HELSINKI UNIVERSITY OF TECHNOLOGY
Department of Computer Science and Engineering
Laboratory of Information Processing Science

Oskar Ojala ja Antti Saarinen

T-106.720 Ohjelmistotekniikan projekti
Tietokanta WWW-julkaisujärjestelmälle

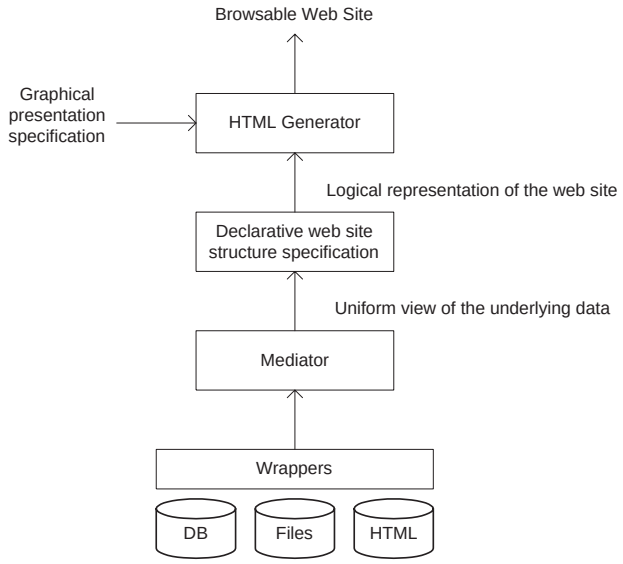
21. Joulukuuta 2003

Sisältö

1 Johdanto

WWW-sivustojen laajentuessa ja niiden sisällön muuttuessa ylläpitäjät joutuvat tekemään muutoksia niihin. Koska tämä urakka käsittää manuaalista HTML-sivujen kirjoittamista, se on varsin työlästä ja hidasta. Useimmiten tehtävä työ käsittää kuitenkin vain melko triviaalia sisällön päivittämistä eikä niinkään varsinaista sivuston rakenteen muuttamista. Tällöin työ soveltuu hyvin myös tietokoneen tehtäväksi.

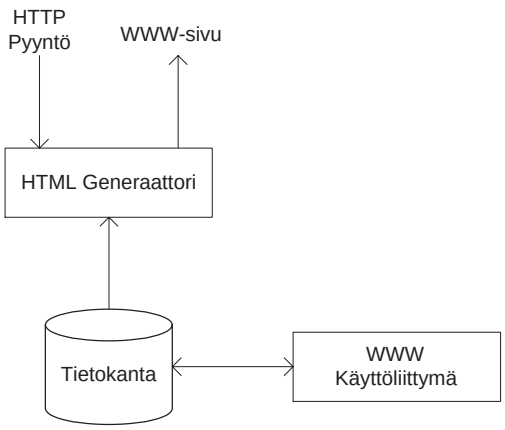
Keskeisessä asemassa on tällöin jonkinlainen tietojärjestelmä, joka huolehtii sivujen generoimisesta ja päivityksestä automatisoiden mahdollisimman suuren osan työstä. Yksinkertaisimmillaan tämä tarkoittaa WWW-sivun generoimista yksittäisen tietokantakyselyn tulosten perusteella; monimutkaisemmat tapaukset voivat käsittää kokonaisen sivuston luomisen. Florescu et al. [Florescu98] esittävät katsauksessaan tällaisen järjestelmän perusrakenteen kuvan 1 mukaisena. Siinä järjestelmä koostuu alimmalla tasolla erilaisesta datasta, josta järjestelmä luo deklaratiivisen kuvauksen. Lopuksi ko. kuvaus muunnetaan HTML-muotoon, jota voidaan palvella ulkopuolisille asiakkaille.



Kuva 1: Tietokantapohjainen WWW-järjestelmä.

Tässä työssä esitetään eräs tällainen järjestelmä keskittyen erityisesti sen tietokantaosuuden suunnitteluun. Koko järjestelmä on esitetty kuvassa 2. Pääasiallisesti se siis koostuu HTML-generaattorista, WWW-käyttöliittymästä sekä tietokantajärjestelmästä. Kuva on varsin pelkistetty, mutta tarkoitus onkin vain antaa yleiskuva kokonaisuudesta.

Tarkoitus siis on, että kun asiakas pyytää HTML-generaattorilta jotakin WWW-sivua, se hakee tietokannasta tarvittavan sisällön sekä sivun loogisen rakenteen, generoi niistä HTML-sivun ja palauttaa sen asiakkaalle. WWW-käyttöliittymä taas



Kuva 2: Toteutettava järjestelmä.

on tarkoitettu järjestelmän ylläpitäjille ja sisällöntuottajille; sitä kautta he voivat syöttää tietokantaan sisältöä. Tarkemmin sanottuna sisällöllä tarkoitetaan tässä tiedostoja, sivujen HTML-runkoja sekä muuta tietoa, kuten uutisartikkeleita, tuotekuvauksia ja niin edelleen. Tietokantajärjestelmän tulee tallentaa tämä tieto ja tarjota sitä HTML-generaattorille tämän niin pyytäessä.

Erityisesti järjestelmässä ovat mukana nk. asiakassovellukset, jotka konseptina tarkoittavat erilaisia sisältötyyppejä joilla on jotakuinkin yhdenmukainen esitystapa eri sivuilla. Niillä pyritään vähentämään tavallisimpien palvelujen rakentamistyötä; esimerkiksi jonkinlainen uutispalvelu löytyy lähes jokaiselta WWW-sivustolta. Rakentamalla järjestelmään suoraan mukaan uutispalvelu se voidaan sisällyttää jokaiselle uudelle sivustolle hyvin nopeasti.

2 Tietokantajärjestelmän vaatimukset

Koko järjestelmän toimivuuden kannalta tietokannan suorituskyky ja ominaisuudet ovat kriittisiä. Niinpä sitä suunniteltaessa erityistä huomiota pyrittiin kiinnittämään siihen, miten toiminnallisuus saataisiin helpoiten tarjottua muille järjestelmän osille, ja miten tietokannan sisäinen rakenne saataisiin mahdollisimman tehokkaaksi.

2.1 Toiminnallisuus

Pelkistetysti tietokantajärjestelmän tärkeimmät toiminnalliset vaatimukset ovat kyky vastaanottaa ja tallentaa tietoa WWW-käyttöliittymältä sekä palvella sitä HTML-generaattorille. Lähemmin tarkasteltuna tietokannan kuuluu voida tallentaa ainakin

- HTML-sivuja
- erilaisia tiedostoja, kuten kuvia ja dokumentteja
- suuri määrä erilaisia “sisältöpalasia” asiakassovelluksille

Lisäksi tarvitaan jonkin verran kaikille sivustoille yhteistä tietoa, kuten käyttäjänimet ja salasanat. Kaikki muut tiedot täytyy luonnollisestikin tallentaa sivustokohtaisesti. Tällöin samannimiset tiedostot eivät mene sekaisin eri sivustojen välillä ja toisaalta esimerkiksi samaa logotiedostoa voidaan käyttää useissa eri paikoissa sivustoa.

HTML-sivut voitaisiin tietysti sisällyttää myös muihin tiedostoihin, mutta ne on mainittu erikseen koska ne ovat yleensä melko pieniä ja ne on syytä saada järjestelmästä ulos erityisen nopeasti. Muut tiedostot taas saattavat olla useiden megatavujen kokoisia, eikä niitä tarvitse lähettää niin usein.

Sisältöpalaset taas ovat hyvin pieniä, muutamien kymmenien tai satojen tavujen kokoisia tietoyksiköitä jotka sisältävät kukin yhden loogisen kokonaisuuden, kuten uutisartikkelin. Ne sisältävät jotakin tietoa, joka voisi uutisartikkelin tapauksessa olla vaikkapa otsikko, varsinainen teksti ja viittaus kuvatiedostoon. Niitä täytyy voida käsitellä hyvin joustavasti ja tehdä erilaisia kyselyjä, kuten esimerkiksi “palauta kaikki 17. heinäkuuta 2004 jälkeen ilmestyneet uutiset”, “palauta viisi uusinta tuotekuvausta” tai “palauta kaikki tuotteet, joiden kuvauksessa on mainittu sana ’sukset’ ”. Kyselyt saattavat olla myös sovelluskohtaisia, kuten “palauta kaikki tuotteet joiden hinta on alle 100 euroa”. Tällainen kysely ei kuitenkaan ole relevantti uutisartikkeleille; tämä asettaa selkeästikin vaatimuksia rajapinnalle, jonka kautta kyselyt tehdään.

Huomioon on otettava myös se, että erilaisia sovelluksia olisi hyvä voida laatia lisää mahdollisimman joustavasti. Parasta olisi, mikäli niitä voisi lisätä ajonaikaisesti; tällöin tietoyksiköihin vaaditaan lisää semantiikkaa, koska HTML-generaattori ei lähtökohtaisesti voi tietää, minkälaisia sovelluksia kulloinkin on käytössä ja millaisia hakuja ne sallivat tehtävän itsestään.

2.2 Suorituskyky

Koska HTML-sivuja täytyy tarvittaessa generoida dynaamisesti HTTP-pyynnön tullessa, täytyy koko järjestelmän toimia rivakasti jotta asiakkaan tarvitsisi odottaa sivua mahdollisimman vähän. Mikäli sivut ovat lyhyellä aikavälillä staattisia, mikä on yleisin tapaus, niitä voidaan tallentaa erilaisiin välimuisteihin HTML-generaattorin päässä eikä niitä niin ollen tarvitse generoida koko ajan. Valmius sellaiseenkin täytyy kuitenkin olla mahdollisimman hyvä.

Suorituskyky asettaa tiettyjä rajoituksia kannan sisäiselle rakenteelle. Taulujen rakenteen pitää olla soveltuva nopeille hauille, ja suuret tiedostot saattavat vaatia erityistoimenpiteitä. Erityisesti niiden kohdalla on vältettävä ylimääräistä kopioimista järjestelmän sisällä, liikuttaen jonkinlaista viittausta tietoon ja lukien tavut konkreettisesti johonkin puskuriin vasta lähetettäessä.

