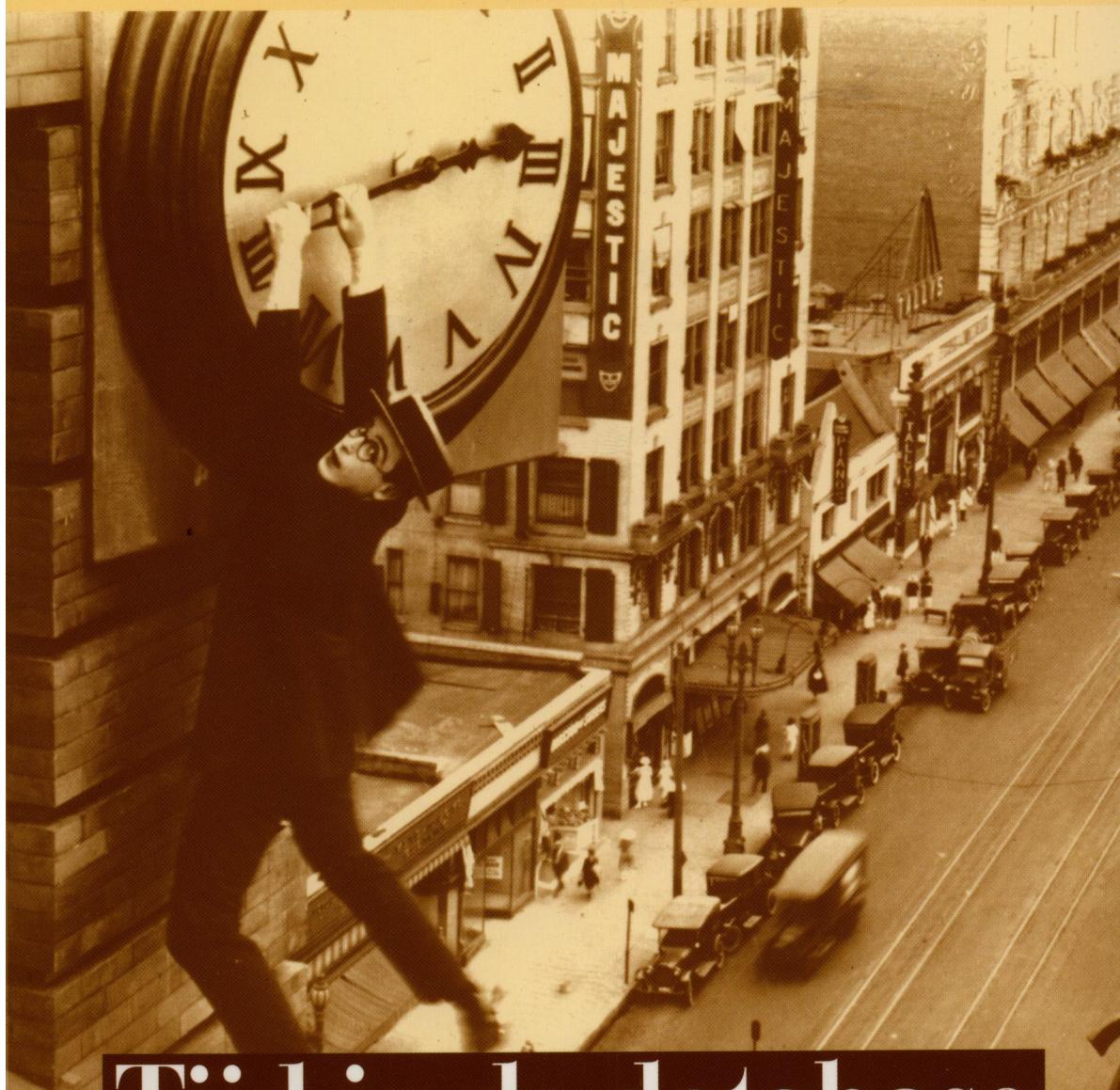


DB/M
ESSAY

Een bundeling van artikelen verschenen in
Database Magazine in de periode 1997 - 1998



Tijd in de database

Tussen eeuwig heden en complex verleden

Dr R.J. Veldwijk

S.J. Cannan FBCS Ir. F.G.W. van Orden

Tijd in de database

Van eeuwig heden tot complex verleden

Dr R.J. Veldwijk
S.J. Cannan FBCS
Ir. F.G.W. van Orden

Array Publications b.v., Alphen aan den Rijn

Voor M.J. Kluit, mijn lievelingsauteur



Voorwoord

Voor u ligt de bundeling van een serie van zes artikelen die tussen september 1997 en april 1998 in Database Magazine zijn verschenen. De artikelen gaan over een en hetzelfde onderwerp, namelijk het probleem van het modelleren van tijdsaspecten in administratieve applicaties. Dit historieprobleem is algemeen erkend als een van de grote problemen – volgens sommigen het grootste probleem – waarmee de ontwikkelaars van vandaag te kampen hebben en er is dan ook enorm veel over gedacht en geschreven. Hoe jammer en hoe ongehoord dat er van al die inspanning bijna niets doorsijpelt naar de praktijk van alledag, noch in de vorm van ontwerp- en ontwikkelvoorschriften, noch in de vorm van ondersteuning door dbms'en en ontwikkeltools.

Mijn eerste ervaring op enige schaal met het historieprobleem dateert van 1990 toen ik verantwoordelijk was voor de totstandkoming van een personeelsinformatiesysteem onder Oracle voor een groot omroepproductiebedrijf in het midden des lands. Het gegevensmodel was niet erg complex, de te realiseren functionaliteit was te overzien, de medewerkers waren vakkundig, maar de noodzaak om aan van alles en nog wat een tijdsdimensie mee te geven maakte het leven tegelijk complex en saai. Sommige projectmedewerkers waren zich er overigens niet eens van bewust dat ze bezig waren met het uitvoeren van steeds hetzelfde werk in een steeds marginaal andere context. Deze collega's waren gelukkig met hun werk en zijn dat bij mijn weten nog steeds. Daarmee zijn we aangekomen bij een eerste doelstelling die ik met deze bundel hoop te bereiken:

Het eerste doel van dit boekwerk is het op u – of via u op uw omgeving – overdragen van ongenoegen over al het overbodige werk dat verbonden is aan het modelleren en implementeren van tijdsaspecten.

De gedachte hier achter is natuurlijk dat ontevredenheid de voedingsbron van elke revolutie is. Anders gezegd: een tevreden code-klopper is geen her-

rieschopper. Wij zouden het toejuichen als de gangbare wijze van historische systeemontwikkeling dezelfde status krijgt als, laten we zeggen, Jaar 2000-werk, dat overigens niets te maken heeft met het historieprobleem.

De tweede historieprobleem-scène speelt zich af tijdens mijn promotie-onderzoek. Wie wetenschappelijk bezig is met database-problematiek kan haast niet om de 'Temporal Database scene' heen. De literatuur over dit onderwerp is zo immens – en de literatuur over mijn onderzoeksobject, flexibiliteit, zo schaars – dat een calculatorende en op scoren gerichte promovendus het onderwerp historie beslist moet mijden. Daarbij kwam nog de overweging dat dit aspect van database-theorie meer dan gemiddeld wordt gedomineerd door de harde jongens van het vakgebied, de wiskundig informatici. Op zich is dat bijna al afdoende verklaring voor het feit dat er van al dit onderzoek bijna niets doordringt tot de werkvloer. Daarbij kwam nog het gevoel dat veel van de artikelen over historie zich concentreren op een te beperkt gebied (het dbms alleen), niet op afzienbare termijn te implementeren zijn en meestal voorbij gaan aan de kernproblemen rond historie in administratieve omgevingen: extreme complexiteit en ondoenlijk integriteitsmanagement.

De derde scène speelt zich af rond de Tien Geboden-serie die ik in 1995 voor Database Magazine mocht schrijven. Natuurlijk moest aan historie een artikel worden gewijd, maar de contouren van een praktisch haalbare oplossing waren op dat moment pas vaag zichtbaar aan het worden. En dus luidde het gebod: "Gij Zult het Historieprobleem Vrezen" Het aantal reacties op dit artikel was groter dan ik ooit op enig ander artikel heb mogen krijgen en dat was voldoende reden om na te gaan denken over een separate artikelserie. Een serie artikelen over een deprimerend moeilijk en schijnbaar specialistisch probleem moet echter toch ook iets van een happy end bieden, maar in de tussentijd was ik tot de conclusie gekomen dat sommige aspecten van het historieprobleem zich fundamenteel niet lenen voor generieke oplossingen. En hoe frustrerend het ook is te zien dat middelmatige mensen gelukkig zijn met het keer op keer uitcoderen van hetzelfde probleem; het is pas echt tragicisch om begaafde en ambitieuze mensen bezig te zien met het bedenken van generieke oplossingen voor problemen waarvoor dergelijke oplossingen niet bestaan. Daarmee zijn we bij de tweede doelstelling van dit boek aangeland:

Het tweede doel van dit boekwerk is het identificeren van aspecten van het historieprobleem waarvoor van geval tot geval een oplossing gezocht moet worden.

Die sleutelscène speelt zich af gedurende de laatste twee à drie jaar. Gedurende deze periode kom ik in contact met een aantal begaafde lieden met wie ik gaandeweg de ambitie ben gaan delen om problemen als historie – voor zover mogelijk – te kraken en de oplossing te implementeren. Geen van beide had ik kunnen doen zonder de hulp van mijn collega's binnen de

Voorwoord

FAA Groep: Martin Boogaard, Edzo Bakker, Diana Koppenol, Jeroen Schaay, Frido van Orden en Michiel Musters. Met name de inbreng van Frido en Michiel bij het identificeren van theoretische problemen (Frido) en bij het tot stand brengen van werkende programmatuur (Frido en Michiel) is van onschatbaar belang voor de onderbouwing van het verhaal dat dit boek vertelt. Pas na implementatie durfde ik het aan om met de gangbare bravoure te beginnen aan de historieserie. En daarbij komen we bij het belangrijkste doel van dit boek:

Het derde doel van dit boekwerk is het bieden van een blauwdruk voor het implementeren van een generieke oplossing van het historieprobleem.

Enfin, het resultaat van al deze scènes ligt er. Applaus of tomaten! Resten mij nog een drietal opmerkingen over de vorm en inhoud:

Anders dan bij Tien Geboden-serie was het hier van meet af aan mijn doel om deze serie uitgegeven te zien. Toch was de verleiding te groot om in te gaan op een artikel in Database Magazine over de automatisering bij het Kadaster. De gang van zaken bij het Kadaster vormt de leidraad voor de besprekings in hoofdstuk 2. Wij menen dat het artikel zich goed laat lezen zonder kennisname van het Kadaster-verhaal. Wie de volledige achtergrond wil meepakken zij verwezen naar DB/M nr. 5 en 7, 1997, die het genoemde artikel en enkele aanvullingen van Kadaster-zijde over hun umfelt bevatten.

De artikelserie beschrijft de theoretische kanten van het historieprobleem en biedt een blauwdruk voor een oplossing. Implementatie vereist natuurlijk dat die blauwdruk wordt vertaald in werkende software en dat is beslist geen eenvoudige aangelegenheid. Om enigszins tegemoet te komen aan de begrijpelijke wens om nader te worden geïnformeerd over het hoe, heeft Frido van Orden (wie anders) een hoofdstuk toegevoegd over de belangrijkste aandachtspunten bij implementatie.

Los van alle – soms lelijke – dingen die ik her en der zeg over onderzoekers en toolverkopers, ben ik beslist niet van mening dat de artikelserie het hele verhaal bevat. In het bijzonder ben ik van mening dat er meer te zeggen valt over de taalaspecten van een generieke implementatie van historie. Gelukkig is Stephen Cannan bereid gevonden om twee voortreffelijke artikelen toe te voegen die in DB/M nr. 5 en 7, 1996. Met de bijdragen van Frido en Stephen hopen wij een acceptabele graad van volledigheid te hebben bereikt.

Maarssen, juni 1998

René Jan Veldwijk

e-mail: renev@faapartners.com

Tijd in context

Een van de grootste uitdagingen van deze tijd is het geschikt maken van administratieve systemen voor het omgaan met historische gegevens. De toevoeging van een tijdsdimensie aan gegevens staat bekend als een van de grote problemen op het gebied van administratieve informatiesystemen en het hoeft daarom niet te verbazen dat al velen hun tanden in het probleem hebben gezet. Informaticawetenschappers hebben diverse varianten en uitbreidingen op het relationele model en meer recent op het objectgeoriënteerde model voorgesteld. Die inspanningen hebben tot op heden echter niet geleid tot additionele voorzieningen in dbms'en en tools en dus houden velen van ons zich regelmatig bezig met het vertalen van historiebehoeften van eindgebruikers in complexe en onderhoudsgevoelige systemen. In dit en de volgende hoofdstukken zullen we het historieprobleem uitgebreid bespreken en op basis van die analyse een daadwerkelijke oplossing voor het historieprobleem voorstellen. Die oplossing voldoet aan de meest gangbare ideeën over historie in de wetenschappelijke wereld, maar voegt een aantal aspecten toe die doorgaans over het hoofd worden gezien. Daarnaast is deze oplossing te implementeren met de hulpmiddelen waarover we vandaag beschikken, zodat theorie meteen kan worden vertaald in praktijk.

Wie ver afstaat van het vak van systeemontwikkelaar heeft vaak weinig begrip voor de dagelijkse problemen van deze snel groeiende beroepsgroep. Vertel iemand die niet met administratieve systemen werkt dat er een historieprobleem bestaat en reken op onbegrip. Stel dezelfde vraag aan een eindgebruiker en de kans op herkenning is al iets groter. Natuurlijk bestaat de kans dat die eindgebruiker denkt aan het jaar 2000-probleem, maar die verwarring is voorbijgaand. Onze gebruiker zal zich na enig doordenken realiseren dat hij bij tijd en wijle lijsten uitdraait die worden opgeslagen in een kast voor latere raadpleging. Misschien herinnert hij zich ook dat het nieuwe datawarehouse een antwoord is op de vraag naar inzicht in het verloop van gegevens in de tijd. Werkt de eindgebruiker bij een salarisadministratie, een uitkeringsinstantie, een pensioenfonds of een financiële instelling dan

bestaat de kans dat hij zijn administratieve systeem gebruikt als een vereidelde kaartenbak en allerlei registraties en berekeningen buiten het systeem voert, mogelijk ondersteund door een spreadsheet als armelui's warehouse.

Stellen we de historievraag tenslotte aan een systeemontwikkelaar en de kans op een blik van herkenning is het grootst. We negeren de minderheid van ontwikkelaars die alle historie-administratie aan zijn eindgebruikers overlaat en zich van geen probleem bewust is. De meerderheid weet waarover we het hier hebben: bij voortdurend voeren onze gebruikers Update- en Delete-acties uit op hun gegevens en elke actie komt overeen met het vernietigen van informatie die mogelijk later van belang blijkt te zijn. Natuurlijk kunnen we mompelen dat het vasthouden van allerlei informatie kostbaar en technisch onmogelijk is, maar dat argument maakt steeds minder indruk bij voortschrijdende hardware-technologie. Veel beter en eerlijker is het om te zeggen dat het vasthouden van historische versies van gegevens onze systemen duur in bouw en complex in onderhoud maakt en dat we de problemen liever bij de eindgebruikers laten liggen, tenzij er natuurlijk een harde noodzaak bestaat om het systeem historiebestendig te maken.

Historie in Database Magazine

Voor we een begin kunnen maken met deze historie is het wellicht goed om aansluiting te zoeken bij datgene wat er al eerder over dit onderwerp in DB/M is gepubliceerd. Dat blijkt vóór 1995 verrassend weinig: één artikel om precies te zijn. In 1995 beschrijft ondergetekende de historieproblematiek in het kader van de Tien Geboden-serie [15], maar in dit artikel wordt al aangegeven dat het probleem veel te uitgebreid is om in één artikel te bespreken. Bovendien blijft het artikel binnen de mogelijkheden die de hulpmiddelen van vandaag bieden. Een meer diepgravende besprekking wordt beloofd, maar vervolgens kondigt Frans Remmen een artikel over het onderwerp aan en publiceert Stephen Cannan onverwacht een serie van twee artikelen over historie en SQL (zie sectie II). In de tussentijd heeft deze databasecolumnist de gelegenheid om met enkele collega's het historieprobleem nader te bestuderen en enkele ideeën over een oplossing voor historie uit te werken en te implementeren in werkende software. Dat feit rechtvaardigt op zichzelf al een terugkeer naar het historieprobleem. Er zijn echter enkele andere redenen die het noemen waard zijn.

Bij de grote automatiseringsbloopers die steeds vaker de pers halen gaat het vrijwel zonder uitzondering om systemen waarin de historieproblematiek een grote, zo niet dominerende rol speelt. Het defensie-salarissysteem, het systeem voor de bevolkingsadministratie, een pensioensysteem, een systeem voor ambtenarenuitkeringen, steeds gaat het om een administratieve systemen waarin historische gegevens (en historische regels) een grote rol spelen. Een vervolg op historie past dus in de actualiteit.

Tijd in context

De artikelen en presentaties die ondergetekende over het onderwerp historie heeft geschreven en gegeven wekken veel meer interesse en reacties op dan exposés over andere onderwerpen. Een vervolg op historie treft dus vermoedelijk een groep belangstellende lezers.

Database-leveranciers als Oracle en Informix die het tijdprobleem altijd hebben genegeerd claimen sinds kort oplossingen te hebben voor 'Time Series'-toepassingen. Daarmee kan het misplaatste idee ontstaan dat het historieprobleem is opgelost. We zullen laten zien dat de 'oplossingen' van deze leveranciers echter alleen interessant zijn voor simpele multimediatoe passingen (videoclips-in-database), niet voor complexe administratieve toepassingen. Een vervolg op historie kan dus dienen als vaccin tegen leverancierspraat.

De oplossingen die worden voorgesteld door wetenschappelijk onderlegde personen voldoen niet aan enkele uitgangspunten die naar onze mening essentieel zijn voor elke bruikbare oplossing voor het historieprobleem. Zelfs de goede theoretische oplossingen voor het historieprobleem zijn onvolledig. Wie zo iets roeft loopt het risico om het wetenschappelijke establishment (inclusief Stephen Cannan) over zich heen te krijgen en doet er verstandig aan de tijd te nemen. Een vervolg op historie kan dus weer leiden tot boeiende gedachtenwisselingen.

Twee versies van het versieprobleem

We beginnen met een situatie zonder enige vorm van historie in de vorm van een database die bestaat uit één enkele tabel die gegevens bevat over de medewerkers van een bedrijf:

Medewerker(Med#, Naam, Sekse, Adres, Woonplaats, Salaris)

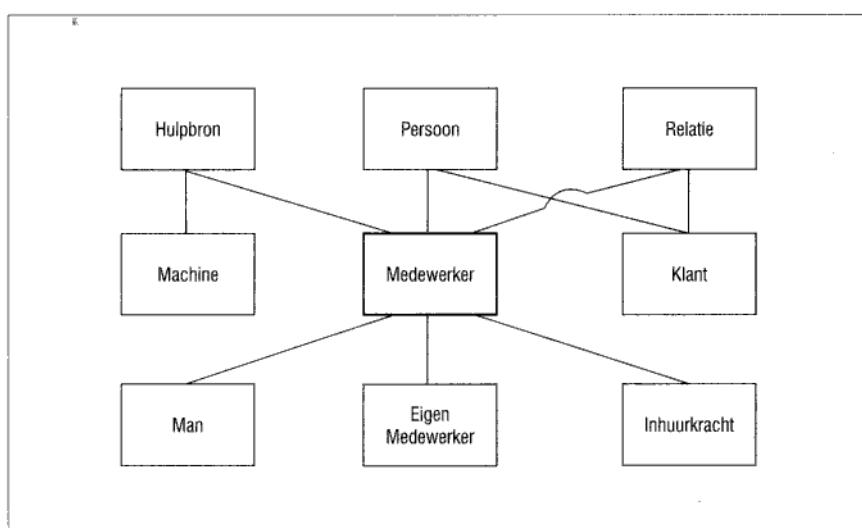
Zoals altijd correspondeert de tabel met een klasse van gelijksoortige objecten (hier: medewerkers van een bedrijf).

De attributen van de tabel corresponderen met de eigenschappen die alle objecten van deze klasse gemeen hebben (hier: code, NAW, sekse en salaris). De individuele objecten worden gerepresenteerd door de tupels (ofwel records) in de tabel. Als er perfecte één-op-één correspondenties bestaan tussen objectklassen/tabellen, eigenschappen/attributen en objecten/tupels een perfecte match opleveren dan is automatisering een feest. Er is in dat geval sprake van een perfecte classificatie. De systeemontwerper heeft de wereld opgedeeld in een eindig aantal klassen en elk individueel object dat de systeemgebruiker tegenkomt past 'zuigend' in precies één van die klassen. Sommige oude Grieken wisten echter al dat het niet goed mogelijk is om de wereld op te delen in elkaar niet overlappende objecten die elk voor zich weer kunnen worden verdeeld over een eindig aantal klassen.

Wat objecten zijn wordt bepaald door de waarnemer en individuele objecten kunnen worden ondergebracht in meerdere klassen tegelijk. In deze fuzzy wereld ontstaat dan altijd de vraag of twee objecten wel echt twee verschillende objecten zijn dan wel twee versies van eenzelfde object. Dit is het versieprobleem: het is onoplosbaar en het bepaalt de inhoud van een groot aantal beroepen. Automatiserders zijn net als politici, juristen, consultants en belastingadviseurs full time bezig met het versieprobleem. Beperken we ons tot ons eigen systeemontwerpvak dan zien we twee basisvarianten van het versieprobleem: één variant staat bekend als het generalisatieprobleem, de andere als het historieprobleem.

Het generalisatieprobleem is in dit boek niet aan de orde. De eenvoudigste variant van het generalisatieprobleem is besproken in de Tien Geboden voor goed database-ontwerp [15].

We zullen het generalisatieprobleem hier slechts aanstippen om de overeenkomst met het historieprobleem te tonen. Het generalisatieprobleem kunnen we laten zien door de Medewerker-kLASSE op te nemen in een netwerk van verwante klassen (zie figuur 1).



Figuur 1. Klassenhierarchieën (versie 1).

Figuur 1 laat zien dat de Medewerker-kLASSE een subklasse is van de klassen Hulpbron, Persoon en Relatie. De klasse van Medewerkers kan weer worden onderverdeeld in de meer verfijnde subklassen zoals Mannen, Eigen Medewerkers of Inhuurkrachten. Medewerkers en Machines zijn in zoverre hetzelfde dat ze vanuit een productiecontext beide kunnen worden gezien als Hulpbronnen. Medewerkers en Klanten zijn tegelijk (natuurlijke) Personen en Bedrijfsrelaties.

Tijd in context

In een relationele of flat-file context levert een situatie als deze grote modelleer- en programmeerproblemen op omdat de situatie waarin een object in de werkelijkheid precies wordt weergegeven door één tupel in de database niet meer kan worden gehandhaafd. Houden we vast aan de gedachte dat elke medewerker wordt gerepresenteerd door een Medewerker-tupel, dan moeten we rubrieken toevoegen aan de tabel Medewerker die relevant zijn voor alle sub- en superklassen waaraan de klasse van medewerkers is gerelateerd. De Medewerker-tabel kan er dan als volgt uit gaan zien:

```
Medewerker(Med#, Naam, Sekse, Adres, Woonplaats,  
Salaris, Kostprijs, KlantStatus, Ind_MilDnst,nd_Jaarcontract,  
Inhuurtarief)
```

Aan Medewerker zijn nu eigenschappen toegevoegd die goed beschouwd thuisoren bij de lagere of hogere objectklassen waaraan Medewerker is gerelateerd. Al deze attributen zijn een bron van problemen. De subklasse-attributen Ind_MilDnst (Man), Salaris en Ind_Jaarcontract (Eigen Medewerker) en Inhuurtarief (Inhuurkracht) leveren problemen op omdat ze op basis van allerlei complexe regels al dan niet verplicht gevuld of juist leeg (null) zijn. Aan de superklasse-attributen Naam, Sekse, Adres, Woonplaats (Persoon), Kostprijs (Hulpbron) en KlantStatus (Relatie) kleeft daarentegen het probleem dat ze ook terugkomen in andere tabellen zoals Machine, Klant en Leverancier. Het probleem met de Medewerker-tabel is dat een medewerker-in-de-werkelijkheid niet alle eigenschappen kent die volgens de tabel aan medewerkers worden toegeschreven. We kunnen natuurlijk ook kiezen voor het creëren van een tabel voor elke klasse in figuur 1. In dat geval worden de eigenschappen van een medewerker echter verdeeld over meerdere tupels en ook dat is niet koosjer. Elke modelleerwijze binnen een relationeel kader is problematisch doordat er geen perfecte afbeelding van objecten op tupels mogelijk is.

