

Een uitbreidbaar platform voor het efficiënt reconstrueren van fresco's

Nicolas Hillegeer

Thesis voorgedragen tot het
behalen van de graad van Master
in de ingenieurswetenschappen:
computerwetenschappen

Promotor:

Prof. dr. ir. P. Dutré

Assessoren:

Prof. dr. D. De Schreye
Dr. A. Lagae

Begeleider:

Dr. B.J. Brown

© Copyright K.U.Leuven

Zonder voorafgaande schriftelijke toestemming van zowel de promotor als de auteur is overnemen, kopiëren, gebruiken of realiseren van deze uitgave of gedeelten ervan verboden. Voor aanvragen tot of informatie i.v.m. het overnemen en/of gebruik en/of realisatie van gedeelten uit deze publicatie, wend u tot het Departement Computerwetenschappen, Celestijnenlaan 200A bus 2402, B-3001 Heverlee, +32-16-327700 of via e-mail `info@cs.kuleuven.be`.

Voorafgaande schriftelijke toestemming van de promotor is eveneens vereist voor het aanwenden van de in deze masterproef beschreven (originele) methoden, producten, schakelingen en programma's voor industrieel of commercieel nut en voor de inzending van deze publicatie ter deelname aan wetenschappelijke prijzen of wedstrijden.

Voorwoord

Allereerst wil ik mijn promotor Prof. dr. ir. Philip Dutré en begeleider Dr. ir. Benedict Brown bedanken om mij de kans te geven dit interessante project te doen. Dank ook aan Dr. Ares Lagae en Prof. dr. Danny De Schreye voor hun interesse in dit werk en de tijd te nemen die nodig is om het aandachtig te doorlezen.

Het zou een doodzonde zijn om niet te benadrukken wat een goede begeleider Benedict is geweest, hij was er altijd als ik hem nodig had om mij te voorzien van nieuwe invalshoeken en manieren van werken. Zonder zijn diepe kennis van het grotere project waarin deze thesis zich bevindt zou deze al bij voorbaat mislukt zijn. Hij bleef altijd vriendelijk ook al deed ik eigenzinnig of lostte ik de verwachtingen niet in. Benedict is een steengoede onderzoeker en een nog beter mens. Als deze thesis niet aan de hoge standaarden van de K.U.Leuven voldoet, kan ik met een gerust hart zeggen dat het niet aan hem zal gelegen hebben.

Dit project vergde veel inspanningen, niet alleen van mezelf, maar ook van anderen. Ik wil dan ook graag mijn ouders, vrienden en kennissen bedanken voor hun niet aflatende steun en inzet. Ook de bijdrage van de K.U.Leuven kan niet onderschat worden, de meer dan bekwame docenten hebben mijn passie voor het vak niet weten te doven maar integendeel aan te wakkeren.

Als laatste maar zeker niet als minste wil ik mijn grote dankbaarheid aan mijn vriendin Barbara onderstrepen. Zij heeft het niet alleen uitgehouden om mij zeer weinig te zien de laatste maanden maar had ook nog eens het geduld om mijn werk te controleren op de fouten die ik blijkbaar zo graag maak. Het is geen gemakkelijke periode geweest en ik weet niet hoe ik het ooit zal kunnen terugbetalen, maar ik ben blij dat er iets voor te tonen valt.

Nicolas Hillegeer

Inhoudsopgave

Voorwoord	i
Samenvatting	iv
Lijst van figuren en tabellen	v
1 Inleiding	1
1.1 Overzicht	2
2 Overzicht van het thera project	3
2.1 De opdelingen van het thera project	3
2.2 Reconstructie, de bestaande oplossingen	4
3 Doelen & Motivatie	9
3.1 Aspecten waarop de thesis tracht te verbeteren	10
4 Ontwerp van het project	15
4.1 De grote lijnen	15
4.2 Modellen	17
4.3 Modulaire Visualisaties	19
5 Bespreking van het database ontwerp	23
5.1 Ontleding van de data	23
5.2 Het oude systeem: XML-bestanden	24
5.3 Alternatieven	27
5.4 SQL of NoSQL	27
5.5 Implementatie	32
5.6 Data mining	41
5.7 Benchmarks	41
6 Synchronisatie	43
6.1 Opzet	43
7 Modules	49
7.1 MatchTileView	49
7.2 Proof of concept: GraphView	49
8 Toekomstig werk	51
9 Besluit	53
A Extra afbeeldingen & schema's	61

A.1 Omzetting SQL naar NoSQL	61
A.2 Overzicht van de applicatie	62
A.3 Detail van de synchronisatiestructuur	63
B Poster	65
Bibliografie	67

Samenvatting

Het reconstrueren van archeologische fresco's is een domein waar computers nuttig werk kunnen verrichten, zowel voor archiveringsdoeleinden als voor digitale reconstructie. De algoritmen voor het automatisch ontdekken van passende fragmenten zijn reeds erg geavanceerd, maar alle resulterende paarvoorstellen moeten nog steeds verwerkt worden. Huidige tools zijn krachtig maar missen integratie, gebruiksvriendelijkheid en uitbreidbaarheid.

Dit eindwerk tracht hierin verandering te brengen door het ontwikkelen van een platform dat doorgedreven analyse en manipulatie van voorstellen toelaat.

Het resultaat is een applicatie die toelaat om verschillende personen vanop een afstand aan eenzelfde collectie te laten werken. Netwerktogankelijk databaseer zorgt ervoor dat de meest actuele informatie steeds beschikbaar is en synchronisatie dat er geen waardevolle vondsten verloren gaan.

Tegenover het vroegere systeem is de sterk uitgebreide analysecapabiliteit ook nuttig. Complexe vragen over de collectie kunnen hiermee gesteld worden.

Als laatste is het mogelijk om snel modules te maken die nieuwe visualisaties toelaten en ze eenvoudig aan te koppelen op het datasysteem. Enkele voorbeelden werden reeds gemaakt voor dit eindwerk.

Lijst van figuren en tabellen

Lijst van figuren

1.1	Een bak vol fragmenten die rond dezelfde locatie zijn gevonden, sommige zijn reeds aan elkaar gezet.	1
2.1	Een voorbeeld Griphos tafelblad, 1 voorgesteld paar en 2 aparte fragmenten zijn reeds ingeladen	4
2.2	Griphos kan gebruikt worden om de posities van fragmenten in hun opslagplaats te onthouden en vervolgens snel terug te vinden. Het kan ook een foto van de bak waarin ze liggen onder het virtuele tafelblad projecteren. (De brokstukken die buiten de bak staan weergegeven zijn verplaatst geweest naar een andere bak)	5
2.3	Browsematches in werking, de balken boven de paren duiden een validatie door de gebruiker aan. Rood betekent bijvoorbeeld “dit voorstel is zeker niet juist”	6
3.1	Het huidige proces met de aanvullingen van de thesis in het blauw	10
4.1	Het abstracte model van de applicatie, links staat de <i>View/Controller</i> en rechts het <i>Model</i> . De controller stuurt een verzoek naar het model voor een bepaalde (sub)set van de data — al dan niet gesorteerd — en het model antwoord met alle paren die voldoen aan de criteria	16
4.2	Basis sorteer- en filteroperaties worden via de gebruikersinterface blootgesteld	19
4.3	De manier van weergeven uit Browsematches werd gekopieerd naar het nieuwe platform, met uitbreidingen	20
4.4	Een vereenvoudigde kijk op de componenten van de visualisatielaag, het hoofdscherm en het model. De componenten in het wit behoren tot de rest van het thera project en zijn niet gemaakt als deel van dit thesisproject. . .	21
5.1	Overzicht van het database systeem, links (groen) is een mogelijke uitbreiding, het middelste deel (oranje) stelt de thesis voor en helemaal rechts (wit) staan de componenten van het thera project.	32
5.2	Een relationele modellering van de data	33
5.3	Duplicaten: het tweede paar is een duplicaat van het eerste, het derde niet. . .	34

5.4	Duplicaten: het tweede paar van figuur 5.3 verwijst nu naar het eerste paar	36
6.1	Een uittreksel van de attributen-fase van het synchronisatieproces. “No Action” duidt aan dat er geen automatische resolutie toegepast is geweest, hierop dubbelklikken geeft een keuzescherf weer (rechts).	44
6.2	Elk type conflict laat een aantal manieren toe om het op te lossen. Deze figuur beeldt een attribuut-conflict voor.	45
6.3	Situaties waarin automatische resolutie gebaseerd op geschiedenis mogelijk is (eerste figuur) en waar contextinformatie nodig is (tweede en derde figuur). Links is hier gelijk aan de meester-database en rechts is de slaaf-database.	47
9.1	De doorsnedes van goede paren tegenover die van slechte paren. Een mens kan met ervaring leren om hieruit reeds veel af te leiden. Yassine Ryad bewees dat een algoritme kan gemaakt worden dat deze informatie in een rangschikking omzet. [1]	56
A.1	Sommige NoSQL systemen — zoals MongoDB — beschikken zoals te zien valt in deze afbeelding over dezelfde analytische kracht als een SQL database. De performantiekarakteristieken zijn echter verschillend. Afbeelding gebruikt met toestemming [2]	61
A.2	Een ruw overzicht van de componenten in het project en hoe het interageert met het reeds bestaande systeem.	62
A.3	Het synchronisatie-subsysteem, de componenten die overeenkomen met de stappen zijn aangeduid.	63

Lijst van tabellen

5.1	Meting van de tijden die elk programma nodig heeft om een collectie paren in te laden	25
-----	---	----

Hoofdstuk 1

Inleiding

Het reconstrueren van fresco's waarvan in opgravingen fragmenten gevonden worden is een moeilijke taak. Men kan het vergelijken met het oplossen van een enorme puzzel waarvan de stukken arbitraire vormen hebben, de meesten hun originele kleur zijn verloren en er vele anderen ontbreken. Daarbovenop ondervinden veel fragmenten erosie over de eeuwen heen, waardoor ze niet meer perfect op elkaar passen en confirmatie nog moeilijker wordt.



Figuur 1.1: Een bak vol fragmenten die rond dezelfde locatie zijn gevonden, sommige zijn reeds aan elkaar gezet.

De stukken met een nog zichtbaar geometrisch patroon of van de rand van het fresco zijn in vergelijking met de anderen eenvoudig met elkaar te verbinden. Zij zijn door een ervaren archeoloog zonder hulp in elkaar te passen. De overige fragmenten die minder informatie bevatten zijn echter een nachtmerrie om aan elkaar te puzzelen. Er ontbreekt informatie die een structuur vanop een afstand laat zien, het menselijke

visuele systeem is blijkbaar niet erg geschikt enkel grillige randen te vergelijken en aan elkaar te zetten. Het enige alternatief lijkt om lokaler te zoeken en elk fragment te vergelijken met elk ander fragment. Deze aanpak is natuurlijk niet mogelijk, er zijn miljoenen combinaties van fragmenten mogelijk.

In deze context situeert zich het *thera*¹ project, dat probeert om het werk van de archeoloog gemakkelijker te maken door middel van een software platform [3]. De redenering achter het project is dat een computer de ondankbare taak voorgesteld in de vorige paragraaf kan automatiseren. Dergelijk systeem werd in 2007 aan de *Princeton* universiteit in Amerika geconcipeerd. Sindsdien is er tot op de dag van vandaag door verschillende onderzoekers van over de hele wereld aan gewerkt. De werking van het systeem wordt in het volgende hoofdstuk nader toegelicht.

Deze thesis draait rond het maken van een uitbreiding op het platform. De uitbreiding moet de gebruikers van het systeem in staat stellen om de beschikbare data op nieuwe manieren te gebruiken, te visualiseren, aan te passen en te delen met medeonderzoekers. Aangaande terminologie zullen een paar woorden veelvuldig terugkomen: de termen fragment, brokstuk of gewoonweg stuk verwijzen altijd naar een enkel gebroken deel van het originele fresco. De termen fragmentpaar, paar en voorstel zijn gereserveerd voor een aaneenkoppeling van twee fragmenten op een bepaalde plaats. Het valt te benadrukken dat twee fragmenten met elkaar verschillende paren kunnen vormen, indien zij op verschillende punten raken (ook al liggen die punten niet zo ver uit elkaar). Om deze reden wordt een paar ook vaak een voorstel genoemd, om te benadrukken dat het een mogelijke configuratie is, maar geen zekere. Verder verwijst deze tekst meermaals naar (automatische) herkenners, dit zijn de identificatiealgoritmen die twee fragmenten aan elkaar proberen passen.

1.1 Overzicht

Hoofdstuk 2 geeft een overzicht van het bestaande werk en hoofdstuk 3 geeft aan op welke vlakken het project zal uitgebreid worden en waarom. Vervolgens behandelt hoofdstuk 4 het algemene ontwerp van de applicatie. Hoofdstukken 5, 6 en 7 gaan dieper in op enkele specifieke implementaties van het project zoals de database, de synchronisatie en de modules. Tenslotte volgt een besluit in hoofdstuk 9.

¹Thera is de oude naam voor het huidige griekse eiland genaamd Santorini, waar het project voor het eerst in de praktijk werd toegepast.

Hoofdstuk 2

Overzicht van het thera project

Zoals eerder vermeld bouwt deze thesis verder op een reeds bestaand project. Om ze beter te kunnen situeren in het geheel is het handig om eerst even het thera project te bekijken om te kunnen zien waar dit project in past.

2.1 De opdelingen van het thera project

Ruwweg gezien kan het reconstrueren van een fresco met behulp van de computer opgedeeld worden in 3 fasen.

Acquisitie Alle gevonden fragmenten worden ingescand met behulp van 3D en 2D-scanapparatuur om zo een virtueel model van elk stuk te bouwen, zie [3].

Identificatie Vervolgens worden deze virtuele fragmenten aan een zogenaamde *matcher* (automatische herkenner) gegeven, die voor elk fragmentenpaar gaat kijken of ze mogelijk op elkaar passen. Er zijn verschillende types ontwikkeld. Eén ervan (genaamd *RibbonMatcher* [3]) kijkt enkel naar de randen van de brokstukken om te zien of ze in elkaar sluiten. Hierdoor is het in staat om passende fragmenten te ontdekken die voor mensen moeilijk te vinden zijn wegens te weinig distinctieve attributen zoals tekeningen. Het onderzoek gaat verder, in 2010 werd er een nieuw type ontwikkeld dat zijn analyse baseert op een combinatie van verschillende eigenschappen, zoals de sporen die een borstel kan nalaten bij het kleuren van een fresco of zelfs de beoordeling van de eerder genoemde *RibbonMatcher* [4].

Classificatie & Reconstructie De derde en laatste stap bestaat uit het classificeren van wat de vorige stappen produceren. Elk voorgesteld paar moet gecontroleerd worden op validiteit. Verschillende statussen kunnen toegekend worden aan een paar, zoals: *geconfirmeerd*, *misschien*, *nee*, *in conflict met een ander paar*, et cetera. Het algoritme produceert in de regel zeer veel paren van brokstukken, waarvan slechts een klein deel correct kan zijn. [BARBARA] De drempel voor het beslissen van wat een paar kan zijn en wat niet wordt in de vorige stap zo laag ingesteld omdat men wil vermijden dat twee fragmenten

die toch passen genegeerd worden. Anders gezegd, de kost van “*false negatives*” wordt hoog ingeschat. Uiteindelijk zal hieruit een gereconstrueerde fresco moeten ontstaan.

2.2 Reconstructie, de bestaande oplossingen

Het thesisproject besproken in deze tekst tracht de reeds bestaande hulpmiddelen van de reconstructiefase aan te vullen. Hievoor werden reeds programma’s ontwikkeld, namelijk *Griphos* en *Browsematches*. Hierop volgt een bondige bespreking van beide programma’s, wat ze kunnen en niet kunnen, en eventuele voor- en nadelen. Deze factoren hebben een belangrijke rol gespeeld bij het bepalen van de richting van deze thesis.

2.2.1 Griphos

Griphos was de eerste applicatie gericht op het weergeven en beoordelen van de resultaten van de voorgaande stappen. Het werd ontworpen met het doel een centraal zenuwstelsel te zijn voor alle informatie en met deze uiteindelijk een (zo goed mogelijk) gereconstrueerd fresco te maken. Het is mogelijk om zowel aparte fragmenten als automatisch herkende fragmentparen op een virtueel tafelblad te plaatsen en te manipuleren (zie figuur 2.1). De idee achter een dergelijke voorstelling komt voort uit een rechtstreekse vertaling naar de computer van wat een archeoloog op een werkelijk tafelblad doet.



Figuur 2.1: Een voorbeeld Griphos tafelblad, 1 voorgesteld paar en 2 aparte fragmenten zijn reeds ingeladen

Griphos wordt echter geplaagd door een paar problemen: het is traag en biedt geen goede manier om de talloze gegenereerde paarvoorstellen na te kijken. De gebruikte metafoor van het virtuele tafelblad waarin gepuzzeld kan worden stelt een belangrijk deel van het reconstructieproces voor, maar schiet tekort als men de resultaten van automatische fragmentpaarontdekking op vloeiende wijze in het proces wil betrekken. Het probleem lijkt te zijn dat er te veel informatie is en Griphos geen goede manier biedt om door de bomen het bos te zien. De aanzienlijke traagheid van sommige delen van het programma komen vooral voort uit de manier van dataopslag voor paren (XML bestanden) en de complexiteit van de visualisatie (afbeeldingen van hoge kwaliteit en/of gedetailleerde 3D-modellen). Dit zorgt soms voor een onaangename werkervaring, zelfs indien men er toch in slaagt de juiste fragmentparen te lokaliseren. Desalniettemin is het een krachtig programma dat veel functionaliteit biedt voor de detailinspectie van brokstukken.

Het heeft zijn nut al op verschillende vlakken bewezen, behalve detailinspectie is het bijvoorbeeld ook in staat om de posities van fragmenten in een bak te onthouden. Dit betekent een grote snelheidswinst wanneer men bijvoorbeeld denkt een goed paar te hebben gevonden en men wil dit met echte fragmenten verifiëren. Als beide fragmenten in dezelfde bak liggen kan het correcte tafelblad ingeladen worden, Griphos kan vervolgens de gezochte fragmenten laten oplichten. Dit is veel efficiënter dan fragmenten opsporen door vormen te vergelijken, zeker omdat de brokstukken vaak moeilijk te onderscheiden zijn zoals te zien valt in figuur 2.2.