2.3 Hajautus

Järjestelmän on oltava myös hajautettavissa useille koneille. Oleellisesti tietokannan näkökulmasta tämä tarkoittaa sitä, että tietokannan pitää olla eri koneella kuin järjestelmän muut osat. Selkeä syy hajautuksen vaatimiselle on suorituskyvyn skaa-

lautuminen; jos koko järjestelmää täytyy ajaa samalla koneella niin suorituskyvylle muodostuu nopeasti pullonkauloja niin suoritintehon, muistin kuin levyn lukemisenkin vuoksi. Myöskin tietoturvan kannalta on parempi, jos tietokanta voidaan saada eri koneelle. Tällöin siellä olevan sensitiivisen informaation suojaaminen esimerkiksi palomuurilla on helpompaa.

3 Toteutusvaihtoehtoja

Tietokantajärjestelmää suunniteltaessa täytyy tehdä useita ratkaisuja, jotka vaikuttavat merkittävästi lopputulokseen. Tässä kappaleessa esitetään keskeisimmät toteutusvaihtoehdot sekä analysoidaan niiden etuja ja haittoja.

3.1 Rajapinnat

Järjestelmän muiden osien kannalta oleellisin osa tietokantaa on rajapinta, jonka läpi se tarjoaa palveluitaan. Mikäli rajapinta tarjoaa hyvin korkean tason palveluita, on muiden osien helpompi käyttää sitä; toisaalta, tällöin tietokantaosuudesta tulee varsin suuri. Välttämättä muut osat eivät tarvitse kovinkaan monimutkaista rajapintaa, mutta jonkinlainen abstrahointi on kuitenkin hyödyllistä.

Suunnitteluvaiheessa havaittiin oleellisesti kolme eri vaihtoehtoa, joiden välillä ratkaisu tehtiin. Tässä keskitytään rajapintaan lähinnä asiakassovellusten kannalta, koska muilta osin rajapinta on melko suoraviivaisesti kirjoitettavissa.

SQL:n kaltainen rajapinta Yksinkertaisin vaihtoehto olisi tarjota melko suoraan SQL:n kaltaista rajapintaa ulos tietokannasta. Tämä tekisi tietokantaosuudesta varsin laihan toteutuksen, jossa SQL-kyselyt oleellisesti vain ohjattaisiin alla olevalle tietokannalle. Tämän vaihtoehdon hyviä puolia ovat yksinkertaisuus, suorituskyvyn vähäinen tarve tietokannan vaatiman suorituskyvyn lisäksi sekä tietokannan sovellusriippumattomuus. Jos sisään otetaan SQL:ää, ei tietokannan tietenkään tarvitse tietää erilaisista sovelluksista tai niiden tietoyksiköiden semantiikasta.

Haittapuoliakin tosin on: järjestelmän muut osat joutuvat nyt hoitamaan suuren osan kaikesta tietokantatoiminnallisuudesta, mukaanlukien uusien taulujen luomisen, taulujen rakenteen päättämisen sekä kyselyjen muodostamisen. Lisäksi käytännöllinen SQL ei ole aina tietokantariippumatonta, joten muutkin osat saattaisivat tulla jossain määrin riippuvaisiksi valitusta tietokannasta. Käytännössä siis koko asiakassovelluksien vaatima tiedon tallennuslogiikka ulkoistettaisiin järjestelmän muille osille.

Sovelluskohtaiset rajapinnat Toinen vaihtoehto on laatia oma rajapinta jokaiselle asiakassovellukselle. Tällöin esimerkiksi uutispalveluille tarjotaan yksi rajapinta, jonka kautta tietokantaan voidaan tallentaa uutisartikkeleita sekä hakea niitä. Tällöin jokaiselle sovellukselle voidaan antaa juuri sen tietoyksiköiden semantiikalle sopiva rajapinta, mikä on muiden osien kannalta tietysti helpompaa: samaa hakurajapintaa ei tarvitse sovittaa sekä uutisille että tuotekuvauksille. Tällöin uutisia

voidaan hakea tietolähteen mukaan ja tuotteita hinnan mukaan; kumpikaan näistä hauistahan ei olisi relevantti toisessa tapauksessa.

Sovelluskohtaiset rajapinnat tuovat kuitenkin tiettyä staattisuutta järjestelmään. Rajapinnat täytyy olla tiedossa käännöshetkellä ja sekä HTML-generaattorin että WWW-käyttöliittymän tulee jo ohjelmakoodissa hakata kiveen käytetyt rajapinnat. Uuden asiakassovelluksen lisääminen vaatii siis uuden rajapinnan ja siten muutoksia kaikkiin järjestelmän osiin. Ajonaikaisesta lisäämisestä ei siis kovinkaan helposti tule mitään. Sovelluskohtaiset rajapinnat myöskin helposti levittäisivät sovelluskohtaisuuden joka puolelle järjestelmää, siten että HTML-generaattorinkin olisi hankalahkoa käsitellä eri sovelluksia jotenkin geneerisesti.

Geneerinen korkeamman tason rajapinta Kolmas vaihtoehto on laatia riittävän geneerinen rajapinta siten, että sama rajapinta sopii kaikille eri sovelluksille. Tällöin koko järjestelmästä saadaan enemmän tai vähemmän sovellusriippumaton, ja uusien sovellusten lisääminen ei vaadi muutoksia tietokantajärjestelmään. Samalla tietokantajärjestelmä yksinkertaistuu, kun jokaiselle sovellukselle ei tarvitse kirjoittaa omaa käsittelijää.

Toisaalta tämä tarkoittaa sitä, että kaikkien sovellusten tietoyksiköiden täytyy istua johonkin ennalta määrättyyn muottiin ja niistä tehtävien hakujen tulee olla jotenkin yhteneviä tai sovelluskohtaisesti ajonaikana parametrisoitavissa. Esimerkiksi voitaisiin vaatia, että jokaiseen tietoyksikköön kuuluu päivämäärä, ja tämän tiedon perusteella hakuja voidaan tehdä. Ko. päivämäärän merkitys voisi tietenkin olla erilainen eri sovelluksilla: uutisartikkelille se on julkaisupäivä ja tuotekuvaukselle vaikkapa tarjouksen viimeinen voimassaolopäivä.

3.2 Tiedon tallennus

Jotta tiedot saataisiin järjestelmästä nopeasti ulos, täytyy sen sisäisen rakenteen olla suunniteltu sitä silmälläpitäen. Keskeisiä kysymyksiä tähän liittyen ovat tietokannan taulujen rakenne sekä tiedostojen tallennus. Tietokannan taulujen tulisi olla sellaisia, että tarvittavat tiedot löytyvät nopeasti eikä niissä ole ylimääräistä tavaraa. Tiedostojen tallennuksessa oleellinen kysymys on, minne ne ylipäätään tallennetaan - tietokantaan vai jonnekin muualle? Lisäksi niiden liikuttelu järjestelmän eri osien välillä täytyy olla joustavaa ilman, että tiedostosta täytyy tehdä kopioita joka kerta kun se välitetään jonkin rajapinnan yli jonnekin päin järjestelmää.

3.2.1 Taulujen rakenne

Tietokannan taulujen rakenteen osalta tärkein asia on se, kuinka varsinaiset sovelluskohtaiset tietoyksiköt tallennetaan. Niiden lisäksihän tietokantaan tallennetaan myös erilaista muuta tietoa lähtien asiakkaiden nimistä ja puhelinnumeroista, mutta niiden tallennusmuoto ei juurikaan vaikuta tietokannan suorituskykyyn tai ylläpidettävyyteen. Sen sijaan tietoyksiköitä joudutaan hakemaan usein ja paljon, tyypillisesti vielä jollakin hakukriteerillä rajaten. Tämä aiheuttaa sen, että mikäli taulujen rakenne ei mahdollista nopeita hakuja, tietokannasta tulee helposti pullonkaula koko

järjestelmän suorituskyvylle. Erilaisia taulurakenteita mietittiin sekä suorituskyvyn, ylläpidettävyyden että erilaisten hakumahdollisuuksien näkökulmista.

Kysymys taulujen rakenteesta riippuu oleellisesti myös siitä, millainen rajapinta tietokantajärjestelmästä tarjotaan ulospäin. Jos rajapinnat ovat sovelluskohtaisia, täytyy taulujenkin olla sellaisia koska tallennettava data ei ole mitenkään yhtenäistä niin että useiden eri sovellusten tietoja edes voitaisiin tallentaa samaan tauluun. Sen sijaan mikäli kaikilla sovelluksilla on jokin sovittu samanlainen tietoyksikkömalli niin useiden sovellusten yhteiset taulut tulevat kysymykseen.

Yksi suuri taulu Suoraviivainen ratkaisu on tallentaa kaikkien sovellusten tietoyksiköt yhteen ja samaan tauluun. Tällöin taulussa täytyy tietenkin olla myös sarake joka kertoo mille sovellukselle ko. tietoyksikkö kuuluu. Tämän vaihtoehdon etuna on ainakin yksinkertaisuus, koska uusia sovellustauluja ei tarvitse luoda tai poistaa dynaamisesti vaan koko tietokanta toimii koko ajan samoilla kiinteillä tauluilla. Ongelmia aiheuttaa kuitenkin skaalautuvuus, kun tietokannan kasvaessa taulu paisuu todella suureksi ja jokaisella pienelläkin haulla saatetaan joutua läpikäymään koko taulu.

Oma taulu jokaiselle sivustolle Eräs tapa tallentaa tietoyksiköt on luoda jokaiselle sivustolle oma taulu, johon on tallennettu kaikki sen käyttämien sovellusten tietoyksiköt. Tällöin taulut pysyvät kohtuullisen kokoisina vaikka järjestelmässä olisi suurikin määrä erilaisia sivustoja. Sitä vastoin mikäli yhdellä sivustolla on suuri määrä sovelluksia joissa on paljon tietoyksiköitä, tämäkin taulu kasvaa suureksi. Ongelma ei kuitenkaan ole niin paha kuin jos kaikki sivustot olisivat samassa taulussa. Edelliseen vaihtoehtoon verrattuna tässä on ikään kuin tehty yksi hakuvaihe valmiiksi, koska haut koskevat aina vain yhtä sivustoa; kun haku tehdään tauluun, jossa on vain sen sivuston tietoja, ei mitään turhia taulurivejä käydä läpi.