Het historieprobleem waarin we hier geïnteresseerd zijn is zoals gezegd nauw verwant aan het generalisatieprobleem. Ook bij dit probleem gaat het om het weergeven van objecten en versies-van-objecten. Ook hier lukt het niet om objecten één-op-één af te beelden op tupels. Gaat het bij generalisatie echter om klassen en subklassen (Medewerker-Inhuurkracht, Hulpbron-Medewerker, enzovoorts), bij historie gaat het om een onderverdeling van een object in tijdgebonden versies. Het probleem van de database-ontwerper is precies als bij generalisatie dat het onmogelijk is om tegelijk zicht te houden op het object en zijn tijdgebonden versies. En net als bij generalisatie moet de ontwerper kiezen tussen het modelleren van de tijdloze objecten of de tijdgebonden objectversies.

Wie geïnteresseerd is in het oplossen van het historieprobleem doet er goed aan om de verwantschap met het generalisatieprobleem en het achter beide problemen schuil gaande versieprobleem in het oog te houden. Wie er in

slaagt om de sleutel te vinden tot het historieprobleem heeft ook een gereedschap in handen om de andere problemen te lijf te gaan. In een volgend hoofdstuk komen we daar nog op terug.

De historie-praktijk: ontwijken en stapelen

We verlaten tijdelijk onze filosofische modus en keren terug naar de oorspronkelijke historieloze tabel Medewerker:

Medewerker (Med#, Naam, Sekse, Adres, Woonplaats, Salaris)

Veronderstel dat de gebruiker van het systeem inzicht wenst in het verloop van het salaris van de medewerkers. Eén oplossing voor dit probleem is dat de gebruiker hiervoor zorgt door op gezette tijden een lijst met medewerkergegevens uit te draaien en deze op te slaan in een kast voor nadere raadpleging. Natuurlijk is de kans groot dat de gebruiker de belangrijkste gegevens zal vastleggen in een spreadsheet of bij voldoende budget zal vragen om een datawarehouse. Deze 'oplossing' lijkt belachelijk maar verheugt zich op dit moment in een grote populariteit. De situatie kan worden getypeerd door de constatering dat het de gebruiker is die een historieprobleem heeft in plaats van de automatiseerder. Krijgt de gebruiker zijn warehouse dan is dit product primair een patch voor het historieprobleem. We zien hier voor het eerst dat historie een dure aangelegenheid is en dat we kunnen kiezen tussen organisatorische kosten en datawarehouse-gerelateerde kosten.

Laten we er van uitgaan dat de automatiseerder zijn verantwoordelijkheid neemt en de tabel Medewerker geschikt maakt voor historische salariessen. Een mogelijkheid om dit te bewerkstelligen is de volgende:

Medewerker (Med#, Naam, Sekse, Adres, Woonplaats, Salaris_jan, Salaris_feb, ..., Salaris_dec)

Deze oplossing lijkt nogal bizarre. Per maand wordt het salaris geregistreerd in de vorm van een repeterende groep. Het salaris kan voor maximaal één jaar worden vastgehouden, maar daaraan kan iets worden gedaan door de repeterende groep groter te maken. In elk geval moet de tabel op gezette tijden worden opgeschoond. Normalisatiefans zullen deze oplossing al helemaal verworpen omdat repeterende groepen in hun ogen verboden zijn en de nieuwe tabel Medewerker dus in de nulde normaalvorm is. Het is daarom al met al onwaarschijnlijk dat in dit geval de salariswens van de gebruiker op deze wijze wordt opgelost. Toch komt in een iets andere context deze foutieve modelleerwijze vaak voor:

Tijd in context

Afdeling(Afd#, Naam, Budget_jan, Budget_feb, ..., Budget_dec)

Veel ontwerpers vinden in dit geval deze modelleerwijze van historie wel gepast. In elk geval komen dergelijke nulde normaalvorm-constructies in financiële systemen regelmatig voor. Waarom is niet erg duidelijk. Het verschil met Medewerker.Salaris is hooguit dat een budget maandelijks wordt opgesteld, terwijl een salaris vaak een meer variabele looptijd heeft. Het argument dat een jaar twaalf maanden heeft en dat er dus geen sprake is van een repeterende groep vindt hopelijk iedereen belachelijk.

Een andere populaire oplossing voor de historievraag ziet er als volgt uit:

Medewerker(Med#, Datum_in, Naam, Sekse, Adres, Woonplaats, Salaris)

Bij deze modelleerwijze wordt definitief gekozen voor het niet meer vastleggen van objecten (medewerkers), maar van versies-van-objecten (medewerker-tijdversies). Een medewerker-in-de-werkelijkheid wordt nu gerepresenteerd door een aantal tupels, precies zoals bij het generalisatieprobleem. De tabelnaam 'Medewerker' dekt hoe dan ook de lading niet meer: de tabel bevat geen medewerkers maar ge-time-slicede versies van medewerkers. Deze oplossing werd reeds besproken in de Tien Geboden voor goed database-ontwerp [15], staat bekend als 'stapelen', is zeer populair en heeft desastreuze gevolgen.

In de eerste plaats kan het bezwaar worden gemaakt dat de gebruiker meer krijgt dan waarom hij heeft gevraagd. Naast een historisch salaris krijgt hij historische NAW-gegevens en geslachtsaanduidingen. Ook deze tabel Medewerker is niet goed genormaliseerd. Naam, Sekse, Adres en Woonplaats worden alleen bepaald door de rubriek Med#. Alleen Salaris wordt bepaald door de volledige primaire sleutel, te weten de combinatie van Med# en Datum_In. Volgens de normalisatietheorie is er dan sprake van attributen die functioneel afhankelijk zijn van een deel van de primaire sleutel en de tabel Medewerker heet dan in de eerste normaalvorm (1NF) te zijn. Praktijkmensen zeggen dan dat de rubrieken Naam, Sekse, Adres en Woonplaats redundantie bevatten en volgens de theorie is dat fout.

Hierboven is al aangegeven dat deze modelleerwijze hoogst onverstandig is, maar de bezwaren van normaliseerders en praktijkmensen vormen daarvoor niet de reden. Wie de tabel Medewerker stapelt kan teruggaan naar de eindgebruiker en roepen dat de historie kan worden geregistreerd van alle eigenschappen. Deze handelwijze wordt doorgaans beloond met de warme genegenheid van de eindgebruikers die een latente maar onverzadigbare behoefte hebben aan volle databases en lege bureauladen. Daarmee is de tabel medewerker weer volledig genormaliseerd en zijn de theoretici uit beeld. De

praktijkmensen mompelen misschien nog iets over spilziek geheugengebruik en redundantie. Ze zullen er op wijzen dat een verandering van salaris leidt tot een kopie van de onveranderde NAW en sekse-gegevens. Het enige goede antwoord op die bezwaren luidt "en wat dan nog?" Zolang de medewerker-attributen geen BLOB's bevatten (zoals een gedigitaliseerde foto van de medewerker) en de medewerker niet elke dag verhuist, van naam of sekse verandert en opslag krijgt is er geen vuiltje aan de lucht. Als er al sprake is van redundantie — en ik vind eigenlijk van niet — dan levert die geen problemen op omdat er geen inconsistenties in de database kunnen voorkomen.

We hebben nu voldoende advocaat van de duivel gespeeld. Voordat we beschuldigd worden van het lidmaatschap van een criminale organisatie volgt hier het echte bezwaar. Dit bezwaar luidt dat er in de medewerkerdatabase altijd behoeft bestaat om achter de veranderlijke medewerkerversies het onveranderlijke medewerkerobject te blijven zien. Dit nu is bij stapelen niet mogelijk.

Het volgende voorbeeld maakt dat pijnlijk duidelijk. In dit voorbeeld voegen we informatie toe over de inzet van medewerkers op projecten:

```
Medewerker(Med#, Datum_in, Naam, Sekse, Adres, Woonplaats, Salaris)
Project(Proj#, Naam, budget)
Inzet(Med#, Datum_in, Proj#, Tarief)
```

We willen de Inzet-tupels laten verwijzen naar de tabellen Project en Medewerker. Het eerste levert geen probleem op omdat we besloten hebben om historie te negeren. De verwijzing naar Medewerker is echter onmogelijk: we kunnen slechts naar medewerkerversies verwijzen. Wie denkt dat Inzet.Datum_in iets zegt over de datum waarop de medewerker op het project werd ingezet vergist zich deerlijk. Het attribuut verwijst naar de actuele, de eerste of de laatste medewerkerversie. Welke versie ook wordt uitverkoren, in alle gevallen is er veel nutteloos programmeerwerk te doen. De oplossing is al met al dermate bizarre dat ik hem nog nimmer heb aangetroffen in de praktijk (en dat wil wat zeggen). De stapelaars van deze wereld doen allemaal het volgende:

```
Inzet(Med#, Proj#, Tarief)
```

Wat is hier gebeurd? De ontwerper heeft gekozen voor het laten vallen van het concept van referentiële integriteit. Of beter, hij heeft dit concept vervangen door een stapel-historie-variant die het relationele model niet erkent en rdbms'en niet ondersteunen:

Tijd in context

Referentiële integriteit (standaardversie)

Voor een set rubrieken van een tabel geldt de regel dat de combinatie van waarden in elk tupel overeenkomt met de waarden in een corresponderende set attributen van een bepaalde tabel, waarbij geldt dat die attributen de primaire sleutel van de betreffende tabel vormen.

Referentiële integriteit (stapel-historie-versie)

Voor een set rubrieken van een tabel geldt de regel dat de combinatie van waarden in elk tupel overeenkomt met de waarden in een corresponderende set attributen van een bepaalde tabel, waarbij geldt dat die attributen de primaire sleutel van de betreffende tabel vormen met aftrek van de historie-attributen.

Nog een praktijk: schaduwen

Naast weglopen, datawarehouses en stapelen is er nog een oplossing die op zichzelf fout maar in elk geval begrijpelijk is. Die oplossing zullen we betitelen als 'schaduwen' en komt neer op het naast elkaar modelleren van het object en zijn versies.

De Medewerker-casus komt er dan als volgt uit te zien.

```
Medewerker(Med#, Naam, Sekse, Adres, Woonplaats, Salaris)
Med_schaduw(Med#, Datum_in, Naam, Sekse, Adres, Woonplaats, Salaris)
```

De schaduwoplossing heeft van alle verkeerde oplossingen de charme dat het onderscheid tussen objecten en versies explicet wordt onderkend. De tabel Medewerker bevat de actuele stand van zaken, Med_schaduw alle voorgaande en toekomstige versies. Van referentiële integriteitsproblemen is geen sprake maar dit is niet zonder schaduwzijden:

- Elke Insert of Update op Medewerker moet worden voorafgegaan door een Insert in Med_schaduw. Dat is minder goed voor de performance.
- Hoe moeten we omgaan met een Delete-operatie als Med_schaduw.Med# een verwijzende sleutel is naar Medewerker?
- Ontwerpers motiveren de schaduwoplossing met het argument dat de meeste database-query's geïnteresseerd zijn in de actualiteit en dus kunnen volstaan met het benaderen van de tabel Medewerker. Dat zal zo zijn, maar voor elke vraag die ook op de toekomst of het verleden moet worden losgelaten dient nu een aparte query te worden geschreven.
- Integriteitsmanagement is alleen praktisch voor actuele medewerkergegevens. Worden alleen actuele gegevens gemuteerd dan is dit acceptabel. In de vele gevallen waarin het gewenst is om gegevens in het verleden of de toekomst te muteren verwordt het systeem tot weinig meer dan een kaartenbak.

- Toekomstmutaties in Med_Schaduw dienen op een moment te worden verwerkt in de tabel Medewerker. Dit vereist het bestaan van een database-alerter of ontwikkeling van eigen scheduling-programmatuur.

Item 4 is de kern van het bezwaar dat elke database-ontwerper tegen de schaduwoplossing zou moeten hebben. De andere bezwaren volgen hieruit of zijn niet onoverkomelijk.

De minst slechte benadering: objecten èn objectversies

Dit inleidende hoofdstuk sluiten we af met de enig juiste benadering, die zoals we in het volgende hoofdstuk zullen zien, eveneens weinig enthousiasme kan wekken. Het goede van de schaduwbenadering wordt hier overgenomen, de oude normalisatiebeginselen worden onverkort toegepast en we houden vast aan alle zegeningen van moderne rdbms-technologie.

```
Medewerker(Med#, Naam, Sekse, Adres, Woonplaats)
Med_Salhis(Med#, Datum_in, Salaris)
```

Het lijkt zo eenvoudig. Waarom kiest niet iedereen voor deze eenvoudige modelleerwijze. Het voor de hand liggende antwoord op die vraag is de al genoemde behoefte van de eindgebruiker aan meer historie dan alleen maar het salaris. De veeleisende gebruiker noopt tot ver doorgevoerde vormen van historie:

```
Medewerker(Med#)
Med_Naamhis(Med#, Datum_in, Naam)
Med_Seksehis(Med#, Datum_in, Sekse)
Med_AWhis(Med#, Datum_in, Adres, Woonplaats)
Med_Salhis(Med#, Datum_in, Salaris)
```

Is dit goed? En zo ja, is het nog leuk? Is dit alles of ligt er nog meer historie-ellende op ons te wachten.

Een geschiedenis van ruimte en tijd

Eens in de zoveel tijd verschijnt er een DB/M met een aantal artikelen dat er haast om vraagt na een jaar of vijf nog eens opnieuw te worden gelezen. Wat mij betreft was DB/M nummer 5 van 1997 zo'n uitgave. Objectrelatieel Oracle8, nieuwe gegevenstypen in DB2 Universal Database, het uur der waarheid voor Informix met zijn Universal Server, wilde OO SQL-extensies en ongetwijfeld een aantal onderwerpen waar we over vijf jaar vermoedelijk nog slechts meevarig om zullen glimlachen. Verder waarschuwt DB/M hoofdredacteur Herbert Boland in datzelfde nummer nogmaals voor het einde van de database-onafhankelijkheid (zegt het voort!) en beweert ondergetekende dat er voor het probleem van het administreren van tijdgebonden gegevens nog geen glimp van een op tools gebaseerde oplossing in zicht is. We spreken elkaar nader over vijf jaar!

Ook vanuit het rustiger perspectief van een inleidend artikel in een serie (zie het eerste hoofdstuk van deze uitgave) werd ik op mijn wenken bediend. Mijn bewering was dat we historie zo veel mogelijk vermijden en waar dat niet mogelijk is doorgaans kiezen voor het zogenaamde 'stapelen', waarbij we veel moois dat relationele databases bieden (met name referentiële integriteit) overboord gooien. Wie schetst mijn verrukking wanneer ik in een artikel over de opslag van geografische informatie door het Kadaster (DB/M 5/97) lees dat men zich zonder enige vorm van gêne overgeeft aan deze praktijk. Ik citeer gedeeltelijk blz. 15:

"... Dit gebeurt door aan ieder object twee attributen toe te voegen: tmin en tmax. Een objectbeschrijving is nu valide vanaf (...) tijdstip tmin en tot (...) tijdstip tmax. Actuele tijdbeschrijvingen kennen een speciale waarde die groter is dan iedere andere tijdsWaarde: MAX_TIME. [...] Wanneer nu een nieuw object wordt opgenomen dan krijgt tmin de dan geldende datum en wordt tmax op MAX_TIME gezet. Wanneer één van de attributen van een bestaand object verandert, wordt niet alleen dit attribuut bijgewerkt, maar wordt het gehele record inclusief gewijzigd attribuut en object_id gekopieerd. In het

oude record wordt tmax vervolgens op de dan geldende datum gezet, net als tmin in het nieuwe record. ..."

Dat het Kadaster een joekel van een database krijgt behoeft geen betoog. Interessanter is dat het Kadaster zo te zien geen tabel kent waarin het object_id zonder toegevoegde historie-attributen (in elk geval tmin) de primaire sleutel is. Bij de uitvoering van de primaire taak van het Kadaster is het noodzakelijk om de relatie te leggen tussen een perceel (object) en de (rechts)personen die in de loop van de tijd het perceel in eigendom hebben. We veronderstellen daarbij dat een (rechts)persoon meerdere percelen in eigendom kan hebben en dat het ook kan voorkomen dat een perceel tegelijk meerdere eigenaren heeft.

Zonder historie is dit geen enkel probleem:

```
Object(object_id, ...)
Eigendom(eig_id, object_id)
Eigenaar(eig_id, object_id, NAW, ... )
```

Uiteraard moet het Kadaster ons kunnen vertellen op welke perioden het eigendom betrekking heeft. Maken we alleen de tabel Eigenaar historisch dan is er nog geen groot probleem:

```
Object(object_id, ...)
Eigendom(eig_id, object_id, tmin, tmax)
Eigenaar(eig_id, ... )
```

We moeten nu uiteraard wel een constraint toevoegen die afdwingt dat er geen overlap van eigendom mag zijn, maar alla. Echt leuk wordt het pas na toevoeging van historie aan de gegevensklasse Object en (ter verhoging van het realiteitsgehalte) aan de tabel Eigenaar:

```
Object(object_id, tmin, tmax, ...)
Eigendom(eig_id, object_id, tmin, tmax)
Eigenaar(eig_id, tmin, tmax, ... )
```

Van Kadaster-kaartenbak naar Kadaster-database

Het database-ontwerp is nu bestand tegen iedere wijziging in de constellatie van eigenaars en percelen. Bouw een schuurtje op uw terrein, sta een lapje grond af voor de Betuwelijn, verander van naam, overlijd, het Kadaster kan het allemaal registreren. Wel jammer dat zelfs dit zeer eenvoudige gegevensmodel een groot aantal complexe constraints bevat waarvoor relationele

Een geschiedenis van ruimte en tijd

dbms'en geen goede oplossing bieden. Jammer ook dat de relationele basisconstraints die worden uitgedrukt door middel van primaire en verwijzende sleutels een heel andere lading hebben gekregen (zie hoofdstuk 1). De volgende regels moeten in elk geval worden afgedwongen om te komen tot een database die onbestaanbare situaties uitsluit:

- Object.tmin < Object.tmax
- Eigendom.tmin < Eigendom.tmax
- Eigenaar.tmin < Eigenaar.tmax
- Voor elke set Object-tupels met eenzelfde object_id geldt dat de diverse tmin-tmax intervallen geen overlap mogen vertonen (en mogelijk zelfs moeten aansluiten).
- Voor elke set Eigendom-tupels met dezelfde waarden voor eig_id en object_id geldt dat de diverse tmin-tmax-intervallen geen overlap mogen vertonen.
- Voor elke set Eigenaar-tupels met eenzelfde eig_id geldt dat de diverse tmin-tmax-intervallen geen overlap mogen vertonen (en mogelijk zelfs moeten aansluiten).
- Voor elk tijdstip op het interval tmin-tmax voor een Eigendom-tupel dient er een geldig Object-tupel voor te komen.
- Voor elk tijdstip op het interval tmin-tmax voor een Eigendom-tupel dient er een geldig Eigenaar-tupel voor te komen.

Probeer deze constraints maar eens in SQL uit te drukken om echt een goed gevoel te krijgen voor welk historieprobleem het Kadaster staat. Van referentiële integriteit is in elk geval geen sprake meer: de plaats van de door het rdbms ondersteunde verwijzende sleutels Eigendom-Object en Eigendom-Eigenaar is ingenomen door de uiterst complexe constraints. Voor elk tijdstip op het interval tmin-tmax voor een Eigendom-tupel dient er een geldig Object-tupel voor te komen. Hieronder geven we de SQL-statements voor de verwijzing van Eigendom naar Object:

Geen historie, normale referentiële integriteit

```
SELECT "Object", e.object_id, "van eigenaar", e.eig_cd, "bestaat niet"
FROM Eigendom e
WHERE NOT EXISTS
    (SELECT 'x' FROM object o
     WHERE e.object_id = o.object_id
    )
```

Is dit een referentiële integriteitsstatement? Ja, want het selecten van alle eigendommen voor een bepaalde object_id kan alleen maar geslaagd zijn als er voor elke eigendom een object_id bestaat. Is dit een referentiële integriteitsstatement? Neen, want het selecten van alle eigendommen voor een bepaalde object_id kan alleen maar geslaagd zijn als er voor elke eigendom een object_id bestaat. Is dit een referentiële integriteitsstatement? Neen, want het selecten van alle eigendommen voor een bepaalde object_id kan alleen maar geslaagd zijn als er voor elke eigendom een object_id bestaat.

Gemodificeerde referentiële integriteit: historie met aansluiting intervallen

```
SELECT "Object", e.object_id, "van eigenaar", e.eig_cd,
       "bestaat niet gedurende (deel van) interval", e.tmin, "-",
       e.tmax
  FROM Eigendom e
 WHERE NOT EXISTS
   (SELECT 'x' FROM object o
    WHERE e.object_id = o.object_id
    AND o.tmin <= e.tmin
   )
 OR NOT EXISTS
   (SELECT 'X' FROM object o
    WHERE e.object_id=o.object_id
    AND o.tmax >= e.tmax
   )
```

Bij gebrek aan ruimte laat ik het aan de lezer over om het SQL-statement uit te werken voor de situatie waarin de objectversies niet hoeven aan te sluiten. Kom, probeer ze eens in SQL uit te drukken en stuur gerust een e-mailtje naar ondergetekende of het Kadaster wanneer het niet lukt.

De stapel-oplossing waarvoor het Kadaster heeft gekozen leidt ook tot problemen met het nog fundamentele begrip van primaire-sleutelintegriteit (vaak ten onrechte entiteit-integriteit genoemd). Door het ontbreken van tabellen waarin tijdloze percelen en eigenaars zijn vastgelegd, is het denkbaar dat eigenaars en percelen een code krijgen die al eens eerder is gebruikt of zelfs nog in gebruik is. Voor het systeem gaat het in dat geval om een en hetzelfde perceel of (rechts)persoon en het is maar de vraag of dat terecht is. Je moet er toch niet aan denken de eigenaarsidentificatie van de Hakkelaar te 'erven', of — nog erger — dat je nieuwe koopwoning het object_id van Johan V's stulpje krijgt. Een verkeerde insert en je bent lid van een criminale organisatie.

Worden alle bovenstaande constraints afgedwongen dan is de kans op fouten als bovenstaande klein, zeker wanneer er wordt geëist dat intervallen tussen percelen en eigenaars moeten aansluiten. Ik moet de eerste historische databasetoepassing waarin alle regels van de bovenstaande soorten goed worden gecontroleerd overigens nog tegenkomen. Mijn ervaring is dat integriteitsmanagement meestal beperkt wordt geïmplementeerd en dat met name mutaties in de toekomst en in het verleden al snel tot inconsistenties leiden. Wellicht valt het risico voor het Kadaster mee, maar voor financiële toepassingen weet ik dat de risico's levensgroot zijn. Ik ken een uitkeringsysteem waarbij alle uitkeringen boven een bepaald bedrag met de hand worden nagerekend, iedere maand weer. Ik weet ook van een gepensio-

Een geschiedenis van ruimte en tijd

neerde die voor een klein baantje dat hij lang geleden kort heeft gehad een pensioen van enkele duizenden guldens krijgt uitgekeerd, elke maand weer. Automatisering en historie lijken elkaar niet goed te verdragen.

Alternatieven voor het Kadaster

De Kadaster-casus is zo interessant dat ze voorlopig nog even niet van mij af zijn. Laat niemand echter denken dat ik onze geografische vakbroeders wil afkammen of voor incompetent wil verklaren. In de eerste plaats is historie voor hun maar bijzaak.

Het echte probleem is het representeren van geografische informatie, ook al zo'n probleem waarmee het relationele model en relationele dbms'en geen raad weten. Mijn verzoekplaatje voor het Kadaster is dan ook Sam Cooke's hit "Don't know much about history; don't know much 'bout geography". Arm Kadaster.

Ook in het Kadaster-artikel is historie maar bijzaak. Zou het gaan om een pensioenfonds, een salarisverwerker of een uitkeringsverstrekker dan zou mijn oordeel minder mild zijn. Voor hun is historie hoofdzaak en mag meer worden verwacht. Een andere reden om het Kadaster niet al te hard te valLEN is dat deze ex-overheidsinstelling er in is geslaagd zijn tarieven aanzienlijk te verlagen. Kom daar maar eens om bij een bedrijf zonder concurrenten.