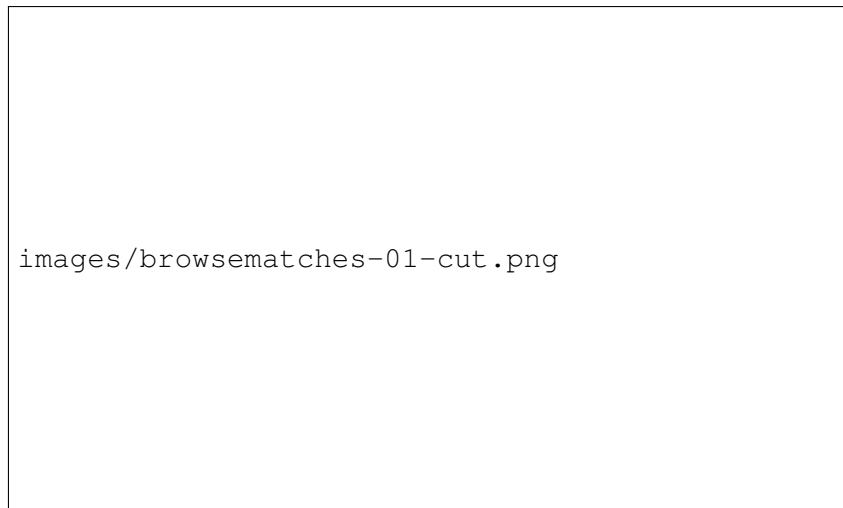


Figuur 2.2: Griphos kan gebruikt worden om de posities van fragmenten in hun opslagplaats te onthouden en vervolgens snel terug te vinden. Het kan ook een foto van de bak waarin ze liggen onder het virtuele tafelblad projecteren. (De brokstukken die buiten de bak staan weergegeven zijn verplaatst geweest naar een andere bak)

// te technisch voor dit hoofdstuk, verplaats naar iets later: [TODO: remove] Het eerste probleem wordt vooral veroorzaakt door trage datatoegang en te complexe visualisaties. Het programma slaagt alles betreffende paren op in een (enorm) XML bestand. Dit is nog doenbaar als men slechts een paar duizend voorstellen heeft maar ondervindt reeds snel onacceptabele vertragingen eens men er meer probeert in te laden. Ter voorbeeld, een kleine voorstellenverzameling van een bepaalde opgraving heeft er reeds 50000. Gekoppeld hieraan laadt Griphos ook steeds een hoge-kwaliteits afbeelding of zelfs een volledig 3D-model in voor elk paar. De combinatie van het gebruik van grote XML-bestanden om informatie over de paren in op te slaan, en het steeds inladen van hoge kwaliteits-afbeeldingen en 3D-modellen. om alle info in op te slaan. Dit is bijzonder inefficiënt en biedt ook niet voldoende mogelijkheden tot uitbreiding. Een ander deel is het gebrek aan zoekmogelijkheden binnen de voorstellen.

2.2.2 Browsmatches

Een eerste prototype om de in Griphos ontbrekende delen aan te vullen, werd Browsmatches genoemd. Het gebruikt eerst de visualisatiecapabiliteiten van Griphos om kleine afbeeldingen te nemen van elk bestaand paar met aan de linkerkant de doorsnede van hun raakvlak. Vervolgens toont het een scherm vol paren en kan men met de pijltoetsen navigeren tussen schermen.



Figuur 2.3: Browsmatches in werking, de balken boven de paren duiden een validatie door de gebruiker aan. Rood betekent bijvoorbeeld “dit voorstel is zeker niet juist”

Dit kleine programma groeide uit noodzaak: het valideren van paarvoorstellen is zo omslachtig met Griphos dat Browsmatches in korte tijd werd gemaakt om het proces te stroomlijnen. Bij het begin van de thesis, om kennis te maken met het ther

project, werd een verbinding gemaakt met Griphos zodat voorstellen die interessant waren in detail bestudeerd konden worden.

Browsematches erfde echter ook enkele van de nadelen van Griphos. Het inladen van de data is traag en vergt zelfs voor kleine datasets veel geheugen. Daarbij is het moeilijk om de informatie uit te breiden of deze gemakkelijk te combineren met de bevindingen van andere onderzoekers. Daarenboven is Browsematches een prototype en niet bedoeld voor algemeen gebruik. Het simpele maar succesvolle concept diende als inspiratie en basis voor deze thesis.

Hoofdstuk 3

Doelen & Motivatie

Het hoofddoel van het thera project is het reconstrueren van fresco's uit de oudheid zo gemakkelijk mogelijk te maken. Er moet een manier gevonden worden om een waardevolle contributie te maken aan het thera-ecosysteem zodat het zijn doel beter kan vervullen. Dit is onder andere mogelijk door delen die ontbreken aan de vorige oplossingen aan te maken of bestaande componenten veelzijdiger te maken.

Gezien de huidige stand van zaken besproken in hoofdstuk 2, valt het op dat er nog behoorlijk wat dingen kunnen toegevoegd worden op het gebied van informatie en het visualiseren ervan. De ongelooflijke hoeveelheid data die het thera project reeds heeft geproduceerd en blijft produceren kan op een betere manier behandeld worden zodat het ware potentieel ervan naar de oppervlakte komt. De laatste stap in het sterk geautomatiseerde virtuele reconstructieproces wordt gekenmerkt door een nood aan interactie met de mens die elk soort hulpmiddel moet krijgen om zo correct en snel mogelijke beslissingen te maken. Natuurlijk gaat er niets boven de feitelijke fragmenten fysisch vastnemen en ze aan elkaar te proberen zetten. Maar gezien dit een zeer tijdsroevende bezigheid is, moet dit zo veel mogelijk beperkt worden.

Er werd gesproken over Griphos en diens weinig ontwikkelde ondersteuning voor het behandelen van automatisch voorgestelde fragmentparen, alsook over de eerste poging om dit recht te trekken: Browsermatches. Deze thesis tracht de lijn van Browsermatches verder te zetten, vertrekkende van de goede ideeën en ervaring waaraan het zijn ontstaan te danken heeft. Dit houdt in dat de focus niet meer zozeer ligt op het actief puzzelen en brokstukken op een tafelblad plaatsen, maar eerder op het beoordelen van de omvangrijke verzameling voorstellen, waarvan slechts een zeer klein deel correct kan zijn. De hoop en verwachting is dat dit zeer kleine deel meteen ook een significante hoeveelheid van de werkelijk overblijvende paren voorstelt.

Daarmee valt te benadrukken dat dit project geen vervanging probeert te zijn van de bestaande software, het is eerder bedoeld als een complement. Merk op dat gevalideerde paren niet kunnen conflicteren en dus rechtstreeks in een groep op een tafelblad geplaatst kunnen worden. In de limiet, wanneer alle mogelijke

paren correct geclassificeerd zijn vloeit hieruit op natuurlijk wijze een (zo compleet mogelijk) fresco voort. Daarom valt de implementatie in de thesis te zien als een extra tussenstap, chronologisch vóór het plaatsen van de fragmenten op een tafelblad en na het uitvoeren van de herkenningsalgoritmen. Figuur 3.1 stelt dit visueel voor.



Figuur 3.1: Het huidige proces met de aanvullingen van de thesis in het blauw

Een van de meest fundamentele verschillen tussen de centrale filosofie van Griphos en die van dit project is dus dat terwijl bij Griphos de focus ligt op aparte geplaatste fragmenten, het thesisproject eerder de automatisch gevonden paren behandelt. Het is in Griphos wel degelijk mogelijk om voorstellen van de automatische paarherkenning in te laden maar de naald in de hooiberg vinden is moeilijk.

3.1 Aspecten waarop de thesis tracht te verbeteren

Hieronder staan verschillende deelaspecten waarop dit thesisproject verbeteringen tracht te maken. Bij elk aspect staat beschreven wat de vereisten zijn waar het project aan zal moeten voldoen en eventueel een voorbeeldscenario. Men kan deze aspecten desgewenst onafhankelijk bekijken maar vaak steunen ze op elkaar om werkelijk tot hun recht te komen. Een visualisatiemethode heeft bijvoorbeeld een manier nodig om de data die het wil visualiseren te krijgen. Omgekeerd is een

databaseersysteem niet bijzonder nuttig zonder de mogelijk de data te manipuleren. Om de verbeteringen en ideeën te bundelen en te testen is er ook een applicatie gemaakt die dient om reeds een voorproef te geven van wat er mogelijk is met de ontwikkelde technologie.

3.1.1 Integratie

Zoals eerder vermeld staat dit thesisproject niet alleen maar maakt het deel uit van een groter geheel. Binnen de grenzen van het mogelijke zou er moeten rekening gehouden worden met de integratie van de geschreven code met die van de andere onderzoekers. Dit verhoogt de kans dat het werk aanvaard wordt en ingang vindt in andere subprojecten.

3.1.2 Collaboratie

Ideaal gezien zouden archeologen steeds toegang moeten krijgen tot hun project op eender welk moment vanop eender welke plaats. Dit kan een gedeelde collectie zijn die over het internet beschikbaar wordt gesteld, of een lokale kopie. Een mogelijk scenario hierbij is dat een ervaren archeoloog in de Verenigde Staten gevraagd wordt om zijn opinie te geven over de huidige stand van zaken (reeds geïdentificeerde correcte paren, moeilijke gevallen, ...). Alle aanpassingen en commentaren die hij maakt worden automatisch ingevoegd en centraal beschikbaar gesteld voor de onderzoekers ter plekke. Op termijn moet het bijvoorbeeld zelfs mogelijk worden om amateurs te laten kijken naar de voorstellen en hun beoordeling te gebruiken om de nog na te kijken voorstellen te rangschikken.

Van cruciaal belang is dat alle verzamelde data (zoals de classificatie van voorstellen) op een robuuste manier opgeslagen, gedeeld en geïncorporeerd kan worden. De toegang naar en manipulatie van deze informatie moet efficiënt zijn. De huidige oplossingen zijn hiervoor ontoereikend en traag, zoals besproken in hoofdstuk 2. De mogelijkheid van meerdere en zelfs lokale kopieën impliceert ook de nood om te kunnen synchroniseren¹. Dit is ook nodig omdat er reeds verschillende huidige verzamelingen bestaan die eventueel in het nieuwe systeem geïmporteerd en gecombineerd moeten worden. Dergelijk systeem is ook nuttig voor het maken van (kleinere) pocketversies en in gebieden waar de internetconnectiviteit niet adequaat of onbestaande is. Belangrijk is dat er steeds een manier is om waardevolle data te combineren en aan te vullen zodat niets verloren gaat.

3.1.3 Schalering

De bestaande oplossingen voor het valideren van voorgestelde paren werken noodgedwongen met een sterk gereduceerde verzameling. Eén van de redenen hiervoor is het gebruik van een groot XML bestand om alles in op te slaan. Deze techniek

¹Naar het model van de zogenaamde *Distributed Version Control System (DCVS)* systemen zoals Git, Mercurial, ...

werkt goed voor bijvoorbeeld het opslaan van tafelbladen — waar er bijvoorbeeld 50 fragmenten en hun locaties moeten opgeslagen worden — maar schiet tekort voor paarvoorstellen. Een snelle rekensom geeft de reden aan: een laag aantal fragmenten voor een specifieke opgraving is bijvoorbeeld 1500. De meeste paarherkenners gaan alle brokstukken een keer met elk andere brokstuk vergelijken en zodoende het meest waarschijnlijke aankoppelingspunt vinden. Het is zelfs mogelijk dat een fragment meerdere keren een plausibel paar vormt met eenzelfde stuk. Naar onder afgerond komen uit deze stap reeds 2 miljoen configuraties gerold, de ene wat waarschijnlijker dan de andere. Dit nummer stijgt kwadratisch in het aantal fragmenten en zelfs met hogere drempels om volgens een bepaalde maatstaf paren automatisch te verwerpen stijgt het aantal configuraties snel.

3.1.4 Gebruiksvriendelijkheid

De uiteindelijke gebruikers van de applicatie zijn niet de ontwikkelaars van het thera project zelf, maar de archeologen. Om deze reden is het belangrijk dat er rekening gehouden wordt met de noodzaak van een visueel aangename en intuïtieve gebruikservaring. Om deze intuïtiviteit te bereiken moet in het programma steeds de aandacht gevestigd blijven op hetgeen het belangrijkste en meest herkenbaar is: de paren en hun fragmenten. Bij voorkeur moet elke operatie gemakkelijk ontdekbaar zijn in de context waar ze kan gebruikt worden, met eventueel een woordje uitleg erbij.

Volgens vele studies op het gebied van gebruikersinterfaces [5, 6, 7] geraken gebruikers gefrustreerd vanaf een operatie een zekere tijd duurt. Deze frustratiedrempel hangt een af van de aard van de operatie (het resultaat), de frequentie waarmee die uitgevoerd wordt, of er visuele tekenen van voortgang zijn en of er tijdens het wachten (steeds) iets anders kan uitgevoerd worden. Om deze reden is het belangrijk dit aspect in acht te nemen bij het ontwikkelen van de applicatie. Dit is vooral zo omdat veel van de acties die mogelijk moeten zijn het potentieel hebben traag te lopen en dus de gebruiker te frustreren. Een algemene vaststelling: de tijd die een operatie mag innemen is omgekeerd evenredig met de frequentie waarmee deze operatie moet uitgevoerd worden. Deze regel in acht nemend is het duidelijk dat bijvoorbeeld het inladen van een scherm vol voorstellen zoals bij Browsematches, het veranderen van een attribuut van een paar, het filteren en sorteren en dergelijke meer acties zijn die met de grootst mogelijke snelheid moeten worden uitgevoerd. Hoe functioneel ook, een programma dat sloom reageert en elke computer op z'n knieën dwingt zal zo weinig mogelijk gebruikt worden [8].

// hoort mss bij [DESIGN] De gebruikersinterface van de oude programma's was goed en kon efficiënt gebruikt worden mits enige training. Maar voor het ondersteunen van collaboratief werken moeten er natuurlijk allerhande nieuwe operaties toegevoegd worden. Tijdens het implementatieproces werd het duidelijk dat de reeds bestaande code van het Browsematches niet uitbreidbaar genoeg was en die van Griphos te complex en belangrijk. Daardoor werd de beslissing genomen om de basisinterface van Browsematches over te nemen maar alle onderliggende code te herschrijven

zodat die uitbreidbaar zou zijn. Bij het opnieuw construeren van dit alles zijn er een aantal verbeteringen gebeurd die niet meteen te maken hebben met het collaboratie aspect maar wel met de workflow van het classificeren. Dit werd gedaan om het hele programma gebruiksvriendelijker en krachtiger te maken in functie van het hoofddoel van het project.

3.1.5 Uitbreidbaarheid

Het samenstellen van werken uit de oudheid is een zeer vakkennis- en ervaringsintensief proces. Hoewel men luistert naar wat de archeologen hierover te vertellen hebben — wat ze graag zouden zien of kunnen doen — zijn er vele zaken die nu nog niet duidelijk zijn maar in de toekomst zeker aan het licht zullen komen. Dit kan bijvoorbeeld zijn omdat de onderzoekers in kwestie niet goed kunnen uitleggen waar ze naar kijken of hoe ze zoeken: na zovele jaren vertrouwen ze op hun moeilijk te definiëren intuïtie. Anderzijds is het vertalen van het proces om fresco's samen te stellen naar de computer nog niet zo vaak geprobeert in het verleden. Dit betekent dat een goede werkwijze in de realiteit misschien niet zonder meer de efficiëntste is als men de transitie naar virtueel reconstrueren maakt (zoals bij Griphos). Daarbovenop zijn er nog vele kansen om innovatieve nieuwe technieken aan te wenden die niet werkbaar zijn als men enkel over fysische fragmenten beschikt.

Het is dus onwaarschijnlijk dat het laatste woord over de ideale metafoor reeds gezegd is waardoor er een grote kans is dat het platform zal moeten vervangen of herbouwd worden als het niet met uitbreidbaarheid in gedachte ontworpen wordt. Er moeten moeiteloos nieuwe delen aan de applicatie en de onderliggende lagen kunnen toegevoegd worden om snel nieuwe ideeën te incorporeren. Dit alles moet best mogelijk zijn zonder de ervaring van gebruikers met oudere versies of andere gebruikersinterfaces te degraderen.

Data

De eerder besproken uitbreidbaarheid die nodig is manifesteert zich op het niveau van de data bijvoorbeeld bijvoorbeeld op deze manier: een onderzoeker vindt een nieuw algoritme om fragmentparen te rangschikken op “goedheid” of men wil informatie over het dikteverschil tussen twee fragmenten opslaan. Idealiter zouden deze zaken als een attribuut bij een paar moeten kunnen toegevoegd worden zodat elke gebruiker er op kan zoeken en sorteren zonder iets extra te hoeven doen.

Enkel data die op geen enkele manier om te vormen valt naar een attribuut van een enkel paar zal een speciale module vereisen om te kunnen gebruiken. Een vereiste is natuurlijk wel dat dit geen effect mag hebben op de delen van het platform die hier geen weet van hebben.

Visualisatie

Er zijn vele visualisatiemanieren denkbaar die elkaar kunnen aanvullen bij het volbrengen van het zoek- en classificatieproces. Zo stelt men zich bij de uitdrukking “fresco’s samenstellen” waarschijnlijk een grote puzzel voor waar men ten alle tijde het overzicht kan behouden en stukken proberen te passen.² Deze puzzel visualisatie is visueel aantrekkelijk en biedt het menselijke patroonherkenningsvermogen³ de mogelijkheid om zich van zijn beste kant te laten zien. Echter, door de grote hoeveelheid aan fragmenten en dus mogelijke paren is het moeilijk om hier aan te beginnen. Maar, hoe meer reeds geconfirmeerde paren er zijn, hoe duidelijker het globale beeld kan worden. Dit staat toe om een overzicht te krijgen van de vooruitgang en gericht te zoeken naar stukken die nog ontbreken. Dit is een voorbeeld van een macro-perspectief.

Een alternatief is te beginnen door te kijken naar waar de computer wél goed in is: fragmenten aan elkaar passen en rangschikken naar kans/overeenkomst/et cetera. Door een gemakkelijk navigeerbare lijst op te stellen van alle voorstellen die de reconstructie algoritmes hebben gedaan, kunnen er snel op elkaar passende fragmenten geïdentificeerd worden. Dit kan men zien als een micro-perspectief of *bottom-up* manier om fresco’s te reconstrueren. Het is ook de aanpak die in Browsermatches gebruikt wordt. Nadat men bijvoorbeeld met deze manier een deel acceptabele paren heeft geïdentificeerd, kunnen deze bijvoorbeeld weergegeven worden als een grote puzzel en dienen zij als beginpunt om verder te puzzelen. Dit maakt het globale beeld veel informatiever: stel dat duidelijke “gaten” ontstaan in een resem goede fragmentparen, dan kan er gericht gezocht worden naar een fragment dat erin past door te zoeken naar een fragment dat met elk van deze insluiters past (indien het gevonden werd bij de opgraving).

Kortom, het is duidelijk dat een visualisatie die in alle gevallen de meest geschikte is niet bestaat. De beschikbare informatie moet soms gewoon op andere manieren worden weergegeven. Om deze reden is het wenselijk om het mogelijk te maken snel nieuwe visualisaties in te bouwen die kunnen communiceren met andere delen van de applicatie en eventueel de informatie manipuleren.

²Het Griphos programma gaat uit van het idee van kleine beheersbare stukken van de reuzenpuzzel (elk tafelblad zou een verzameling kunnen zijn van stukken die gerelateerd waren, bijvoorbeeld door hun vindplaats)

³Iets waar computers de mens nog altijd niet in evenaren. Een ander therapeutisch subproject onderzoekt wel de mogelijkheid om zogenaamde ‘clusters’ te identificeren en te gebruiken voor reconstructie.