Oma taulu jokaiselle sovellukselle Taulut voidaan rakentaa myös siten, että jokaisella sovelluksella on yksi taulu johon kaikkien eri sivustojen ko. sovellukseen liittyvät tietoyksiköt tallennetaan. Tällöin voidaan sallia sovelluskohtaiset tietoyksikkörakenteet, kun eri sovellusten tietoja ei tarvitse voida tallentaa samaan tauluun. Kutakin sivustoa päivitettäessä, eli uusia tietoyksikköjä lisättäessä, päivitykset jakaantuvat useille eri tauluille. Tällöin myös tietokannan sisäisessä optimoinnissa voidaan tehdä valintoja taulun tyypin perusteella: esimerkiksi uutistaulu todennäköisesti päivittyy tiheämmin kuin tuotekuvasto, jolloin tuotekuvasto voidaan asettaa hitaammalle levylle. Tosin edelleen sivustomäärän kasvaessa tämäkin taulu kasvaa melko suureksi.

3.2.2 Tiedostojen tallennus

Karkealla tasolla järjestelmän tallentama tieto on joko HTML-muotoista dataa sivujen esitystä varten tai binääritiedostoja. Binääritiedostot käsittävät sivuilla olevat kuvat ja erimuotoiset dokumentit esim. PDF tai Microsoft Word, joita web-

sivustoilta voidaan ladata. Käsitteenä "dokumentti" on laaja; käyttäjä saattaa haluta web-sivustolleen reaaliaikaisen videovirran, musiikkipätkän tai yksinkertaisen muotoillun tekstidokumentin. Kaikkia erilaisia tiedostomuotoja, joita käyttäjä haluaa, ei luonnollisesti pysty ennakoimaan, mutta koska HTTP-protokolla pyytää kaikki linkitetyt tiedostot (myös kuvat) omina pyyntöinään, niin tallennusjärjestelmän täytyy vain kyetä tallentamaan tiedostot sellaisenaan ja tarjota ne binääridatana järjestelmän muille osille, sillä toisin kuin HTML-pohjaista dataa tiedostoja ei tarvitse käsitellä ennen niiden lähettämistä web-clientille.

Perinteisesti on katsottu käyttöjärjestelmän tarjoavan hierarkkisen tiedostojärjestelmän sopivan hyvin tiedostojen tallennukseen. Se tarjoaa tietyt perusominaisuudet: hakemistot, oikeuksien hallinnan ja tiedoston koon ja muutospäiväyksen tallennuksen. Tiedostojärjestelmä ei kuitenkaan tarjoa transaktioiden hallintaa ACID-periaatteen, toisin kuin transaktioita tukeva relaatiotietokanta. Tietokannalla rivejä voi lisätä ja poistaa atomisesti ja käyttämällä transaktioita voidaan tehdä monimutkaisia operaatioita ilman, että tietokanta jää epäkonsistenttiin tilaan. Tiedostojärjestelmässä mikään ei estä kahta ohjelmaa kirjoittamasta saman tiedoston samaan kohtaan samaan aikaan. Käyttöjärjestelmä tarjoaa yleensä erilaisia lukitusoperaatioita tiedostojen yhtäaikaiseen käyttöön, mutta tietokannassa nämä toimivat ohjelmoijalta suhteellisen näkymättömissä, tehden ohjelmoinnin helpommaksi ja koodin lyhyemmäksi. [Maciaszek90]

Vertaillessa tiedostojen tallennusta tietokantaan ja tiedostojärjestelmää nousee väistämättä esiin kysymys suorituskyvystä ja joustavuudesta. Nykyaikaiset tiedostojärjestelmät pystyvät käsittelemään isojakin tiedostoja. MySQL tarjoaa blob-tyyppisen attribuutin, johon pystyy tallentamaan aina 4 gigatavua tietoa. Johtuen tietokantahallintajärjestelmän rakenteesta niin tämä ei käytännössä ole ainoa rajoitus: tiedoston pitää mahtua muistiin jos se aiotaan ladata ja lähettää eteenpäin. Pikaisella arvioinnilla tullaan helposti siihen tulokseen, että suunnittelemallamme WWW-julkaisujärjestelmällä 32-bittinen muistiarkkitehtuuri ei riitä käsittelemään neljän gigatavun tiedostoja, joten se ja käytännön seikat (kuka haluaa ladata WWW:stä neljän gigatavun tiedoston?) tekevät tämän rajoituksen epäolennaiseksi meidän kannaltamme.

Yhtenä vaatimuksena järjestelmälle oli hajautettavuus, erityisesti mahdollisuus ajaa itse julkaisujärjestelmää yhdellä koneella ja tallentaa tieto toiselle koneelle. Tietokanta tarjoaa TCP/IP-yhteydellä automaattisesti tällaista hajautettavuutta, mutta tiedostojärjestelmä on useimmiten paikallinen. Jos halutaan käyttäätiedostojärjestelmää voi se pohjautua client-server -pohjaiseen tiedonsiirtoon verkon yli (esim. käyttäen FTP-protokollaa) tai RPC (Remote Procedure Call) -tyyppisellä levynjaolla (esim. NFS mutta periaatteessa myös SMB-jaolla jos Windowsin käyttö tulisi jostain syystä ajankohtaiseksi). Tapoja on useita, mutta FTP tarjoaa yksinkertaisen, hyväksi todetun ja yleisessä käytössä olevan protokollan, joka on helppo toteuttaa ja toimii LANissa hyvin. Vastaavasti NFS tarjoaa suuren joustavuuden, sillä ohjelmassa tiedostoja voidaan käsitellä ikäänkuin ne olisivat paikallisella levyllä, tehden ohjelmoinnin ja testauksen helpoksi. NFS on myös laajalti käytössä, myös isoissa organisaatioissa, joten sen toimintavarmuutta ja suorituskykyä voidaan pitää

riittävänä.

Viimeinen tiedostojen tallennukseen vaikuttava seikka on miten tiedostot halutaan hakea. Jos tiedostot tallennetaan tietokantaan saadaan kaikki tieto luontevasti samasta paikasta. Jos tiedosto tallennetaan levylle voi sen hakemisto- ja tiedostonimeen laittaa tietoa siitä (esim. mihin web-sivustoon se kuuluu) ja luontipäivämäärän voi pyytää tiedostojärjestelmältä. Kuitenkin tiedostolta vaaditaan järjestelmän puitteissa muutakin metadataa: tiedoston tyyppi, nimi, jolla tiedosto haetaan ja tiedostoa kuvaava otsikko (jota voidaan käyttää esim. hakuihin). Nimen, jolla tiedosto haetaan ja sivusto, johon se kuuluu voidaan tallentaa tiedostonimeen, mutta tyyppi ja etenkin vapaa otsikkoteksti ovat hankalasti tallennettavissa tiedoston nimeen. Näin ollen jos aiotaan käyttää tiedostojärjestelmää tiedostojen tallennukseen on luontevaa käyttää tietokantaa tiedostoon liittyvän metadatan tallennukseen.

4 Järjestelmän toteutus

Edellisen luvun vaihtoehdoista täytyi lopulta valita toteutettavat ratkaisut. Aina ei voitu valita yksikäsitteisesti parasta vaihtoehtoa, koska sellaista ei ollut, mutta kokonaisuudesta saatiin kuitenkin kohtuullisen käyttötarkoitustaan vastaava.

Käytännössä suurin osa toiminnasta tapahtuu tarjotun rajapinnan alla, missä sen toteuttavat oliot toimivat WWW-käyttöliittymän ja HTML-generaattorin palveluksessa. Asioiden pitämiseksi mahdollisimman yksinkertaisina suorituskyvystä tinkimättä päätettiin tarjota yksi ja yhteinen rajapinta ulospäin. Koska tietokanta- ja tiedostojärjestelmät, joissa varsinainen tieto säilytetään, tarjoavat jo valmiiksi synkronointi- ja transaktio-ominaisuuksia, vältytään sen manuaaliselta tekemiseltä tarjoamalla käyttöön useita yhteysolioita (sanalla yhteys viitataan tässä järjestelmän tietoliikenneyhteyteen tietokantaan ja etätiedostojärjestelmään). Tietokanta tallentaa asiakkaiden tiedot ja kaikki sovellusten tietoyksiköt, kun taas tiedostot ovat tavallisessa tiedostojärjestelmässä.

4.1 Rajapinnoitus

Ehkäpä oleellisin kysymys järjestelmän toteuttajan kannalta on tarjottavan rajapinnan valinta. Koska asiat haluttiin pitää yksinkertaisina ja sovellusriippumattomina sekä tietokannan sisällä että myös järjestelmän muissa osissa, päädyttiin valitsemaan yksinkertainen sovellusgeneerinen rajapinta.

Käytännössä rajapintaan tuli seuraavat metodit:

`addClient, removeClient`	Asiakkaiden (sivustojen) lisäys ja poisto
`addFile, removeFile`	Tiedostojen lisäys ja poisto
`getFile`	Tiedoston hakeminen
`createApplication`	Sovelluksen luominen
`createAppInstance`	Sovelluksen instantiointi johonkin sivustoon
`getAppInstances`	Käytössä olevien sovellusten haku
`insertAppItem, deleteAppItem`	Tietoyksikköjen lisäys ja poisto
`getAppItems`	Tietoyksikköjen haku tietyin kriteerein
`getAppItemHeaders`	Tietoyksikköjen haku, palauttaen vain otsaketiedot

Osa metodeista on melko itsestäänselviä, mutta muutamat kaipaavat lisähuomautuksia. Ensinnäkin, sama rajapinta saatiin palvelemaan kaikkia sovelluksia vaatimalla kaikkien tietoyksiköiden taipuvan samaan muottiin. Tämä muotti ei sinänsä rajoita millään tavalla tallennettavan datan määrää tai muotoa; tämä tietenkin tarkoittaa myös sitä, että tietokannalla ei voi olla mitään semanttista käsitystä tietoyksiköistä. Yleisin ratkaisu olisi tietysti ollut sallia tietoyksiköiden olevan määräämättömän pitkiä bittijonoja, mutta esimerkiksi uutisten ollessa kyseessä tietokanta ei voi hakea uusimpia uutisia tai tehdä mitään muitakaan hakuja. Tämän vuoksi bittijonon lisäksi jokaiselta yksiköltä vaaditaan myös otsikko, kaksi päivämäärää ja tunnisteteksti, joita kukin sovellus voi käyttää haluamallaan tavalla. Jos siis jokin sovellus ei määrittele päivämääriä ja tekee kaikki haut muilla kriteereillä, se voi jättää ne tyhjiksi. Rajoitus piileekin siinä, että mikäli jokin sovellus tarvitsisi kolme päivämäärää, niitä ei ole käytettävissä. Ongelma on kierrettävissä tekemällä haut kahden päivämäärän perusteella ja hylkäämällä osa yksiköistä tietokantajärjestelmän ulkopuolella sovelluskohtaisella kriteerillä - joskin tällöin tietokannasta joudutaan hakemaan ylimääräistä tietoa.