Nog meer begrip voor het Kadaster ontstaat bij het beoordelen van de alternatieven die in het voorgaande hoofdstuk zijn besproken. Die alternatieven zijn:

- sla geen historie op
- maak een separaat datawarehouse
- werk met repeating groups
- werk met schaduwtabellen
- werk met objecten en historische versies.

De eerste twee alternatieven zijn hier duidelijk onacceptabel. Het kunnen beantwoorden van vragen als "wie was in 1920 de eigenaar van het terrein dat nu bekend staat als het Utrechtse Griftpark" is de core business van het Kadaster. Alternatief werk met repeating groups leidt tot het onderstaande monstrum:

```
Object(object_id, tmin_obj, tmax_obj, ..., eig_id_1, tmin_eig_1,
tmax_eig_1, ..., ..., eig_id_n, tmin_eig_n, tmax_eig_n. ...)
```

Omdat ik mij niet kan voorstellen dat iemand buiten de wereld van financiële systemen iets dergelijks bedenkt laat ik een nadere bespreking achterwege.

Het alternatief 'werk met schaduwtabellen' was nog wel voorstelbaar geweest. Het Kadaster voorbeeld zou dan de volgende vorm hebben aangenomen:

```
Object(object_id, ...)  
Schaduw_Object(object_id, tmin, tmax, ...)  
Eigendom(eig_id, object_id, tmin, tmax)  
Schaduw_Eigendom(eig_id, object_id, tmin, tmax)  
Eigenaar(eig_id, ... )  
Schaduw_Eigenaar(eig_id, tmin, tmax, ... )
```

De actuele kadastrale gegevens zijn hier opgenomen in een eenvoudig database waarin alle fundamentele regels zijn vormgegeven door middel van primaire en verwijzende sleutels. Naar elke tabel wordt verwezen door een schaduwtafel die er precies zo uitziet als de hierboven beschreven tabellen. Wie de Kadaster-casus goed leest begrijpt waarom deze oplossing geen soelaas biedt: het Kadaster wil in staat zijn om gegevens te wijzigen voor andere tijdstippen dan vandaag en alle vragen die voor het heden worden gesteld kunnen even goed worden gesteld voor het verleden of de toekomst. Voor mutaties in het verleden en de toekomst is het integriteitsmanagement niets eenvoudiger geworden en het is al evenmin erg aantrekkelijk om voor elke informatie behoeft twee query's te schrijven, één voor het heden en één voor het niet-heden.

Wat resteert is het alternatief 'werk met objecten en historische versies', de oplossing die in het vorige hoofdstuk als 'goed' werd aangemerkt:

```
Object(object_id)  
Object_Vorm(object_id, tmin, tmax, <begrenzingsgegevens>)  
Object_Hyp(object_id, tmin, tmax, <hypothekgegevens>)  
Object_Pacht(object_id, tmin, tmax, <erfpachtgegevens>)  
Object_Gemeente(object_id, tmin, tmax, <gemeentegegevens>)  
  
Eigendom(eig_id, object_id)  
Eigendom_Periode(eig_id, object_id, tmin, tmax)  
  
Eigenaar(eig_id)  
Eigenaar_Naam(eig_id, tmin, tmax, <naamgegevens>)  
Eigenaar_AW(eig_id, tmin, tmax, <adres/woonplaatsgegevens>)
```

We zien hier een onderscheid tussen tijdonafhankelijke gegevensklassen (Object, Eigendom, Eigenaar) en tijdafhankelijke gegevensklassen die het verloop van bepaalde attributen of groepen attributen weergeven. Het tijloze feit van het bestaan van het object

Een geschiedenis van ruimte en tijd

(perceel, eigenaar of eigendomsrelatie) is hier losgekoppeld van het voorbijgaande bestaan en de voorbijgaande waarde van eigenschappen. Ik heb voor de klassen Object en Eigenaar maar enkele groepen verzonnen die voor de hand liggen of waarvan het artikel gewag maakt. Aan de gegevensklasse eigenaar zijn slechts NAW-gegevens toegevoegd. Omdat naamgegevens enerzijds en adres/woonplaatsgegevens anderzijds los van elkaar plegen te wijzigen, zijn ze opgenomen in separate tabellen.

Interessant is de wijze waarop de tabel Eigendom is aangepakt. In het 'historische' hoofdstuk in Tien Geboden voor goed database-ontwerp [15] gaf ik aan dat een dergelijke splitsing strikt genomen niet noodzakelijk is en de tabel Eigendom_Periode dus overbodig is. Hoewel die observatie nog steeds geldt, heeft de hier gekozen oplossing het voordeel dat er onderscheid wordt gemaakt tussen het bestaan van een eigendomsverhouding en de periode waarin een eigendomsverhouding gold. Dit voordeel speelt met name bij de situatie waarin eenzelfde persoon (een speculant?) diverse malen eigenaar is van eenzelfde perceel. Bij de bespreking van onze grensverleggende oplossing voor historie komen we nog terug op deze 'Heintje Davids'-variant van het historieprobleem.

Ik weet niet hoe het de lezer vergaat, maar het zou me niet verbazen wanneer deze steeds meer begrip heeft gekregen voor het Kadaster. Zelfs het enige redelijke alternatief, werk met objecten en historische versies, oogt niet aantrekkelijk. Natuurlijk is het een voordeel om de rdbms-features die basisconcepten zoals primaire en verwijzende sleutels ondersteunen te kunnen blijven gebruiken, maar elke vraag leidt nu tot een veelheid aan join-operaties. Ook nu nog geldt een veelheid aan tmin/tmax-constraints en dient daarnaast nog steeds te worden afgedwongen dat een eigendom-verwijzing naar een perceel of een eigenaar betrekking heeft op een voor dat interval bestaand perceel en een bestaande eigenaar. Daarmee is niet gezegd dat het Kadaster goed bezig is. Zeker bij een database met veel soorten gegevens is het alternatief 'werk met objecten en historische versies' ondanks alles beter. Op basis van al deze verzachtende omstandigheden komen we slechts tot een voorwaardelijke veroordeling. De echte schuldigen heten niet "Kadaster", "bank", "salarisverwerker", "pensioenfonds" en "uitkeringsinstantie", maar Codd, Oracle, Informix, IBM en — speciaal voor het Kadaster — Computer Associates. Het Kadaster is het slachtoffer van zijn omgeving.

Barbra Streisand meets Het Kadaster

Het Kadaster houdt zich bezig met lijnen waarmee oppervlakken worden samengesteld en het Kadaster-artikel gaat primair over het vastleggen van oppervlakken door middel van lijnstukken. Historie lijkt daarmee vergeleken maar een eenvoudig probleem. Tijd is te beschouwen als een rechte lijn die nergens begint en nergens eindigt. "No vestige of a beginning, no prospect of an end", zo typeert Stephen J. Gould in zijn boek "Time's Arrow, Time's cycle" het westerse wetenschappelijke denken over het wezen van de tijd.



11

Christelijke theologen zagen de tijd ook als lineair, zij het met een duidelijk begin in de schepping (4004 voor Christus volgens de schriftgeleerden) en een duidelijk eind met de wederkomst van Christus. Andere culturen beschouwden de tijd als cyclisch: alles wat gebeurt herhaalt zich voortdurend, er is niets nieuws onder de zon en alle dingen zijn onuitsprekelijk vermoedend. Sinds enkele decennia weten we dat de tijd circa vijftien miljard jaar geleden begon met de 'Big Bang' en mogelijk zal eindigen met een 'Big Crunch'. Vragen wat daarvoor gebeurde en daarna zal gebeuren is zinloos. We weten dankzij Einstein ook dat tijd afhankelijk is van relatieve plaats en snelheid en dat een intergalactisch kadaster pas echt een nachtmerrie oplevert.

Ik geef dit kleine exposé niet alleen omdat het leuke stof oplevert voor feesten en partijen, maar om te laten zien dat elke cultuur de tijd beschouwt als lineair: startpunten, eindpunten en loopjes doen daar niets aan af. Voor sommigen, in het bijzonder Barbra Streisand en een aantal automatiserders, ligt het echter heel anders. Ik citeer Barbra's chanson "Memories" van haar CD One Voice:

...Can it be that life was all so simple then
Or has time rewritten every line
If we had the chance to do it all again
Tell me, would we, ohohoh could we? ...

Barbra S. weet wat de automatiserders van uw pensioenfonds, uw salaris-administratie, uw overheid en ons aller Kadaster weten. Hierbij verklap ik dit revolutionaire inzicht: tijd is niet lineair, tijd is een oppervlak! Dat het Kadaster hiervan op de hoogte is blijkt uit het artikel in DB/M 5/97. Uit het Kadaster-citaat op pagina 19 heb ik uit didactische overwegingen één klein zinnetje weggelaten (weergegeven als [...]):
"Daarnaast wordt onderscheid gemaakt tussen het tijdstip waarop een object in de database werd aangepast en het tijdstip waarop dit in de werkelijkheid — dus 'in het veld' — is waargenomen. Dit laatste wordt aangegeven met object_dt."

Dit zinnetje bevat een explosieve lading die een geheel nieuwe dimensie aan het historieprobleem toevoegt. Zojuist, toen historie bij het Kadaster nog een simpele lijn was, konden we bijvoorbeeld de volgende vraag stellen: "Wie was op 1 januari 1990 de eigenaar van perceel Loosdrecht/Sectie-C/5033?". Antwoord: de heer La Housse.

Die vraag is een eenvoudig geval van een meer algemene vraag waarvan onderstaand een voorbeeld wordt gegeven: "Wie was op 1 januari 1990 de eigenaar van perceel Loosdrecht/Sectie-C/5033 volgens de kennis die het Kadaster bezat op 15 februari 1990?". Antwoord: ik weet het niet en het Kadaster vermoedelijk evenmin.

Een geschiedenis van ruimte en tijd

Vragen als deze zijn aan de orde van de dag bij systemen waarin historie werkelijk belangrijk is en de beantwoording ervan kost zoveel geld dat we er alleen iets aan doen als het echt niet anders kan. Een alleraardigste test kan worden uitgevoerd door samen met twee collega's contact op te nemen met uw pensioenfonds. Elke persoon stelt één van de volgende vragen:

- Mijn salaris is met terugwerkende kracht tot 1 januari 1997 met f 500,— verhoogd. Kunt u mij aangeven hoeveel premie (backservice) er nu extra moet worden betaald?
- U heeft mijn geboortedatum verkeerd geadministreerd. Ik ben precies twee jaar ouder dan u denkt. Hoeveel premie heb ik te veel of te weinig betaald?
- Ik heb vorig jaar mijn geslacht operatief veranderd van vrouw naar man en moet dat nog doorgeven voor uw administratie. Hoeveel premie krijg ik terug nu u weet dat ik drie jaar minder te leven heb?

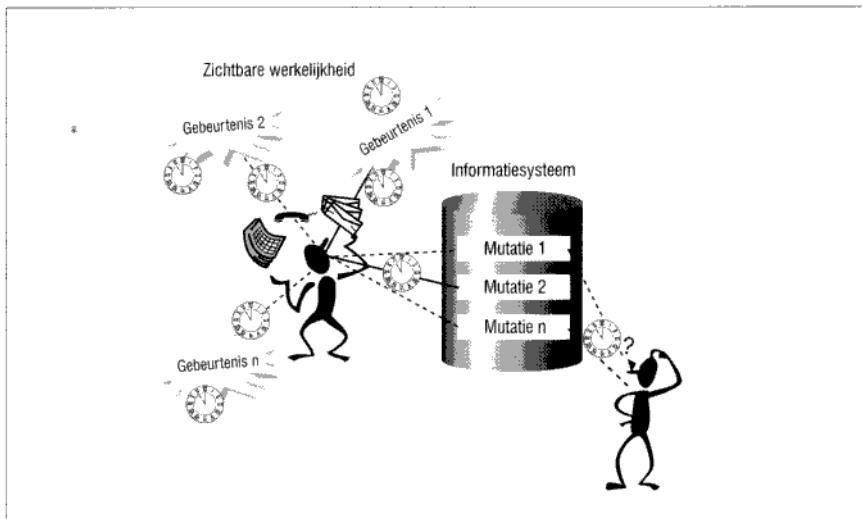
Alle drie beschreven situaties komen voor. U voelt vermoedelijk aan dat hier drie keer precies dezelfde vraag wordt gesteld en dat is ook zo. De administratie moet vanaf een bepaald punt in de tijd worden gecorrigeerd op basis van een foutief gegeven. Echter, de eerste vraag (salarisverhoging met terugwerkende kracht) kan door elk pensioenfonds zo worden beantwoord. De vraag over het verkeerd administreren van de geboortedatum lukt meestal ook nog wel. Bij de vraag over de geslachtsverandering wordt in veel gevallen teruggerept op perkament en ganzenveer, meestal op basis van het excus dat uw vraag wel heel bijzonder is. Voorbeelden als deze laten zien dat historie per geval wordt opgelost, desnoods vele malen binnen een en hetzelfde systeem.

De wetenschap: valid time en transaction time

Zoals gezegd is het tweedimensionale karakter van de tijd onder informatici bekend. De onvermijdelijkheid van dit verschijnsel wordt duidelijk wanneer we fundamenteel kijken naar het karakter van een informatiesysteem. Waarom bouwen we eigenlijk informatiesystemen? Een student informatica of informatiekunde zal vermoedelijk antwoorden dat een informatiesysteem menselijke arbeid automatisert, maar dat is voor administratieve systemen maar zeer gedeeltelijk waar. Het is juister om een administratief systeem te zien als een surrogaatwerkelijkheid. De echte werkelijkheid verandert voortdurend en het systeem dient deze veranderingen steeds te weerspiegelen. Figuur 2 beeldt dit proces van 'data capture' en 'data retrieval' plastisch uit. Data retrieval gebeurt bij voorziene veranderingen door eindgebruikers (mutaties) en bij onvoorziene veranderingen door automatiseerders (onderhoud). Beide activiteiten kosten handenvol geld en de vraag dringt zich op waarom we de moeite nemen om met veel moeite een kopie van de werkelijkheid te trekken als de werkelijkheid zelf ook kan worden geraadpleegd. Het antwoord op die vraag is dat de echte werkelijkheid veelal veel minder toegankelijk is dan de surrogaatwerkelijkheid van het informatiesysteem.



Zelfs als het systeem niet veel meer is dan een kaartenbak worden de kosten van systeemontwikkeling, -onderhoud en -gebruik vaak terugverdiend. Figuur 2 laat echter al zien dat we altijd een prijs betalen voor het gebruik van een informatiesysteem als 'communication enabler' en dat de kennis van de werkelijkheid zelf ook geen probleemloze zaak is. In de eerste plaats is er een tijdsverloop tussen het tijdstip waarop een gebeurtenis optreedt en het tijdstip waarop die gebeurtenis wordt waargenomen (het linkermannetje in figuur 2). Beide tijdstippen zijn in beginsel van belang: het waarnemen van een gebeurtenis is zelf ook een gebeurtenis. Overigens hoeft dit tijdsverloop niet aan de gebruiker te liggen. Denk maar aan een situatie waarin het Kadaster door traagheid van de notaris pas na een maand doorkrijgt dat er een perceel is verkocht. Of nog fundamenteeler: denk aan een situatie waarin in maart 1998 wordt afgesproken dat de werknemers een nieuwe CAO krijgen die ingaat op 1 januari 1998.



Figuur 2. De oorsprong van meerdimensionale tijd.

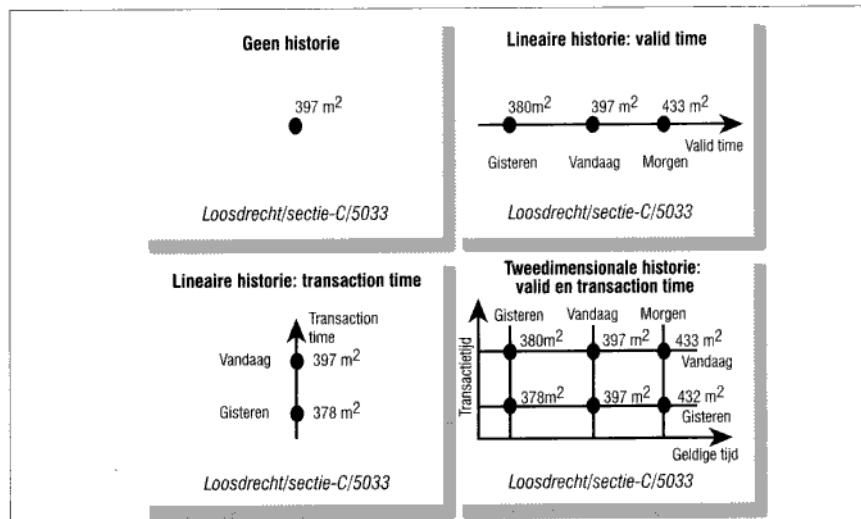
Is een gebeurtenis eenmaal waargenomen, dan zal deze worden vastgelegd in het informatiesysteem. Ook tussen waarneming en transactie is er weer sprake van tijdsverloop. Vervolgens zullen andere gebruikers kennis nemen van de gewijzigde werkelijkheid door raadpleging van het informatiesysteem. Er is geen verschil tussen het waarnemen van een gebeurtenis in de werkelijkheid en in het informatiesysteem: beide zijn gebeurtenissen op zichzelf en als zodanig de moeite van het vastleggen waard. Idealiter zouden dus de volgende soorten tijdstippen moeten worden vastgelegd:

- het tijdstip waarop een gebeurtenis plaatsvindt (valid time of geldige tijd)

Een geschiedenis van ruimte en tijd

- het tijdstip waarop een gebeurtenis aan het systeem kenbaar is gemaakt (transaction time of mutatietijd)
- het tijdstip waarop een verandering wordt waargenomen in de (surrogaat)werkelijkheid door een systeemgebruiker (observation time).

De modale informaticawetenschapper erkent slechts de tijdsoorten valid time en transaction time. Erkenning van tijdsoort observation time (mijn term) vereist onder meer vastlegging van alle raadpleegoperaties op een database. De informaticawetenschap is nog niet rijp voor de haast psychadelische effecten die ondersteuning van observation time met zich meebrengt. En uiteraard is de techniek nog lang niet toe aan het vastleggen van alle raadpleegoperaties. 'Aangetekend' communiceren door middel van informatiesystemen en weten wie wat had kunnen weten zal nog heel lang een droom blijven. In de tussentijd moeten we het bij gebrek aan tool-ondersteuning doen met de nachtmerrie van tweedimensionale tijd. Daarbij is elk tijdloos feit — zoals het bestaan van perceel X of eigenaar Y — ondergebracht in de valid time- en transaction time-dimensies. Figuur 3 toont de registratie van de oppervlakte van perceel Loosdrecht/sectie-C/5033 voor diverse plaatsen op het kadastrale tijdoppervlak.



Figuur 3. Twee soorten tijd.

We merken het volgende op:

- de twee soorten tijd zijn onafhankelijk van elkaar en kunnen los van elkaar of in combinatie met elkaar worden gebruikt
- bij valid time is er sprake van verleden, heden en toekomst; bij transaction time daarentegen bestaat er geen toekomst

- het ontbreken van historie (figuur 3 linksboven) impliceert dat het feit altijd heeft bestaan, altijd zal blijven bestaan en nimmer is gewijzigd. Een mutatie vervangt deze fictie door een nieuwe.
- het ondersteunen van alleen valid time-historie (figuur 3 rechtsboven) maakt het mogelijk om het verloop van een feit te volgen. Het heden heeft geen speciale status in vergelijking met de toekomst of het verleden. Wijzigingen van heden, verleden en toekomst overschrijven echter eerder bestaande visies.
- het ondersteunen van alleen transaction time-historie (figuur 3 linksonder) staat bekend als mutatielogging. Wijzigingen in de toekomst en het verleden zijn niet toegestaan, waardoor er geen constraints aan het gegevensmodel hoeven worden toegevoegd. Vroegere constellaties van gegevens zijn in theorie te reconstrueren.
- het ondersteunen van tweedimensionale historie maakt het mogelijk om iedere constellatie van feiten te reconstrueren en voor wat betreft de valid time te wijzigen. Informatie gaat nooit verloren. Toekomst en verleden zijn evengoed toegankelijk en wijzigbaar als het heden.
- Ook bij tweedimensionale historie is er sprake van één werkelijkheid. Die werkelijkheid is weliswaar veranderlijk en niet altijd up-to-date vastgelegd, maar elke wijziging in het systeem wordt geacht direct bij de systeemgebruikers bekend te zijn. Achter deze fictie (zie figuur 2) ligt het versieprobleem dat in voorgaande hoofdstuk al werd genoemd.

Nog een keer het Kadaster

Ter afronding: welk gevolg heeft de uitbreiding met tweedimensionale tijd voor de Kadaster-database? Zie hieronder:

```
Object(object_id, tmin, dtmin, tmax, dtmax, ...)
Eigendom(eig_id, object_id, tmin, dtmin, tmax, dtmax)
Eigenaar(eig_id, tmin, dtmin, tmax, dtmax, ...)
```

Zoals te verwachten was krijgt elke historische tabel nu nog een tijddattribuut in de primaire sleutel. Alle historieproblemen verergeren nu dramatisch. Huiver mee:

- Object.tmin ≤ Object.tmax
- Eigendom.tmin ≤ Eigendom.tmax
- Eigenaar.tmin ≤ Eigenaar.tmax
- Object.dtmin ≤ Object.dtmax
- Eigendom.dtmin ≤ Eigendom.dtmax
- Eigenaar.dtmin ≤ Eigenaar.dtmax
- Voor elke set Object-tupels met eenzelfde object_id geldt dat de diverse tmin-tmax-intervallen geen overlap mogen vertonen (en mogelijk zelfs moeten aansluiten).

Een geschiedenis van ruimte en tijd

- Voor elke set Eigendom-tupels met dezelfde waarden voor eig_id en object_id geldt dat de diverse tmin-tmax-intervallen geen overlap mogen vertonen.
- Voor elke set Eigenaar-tupels met eenzelfde eig_id geldt dat de diverse tmin-tmax-intervallen geen overlap mogen vertonen (en mogelijk zelfs moeten aansluiten).
- Voor elke set Object-tupels met dezelfde waarden voor object_id en tmin geldt dat de diverse dtmin-dtmax-intervallen moeten aansluiten.
- Voor elke set Eigendom-tupels met dezelfde waarden voor eig_id, object_id en tmin geldt dat de diverse dtmin-dtmax-intervallen moeten aansluiten.
- Voor elke set Eigenaar-tupels met dezelfde waarden voor eig_id en tmin geldt dat de diverse dtmin-dtmax-intervallen moeten aansluiten.
- Voor elk tijdstip op het interval tmin-tmax voor een Eigendom-tupel dient er een geldig Object-tupel voor te komen.
- Voor elk tijdstip op het interval tmin-tmax voor een Eigendom-tupel dient er een geldig Eigenaar-tupel voor te komen.
- Voor elk tijdstip op het interval tmin-tmax, gecombineerd met elke waarde van dtmin voor een Eigendom-tupel dient er een geldig Object-tupel voor te komen.
- Voor elk tijdstip op het interval tmin-tmax gecombineerd met elke waarde van dtmin voor een Eigendom-tupel dient er een geldig Eigenaar-tupel voor te komen.

De beide laatste constraints zijn de uitdrukking van de TWK-nachtmerrie (TWK staat voor TerugWerkende Kracht). Ze vreten systeemresources en zijn niet in SQL uit te drukken. Integriteitsmanagement in een TWK-database is bij de huidige stand van theorie en techniek een vrijwel hopeloze zaak.

Merk ook op dat in een database als deze geen updates en deletes meer plaatsvinden. Elke mutatie is een insert. Merk ook op dat het verstandig is om de nieuwe tijdinformatie precies zo te modelleren als de eerder opgenomen informatie. Het Kadaster lijkt dit na te laten en mocht dit zo zijn dan is het inconsistent en onverstandig. Het opnemen van tmax is immers ook strikt genomen overbodig voor Object en Eigenaar (maar niet voor Eigendom!) wanneer Heintje Davids-constructies verboden zijn. De voordelen die opname van tmax biedt gelden onverkort voor dt_max. Overigens zegt het Kadaster-artikel niets over de keuze van primaire sleutels en het is niet geheel ondenkbaar dat het Kadaster terugschrikt voor opname van dt_min in de primaire sleutels. Veel automatiserders schrikken als het puntje bij paaltje komt terug voor het doorvoeren van tweedimensionale tijd. Geldt dit ook voor het Kadaster dan is de vraag "Wie was op 1 januari 1990 de eigenaar van perceel Loosdrecht/Sectie-C/5033 volgens de kennis die het Kadaster bezat op 15 februari 1990?" nimmer met zekerheid te beantwoorden.