Hoofdstuk 4

Ontwerp van het project

Eenmaal de belangrijkste vereisten gekend zijn kan er een ontwerp opgetekend worden. Daarom even de grote lijnen die te concluderen vallen uit hoofdstuk 3:

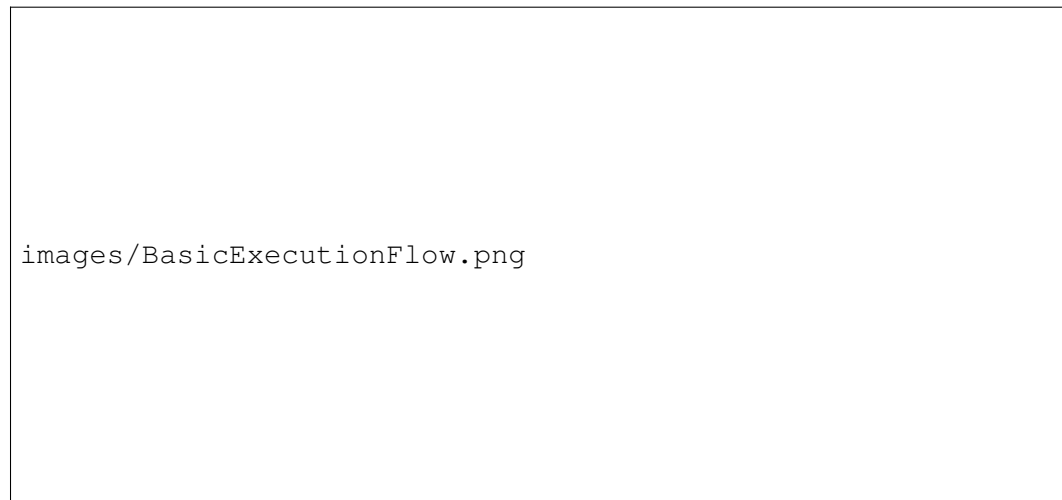
- De data en het visuele aspect van de applicatie moeten zo ontkoppeld mogelijk zijn, elk moet apart uitbreidbaar zijn
- Zeer grote hoeveelheden data moeten vlot kunnen behandeld en genavigeerd worden: zoeken, filteren, sorteren, aanpassen, ...
- De data moet zowel centraal als decentraal toegankelijk zijn en synchronisatie toestaan
- Compatibiliteit met de reeds geschreven onderdelen van het project moet indien mogelijk bewaard blijven
- Om het ontwerp te verifiëren moet een werkend programma gemaakt worden, gebruiksvriendelijkheid, functionaliteit en snelheid zijn hierbij belangrijk

4.1 De grote lijnen

Van alle vereisten die rechtstreeks invloed kunnen uitoefenen op de architectuur, is de splitsing van de data en de visualisatie waarschijnlijk de meest fundamentele. Een beproefde aanpak om dit te realiseren is het ontwerppatroon genaamd *Model-View-Controller (MVC)* [9].

Het doel is een ontwerp te maken waarin elke verschillende visualisatiemethode kan verbonden worden aan een geschikte databron, wat die ook moge zijn. Enkele zaken moeten echter nog vastgelegd worden, namelijk: wat is de basiseenheid van data? Waar zal een visualisatiemodule achter vragen? Zoals in figuur 3.1 te zien is, ligt de focus bij dit project op koppelingen van fragmenten in plaats van fragmenten op zich.

Er zijn op het eerste zicht twee alternatieven om dit te modelleren. De eerste mogelijkheid is een soort van *MatchedFragment* die een fragment beschrijft plus een lijst met alle fragmenten die er potentieel aan gekoppeld kunnen worden en op welke locatie. De tweede mogelijkheid is om elk paar een apart object te laten voorstellen (bvb. genaamd *FragmentPair*). Conceptueel kan dit gezien worden als een graaf, waarop de fragmenten de knopen voorstellen en de paren de zijden. Het eerste alternatief lijkt het voordeel te hebben dat het gemakkelijk is om na te kijken of een fragment reeds “bezet” is, alle mogelijke gekoppelde fragmenten zijn immers meteen beschikbaar. Dit soort opstelling bevat op het eerste zicht veel redundantie, een fragment zal op die manier een verwijzing met attributen naar een fragment bevatten en vice versa. De redundantie vermijden en een verwijzing als een apart object voorstellen waar beide fragmenten naar kunnen verwijzen is eigenlijk niets anders dan de tweede optie (een *FragmentPair*). Op die manier wordt de situatie omgedraaid en kan een fragmentenpaar verwijzen naar de fragmenten die het opbouwen. Daarbij kan het probleem van hoe de bezetting van een brokstuk te weten te komen opgelost worden door de vereiste zoekfunctionaliteit van het datamodel te benutten. Om die reden wordt ervoor gekozen om een fragmentpaar te gebruiken in plaats van een fragment met referenties naar alle buur fragmenten. Eenmaal de basiseenheid van informatie gekozen is, valt de kern van de applicatie volgens MVC uit te beelden als in figuur 4.1.



Figuur 4.1: Het abstracte model van de applicatie, links staat de *View/Controller* en rechts het *Model*. De controller stuurt een verzoek naar het model voor een bepaalde (sub)set van de data — al dan niet gesorteerd — en het model antwoordt met alle paren die voldoen aan de criteria

De gebruikersinterface en de achterliggende bibliotheken werden gemaakt met behulp van de Qt toolkit [10] in C++. Hierdoor draait de applicatie op zowel UNIX als Win-

dows systemen. Zoveel mogelijk zware berekeningen worden in parallel uitgevoerd, waardoor de applicatie snel opstart (< 1 seconde) en responsief blijft, dit komt de gebruikerservaring ten goede.

4.2 Modellen

Een model is een abstracte voorstelling van een databron, het stelt de ontwikkelaar in staat om op een eenvoudige manier naar objecten te vragen en ze aan te passen zonder zich druk te hoeven maken over waar die vandaan komen. Dit is noodzakelijk om de nodige flexibiliteit te kunnen garanderen, zodat bijvoorbeeld een XML-bestand op de thuiscomputer plots kan ingewisseld worden voor een database kilometers verder zonder aan functionaliteit in te boeten.

Met enkele uitzonderingen handelt het type model gemaakt voor dit thesisproject enkel in fragmentparen. Op zijn minst bestaat zo'n paar uit twee fragmenten, een transformatie die beide fragmenten tegen elkaar plaatst en eventueel een resem attributen. De objecten die geproduceerd worden door de identificatiealgoritmen van het thesa project voldoen reeds aan deze twee voorwaarden. Er bestaan al veel componenten die hiervan gebruik maken dus dezelfde interface recycleren bevordert de integratie met het moederproject.

De mogelijke operaties op een model kunnen in twee categorieën ondergebracht worden, opvragingen en aanpassingen. Vooral opvragingen zijn interessant omdat ze de ontwikkelaar en gebruiker in staat stellen hun eigen criteria te stellen over welke paren terugkomen. Een voorbeeld is: geef alle fragmentparen uit bak 33 van collectie WDC13 gerangschikt op dalende volumedoorsnede die als "mogelijk correct" te boek staan en commentaar hebben gekregen. Op die manier kan men snel navigeren naar de gewenste deelverzameling.

4.2.1 Opvragen

Het concept achter het opvragen van fragmentparen is dat er steeds wordt begonnen vanuit de volledige beschikbare verzameling die op het moment in de databron aanwezig is. De standaardoperaties zijn vervolgens **filteren** en **sorteren**.

Sorteren is eenvoudig en kan op eender welk attribuut in stijgende of dalende zin gebeuren. Filters gebruiken een krachtige syntax die volledig gelijk is aan die van de *WHERE*-clausule van een SQL zin¹ (voorbeeld: broncode 4.1). Visualisatiemodules kunnen indien gewenst geavanceerde gebruikers zelf filters laten verzinnen, maar veelgebruikte filter- en sorteeroperaties worden ook als knoppen en invoervelden

¹SQL of Structured Query Language is een declaratieve taal om vragen te stellen aan een database. Er zijn vele verschillende implementaties die allen (bij benadering) dezelfde taal verstaan, men groepeerde deze doorgaans onder de naam SQL databases.

blootgesteld, zoals in figuur 4.2.

De voorwaarde die aan filters gesteld wordt is dat ze enkel attributen bevatten die werkelijk bestaan, filters die naar een onbestaand attribuut refereren worden genegeerd. Er kunnen meerdere filters op eenzelfde model actief zijn, deze werken dan conjunctief. Op die manier kunnen meerdere visualisaties filters plaatsen zonder met elkaar in conflict te komen.

De voornoemde operaties kunnen onafhankelijk van de soort databron gebruikt worden, het model is verantwoordelijk voor een correcte implementatie. Hoewel deze reeds krachtig zijn, zou het zonde zijn moest er functionaliteit van de databron verloren gaan. Stel dat deze een eigen taal heeft (zoals SQL) waarmee aan arbitraire data-analyse kan gedaan worden en de gewenste functionaliteit wordt niet aangeboden via het model. In dat geval is er de mogelijkheid om, op eigen gevaar, zelf een verzoek voor te stellen. Indien het model beslist dat het type verzoek overeenkomt met de databron, zal die het verzoek versturen en proberen fragmentparen te maken van het resultaat (indien het verzoek een opvraging was).

Broncode 4.1: Een voorbeeld van hoe een model kan gebruikt worden in een applicatie

```
IMatchModel *model = ...; // een model

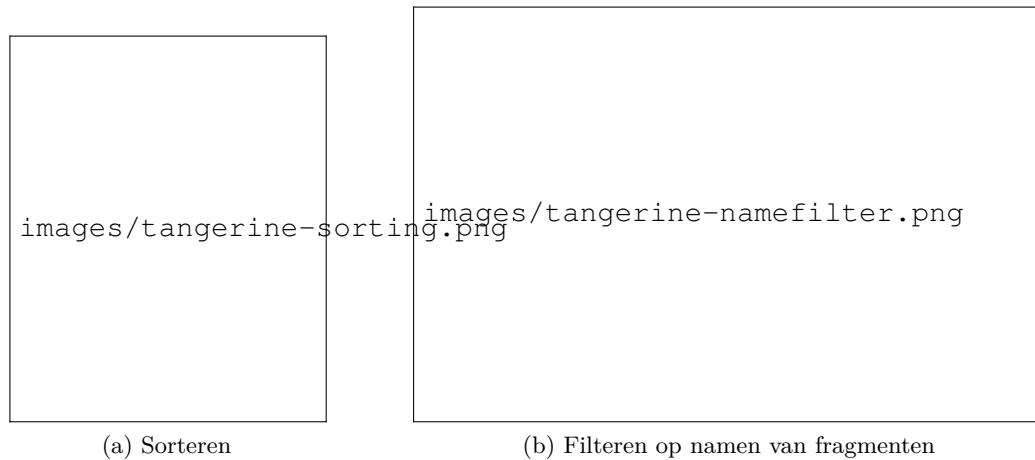
model->sort("error", Qt::AscendingOrder); // alles moet gesorteerd worden op het "error"
    attribuut, van laag naar hoog
model->filter(
    "duplicates_filter",
    "duplicate = 0"
); // eis dat het "duplicaat"-attribuut van een paar gelijk is aan 0, wat betekent dat het geen
    duplicaten heeft
model->filter(
    "probability_filter",
    "probability > 0.8 OR volume < old_volume"
); // de probabiliteit moet groter zijn dan 0.8 of het volume moet kleiner zijn dan het oude
    volume
model->filter(
    "my_new_status_filter",
    "status IN (1,2)"
); // het paar moet reeds "goedgekeurd" of "waarschijnlijk goed" zijn

...

int max = model->size(); // de hoeveelheid paren die aan de filters voldoen

for (int i = 0; i < model->size(); ++i) {
    IFragmentPair &pair = model->get(i);

    // doe iets met het paar
}
```



Figuur 4.2: Basis sorteer- en filteroperaties worden via de gebruikersinterface blootgesteld

4.2.2 Aanpassingen

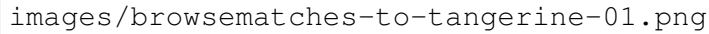
Veranderingen aan de inhoud van de database gebeuren omwille van compatibiliteitsoverwegingen via twee manieren. Globale aanpassingen zoals “deze groep paren zijn duplicaten van dit paar” gaan via het model en lokale aanpassingen zoals enkele attributen via het paar zelf. Hoofdstuk 5 over het ontwerp van de database licht dit verder toe.

4.3 Modulaire Visualisaties

Het laatste stuk van de puzzel is de manier waarop de gebruiker met het systeem interageert: de gebruikersomgeving. Deze bestaat uit een schil met daarop een aantal modules die verschillende perspectieven geven op de aanwezige data (meestal onder de vorm van paren). De applicatie (codenaam “Tangerine”) maakt bij het opstarten de schil aan die de connectie maakt met de databeheerlaag en in staat is verschillende visualisatieplugins te laden (zie figuur 4.4). Dankzij deze flexibiliteit kunnen er in de toekomst met weinig moeite nieuwigheden worden toegevoegd, des te meer omdat visualisatieprototypes zich niet moeten drukmaken om de toevoer van data.

De eerste en meest uitgewerkte van de modules werd *MatchTileView* gedoopt, omdat het de paren (*matches*) als tegels laat zien. Dit is dezelfde manier als degene die zo succesvol in het Browsematches prototype werd gebruikt, maar uitgebreid qua mogelijkheden. Hoofdstuk 7 bespreekt de toegevoegde functionaliteiten en andere modules.

Elke module krijgt van de applicatie een model toegewezen waar het fragmentparen



```
images/browsematches-to-tangerine-01.png
```

Figuur 4.3: De manier van weergeven uit Browsematches werd gekopieerd naar het nieuwe platform, met uitbreidingen

uit kan opvragen. Dit kan een nieuw model (zonder filters noch sortering) of een gedeeld model zijn. Een gedeeld model behoort niet exclusief tot een module en betekent bijvoorbeeld dat als er één beslist om te sorteren op een attribuut zoals “het verschil van de dikte tussen twee fragmenten”, plots alle modules die gebruikmaken van ditzelfde model over een gesorteerde dataset beschikken. Via speciale signalen worden zij hiervan op de hoogte gebracht, zodat ze kunnen beslissen of het nodig is een actie te ondernemen. Dit kan handig zijn voor pure visualisatieplugins die geen zoekmogelijkheden aan de gebruiker blootstellen, het kan dan vertrouwen op andere modules om data aan te leveren.

Indirecte communicatie via het model is (voorlopig) de enige manier waarop modules elkaar kunnen beïnvloeden. De structuur van de componenten ziet er uit als in figuur 4.4. Merk op dat er een plugin is (*DetailView*) die geen gebruik maakt van fragmentparen maar eerder van een virtueel tafelblad net als Griphos. Zoals eerder aangehaald kunnen fragmentparen automatisch op een tafelblad gezet worden. Dit tafelblad kan dan in 3D weergegeven worden door *DetailView*, waarover meer in hoofdstuk 7.



images/VisualizationExtract.png

Figuur 4.4: Een vereenvoudigde kijk op de componenten van de visualisatielaag, het hoofdscherm en het model. De componenten in het wit behoren tot de rest van het thera project en zijn niet gemaakt als deel van dit thesisproject.

Hoofdstuk 5

Bespreking van het database ontwerp

De visuele delen van het platform bekommeren zich met de **de manier waarop** er data moet weergegeven worden, de gebruiker kiest **wat** er moet weergegeven worden. Om de data hiervoor aan te voeren is er een data laag nodig die elk verzoek juist en efficiënt kan verzorgen.

De hoofdpunten waaraan deze laag moet voldoen werden reeds opgesomd in hoofdstuk 3, namelijk: snelheid, universele toegang, synchroniseerbaarheid en maximale flexibiliteit voor data-analyse. Eerst wordt een analyse gemaakt van de structuur van de data. Vervolgens wordt er gekeken naar de huidige methode die het thera project hiervoor hanteert. Enkele alternatieven worden besproken en op basis daarvan wordt een keuze gemaakt.

Alle tijdmetingen werden opgenomen met hetzelfde systeem, een AMD X2 5600 (dual core) met 2GB RAM en een 7200 RPM harde schijf. Als besturingssysteem werd Windows 7 64-bit gebruikt.

5.1 Ontleding van de data

5.1.1 Aanwezige data

De kern van de data is een verzameling fragmentparen, deze bezitten elk een paar onmisbare eigenschappen en vele optionele attributen. De basiseigenschappen beperken zich tot de namen (of een identificatienummer) van de fragmenten en een transformatiematrix die het éne brokstuk aan het andere koppelt in de ruimte. Er is geen enkel paar dat deze gegevens mist. Verder kunnen de herkenningsalgoritmen en eventuele *post-processing*-algoritmen allerlei soorten data toevoegen als attributen. Dit kan gaan om een maatstaf afhankelijk van het pasproces zoals de zogenaamde *RibbonMatcherError* [3] of een algemene eigenschap zoals het overlappende volume van twee brokstukken. Verder kan een gebruiker eigen data toevoegen, zoals

een validatie, commentaar bij een voorstel, et cetera.

5.1.2 Extra data

De data uit het vorige stuk is reeds terug te vinden in het thera project, maar om collaboratie toe te laten moeten een paar zaken zoals gebruikers en geschiedenis toegevoegd worden.

Gebruikers

Behalve paren en attributen is het in een collaboratieve omgeving handig om gebruikers bij te houden. Omdat het systeem in eerste instantie voor selecte groepen is, is een gebruikersnaam en een mailadres voldoende. Op die manier zou men kunnen vastleggen wie welk stukje data heeft toegevoegd of veranderd. Dit is nuttig om de oorzaak van problemen op te sporen als die zich voordoen en te beslissen wiens data voorrang krijgt als eenzelfde fragment door een andere persoon gewijzigd wordt (zie synchronisatie).

Geschiedenis

Synchronisatie is een vereiste, hetgeen detectie en resolutie van conflicten impliceert. Om de resolutie voor een deel te automatiseren is een geschiedenis van de attributen nodig (zie hoofdstuk 6). Dit wil zeggen een lijst die aangeeft op welk moment, door wie en naar welke waarde een attribuut is veranderd. Het bijhouden van een geschiedenis heeft ook andere voordelen, zo kan bijvoorbeeld de geschiedenis van een “commentaar”-attribuut gebruikt worden als een manier om te corresponderen. De overgang van validaties van een paar (bijvoorbeeld *niet geweten* → *misschien* → *niet correct* of *niet geweten* → *misschien* → *correct*) kan een nuttige bron van data zijn voor zowel mensen (statistieken) als computers (lerende algoritmen).

