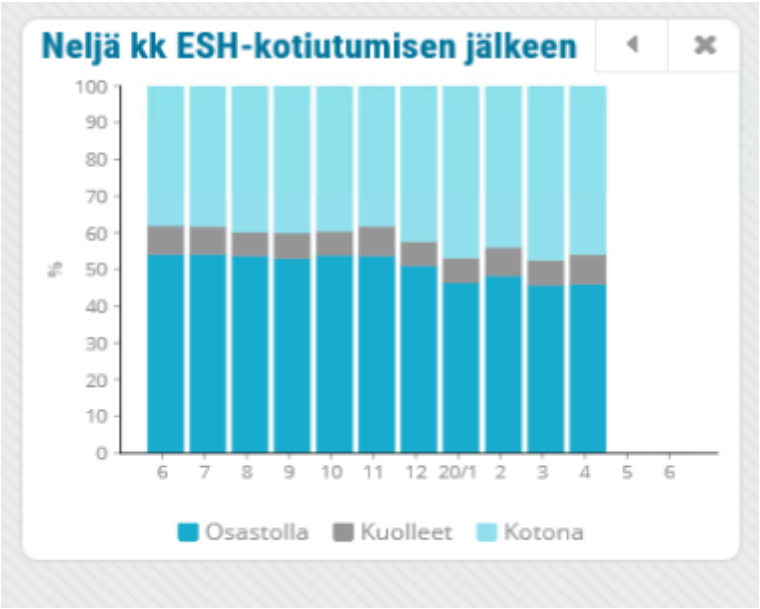

Image 20 Graph showing the long term success of treatments

One data point that hospitals usually are very interested in is how successful the treatments are long term. Image 20 simplifies one aspect of this into an easily digested graph. It shows the state of a subset of people 4 months after their ward period has ended. The x axis is split into months, and the y axis shows the percentage compared to total. Light blue means they are at home, grey means they have died, and dark blue means they are back in a hospital for one reason or another. This provides hospital managers with clear datapoints of things they can strive to improve on, as well as making anomalies easy to spot

Image 21 Drop down showing all kinds of implemented reports

To get a more generalized view for the kinds of data that can now be analyzed, Image 21 shows five different subcategories of Outpatient care (avohoito). Outpatient care is divided into 5 categories. The subcategories in question are specialized health care (ESH), primary health care (PTH), oral health care (Suun terveydenhuolto), Referrals (Lähetteet), and Client plans (palvelusuunnitelmat). All of these categories link to their own reports, and if a certain modification of the report is often wanted, then it can be added as a subcategory under the report link. For example the specialized health care has two sublinks to choose from, Visits per specialty (Käynnit erikoisaloittain), and visits per municipality (Käynnit kunnittain). These subcategories, reports included and even the individual filter options can also be changed based on user clearance or user id, leading to a highly customized experience tailored for each person using the reporting program.

8 Conclusions

The wanted data from all participating hospitals was successfully gathered, converted, sent to a shared server, and ultimately merged into a shared database. Neotides reporting program Exreport was implemented to use the merged data and whole new ways of analyzing patient data became possible.

The two most important results from this, is that it became possible to analyze patient flows between hospitals, and that it became possible to analyze data from the participating hospitals as a whole instead of being restricted to a single hospital. Neither of these things were possible before this project. This has implications both on the short term and long term. In the short term, the people using this data for analysis can directly see their strengths and weaknesses compared to other hospitals when it comes to time spent per patient, readmittance rates, patients per nurse, and other key metrics that all hospitals want to monitor. The metrics for long term quality of care also became more accurate as a result of this work, since it became possible to check patient visits to other hospitals after a certain, instead of just knowing this data for people that returned to the same hospital with diagnoses related to earlier procedures.

These among other things make it possible for the hospital management to change their long term plans as a basis of this new data to improve their quality of care, time efficiency or other aspects that they deem worthwhile to improve.

As a result of this work, it also became easier to spot anomalies that would previously have gone undetected, for example if certain patients have a lot more visits than normal, but just spread out over different hospitals. Another example of anomalies that can now be detected would be if certain departments have a much greater expenditure compared to similar departments in other hospitals without having any better other statistics, or if some specific diagnoses are much more prevalent in one specific hospital.

There are still things that can be improved related to this project. The scope could be increased in many ways to get even more valuable information. More hospitals could participate to make the data in the scope even more valuable for analysis, more tables

could be included to analyze more aspects, and more data columns can be added to increase the number of things that can be analyzed to the data already gathered.

The participating hospitals also quickly realized the worth of increasing the scope from its original level and included some more data points to add to the project after a while. For example, the Raisoft data package was later implemented by coworkers at Neotide while I continued work on my part. But there are still parts of the scope that could potentially be worthwhile to expand on later.

But there is one glaring problem with increasing the scope any more than it already is, and that is the data gathering standards, or rather the lack of them. Each hospital has their own opinions on what data to gather and to what degree of accuracy. For future considerations, some level of standardization in the data gathering practices that all Finnish hospitals (or at least Ostrobothnian) follow would help immensely in reducing the amount of work and time needed for any future collaborations among hospitals. Right now, some hospitals don't even require diagnoses to be written down, or at least don't enforce it.

Some sort of data gathering standardization should also help hospitals in the long term in other aspects. It would for example become less of a hassle to input data sent from other hospitals into their own system during the transfer of a patient, and some manual work could potentially be automated in such cases.

Bibliography

Abrams, C., Kahn, M., Marrs, K., Steib, S. (1994) Unifying Heterogeneous Distributed Clinical Data in a Relational Database, 644-648

Batini, C., Lenzerini, M. (1983) A Methodology for Data Schema Integration in the Entity Relationship Model. IEEE Transactions on Software Engineering SE-10(6), 650-664. DOI:10.1109/TSE.1984.5010294: https://www.researchgate.net/publication/221268631_A_Methodology_for_Data_Schema_Integration_in_the_Entity_Relationship_Model

Batini, C., Lenzerini, M., & Navathe, S. (1986). A comparative analysis of methodologies for database schema integration. ACM Computing Surveys (CSUR), 18(4), 323-364. doi:10.1145/27633.27634

CGI (2013) Pegasos: Ratkaisu terveydenhuollon kokonaisjärjestelmäksi, Product Page: https://www.cgi.com/sites/default/files/files_fi/Brochures_publications/pegasos_tietovarasto.pdf

Cholvy, L., & Moral, S. (2001) Merging databases: Problems and examples. International Journal of Intelligent Systems, 16(10), 1193-1221. doi:10.1002/int.1056

Knuth, D (1997) The art of computer programming, Third edition, 308-348, ISBN 0-201-89683-4

Scully, K., Pates, R, Desper, G., Connors, A., Harrell, F., Pieper, K., Hannan, R., Reynolds, R. (1997). Development of an enterprise-wide clinical data repository: Merging multiple legacy databases. Proceedings : A Conference of the American Medical Informatics Association.AMIA Fall Symposium, , 32-36. ISSN: 1067-5027

Tieto OY (2013), Effica-vuosijulkaisu 2013, Sisältökuvaus, Available online https://dev.hel.fi/paatokset/media/att/43/4355c23d522f0a11632df3794c6af9ada7220200.pdf

Tieto OY (2015) Lifecare Sosiaalipalvelut: Product Page: https://www.tieto.com/fi/toimialat/sosiaali-ja-terveydenhuolto/sosiaalihuolto/lifecare-sosiaalipalvelut/