Hoe nu verder?

Na dit tweede hoofdstuk hebben we zowel inzicht gekregen in de praktijk als in de theorie. Een oplossing voor het historieprobleem is echter nog niet in zicht. Eventuele wanhoop is begrijpelijk maar ontrecht en in elk geval voorbarig. In het volgende hoofdstuk zetten we de puntjes op de theorie en leggen we de grondslag voor een generieke oplossing van tweedimensionale historie.

Onverwerkt verleden, troebele toekomst

In dit hoofdstuk zullen we proberen om de stap te zetten van probleemschrijving naar probleemoplossing. In hoofdstuk 1 hebben we gezien dat historie een probleem is waarvoor bij de huidige stand van de techniek vele uitzichtloze en één of twee slechte oplossingen bestaan. In hoofdstuk 2 zagen we hoe er routinematig wordt gekozen voor foutieve stapel-achtige oplossingen en hebben we daarna geprobeerd om aannemelijk te maken dat het bewaren van historie in een informatiesysteem eigenlijk altijd gewenst is. We hebben ook gezien dat de onmogelijkheid een informatiesysteem synchroon te laten lopen met de werkelijkheid er voor zorgt dat een informatiesysteem met enig historisch bewustzijn twee tijdsdimensies moet kennen. Daarmee lijken we op een dieptepunt aangekomen in onze speurtocht naar een oplossing voor het historieprobleem.

Zoals al werd aangegeven in beide voorgaande hoofdstukken is ons historieprobleem het directe gevolg van het ontbreken van historie-ondersteuning in de metagegevensmodellen waarop dbms'en en ontwikkeltools zijn gebaseerd. In dit hoofdstuk zullen we proberen om de historieproblematiek vrij volledig en in alle ernst uit te werken binnen het in het eerste hoofdstuk geïntroduceerde 'juiste' modelleerkader. Wie dit hoofdstuk leest begrijpt waarom ik heb overwogen om het als titel 'Het Kadaster slaat terug' mee te geven.

Voor onze bespreking van het historieprobleem keren we terug naar ons oorspronkelijke medewerker-voorbeeld, dat we nog uitbreiden met afdelingsinformatie:

Medewerker (Med#, Naam, Sekse, GebDat, Adres, Woonplaats, Salaris,)

Afdeling (Afd#, Naam, Budget)

We baseren onze discussie op de volgende uitgangspunten:

- Not Null-constraints: alle attributen zijn verplichte primaire-sleutelconstraints: de waarden van Medewerker.Med# en Afdeling.Afd# moeten uniek zijn over de respectievelijke tabellen
- referentiële-integriteitconstraint: elke waarde van Medewerker.Afd# moet voorkomen in Afdeling.Afd
- user-defined constraint #1: elke afdeling heeft minstens één medewerker
- user-defined constraint #2: het budget van een afdeling (Afdeling.Budget) moet groter zijn dan de som van de salarissen (Medewerker.Salaris) van de aan die afdeling verbonden medewerkers.

De Historie-Transformatie Conventie

Het gegevensmodel hierboven bevat geen expliciete historie en bevindt zich zoals we in het vorige hoofdstuk hebben besproken in het eeuwige heden. Medewerkers, afdelingen en hun eigenschappen hebben voor het systeem altijd bestaan en zullen altijd blijven bestaan. Wijzigingen komen altijd neer op het vernietigen van informatie. Let wel, dit is een interpretatie van niet-historische data vertaald naar een historische context. Andere logische interpretaties dan deze zijn overigens moeilijk te bedenken, maar deze stilzwijgende afspraak zal later essentieel blijken bij het ontwikkelen van een generieke oplossing van het historieprobleem. Vandaar dat we deze regel, die we de Historie-Transformatie Conventie (HTC) zullen noemen, zorgvuldig moeten formuleren:

Regel 1:

Gegevens waaraan enige vorm van historie ontbreekt worden verondersteld onveranderlijk te zijn gedurende het volledige tijdsinterval waarop ze zouden kunnen hebben bestaan.

De HTC zegt dat indien de bovenstaande tabel Medewerker een tupel bevat voor een medewerker 'M1', dat alle gegevens van deze medewerker als onveranderlijk kunnen worden beschouwd vanaf het begin der tijd tot het einde der tijden. In het voorgaande hoofdstuk hebben we daarvoor de kosmologisch correcte termen 'Big Bang' en 'Big Crunch' geïntroduceerd. Praktischer is natuurlijk het vroegste en het laatste punt in de tijd dat door het dbms wordt ondersteund. Merk op dat de HTC ook van toepassing is op de tweedimensionale variant van het historieprobleem die in het vorige hoofdstuk is geïntroduceerd. Zouden de medewerker-gegevens wel historisch zijn opgeslagen, dan nog zou de HTC van nut zijn. Het systeem zou in dat geval namelijk wel een zinnig antwoord kunnen geven op de vraag "Wat verdiende medewerker 'M1' op 1/1/1997?", maar nog niet op de vraag "Wat verdiende medewerker 'M1' op 1/1/1997 volgens de kennis die het systeem bezat op 30/6/1997?". Voor een eendimensionaal historisch systeem zegt de

Onverwerkt verleden, troebele toekomst

HTC dat op deze vraag hetzelfde antwoord moet worden gegeven als op de vraag zonder terugwerkende kracht.

De HTC zorgt er voor dat een systeem naar eer en geweten antwoord geeft op historische vragen, ook wanneer de tijdsinformatie geheel of gedeeltelijk ontbreekt. Misschien vindt u de Historie-Transformatie Conventie onacceptabel, maar de consequentie van dat standpunt is wel dat een systeem alleen bruikbaar is wanneer alle er in opgenomen gegevens naar de twee historiedimensies worden vastgelegd. We hebben in het tweede hoofdstuk betoogd dat zelfs wanneer dat gebeurt er nog een informatiekort is, omdat het tijdstip waarop gebruikers kennis nemen van de informatie eigenlijk ook moet worden vastgelegd. Hoe het ook zij, voor de gebruikers van het merendeel van de informatiesystemen is de HTC een fact of life. Voor hen is zonder deze conventie vrijwel geen enkele vraag aan een informatiesysteem te beantwoorden. Voor ons automatiserders biedt de HTC bescherming tegen de totale krankzinnigheid en biedt het u de mogelijkheid om dit veeleisende boek te negeren. Hier stellen we vast dat gebruikers ten minste een latente behoefte hebben aan informatiesystemen met een geheugen: als (tweedimensionale) historie niets zou kosten dan zou iedereen het willen hebben. De waarde van een werkelijk generieke oplossing van het historieprobleem is nauwelijks in geld uit te drukken.

L'histoire de 'MI'

Laten we er van uit gaan dat assertieve gebruikers eisen dat van alle medewerker-gegevens in het informatiesysteem het verloop in de tijd kan worden vastgelegd, maar dat ze niet denken aan gegevensvastlegging met terugwerkende kracht. In de terminologie die in het vorig hoofdstuk werd geïntroduceerd is er dan sprake van toevoeging van valid time, maar niet van transaction time. De in het eerste hoofdstuk besproken minst slechte modelleerwijze levert het onderstaande, deprimerende resultaat op:

```
Medewerker (Med#)
Med_NaamHis (Med#, TVmin, TVmax, Naam)
Med_SeksHis (Med#, TVmin, TVmax, Seks)
Med_GebDatHis (Med#, TVmin, TVmax, GebDat)
Med_AdrWplHis (Med#, TVmin, TVmax, Adres, Woonplaats)
Med_SalHis (Med#, TVmin, TVmax, Salaris)
Med_AfdHis (Med#, TVmin, TVmax, Afd#)

Afdeling (Afd#, Naam, Budget)
```

Net als in hoofdstuk 1 zien we weer een explosie van historietabellen en krijgen we alsnog begrip voor de stapeloplossing die een groot deel van automatiserend Nederland – waaronder het Kadaster – kiest. Ik kan op dit punt

alleen vragen om geduld: het historie-medicijn moet in zijn geheel worden doorgeslikt voordat er zicht is op genezing van de patiënt. Gelieve mij verder bergafwaarts te volgen.

Kijk naar het model en constateer dat alle attributen zijn ondergebracht in historietabellen. Er is daarbij geen sprake van één tabel per historische rubriek. Attributen die gezamenlijk plegen te veranderen (zoals adres en woonplaats) zijn in één historie-tabel opgenomen. We spreken dan van een historische groep. Verhuist medewerker 'M1' naar een ander adres in dezelfde woonplaats dan vindt er zoals bij stapeelen of schaduwen een kopieeractie plaats, maar daar staat tegenover dat een verhuizing van Herenstraat 57 in Groningen naar Herenstraat 57 in Bussum direct herkenbaar is. Op basis van die redenering is het overigens ook denkbaar om naam en sekse in één historische groep op te nemen, maar dat terzijde. Merk op dat het denkbaar is om alle attributen van een tabel in één historische groep onder te brengen. We zitten dan dicht bij de in het eerste hoofdstuk besproken schaduwoplos-sing, waarbij de gehistoriseerde attributen echter redundant in de tabel Medewerker achterbleven. Kiezen we voor één historische groep en gooien we de bijna lege tabel Medewerker weg, dan komen we tot de stapeloplos-sing van het Kadaster. Zoals we hebben gezien wordt dan tegelijk met de tabel Medewerker ook het concept van referentiële integriteit vermoord en wordt het idee van primaire sleutels als representatie van objecten in de werkelijkheid verkracht. Welkom in het stenen tijdperk!

De HTC en de database-constraints

Misschien dacht u dat de Historie-Transformatie Conventie het leven van systeemgebruikers en systeemontwikkelaars vergemakkelijkt, maar dat is slechts waar voor het geval waarin een database geen historie bevat. In de situatie hierboven waarin alle Medewerker-gegevens zijn gehistoriseerd geeft de filosofie achter de HTC aanleiding tot nare consequenties. Voor een database met historie geldt namelijk onvermijdelijk een regel die het spiegelbeeld is van de HTC:

Regel 2:

Een constraint die geldt voor het heden dient ook te gelden voor elk ander tijdstip.

Uitgaande van de voor de valid time gehistoriseerde medewerker-tabel dienen de meeste regels te worden herschreven volgens de regels van de HTC:

- Not Null-constraints: alle attributen zijn verplicht voor elk tijdstip waarop een Medewerker bestaat
- referentiële-integriteitconstraint: elke waarde van Medewerker.Afd# moet voorkomen in Afdeling.Afd voor elk tijdstip waarop een Medewerker bestaat

Onverwerkt verleden, troebele toekomst

- user-defined constraint #1: elke afdeling heeft minstens één medewerker op elk moment in de tijd
- user-defined constraint #2: het budget van een afdeling (Afdeling.Budget) moet groter zijn dan de som van de salarissen (Medewerker.Salaris) van de aan die afdeling verbonden medewerkers voor elk moment waarop afdelingen en medewerkers bestaan.

Wie deze regels bekijkt is snel genezen van het idee dat de HTC het leven van een automatiseerder gemakkelijker maakt. Let maar eens op.

De nieuwe Not Null-constraints leiden tot de vraag: "Wanneer bestaat een medewerker?". Het enige antwoord op die vraag is: "Op elk moment dat valt binnen een TVmin-Tvmax-interval waarop voor een medewerker historische gegevens zijn opgeslagen!". Dit betekent dat op elk tijdstip waarop medewerker 'M1' een tupel in Med_NaamHis heeft er ook een 'M1'-tupel moet bestaan in de historie-tabellen Med_SekseHis, Med_GebDatHis, Med_AdresWplHis, Med_SalHis en Med_AfdHis. En uiteraard is die regel wederkerig voor alle andere medewerker-historietabellen dan Med_NaamHis. Deze regels moeten worden opgenomen als user-defined constraints en kunnen ook goed in SQL worden uitgedrukt, maar leuk is anders.

Over naar de referentiële-integriteitconstraint. Deze lijkt geen problemen op te leveren, maar schijn bedriegt. Omdat Afdeling niet historisch is wordt elke afdeling geacht te bestaan vanaf de Big Bang tot aan de Big Crunch en dus is er geen probleem. Helaas is het echter onmogelijk om enig Afdeling-tupel te verwijderen. Afdelingen die in de werkelijkheid niet meer bestaan blijven in het systeem aanwezig zolang de medewerker-historie bestaat. En dus is het ook mogelijk om medewerkers op die verwijderde afdelingen te plaatsen. Een veel toegepaste oplossing voor dat probleem is het toevoegen van een tweewaardig statusattribuut aan de tabel Afdeling dat aangeeft of de afdeling nog bestaat:

Afdeling(Afd#, Naam, Budget, Status)

Het statusattribuut is de meest primitieve vorm van historie die er bestaat, maar is een populair middel om de effecten van historie binnen de perken te houden. Het werken met een statusattribuut is echter strijdig met de HTC omdat de status zelf ook geacht wordt altijd de waarde te hebben gehad die hij op dit moment heeft. Wel acceptabel is de volgende oplossing:

Afdeling(Afd#, TVmin, TVmax, Naam, Budget)

Daar zijn onze oude bekenden weer terug, zij het dat TVmin hier geen deel uitmaakt van de primaire sleutel (en dus niet onderstreept is weergegeven). De referentiële-integriteitconstraint tussen Medewerker en Afdeling kan blijven bestaan, maar uiteraard moet er wel een user-defined constraint worden toegevoegd voor de tabel Med_AfdHis. Die constraint geeft aan dat de afdeling waarop een medewerker is geplaatst dient te bestaan gedurende het gehele interval waarop de medewerker op die afdeling werkt. Samen met de set Not Null-constraints ontstaat er nu een situatie waarin op elk moment in de tijd alle gegevens voor een medewerker en zijn/haar afdeling bestaan of alle medewerker-gegevens ontbreken en daarmee is aan regel 2 voldaan.

Over naar user-defined constraint nummer 1: "Elke afdeling heeft minstens één medewerker op elk moment in de tijd". Deze regel resulteert in een absolute ramp wanneer de tabel Afdeling niet historisch is. De HTC eist dan namelijk dat er voor elk moment in de tijd medewerkers bestaan. De oplossing voor dit probleem is uiteraard wederom het historiseren van de tabel Afdeling. Deze constraint is overigens het spiegelbeeld van een referentiële-integriteitconstraint en die gedachte leidt tot het formuleren van de volgende ontwerpregel:

Regel 3a:

Alle tabellen die een verwijzende sleutel hebben naar een historische tabel dienen minimaal dezelfde graad van historie te bezitten.

Deze regel zegt dat het onmogelijk is om de tabel Medewerker niet historisch te houden wanneer de tabel Afdeling historisch wordt gemaakt en dat elke tabel die verwijst naar de tabel Medewerker op haar beurt ook weer historisch moet zijn. De uitdrukking "dezelfde graad van historie" verwijst naar het bestaan van valid time en transaction time. Zouden we de tabel Afdeling terugwerkend historisch maken dan zou ook Medewerker terugwerkend historisch moeten worden gemaakt. En dit is nog maar de pragmatische variant van historiemodellering. Wie zich niet houdt aan de regels 2 en 3a heeft geen database maar een veredelde kaartenbak. Puur theoretisch is regel 3a zelfs nog te zwak en moet regel 3b gelden:

Regel 3b:

Alle tabellen in een database die door middel van constraints aan elkaar zijn gerelateerd dienen zich in dezelfde staat van historie te bevinden.

Ik ben regel 3b nog niet in de database-literatuur tegengekomen, maar dat zegt wellicht alleen iets over de omvang van de literatuur over temporal databases (die enorm is) en de belezenheid van deze auteur (die schiet mogelijk tekort). Mocht regel 3b werkelijk nog niet zijn beschreven dan neem ik direct de vrijheid om de stelling van Veldwijk bij u te introduceren in de vorm van regel 3c:

Onverwerkt verleden, troebele toekomst

Regel 3c:

Voor een database die niet voldoet aan regel 3b bestaat geen passende verzameling constraints die de geldende werkelijkheid represeneert. Een dergelijke database laat of het opnemen van onbestaanbare situaties toe of verbiedt het opnemen in de database van wel toegestane situaties.

Als regel 3c correct is dan is de enige beheersbare vorm van historie 'geen historie'. Ik zal hier niet proberen om de regel te bewijzen, maar sta open voor nadere toelichting en eventuele pogingen tot weerlegging. De discussie over het historisch maken van de tabel Afdeling moet voor dit moment volstaan. Voor wie een beter gevoel wil krijgen voor de juistheid van regel 3c is het een aardige vingeroefening om de casus uit te werken voor het geval waarin transaction time moet worden toegevoegd aan de tabel Med_SalHis teneinde het mogelijk te maken om een medewerker met terugwerkende kracht een ander salaris te geven:

```
Med_SalHis(Med#, TVmin, TTmin, TVmax, TTmax, Salaris)
```

User-defined constraint nummer 2: "Het budget van een afdeling (Afdeling.Budget) moet groter zijn dan de som van de salarissen (Medewerker.Salaris) van de aan die afdeling verbonden medewerkers voor elk moment waarop afdelingen en medewerkers bestaan" voegt geen nieuwe inzichten meer toe. Wel is deze constraint een prachtige kroongetuige voor de juistheid van regel 3c.

Met regel 3c lijkt het absolute dieptepunt in deze serie bereikt. De integriteit van een database met historie die op de hier beschreven wijze is gestructureerd en die voldoet aan regel 3a is in de praktijk goed te garanderen. Omgekeerd wil ik de stelling neerleggen dat de integriteit van een tweedimensionale historische database die is gestructureerd volgens andere principes zoals schaduwen of stapelen in de praktijk niet is af te dwingen, tenzij alleen mutaties voor het heden worden toegestaan. Uit die positie volgt dat de Kadaster-database van het voorgaande hoofdstuk door een kwaadwillende en bekwame gebruiker altijd valt te corrumpieren. Uiteraard geldt dit ook voor andere systemen waarin historie van belang is zoals het systeem waar mee uw salaris wordt uitbetaald.

Voordat we over gaan tot het introduceren van een generieke oplossing voor het historieprobleem dienen we nog een aantal kanten van het historieprobleem te bespreken die het probleem nog iets uitdagender maken.

The Return of ... Heintje Davids

Het doet me genoegen om voor de verandering eens een volledig onvertaalbare Nederlandse term te introduceren voor het weer tot leven komen van

dode database-objecten: Heintje Davids (HD) historie. Elke goede vorm van historische database dient in de ondersteuning van dit concept te voorzien.

Om HD-historie te introduceren zal ik mijzelf als voorbeeld nemen: uw database-columnist is bijna zeven jaar gelukkig geweest als medewerker van één van Neerlands grotere softwarehuizen. Wanneer het citaat van Barbra Streisand uit het voorgaande hoofdstuk klopt, is het uiteraard denkbaar dat ik nog eens bij dat bedrijf (Raet) terugkeer. Een beetje historische database dient het dan mogelijk te maken dat ik mijn oude personeelsnummer 'UTRVE' weer terugkrijg waardoor aan het personeelssysteem bekend is dat René Veldwijk weer terug is. Personeelsmedewerkers en managers houden van zoiets. Ons database-ontwerp voorziet wonderwel in deze eis, althans zolang de 'UTRVE'-historie nog niet fysiek uit het systeem is verwijderd. De tabel Medewerker bevat in dat geval immers nog steeds mijn oude personeelsnummer. Natuurlijk moet de personeelsmedewerker mijn oude code wel hergebruiken, maar de kans dat deze per ongeluk aan een nieuwe medewerker is uitgereikt is niet groot. Het hergebruiken van een primaire sleutel betekent immers dat een gebruiker willens en wetens een object uit het verleden weer activeert.

Een aardige manier om tegen HD-historie aan te kijken is het weer boven water halen van de hierboven verketterde objectstatus-benadering. Zoals we ook aan Afdeling deden zouden we aan de gestripte tabel Medewerker een soortgelijk attribuut kunnen toevoegen:

Medewerker (Med#, ... , Status)

Natuurlijk is deze oplossing even verwerpelijk bij Medewerker als hiervoor bij Afdeling en om dezelfde reden. Het idee van HD-historie implieert dat een status voortdurend kan wisselen tussen actueel of historisch, 'in-dienst' of 'uit-dienst' levend of dood. Wat ligt er meer voor de hand dan Medewerker.Status daadwerkelijk historisch te maken:

Medewerker (Med#)
Med_HD(Med#, TVmin, TVmax, Status)

We kunnen nu afspreken dat een medewerker voor elk moment in de tijd, van de Big Bang tot de Big Crunch, een status actief/inactief heeft. Daarmee wordt het mogelijk om elke Not Null-constraint (zie hierboven) aan de informatie in Med-HD op te hangen en wordt ook de problematiek van referentiële integriteit en user-defined constraint nummer 1 vereenvoudigd. Het is natuurlijk ook mogelijk om MED-HD. Status weer uit de tabel Med_HD te verwijderen en af te spreken dat een medewerker alleen bestaat op tijdstippen die vallen tussen TVmin en TVmax.

Onverwerkt verleden, troebele toekomst

Constraints unlimited

Het idee van de Historie-Transformatie Conventie zoals uitgedrukt in regel 2 houdt in dat het mogelijk moet zijn om de integriteit van een database te controleren voor elk moment in de tijd. In een database zonder historie is het mogelijk om het overgrote deel van de user-defined constraints uit te drukken in SQL. In een conform regel 3a of 3b gehistoriseerde database is dit als regel niet mogelijk voor constraints die betrekking hebben op meerdere tupels, al dan niet in één tabel. Voor een domeinconstraint (Medewerker.Sekse = 'M' of 'V') of een referentiële-integriteitconstraint is een historisch SQL-equivalent nog wel mogelijk. In ons voorbeeld geldt dit ook nog voor user-defined constraint nummer 1, die al was neergezet als een omgekeerde referentiële-integriteitconstraint. User-defined constraint nummer 2, "Het budget van een afdeling (Afdeling.Budget) moet groter zijn dan de som van de salarissen (Medewerker.Salaris) van de aan die afdeling verbonden medewerkers", is echter minder meegaand. Zonder historie kan deze regel als volgt worden uitgedrukt:

```
SELECT 'Te laag budget voor afdeling', Afd#
FROM Afdeling a
WHERE a.Budget <
      (SELECT SUM(m.Salaris)
       FROM Medewerker m
       WHERE m.Afd# = a.Afd#
      )
```

Probeer deze maar eens uit te drukken voor een gehistoriseerde database. Controle voor één punt op de tijdlijn (valid time historie) of het tijdvlek (valid time en transaction time historie) is wel mogelijk, maar een volledige controle is niet in een SQL-statement uit te drukken. Erger: de enige procedure die in het algemeen garandeert dat de inhoud van een database voor elk tijdstip aan een bepaalde complexe constraint voldoet heeft een weerzinwekkend brute force karakter:

Regel 4:

Een constraint die in een gehistoriseerde situatie niet in een SQL-statement is uit te drukken kan alleen sluitend worden gecontroleerd voor de gehele database door een controle uit te voeren voor elk moment waarop een wijziging van relevante gegevens heeft plaatsgevonden.

Ik weet het, regel 4 is cryptisch en de ruimte voor een diepgaande toelichting ontbreekt. Neem als voorbeeld onze budget-constraint. In een situatie zonder historie kan deze worden geschonden bij de volgende database-mutaties:

- toevoegen van een Medewerker-tupel
- wijzigen van Medewerker.Salaris

- wijzigen van Medewerker.Afd#
- wijzigen van Afdeling.budget.

In dit voorbeeld dient de constraint te worden gecontroleerd voor elk tijdstip waarop een van deze mutaties is doorgevoerd. Wanneer de constraint-handler die het gehistoriseerde SQL-statement uitvoert geen begrip heeft voor het feit dat de te controleren verzameling tupels kan worden beperkt tot de medewerkers van één afdeling, dan zal bovendien voor elke mutatie de complete database moeten worden gecontroleerd. Ik ben er van overtuigd dat het onmogelijk is om een voldoende slimme handler te schrijven en dus is het bij grote historische databases nauwelijks mogelijk om op een generieke wijze en met voldoende performance de integriteit van de gegevens door de tijd heen te controleren.