5.2 Het oude systeem: XML-bestanden

De fragmentparen en hun attributen worden door automatische herkenners opgeslagen in een XML bestand zoals in lijst 5.1. Dit formaat is uitermate geschikt voor het overbrengen van de data naar andere subsystemen en is leesbaar door een mens. Het is minder geschikt als een permanent opslag- en zoekformaat. Griphos en Browsesearch gebruiken het XML-formaat ook voor dat laatste, met enkele nadelen tot gevolg. Om een redelijke snelheid te behouden tijdens het zoeken of sorteren moeten ze het bestand volledig in het werkgeheugen plaatsen. Voor een document met 50000 paren neemt dit reeds een goede 300MB in beslag voor paren met elk 11 attributen (zonder afbeeldingen of 3D-modellen). Dit extrapoleren naar een miljoen paren geeft 5GB aan vereist werkgeheugen, hetgeen bij de meeste systemen van vandaag leidt tot het wegschrijven van data naar het bestandssysteem of eerder nog een geheugenallocatiefout.

Broncode 5.1: Uittreksel van een fragmentpaarbestand (hier worden slechts 2 paren getoond)

```
<!DOCTYPE matches-cache>
<matches version="1.0" >
  <match status="0" error="0.187266" old_volume="0.143461" conflict="... 3007 2991 ..."
    num_conflicts="18" tgt="WDC1_0030" overlap="0.401502" volume="1.02974" id="
    2677" src="WDC1_0037" xf="9.96e-01 -8.49e-02 ... 1.00e+00 " />
  <match status="0" error="0.21624" old_volume="0.183338" conflict="... 2710 2666 ..."
    num_conflicts="24" tgt="WDC1_0030" overlap="0.288536" volume="2.51007" id="
    2678" src="WDC1_0037" xf="9.99e-01 -1.62e-02 ... 1.00e+00 " />
</matches>
```

Het inladen van dit XML bestand is tijdrovend. Tabel 5.1 laat zien dat dit minutenlang kan duren en geeft een vooruitkijk op de performantie die het thesiproject haalt. Griphos en Browsmatches gebruiken dezelfde manier om data in te lezen, maar vertonen evenwel een groot verschil in laadtijd. De reden is dat Griphos meteen ook de afbeeldingen van elk fragment ophaalt. Meer nog, na het laden van 50000 paren neemt het Griphos proces 1.5GB RAM in beslag. Het lijkt erop dat dit geheugen niet onmiddellijk vrijgemaakt noch herbruikt wordt, want een tweede keer de lijst met fragmentparen openen zorgde er op het testsysteem met 2GB RAM voor dat Griphos keer op keer werd afgesloten wegens overallocatie. Browsmatches (en de thesapplicatie) laden de afbeeldingen pas wanneer ze eigenlijk nodig zijn en besparen op die manier veel geheugen. Het thesiproject gaat eigenlijk nog een stapje verder en laadt zelfs de fragmenten pas in wanneer ze nodig zijn, dit komt later aan bod. De reden dat de tijd dan niet gewoon 0 seconden aangeeft is omdat er steeds moet geteld worden hoeveel voorstellen er in totaal aanwezig zijn, dit kost bij sommige databaseimplementaties wat tijd indien die net opgestart zijn.

	Griphos	Browsmatches	Thesis
~4000 paren laden	54 sec	16 sec	0 sec
~50000 paren laden	7 min 18 sec	2 min 47 sec	0-3 sec*
~250000 paren laden	niet getest	niet getest	0-15 sec*

Tabel 5.1: Meting van de tijden die elk programma nodig heeft om een collectie paren in te laden

* De tijden voor de thesis variëren afhankelijk van de achterliggende implementatie en hoeveel data er nog in het geheugen geladen is. De maximale tijden kwamen bij tests enkel voor net na het opstarten van de database, de gemiddelde laadtijd is ongeveer 100 milliseconden.

Aangezien een verzameling objecten van dergelijke grootte niet volledig kan weergegeven worden, is het nodig te graven achter de juiste informatie. Het thesaproject biedt

een zelfgemaakt predicaatlogica systeem aan. Het systeem wordt weinig gebruikt, maar een analyse van de code levert de volgende conclusies: er is één variabele en deze stelt steeds een paar voor, ze is altijd universeel gekwantificeerd. De ontwikkelaar kan dan in dit raamwerk een beperkte logische zin opstellen. Predicaten kunnen aaneengeschakeld worden door middel van operatoren. Conjunctie (\wedge), disjunctie (\vee) en exclusie (\oplus) zijn de ondersteunde operatoren. Negatie (\neg), implicatie (\implies) en equivalentie (\iff) ontbreken, alsook de existentiële kwantor (\exists). Vergelijking 5.1 geeft het abstract model van een verzoek, vergelijking 5.2 geeft een voorbeeld voor een mogelijke invulling.

$$\forall p(paar(p) \wedge (\mathbf{zin})) \quad (5.1)$$

$$\forall p(paar(p) \wedge (volume(p, >, 0.78) \vee error(p, \leq, 0.22)) \quad (5.2)$$

De zin van vergelijking 5.2 invoeren zal alle paren weerhouden die aan het eerste of het tweede predicaat voldoen, gelijkaardig aan de *findall*-functor die in vele prolog-implementaties terug te vinden is. Een volledige eerste orde logica is zeer expressief maar die expressiviteit is niet volledig beschikbaar omwille van de genoemde beperkingen. Door het combineren van verschillende van deze zinnen met applicatielogica kan de verloren uitdrukingskracht teruggewonnen worden maar dit komt de efficiëntie noch de uitbreidbaarheid ten goede. In dit systeem proberen uit te drukken dat alle aangrenzende paren van een verzameling voorstellen moet moeten weerhouden worden is moeilijk, zeker als de eis erbij komt dat de aangrenzende paren de originele paren moeten connecteren (clustering). Groeperen of aggregeren is ook niet mogelijk en vereist dus applicatielogica. Kortom, het is niet mogelijk zinnen te maken die over verschillende paren tegelijkertijd gaan.

Het predicaatsysteem biedt dankzij de dichte integratie met de automatische herkenners wel de mogelijkheid aan om een predicaat aan te maken dat eist dat een paar past volgens een specifieke herkenner. Dit paar kan bijvoorbeeld door een mens of een andere herkenner gevormd zijn. Deze flexibiliteit wordt echter in de werkelijkheid niet veel gebruikt, voorlopig enkel als een gebruiker vraagt om twee fragmenten aan elkaar te koppelen. Een gewone methode-oproep zou hier echter ook volstaan. Ondanks het feit dat er duidelijk tijd en denkwerk in deze manier van werken is gestoken, is het niet echt optimaal. Het is niet erg handig om mee te werken, mist expressiviteit en is traag. De eerste opmerking is niet zo gemakkelijk objectief aan te tonen, maar het schaarse gebruik binnen de bestaande componenten is reeds een aanwijzing. Browsermatches vraagt bijvoorbeeld gewoon alle paren op en filtert en sorteert de lijst zelf door er telkens volledig over te itereren. Voor de performantie te meten zijn spijtig genoeg geen uitgebreide vergelijkende tests uitgevoerd. De simpelste soort zoektochten, namelijk een conditie op een enkel numeriek attribuut, nam echter stevast > 3000 milliseconden in beslag na een paar herhaalde zoektochten op een database van 4000 paren. Verder kan dit systeem enkel overweg met paren, een geschiedenislijst of andere zaken doorzoeken gaat niet. Omdat een zin in deze predicaatlogica niet in pure tekst kan uitgedrukt worden is

de uitbreidbaarheid naar de gebruikerskant toe eerder beperkt.

Zelfs indien de implementatie van de predicaatlogica zou gegeneraliseerd worden – wat geen eenvoudige taak is – zou er nog veel werk nodig zijn om het te laten werken op grote collecties. Op dit moment maakt het systeem geen gebruik van versnellingsstructuren en is het dus geforceerd om steeds de volledige verzameling door te lopen (voor elk predicaat).

5.3 Alternatieven

In deze sectie worden enkele alternatieven op het huidige systeem besproken. Er zijn reeds vele complexe stukken software geschreven die dezelfde of betere capabiliteiten hebben als het huidige systeem, en de minpunten ervan niet vertonen.

De huidige combinatie van XML-bestanden en predicaatlogica is niet houdbaar omdat het onder andere steeds de volledige collectie voorstellen in het geheugen nodig heeft. Afgezien hiervan zou het verloren werk zijn een netwerkprotocol uit te vinden om collaboratief te werken aan eenzelfde database: dit soort zaken bestaat reeds en die implementaties zijn efficiënt en goed getest. Daarom kan tenminste het zoeken op predicaatlogica niet behouden worden.

Commerciële oplossingen komen niet in aanmerking omdat dit financieel niet haalbaar is voor de vaak ondergefinancierde archeologieprojecten, ze zouden ook moeilijk te verkrijgen zijn om te testen voor dit thesisproject. Het is puur toeval maar ook een voordeel dat alle vergeleken systemen ook vrije software zijn, hun broncode is te verkrijgen en aan te passen indien gewenst.

XML-bestanden kunnen als een conventionele database gebruikt worden, systemen als Sedna en eXist zijn gebaseerd op XML-bestanden maar zijn niet bedoeld voor grote verzamelingen [11]. Verder kijken dan XML-oplossingen levert een grote variëteit aan oplossingen, waarvan de populairsten in het vakjargon onderverdeeld worden in SQL- en NoSQL-databases. Een oppervlakkige studie van de mogelijkheden van beide systemen wijzen uit dat ze voldoende krachtig zijn om aan alle vereisten te voldoen. Om een optimale keuze te maken is een iets diepere studie nodig. De volgende sectie beschrijft de verschillen tussen beide aanpakken en probeert uit te zoeken welke de meest geschikte is.

5.4 SQL of NoSQL

Deze sectie bespreekt de cruciale verschillen en gelijkenissen tussen de SQL en NoSQL aanpakken en hoe deze van toepassing zijn op dit project. Omdat het NoSQL domein zo uitgebreid is werden verschillende implementaties nader bekeken, deze zijn: Berkeley DB, Tokyo Cabinet, CouchDB, MongoDB, Cassandra en Neo4j

5.4.1 Datamodel

SQL is een afkorting voor *Structured Query Language* en staat voor zowel de taal die gebruikt wordt om met de database te communiceren als de databases zelf. De taal en de structuur van dit soort databases is gebaseerd op het formele model van de relationele calculus [12]. Kort gezegd moet alle data passen in tabellen die bestaan uit rijen en kolommen. Elke kolom heeft een naam en bevat een specifiek type data. Een voorstel zou dus overeenkomen met een rij van een aantal kolommen: een identificatienummer, twee namen van fragmenten, een transformatie en een aantal attribuut-kolommen. Het “relationele” van de relationele calculus bevindt zich in het feit dat tabellen aan elkaar verbonden kunnen worden door middel van sleutelwaarden. Aan de hand van een sleutel kan een enkele rij in een tabel geïdentificeerd worden. Het identificatienummer van een paar is in dit geval een sleutel voor voor de tabel van alle paren. In de geschiedenistabel zal een rij – die een aanpassing voorstelt – steeds een kolom met het identificatienummer van het paar waarop de aanpassing is gebeurd bevatten. In dit project zijn er twee relaties, die tussen paren en hun geschiedenis, en die tussen de geschiedenis en gebruikers.

Op het eerste zicht lijkt het relationele concept goed te passen voor de data, maar wat is het alternatief? Gezien het feit dat NoSQL in feite staat voor niet-relationele database, is het niet verassend dat dit een brede waaier van verschillende ideeën en implementaties voorstelt. De drie meest invloedrijke subcategorieën zijn *sleutel-waarde*, *document* en *graaf*. Een bondige bespreking wijst uit welke toepasselijk is:

Sleutel-waarde

sleutel-waarde databases zijn in feite niets meer dan een externe hashtabel met wat extra functionaliteit en is een idee dat reeds lang bestaat (Berkeley DB bijvoorbeeld al 35 jaar). Het is mogelijk om alle data van het project in dit soort databases op te slaan door voor elk attribuut een hashtabel aan te maken, maar er wordt geen ondersteuning gegeven voor relaties tussen deze tabellen, een object ophalen vergt dus tenminste zoveel hash-opzoeken als het attributen heeft. Daarbovenop is de ondersteuning voor zoeken of sorteren bijna niet bestaande. Dit type databases moet dus niet overwogen worden.

Document

Document databases zijn het type databases waar bedrijven als Google en Amazon hun infrastructuur op gebaseerd hebben, het spreekt voor zich dat dit impliceert dat ze zeer goed schaleren naar grote datahoeveelheden. Het is een verdere uitwerking van het *sleutel-waarde* concept in de zin dat de *waarde* een structuur kan hebben: het kan uit verschillende velden bestaan. Wat is dan in feite het verschil met de eerder besproken relationele databases? Daar waren ook een sleutelkolom en verschillende niet-sleutelkolommen (waarden). Er zijn een paar kleine verschillen, maar het voornaamste dat van toepassing is op dit project is dat de structuur van de waarden

kan variëren. Dit wil zeggen dat een bepaald voorstel een “commentaar”-veld kan hebben en een ander niet. Lijst 5.2 geeft een voorbeeld van hoe dit eruitziet. Het lijkt er op dat *document* databases qua datamodellering nog net iets beter geschikt zijn voor voorstellen dan een relationeel model. De formele relaties tussen voorstellen, geschiedenis en gebruikers onderbreken echter. Dit kan gesimuleerd worden door de applicatie indien nodig.

Broncode 5.2: Een document database kan variërende kolommen binnen een tabel aan

```
...
id: 475 => { sourceFragment: "WDC13_BAK33_0049", targetFragment: "
WDC13_BAK33_0021", transformation: "...", status: "misschien", comment: "dit ziet er een
goed voorstel uit, zeker controleren" }
id: 476 => { sourceFragment: "WDC13_BAK33_0004", targetFragment: "
WDC13_BAK33_0020", transformation: "...", status: "niet geweten" }
id: 477 => { sourceFragment: "WDC13_BAK12_0064", targetFragment: "
WDC13_BAK15_0040", transformation: "...", status: "incorrect", duplicate: 6140 }
...
```

Graaf

Een fresco, bekeken vanuit het standpunt dat het opgemaakt is uit verschillende fragmenten (iets waar de originele kunstenaars misschien een probleem van zouden maken), kan men voorstellen als een graaf. Elk fragment is een knooppunt en elk paar een zijde. Ook dit lijkt dus een valide aanpak. De geschiedenislijst heeft niet zo’n natuurlijke vertaling naar een graaf, maar de meeste databases van dit type laten ook knooppunten zonder zijden toe, waardoor een “normalere” aanpak kan gesimuleerd worden. Alternatief kan men ook gewoon een andere database gebruiken voor de geschiedenis en gebruikers.

Het voordeel dat deze modellering heeft is dat het soort verzoeken dat moeilijk te optimaliseren valt in andere databasetypes, bijzonder snel kunnen beantwoord worden. Zo’n verzoek is bijvoorbeeld: geef alle fragmenten die te bereiken zijn van dit fragment en geconfirmeerd eraan passen (d.w.z.: de zijde(n) die hen verbinden hebben zijn gevalideerd). Dit verzoek zou een cluster van correcte fragmenten teruggeven en dus een stukje gereconstrueerd fresco voorstellen. Dit laatste is met een zeker snelheidsverlies nog uit te voeren met conventionele databases, maar er zijn nog moeilijkere denkbaar. Bijvoorbeeld: geef de cluster die het dichtst bij dit fragment ligt en alle fragmenten die op het pad ernaartoe liggen met een fout kleiner dan een getal. Omwille van exponentiële explosie is het moeilijk zonder versnellingsstructuren op dit verzoek te antwoorden, *graaf* databases zijn hier speciaal voor gemaakt. Dit impliceert wel een onacceptabele vertraging op

5.4.2 Expressiviteit

Het doel van dit project is tweevoudig: aan de ene zijde moeten paren op een snelle manier uit een database gehaald worden, anderzijds moet er dit op zo'n divers mogelijke manieren gebeuren. Anders geformuleerd: op welke soort vragen kan een database allemaal antwoorden? Dit bevindt zich in het domein van de expressiviteit. De gereduceerde predicaatlogica van het reeds bestaande systeem is niet expressief genoeg om vragen die meerdere paren aangaan te beantwoorden, zoals hierboven aangetoond.

De expressiviteit van SQL zonder procedures is volledig gelijk aan die van volledige eerste orde logica indien oneindige tabellen niet toegelaten worden, zoals werd aangetoond door Abiteboul et al. [13]. Meer nog, de SQL:2008 standaard is Turing compleet¹ en heeft dus dezelfde expressiviteit als gelijk welke programmeertaal. Dit laatste is wel nog niet door alle makers van SQL databases geïmplementeerd. Op het vlak van data-analyse is SQL dus moeilijk te verslaan.

Het lijkt erop dat er nog geen (succesvol) onderzoek werd gedaan naar de formele expressiviteit van NoSQL-type databases, dit verschilt ook per implementatie. Binnen elke subcategorie zijn de implementaties niet uniform, sommigen bieden zoekfunctionaliteit aan en andere laten dit over aan de applicatie. De algemene filosofie van het NoSQL concept is om expressiviteit en integriteit (van data) om te ruilen voor performantie. Dat gezegd zijnde, de meest uitgebreide van het eerder besproken *document* type, MongoDB, even expressief als SQL als men binnen een tabel blijft [2]. Figuur A.1 geeft dit grafisch weer. Het ontbreken van het relationele aspect bij NoSQL betekent dat twee tabellen niet kunnen verbonden worden door enkel de functionaliteit van de database te gebruiken. De gebruikelijke oplossing hiervoor is denormalisatie, een proces dat verschillende tabellen samenvoegt. De redundantie en de conflicten die hierdoor ontstaan worden dikwijls voor lief genomen uit snelheidsoverwegingen. Ruimte op een harde schijf is niet duur. Een alternatieve oplossing is om in de applicatielaag meerdere verzoeken uit te voeren en deze zelf te combineren, dit zorgt wel voor een gelijkaardig probleem als dat van de predicaatlogica.

De conclusie is dat SQL de meest expressieve taal is, sommige NoSQL varianten komen in de buurt en evenaren SQL met omwegen (denormalisatie).

5.4.3 Gebruiksgemak

Hoe gemakkelijk is het — voor de ontwikkelaar en de gebruiker — om het systeem te gebruiken? Hier heeft wederom SQL een licht voordeel, alle verzoeken worden uitgedrukt in een declaratieve taal die niet moet gecompileerd worden voor gebruik. De meeste NoSQL systemen hebben geen gelijkaardig tekstsysteem en vertrouwen op bindingen met de programmeertaal, wat het moeilijker maakt om complexe verzoeken

¹Dit is niet steeds een voordeel wegens het halting probleem, er bestaan queries in deze nieuwe standaard die niet eindigen.

van een gebruiker te incorporeren.