Toinen huomionarvoinen kohta on tiedostojen haku. Alunperin suunniteltiin, että suurista tiedostoista voitaisiin hakea vain viittaus (esim. hakemistopolku) jolloin vältettäisiin tiedoston turha kopiointi järjestelmän sisällä. Tästä ei kuitenkaan tulle ongelmaa, koska Javassa tiedostoista luettuja tavujonoja kuljetetaan luontevasti eräänlaisina viittauksina (`java.io.InputStream`). Lisäksi ko. ominaisuus olisi vaatinut tietokantajärjestelmältä jonkinlaista korkeamman tason lukitusta viittauksen käyttöaikana.

Edelleen tietoyksiköitä haettaessa selvittiin yhdellä metodilla niiden vaihtelusta huolimatta, koska ne oli pakotettu samaan muottiin. Rajapinnan käyttäjä kertoo vain tälle metodille, mistä sovelluksesta on kysymys ja millä kriteerillä hakutehdään, sekä antaa yhdestä kahteen parametriä (esim. näiden kahden päivämäärän välissä olevat yksiköt). `getAppItemHeaders` lisättiin tehokkuuden vuoksi, koska useilla sivustoilla esim. uutispalvelun ollessa kyseessä halutaan jonkinlaiselle alkusivulle uusimpien uutisien otsikot muttei niiden varsinaista sisältöä. Tämä metodi palauttaa vain kunkin tietoyksikön otsikon ja päivämäärät, eikä haaskaa aikaa varsinaisen datan, jota voi olla paljonkin, siirtämiseen.

4.2 Valittu tietokannan rakenne

Edellisessä kappaleessa kuvattu rajapinta määräsi pitkälti tietokannan sisäisen rakenteen. Koska jokainen sovellus tallensi tietokannan näkökulmasta samanlaista tietoa, sovelluskohtaisia tauluja ei tarvittu. Tietoyksiköt tallennettiin kuitenkin sovelluskohtaisesti, koska muutoin kantaan olisi saatu vain vähän todella suuria tauluja; oleellisestihan jakamalla data sovelluskohtaisesti vain tehdään yksi hakuvaihe valmiiksi, kun tietokannan ei tarvitse eritellä eri sovellusten tietoyksiköitä toisistaan. Joka tilanteessa kukin haku kuitenkin koskee vain yhtä sovellusta. Toiseenkaan äärilaitaan ei kuitenkaan menty tekemällä omaa taulua jokaiselle kunkin sovellustyypin ilmentymälle.

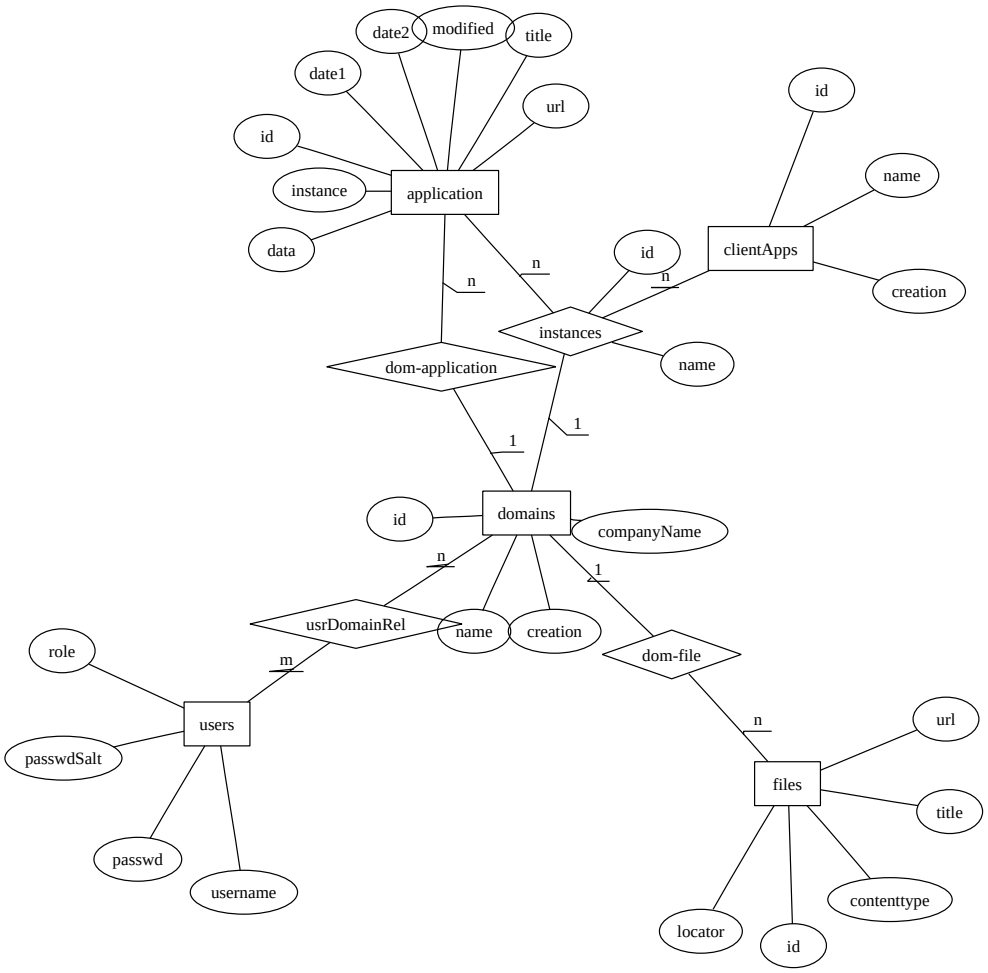
Muut kannan osat eivät vaikuta suorituskykyyn niin merkittävästi; näin ollen niiden taulujen rakenteita ei ollut mitään syytä erityisesti viilata. Kuvassa 3 näkyy lopputuloksena olleen kannan E/R-kaavio. Kaaviossa on esitetty users-taulu, vaikka käytännön toteutuksessa se on erillinen. Idea on siinä, että users-taulu toimii juuri kuin kaaviossa esitetty users-taulu, mutta asiakkaan vaatimuksena oli hoitaa autentikaatio erikseen, jolloin users-taulu voidaan siirtää toisen tietokantajärjestelmään. Luonnollisesti tämä edellyttää domain-taulun kopioimista, joten kaaviossa esitetty malli on "tyylikkäämpi" ja käytännössä sitä voi helposti käyttää tässä järjestelmässä. Järjestelmä tekee siis autentikaatiotietokannan ylläpidon varsin vapaasti toteutettavaksi käyttäjän tarpeista riippuen (kts. kohta 4.3).

Kannan rakenteessa tavoiteltiin normalisaatiota, pitäen kuitenkin mielessä että käytännön toimivuus on tärkeämpää kuin teoreettinen eleganssi. Suunnittelun jälkeen tietokannan taulut täyttävät seuraavat ehdot:

- jokaisella rivillä on sama määrä sarakkeita, ja ne sisältävät vain atomista tietoa
- mitkään sarakkeet eivät sisällä monisteista tietoa
- minkään sarakkeen tiedot eivät ole johdettavissa muiden sarakkeiden tiedoista
- taulu ei sisällä moniarvoisia riippuvuuksia

Normaalimuotoinen rakenteen suurin etu on tae siitä, että taulut eivät sisällä samaa tietoa moneen kertaan. Myöskin yleisesti ottaen taulujen normaalimuotoisuutta pidetään eleganttina ja tavoiteltavana asiana. Moninkertaisesti saman tiedon tallentaminen heikentää myös suorituskykyä, kun haettavissa tauluissa ja palautettavissa tuloksissa on ylimääräistä tietoa mukana.

Toisaalta tiukka normaalimuotoon pyrkiminen tekee lisäyksistä ja päivityksistä vaikeaa ja monimutkaiset kyselyt ovat tehottomia; tästä syystä joissain sovelluksissa on harkittava denormalisaatiota [Greenspun98]. Suunnitellussa tietokannassa ei ole mitään päivitysanomalioita; kaikki päivitykset pystytään tekemään yhdellä SQL-lauseella, joka ei sisällä alikyselyjä. Hauissa joudutaan tekemään liitoksia, mutta jokaisessa taulussa käytetään numeerisia indeksejä nopeuttamaan tätä ja hinta on pieni verrattuna denormalisoitujen taulujen päivitysanomalioihin. Numeeriset erityiset indeksit ovat ehkä suurin poikkeus "oikeaoppisesta" taulun muodostamisesta, mutta



Kuva 3: Tietokannan E/R-kaavio.

juoksevan indeksinumeron poistaminen olisi merkinnyt yksilöivien tietoattribuuttien käyttöä indekseinä ja nämä ovat käytännössä kaikki varchar-tyyppisiä, joiden pituus voi hyvinkin olla yli 100 merkkiä. Myös useamman attribuutin käyttö yhteen indeksiin tekee proseduraalisen ohjelmoinnin vaikeaksi, sillä indeksiä ei enää säätele yksi muuttuja, vaan yhdistetyyppi. Nämä poikkeukset huomioonottaen päästiin kuitenkin korkealle normalisaatioasteelle ja käytännössä tietokannan sisäiset rajoitteet (jopa verrattain rajoittuneella tietokantahallintajärjestelmällä kuten MySQL:llä) ja taulujen muoto pitävät pitkälti huolen siitä, ettei kantaan tule virheellistä tietoa, mikä on suuri etu tämän tyyppisessä järjestelmässä.