Merk op dat de domein-constraint "Medewerker.Salaris > 0" en de referentiële-integriteitconstraint tussen Medewerker en Afdeling er voor zorgen dat een verwijdering van een Medewerker-tupel respectievelijk een Afdeling-tupel niet tot schending van de budget-constraint kan leiden. De transformatie van de referentiële-integriteitconstraint naar een historische variant zorgt er voor dat dit ook zo blijft in een gehistoriseerde database. Leve de Historie-Transformatie Conventie.

Tijd voor sleutels: surrogaatsleutels

Bij het historiseren van de tabel Medewerker werd aangegeven dat hypothetische assertieve gebruikers eisen dat van alle medewerker-gegevens in het informatiesysteem het verloop in de tijd kan worden vastgelegd. Vermoedelijk hebben de meeste lezers niet gemerkt dat we niet aan deze voorwaarde hebben voldaan. Kijk nog eens naar de gehistoriseerde database: het is niet mogelijk om het historisch verloop van Medewerker.Med# vast te leggen!

Veel automatiserders stellen zich op het standpunt dat een object (hier: een medewerker) in de realiteit wordt gerepresenteerd door de waarde(n) van zijn primaire sleutel (hier: de waarde van Med#). Verandert deze waarde dan is er sprake van een nieuw object. Dit argument werd (wordt?) vaak gebruikt om het muteren van primaire sleutels te verbieden en bespaart veel programmeerwerk in rdbms-omgevingen die geen cascaded update ondersteunen. Ik vermoed dat het tot voor kort bewerkelijke karakter van het doorvoeren van mutaties op primaire sleutels debet is aan deze merkwaardige, zo niet onzinnige, opvatting. De primaire sleutel vormt de verbinding tussen een object in de werkelijkheid en de representatie van dat object in de database. Gebruikers hebben een onbedwingbare neiging om die verbinding te baseren op een herkenbare eigenschap van het te administreren object. Wij automatiserders houden deze praktijk niet tegen en in veel gevallen is het gebruiksgemak van een betekenisvolle primaire sleutel niet te betwisten. Neem wederom mijn oude Raet-personeelsnummer 'UTRVE'. Voor mijn col-

Onverwerkt verleden, troebele toekomst

lega's was het duidelijk dat het ging om een medewerker van regio Utrecht, dat zijn voornaam begon met een 'R' en de eerste twee letters van zijn achternaam 'VE' waren. Doordat eenmaal uitgegeven sleutels niet wijzigden gaf (geeft?) dit een accuraat beeld van de organisatie en de naam van de medewerker ten tijde van het begin van het dienstverband. De afdeling Utrecht is sindsdien overgegaan in de Regio Midden Nederland en daarna min of meer in de Business Unit Cross Platform Services. Mijn naam is nooit veranderd maar mijn ex-collega Marja N. kon na haar scheiding van meneer N. maar met grote moeite haar code 'UTMNE' veranderd krijgen in 'UTMPO'.

Met de ondersteuning van cascaded updates in moderne rdbms'en staan we tegenwoordig natuurlijk toe dat primaire sleutels worden gewijzigd. Om de verbinding met de werkelijkheid te behouden is het dan uiterst wenselijk om ook Medewerker.Med# te historiseren. Dit leidt tot de volgende databasestructuur voor de medewerker-gegevens:

```
Medewerker(X)
Med_CodeHis(X, TVmin, TTmax, Tvmax, Med#)
Med_NaamHis(X, TVmin, TTmax, TVmax, Naam)
Med_SekseHis(X, TVmin, TTmax, TVmax, Sekse)
Med_GebDatHis(X, TVmin, TTmax, TVmax, GebDat)
Med_AdresWplHis(X, TVmin, TTmax, TVmax, Adres, Woonplaats)
Med_SalHis(X, TVmin, TTmax, TVmax, Salaris)
Med_AfdHis(X, TVmin, TTmax, TVmax, Afd#)
```

Ziehier de ultieme historische database. Attribuut X is een zogenaamd 'surrogaatattribuut' ofwel een 'surrogate', een soort object-id zoals dat ook wel wordt gepropageerd door de objectgeoriënteerden. De primaire-sleutelconstraint moet er nu ook aan geloven:

Primaire-sleutelconstraint: de waarden van Med_CodeHis.Med# moeten uniek zijn voor elk moment in de tijd.

Merk op dat ook nu nog de tabel Medewerker blijft bestaan als houdster van betekenisloze referenties naar alle medewerker-objecten die hebben bestaan, bestaan of zullen bestaan en tevens als doel voor sleutelverwijzingen. We blijven ons best doen om het relationele model niet definitief overboord te gooien.

A medewerker is born: user-defined time

Ook een database zonder historie kan attributen herbergen die een historische geur hebben. Zo is er in ons voorbeeld de rubriek Medewerker.GebDat die vanzelfsprekend geboortedatum van de medewerker bevat. Het ligt voor de hand om een constraint op te nemen die aangeeft dat een Medewerker-tupel alleen mag bestaan voor data die vallen na de geboorte van de medewerker. Gesproken wordt dan wel van user-defined time. Dit concept levert

als zodanig geen problemen op: toevoeging van tijd maakt het mogelijk om regels uit te drukken in constraints die in een database zonder historie door de gebruiker zelf moeten worden afgedwongen. Merk ook op dat de code 'UTRVE' nu kan worden gebruikt door een andere Raet-medewerker zonder dat dit voor het personeelssysteem betekent dat René Veldwijk en de medewerker één en dezelfde persoon zijn.

Tijd in korrelvorm: granulatie

Een filosofisch zeer interessante vraag is of het verstrijken van tijd een continu proces is of dat tijd net als materie een atomair karakter heeft. Volgens een natuurkundige theorie heeft de tijd inderdaad een korrelvormig karakter en duurt een tijd-atoom of *Chronon* circa 10^{-34} seconde. Voor ons is uiteraard de maximale nauwkeurigheid van ons dbms of besturingssysteem van belang en uiteraard is de tijdondersteuning in elk product minder nauwkeurig bekend. Voor logging (transaction time) is het altijd de bedoeling dat de tijd zo nauwkeurig mogelijk wordt geregistreerd, maar voor de gewone of geldige tijd is een minder verfijnde nauwkeurigheid ofwel 'granulatie' vaak voldoende of zelfs wenselijk. Kiezen we een granulatie van een seconde voor Med_AdrWplHis dan leidt elke doorgevoerde typefout tot toevoeging van een tupel. We kunnen ons op het standpunt stellen dat deze mutatie kan hebben geleid tot verkeerde informatieverstrekking en dus moet worden geregistreerd, maar dat is niet altijd een redelijk standpunt. Praktischer is het vaak om mutaties binnen één bepaald tijdsinterval elkaar te laten overschrijven. Zo is het goed denkbaar om de adresgegevens met de nauwkeurigheid van een dag vast te leggen. Verhuist iemand twee keer op een dag dan wordt alleen het laatste adres vastgehouden. Een nog beter voorbeeld wordt gevormd door Afdeling.Budget dat zeer wel per jaar kan gelden en ook slechts eenmaal per jaar hoeft te worden vastgelegd:

Afd_BudHis (Afd#, TVmin, TVmax, Budget)

Is het budget een jaarbudget dan is het goed te verdedigen om de nauwkeurigheid van TVmin tot één jaar te beperken, bijvoorbeeld door het attribuut als integer te declareren indien uw rdbms geen expliciete ondersteuning voor granulatie biedt. Wellicht willen we dat de mutaties van Afdeling.Budget worden gelogd, maar dan is het te overwegen om daarvoor een separate tabel aan te leggen:

Afd_BudLogHis (Afd#, TTmin, TTmax, Budget)

We zien dat hier de transaction time wordt gebruikt en we hebben al gezien dat er daarbij altijd voor de maximale nauwkeurigheid moet worden geko-

Onverwerkt verleden, troebele toekomst

zen.

Enkele regels met betrekking tot granulatie:

Regel 5:

Granulaties die in een historische database worden toegepast mogen elkaar niet overlappen.

Deze regel zegt dat het is toegestaan om de granulaties 'jaar', 'maand' en 'dag' in een database te gebruiken, maar dat het gebruik van de combinatie 'Week' en 'Maand' of 'Jaar' is verboden. Evenmin is het mogelijk om kalendersystemen te combineren in een database. Een database waarin gegevens per christelijk jaar of maand worden vastgelegd is bijvoorbeeld niet ramadan-compatibel.

Regel 6:

Een historische tabel die een verwijzende sleutel naar een andere tabel heeft dient een minstens even fijne granulatie te hebben als deze tabel.

Regel 6 zegt dat het niet zinvol is om afdelingen per maand en medewerkers per jaar te registreren. Het verdwijnen van een afdeling is dan immers alleen mogelijk bij de jaarovertrek. Het logisch verwijderen van een afdeling op 30 november leidt dan immers tot een schending van de referentiële integriteit voor de maand december. Voor de historische groepen van een tabel geldt een soortgelijke redenering.

Forget operator

In de discussie omtrent The Return of ... Heintje Davids werd tussen de regels door nog een ander historieprobleem geïntroduceerd, te weten het fysiek uit het systeem verwijderen van gegevens. In een systeem met valid time-ondersteuning vinden geen fysieke deletes meer plaats. Het verwijderen van een medewerker leidt tot het updaten van TVmax met een tijdsindicatie tot wanneer dit gegeven geldig was. Voor het fysiek verwijderen van gegevens uit de database moet een nieuwe operator in het leven worden geroepen: de forget operator. Uiteraard is het van belang dat die operator alleen door geautoriseerde gebruikers kan worden gebruikt.

Historische bloopers: de problematiek van het doormuteren

Stel dat de medewerkers in ons voorbeeldsysteem vandaag met terugwerkende kracht tot 1 januari 1997 enkele procenten salarisverhoging krijgen. Het is nu alleszins denkbaar dat voor sommige afdelingen het totaal aan salarissen het afdelingsbudget overstijgt en dat user-defined constraint nummer 2 nu wordt geschonden. Dit probleem is allermildest theoretisch: een collega heeft in deze problematiek flink zijn tanden moeten zetten bij zijn werk-



zaamheden voor een grote verzekeraar en het komt vooral voor bij systemen die terugwerkende kracht historie ondersteunen. Over dit probleem van 'doormuteren' kan ik voor de verandering eens kort en krachtig zijn: dit probleem is een probleem van de gebruiker, niet van ons automatiserders. Als de regel geldt dat het afdelingsbudget voldoende moet zijn om de salarissen te betalen dan zal ook het afdelingsbudget moeten worden aangepast of de salarisverhoging gaat niet door. Natuurlijk levert dit vaak complexe en veelomvattende transacties op, maar daarvoor leveren moderne dbms'en en tools wel een oplossing.

Your worst nightmare: metahistorie

Voor wie denkt alles al te hebben gehad heb ik nog één uitsmijter: wat gebeurt er wanneer een constraint op een historische database wordt toegevoegd, gemodificeerd of gewijzigd? Iedereen die met historische databases werkt loopt vroeg of laat tegen dit probleem aan en voor de ongelukkigen die belast zijn met het in de lucht houden van salaris-, uitkerings- of pensioensystemen is dit aan de orde van de dag. Zo ziet ons parlement er geen been in om bijvoorbeeld uitkeringsregels met terugwerkende kracht te veranderen. Stel je voor dat er tegelijkertijd ook met terugwerkende kracht gegevens worden gewijzigd, bijvoorbeeld wanneer een uitkeringsgerechtigde vandaag meldt dat zij vanaf 1 juni 1997 samenwoont. Het tegelijk terugwerkend wijzigen van zowel berekenings- en/of integriteitsregels als gegevens waarop die regels betrekking hebben is de ergst denkbare situatie. Een dergelijk systeem kan alleen opereren wanneer het is geïntegreerd met een eveneens historische metadatabase en zelfs dan bestaan er problemen die theoretisch onoplosbaar zijn.

L'histoire se répète: hoop doet leven!

Het doel van dit hoofdstuk was het systematisch en nagenoeg volledig bespreken van de vreselijke problemen die historische databases met zich meebrengen. Kent een database op enige schaal historie, dan valt het te hopen dat de systeemgebruikers gedisciplineerd en voorzichtig omgaan met het doorvoeren van mutaties omdat ze nagenoeg altijd werken met een veredelde kaartenbak. Dbms- en tool-leveranciers bieden geen oplossingen voor de problematiek van historische databases. Erger, ze hebben veelal geen flauw benul van de problemen en dus is een oplossing van die zijde ook niet in zicht. De keuze gaat daarom tussen:

- het niet historiseren van databases en het eventueel opzetten van data-warehouses voor speciale historie-eisen. In zeer veel gevallen is dit een acceptabele, zij het kostbare mogelijkheid.
- het accepteren van een beperkte vorm van integriteitsmanagement (kaartenbak-plus systemen) en/of het verbieden van mutaties voor andere tijdstippen dan het heden.

Onverwerkt verleden, troebele toekomst

- het bedenken van een generieke oplossing van het historieprobleem op basis van de constatering dat het historieprobleem zich steeds in dezelfde vorm aan ons voordoet.

De volgende hoofdstukken zullen gewijd zijn aan de laatstgenoemde benadering.

L'histoire ne se répète plus

In dit hoofdstuk zullen we proberen om een oplossing te geven voor de historieproblematiek zoals we die hiervoor hebben besproken. Die oplossing heeft een generiek karakter en is bij de huidige stand van de rdbms- en tooltechnologie te implementeren. Dit hoofdstuk en het volgende beschrijven de eisen waaraan een generieke oplossing moet voldoen, hoe de oplossing van ondergetekende er uit ziet, welke hiervoor besproken historie-aspecten nog niet zijn geïmplementeerd en welke aspecten niet worden ondersteund omdat ze voor altijd onoplosbaar zijn.

Het lezen van de eerste drie hoofdstukken laat zich misschien nog het best vergelijken met een verblijf in het Russische ruimtestation Mir. Steeds wanneer je de ellende denkt te hebben verwerkt, staat er een nieuwe onaangezane verrassing te wachten. Toch is met het bespreken van alle historieellende de basis voor een oplossing gelegd. Om te beginnen levert de besprekking van de historieproblematiek zoals we die hebben gegeven een referentiekader, waaraan we oplossingen voor historie kunnen toetsen. Die redenering kan ook worden omgedraaid: wie met een generieke oplossing voor de historieproblematiek wil komen dient zijn oplossing vooraf te toetsen aan de problemen die bij conventionele implementaties spelen. Daarnaast zullen we zien dat in de besprekking van het probleem de oplossing al besloten ligt. De oplossing die we in dit hoofdstuk zullen uitwerken is niettemin geen eenvoudige. Om gênante situaties uit te sluiten is met de publicatie gewacht totdat implementatie in werkende software een feit was. Nu dat het geval is, is de verleiding groot om te beginnen bij de implementatie en u een 'en toen ... en toen ...'-verhaal voor te schotelen. Uiteraard zullen we aan die verleiding weerstand bieden en de theorie zoals altijd een leidende rol geven.

Laten we daarom beginnen met het formuleren van twee basale vragen:

- Wat bedoelen we met een de term 'generieke oplossing' voor de historieproblematiek?

- Is het historieprobleem überhaupt oplosbaar en, zo ja, onder welke condities?

Wat zijn generieke oplossingen?

Beginnen we met de eerste vraag dan zien we dat er bij historie altijd sprake is van een tweetal verwante problemen:

- het bevragen van een historische database is vaak uiterst complex
- bij het muteren van een historische database is het problematisch om de integriteit van gegevens in een historische database te waarborgen. We hebben gezien dat de keuze gaat tussen het laten vallen van elementaire relationele integriteitsconcepten of het accepteren van een explosie van tabellen, sleutels en user-defined constraints.

Beide problemen komen er op neer dat programmeren op een database met historie een ellendige zaak is in vergelijking met het programmeren op een gelijksortige database zonder historie. Een oplossing voor het historieprobleem moet deze twee zaken uiteraard wegnemen. De programmeur noch de gebruiker mogen worden lastig gevallen met historische update- en raadpleegproblemen. In het eerste hoofdstuk heb ik gesteld dat de 'time-series' oplossingen die worden aangeboden door rdbms-leveranciers geen soelaas bieden bij complexe administratieve toepassingen. Nu kan worden uitgelegd waarom dat zo is. De voorgestelde 'oplossingen' gaan voorbij aan het probleem van integriteitsmanagement (probleem 2) in een historische database, terwijl integriteitsmanagement in elke administratieve omgeving de alfa en omega vormt. De voorstellen die in de SQL-wereld worden gedaan lijden aan precies dezelfde kwaal: gedrag is koning, integriteitsmanagement lijkt iets voor ouden van dagen. De generieke oplossing die in dit hoofdstuk wordt uitgewerkt gaat uit van integriteitsbehoud en leidt daaruit adequaat gedrag af.

Het begrip 'oplossing' wordt steeds voorafgegaan door het bijvoeglijk naamwoord 'generiek'. Met die term wordt gedoeld op het streven om met een oplossing te komen die even bruikbaar is voor het Kadaster-systeem als voor een salarissysteem, een logistiek systeem of een systeem voor een woningbouwcorporatie. We willen geen serie deeloplossingen voor deelproblemen, geen historie-cartridges, maar een werkelijk universele oplossing. Als zo'n oplossing onvolledig is en er voor een specifiek systeem meer nodig is, dan dient die universele oplossing niettemin integraal bruikbaar te blijven. Voor minder doen we het niet. Een implementatie die daarin slaagt krijgt al snel het label 'tool' opgeplakt en dat is helemaal terecht. Een generieke oplossing voor het historieprobleem vereist uitbreidingen op het niveau van het rdbms en — zoals we in het volgende hoofdstuk zullen zien — op het niveau van frontend-tools. En eigenlijk is er bij een generieke oplossing sprake van een uitbreiding op het relationele model waarop de rdbms'en zijn gebaseerd. Zo'n uitbreiding mag geen enkele beperking opleggen aan de bestaande features van het relationele model en evenmin aan de functionaliteit van

L'histoire ne se répète plus

relationele dbms'en waarin dit model is uitgewerkt. Al met al moeten we gelijktijdig op de stoelen van metamodelontwerper, rdbms-leverancier en toolleverancier gaan zitten: voorwaar geen geringe opgave. Daarbij hebben we wel het geluk dat noch het relationele model noch de rdbms'en van vandaag iets te melden hebben over historie. We hoeven alleen maar iets toe te voegen en niets af te breken. Daarnaast zijn de mogelijkheden om generieke uitbreidingen aan rdbms'en toe te voegen de laatste jaren sterk toegenomen door triggers, stored procedures en (recent) objectrelationele extensies. Bij sommige moderne frontend-tools is de situatie nog rooskleuriger: met name in objectgeoriënteerde frontend-tools is het mogelijk om objectklassen op toolniveau te instantiëren en zelf algemene functionaliteit aan het frontend-tool toe te voegen. Minstens even fijn is de mogelijkheid om functionele objecten — met name visuele objecten — dynamisch te creëren, te modifiveren en te vernietigen.

Contouren van een generieke oplossing

Een werkelijk generieke oplossing kenmerkt zich door het volledig uit het zicht verdwijnen van de problemen waarvoor die oplossing is bedacht. Een ondoorzichtige maar populaire term voor dit verschijnsel is transparantie. Bij het ontbreken van een generieke oplossing is de oplossing voor een probleem terug te vinden in de applicatie-code en dat kost tijd en geld, telkens weer. Er zijn vele voorbeelden van de Siamese tweeling genericiteit en transparantie te geven, maar het voorbeeld dat we hier zullen gebruiken is het begrip index. Voor de introductie van rdbms-technologie bepaalde de programmeur het toegangspad dat werd gebruikt. Ik herinner mij met gepaste afkeer het statement "Use Index ..." in mijn Clipper-applicaties en het gevoel van verrukking toen het in Oracle versie 4 onmogelijk was in een programma aan een index te refereren. Oracle — en elk ander serieus rdbms — bepaalt zelf wel welke index wordt gebruikt. Het generieke mechanisme dat deze taak verricht luistert naar de naam optimizer. Bij een goede generieke oplossing voor een probleem is er altijd sprake van:

- minder programmeerwerk en meer parametriserwerk: voor elke vijf programmeurs die verdwijnen komt er een DBA terug
- meer flexibiliteit: parameters zijn aanpasbaar zonder ingrijpen in de programmatuur (Oracle overleeft een Drop Index, Clipper niet)
- voorspelbaar gedrag: een vraag na de creatie van een index levert hetzelfde resultaat op als daarvoor, hooguit komt het antwoord sneller
- betere performance doordat het generieke mechanisme meer kennis heeft dan een programmeur: een goede rdbms-optimizer negeert het bestaan van een index als een vraag op die index onvoldoende selectief is.

Van een generieke oplossing voor het historieprobleem verwachten we precies hetzelfde. De generieke oplossing voor historie die hier wordt geïntroduceerd:

- verbergt de historische toestand van de database voor de programmeur en verzwaart de taak van de (meta-)DBA
- maakt het mogelijk dat de historische toestand van de database — en daarmee de werking van de programmatuur — wordt gewijzigd door een (meta-)DBA zonder dat programma-aanpassingen nodig zijn
- levert voor elke vraag een voorspelbaar antwoord, ongeacht (wijzigingen in) de historische toestand van de database
- kan qua performance concurreren met een conventionele oplossing.

Eerherstel voor SQL-3

Laten we kijken of een dergelijke compromisloze generieke oplossing haalbaar is en wat het met zich meebrengt. Om te beginnen volgt uit item 1 (de historische toestand van de database is verborgen voor de programmeur) dat de programmeur altijd een database zonder historie ziet, zoals een rdbms-programmeur geen weet hoeft te hebben van het bestaan van indexen. We grijpen terug op het medewerker/afdeling-voorbeeld:

```

*  

Medewerker(Med#, Naam, Sekse, GebDat, Adres, Woonplaats, Salaris,  

Afd#)  
  

Afdeling(Afd#, Naam, Budget)

```

Deze database bevat geen historie en diende als uitgangspunt voor het vele geploeter met geldige tijd, transactietijd (logging) en de combinatie van die soorten tijd in ondersteuning voor terugwerkende kracht. Alle ellende die we hebben besproken komt voort uit het steeds wijzigen van de gegevensstructuur, de integriteitregels die op die structuur zijn gedefinieerd en de programma's die op die structuur werken. Al deze problemen lossen we in een klap op door alleen te willen kijken naar de prehistorie. Historie is dan transparant geworden en programmeren op historie behoort tot het verleden! Misschien willen we Medewerker.Naam, Adres en Woonplaats en Afd# historisch vastleggen. Misschien willen we Medewerker.Salaris terugwerkend historisch registreren. Misschien willen we mutaties op Afdeling.Budget loggen. Misschien wil de gebruiker morgen weer iets heel anders, het boeit allemaal niet meer. Aan de gegevensstructuur is niets te zien; precies zoals we aan de definities van Medewerker en Afdeling niet kunnen zien welke indexen op welke attributen zijn gelegd. Zoals het benutten en bewaken van indexen een zaak van het rdbms is, zo is het vertalen van een vraag op de niet-historische gegevensstructuren naar een gelijkwaardige historische vraag een zaak van ons generieke historiemechanisme. Vervolgens komt natuurlijk wel de vraag op hoe we kunnen vragen naar de historische status van gegevens als die status nergens is te zien. De oplossing voor dit probleem is tweeledig:

L'histoire ne se répète plus

- de onveranderlijke niet-historische ('logische') gegevensstructuur dient te worden losgekoppeld van de in het rdbms geïmplementeerde ('fysieke') structuur
- de data access taal (lees: SQL) dient te worden uitgebreid met taalconstructies waarmee de twee tijdsdimensies kunnen worden uitgedrukt.

De historische extensies op SQL die door Stephen Cannan zijn beschreven (zie pagina 97 t/m 119) bevatten inderdaad dergelijke extensies en ik heb geen suggesties ter verbetering. Een statement op de database dat vraagt naar alle medewerkers die werkzaam waren op afdeling A1 en in 1993 meer verdiensten dan een ton zou er als volgt kunnen uitzien:

```
VALIDTIME PERIOD '[DATE '1993-01-01' ~ DATE '1993-12-31']'
SELECT Med#, Naam
FROM   Medewerker
WHERE  Afd# = 'A1'
      AND Salaris > 100.000
ORDER BY Med#
```

Onzichtbare structuren en integriteitsregels

Extensies als deze op SQL zijn beslist noodzakelijk, maar zoals gezegd is het onderwerp 'integriteitsmanagement' in onze visie koning. Extensies op SQL komen pas veel later. We beginnen daarom met een blik op de gegevensstructuren en de integriteitsregels. We gaan ervan uit dat de meta-DBA datgene wil doen wat we hierboven reeds als mogelijkheid schetsten:

- Medewerker.Naam dient historisch (valid time) te worden vastgelegd met een granulatie (fijnkorreligheid) van een jaar
- Medewerker.Adres, Woonplaats en Afd# dienen eveneens historisch (valid time) te worden vastgelegd met de fijner granulatie van een maand (sommige mensen verhuizen vaak)
- opdat we de effecten van salariswijzigingen met terugwerkende kracht willen kunnen afhandelen, leggen we Medewerker. Salaris terugwerkend (valid en transaction time) vast. De granulatie van de valid time component is één dag
- alle mutaties op Afdeling.Budget dienen te worden gelogd (transaction time).