NoSQL schuift de verantwoordelijkheid voor het verbinden van verschillende soorten data door aan de applicatie. Als het gaat om een verzoeken als “vind een paar met eigenschap X en geef daar de geschiedenis van” is dit niet zo erg, in NoSQL moet dan eerst een verzoek gestuurd worden voor het paar en vervolgens één voor de geschiedenis. In SQL kan dit in een enkele query, NoSQL verspilt dus de bandbreedte van één identificatienummer van een paar (en de tijd van een bericht heen-en-weer naar de server). Als de paren die weerhouden moeten worden echter geconditioneerd zijn op de geschiedenis en op eigen attributen is het bij NoSQL moeilijk te vermijden dat er een grote hoeveelheid data heen-en-weer moet gestuurd worden. Denormalisatie is dus onontbeerlijk. Het opslagen van de geschiedenis als een onderdeel van het attribuut zelf maakt het onmogelijk om deze data voor te sorteren op tijdstip, met als gevolg dat incrementele synchronisatie zoals besproken in 6 niet mogelijk zou zijn.

5.4.4 Conclusie

Beide modellen ondersteunen in principe dezelfde soort operaties, de verschillen bevinden zich vooral in het gebruiksgemak en de performantie. NoSQL biedt grote snelheid op vaak uitgevoerde verzoeken en gemakkelijke schaleerbaarheid [14], SQL biedt gebruiksgemak en uitgebreide data-analyse zonder teveel in te boeten aan snelheid. Het valt niet te vergeten dat er talloze bedrijven zijn die SQL databases gebruiken om tabellen met miljoenen rijen te manipuleren [15, 16], de advertentiedienst van Google (*AdSense*) is trouwens gebaseerd op MySQL. Een voordeel van SQL systemen is dat er eenvoudig tussen verschillende implementaties kan gemigreerd worden als de vereisten veranderen. Een broekzakversie die bij momenten geen connectie met het internet heeft kan op het compacte SQLite vertrouwen terwijl een krachtige server een heel grote database kan draaien met MySQL/PostgreSQL/Oracle/.... Al deze verschillende implementaties kunnen met dezelfde subsystemen aangesproken worden. In conclusie, aangezien het onwaarschijnlijk is dat de performantie van een SQL database tekort schiet noch of het geoorloofd is aan expressiviteit en gebruiksgemak in te boeten, is dit een veilige keuze. De trage respons van NoSQL databases op zogenaamde ad-hoc (arbitraire) verzoeken is de finale nagel in de doodskist.

De wereld van het databasebeheer bevindt zich in een stroomversnelling en op elk gegeven moment kan er een fantastische nieuwe oplossing uit de bus komen. De manier waarop er nu van een XML database gemigreerd zal worden naar een andere technologie, kan later (gemakkelijker) herhaald worden als er een duidelijk betere oplossing zich voordoet. Het gros van de operaties op objecten vindt plaats via het model (zie hoofdstuk 4), een database-agnostische interface. De schijnbare uitzondering op deze regel lijkt het filteren te zijn, waar een valide (vereenvoudigde²) SQL WHERE declaratie gebruikt wordt om te zoeken. Deze zijn echter niet zo complex (zie 4.1) en

²Hoewel er niet gecontroleerd wordt of er geen SQL subqueries in de modelfilter aanwezig zijn, is het gebruik ervan niet “officieel” ondersteund

dus eenvoudig door het model om te bouwen naar iets wat elke onderliggende database wél kan begrijpen. Indien dit niet genoeg is, is er nog de optie om meerdere filtertypes toe te laten, een SQL-type, een MongoDB-type, enzovoort. Een implementatie van een model zal dan zelf moeten uitzoeken hoe het alles moet vertalen of gewoon de filter weigeren. De linkerkant van figuur 5.1 geeft aan welke componenten zouden moeten gemaakt worden om een nieuw database-type te ondersteunen, de rechterkant toont de integratie tussen het oude en het nieuwe systeem gebaseerd op SQL.



Figuur 5.1: Overzicht van het database systeem, links (groen) is een mogelijke uitbreiding, het middelste deel (oranje) stelt de thesis voor en helemaal rechts (wit) staan de componenten van het thera project.

5.5 Implementatie

Ondanks de simpliciteit van de te modelleren data zijn er verscheidene manieren om dit te doen binnen het relationele concept. In figuur 5.2 staat een schematisch model dat moet omgezet worden in SQL tabellen. Het staat buiten twijfel dat tenminste de geschiedenis en de gebruikers in een aparte tabel moeten: voor elk attribuut kunnen er immers verschillende rijen in een geschiedenistabel staan, en meerdere rijen in een geschiedenistabel kunnen door dezelfde gebruiker veroorzaakt zijn. Attributen zouden als kolommen in de “voorstel” tabel kunnen gaan, of elk een eigen tabel hebben. De eerste versie komt min of meer overeen met hoe een NoSQL *document* database dit zou voorstellen. Het voordeel van de kolom-aanpak (die *denormaler* is dan de tabel-aanpak) is grotere snelheid bij het opvragen (geen JOINS³) en het

³Het commando om twee tabellen met elkaar te verbinden heet een JOIN in SQL. Hoewel de database dit expliciet voorziet en opsplitsen in tabellen aangemoedigd wordt, is het verbinden van veel tabellen vaak nefast voor de performantie.

neemt minder ruimte in op de harde schijf omdat er geen verwijzingen moeten bijgehouden worden voor elk attribuut. Het laatste voordeel voert de snelheid nog eens op want er kan zo meer data in het werkgeheugen blijven. Anderzijds laat de tabel-aanpak toe om attributen te definiëren die niet voor alle fragmentparen bestaan, bijvoorbeeld als slechts 1000 van het miljoen paren een commentaar bevat. Op die manier kan afhankelijk van de verdeling van de attributen zelfs nog meer ruimte bespaard worden. SQLite, de implementatie voor kleine lokale kopieën kan ook geen kolommen verwijderen van een tabel. Omdat het eenvoudig is de tabel-aanpak om te zetten in de kolom-aanpak is deze preferibel om te beginnen. De eigenlijk beslissing kan uitgesteld worden tot wanneer blijkt dat er performantieproblemen zijn.



images/databasemodel.png

Figuur 5.2: Een relationele modellering van de data

5.5.1 Normale attributen

Dit model heeft een groot voordeel: het is voorspelbaar maar uitbreidbaar. De voorspelbaarheid van het attribuut-formaat zorgt ervoor dat queries automatisch kunnen gegenereerd worden, omdat elk veld aan een specifieke structuur voldoet. Steeds is het mogelijk om attributen toe te voegen en te verwijderen: de databaselaag “herkent” elke tabel die eruitziet als een attribuut en brengt alle geïnteresseerde componenten hiervan op de hoogte.

5.5.2 Complexe attributen


De (verplichte) voorspelbaarheid van de attributen kan ook op een dwangbuis lijken: er is zeker informatie denkbaar die niet simpelweg in één kolom past of gaat over meer dan één paar tegelijkertijd. In extremis is het mogelijk om dit soort data toe te voegen en te onderhouden door een module te schrijven. Er bestaan dan tabellen waarmee de databaselaag niet kan werken, maar de module kan wel via ruwe queries aan die data. Dit heeft natuurlijk het nadeel dat een heel deel van de infrastructuur op die manier ontweken wordt. Andere modules kunnen deze data niet “ontdekken” en weergeven en de queries worden niet automatisch geoptimaliseerd. Filters en sorteeroperaties zijn dan ook niet toegankelijk via de normale interface.

Een oplossing hiervoor zijn meta-attributen. Deze zijn niet schrijfbaar omdat ze opgemaakt zijn uit samengestelde data van andere tabellen. Voor de rest gedragen ze zich exact zoals normale attributen. Dit wil zeggen dat ze automatisch kunnen ontdekt en gebruikt worden. Het beste hieraan is dat dit allemaal in de declaratieve taal van de database kan gebeuren, waardoor extra modules of een hercompilatie van het programma niet nodig zijn.

Deze functionaliteit wordt geïmplementeerd met behulp van SQL Views, een manier om een verzoek als een tabel voor te stellen. Als er een verzoek op onregelmatig gevormde data kan gemaakt worden dat **lijkt** op een attribuut, kan dit een meta-attribuut zijn. Om de zojuist beschreven zaken wat minder abstract te maken, volgt een voorbeeld van een twee echte meta-attributen:

Duplicaten

De automatische herkenners stellen soms paren voor die hard op elkaar lijken maar niet exact dezelfde zijn (zie figuur 5.3). Om data-analyse redenen worden deze niet automatisch verworpen, ze hebben licht verschillende eigenschappen en het kan nuttig zijn deze te onderzoeken.



images/duplicates.png

Figuur 5.3: Duplicaten: het tweede paar is een duplicaat van het eerste, het derde niet.

Als men niet specifiek op zoek is naar duplicaten, is het dikwijls beter om enkel het beste paar uit een groep duplicaten te laten zien. Tegelijkertijd moet de gebruikersomgeving een visuele hint kunnen geven dat een bepaald voorstel enkele duplicaten heeft. Opslaan welk paar representatief is, kan nog voorgesteld worden als een attribuut. De waarde van dit attribuut, “duplicaat” genaamd, is een verwijzing naar het representatieve paar van de groep of het getal 0. Als de waarde 0 is, geeft dit aan dat een paar de beste uit zijn groep is (elk paar is dus in het begin de beste uit de groep die alleen het paar zelf bevat). Als men wil aanduiden dat een groep paren een duplicaat is van een ander, volstaat het om de waarde van hun “duplicaat”-attribuut op het identificatienummer van dit paar te zetten.

Er is slechts één probleem over: door te kijken naar het “duplicaat”-argument kan men niet weten of dit paar duplicaten heeft of niet (enkel of het een duplicaat of een representatief paar is). Op het eerste zicht lijkt het nodig een apart attribuut “aantal_duplicaten” bij te houden en dit telkens aan te passen als er ergens een in “duplicaat” een verwijzing veranderd. Dit is mogelijk maar zeer gevoelig voor corruptie (desynchronisatie van beide tabellen). Eigenlijk kan het aantal duplicaten rechtstreeks uit het “duplicaat”-attribuut worden gehaald: voor het paar waarvan men wil weten of het een duplicaat heeft, tel hoeveel ernaar verwijzen. Een apart attribuut bijhouden zou dus redundant zijn. Deze telsom valt in een SQL-verzoek te vertalen:

Broncode 5.3: Deze query kan als view dienen om het aantal duplicaten per paar als meta-attribuut te gebruiken

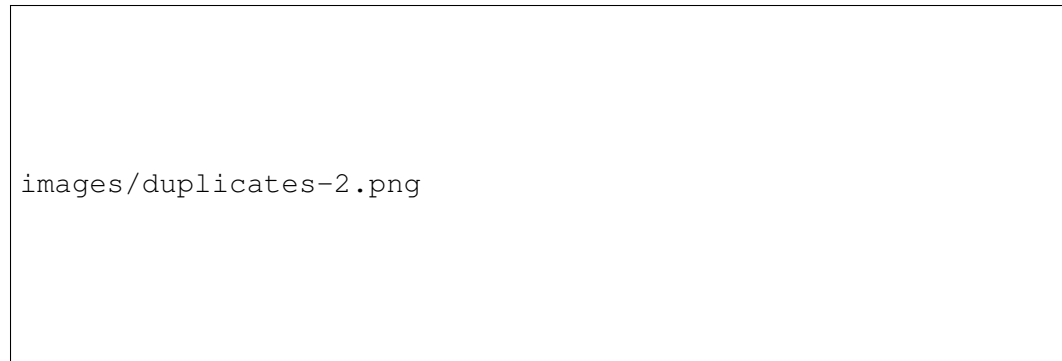
```
SELECT
  duplicate AS match_id,
  COUNT(duplicate) AS num_duplicates
FROM duplicate
GROUP BY duplicate
```

De databaselaag kan van deze zin een meta-attribuut maken dat aangeeft hoeveel duplicaten een paar heeft. Het gedraagt zich als een extra tabel maar is in feite gewoon een datatransformatie. De *MatchTileView* module die later wordt besproken kiest ervoor om dit meta-attribuut weer te geven door een paar laagjes onder een afbeelding van het paar te tekenen, zoals in figuur 5.4.

Correspondentie

Een ander voorbeeld van een meta-attribuut is correspondentie. Alle commentaren die ooit op een fragment zijn gegeven kunnen aan elkaar worden gezet! Een voorbeeld van een query die hiervoor kan dienen is broncode 5.4

Zo zijn er nog vele andere meta-attributen denkbaar, er is geen beperking op hoeveel tabellen er moeten gecombineerd worden om het te verkrijgen zolang het resultaat maar de vorm van een normaal attribuut heeft.



Figuur 5.4: Duplicaten: het tweede paar van figuur 5.3 verwijst nu naar het eerste paar

Broncode 5.4: Deze query kan als view dienen om de geschiedenis van een “commentaar”-attribuut om te vormen tot een “correspondentie”-attribuut

```
SELECT
  match_id,
  GROUP_CONCAT(COMMENT SEPARATOR '|' ) AS correspondence
FROM comment_history
GROUP BY match_id
ORDER BY TIMESTAMP DESC
```

5.5.3 Querygeneratie

Omdat verschillende modules en bij extensie gebruikers andere noden hebben, worden de verzoeken naar de database steeds opnieuw gegenereerd uitgaande van een stel parameters.

De eerste iteratie van het systeem was zeer eenvoudig: een query bestond uit een sorteerveld, sorteerrichting en een enkele filter op een attribuut. Dit werkte acceptabel voor kleine collecties zoals degene waar Browsematches en Griphos mee moeten werken, maar was niet echt vernieuwend.

Een aanzienlijk probleem was bijvoorbeeld dat alle fragmenten die voldeden aan de filtercriteria werden opgehaald. Een duizendtal paren van een SQL implementatie die lokaal draait (of zelfs volledig in het geheugen, zoals SQLite) is snel. Tienduizenden paren gaat niet meer zo vlot. Niet alleen nemen die veel geheugen in, de informatie in het verzoek moet elke keer verwerkt worden en omgezet in evenveel C++ objecten. Dit is vooral traag omwille van de opslag van de transformatiematrix als pure tekst, die telkens omgezet wordt naar een matrix van (32-bit) decimale getallen. De tekstopslag laat arbitraire precisie toe, wat belangrijk is om het voorstel van

de automatische herkenner zo getrouw mogelijk te maken. De volgende subsecties bespreken eerst hoe filters mogelijk zijn en daarna enkele optimalisaties die het systeem efficiënt maken voor grote datasets en over (trage) netwerkverbindingen.

Filters

In hoofdstuk 4 werd reeds uit de doeken gedaan dat er meerdere filters tegelijkertijd mogelijk zijn. Alle attributen zitten opgeslagen in aparte tabellen, in SQL moet een query deze expliciet vermelden. Aangezien de filters willekeurige tekst zijn die hopelijk welgevormde SQL voorstellen, is een tekstanalyse nodig om te bepalen op welke attributen ze steunen. Dit wordt gedaan met reguliere expressies⁴. Alle filters worden conjunctief (met het “AND” sleutelwoord) in een WHERE-clausule gezet en alle ontdekte afhankelijkheden worden met een “JOIN” in het domein gebracht.

Merk op dat indien eerder de kolom-aanpak — d.w.z. alle attributen als extra kolommen in de parentabel — was genomen, deze tekstanalyse niet nodig zou geweest zijn. Gelukkig blijkt de analyse nuttig voor andere optimalisaties die hieronder worden beschreven.

Standaardisatie

Ondanks het feit dat SQL een gestandaardiseerde taal is, gebruikt elke implementatie wel een eigen dialect. Daarom loopt een preprocessor steeds door de volledige query om de dialecten te unificeren, ook hiervoor worden reguliere expressies ten volle benut.

Paginatie

Tienduizenden paren ophalen werd al aangegeven als een oorzaak van traagheid, zelfs op lokale installaties. Hetzelfde probleem doet zich nog veel erger voor als de verbinding naar de database over het netwerk loopt. Zonder de efficiëntieverliezen van de databaseopslag neemt elk paar zonder attributen ongeveer 924 bytes in⁵. Het is niet ongewoon om verzamelingen van 10000 paren te weerhouden, wat een ondergrens van 8.8MB over het netwerk zou betekenen. Een redelijk doel lijkt om alle verzoeken binnen een seconde af te werken, wat niet mogelijk is met zulke hoeveelheden data.

Om die reden zal het model bijna nooit de volledige set aan de database opvragen, maar slechts een klein deeltje. Allereerst wordt aan de database gevraagd hoeveel paren aan de criteria voldoen, zodat het model kan rapporteren dat het X aantal paren in bezit heeft. Dit laadt ook meteen veel noodzakelijke data in het werkgeheugen van de server. Als een component dan vraagt om een bepaald paar, zal het model kijken of de aanvraag binnen het bereik van het huidige venster past. Indien niet, wordt het venster opgehaald waarin de aanvraag wel past. Dit proces is transparant

⁴Een formele taal die het mogelijk maakt om ingewikkelde patronen in tekst te herkennen.

⁵Het grootste deel hiervan is afkomstig van de tekstvoorstelling van de transformatiematrix

voor de componenten die van het model gebruik maken. Ze kunnen de parameters zoals venstergrootte wel beïnvloeden door hints te geven, hoewel het model niet verplicht is deze te volgen. Dit is vooral handig voor componenten die een willekeurig toegangspatroon naar de data hebben, waarvoor vensters bijzonder inefficiënt zijn omdat er bij elke aanvraag een volledig nieuw venster opgehaald wordt.

Voorladen

Een component kan aan een fragmentpaar vragen welke waarde het heeft voor een bepaald attribuut. De eerste versies van zo'n fragmentpaar-objecten vuurden een query af naar de database elke keer een attribuut opgevraagd werd. Dit is eenvoudig en zorgt ervoor dat de waarde altijd zo actueel mogelijk is. De query op zich is bijzonder eenvoudig en kan razendsnel afgehandeld worden door de database. In een venster van de belangrijkste module passen 20 fragmentparen. De module heeft van elk paar ongeveer 5 attributen nodig om weer te geven, dit geeft 100 extra verzoeken aan de database na het ophalen van de paren zelf. Voor een lokale server is dit geen probleem, maar als de verzoeken over het netwerk moeten komen er tenminste 100 maal de heen-en-weer tijd van een netwerkpakket bij.⁶

Onafhankelijk van de snelheid van het verzoek maakt deze techniek werken over het netwerk te traag. Om die reden beschikt elk fragmentpaar over een datastructuur waarin het eerst zal kijken of het geen lokale kopie van het attribuut heeft. De querygenerator werd voorzien van de capaciteit om een parameter "voorlaadvelden" aan te nemen, die ervoor zorgt dat een paar reeds voorgevuld wordt met de gevraagde attributen. Het aantal queries per venster loopt dan terug naar 1. Dit heeft geen impact op de versheid van de data, de attributen worden in ieder geval slechts bij het laden van het venster opgevraagd.