4.3 Poolaus

Suunnittelun edetessä ja ensimmäisiä protyyppejä kokeiltaesssa huomattiin, että koska HTML:ä generoivien moduulien, jotka ovat suoraan yhteydessä edustapalveli-

miin, on luonteva toimia siten, että oliot luodaan lennosta kun sivupyyntö edustalle tulee, on tietokantajärjestelmää sovitettava tähän vaatimukseen.

Käytännössä jokin Java Servlet saa HTTP-clientiltä HTTP-pyynnön, joka johtaa siihen, että tietokannassa olevia tietoja on joko haettava tai muutettava. Koska Servletit joutuvat luomaan ja instantioimaan käyttämänsä oliot jokaisella suorituskerralla, johtaa tämä tietonkantajärjestelmän näkökulmasta siihen, että tietokantarajapinnan toteuttava olio luodaan ja alustetaan jokaisen Servletin suorituksen yhteydessä.

Tietokantajärjestelmälle uuden olion luominen jokaisen pyynnön yhteydessä toimii huonosti; JDBC:n on alustuksen yhteydessä luotava TCP/IP:ä käyttäen yhteys tietokantaan, jonka jälkeen varsinainen kyselyjen tekeminen voi alkaa. Uuden yhteyden luominen vaatii noin 10-20 pakettia, johtuen tarvittavasta autentikoinnista, yhteydenluonnissa annetuista komennoista (mm. tietokannan valinta) ja TCP:n vaatimista yhteydenluonti- ja kuittausviesteistä. Itse kysely, joka kannasta halutaan tehdä, saattaa hyvinkin vaatia vain yhden viestin tietokannalle ja toisen vastaukseksi, joten selvästikin yhteydenmuodostuksesta tulee pullonkaula jos se joudutaan tekemään joka kerta kun jotain halutaan tietokantajärjestelmästä hakea.

Tämä mielessä tietokantajärjestelmän rajapinnan toteuttava luokka on suunniteltu pitämään yhteyttä tietokantaan yllä jatkuvasti ja jakaa sen metodeilleen. HTML:ä generoivilla moduuleilla ei vain ole mekanismeja jakaa samoja tietokantajärjestelmän instansseja keskenään, koska HTML:n generoivat oliot luodaan aina tarpeen mukaan.

Ratkaisuksi tähän kehitettiin ns. poolausmalli, joka on yleisesti käytössä mm. webpalvelimissa. Keskeisenä ideana on pool, “allas”, josta on olemassa ainoastaan yksi instanssi singleton-mallin mukaan. Tämä pool hallinnoi nimensä mukaisesti tietokantarajapinnan toteuttavia olioita ja antaa niitä tarvittaessa “altaastaan” olioille, jotka niitä pyytävät.

Kun poolista ensimmäisen kerran pyydetään tietokantarajapinnan mukaista oliota, alustaa pool itsensä ja luo tietokantarajapinnan olion. Tarpeen mukaan pool luo lisää olioita altaaseen tiettyyn ylärajaan saakka. Kun HTML:ä generoivat tai kantaan uutta tietoa sisällyttävät oliot eivät enää tarvitse tietokannan palveluja luovutetaan tietokantarajapinnan olio takaisin “altaaseen”.

Tälle toteutustavalle on luonnollisesti olemassa vaihtoehtoja. Yksi ehdotettu tapa olisi ollut singleton-mallin mukainen tietokantarajapinta, mutta siitä luovuttiin nopeasti koska se johtaa suorituskykyongelmiin; käytännössä tietokantarajapinnan toteuttava olio ei kovin helposti voi olla re-entrantti johtuen siitä, että tietokantayhteys on jaettu, mikä vaatii hyvin tarkkaa suunnittelua ja toteutusta synkronoinnin osalta ettei kilpailutilannetta tietokantaa käyttävien metodien kesken synny. Rinnakkaisjärjestelmiä on erityisen vaikea testata luotettavasti, joten ohjelman toimivuus pitäisi pystyä osoittamaan pitkälti teoreettisin keinoin, mikä ei myöskään ole helppoa rinnakkaisessa järjestelmässä. Tämä ei sinänsä ole krittinen este, mutta rajoitus jonka takia vain yksi säie saa suorittaa jotakin tietokantajärjestelmän metodia tietyllä ajanhetkellä tekee tietokantajärjestelmästä pullonkaulan. Poolauksessa tietokantayhteydet on luotu valmiiksi ja käytössä on useita yhteyksiä, jolloin

ratkaisu ei itsessään muodosta pullonkaulaa. Haittapuolena on, että poolaus vaatii suuremman määrän ohjelmistoakerroksia ja huomattavasti enemmän muistia kuin singleton-malli. Käytännössä nämä olivat kuitenkin varsin merkityksettömiä ongelmia, sillä tietokantarajapinnan oliot eivät kuluta kovinkaan paljoa muistia ja toteutus ei muuttunut kohtuuttoman monimutkaiseksi.

Poolaus itsessään vaatii synkronoinnin, mutta vain kun "olioallasta" ylläpidetään, mikä on varsin nopea operaatio. Vastapainoksi itse tiedon haulle ja lisäykselle tietokantaan toteuttiin erillinen käyttäjäautentikoinnin tekevä luokka singleton-mallilla. Teknisesti tämä luokka kuuluu tietokantajärjestelmään, mutta siinä on vain yksi julkinen metodi, jolla www-julkaisujärjestelmään kirjautuva ylläpitäjä autentikoidaan. Tämä luokka tehtiin kuitenkin omaksi osakseen, sillä autentikaatiotietokanta ei asiakkaan järjestelmässä ole välttämättä samalla koneella, jonka takia tarvitaan oma tietokantayhteys autentikaatiota varten. Koska autentikaatio tehdään suhteellisen harvoin ja se on nopea operaatio, niin se toteutettiin yksinkertaisuuden vuoksi singleton-mallin mukaisella luokalla/oliolla.

Poolausmallilla saadaan muitakin etuja, kuten domain-nimien cachetuksen keskitetty toteutus (kts. 4.5), WWW-järjestelmän edustapalvelimien cacheihin olennaisesti liittyvän riippuvuustarkastimen (dependency tracker) ja cachen invalidoijan keskitetty käynnistys ja hallinta ja helpompi tietokantayhteyksien hallinta (esim. samanaikaisten yhteyksien maksimimäärän rajoittaminen). Nämä kohdat eivät välttämättä edusta olio-ohjelmoinnin ideaalista arkkitehtuuria (kts. [Schach02]), mutta on syytä korostaa, että tietokantarajapinnan toteuttava luokka ja poolausluokka ovat vahvasti kytköksissä keskenään, mikä myös monella tapaa hyödyntää oliomallin etuja (esim. luonteva enkapsulointi).

4.4 Valittu tiedostontallennus

Tiedostontallennustapaa valitessa www-julkaisujärjestelmälle, jolla oli annetut vaatimukset, olivat valintakriteerit joustavuus, skaalautuvuus ja suorituskyky. Vaihtoehdot olivat joko tiedostot tiedostojärjestelmään ja taulu tietokantaan, jossa on tiedostoihin liittyvää metadataa ja tiedostojen sijainnit levyllä tai tiedostot ja niiden metadata tallennus tietokantaan.

Vaatimusten perusteella meidän täytyy tietää millä www-sivustolle tiedosto kuuluu, sekä hieman tiedostokohtaista metadataa. Itse tiedoston sisältöön ei tarvitse koskea: se tallennetaan kokonaisena levylle ja ladataan kokonaisena sieltä ilman, että sitä tarvitsisi mitenkään käsitellä. Nämä vaatimukset huomioonottaen molemmat tallennustavat tarjoavat riittävän joustavuuden.

Joustavuuden ja toteutuksen mielekkyyden puitteissa arvioitiin tiedostojärjestelmään tallentamista ja pitäisikö käyttää NFS-tyyppistä levyjakoa vai jonkinlaista client-server -arkkitehtuuria, jolloin ohjelmisto pyytää tiedostopalvelimelta (ohjelmistolta) saada tietyn tiedoston (esim. FTP). Päädyttiin siihen, että koska NFS on joustava, sille on helppo ohjelmoida ja se ei tiettävästi ole este suorituskyvylle, niin jos päädytään tiedostojärjestelmän suoraan käyttöön käytetään NFS:ä hajautukseen. Etuna tässä on myös se, että virheiden etsiminen ja korjaaminen on huomattavasti helpompaa kuin jos käytettäisiin jonkinlaista FTP:n tyyppistä yhteyttä.

Tietokanta voidaan laittaa eri koneelle kuin www-julkaisujärjestelmän ohjelmisto ja niin voidaan tiedostotkin, joten skaalautuvuus ei tuota ongelmia. Molemmissa voidaan myös hyödyntää RAIDia ja klustereita, joten kasvuvaraa löytyy.

Suorituskyky on luonnollisesti tärkeää; järjestelmän tulisi toimia ilman huomattavaa investointia lisälaitteistoon. Koska järjestelmän vaatimukset olivat kohtuullisen selvät ja tiedostontallennus ei vaatisi kovin monimutkaisia ratkaisuja, päätettiin tehdä suorituskykymittauksia. Lopullinen ajoalusta tulee olemaan Java 1.4 käyttäen servlettejä joko Tomcatilla tai Jettyllä ja tietokanta tulee olemaan MySQL. Tämä tiedossa tehtiin testi, jossa koetietokantaan ladattiin kuva ja sama kuva kopioitiin levylle. Kuva ladattiin sitten molemmilla tavoilla (tiedostosta ja tietokannasta) ja mitattiin kuinka nopeasti se käy.

Varsinainen testilaitteisto ja asetelma oli seuraava: käytettiin web-palvelimena Tomcat 4:ä yhdellä 1.5 GHz AMD Athlon -koneella, jossa oli 512 Mt muistia eikä muuta kuin välttämättömät prosessit käynnissä. Tiedosto- ja tietokantapalvelimena oli 450 MHz Intel Celeron, jossa oli 256 Mt muistia, MySQL 4.0 ja myös mahdollisimman vähän prosesseja käynnissä. Koneet yhdistettiin 100 Mbps ethernet-verkolla, jossa ei ollut muista laitteita testin aikana. Molemmissa koneissa NFS-tuki on osana kerneliä (Linux 2.4.20) suorituskyvyn takaamiseksi. Java-virtuaalikoneena toimi Sunin 1.4.2.