In eerste aanleg dienen de niet-historische tabellen met hun rubrieken en sleutels te worden aangemaakt in de rdbms-omgeving. Gebruik van standaard tools à la SDW of Oracle Designer/2000 biedt geen soelaas omdat deze producten niet uitbreidbaar zijn, wat hier concreet betekent dat ze niet historisch bewust kunnen worden gemaakt. Het is daarom nodig om een eigen dictionary-hulpmiddel te ontwikkelen dat wel historisch bewust is. Figuur 4 toont een deel van een invoerscherm waarmee de niet-historische (logische)

4.2.2. Invoer gegevensstructuren en sleutels

gegevensstructuren en sleutels worden ingebracht. Het scherm toont een deel van de dictionary-data waarvan uit de tabellen en sleutelconstraints worden gegenereerd. De meeste rubrieken spreken voor zich of zijn niet van belang. Uitzonderingen zijn de rubrieken Gen? in de beide gegevensblokken en de rubriek AM? in het attributenblok. De Gen?-rubrieken geven aan of een tabel of rubriek al dan niet is gegenereerd door een generiek mechanisme zoals het historiemechanisme. Uiteraard is daarvan op dit moment nog geen sprake. De rubriek AM? komt min of meer overeen met het huidige null-begrip. Merk op dat alleen Medewerker.Gebdat null mag zijn.

Figuur 4. Dictionary-scherm voor invoer gegevensstructuren, sleutels, enzovoort.

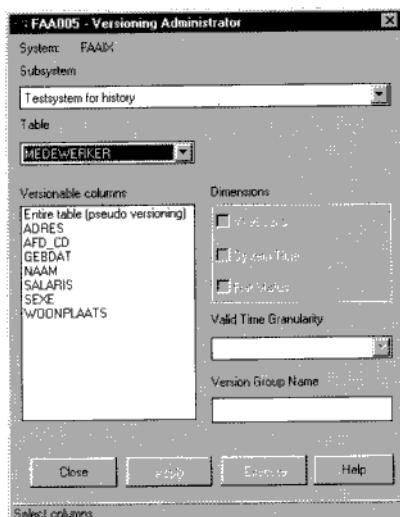
Na inbrengen van de tabellen, attributen, relationele sleutels (tabblad Keys), domeinen (tabblad Domains) en mogelijk nog enkele user-defined constraints (niet getoond), kunnen de tabellen Afdeling en Medewerker en hun sleutels worden gegenereerd op rdbms-niveau. We kunnen nu programmatuur ontwikkelen op de tabellen Medewerker en Afdeling en deze later historisch maken, precies zoals we ook na implementatie van een systeem nog een index kunnen toevoegen. We kunnen natuurlijk ook de historische constellatie direct aangeven. Nu is het tijd voor meta-DBA-werkzaamheden die neerkomen op het aanduiden van de gewenste vorm van historie. Dit gebeurt met behulp van het versioning administrator-scherm (figuur 5).

Met de versioning administrator is het mogelijk om rubrieken of groepen van rubrieken te (ont)historiseren. Daarnaast is het mogelijk om flattering, een eenvoudige vorm van versie-ondersteuning, op mik-en-klik-basis te implementeren.

In het eerste hoofdstuk stelden we dat het historieprobleem een eenvoudige (...) variant is van een algemener versieprobleem. Met de aanpak van dit versieprobleem is zo te zien ook een begin gemaakt, maar hier zullen we dat

L'histoire ne se répète plus

nog negeren. We zien dat het mogelijk is om een tabel te selecteren en dat van deze tabel alle rubrieken worden getoond met uitzondering van de primaire sleutel Med#. Daarmee zijn we bij de eerste beperking van ons historiemechanisme aangekomen. In het vorige hoofdstuk hebben we gezien dat het mogelijk is om een primaire sleutel te historiseren en dat die mogelijkheid ook uiterst gewenst is. We hebben echter ook gezien dat de impact van zo'n wijziging op de structuur van de database en op de user-defined constraints helaas zeer groot is en daarom zijn we dit probleem vooralsnog uit de weg gegaan. Alternatieve sleutels, verwijzende sleutels en natuurlijk gewone attributen mogen naar hartelust worden gehistoriseerd, maar primaire sleutels niet. In een volgend leven hoop ik dat probleem ook nog eens op te lossen.



Figuur 5. De versioning administrator.

Behalve de niet-primaire sleutelrubrieken ondersteunt de versioning administrator ook het historiseren van een gehele tabel. We spreken dan van pseudohistorie of, algemener, van pseudo-versioning. We kunnen daarmee aangeven dat we geen enkele rubriek historisch willen volgen, maar wel willen weten gedurende welke tijdsintervallen een object (hier een medewerker) heeft bestaan. Het historiseren van één of meer rubrieken betekent natuurlijk impliciet dat de gehele tabel pseudo-historisch wordt: weten we dat medewerker 'M1' in 1993 op adres X woonde, dan weten we ook dat het medewerker-object in 1993 bestond.

De versioning administrator maakt het mogelijk tegelijk een rubriek of een groep rubrieken te selecteren. De enige beperking is dat de rubrieken van een samengestelde verwijzende sleutel bij elkaar moeten blijven. Vervolgens

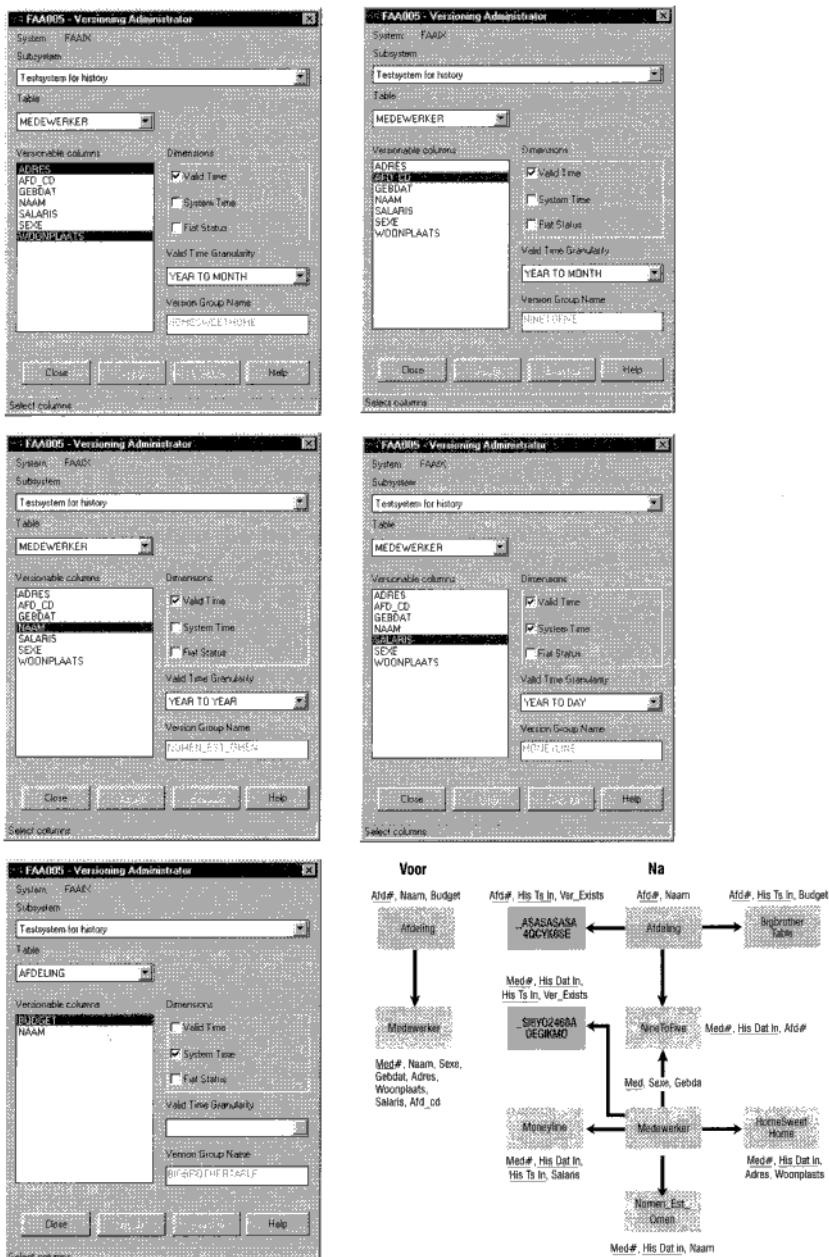
is het mogelijk om voor de uitverkoren rubriek of groep rubrieken aan te geven naar welke historiedimensies de belangstelling uitgaat. We onderscheiden daarbij de tijdsdimensies valid time (ofwel geldige tijd, ofwel tijd-in-de-werkelijkheid) en system time (ofwel transactietijd ofwel mutatietijd). Zoals we hebben gezien in het tweede hoofdstuk is elke keuze of combinatie van historie-keuzes denkbaar en zinvol (en dat geldt ook voor de nog niet besproken versiedimensie, maar dat terzijde). Wordt aangegeven dat de valid time moet worden vastgelegd, dan dient te worden aangegeven wat de granulariteit (valid time granularity) is. Kiezen we voor een granulariteit jaar dan wordt alleen de laatste mutatie in een jaar historisch vastgehouden (zie vorig hoofdstuk). Tenslotte is er bij historie steeds sprake van het aanmaken van tabellen waarin de historie is opgeslagen. Het is natuurlijk absoluut verboden dat een programmeur die tabellen direct benadert, maar de meta-DBA mag niettemin zelf een tabelnaam opgeven; we houden de schijn van controle op. Ook dit is overigens niet ongewoon: een DBA mag evengoed een index aanmaken op Medewerker.Salaris met als naam 'DEBAASISGEK'. Niemand ziet dergelijke objecten in de broncode en voor het rdbms maakt het allemaal niets uit.

De 'Apply'- en 'Execute'-buttons maken het werk tenslotte af. Na opgave van de gewenste historiestructuur voert Apply de wijzigingen door in de dictionary, waarna Execute een imposante serie 'Create Table'-, 'Alter Table'-, 'Create Constraint'- en 'Create Trigger'-statements uitvoert. Figuur 6 toont alle acties die we onze meta-DBA laten uitvoeren en de resulterende fysieke databasesstructuur. Na uitvoering van deze acties is het aantal tabellen toeegenomen tot negen. We merken het volgende op:

Bij het historiseren van de tabellen Afdeling en Medewerker dient als zodanig te worden bijgehouden wanneer van wanneer tot wanneer een medewerker bestaat (mogelijk voor beide tijdsdimensies). Het historiemechanisme voegt daarvoor aan de twee tabellen een attribuut Ver_Exists (Version Exists indicator) toe. Dit attribuut wordt vervolgens — net als attributen als Medewerker.Naam en Afdeling.Budget — weer door het historiemechanisme gehaald en resulteert in zogenaamde pseudo-versioning-tabellen met de systeemgegenererde poëtische namen "_ASASASASA4QCYK6SE" en "_SI8YO2468ACEGIKMO". Daarmee is voorzien in het fenomeen Heintje Davids-historie (zie vorig hoofdstuk). Merk op dat de tijdsdimensies die aan deze tabellen worden toegekend worden bepaald door de tijdsdimensies die voor de explicet historisch gemaakte attributen zijn gedefinieerd. De pseudo-versioning tabel van Afdeling kent daarom alleen Transaction Time (His_Ts_In), terwijl die van Medewerker beide tijdsdimensies kent (His_Dat_In en His_Ts_In).

Historisch gemaakte attributen met een verschillende granulatie moeten natuurlijk terechtkomen in diverse historie-tabellen. Is de granulatie gelijk, dan is het zowel mogelijk om ze in één tabel op te nemen (Adres en Woonplaats in Bigbrothertable), als om ze in aparte tabellen onder te brengen (Afd# in een separate tabel Ninetofive).

L'histoire ne se répète plus



Figuur 6. Acties van de META-DBA en gevolgen voor het gegevensmodel, in schema rechts.

7. De nieuwste versie van de database

De gegevensinhoud van de niet-historische database moet worden omgezet naar de historische database. Bij het (meer) historisch maken van een database is dat geen probleem. Bij het niet (of minder) historisch maken van een database leidt dit tot het vernietigen van alle gegevens uit verleden en toekomst.

De vele extra primaire en verwijzende sleutels die zijn gegenereerd dwingen een groot deel van de geldende integriteitsregels af. Er bestaan echter ook regels die niet door de gegenereerde structuren worden afgedwongen, zoals:

- een gehistoriseerde rubriek mag alleen voorkomen voor een tijdstip waarop er ook een pseudo-version-tupel bestaat
- een verplichte rubriek dient te bestaan voor elk moment waarop er een pseudo-version-tupel bestaat
- een gehistoriseerde verwijzende sleutel mag alleen verwijzen naar een niet-historisch object indien dat object gedurende het interval waarvoor de verwijzing geld ook bestaat
- alle user-defined constraints op de niet-historische database moeten worden vertaald naar het historische equivalent.

De programmeur 'ziet' de gehistoriseerde attributen als onderdeel van de niet-historische tabel. Een update van de virtuele rubriek Medewerker.Adres wordt vertaald naar een insert (of een update) op de tabel Homesweethome. Alle noodzakelijke functionaliteit kan worden uitgegenererd naar een beperkt aantal soorten database-triggers. Een insert in de logische tabel Medewerker kan daarom automatisch worden gecontroleerd en afgehandeld. Doordat de acties worden uitgegenererd in de database-trigger is er geen sprake van een client/server-bottleneck, met grote voordelen voor de performance.

Table	Column	Type	Upd?	Sngl?	Schr?	Bref?	Short	Description	Subs_Nm	Lock	Ref_Visual	Grant_J
HOMESWEETHOME	ID	T	N	N	Y	Y	JGA	Table generated by the FAA versioning mechanism	DBMDEMO:P		YEAR TO	
MEDWERKER	NAME	T	Y	N	Y	N	MED	MEDWERKER	DBMDEMO:P		YEAR TO	
MONEYLINE	ADRES	T	N	N	Y	Y	4BL	Table generated by the FAA versioning mechanism	DBMDEMO:P		YEAR TO	

Seq#	Column	AN?	IN?	Domain	Description	Subs_Nm	V_Sys_Nm	Ind?	Log	Grid	Free_Nm
1	MED_CD	N	N	MED_CD	MEDWERKER ID	DBMDEMO			N		
2	NAAM	N	N	NAAM	VOLLEDIGE NAAM	DBMDEMO	NOMENEST_OOPEN		N		
3	SEXE	N	N	SEXE	MAN OF VROUW	DBMDEMO			N		
4	GEBOAT	N	N	GEBOAT	GESCHORTEDATUM	DBMDEMO			N		
5	ADRES	N	N	ADRES	VOLLEDIG ADRES	DBMDEMO	HOMESWEETHOME		N		
6	WOONPLAATS	N	N	WOONPLAATS	WOONPLAATS	DBMDEMO	HOMESWEETHOME		N		
7	SALARIS	N	N	SALARIS	BRUTO MAANDSALARIS	DBMDEMO	MONEYLINE		N		
8	AFD_CD	N	N	AFD_CD	AFDELING WAAROP MEDEWERKER WERKT	DBMDEMO	NINETATIVE		N		
9	VER_EXISTS	I	N	INDICATOR	Column generated by the FAA versioning mechanism	VER	SIRKUZBASCEGRMO		N		

Figuur 7. De nieuwe tabel Medewerker.

L'histoire ne se répète plus

Het rdbms kent geen historie en maakt dus geen enkel onderscheid tussen originele en gegenereerde tabellen. Het historiemechanisme moet afdwingen dat iedereen afblijft van de gegenereerde tabellen.

We zien dit effect wanneer we na het doorvoeren van de historiseringssacties weer kijken naar de gegevensstructuren in de dictionary (zie de figuren 7, 8 en 9).

The screenshot shows the DBDETOB interface with the following details:

Tables, Columns tab is selected.

Keys tab is also visible.

Domains tab is visible.

R-table/Column Browser tab is visible.

Domain/Key Browser tab is visible.

System Tables section:

SysNm	Table_Nm	Type	Upd?	Sng?	Sch?	Gen	Short	Description	Subs_Nm	Lock	His_View	Grand	La
FAAX	NINNE10FME	T	N	N	Y	Y	A20	Table generated by the FAA versioning mechanism	DBMDEMOP		YEAR TO		
FAAX	NOMEN_EST_OOPEN	T	N	N	Y	Y	A20	Table generated by the FAA versioning mechanism	DBMDEMOP		YEAR TO		
FAAX	ASASASASA40CYKSE	T	N	N	Y	Y	A20	Table generated by the FAA versioning mechanism	DBMDEMOP		YEAR TO		

Columns section:

Seq#	Column	AMT	IMT	Domain	Description	Subs_Nm	V.Grp.Nm	Int?	Log	Gen	File_Nm
1	AFD_CD	N	N	AFD_CD	MEDDELINGSCODE	DBMDEMOP				Y	
2	VER_EXISTS	N	N	INDICATOR	Column generated by the FAA versioning mechanism	VER				Y	
3	HIS_TS_IN	N	N	VER_TS_IN	Column generated by the FAA versioning mechanism	VER				Y	
4	HIS_TS_END	N	N	VER_TS_END	Column generated by the FAA versioning mechanism	VER				Y	

Figuur 8. De pseudo-versioning tabel voor Medewerker.

The screenshot shows the DBDETOB interface with the following details:

Tables, Columns tab is selected.

Keys tab is also visible.

Domains tab is visible.

R-table/Column Browser tab is visible.

Domain/Key Browser tab is visible.

System Tables section:

SysNm	Table_Nm	Type	Upd?	Sng?	Sch?	Gen	Short	Description	Subs_Nm	Lock	His_View	Grand	La
FAAX	MEDWERKER	T	Y	N	Y	N	MED	MEDWERKER	DBMDEMOP		AAAZONE		
FAAX	MONEYLINE	T	N	N	Y	Y	A20	Table generated by the FAA versioning mechanism	DBMDEMOP		YEAR TO		
FAAX	NINNE10FME	T	N	N	Y	Y	A20	Table generated by the FAA versioning mechanism	DBMDEMOP		YEAR TO		

Columns section:

Seq#	Column	AMT	IMT	Domain	Description	Subs_Nm	V.Grp.Nm	Int?	Log	Gen	File_Nm
1	MED_CD	N	N	MED_CD	MEDWERKER ID	DBMDEMOP				Y	
2	SALARIS	N	N	SALARIS	BRUTO MAANDSALARIS	DBMDEMOP				Y	
3	HIS_DAT_IN	N	N	VER_DT_IN_YDAY	Column generated by the FAA versioning mechanism	VER				Y	
4	HIS_DAT_END	N	N	VER_DT_END_YDAY	Column generated by the FAA versioning mechanism	VER				Y	
5	HIS_TS_IN	N	N	VER_TS_IN	Column generated by the FAA versioning mechanism	VER				Y	
6	HIS_TS_END	N	N	VER_TS_END	Column generated by the FAA versioning mechanism	VER				Y	

Figuur 9. De gegenereerde tabel Moneyline met een terugwerkend historisch gemaakte Salaris.

Time-out?

We hebben op dit moment met gebruikmaking van een eenvoudige dictionary-architectuur al heel veel bereikt. Natuurlijk bevat dit hoofdstuk alleen de meest essentiële aspecten van het generieke historiemechanisme, maar wie dit resultaat aanhoudt tegen de besprekings in de vorige hoofdstukken, kan zien dat we voor een groot aantal problemen een generieke oplossing hebben gevonden. Dat is alleen maar mogelijk omdat het spreekwoord "l'histoire se répète" voor historische databases onverkort opgaat.

Het hier besproken generieke mechanisme kan in elk robuust rdbms worden gedefinieerd, bij voorkeur door de leverancier, maar als het historieprobleem omvangrijk genoeg is ook door een bekwame applicatie-ontwikkelaar. Natuurlijk is een implementatie wel rdbms-specifiek, al was het alleen maar omdat de trigger-syntax en -functionaliteit van product tot product verschilt. Belangrijker is echter dat een gegevensmodel waarop dit mechanisme wordt losgelaten nog geen historie bezit of dat alle historische programmatuur radicaal wordt herschreven (lees: weggegooid).

We hebben in dit hoofdstuk alleen de gegevens- en integriteitsaspecten van een generieke oplossing voor historie besproken; de functionele kant van de zaak is nog maar nauwelijks aangestipt. We zijn dus op dit moment vatbaar voor het verwijt dat we eerder hebben gemaakt in de richting van de rdbms-leveranciers en de SQL-standaardisatiecommissie. Een historie-oplossing is niet alleen een zaak van de DBA, die we hierbij hebben gepromoveerd tot meta-DBA, maar ook van de programmeurs en de eindgebruikers. In het volgende hoofdstuk zullen die betrokkenen aan hun trekken komen.

Van integriteit naar functionaliteit

In het vorige hoofdstuk hebben we een algemeen toepasbare, generieke oplossing voor het historieprobleem voorgesteld. Die oplossing gaat uit van het onzichtbaar voor de programmeur omzetten van een gegevensmodel zonder historie in een equivalent gegevensmodel met de gewenste vorm van historie. Het hoofdstuk was geheel gewijd aan de problematiek van het hergenereren van gegevensstructuren, het herdefiniëren van sleutels en het opsommen van de eveneens te genereren database-constraints en -triggers. We gaan nu verder met het bespreken van de functionele of gebruikersaspecten waarin een generieke oplossing voor het historieprobleem moet voorzien. Evenals in het voorgaande hoofdstuk zijn de voorbeelden ontleend aan de implementatie van de hier besproken modelleeroplossing die door de collega's van schrijver dezes is ontwikkeld.

De gebruiker komt in beeld

We gaan nu over naar het perspectief van de eindgebruiker. Om dat te doen moeten we de index-metafoor, waarop we onze oplossing voor het historieprobleem hebben gebaseerd, loslaten. Indexen, tablespaces, clusters enzovoort zijn fysieke constructies die niets te maken hebben met de logische structuur van de gegevens. Dergelijke constructies zijn onzichtbaar voor zowel de programmeur als de eindgebruiker. Het belang dat de eindgebruiker bij dergelijke fysieke constructies heeft kan in één woord worden samengevat: performance. Het belang van de systeemontwikkelaar is dat het aantal regels code en de complexiteit van die code sterk afneemt en dat het bovendien mogelijk wordt om de gegevensopslag te herstructureren zonder dat kostbare ingrepen in de broncode nodig zijn. We zien dat de voordeelen die een generiek historiemechanisme voor de ontwikkelaar biedt precies eerder zijn: minder regels code, minder complexe code en minder noodzaak tot onderhoud van code. Voor de gebruiker ligt het echter heel anders. Het toevoegen van historie aan een database zonder historie heeft alleen dan zin

wanneer dat werkelijk zichtbaar is voor de eindgebruikers. Anders dan bij het toevoegen van een index dient het toevoegen van historie aan een database zich te vertalen in additionele functionaliteit. Waar het bij indexen, tablespaces en clusters gaat om fysieke structuren en fysieke gegevensonafhankelijkheid, gaat het ons bij historie ook om additionele logica en logische gegevensonafhankelijkheid. Het tot stand brengen van logische gegevensonafhankelijkheid is sinds de introductie van het relationele model in 1969 de Heilige Graal van de database-theorie gebleken. In bijna dertig jaar is het idee van gegevensonafhankelijkheid voorbij het domein van fysieke structuren een fata morgana gebleken. Nu zullen we die noot moeten kraken. In het afgelopen jaar is dat door mijn collega's gebeurd en de voorbeelden die we hieronder zullen geven zijn afkomstig van hun implementatie.