Geforceerde indexen

Een index is een versnellingsstructuur die de database toelaat om niet de volledige collectie af te gaan bij elk verzoek. Meestal gaat het hier over een binaire boom die ervoor zorgt dat queries op exacte waarden en bereiken (vb. $error < 0.50140 \wedge error > 0.45713$) zeer snel gaan. Indexen zijn gedefinieerd per kolom of groepering van kolommen, het voorbeeld van daarnet heeft dus een index op de "error" kolom nodig. Dit is één van de zaken die het predicaten-systeem van het thera project mistte.

De query planner van MySQL en SQLite is niet zo geavanceerd als die van de commerciële databases, en maakt daarom nogal eens grove fouten als het aankomt op het kiezen van de juiste index. Meer nog, deze twee implementaties zijn gelimiteerd tot één index per query, en als deze verkeerd gekozen wordt kan dat soms desastreuze gevolgen hebben voor de snelheid. Om deze cruciale beslissing te maken kijken zij vooral naar de selectiviteit. Samengevat: de query planner kiest over het algemeen de index voor de conditie die de dataset hopelijk het meest verkleint. Als er geen

⁶Queries worden niet parallel uitgevoerd

condities maar er wel een gesorteerd resultaat is aangevraagd, zal de planner een index gebruiken om te sorteren als die bestaat (binaire bomen worden gesorteerd bijgehouden).

Het probleem is dat deze heuristiek geen rekening houdt met een paar belangrijke zaken. Stel een database met een miljoen paren voor en een conditie in een query brengt dit terug naar een half miljoen, dit half miljoen is niet gesorteerd. Indien er toch gesorteerd moet worden, moet dat in situ gebeuren, zonder een index. Dit is reeds zeer traag op zich. Daarbovenop past zo'n resultaat niet in het werkgeheugen, dus de database sorteert in stukjes door het resultaat weg te schrijven naar het bestandssysteem. Dit gebeurt allemaal ondanks het feit dat er uiteindelijk slechts enkele paren gaan opgevraagd worden.

Daarom is het beter om in dit geval selectiviteit te laten voor wat het is en een index te gebruiken om het resultaat reeds gesorteerd op te vragen. Op die manier moet er geen tijdelijk bestand aangemaakt worden. De querygenerator van dit project is geavanceerd genoeg om te detecteren wanneer het beter is om een sorteerindex te forceren en doet dit ook.

Snelle paginatie

Paginatie is goed om de hoeveelheid getransfereerde data te minimaliseren, maar de simpele vorm is traag voor grote databases gecombineerd met vensters ver in de dataset. Zo kan een query die voor venster $[100, 120]$ nog binnen de 50 milliseconden afgewerkt kan worden, plots 7000 milliseconden duren voor venster $[10000, 10020]$. De reden heeft onder andere te maken met het feit dat voor de tweede query alle 10000 rijen die niet moeten opgehaald worden toch worden nagekeken. De database kan niet op voorhand weten welke rij precies de 10000'ste is onder de geldende criteria. Bij grote databases is het erg waarschijnlijk dat deze 10000 rijen niet in elkaars buurt staan op de harde schijf, wat de lange zoektijd verklaart.

Een oplossing is om de database contextinformatie mee te geven [17]. Als het vorige opgevraagde venster afdalend gesorteerd was op de *fout*, dan zal het volgende venster beginnen met een *fout* kleiner of gelijk aan de fout van het einde van het huidige venster. Door dit mee te geven in conditie kan de database gebruik maken van een index en zo de opvraagtijd dramatisch inkorten. Voor dit project is deze techniek uitgebreid geweest naar een krachtigere variant die “relatieve positionering” werd gedoopt. Dit wil gewoon zeggen dat de querygenerator altijd de meest dichtbijzijnde gekende waarde zal gebruiken om een nieuw venster op te halen.

Late rijopzoeking

Late rijopzoeking is uiteindelijk de meest effectieve optimalisatie van allemaal. Het maakt sommige queries zo snel dat het effect van de andere optimalisaties niet meer te merken is. Eén van de voornaamste redenen waarom queries traag zijn is

dat elke keer er een waarde uit een rij nodig is, alle gevraagde kolommen worden opgehaald, of de rij nu in het uiteindelijke resultaat zit of niet. Als het veld dat nagekeken moet worden een simpel geheel getal is van 4 bytes, is het bijzonder kwistig om meteen nog eens een slordige 1000 bytes aan misschien onnodige informatie op te halen. Om de zaken erger te maken staan deze 1000 bytes met grote kans opnieuw niet dicht bij elkaar op de harde schijf⁷, wat voor hoge zoektijden zorgt.

Dit slecht gedrag ontwijken kan door middel van een subquery. Het basisidee is: de binnenste query reduceert de dataset zoveel mogelijk en haalt enkel de nodige informatie op om de correcte rijen te identificeren (een sleutel). De buitenste query haalt dan al de nodige informatie op aan de hand van die sleutel. Met een venster kunnen er dus nooit meer rijen volledig opgehaald worden dan de venstergrootte.

Om de binnenste query zo efficiënt mogelijk te maken is het voordelig te weten welke velden en tabellen nodig zijn om de dataset te reduceren en welke kunnen doorgeschoven worden naar de buitenste query. Dat is waar de afhankelijkheidsanalyse van daarstraks goed van pas komt.

Normaalgezien zou de queryplanner van de database deze optimalisatie zelf moeten uitvoeren, maar zowel SQLite, MySQL en verassend genoeg PostgreSQL⁸ leken hier (veel) voordeel bij te halen.

Experimenteel: materialized views

Een spijtig nadeel van de kracht van meta-attributen is dat ze queries sterk vertragen. Hun dynamische aard zorgt ervoor dat de achterliggende query steeds wordt uitgevoerd voor elk verzoek. Een rij ophalen van “aantal_duplicaten” impliceert op die manier een volledige scan de “duplicaat” tabel. Vele oplossingen voor dit probleem werden getest maar geen enkele bracht de nodige verbetering of waren slechts van toepassing op een beperkte subset van de mogelijke verzoeken. Het enige wat overbleef was doen wat meta-attributen net proberen te vermijden: een echte tabel maken van de view. Natuurlijk moest dit gebeuren zonder de nadelen van een echte tabel die niet gedesynchroniseert kan geraken met de originele tabellen.

Het concept op zich heet een gematerialiseerde view. De opzet is om een automatische verbinding te maken tussen de moedertabellen en het meta-attribuut, zodat elke update gepropageerd wordt. Dit is een simpel uitgangspunt, maar heeft een verassend moeilijke uitwerking. De huidige implementatie is zeer rudimentair en werkt alleen in PostgreSQL, de implementatie met de meest geavanceerde programmeermogelijkheden. De resultaten zijn echter veelbelovend, zie sectie 5.7.

⁷Dit proximateitsargument is niet van toepassing op *Solid State Drives*

⁸De queryplanner van PostgreSQL staat bekend als een van de meest geavanceerde, en zou zelfs specifiek late rijopzoeking ondersteunen.

5.5.4 Collaboratie

De verschillende optimalisaties uit de vorige secties maken het mogelijk om een externe database te gebruiken en toch met grote snelheid te navigeren in de dataset. Dit betekent dat iedereen steeds de meeste actuele data te zien krijgt. In deze opstelling is synchronisatie niet nodig, zolang er een netwerk- of internetverbinding beschikbaar is. Als meerdere gebruikers tegelijkertijd met dezelfde paren aan het werken zijn, zullen ze elkaars veranderingen niet zien verschijnen tot ze het venster vernieuwen. Voor deze situatie is een oplossing gemaakt die gebruikmaakt van het feit dat de meeste queries snel afgehandeld worden. Als het model merkt dat het ophalen van een venster snel gaat, zal het periodisch proberen dit venster te vernieuwen. De periode past zich aan de omstandigheden aan, hoe langer het duurt om een venster op te vragen, hoe langer het model zal wachten. Dit is zo omdat de reden voor de traagheid misschien een overbelasting van de database is.⁹

Dankzij dit systeem verschijnen de laatste nieuwe veranderingen steeds op het scherm van de gebruiker, zodat geen dubbel werk wordt verricht.

5.6 Data mining

Maak ketting van (yes+maybe) —> zoek naar alle burens binnen 1 hop (algoritme is geschreven) — FINDNEIGHBOURS.SQL

Meer mogelijkheden: geef alle paren die recent zijn aangepast.

5.7 Benchmarks

Er zijn reeds een paar namen gevallen van gebruikte database implementaties: SQLite, MySQL en PostgreSQL. In het begin draaide Dankzij de flexibiliteit die nodig was om zowel SQLite als MySQL het grootst mogelijke deel van de code te laten delen

5.7.1 Indexering

Analyse van datatoegangspatroon: vooral SELECT, ORDER BY, GROUP BY —> veelvuldig gebruik van indexes

optimize voor fast reads —> inserts kunnen lokaal gedaan worden, updates hopelijk niet zo veel of lokaal

5.7.2 Opstelling

Beide databases kregen 512 MB RAM aan werkgeheugen toe. Het is natuurlijk niet mogelijk om exact dezelfde configuratie te krijgen omdat de opties verschillen.

De query cache staat af Om de variabiliteit van de filesystem (nederlands) cache een beetje buiten spel te zetten zijn al deze routines “opgewarmd”

⁹Geïnspireerd door het dynamisch gedrag van het TCP pakketprotocol

Pagination Late row lookup

Ideaal = minder dan een seconden voor elke gegeven query vanuit een gebruikersinteractie standpunt (System Response Time and User Satisfaction pagina 5)

Effect van DB configuratie (veel geheugen...)

Suggested workaround voor het text probleem -> sphinx, restrict fragment names (niet ZO gemakkelijk), string + nummer Suggested workaround voor het indexing probleem (zoals gezien voor status IN (...)) -> force een index?! dunno... hij pakt in ieder geval de verkeerde!

'High Performance MySQL', Second Edition, O'REILLY, ISBN: 978-0-596-10171-8
MySQL Reference Manual for version 5.1

<http://nlp.stanford.edu/IR-book/html/htmledition/permuterm-indexes-1.html> (dit is hoe wildspeed werkt) (LIKE performance lijkt niet zo slecht in Postgres, het is de sorting eerder...)

<http://www.slideshare.net/techdude/how-to-kill-mysql-performance>

<http://stackoverflow.com/questions/1540590/how-to-speed-up-like-operation-in-sql-postgres-preferably> <— use trigrams (fail), MAAR BETER IN 9.1 (future research) <https://cgsrv1.arcc.csiro.au/views-for-postgresql/>

Hoofdstuk 6

Synchronisatie

Het thera project bestaat al een tijdje en er zijn reeds vele XML-bestanden met validaties van voorstellen in omloop. Ondanks de eventuele overgang van het oude naar het nieuwe systeem is het belangrijk dat deze resultaten niet verloren gaan, het heeft immers veel werk gekost om ze te maken. Als bestanden verschillende paren bevatten — omdat ze bijvoorbeeld door een ander identificatiealgoritme zijn gegenereerd — maar over eenzelfde opgraving gaan, is het ook nuttig om deze te consolideren. Indien er geen internetverbinding is of even geen storing van anderen wil hebben, kan een lokale kopie geëxtraheerd en later terug geïmporteerd worden. Dit alles kan gedaan worden met de synchronisatiecomponenten van dit project.

6.1 Opzet

Om te synchroniseren selecteert men een meester- en een slaaf-database, de conventie is dat de slaaf door de meester zal geabsorbeerd worden. Na het proces heeft de meester alle gewenste veranderingen en blijft de slaaf onveranderd over. Er zijn verschillende fasen (voorlopig 3): **Gebruikers**, **Paren** en **Attributen**. In elke fase krijgt de gebruiker een scherm te zien met daarin de verschillen tussen beide databases, het biedt de mogelijkheid om aanpassingen te maken alvorens naar de volgende stap te gaan. Het is belangrijk gebruikers keuzes te laten maken waarin ze geïnteresseerd zijn en ze niet lastig te vallen met keuzes die ze niet willen maken, beide zaken negeren leidt tot frustraties [8]. De algoritmes proberen daarom steeds een standaardkeuze in te vullen aan de hand van heuristieken, deze kunnen indien gewenst verworpen worden. Als er door gebrek aan informatie geen automatische beslissing kan genomen worden, moeten alle conflicten eerst manueel opgelost worden. Figuren 6.1 en 6.2 tonen hoe deze stappen eruitzien en figuur A.3 geeft een schematische kijk op het systeem. Sommige SQL-implementaties bieden automatische replicatie aan, dit wordt niet gebruikt omdat het minder flexibel is qua conflictresolutie noch compatibel over implementaties heen.

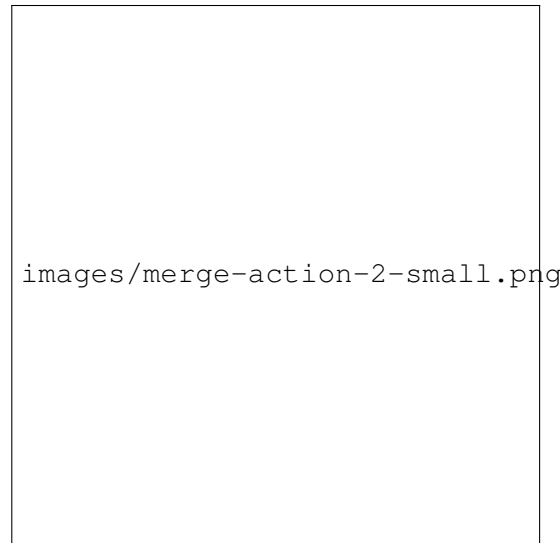


Figuur 6.1: Een uittreksel van de attributen-fase van het synchronisatieproces. “No Action” duidt aan dat er geen automatische resolutie toegepast is geweest, hierop dubbelklikken geeft een keuzeschermdoosje weer (rechts).

6.1.1 Paren

In deze stap worden de voorstellen zonder attributen uit beide databases vergeleken en gesynchroniseerd. Deze objecten bevatten een identificatiecode, twee namen van fragmenten en een 3D-transformatiematrix. De identificatiecode is echter geen uniek gegeven over alle databases, eerder een manier om efficiënt naar het object te verwijzen binnen eenzelfde database. Het is echter duidelijk dat als twee voorstellen uit dezelfde fragmenten bestaan en dezelfde transformatie hebben, ze identiek kunnen geacht worden. De aanwezigheid van afrondingsfouten en duplicaten is problematisch. Duplicaten zullen per definitie een gelijkaardige transformatiematrix hebben, hoe ze te onderscheiden van afrondingsfouten? De matrix van een voorstel uit de slaaf-database wordt afgetrokken van de matrix van elk voorstel uit de meester-database dat uit dezelfde fragmenten bestaat, hiervan wordt vervolgens de Frobenius norm¹ genomen. Hieruit ontstaat een rij van getallen die aangeven hoe gelijkaardig het voorstel is aan elk van de mogelijke overeenkomstige voorstellen uit de meester-database. Het kleinste getal geeft het meest overeenkomstige paar aan. Indien dit kleinste getal een ordegrrootte (10x) kleiner is dan het maximale verschil tussen dit overeenkomstig paar en diens duplicaten, wordt aangenomen dat het de paren in kwestie identiek zijn. Indien niet wordt het paar uit de slaaf-database als een nieuwe paar ingevoerd.

¹De euclidische norm van de vector die ontstaat als men de rijen of kolommen van een matrix achter elkaar plaatst.



Figuur 6.2: Elk type conflict laat een aantal manieren toe om het op te lossen. Deze figuur beeldt een attribuut-conflict voor.

De volledige voorstellenlijst van beide databases moet dus nagekeken worden. Eenmaal beiden in het geheugen geladen zijn verloopt het vergelijkingsproces vrij snel (er wordt een hashmap aangemaakt om in constante tijd te kunnen zien welke voorstellen uit dezelfde fragmenten bestaan). Dit wil natuurlijk wel zeggen dat het synchroniseren van pure paren op zich over het internet niet werkbaar is met een database van miljoenen elementen. Gelukkig gebeurt het niet vaak dat onderzoekers paren toevoegen.

Paren die overeenkomen worden aan een identiteitsvertaler toegevoegd, waardoor latere fasen weten dat twee verschillende identificatienummers naar hetzelfde object verwijzen. Een paar dat niet gevonden werd in de meester-database wordt niet automatisch ingevoerd maar als een conflict weergegeven. De gebruiker kan gemakkelijk kiezen om dit toch te doen door “assign new id” als actie toe te kennen en aan te vinken om dit te doen voor alle gelijkaardige conflicten (zoals in figuur 6.2). Het alternatief is “don’t merge”, wat ervoor zorgt dat de nieuwe paren niet toegevoegd worden (en genegeerd in de volgende stappen).

6.1.2 Attributen

Het samenvoegen van attributen verloopt op een andere manier. Het is gemakkelijk te detecteren wanneer zich een conflict voordoet, namelijk als de waarden verschillen. Dit conflict oplossen gaat echter niet automatisch, als een *commentaar*-attribuut bijvoorbeeld twee verschillende teksten bevat, welke is dan de juiste? Misschien de meest recente, zeker als blijkt dat een vorige waarde van de meest recente gelijk is aan de minder recente (gemeenschappelijk ouder). Om automatische resolutie te

ondersteunen moet er dus een geschiedenislijst bijgehouden worden. Daarnaast kan nog gekeken worden naar de semantische inhoud van het attribuut om eventueel een oordeel te vellen als de geschiedenismethode faalt.

Gemeenschappelijke ouder

Indien beide databases voor een bepaald attribuut een stukje geschiedenis delen, is er een kans dat het conflict automatisch opgelost kan worden. Stel dat in een kopie van een database een attribuut veranderd en ditzelfde attribuut is niet aangeraakt in de originele database. In dat geval kan de nieuwe waarde zonder meer overgenomen worden bij synchronisatie. Het kijken naar een gemeenschappelijke ouder om te synchroniseren wordt in de literatuur *3-way merging* [18] genoemd. De zojuist beschreven situatie komt overeen met het meest linkse vak in figuur 6.3. Het is de meest voorkomende want uit de grote hoeveelheid paren is de kans klein dat exact dezelfde (in een korte tijdsspanne) bewerkt worden. In het geval dat dit wel zou gebeuren — het middelste vakje in de figuur — kan er niet op die manier gesynchroniseerd worden. Hetzelfde doet zich voor als de beide databases geen geschiedenis delen, deze situatie wordt voorgesteld door het meest rechtse vakje. Samenvattend kan er dus niet automatisch gehandeld worden als de geschiedenis divergeert ofwel niet gedeeld is.