Tomcattiin tehtiin minimaaliset servletit, yksi tiedoston lataamiseen tietokannasta, toinen tiedoston lataamiseen tiedostosta. Molemmat Servletit palauttavat HTTP-vastauksen, jonka MIME-tyyppi on image/jpeg ja sisältää kuvan. Tiedoston tietokannasta lataava koodi oli seuraava:

```
String selectQuery = "SELECT id,payload FROM img";
ResultSet rs = stmt.executeQuery(selectQuery);
rs.next();
Blob blb = rs.getBlob("payload");
byte img_data[] = blb.getBytes((long) 1,(int) blb.length());
outStream.write(img_data);
outStream.flush();
```

Kuvan tiedostosta lataava koodi oli saatiin eräästä Sunin esimerkistä:

```
int c = 0;
byte[] img_data = new byte[40000];
File inputFile = new File("/mnt/test1.jpg");

try {
    FileInputStream in = new FileInputStream(inputFile);
    while ((temp = in.read()) != -1) {
        img_data[c] = (byte) temp;
        c++;
    }
    in.close();
```

```
    outStream.write(img_data, 0, c);
    outStream.flush();

} catch (Exception e) {}
```

Testit tehtiin ApacheBench -ohjelmalla (tulee Apache-webserverin lähdekoodin mukana, `http://httpd.apache.org`), jolla voi suorittaa annetun määrän sivun latauksia. Ohjelmalle voi myös antaa yhtäaikaisten hakujen määrän, joten se tekee testaajan työn huomattavasti helpommaksi.

Saatiin taulukossa 1 esitetyt tulokset, jotka ovat muotoa mediaani pyynnöistä/sekuntti.

Pyyntöjä	4000	3587
Pyyntöjä samanaikaisesti	5	2
Tiedosto	20	21
Tietokanta	90	86
PHP:Tiedosto		306
PHP:Tietokanta		187
Staattinen tiedosto		542

Taulukko 1: Tiedonsiirtotuloksia, palveltuja pyyntöjä/sekuntti

Vertailun vuoksi testattiin myös tiedoston ja tietokannasta lataamista PHP-skriptikielellä ja pyytämällä staattista tiedostoa suoraan webserveriltä ilman mitään ohjelmaa välissä. PHP- ja staattinen tiedosto-testerissä käytettiin Apache-webpalvelinta versio 1.3.27:ä, johon PHP oli käännetty moduulina hyvän suorituskyvyn varmistamiseksi. Staattinen tiedosto ladattiin paikallisesta levyjärjestelmästä, joten sillä saadut tulokset edustavat korkeimpia saavutettavissa olevia tuloksia.

Tietokannasta lataaminen on yllättäen neljä kertaa nopeampaa kuin tiedostosta, mutta kummatkin Java-tulokset ovat surkeita verrattuna PHP-tuloksiin ja erityisesti tiedoston lataamiseen Apachella.

Tuloksia tarkastellessa käy ilmi, että joko Java yleensäkin on tehoton tai sitten servleteillä, joilla testattiin, on jotain vikaa. Katsomalla tiedostonlatauskoodia ongelma on ilmeinen. Alla on korjattu koodi tiedostosta lataamiseen, jolla tehtiin uusi testi:

```
int temp;
int c = 0;
File inputFile = new File("/mnt/test1.jpg");
temp = (int) inputFile.length();
byte[] img_data = new byte[temp];

try {
    FileInputStream in = new FileInputStream(inputFile);
```

Pyyntöjä	4000	3587
Pyyntöjä samanaikaisesti	5	2
Tiedosto, v2	376	316

Taulukko 2: Tiedonsiirtotuloksia parannetulla Java-koodilla, palveltuja pyyntöjä/sekuntti

Pyyntöjä	4000	3587
Pyyntöjä samanaikaisesti	5	2
Tiedosto	682	707
Tiedosto, v2	12805	10779
Tietokanta	3073	2939
PHP:Tiedosto	10439	
PHP:Tietokanta		6393
Staattinen tiedosto		18563

Taulukko 3: Lopulliset tiedonsiirtotulokset, kt/s

```
    c = in.read(img_data);

    in.close();
    outStream.write(img_data, 0, c);
    outStream.flush();

} catch (Exception e) {}
```

Uudella koodilla ajettiin uudet testit, tulokset ovat esitettyinä taulukossa 2.

Eli tiedostosta lataamisen nopeus nousi huomattavasti ja näin ollen tiedostosta lataaminen Javalla on varsin tehokasta. Staattisen tiedoston lataamista Apachella voidaan pitää optimitapauksena ja tämä tulos on enemmän kuin puolet siitä. Taulukossa 3 ovat tulokset esitettyinä siirrettyinä kilotavuina sekunnissa:

Näistä tuloksista voidaan nähdä, että 100Mbps verkossa tiedostolataamisnopeudet ovat käytännössä maksimitasolla, ellei NFS käytä tehokkasti lukucachea. Testi kuitenkin kuvaa käytännön tilannetta ja kantaan ei kohdistettu esim. transaktiopuskureita vaativia kyselyjä, joten voidaan perustellusti olettaa, että saadut tulokset edustavat annetulla laitteistolla ja ohjelmistolla parasta saavutettavissa olevaa tasoa.

Koska joustavuudessa ja skaalautuvuudessa tilanne oli eri vaihtoehtojen osalta suunnilleen tasan ja suorituskyvyssä tiedostosta lataaminen oli ylivoimaisesti parempi kuin tietokannasta, niin päädyimme käyttämään tiedostojärjestelmää tietojen tallenukseen. NFS:ä käytetään hajautukseen ja järjestelmä toimii siten, että tiedostojen metadata tallennetaan tietokantaan, johon tallennetaan myös tieto siitä, minne tiedosto on tallennettu.

4.5 Suorituskyky ja skaalautuvuus

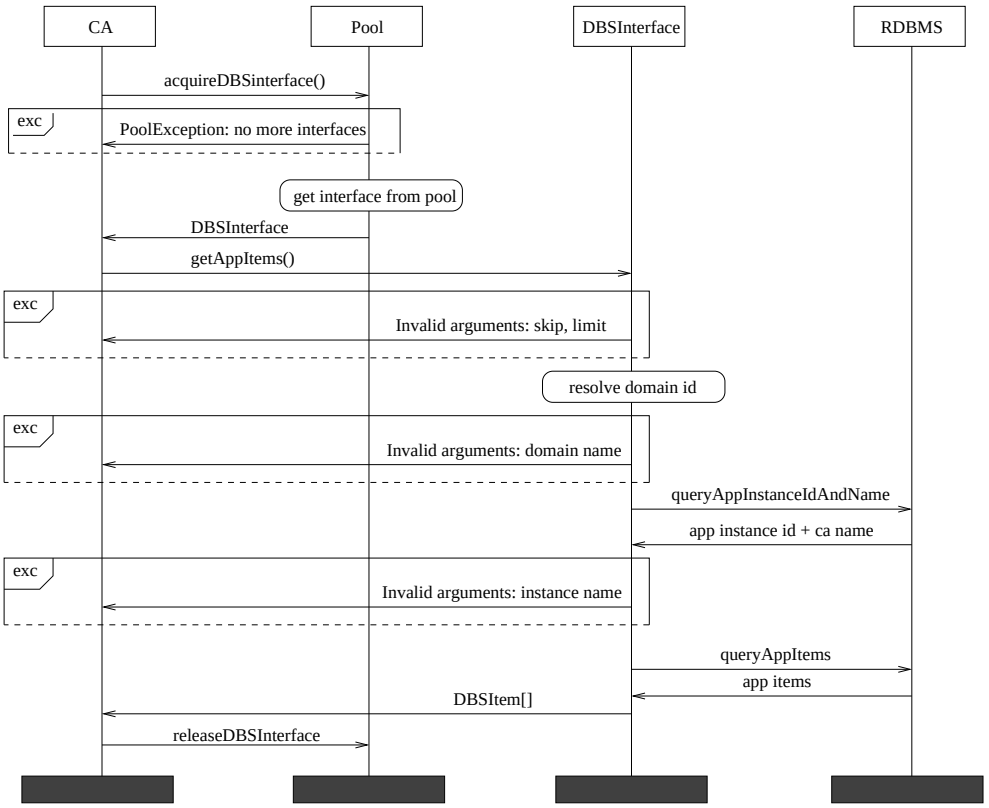
Järjestelmän suorituskyky ja skaalautuminen olivat perustavoitteita ja näiden saavuttamiseksi käytettiin monia erilaisia keinoja sekä suunnittelu että toteutusvaiheessa, pyrkien kuitenkin pitämään toteutuksen suhteellisen yksinkertaisena ja siistinä. Arkkitehtuuritasolla lähdettiin siitä, että tietokanta voi sijaita eri koneella kuin J2EE-sovelluspalvelin ja että taulujen rakenne sopii hyvin WWW-julkaisujärjestelmän tarpeisiin. Poolauksella (kts. kohta 4.3) luodaan yleinen rajapinta järjestelmään, mutta sen perimmäinen tarkoitus on hyvän suorituskyvyn ja tasaisen vasteajan varmistaminen. Myös implementaation yksityiskohdat ovat olennaisia suorituskyvyn takaamiseksi, esim. tehokkaiden SQL-lauseiden käyttö parantaa suorituskykyä huomattavasti.

Suorituskyky ei kuitenkaan ollut ainoa prioriteetti; rajapinnan tuli myös tarjota järkeviä poikkeuksia virhetilanteissa, jotta käyttäjää voidaan informoida miksi jokin tapahtuma epäonnistui (eikä pelkästään tulostaa proseduurikeon sisältöä näytölle). Tämä asetti rajoitteita, sillä käytännössä tiedon haku tietokannasta olisi aina onnistunut yhdellä SQL-lauseella käyttämällä riittävästi liitoksia, mutta jos jokin parametri olisi väärin saataisiin tulokseksi vain tyhjä joukko ilman tietoa siitä, mikä meni pieleen. Myös tiedot, jotka tietokantaa käsittelevät metodit saavat parametreina, täytyy eskapoida etteivät ne sotke SQL-lauseita, mutta tämän vaikutus suorituskykyyn on pientä.