The screenshot shows a database application window with two tables displayed in separate windows:

Afdeling Table:

afd_cd	naam	budget
A1	Logistiek	1500000
A2	Ballistiek	1467841
A3	Statistiek	780000
A4	Heuristiek	200000

Medewerker Table:

med_cd	naam	sext	geboorted	adres	woonplaats	salair	afd_cd	naam
M1	Johnny Constantine	M	1959-12-31	Middenstraat 99	Oud Beijerland	5000.00	A1	Logistiek
M2	Donna Telf	V	1953-12-31	Verborgen laan ??	Kerkenhuizen	6400.00	A1	Logistiek
M3	"?"	V	1948-02-29	Zwierpelaag 89	Moordrecht	6869.00	A4	Heuristiek
M4	Isaac Asimov	M	1920-04-29	Dowmwhere Alley	Aurora	6500.00	A3	Statistiek
M5	Karen Carpenter	V	1952-04-18	Sheobeedoo Ooste	Xanadu	8000.00	A3	Statistiek

Figuur 10. De actuele inhoud van de tabellen Afdeling en Medewerker.

We beginnen met een eenvoudige blik op de tabellen Afdeling en Medewerker in figuur 10. Er bestaan vier afdelingen en vijf medewerkers en we zien geen spoor van historie. Dit betekent dat we niet weten wanneer de gegevens zijn ingevoerd en gewijzigd. We kunnen niet direct zien of een logische update van een bepaald gegeven leidt tot het vernietigen van informatie (een fysieke update) of tot het voeren van een wijziging van het gegeven voor een bepaald tijdsinterval (een fysieke insert in één van de gegeneerde tabellen). Slechts door het raadplegen van figuur 6 (zie pagina 57) kunnen we zien dat een update van Medewerker.Gebdat onherroepelijk is, terwijl een update van Medewerker.Salaris nooit tot enig verlies aan informatie kan leiden. We zien ook dat bekend moet zijn wanneer een medewerker is opgevoerd en wanneer hij in de geldige tijd is ontstaan. Voor een afdeling weten we wel wanneer deze is opgevoerd, maar niet wanneer deze in de

Van integriteit naar functionaliteit

wereld buiten het systeem is ontstaan. Conform de HTC veronderstelt het historiemechanisme dus dat elke afdeling altijd heeft bestaan en altijd zal blijven bestaan, ondanks het voorkomen van mutatietijdstippen in de tabel _ASASASASASA4QCYK6SE. Voor de aandachtige lezer merken we op dat deze constellatie van historie zich niet verdraagt met een user-defined constraint die in hoofdstuk 3 aan dit voorbeeld was toegevoegd: "Elke afdeling heeft minstens één medewerker (voor elk moment in de tijd)." We zullen deze constraint voor het gemak maar vergeten.

De tijd komt in beeld

Bekijken we figuur 10 aandachtiger dan zien we toch een glimp van historie. De Afdeling- en Medewerker-windows bevatten rechtsboven drie knoppen, waarvan vooral de meest linker iets van een tijdsaspect suggereert. Verder zien we onderin het master-window een tijdsinterval in de geldige tijd en een tijdstip in de transactietijd. We kijken aan tegen het heden in zowel de valid time en de transaction time: het onvergetelijke moment waarop ondergetekende dit hoofdstuk schreef. Die tijdstippen hebben betrekking op het actuele window: in dit geval natuurlijk het Medewerker-window. Activeren we het Afdeling-window dan zien we alleen de Transaction Time, daar de Valid Time voor Afdeling niet wordt bijgehouden.

The screenshot shows three overlapping Microsoft Access windows, each displaying a table named 'Medewerker' (Employee). The top window shows data for the current valid time (ValidTime = 1999-01-01). The middle window shows data for the transaction time (TransactionTime = 1999-12-31). The bottom window shows data for the previous valid time (ValidTime = 1999-12-31).

med_id	naam	sleutel	geboorte	gesl.	adres	woonplaats	telefoon	std_id	team
M1	Johnny Comelyste	M	1959-12-31	M	Milieuwinkelsteeg 99	Oud Beijerland	5000,00	A1	Logistiek
M2	Donna Tofft	V	1963-12-31	V	Verborgen leen ??	Nergenshutzen	6400,00	A1	Logistiek
M3	""	V	1948-02-29	Z	Zweepdag 69	Moordrecht	6969,00	A4	Heuristiek
M4	Isaac Asimov	M	1920-04-28	M	Dowmwhen Alley	Aurora	9500,00	A3	Statistiek
M5	Karen Carpenter	V	1952-04-19	V	Shooneedoo Clos	Xandu	6000,00	A3	Statistiek

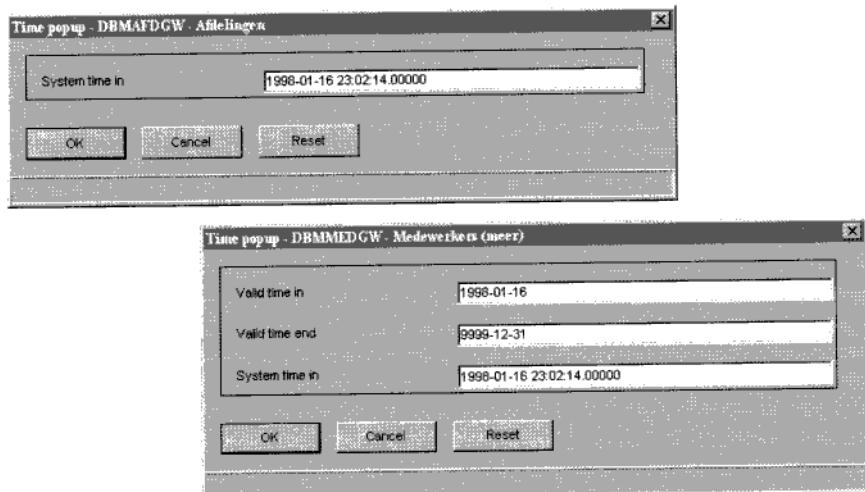
Figuur 11. De tabel Medewerker op drie tijdstippen in de valid/transaction time.

De tijdstippen voor de twee windows van figuur 10 zijn beide dezelfde: het NU. Het wordt pas leuk wanneer we de gegevens voor verschillende tijdstippen bekijken. Figuur 11 toont een drietal Medewerker-windows: het bovenste toont inhoud van de tabel op 1 januari 1980, het middelste de actuele situatie en het onderste de verwachte situatie op 1 januari 2000. (Het systeem is echt hi-tech, want millennium-bestendig!). De transaction time is steeds gelijk aan het heden. Zouden we deze instellen op 15 januari 1998 dan zouden we drie lege windows zien: alle gegevens zijn namelijk na die datum ingevoerd. In 1980 kenden we slechts de medewerkers 'M0' (Heintje Davids) en M2 ('Donna Tartt'). Op dit moment is de eerste even weg, maar in 2000 zien we hem weer terug. Verder zien we onder meer dat tussen nu en 2000 medewerkers zullen verhuizen, van naam en van afdeling zullen veranderen en in salaris vooruit zullen gaan. Misschien is ook hun sekse of geboortedatum aan verandering onderhevig, maar figuur 6 laat zien dat het systeem daarover nooit uitsluitsel kan geven. Willen we die mutaties ook kunnen volgen dan moeten we onze toevlucht zoeken tot het op gepaste tijden uitdraaien van informatie, het opzetten van een datawarehouse of natuurlijk het als-nog historiseren van deze attributen.

Ziedaar logische gegevensonafhankelijkheid. De programmeur ontwikkelt zijn schermen voor de actuele situatie en het historiemechanisme zorgt op basis van de door de meta-DBA opgegeven instellingen er voor dat de geldige gegevens worden opgehaald uit de tabellen NineToFive, HomeSweetHome, Nomen_Est_Omen, Moneyline en natuurlijk Medewerker zelf. Daarbij moeten we bekennen dat we de gehistoriseerde rubrieken niet fysiek uit de Medewerker-tabel hebben verwijderd en dat we de actuele gegevens redundant in Medewerker bijhouden. Door deze onzichtbare toepassing van het schaduwtafel-mechanisme (zie hoofdstuk 1) wordt het mogelijk om het opvragen van actuele gegevens direct uit te voeren op de tabel Medewerker. De gebruiker die niet geïnteresseerd is in het verloop van gegevens merkt daardoor bij raadplegen niets van de historiseer-acties van de meta-DBA. Het opvragen van gegevens voor andere momenten dan het heden vereist natuurlijk een uitgebreide join-actie waarbij in het Medewerker-voorbeeld zes tabellen betrokken zijn, maar een beetje modern rdbms doet dat niet kreunend doch fluitend.

De vraag dringt zich op hoe de gebruiker kan aangeven naar welke tijdstippen zijn of haar interesse uitgaat. Daarmee komen we terecht bij de meest linker van de drie genoemde historie-buttons. Figuur 12 laat zien wat er gebeurt wanneer de medewerker de knop met het analoge klokje activeert vanuit het Afdeling-window respectievelijk het Medewerker-window. Activering van de Time Popup-button op afdelingsgegevens leidt tot een popup-window waarin de systeemtijd (transaction time) kan worden opgegeven. Meer is niet zinvol omdat alleen deze tijdsdimensie voor Afdeling wordt vastgelegd. Desgewenst kan voor deze dimensie een tijdstip in het verleden worden ingevuld. Een tijdstip in de toekomst is alleen zinvol wanneer het systeem kan functioneren als een kristallen bol en is dus verboden.

Van integriteit naar functionaliteit



Figuur 12. De Time Popup-button in actie op Afdeling- en Medewerker-windows.

Wie een tijdstip in het verleden selecteert ziet de situatie zoals die op dat moment bekend was, maar kan geen mutaties meer doorvoeren. Wie hiervan niet direct de logica inziet raadplege de opmerkingen over transaction time in hoofdstuk 2.

Activeren we de Time Popup-button op medewerker-gegevens dan zien we een uitgebreider scherm waarin we naast een tijdstip in de transaction time ook een interval in de valid time kunnen opgeven. De gegevens die vervolgens worden getoond hebben betrekking op de het tijdstip transaction time/begin interval valid time. De rubriek "Valid time end" maakt het mogelijk om bij een wijziging aan te geven tot welk moment deze geldt. Standaard is dat het einde der tijden. Selecteren we voor de valid time een ander tijdstip dan het heden dan mogen we in beginsel naar hartelust muteren, mits de transaction time maar blijft ingesteld op het heden. Merk op dat met name mutaties in het verleden tot verwarring kunnen leiden, tenzij de meta-DBA zowel de valid time als de transaction time heeft geactiveerd.

De gebruiker krijgt al met al met de Time Popup-button een beeld van de wijze waarop de tabel is gehistoriseerd (zij het niet per rubriek) en daarmee van het effect van zijn mutaties. Zou de meta-DBA besluiten om het salaris alleen valid time historisch op te slaan dan zou de Time Popup-button daarna alleen de valid time tijdsparameters tonen. Verwijderd de meta-DBA alle historie van de tabel Medewerker dan verdwijnen alle drie de historie-buttons van het scherm. Activeert de gebruiker de Time Popup-button in het master-window dan worden alle dimensies getoond en toegepast op alle geopende windows.

Integriteitsmanagement: nog één keer slikken

Daarmee komen we nog eenmaal bij een onaangenaam probleem dat te maken heeft met integriteitsmanagement. Het historiemechanisme dient de integriteitsregels af te dwingen voor elk moment in de tijd, om preciezer te zijn: voor elk punt in het valid/transaction time-vlak (zie figuur 3). Een mutatie voor een tijdsinterval is te beschouwen als een mutatie over een (eindig) aantal tijdstippen en dat geeft problemen bij het afdwingen van alle niet-historische integriteitsregels.

Standaard relationele constraints zoals domeinregels, primaire sleutels, alternatieve sleutels en verwijzende sleutels kunnen eenvoudig historiebestendig worden gemaakt. Anders ligt het bij complexe user-defined constraints. Daarvoor geldt de volgende deprimerende regel:

Regel 7:

Er bestaat geen generiek mechanisme dat een willekeurige constraint controleert voor alle momenten in het tijdsinterval en het is vanuit het perspectief van performance ondoenlijk om op een brute-force wijze elk tijdstip in het interval te valideren.

Het enige dat mogelijk is, is het bedenken van een aantal heuristieken. Een voorbeeld is het uitvoeren van een controle op het begin- en eindtijdstip van de mutatie en voor het heden (indien dit in het interval ligt). Zo'n benadering is wel praktisch maar leidt toch tot een potentieel integriteitslek in een systeem. Bedenk wel dat dit niets met het historiemechanisme te maken heeft, maar inherent is aan het werken met een historisch systeem dat mutaties voor andere tijdstippen dan het heden toelaat. Wie dergelijke mutaties verbiedt heeft geen probleem maar in de meeste gevallen schieten we dan wel met een kanon op een praktijkmuug.

Wel vervelend is het probleem dat het in veel gevallen niet mogelijk is om in één SQL-statement na te gaan of een database voor alle tijdstippen integer is. Neem bijvoorbeeld een mogelijke regel die uitdrukt dat de som van de salarissen van de medewerkers op een afdeling niet groter mag zijn dan het afdelingsbudget. Het is onmogelijk om het eenvoudige SQL-statement dat deze regel uitdrukt te vertalen in een SQL-statement dat dit voor elk tijdstip uitvoert en ook nog performt. Toch is het in batch-modus kunnen controleren van de integriteit van een historische database van groot belang en dus werkt één van mijn collega's op dit moment aan een generiek mechanisme om willekeurig complexe user-defined constraints te controleren voor elk moment in de tijd waarop zo'n regel redelijkerwijs kan zijn geschonden.

De nut-en-noodzaak-discussie

Alles wat we tot nu toe hebben besproken is gebaseerd op het concept van de HTC. De HTC maakt het mogelijk om de gegevensstructuren, de integri-

Van integriteit naar functionaliteit

teitsregels en de programma's van een systeem te transformeren naar een equivalent voor een andere vorm van historie. Databases en programma's die inwerken op databases worden vrijwel probleemloos historie-bewust en tentoongespreiden een voorspelbaar en logisch gedrag. Een mechanisme als dit is voor veel systemen van onschabare waarde. Een technisch topmanager van een leverancier van ziekenhuissystemen vertelde ons onlangs dat zijn systemen een factor tien goedkoper zouden uitvallen wanneer er geen tijdsproblematiek aan de orde zou zijn. Voor ons buikgevoel is een factor drie in de meeste historie-gevoelige omgevingen realistischer, maar het potentiële kostenvoordeel is in elk geval groot. De algemeen directeur van hetzelfde bedrijf had ons al eerder ongelovig gevraagd of een generiek historiemechanisme op zijn pakket zou kunnen worden losgelaten. We konden hem geruststellen: een generiek historiemechanisme kan pas echt van nut zijn wanneer het wordt losgelaten op een gegevensverzameling zonder enige vorm van historie. Dit is een uiting van het orthogonaliteitsprincipe dat al eens eerder onderwerp van bespreking is geweest [15]. Wie een generiek mechanisme wil installeren in een bestaande omgeving dient dus alle bestaande historie-aspecten uit zijn gegevensmodel en programmacode te slopen. In de meeste gevallen is het dan beter om het systeem opnieuw te bouwen, temeer daar een generieke oplossing ook enkele eisen aan de structurering van de broncode stelt.

De Historie-Transformatie Conventie voorbij

Kijken we naar de historie-werkelijkheid van alledag dan zullen we mogelijk ook in nieuwbouwomgevingen bezwaren ontmoeten. Neem bijvoorbeeld aan dat de gebruiker in bulk wijzigingen wenst door te voeren op de adresgegevens van de medewerkers. En neem aan dat die verhuisdata op vele verschillende tijdstippen vallen. De gebruiker zou dan na vrijwel elke wijziging de Time Popup-button moeten activeren en mogelijk als RSI-slachtoffer (muis-gehandicapte) in de WAO belanden. Het zou dus mogelijk moeten zijn om de tijdgegevens per getoond record op te geven.

Dit probleem is goed oplosbaar en als we het al hadden geïmplementeerd zou ik het zeker hier bespreken. Moeilijker wordt het wanneer de gebruiker vragen heeft die betrekking hebben op meerdere tijdstippen. Voor dergelijke cross-temporal informatiebehoefte geldt een eenvoudige regel:

Regel 8:

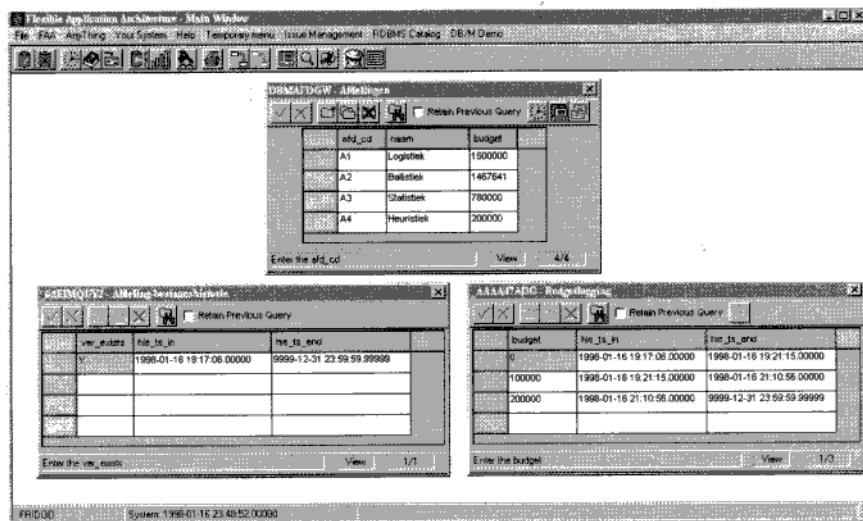
Het is theoretisch onmogelijk om een generieke oplossing te formuleren voor een informatiebehoefte die betrekking heeft op meer dan één moment in de valid/transaction time.

Neem als zeer eenvoudig voorbeeld de wens om van de medewerkers de huidige naam en een toekomstig salaris te tonen. Zijn beide gegevens verplicht dan is niet duidelijk wat het historiemechanisme moet doen wanneer

de medewerker nu nog niet of straks niet meer bestaat. Meer ingewikkelde (en interessante) voorbeelden hebben betrekking op mutaties met terugwerkende kracht. Volgens regel 8 is het niet mogelijk om een generieke procedure te bedenken die het mogelijk maakt om bijvoorbeeld een nabetaling te berekenen voor een salarisverhoging met terugwerkende kracht. Dit is overigens iets minder erg dan het lijkt. Een volledig generieke, algemeen geldige, oplossing voor cross-temporal behoeften mag dan onmogelijk zijn, voor een specifiek systeem is een algemeen geldende oplossing wel mogelijk. Een pensioenverzekeraar die op een generiek historiemechanisme zoals hier beschreven een procedure schrijft die de aanspraken herberekent bij een salarisverhoging, kan dat programma vermoedelijk volledig hergebruiken voor het doorrekenen van het effect van een veranderende geboortedatum. En lukt ook dat niet dan resteert een aanzienlijke vereenvoudiging van de programmatuur.

Ook al noopt de theorie tot pessimisme, in de praktijk zijn er allerlei praktische gimmicks mogelijk die in veel gevallen afdoende zijn. Een zo'n mechanisme gaat schuil achter de tweede historie-button, de Verloop-button. Figuur 13 en 14 laten zien wat het effect van het activeren van deze button kan zijn op de tabellen Afdeling en Medewerker.

De Verloop-button is alleen actief wanneer de gebruiker een gehistoriseerde rubriek of een primaire sleutelrubriek selecteert. In het geval van de Afdeling-tabel is de button dus niet actief wanneer de cursor op Afdeling.Naam is geplaatst en hetzelfde geldt voor Medewerker.Gebdat en Medewerker.Sekse. De twee onderste windows in figuur 13 zijn het resultaat van het activeren van de ladenkast-button vanaf de rubrieken Afd# respec-



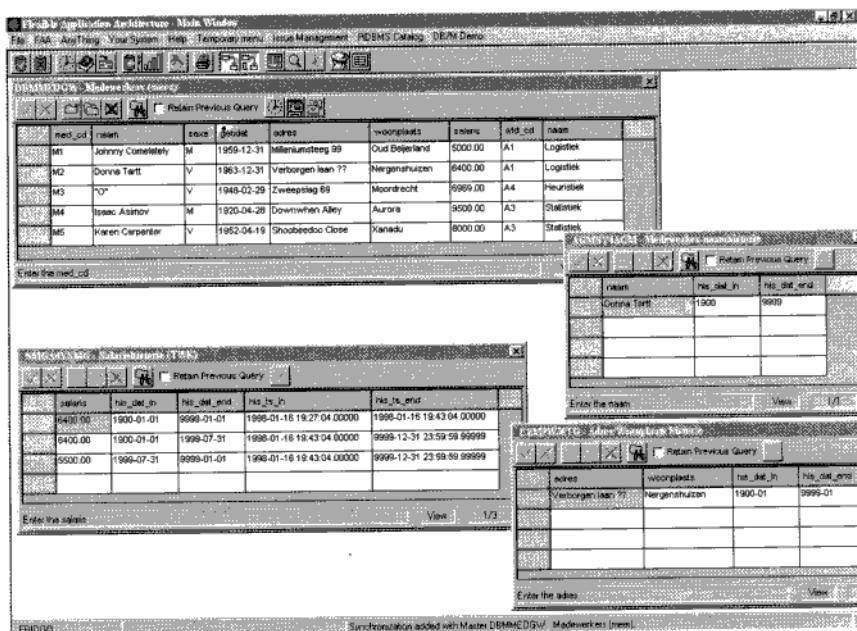
Figuur 13. De Verloop-button in actie op Afdeling 'A4' Afd# en Budget.

Van integriteit naar functionaliteit

tievelijk Budget. Het linker window toont het verloop van het bestaan van de afdeling die in het bovenste window is geselecteerd. Het rechter window doet hetzelfde voor het verloop van het afdelingsbudget. Omdat het gaat om de transaction time dimensie is de tijdgranulatie zo verfijnd als het rdbms toestaat.

Uiteraard zijn de verloopschermen nietmuteerbaar omdat ze zijn gedefinieerd op de gegenereerde tabellen _ASASASASASA4QCYK6SE en BigbrotherTable.

De situatie voor de tabel Medewerker in figuur 14 is met deze kennis eenvoudig te interpreteren. Merk op dat de salarishistorie beide tijdsdimensies toont en dat de valid time rubrieken worden getoond conform de granulatie die door de meta-DBA in het voorgaande hoofdstuk is opgegeven. Uit het



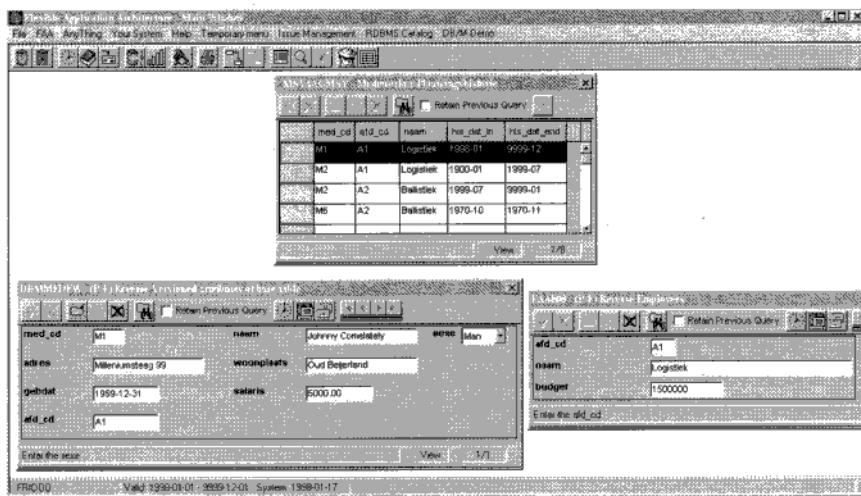
Figuur 14. De Verloop-button in actie op Medewerker: Med#, Naam en Adres/Woonplaats.

salarisverloop-window kunnen we opmaken dat de salarisgegevens zijn ingevoerd op 16 januari 1998 tussen half acht en kwart voor acht 's avonds. Tot kwart voor acht gold een salaris van f 6.400 over het interval 1/1/1900 tot het einde der tijden. Om 19:43 werd aangegeven dat de medewerker per 31/7/1999 f 5.500 zal gaan verdienen. Deze enkele mutatie leidt onvermijdelijk tot twee inserts. Wie vraagt naar het salaris op 1/1/2000 met de kennis die het systeem bezat voor kwart voor acht 's avonds op 16/1/1998 krijgt f 6.400 terug, voor mutatietijdstippen na dat moment is het antwoord f 5.500.