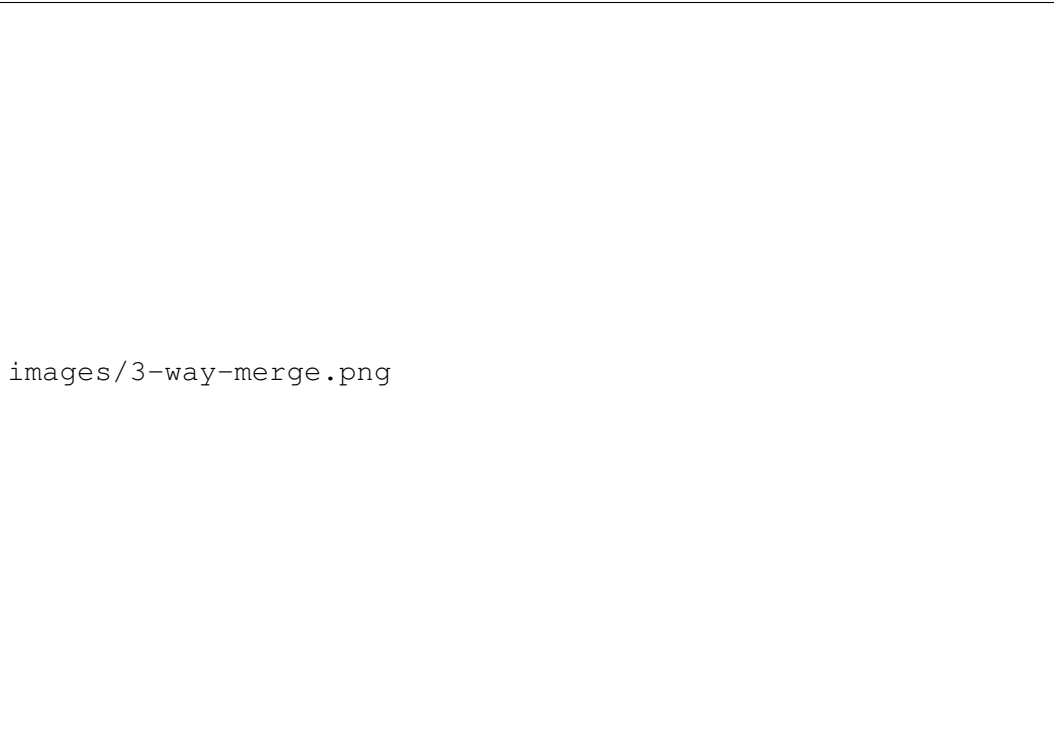
Context

Toch zijn er situaties denkbaar waar er toch een oplossing is. Stel dat de validatiestatus van een bepaald voorstel “niet geweten” is en de database wordt gesplitst in database A en B. In database A wordt vervolgens de status op “misschien” gezet en in B op “correct”. Als de synchronisatie afweet van de semantiek van statussen, zal het correct kunnen afleiden dat “correct” een krachtiger status is en dit verkiezen. Voor commentaren kan bijvoorbeeld gelden dat het meest recente de bovenhand krijgt. Dit zijn contextbeslissingen, waarbij de component op de hoogte moet zijn van de betekenis achter de waarden. Dit soort acties kan over het algemeen niet overgezet worden van het éne attribuut op het andere, hetgeen aparte routines voor elk contextgevoelig attribuut veronderstelt.

Er zijn ook attributen waar dit soort beslissingen niet eenvoudig kunnen genomen worden (bvb. duplicaten), of waarvoor de routines nog niet gemaakt zijn. In dit laatste geval kan er geen automatische resolutie plaatsvinden en moet de gebruiker ingrijpen alvorens verder te gaan. Er zijn heel wat verschillende acties geldig op een attribuut conflict, enkele daarvan zijn: “pick most recent”, “prefer user [username]”, “prefer slave” en “don’t merge”.

6.1.3 Incrementele synchronisatie

Omdat er een geschiedenis wordt bijgehouden is incrementele synchronisatie voor attributen mogelijk als beide databases geschiedenis delen. De geschiedenis wordt



images/3-way-merge.png

Figuur 6.3: Situaties waarin automatische resolutie gebaseerd op geschiedenis mogelijk is (eerste figuur) en waar contextinformatie nodig is (tweede en derde figuur). Links is hier gelijk aan de meester-database en rechts is de slaaf-database.

voor elk attribuut apart bijgehouden dus de gemakkelijkste en meest robuuste oplossing is om per attribuut een binair zoekalgoritme te laten zoeken naar het laatste gemeenschappelijke punt. In feite is dit hetzelfde algoritme als hetgene dat gebruikt wordt om de gemeenschappelijke ouder te vinden, met het verschil dat er in deze versie naar alle paren wordt gekeken in plaats van slechts één. Dit is robuust omdat er veel verschillende manieren zijn om databases te kopiëren en er dus niet kan vertrouwd worden op het aanwezig zijn van splitindicatoren. Enkel de kost van het binair zoeken en het overzetten van de veranderde attributen is dan van belang. Op een database van 50000 paren kan dit een groot verschil opleveren als er bijvoorbeeld slechts 100 attributen veranderd zijn. Op die manier wordt synchroniseren over het internet mogelijk.

Hoofdstuk 7

Modules

Hier komen de modules!!!!

Pure grafische plugins die op andere plugins vertrouwen voor data-selectie: Om de werkbaarheid van dit systeem uit te testen werd een voorbeeldmodule ontwikkeld die alle huidige paren in een grafe plaatst en deze met een ontwaringsalgoritme probeert te plaatsen, zodat men een globaler beeld kan krijgen van de huidige selectie.

7.1 MatchTileView

Let uit wat de doorsneden doen petra [\[19\]](#)

Conflicten, verzamelingsreductie door conflicten (hoeveel percentage kan zo uitgeruled worden?) —> space-filling curve

zoeken naar burens, niet conflicterende burens, duplicaten aanduiden, icoonbalken voor commentaren, DetailView, blabla

7.2 Proof of concept: GraphView

Hoofdstuk 8

Toekomstig werk

Met dit project is ook een verdere stap gezet naar mobiele toepassingen. Het data-luik staat bijvoorbeeld toe om applicaties voor tablets te schrijven die beroep kunnen doen op een externe database om een groot deel van het zware werk over te nemen. Deze soort programma's kunnen het manueel verifiëren van paren sneller en aangenamer maken. De huidige werkwijze is als volgt: nadat er een resem waarschijnlijke paren zijn geïdentificeerd, kan men de kisten met fragmenten uit de opslagruimte halen en nakijken welke er écht passen. Gezien de grote hoeveelheid brokstukken is het niet mogelijk om ze allemaal bij de hand te houden. Daardoor duurt het altijd even voor de gewenste fragmenten gevonden worden. In het slechtste geval wordt er gewerkt op een (krachtige) desktop. Hierdoor is het nodig is om ofwel de namen en locaties van de fragmenten te onthouden, of een heleboel afbeeldingen af te drukken. Na het fysisch testen van de fragmenten moet men dan terug naar de desktop om de bevindingen in te geven. Gelukkig behoort een laptop ook tot de mogelijkheden hoewel applicaties als Browsematches en Griphos niet bepaald licht zijn. Het performantieprobleem wordt natuurlijk reeds een deel verholpen door het invoegen van een externe database. Ook kan men nu gemakkelijker met meerdere mensen en laptops tegelijkertijd werken aan de validaties wegens de automatische synchronisatie. Een stap verder zou zijn om een tablet te gebruiken. Er zijn reeds in het verleden experimenten geweest binnen het thera project om aanraakgevoelige omgevingen te maken en de hoop is dat tesamen met de resultaten van deze thesis er in de toekomst iets concreets van gemaakt kan worden.

Het versturen van afbeeldingen over het netwerk! 3D-modellen? render? (hoe dit efficient te doen? niet opnieuw uitvinden van het wiel, gebruik een goede codec)

Hoofdstuk 9

Besluit

Dit project heeft een lange weg afgelegd. De initiële beschrijving van de thesisopdracht ging over het maken van (innovatieve) gebruikersomgevingen om te helpen met het vinden van passende fragmentparen. De kernvraag luidde: “welke informatie is nodig om te kunnen zeggen of een paar wel of niet past? Hoe kan die weergegeven worden? Kan dit bijvoorbeeld voor 1000 paren tegelijkertijd?” Het zoeken naar dé nieuwe weergave van fragmentparen die een archeoloog in staat zou stellen om op efficiëntere wijze naar de enkele correcte paren in een zee van incorrecte voorstellen te zoeken leverde lange tijd niets op.

Vele evidente en minder evidente weergaven waren reeds vóór dit thesisproject te vinden in het thera project. Men kan de fragmenten op allerlei manieren bekijken. Behalve de gewone weergave in kleur valt de achterzijde te bezien, de contour alleen, de dikte, Interessant zijn de normalen van het oppervlak die kunnen gevisualiseerd worden om een beter idee te krijgen van het reliëf (zo kunnen onder andere krassen en borstelafdrukken duidelijk worden). Tevens kan de doorsnede van de plaats waar beide fragmenten elkaar raken bekeken worden, dit bleek zo nuttig te zijn dat het in Browsematches en dit thesisproject standaard aan de linkerkant van een paar wordt weergegeven. Eén mogelijke uitwerking van de thesis zou dus geweest zijn nog een alternatieve voorstelling te vinden die snelle en accurate beslissingen toelaat en bij voorkeur zeer compact (1000'en paren tegelijk) is. Helaas is het op dit vlak qua ideeën nooit verder gekomen dan incrementele verbeteringen op de huidige resultaten (bvb. niet-lineaire doorsneden van fragment visualiseren).

Grosso modo vallen er twee aanpakken te onderscheiden om het identificeren van paren te verbeteren: ofwel weet een algoritme de paren te rangschikken op zo'n manier dat de correcte paren eerst te zien zijn óf men kan ervoor zorgen dat (zeer) grote deelverzamelingen in korte tijd kunnen beoordeeld worden (hoge doorvoer). Anders gezegd: gericht zoeken naar paren of ze allemaal afgaan.

De tweede aanpak krijgt echter te maken met een fundamentele limiet gebaseerd op de informatietheorie.

De twee aanpakken zijn niet logisch onderling uitsluitend, maar hoe succesvoller de eerste aanpak is des te minder impact heeft de tweede. Immers als het overgrote

deel van van de correcte paren op een rij kan gezet worden zullen de weinige slechte paren het reconstructiewerk niet lang verhinderen zelfs als er helemaal niet virtueel nagekeken wordt. Vice versa geldt dit ook: als alle voorstellen op een redelijke tijd virtueel beoordeeld kunnen worden, moeten ze niet noodzakelijk in een goede volgorde staan. Kan dit echter wel? Misschien is de zoektocht naar een methode om zo snel mogelijk met het menselijke oog te verifiëren wel mislukt omdat er . Het is lastig om steeds één per één te moeten werken zoals in Griphos, maar na een zeker punt zijn er efficiëntieverliezen. Hoe compacter de data kan voorgesteld worden, hoe gemakkelijker het in een getal kan omgezet worden. In de limiet

Het verschil is dat terwijl de sorteeraanpak theoretisch niet moet onderdoen voor een mens die alle brokstukken met de hand aan elkaar probeert te passen¹, er wél een limiet staat op de “doorvoersnelheid” van het correct/niet-correct proces. Het komt erop neer dat

maar er is een belangrijk verschil: er is een duidelijke limiet aan het “zo veel mogelijk tegelijkertijd principe”. Eenmaal het zo klein wordt dat er comfortabel duizenden op een scherm passen, is het ook klein genoeg om accuraat naar een getal omgezet te worden (of het is reeds een getal). simpelweg door de rekening te houden met de mogelijke informatieinhoud van een klein aantal pixels. (uitlet over throughput limiet). Gezien het feit dat er een limiet is aan deze de maximale Hiermee rekening houdend en zien dat er miljoenen paarvoorstellen zijn

Het feit blijft dat er nog steeds geen visualisatie is die compact genoeg en toch de nodige zekerheid geeft dat een paar correct dan wel niet correct is. Een incorrect paar vinden is echter niet zo nuttig als het vinden van een incorrect paar. Dit is zo omdat

Misschien is het niet zo gemakkelijk om te zien welke paren correct zijn, maar is het wel mogelijk om snel te zien welk zeker niet correct zijn. Dan stelt zich het probleem dat er een manier moet zijn om deze snel te selecteren en te onderscheiden van degenen die dit niet zijn.

Andere beloftevolle insteken, zoals het gebruiken van contextinformatie en menselijke input om een virtuele classifier te trainen worden reeds onderzocht door onderzoekers Antonio Garcia Castaneda (Clusters) en Tom Funkhouser (Machine Learning) respectievelijk.

Mijn suggestie is: dynamische herberekening met procesverloop. In de realiteit wordt de finale conclusie over een voorstel gemaakt als men de fysieke brokstukken op de aangegeven plaats aan elkaar zet en ze “klikken”. Door erosie is de klik op zich niet altijd 100% vast waardoor het gebeurt dat zelfs met de fragmenten in de hand een amateur nog altijd geen sluitende conclusie kan geven. Evenwel is het één van de krachtigste controles en zijn er niet veel voorstellen die twijfelgevallen blijven. Misschien is (een deel van de) oplossing om de weergave dynamisch te maken en zo te pogen de stabiliteit van het voorstel te meten. Uit ervaring is gebleken dat mensen

¹Gegeven dat de data volledig genoeg is, resolutie is belangrijk. Dit valt te zien door in te beelden dat de resolutie zeer laag is en alle fragmenten gedigitaliseerd worden als perfecte kubussen van verschillende grootte.

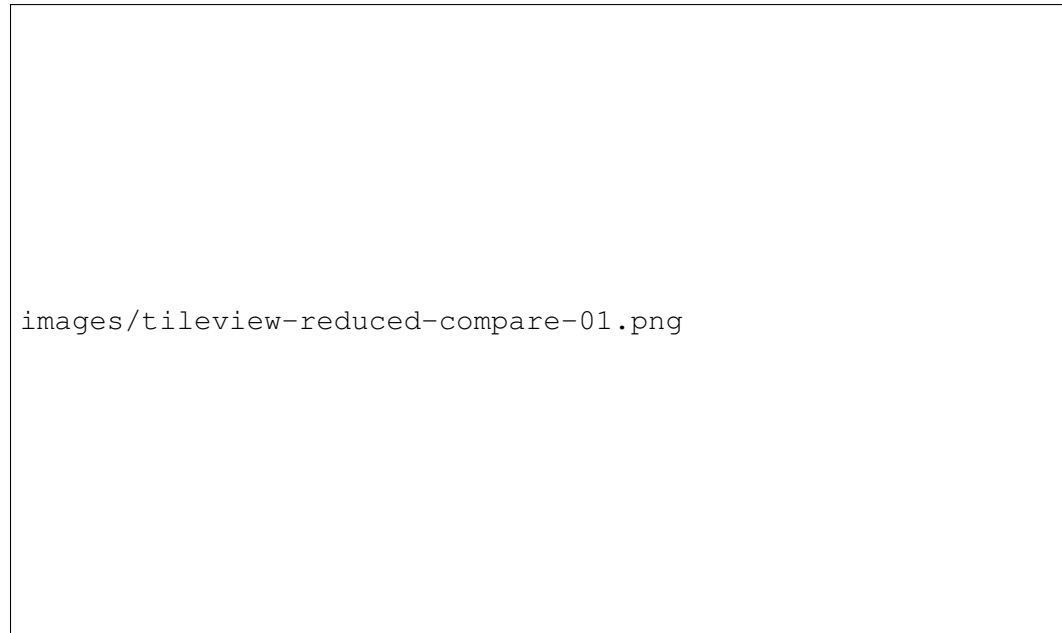
snel kunnen opmerken wanneer een voorstel **potentieel** heeft, maar niet in een oogopslag kunnen beslissen over de juistheid. Vaak blijft er zelfs na een betere kijk op de statische virtuele beelden nog genoeg twijfel over: noch de gewone visualisatie nog de beschikbare alternatieven geven uitsluitel. In acht nemend dat de automatische passers van het thera project rekening moeten houden met computationele efficiëntie en daarom niet te granulair kunnen zoeken [3], is er dikwijls nog wat ruimte voor verbetering. Een manier om dit op te merken is dat de doorsneden van paren die correct blijken te zijn vaak nog delen vertonen waar de volumes van beide brokstukken elkaar snijden. De erosie die plaatsvindt op de fresco's die onderzocht worden is echter puur subtractief, er zet zich geen extra materiaal vast op de brokstukken. Er is dus een goede kans dat een juist voorstel verder kan geoptimaliseerd worden.

[insert image v. doorsnede op linker of rechterkant met uitleg van de kleuren]

Interessante voostellen zouden daarom kunnen gebruik maken van een adaptieve versie van de passer die, beginnend van de originele positie, heel granulair kan zoeken. Wat voor miljoenen combinaties te kostelijk is, is voor een enkel voorstel slechts een peulenschil. Dit proces van iteratieve verbetering kan getoond worden aan de gebruiker, door een tijdsverloop van de doorsnede van het paar tijdens de optimalisatie weer te geven. Het komen en gaan van gebieden waar de geometrie van de fragmenten snijdt en waar ze verder uit elkaar gaat kan op zich reeds informatief zijn, maar het eindpunt is cruciaal. Na een volledige optimalisatie kan er een beeld getoond worden van lichte verschuivingen van het optimale punt. De hypothese is dat — als de resolutie van de 3D-opname voldoende is — de doorsnede van een paar dat in de realiteit klikt herkenbare patronen zal vertonen als het verschoven wordt. Hierover valt te verstaan dat er weerstand zal zijn in de vorm van snijdende geometrie. Er wordt hier abstractie gemaakt van enkele belangrijke implementatiedetails, zoals de richting waarin verschoven wordt. Dit kan gaan van eenvoudigweg de richting parallel kiezen aan de scheidingslijn tussen de uiterste punten, tot een complexe fysische simulatie die de fragmenten als rigide lichamen voorstelt en een combinatie van drukkende en schuivende krachten uitoefent. In de eerste vorm kan de weerstand “gemeten” worden door de gebieden van snijdende geometrie te sommeren. De tweede vorm laat toe de frictie onrechtstreeks te meten aan de hand van bijvoorbeeld verwachte tegenover reële snelheid in een bepaalde richting, of beter nog de kracht die nodig was om een verschuiving te krijgen. Het is onduidelijk of de toegenomen complexiteit van een fysische simulatie in verhouding staat met de resultaten.

Natuurlijk kan een aspect van deze beweging ook opgenomen worden als een karakteristiek in lerende algoritmen, men zou het “stabiliteit” kunnen noemen. Dit is een voorbeeld van een eigenschap die als discriminator kan gebruikt worden op voorstellen die reeds goed of interessant te noemen zijn, een slecht voorstel zou namelijk veel stabiliteit kunnen vertonen door de vele intersecties die zelfs na optimalisatie overblijven (afhankelijk van de precieze methode waarop men stabiliteit meet). Er zijn ook goede voorstellen waar de stabiliteit laag kan zijn doordat het materiaal teveel is afgevlakt of de breuk gewoonweg te rechtlijnig is. De stabiliteit lijkt dus

geen gouden graal te zijn. Daarom wordt hoe langer hoe duidelijker dat net zoals een mens meerdere eigenschappen in rekening brengt bij het oordelen, een computer dit ook moet doen.