Käytännössä järkevien poikkeusten (ja sitä myötä virheilmoitusten) saaminen vaati sitä, että tehdään muutama kysely tietokannasta ennen varsinaista kyselyä, joka palauttaa halutun tiedon. Nämä itse datan hakua edeltävät kyselyt selvittävät onko olemassa annetun nimistä domainia, sovelluksen instanssia jne. Implementoinnissa keskityttiin aluksi toiminnallisuuteen ja toimivuuteen, jolloin suorituskyky ei vielä päässyt huippunsa. Poolin (kts. 4.3) toteutuksen yhteydessä hiottiin SQL:ä tehokkaaksi ja lisättiin hajautustaulukko, joka cachettaa domainin nimen vastaavaan id:en, tarkoituksena nopeuttaa erityisesti HTML-tyyppisen tiedon hakemista ja tallennusta tietokantaan. Cachetuksen valinta oli sikäli luontevaa, että domainien tiedot muuttuvat harvoin, joten cachea on helppo ylläpitää, se on pienikokoinen ja domainin nimeä vastaavaa id:ä tarvitaan koko ajan (looginen vaihtoehto olisi ollut domainin nimen laittaminen pääavaimeksi tietokannan tauluun, mutta koska nimi voi olla hyvinkin pitkä tätä ei pidetty tehokkaana ratkaisuna).

Haettaessa tietokantajärjestelmästä HTML-sivuille tulevaa tietoa tapahtuva prosessi on pääpiirteittäin kuvattu sekvenssikaaviota muistuttavassa MSC-kaaviossa 4. Kaaviossa on erityisesti esitetty erilaisia poikkeuksia joita voi esiintyä kutsun seurauksena. Kaaviossa on kuvattu lopullinen muoto, eli domain-cache on käytössä.

Toteutuksen aikana arkkitehtuuri havaittiin ongelmattomaksi implementoida ja suorituskyvyssä ei tullut esiin ilmeisiä ongelmia. Luonnollinen ongelma web-pohjaisessa sovelluskehityksessä on se, että järjestelmää on erittäin vaikea testata jokaista vastaan tulevaa tilannetta varten: on vaikea arvioida lopullista järjestelmään tallennetun tiedon määrää, käyttäjämäärät saattavat vaihdella rajusti ja hyvinkin nopeasti ja WWW kehittyy nopeasti, jonka takia on vaikea ennustaa ohjelmiston elinikää ja tulevia vaatimuksia sille. Tämä mielessä päätimme kuitenkin tehdä joitain suoritus-



Kuva 4: Tyypillinen käyttöskenaario: sisältöelementtien hakeminen tietokannasta

kykytestejä, joiden tarkoitus on antaa käsitys järjestelmän suhteellisesta suorituskyvystä, mutta myös antaa jonkinlaista viitettä siitä, missä pullonkaulat ovat ja miten suorituskykyä voi tehokkaimmin parantaa.

Testejä arvioidessa on syytä pitää mielessä, että WWW-julkaisujärjestelmäämme liittyy olennaisesti hajautettu cache, joten tietokantaan tehtäviä hakuja täytyy tehdä vain sivujen päivittyessä. Järjestelmä on kuitenkin niin hidas kuin sen hitain osa ja suurien päivitysten yhteydessä tietokantaan saattaa kohdistua paljon hakuja lyhyessä ajassa, joten tietokantajärjestelmän suorituskyky ei ole yhdentekevä.

Testeissä käytettiin pitkälti samaa alustaa kuin kohdan 4.4 testeissä. Erona oli Linuxin kernelin versio 2.4.22, Tomcatin sijasta käytettiin Jetty 4.2.14:a, koska sen käyttö lopullisessa järjestelmässä näyttää todennäköiseltä ja Athlon-pohjaisessa koneessa oli 1 Gt muistia. Aluksi ajettiin ApacheBenchiä siten, että tietokanta, ApacheBench ja sovelluspalvelin olivat samalla koneella (Athlon-pohjainen kone). Sen jälkeen ajettiin testit siten, että MySQL-tietokanta oli eri koneella kuin sovelluspalvelin. Tulokset on esitetty taulukossa 4.

Testissä haettiin yksinkertaista servlettiä, jota tarjoiltiin Jettyn välityksellä. Servletti hakee tietokannasta XML-dataelementin ja tulostaa sen, HTML-sivun ollessa näin 4005 tavun kokoinen (mitään XML:stä HTML:än muunnosta ei tehdä, eli tar-

Pyyntöjä	8000	8000
Pyyntöjä samanaikaisesti	6	16
Apache	893	882
Servlet	369	328
ilman cachea	319	293
ilman joinia	283	270
Tietokanta eri koneella kuin sovelluspalvelin		
Servlet	340	352
ilman cachea	284	301
ilman joinia	240	239

Taulukko 4: ApacheBench: sivupyyntöjä sekunnissa

koituksena on pelkästään testata tietokantajärjestelmän suorituskykyä tyypillisessä tapauksessa, ei toiminnallisuutta).

Taulukossa 4 Apache tarkoittaa tiedoston hakemista staattisena tiedostona Apache 1.3.28 HTTP-palvelimelta. Servlet on Jettyn kautta tarjottu Servletti, jossa tietokantarajapinta on pitkälti lopullisessa muodossaan. Samaa servlettiä on testattu myös siten, että tietokantarajapinnasta on poistettu domain-nimien cachetus, sen jälkeen cachetus on ollut pois päältä ja liitoksia käyttävät SQL-kyselyt on korvattu triviaaleilla kyselyillä.

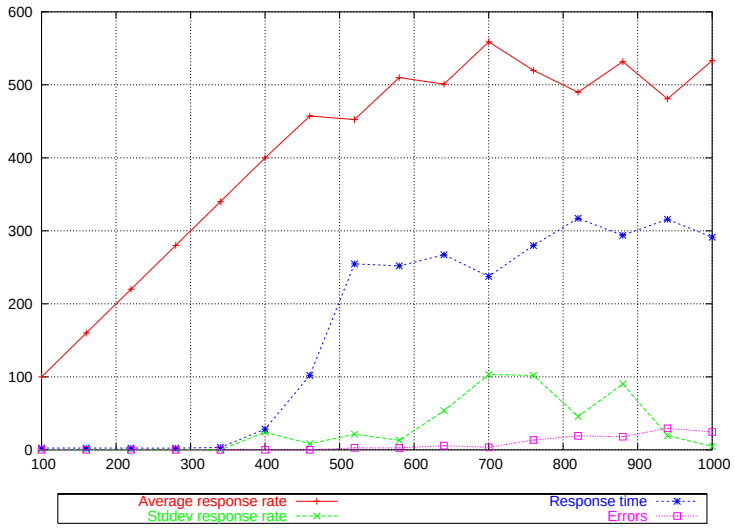
Taulukosta 4 nähdään, että domain-nimien cachetus parantaa suorituskykyä 10-20%, joten 16% suorituskyvyn parannusta voidaan pitää hyvänä perusarviona sen hyödyistä. Parempien SQL-kyselyiden käyttäminen lisäsi suorituskykyä 9-26%. Kuten olettaa saattoi, on suorituskyvyn parannus suurin silloin kun verkko lisää latenssillaan tietokantahakuun kuluvaa aikaa.

Saadut tulokset ovat sikäli mielenkiintoisia, että kun tietokantapyyntöjä vähentävät optimoinnit kytketään päälle, niin tehtäeassä useita samanaikaisia pyyntöjä pystytään siirtämällä tietokanta eri koneelle palvelemaan suurempaa määrää pyyntöjä. Tämä ja osittain myös se että optimoinnit hyödyntävät erityisen hyvin hajautettua ratkaisua paljastuu osittain tarkkailemalla prosessoritehon kulutusta (muistin kulutus ei kertaakaan lähennellyt koneen fyysisen muistin määrää ja swappia ei käytetty kertaakaan, vaikka Java-virtuaalikone veikin paljon muistia). Kaikissa testeissä jotka ajettiin yhdellä koneella CPU:n kulutus oli 100%, josta noin 15% meni käyttöjärjestelmän toimintoihin. Kun tietokanta siirrettiin eri koneelle, käytti se 55-65% CPU:n ajasta sillä koneella (on syytä huomata, että kyseinen kone on huomattavasti hitaampi kuin kone jolla sovelluspalvelinta ajettiin), mutta kun käytössä ei ollut mitään optimointeja oli sovelluspalvelimen CPU:n kulutus noin 80%, eli verkko rajoitti suorituskykyä (1 Gbps ethernetillä tilanne olisi saattanut olla toinen). Kun optimoinnit kytkettiin päälle ja pyyntöjä tehtiin 16 samanaikaisesti pysyi tietokantapalvelimen rasitus ennallaan mutta sovelluspalvelimen kuorma nousi 100%:in.

Tehdyissä testeissä on se ongelma, ettei voitu realistisesti simuloida tietokannan

tilaa, jossa se on tyypillisessä Internet-käytössä [Greenspun98]. Kuitenkin tuloksista voidaan selvästi nähdä, että Javalla toimiva sovelluspalvelin on pääasiallinen pullonkaula järjestelmässä ja nykyään verrattain perinteisellä lähiverkolla voidaan parantaa suorituskykyä suurilla kuormilla jos tietokantapalvelin eriytetään omalle koneelleen.

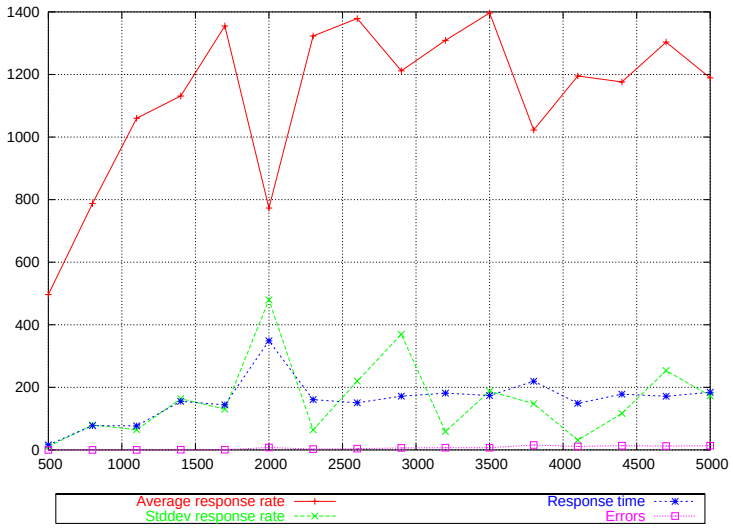
Apachebenchillä tehdyissä testeissä ei kuitenkaan saada koko kuvaa suorituskyvystä. Mosberger ja Jin [Mosberger03] ovat perustelleet tehokkaan ja joustavan työkalun käyttöä mittaamaan HTTP-palvelimen käyttäytymistä eri kuormituksilla. ApacheBench-testien perusteella halusimme tietää tarkemmin miten tietokantajärjestelmä käyttäytyy erilaisilla käyttöprofiileilla. Testaukseen käytettiin httperfiä ([Mosberger03]) Autobench-skriptin avulla (`http://www.xenoclast.org/autobench/`). Tulokset testeistä on esitetty kaavioissa 5 ja 6.