Figuur 9.1: De doorsnedes van goede paren tegenover die van slechte paren. Een mens kan met ervaring leren om hieruit reeds veel af te leiden. Yassine Ryad bewees dat een algoritme kan gemaakt worden dat deze informatie in een rangschikking omzet. [1]

Waarschijnlijk komt dit omdat het niet de juiste insteek is. Er is een maximale informatiedensiteit. Het bekijken van een enorme hoeveelheid aan paren tegelijkertijd. Niettemin zou het kunnen dat er inderdaad een visualisatie bestaat die een optimale relevante informatiedensiteit bezit. Een voorbeeld van een dense voostelling is een intensiteitsmap van zoveel mogelijk fragmentparen (stel 1 pixel per paar) waar de intensiteit gelijkgesteld wordt aan de beoordeling van een automatische herkenner of een combinatie van meerdere herkenners en menselijke input (cfr. Machine Learning)[20]. Maar hier kan een archeoloog niet zoveel meer uit leren dan gewoon te sorteren op deze karakteristiek en in die volgorde paar na paar na te kijken. Als een karakteristiek werkelijk een onderscheid kan maken tussen correcte en niet-correcte paren zal dit in ieder geval moeten gebeuren. Dit soort dense visualisaties heeft dus vooral nut voor de ontwikkelaars van de karakteristieken zelf, zij kunnen een beter overzicht krijgen van de verdeling binnen de set van alle paren. Het passen van fragmenten is een complexe taak, maar niet zo complex dat het onmogelijk lijkt om een getal te plaatsen op de relatieve “goedheid” van een paar. Het zoeken naar zo’n

getal of karakteristiek is daarom op zich een waardevolle bezigheid. Dat was echter niet het onderwerp van deze thesis, maar van een andere die in dezelfde tijd aan de K.U.Leuven werd gemaakt binnen het thera project [referentie thesis yassine].

In het kort zou er dus een weergave moeten gevonden worden die:

- Compact is
- De relevante informatie bevat voor het beslissen over de correctheid van een paar
- Niet sorteerbaar is (dan is het probleem opgelost)
- Menselijke verificatie nodig heeft

Een goed voorbeeld van een voorstelling die compacter is dan het voorstellen van de paren zelf en niet sorteerbaar is, is het weergeven van de doorsnede.

Geen revolutionaire ideeën voor visualisaties die een thesis kunnen vullen en een heleboel reeds lopende projecten op alle vlakken, wat nu gedaan? Uit een kijk op de deficiënties van het thera project bleek dat er aan nieuwe delen werd gewerkt maar de infrastructuur niet optimaal was om alles aan elkaar te knopen, om de resultaten bij de gebruiker te brengen. Het project klaarmaken voor een eventueel nieuw paradigma en het toegankelijker maken van de informatie die reeds beschikbaar was, werden het nieuwe doel. Dit heeft het effect gehad dat er spijtig genoeg minder kon geconcentreerd worden op één bepaald onderwerp. Geen enkel onderdeel was apart groot genoeg om als onderwerp te dienen voor een volwaardig thesis.

Net zoals een gebroken fresco in feite een puzzel is, kan men het thera project zien als de som van vele delen die in elkaar passen. Dit thesisproject is bedoeld als een stuk dat een ander perspectief biedt op het geheel en het in staat moet stellen om meer en sneller resultaten te boeken. Het complementeert de bestaande aanpakken en maakt meteen ook de weg vrij voor toekomstige uitbreidingen. Het zorgt ervoor dat de resultaten van de automatische paarherkenning nog nuttiger gebruikt kunnen worden. Omdat het finale validatiewerk noodzakelijk door mensen moet gebeuren, is het geproduceerde werk waardevol: het kan niet zonder meer opnieuw door een algoritme gegenereerd kan worden. De voor dit thesisproject gemaakte componenten proberen er onder andere voor te zorgen dat het verlies van informatie zo min mogelijk voorkomt door robuuste dataopslag en synchronisatie mogelijk te maken.

Op het vlak van ontginning van nuttige informatie met nieuwe visualisaties en nieuwe manieren om de juiste patronen te ontdekken zijn er natuurlijk nog steeds vele opportuniteiten. Want — zoals opgemerkt in een recente paper over het thera project [citatie siggraph submission 2011] — het vinden van de juiste paren is zoals zoeken naar een naald in een hooiberg. Met elke nieuwe toevoeging aan de mogelijkheden van het platform is er de kans dat deze een manier is om de hooiberg te verkleinen, door te lichten met X-stralen of gewoonweg op een grote krachtige magneet in een

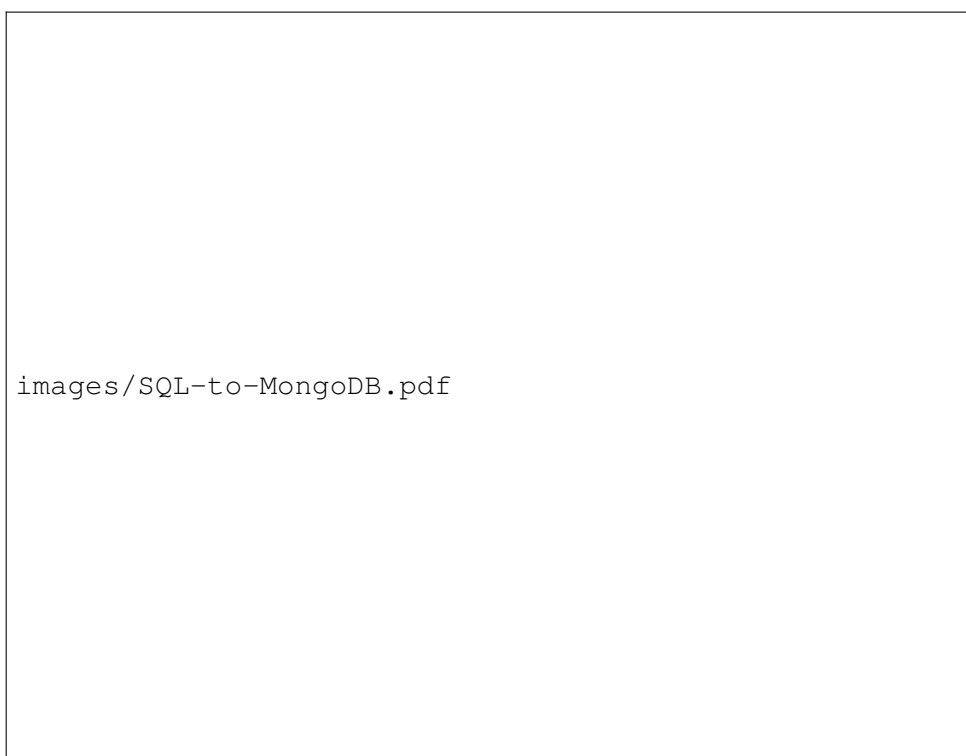
windtunnel te plaatsen. Naar deze laatste methode is iedereen natuurlijk op zoek. Tot dan is het zeker belangrijk dat men gemakkelijk kan experimenteren alsook bijhouden en opvragen welk deel van de berg reeds doorkamt is, waar de gevonden naaldrijke aders zitten en wat hun eigenschappen zijn. Misschien zijn de naalden immers niet van metaal. . .

Bijlagen

Bijlage A

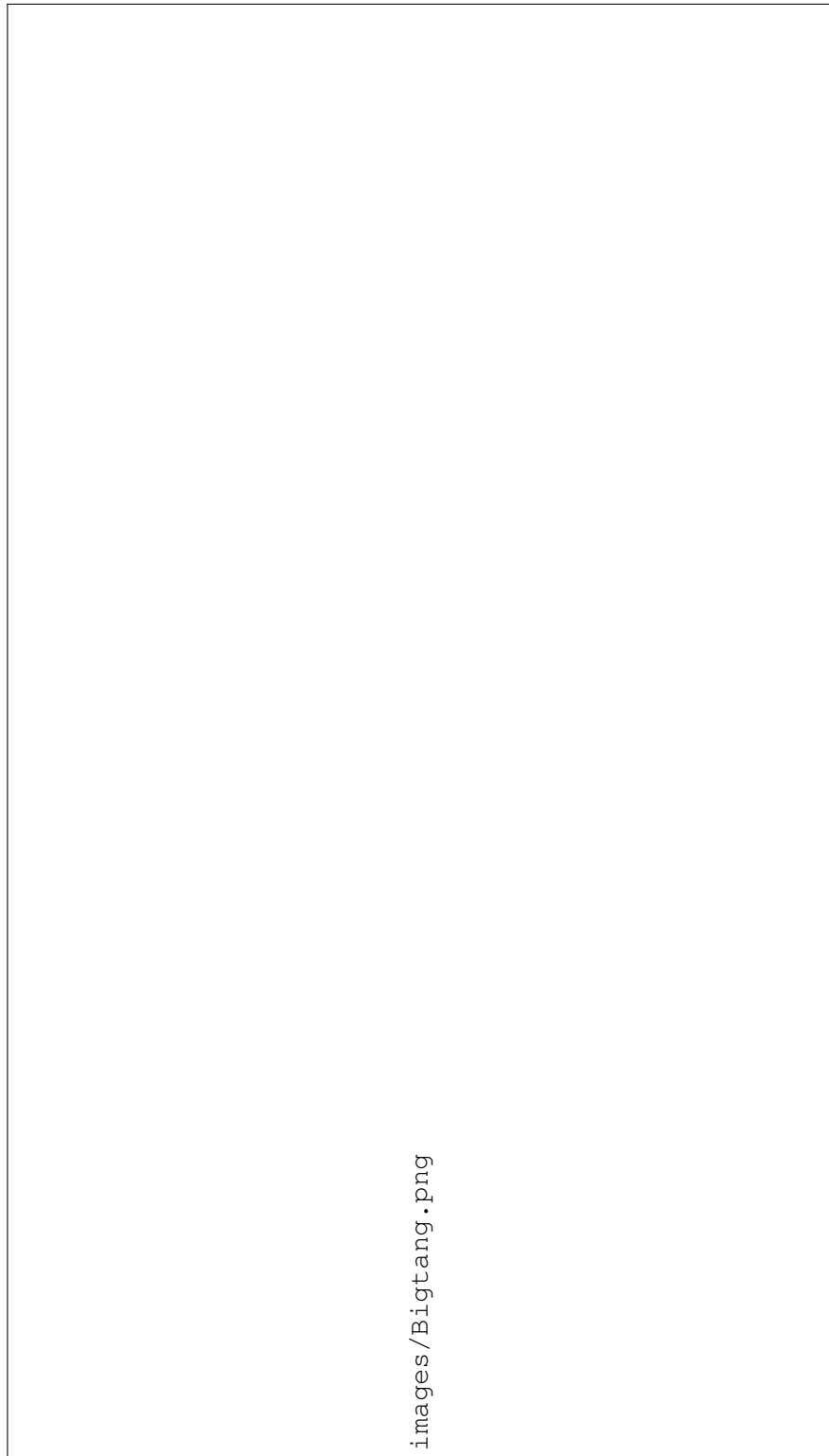
Extra afbeeldingen & schema's

A.1 Omzetting SQL naar NoSQL



Figuur A.1: Sommige NoSQL systemen — zoals MongoDB — beschikken zoals te zien valt in deze afbeelding over dezelfde analytische kracht als een SQL database. De prestatiekenmerken zijn echter verschillend. Afbeelding gebruikt met toestemming [2]

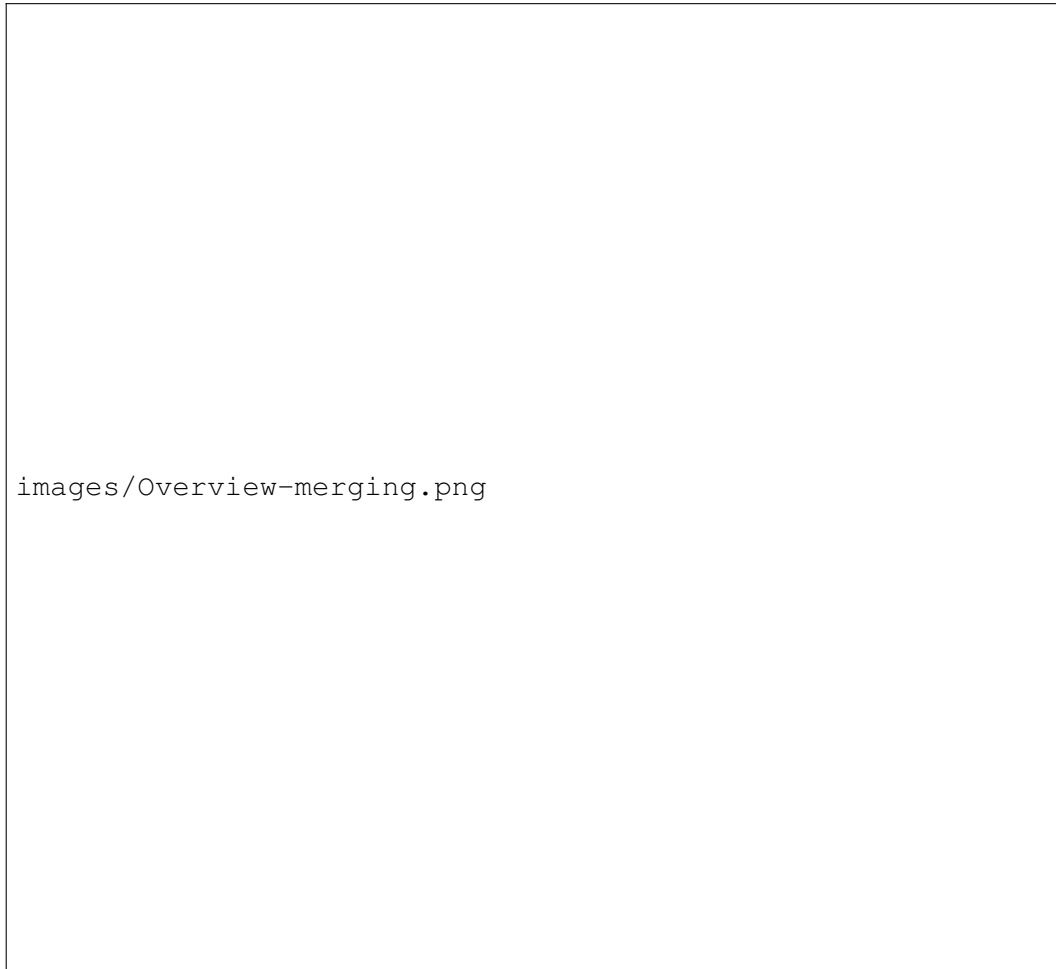
A.2 Overzicht van de applicatie



images/Bigtang.png

Figuur A.2: Een ruw overzicht van de componenten in het project en hoe het interageert met het reeds bestaande systeem.

A.3 Detail van de synchronisatiestructuur



Figuur A.3: Het synchronisatie-subsysteem, de componenten die overeenkomen met de stappen zijn aangeduid.

Bijlage B

Poster

poster/poster.pdf

Bibliografie

- [1] Y. Ryad, “Automatic classification of candidate fresco matches,” Master’s thesis, Katholieke Universiteit Leuven, 2011.
- [2] R. Osborne, “InfoGraphic: Migrating from SQL to MapReduce with MongoDB.” 2010.
- [3] B. J. Brown, C. Toler-Franklin, D. Nehab, M. Burns, D. Dobkin, A. Vlachopoulos, C. Dumas, S. Rusinkiewicz, and T. Weyrich, “A system for high-volume acquisition and matching of fresco fragments: Reassembling Thera wall paintings,” *ACM Transactions on Graphics (Proc. SIGGRAPH)*, vol. 27, Aug. 2008.
- [4] C. Toler-Franklin, B. Brown, T. Weyrich, T. Funkhouser, and S. Rusinkiewicz, “Multi-feature matching of fresco fragments,” in *ACM Transactions on Graphics (Proc. SIGGRAPH Asia)*, Dec. 2010.
- [5] J. A. Hoxmeier, P. D. and C. D. Manager, “System response time and user satisfaction: An experimental study of browser-based applications,” in *Proceedings of the Association of Information Systems Americas Conference*, pp. 10–13, 2000.
- [6] B. Shneiderman, “Response time and display rate in human performance with computers,” *ACM Comput. Surv.*, vol. 16, pp. 265–285, September 1984.
- [7] J. Nielsen, *Usability Engineering*. San Francisco: Morgan Kaufmann, 1994.
- [8] J. Spolsky, “User interface design for programmers.” <http://www.joelonsoftware.com/uibook/fog0000000249.html>, 2001.
- [9] J. Deacon, “Model-view-controller (mvc) architecture.” <http://www.jdl.co.uk/briefings/MVC.pdf>, 2009. [15/03/2011].
- [10] Nokia, “Qt online reference documentation.” <http://doc.qt.nokia.com/>, 2011.
- [11] J. Green, “A comparison of the relative performance of xml and sql databases in the context of the grid-safe project,” October 2008.
- [12] E. F. Codd, *The Relational Model for Database Management, Version 2*. Addison-Wesley, 1990.

- [13] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of Databases*. Addison-Wesley, 1995.
- [14] Anonymous, “Cassandra documentation.” <http://wiki.apache.org/cassandra/DataModel>, 2011. [07/06/2011].
- [15] EnterpriseDB, “Replication, clustering and connection pooling.” http://wiki.postgresql.org/wiki/Replication,_Clustering,_and_Connection_Pooling, 2011.
- [16] Oracle, “Mysql cluster.” <http://www.mysql.com/products/cluster/>, 2011.
- [17] S. S. Bhati and R. James, “Efficient pagination using mysql,” Presented at the Percona Performance Conference 2009, 2009. [22/07/2011].
- [18] S. Khanna, K. Kunal, and B. C. Pierce, “A formal investigation of diff3,” in *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)* (Arvind and Prasad, eds.), Dec. 2007.
- [19] B. J. Brown, L. Laken, P. Dutré, L. V. Gool, S. Rusinkiewicz, and T. Weyrich, “Tools for virtual reassembly of fresco fragments,” in *International Conference on Science and Technology in Archaeology and Conservations*, Dec. 2010.
- [20] T. Funkhouser, H. Shin, C. Toler-Franklin, A. G. Castañeda, B. Brown, D. Dobkin, S. Rusinkiewicz, and T. Weyrich, “Learning how to match fresco fragments,” in *Eurographics Area Track on Cultural Heritage*, Apr. 2011.

Fiche masterproef

Student: Nicolas Hillegeer

Titel: Een uitbreidbaar platform voor het efficiënt reconstrueren van fresco's

Engelse titel: Een uitbreidbaar platform voor het efficiënt reconstrueren van fresco's

UDC: 621.3

Korte inhoud:

Hier komt een heel bondig abstract van hooguit 500 woorden. \LaTeX commando's mogen hier gebruikt worden. Blanco lijnen (of het commando `\par`) zijn wel niet toegelaten!

Thesis voorgedragen tot het behalen van de graad van Master in de
ingenieurswetenschappen: computerwetenschappen

Promotor: Prof. dr. ir. P. Dutré

Assessoren: Prof. dr. D. De Schreye
Dr. A. Lagae

Begeleider: Dr. B.J. Brown