Kuva 5: Httperf

Testeissä tehtiin aina 50:stä 500:an yhteyttä sekunnissa 30 yhteyden askelissa HTTP-palvelimeen. Kaaviossa 5 jokaisella yhteydellä tehtiin 2 sivupyyntöä. Kaaviossa 6 tehtiin muuten sama testi, mutta nyt pyyntöjä oli 10 jokaista yhteyttä kohden. Testit tehtiin siten, että sovelluspalvelin ja tietokantapalvelin olivat samalla koneella ja lähiberkon yli käytetty kone toimi testikoneena, jolta pyynnöt tehtiin. Kaavioissa näkyy keskimääräisen vastausmäärän lisäksi keskihajonta, vastausajan muutos sekä virheiden määrä.

Kaaviosta 5 voidaan nähdä, että palvelin kyllästyy noin 500 pyyntöä/sekunnissa paikkeilla, kun taas kaaviossa 6 kyllästyminen tapahtuu hieman yli 1000:n pyynnön/sekuntti kohdalla. Paremmalla testilaitteistolla olisi saavutettu tarkempia tuloksia, mutta tuloksista saadaan kuitenkin se käsitys, että yhteydenmuodostuksen nopeus on ratkaisevaa suorituskyvyn kannalta. Jos HTTP-palvelimen käyttäjät selailevat useita sivuja kerralla kaksinkertaistuu suorituskyky helposti. Kun samalla yhteydellä tehdään useita pyyntöjä, muuttuu HTTP/sovelluspalvelin pullonkaulak-



Kuva 6: Httperf

si ja suorituskyky hyppelehtii paljon tietyn maksimitason ympärillä vasteaikojen vaihdellessa ja kasvaessa huomattavasti. Tämä tilanne on kuitenkin parempi kuin se, että suorituskyky romahtaa voimakkaasti tietyssä pisteessä, mikä tapahtuu tietyillä alustoilla samanlaisilla testeillä. Osittain tämä johtuu siitä, että poolauksessa asetettu tietokantayhteyksien maksimimäärä (kts. 4.3) rajoittaa kulutettujen resurssien määrä, mikä estää palvelimen täydellisen hyytymisen kovassa kuormituksessa (luonnollisesti tämä johtaa siihen, että monet clientit saavat ilmoituksen siitä että palvelu on tällä hetkellä tavoittamattomissa, mutta parempi tarjota palvelua joillekin käyttäjille kuin ei kenellekään). Loppujen lopuksi 1000:n pyynnön palveleminen sekunnissa on varsin hyvin ilman cachettavaa sovelluspalvelinta toimivalle tietokantapohjaiselle web-sovellukselle, varsinkin kun käytössä ei ollut kallista laitteistoa.

4.6 Yksityiskohtia

Koko järjestelmää rakennettaessa joudutaan myös miettimään varsin käytännöllisiä kysymyksiä alkaen tietokannan valinnasta. Lisäksi joudutaan valitsemaan tapa, jolla ko. tietokantaan saadaan yhteys eri koneelta sekä tapa hakea myöskin tiedostot sieltä. Myöskin konkreettinen toteutustyö vaatii hieman pähkäilyä, liittyen ainakin synkronointiin sekä säikeiden käyttöön.

Seuraavassa esitetään suunniteltaessa ja toteutettaessa vastaan tulleita yksittäisiä kysymyksiä.

Unicode-tuki Vaikka todenäköisesti kannan pääasiallisessa käyttökohteessa tullaan todennäköisesti näkemään vain suomen- ja englanninkielistä materiaalia, päätettiin toteutuksessa ottaa kantaa myös muihin merkistöihin. Koska tietokannoista ei löytynyt suoraa Unicode-tukea (ks. seuraava kappale) ja sitä ei uskallettu jättää

kokonaan poiskaan, päätettiin eskapoida kaikki ei-ASCII merkit. Eskapointi toteutettiin rajapinnan alapuolella siten, että järjestelmän muiden osien, jotka luontevasti käyttivät Unicode-merkkijonoja, ei tarvinnut huolehtia siitä.

Tietokannan valinta Alla oleva tietokanta on yksi tärkeimpiä ratkaisuja koko järjestelmässä. Koska budjetti oli rajoitettu, kalleimmat vaihtoehdot eivät tulleet kysymykseen. Kuitenkin tietokannalta vaadittiin joitakin ominaisuuksia, joita ei löydy aivan kaikista kannoista, päällimmäisenä tuki transaktioille, Unicodelle sekä kokoriippumattomille tekstihauille. Käytännössä vaihtoehdot olivat Postgres sekä MySQL 4.0 ja 4.1. Postgres ei kuitenkaan osannut kokoriippumattomia tekstihakuja, MySQL 4.0 ei sisältänyt tukea Unicodelle ja versio 4.1, jossa olisi Unicode-tuki, oli vielä beta-vaiheessa.

Monisäikeistys Suunnitteluvaiheessa tiedostettiin myös mahdolliset pullonkaulat, joita saattaa tulla vastaan mikäli koko järjestelmä toimii vain yhdessä säikeessä. Tällöin esimerkiksi levyltä luku saattaa odotuttaa itseään, kun samalla voitaisiin tehdä jotakin muuta. Lopulta päädyttiin kuitenkin tekemään koko järjestelmä ilman eksplisiittistä säikeistystä; kohdassa 4.3 esitetty usemman rajapintaolion käyttäminen samanaikaisesti poolauksen avulla mahdollistaa käytännössä implisiittisen monisäikeistyksen, sillä käytössä on samanaikaisesti useita yhteyksiä tietokantaan. Mitään eksplisiittisiä säikeitä ei kuitenkaan luotu, vaan järjestelmä perustuu siihen, että J2EE:n mukainen Servlet-palvelin (Tomcat tai Jetty) pitää yllä virtuaalikonetta ja siellä yhtä tietokanta-“poolia”, jota käyttämällä useat servletit voivat samanaikaisesti käyttää tallennusjärjestelmää.

5 Yhteenveto

Kaiken kaikkiaan toteutettu järjestelmä täytti sille asetetut vaatimukset (luku 2). Vaatimukset koskivat toiminnallisuutta, suorituskykyä ja hajautusta; suorituskyvyn suhteen mitään tarkkoja suoritusaikarajoja ei asetettu, vaan vaatimuksena oli pikemminkin “riittävä” nopeus; tässä sillä tarkoitetaan lähinnä sitä, että tietokannan hitaus ei muodostu esteeksi järjestelmän järkevälle käytölle. Tietokannasta tulee todennäköisesti järjestelmän pullonkaula suorituskyvyn suhteen, mutta se oli odotettavissa riippumatta toteutuksen laadusta.

Toiminnallisuutta koskevat vaatimukset tulivat täytettyä. Alkuperäinen suunnitelma ei tosin ollut ottanut huomioon kaikkia toiminnallisia vaatimuksia ja niitä jouduttiin kehitysvaiheessa lisäämään; koska kaikki muiden tahojen esittämät toiminnalliset vaatimukset toteutettiin, ei siihen selkeästikään jäänyt puutteita. Järjestelmän laajentamista ajatellen ominaisuuksia voitaisiin edelleen lisätä merkittävästi, mutta tähän tarkoitukseen nykytoiminnallisuus on riittävä.

Hajautuksenkin suhteen vaatimus oli lähinnä se, että järjestelmän voi hajauttaa. Tämäkin vaatimus täyttyi, koska hajautus otettiin suunnittelun lähtökohdaksi heti alussa.

Mikäli järjestelmää ajattelee laajempaan levitykseen ja kaupalliseen käyttöön, sen ominaisuuksia olisi edelleen parannettava ainakin tietoturvan, siirrettävyyden ja skaalautuvuuden osalta. Mikään näistä asioista ei muodostu ongelmaksi tässä tapauksessa (asiakas on verrattain pieni ja tietokanta on eristetyssä sisäverkossa), mutta yleisemmässä tapauksessa ne kaikki olisi otettava tarkemmin huomioon. Siirrettävyyden puutteella tässä viitataan lähinnä toteutuksen tietokantariippumattomuuteen, koska toteutus riippuu tällä hetkellä MySQL:stä. Toteutusvaiheessa tavoiteltiin täyttä SQL/92-kelpoisuutta mutta käytännössä se ei yksinkertaisesti onnistunut. Skaalautuvuuden osalta parannettavaa olisi kyselyiden ja tietokannan rakenteen optimoinnissa; tällä hetkellä järjestelmään saattaa muodostua pullonkauloja mikäli liikenne kasvaa kovin suureksi.

Viitteet

[Florescu98] Florescu, D.; Levy, A.; Mendelzon, A. 1998. *Database Techniques for the World Wide Web: A Survey.* ACM SIGMOD Record 27:3, September 1998, 59-74.

[Greenspun98] Greenspun P., *SQL for Web Nerds*, `http://philip.greenspun.com/sql/`

[Maciaszek90] Maciaszek L. A., *Database Design and Implementation*, Prentice-Hall, 1990

[Mosberger03] Mosberger D., Jin T., *httperf—A Tool for Measuring Web Server Performance*, `http://www.hpl.hp.com/personal/David_Mosberger/httperf.html`

[Schach02] Schach S. R., *Object-Oriented and Classical Software Engineering*, McGraw-Hill, Fifth Edition, 2002