Hoeveel eenvoudiger is het om een programma te maken dat een eenmalige inhouding van f 900 bruto verzorgt.

Mutaties mogen zoals bekend alleen worden doorgevoerd op de tabel Afdeling zelf. Sterker nog: het is verboden om welke vorm van programmatuur op deze tabellen te ontwikkelen. De door de Verloop-button geactiveerde windows zijn gecreëerd door middel van generiek mechanisme en de gewenste vorm van presentatie is gedefinieerd met een generiek presentatiemechanisme. Alle programmatuur op de historietabellen is gegenereerd door het dictionary-hulpmiddel en dus bestand tegen veranderende wijzen van historisering door de meta-DBA. We besteden hier geen aandacht aan dergelijke generieke mechanismen, omdat dit de aandacht afleidt van de historieproblematiek. We merken wel op dat ze van wezenlijk belang zijn voor een effectieve oplossing voor het historieprobleem. Ondanks dat kan ik het niet laten om op te merken dat de schermen op Afdeling en Medewerker eveneens zijn gegenereerd en geparametriseerd en wel door precies hetzelfde mechanisme dat de verloop-tabellen toont. Het is daardoor ook mogelijk om de historie-tabellen los van de actuele tabellen te bekijken, om single-record windows van de verloopgegevens aan te maken of om de master/detail-verhouding om te draaien. Vooral dit laatste biedt geweldige mogelijkheden: een constructie waarin alle medewerkers worden getoond die ooit hebben gewerkt op afdeling 'A2' is eenvoudig gemaakt.

We verlaten de bespreking van de Verloop-button met figuur 15 die op basis van de gegenereerde tabel NineToFive laat zien welke afdeling en medewerker er op 1 januari 1998 schuil gingen achter de combinatie 'M1'/'A1'. Ook hier is weer van het flexibele master/detail systeem gebruik gemaakt,



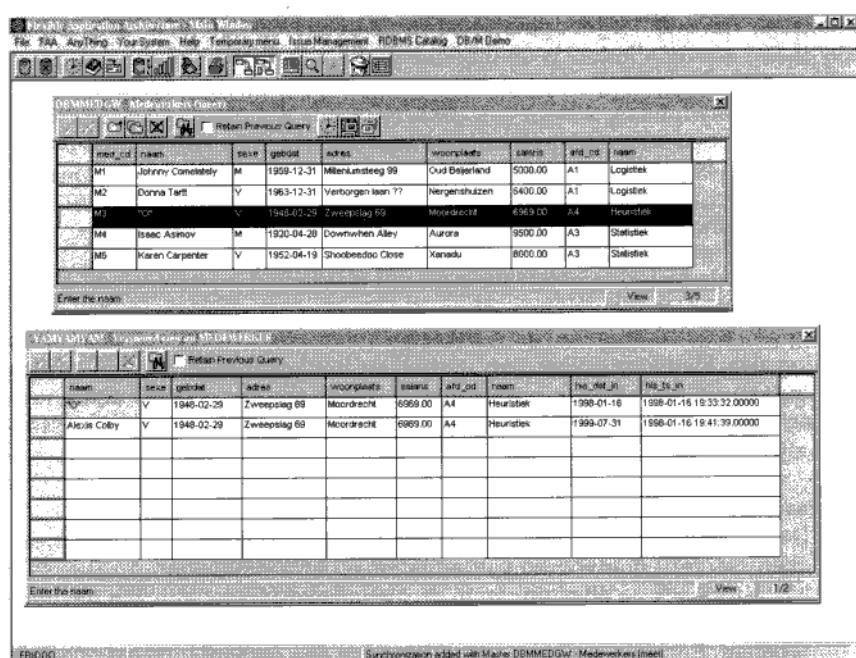
Figuur 15. Een 'hands free' constructie op basis van de historie-tabel Ninetofive.

Van integriteit naar functionaliteit

maar nu zijn meester en slaaf verwisseld. Door het generieke mechanisme dat dit verzorgt maakt het allemaal niets uit. We zien dat de medewerker die op die datum Johnny Comelately heette werkzaam was op de afdeling die altijd Logistiek zal blijven heten en dat die relatie tot het einde der tijden zal blijven bestaan. Wanneer deze informatie aan het systeem kenbaar was gemaakt valt niet te achterhalen omdat de tabel NineToFive geen transaction time kent. Bedenk dat deze constructie geen enkele vorm van programmeering vereist en door een willekeurige gebruiker kan worden gerealiseerd. We zijn niet alleen de schaamte maar ook de beperkingen van de HTC voorbij!

Totaalverloop: de overviewview en user-defined constraints

Het idee van een Verloop-button is leuk, maar niet leuk genoeg. Het probleem is dat de door deze button gegenereerde schermen samenvallen met de specifieke historische groep. Een gebruiker heeft daarvan niet altijd een boodschap. De gegevens waarvan de gebruiker het verloop wenst te zien kunnen van geval tot geval verschillen. Om aan deze wensen tegemoet te komen genereert het historiemechanisme per gehistoriseerde tabel een view die het historische verloop van alle gegevens in die tabel toont. Per mutatie levert deze overviewview één record. Activering van de meest rechter button



Figuur 16. De geschiedenis van medewerker "O".

van de drie historiebuttons levert een totaal verloop op voor de geselecteerde medewerker 'M3' (figuur 16).

Natuurlijk is alles wat mogelijk is voor de individuele historische tabellen ook mogelijk op de gegenereerde overviewview. De overviewview is ook een uitkomst voor wie opziet tegen het bouwen van een echte parser om SQL-query's en constraints op correcte wijze van de vastgelegde historie gebruik te laten maken. Dat die noot op enigerlei wijze gekraakt moet worden zal duidelijk zijn. Vervang de tabelnamen in een query door de namen van de overviewviews (natuurlijk automatisch), voeg voor de tijdsdimensies tijdstippen toe waarnaar de interesse uitgaat en voilà.

Tenslotte: logische deletes en fysieke forgets

Ons historiemechanisme biedt ondersteuning voor inserts, updates en deletes, waarbij we hebben gezien dat er voor wat betreft de valid time dimensie sprake is van een interval waarop de mutatie betrekking heeft. Overigens levert een logische delete database-technisch gezien geen fysieke delete op maar enkele updates en inserts. Een historische database heeft dus de neiging om onbeperkt te groeien. De logische delete is geïmplementeerd als button in de diverse schermen. Het is echter tevens denkbaar dat een gebruiker (logischerwijs de applicatiebeheerder) een bepaald gegeven werkelijk wenst te verwijderen. In hoofdstuk 3 maakten we daarvoor al melding van de noodzaak van een Forget-operator naast de delete-operator. De 'schaar-button' in het master window is de implementatie van deze behoefte aan vegetelheid. Uiteraard is het mogelijk om deze button voor een gebruiker van het scherm te verwijderen. Alleen een geautoriseerde gebruiker mag informatie definitief uit een historisch systeem verwijderen.

Enkele implementatie-gerelateerde opmerkingen

We hebben de historieproblematiek nu vrij diepgaand besproken, zij het op een sterk conceptueel niveau. Voor daadwerkelijke implementatie is dit boekwerk natuurlijk geen goede handleiding. Wel biedt het naar onze mening een blauwdruk voor een dergelijke implementatie. We zullen hier niet ingaan op de techniek van realisatie, maar noemen slechts een aantal eigenschappen van dbms- en frontend-producten die gewenst of zelfs essentieel zijn:

Rdbms-aspecten

- Database-triggers — Van belang voor het bedrijfszeker en snel (geen client/server!) doorvoeren van mutaties op historietabellen
- Meta-rollback — Wijzigingen in de historische constellatie zijn gevoelig voor fouten. Een foutsituatie dient te leiden tot het 'ontcreëren' van tabellen. Een rdbms dat na elke DDL-actie een commit uitvoert noopt de programmeurs tot het bouwen van allerlei overbodige cleanup-software.

Van integriteit naar functionaliteit

- Alerters — Het voorkomen van toekomstmutaties moet leiden tot het bijwerken van de actuele tabellen bij het verstrijken van het moment waarop deze mutaties actueel worden. Het is zeer uit oogpunt van integriteitsbeheer wenselijk dat een rdbms dergelijke 'time-triggered' acties ondersteunt.

Frontend-aspecten

- Dynamische creatie van schermobjecten — Schermobjecten zoals buttons en velden dienen dynamisch te worden gecreëerd, afhankelijk van de historie-constellatie zoals beschreven in de dictionary. Is dit niet mogelijk dan is het onvermijdelijk om broncodegeneratoren te schrijven, met alle beheersproblemen van dien.
- Instantieerbare OO klasseboom — Historie moet kunnen inwerken op eerder ontwikkelde programmatuur zonder dat deze behoeft te worden aangepast. Voor wat betreft de functionele aspecten vraagt dit in de praktijk om een objectgeoriënteerd ontwikkelhulpmiddel dat het toelaat dat de frontend-klassen worden geïnstantieerd door de ontwikkelaars.

Wie met moderne hulpmiddelen werkt kan direct aan de slag, maar natuurlijk is het verre te prefereren dat de leveranciers van ontwikkelhulpmiddelen zelf de handschoen opnemen. Wie vandaag begint met een omvangrijke applicatie met veel historie-ellende staat al met al voor een moeilijke keuze.

Beyond history ...

Bij het schrijven van mijn eerste artikel over historieproblematiek in DB/M [zie ook 15] had ik al het gevoel dat historie de poort is naar nog veel interessantere problemen. De historie-aspecten waarop ik destijds door gebrek aan DB/M-ruimte en — toegegeven — ook door gebrek aan kennis moest laten liggen zijn nu, zoals destijds beloofd, alsnog besproken. Deze uitgave is overigens hopelijk niet het laatste woord, maar mogelijk wel mijn laatste woord over de historieproblematiek. Van de nog interessantere problemen voorbij historie zijn op dit moment nog slechts vage contouren zichtbaar. Op die contouren wil ik, zonder veel pretenties, ingaan in het volgende hoofdstuk.

Voorbij tijd en werkelijkheid

In dit hoofdstuk werpen we nog eenmaal een blik terug op de vorige hoofdstukken (natuurlijk zonder nodeloos in herhaling te vallen) en behandelen we alsnog een praktisch aspect van historie dat we tot nu hebben laten liggen. Daarna verkennen we heel voorzichtig het perspectief dat voorbij het historieprobleem opdoemt. Voorbij een generieke oplossing van het historieprobleem begint een even fascinerende als fuzzy wereld van kansen, offertes, visies, scenario's, what-ifs, extrapolaties en nog veel meer. We hebben het dan over de problematiek van 'versioning', een informatica-problematiek aan (of misschien voorbij) de grenzen van het menselijk weten.

Een korte recapitulatie van het voorafgaande. In hoofdstuk 1 werd het historieprobleem geïntroduceerd als een probleem dat eigenlijk geen introductie behoeft. Het historieprobleem is bij nagenoeg iedere administratief gemaakte IT'er bekend omdat het in heel veel administratieve systemen voorkomt en er nog geen schijn van een tool-oplossing in zicht is. In de daarop volgende hoofdstukken werd aan de hand van de Kadaster-casus geschatst dat het historieprobleem slechts bij acceptatie van een zeer stringent denkkader te kraken is, en dan nog maar gedeeltelijk en tegen zeer hoge bouw- en onderhoudskosten. De Kadaster-casus met zijn terugwerkende kracht gegevensvastlegging (tweedimensionale historie) zal alleen door sadistisch aangelegde lezers (en door masochistisch aangelegde Kadaster-medewerkers) als inspirerend zijn ervaren. In hoofdstuk 3 bleek vervolgens hoe hopeloos complex de theorie van een historische database is. In hoofdstuk 2 werden met hulp van Barbra Streisand twee elementaire vormen van historie geïntroduceerd (valid time en transaction time) die voor elke rubriek in een database – en desgewenst tegelijkertijd – kunnen optreden (figuur 3).

In hoofdstuk 3 werd daarna aannemelijk gemaakt dat de informatie in een database waarin niet aan alle gegevenssoorten dezelfde vorm van historie (niet-historisch, valid time historisch, transaction time historisch, valid èn transaction time historisch) is toegevoegd niet altijd correct kan worden geïn-



terpreteerd of gemuteerd. Verder bleek in een historische database dat logische update- en delete-operaties moeten worden omgezet in fysieke inserts en updates en dat een historische database derhalve de neiging heeft om in omvang toe te nemen. Er was goed beschouwd maar één stukje goed nieuws, namelijk de Historie-Transformatie Conventie (HTC) die — mits geaccepteerd — een mogelijkheid biedt om gegevens in een vorm van historie om te zetten in een andere vorm van historie. Op basis van die HTC — die gelukkig door iedereen al dan niet stilzwijgend wordt geaccepteerd en toegepast — werd in hoofdstuk 4 en 5 een benadering voor het historieprobleem geïntroduceerd die de complexiteit van een historische applicatie onzichtbaar maakt voor de programmeur. Het voorgestelde recept lijkt de steen der wijzen:

- ontwerp en implementeer een historieloos gegevensmodel
- programmeer er op los
- voeg historie toe naar smaak
- hergenereer de database en de programmatuur
- point 'n click eventueel nog een handvol historie-schermen.

Het lijkt te mooi om waar te zijn. Waar staan de kleine lettertjes? Waarom is een dergelijke generieke oplossing voor het historieprobleem niet al lang gemeengoed? Waarom moet een radicaal generiek model waarmee het historieprobleem wordt gekraakt uit de Nederlandse polder komen?

Oplosbare en onoplosbare vraagstellingen

Om met die eerste vraag te beginnen: de kleine lettertjes zitten in dezelfde Historie-Transformatie Conventie waarop de door ons uitgewerkte oplossing is gebaseerd. We hebben dat al even aangestipt in het vorige hoofdstuk (zie Regel 8), maar enige verdieping lijkt gewenst. Die HTC maakt het mogelijk een historische interpretatie te geven aan een niet-historische database.

Op basis van die vertaling van niet-historie naar historie kunnen slimme conversieprogramma's worden bedacht die we kunnen loslaten op historieloze programma's en gegevensstructuren. Maar dit alles voorziet beslist niet in het invullen van elke willekeurige informatiebehoefte. Figuur 17 geeft voorbeelden van vergelijkbare historie-vragen die op basis van de in hoofdstuk 4 en 5 geschatte oplossing wel/niet oplosbaar zijn.

Voor het goede begrip: de term 'oplosbaar' houdt in dat het mogelijk is om een query of een mutatie op een database zonder historie algoritmisch, conform de HTC, te converteren naar een gelijkwaardige query of mutatie met de gewenste vorm van historie. Voor de 'onoplosbare' vragen is dat niet mogelijk op een wijze die gegarandeerd correct is. De reden daarvoor is dat het zonder extra informatie niet mogelijk is om elk voorkomen van medewerkers en rekeninghouders correct te behandelen. Wat bijvoorbeeld te doen met na 1 juli 1993 opgezegde rekeningen en per 1 februari 1998 in dienst getreden medewerkers, om maar twee van de eenvoudigste voorbeelden te noemen. De vragen zijn natuurlijk niet echt onoplosbaar: de oplossing

Voorbij, tijd en werkelijkheid

bestaat uit een stukje informatieanalyse en enig klassiek programmeerwerk. Onoplosbaar betekent slechts dat dit handwerk nimmer kan worden overgenomen door algemeen toepasbare tools.

Oplosbare vragen

1. Geef de gegevens per 1 juli 1993 van de medewerkers die op die datum werkzaam waren op de afdeling inkoop
2. Reproduceer het bankafschrift per 1 juli 1993 van rekeningnummer 182093.
3. Ken een marktwaarde toeslag toe aan de IT-medewerkers in functiegroep 'M10' en hoger.
4. Toon de gegevens van medewerkers wier maandsalaris hoger is dan 2000 Euro's.

"Onoplosbare" vragen

1. Geef de gegevens per 1 juli 1993 van de medewerkers die vandaag werkzaam zijn op de afdeling inkoop.
2. Reproduceer het bankafschrift per 1 juli 1993 van rekeningnummer 182093 met de actuele adresgegevens van de rekeninghouder.
3. Ken een marktwaarde toeslag toe aan de IT-medewerkers in functiegroep 'M10' en hoger ingaande per 1 januari 1998.
4. Toon de gegevens van medewerkers wier maandsalaris in de periode van 1 juli 1993 tot heden hoger is dan 2000 Euro's.

Figuur 17. Voorbeelden van beantwoordbare en onbeantwoordbare vragen.

Wat is het verschil tussen de oplosbare en de onoplosbare vragen? Antwoord: de oplosbare vragen hebben betrekking op gegevens die alle betrekking hebben op hetzelfde punt in de tijd, of – nog beter – op hetzelfde punt in het valid time/transaction time-tijdvak (zie figuur 3, rechtsonder zie pagina 29). De onoplosbare vragen hebben daarentegen betrekking op meerdere tijdstippen of op tijdsintervallen (lees: vele tijdstippen). Vrij naar Isaac Asimov en in navolging van sommige 'temporal database'-specialisten kunnen we dan praten over "crosswhen query's". Query's of mutaties in de toekomst (upwhen) of in het verleden (downwhen) vormen geen enkel probleem voor ons historiemechanisme, maar tegen crosswhen query's is geen kruid gewassen. Het betreffende tijdgebonden programmeerwerk wordt door het transparant zijn van de historie overigens een stuk gemakkelijker en blijft bovendien correct werken bij een wijziging van de vorm van historie waaraan de gegevensstructuren onderworpen zijn. Al met al dus toch goed nieuws: in

heel veel gevallen is het aan historie verbonden extra programmeerwerk geheel onnodig en waar dat niet het geval is neemt de complexiteit van de programmatuur fors af. Party-time in de polder!

Waar blijven de tools?

De tweede vraag die we ons stelden had betrekking op het volledig ontbreken van generieke oplossingen voor het historieprobleem van de kant van de gevestigde makers van dbms-producten en frontend tools. In hoofdstuk 1 meldden we al dat er niets op de markt is en dat er ook niets lijkt aan te komen. Ware dat anders dan zouden eenvoudige applicatie-ontwikkelaars zoals ondergetekende en collega's niet de moeite hoeven nemen om het wiel zelf uit te vinden. Hoe is het mogelijk dat er niets gebeurt aan een probleem dat op zo veel plaatsen speelt? Een gemakzuchtig antwoord zou kunnen zijn dat de betreffende leveranciers niet weten wat de problemen zijn. Voor een deel is dit ongetwijfeld juist: de tool-verkopers die ik spreek geloven vaak orecht dat ze er met een rdbms-extensie voor 'time-series data' inderdaad zijn. Hoe jammer dat wat leuk is voor het opslaan van audio en video in een multimediatdatabase vrijwel nutteloos is voor administratieve toepassingen die verder gaan dan een kaartenbak.

Een meer diepgaand antwoord op de vraag waar de toolbouwers blijven heeft te maken met de reikwijdte van de eisen waaraan een generieke oplossing voor het historieprobleem moet voldoen. In hoofdstuk 4 bespraken we de vereisten voor het op transparante wijze onderbrengen van historie in een database. Wat we daar feitelijk deden was het beschrijven van een uitbreiding van het relationele model met tijdsaspecten, een eigen 'temporal relational model' zo u wilt. Aan inspanningen gericht op het bedenken van dergelijke modellen ontbreekt het bepaald niet in de wetenschappelijke informaticawereld. Uiteindelijk houdt het echter meestal op bij het beschrijven van de noodzakelijke logische gegevensstructuren en de bijbehorende query en manipulatietaal. Minstens zo belangrijk is echter dat historische configuraties — net zoals indexen — onzichtbaar voor de programmeur kunnen worden gewijzigd en dat de effecten daarvan op de functionele kant van het systeem correct worden doorgevoerd. Hoofdstuk 5 toont een mechanisme dat dit in praktisch alle gevallen mogelijk maakt. Het herstructureren van gegevensstructuren en integriteitsregels aan de ene kant en het herdefiniëren van de gebruikersinterface aan de andere kant leveren tezamen een praktische oplossing. Helaas ontbreekt dit besef veelal in de gefragmenteerde wetenschappelijke werelden en jammer genoeg staan de afdelingen "database" en "frontend tools" van de belangrijkste leveranciers al eveneens heel ver van elkaar af.

Een laatste reden waarom historiebestendige hulpmiddelen maar niet op de markt komen heeft te maken met het feit dat een generieke oplossing voor historie niet op zichzelf staat, maar moet worden ingebed in een veelheid van

Voorbij-tijd en werkelijkheid

andere slimme features die deels nog ontbreken in de frontend-hulpmiddelen van vandaag. Om een generiek historiemechanisme praktisch toepasbaar te maken is onder meer nodig om te beschikken over:

- Een generiek mechanisme om integriteitsregels af te vangen. Bij het veranderen van de historische constellatie van gegevens verschijnen, verdwijnen en veranderen allerlei integriteitsregels (zie hoofdstuk 3). Zonder een intelligent 'constraint enforcement-mechanisme' houdt dan alles op. Vrijwel geen enkel frontend-product beschikt over een dergelijk mechanisme (Usoft is de uitzondering). Gelukkig is een dergelijk mechanisme – dat ook los van historie goud waard is – in de meeste frontend-omgevingen eenvoudig zelf te bouwen.
- Een generiek mechanisme om presentatie-aspecten vorm te geven. Het historismechanisme genereert allerlei historietabellen met een in beginsel tijdelijk karakter. Op dergelijke 'ephemeral tables' mag nooit worden geprogrammeerd, maar de gebruiker moet wel kunnen angeven hoe de gegevens moeten worden getoond. Moderne tools beschikken over een uitstekend hulpmiddel om dit te realiseren, namelijk een window painter. Jammer dat je met zo'n tool ook kunt programmeren. De enige goede oplossing is om zelf maar een window painter te realiseren. Heel frustrerend en tamelijk kostbaar!
- Een generiek mechanisme om gegenereerde integriteitsregels te vertalen in intelligent gedrag. De pop-up schermen van het vorige hoofdstuk zijn feitelijk master/detail-constructies. De afdeling 'frontend-tools' van de doorsnee leverancier is er nog niet van op de hoogte dat rdbms-producten tegenwoordig kennis hebben van allerlei regels zoals foreign keys en domeinregels. In dergelijke tool-omgevingen moet daarom heel veel worden geprogrammeerd dat eenvoudig kan worden geparametiseerd. Een eenvoudig generiek master/detail-mechanisme voorkomt sowieso heel veel werk en is een onmisbare voorwaarde voor het ontwikkelen van de hands-free gebruikersinterface zoals die is beschreven in het vorige hoofdstuk.

Features als deze liggen aan de basis van elke generieke oplossing voor het historieprobleem. Ik zou de leveranciers van frontend-tool willen aanraden om een wat minder aandacht te besteden aan 'roze button'-aspecten en wat meer aandacht te besteden aan architecturale aspecten zoals de hierboven besproken. Hoe dan ook: tool-based oplossingen zijn ver, heel ver weg.

Bij ontstentenis van 'off the shelf'-oplossingen komt natuurlijk de vraag op of het praktisch is om zelf een oplossing voor het historieprobleem te realiseren en, zo ja, wat de rol van dit boek daarin kan zijn. In dat verband wil ik opmerken dat dit boek een tamelijk volledige blauwdruk beoogt te zijn voor zelfmade oplossingen. Daarbij moet echter worden benadrukt dat een blauwdruk iets heel anders is dan een implementatievoorschrift en dat er, zoals hierboven beschreven, nog enkele andere zaken spelen die moeten worden opgelost. Al met al is het zelf ontwikkelen van een generieke oplossing voor historie alleen de moeite waard wanneer er sprake is van omvangrijke toepass-

ingen met veel historie. In dergelijke omgevingen kan dit boek mogelijk dienen als leidraad of inspiratiebron, maar veel implementatie-aspecten zijn hier niet beschreven. Wie verantwoordelijk is voor de realisatie van grote historische systemen, beschikt over een goed rdbms en moderne 4GL-hulpmiddelen (of C++ of Java), de hier uiteengezette theorie kan volgen en de screendumps in de hoofdstukken 4 en 5 kan lezen is van harte uitgenodigd om aan de slag te gaan!

Logging van gebruikers en programma's

Net als bij de Tien Geboden-serie hebben sommige artikelen — bij eerste publicatie in DB/M — aanleiding gegeven om ondergetekende te benaderen met vragen en suggesties. Die vragen en suggesties waren van onschabare waarde om te beoordelen of de bedoelde boodschap voldoende duidelijk is en of het betoog aansluit bij de praktijk zoals die door de lezers wordt ervaren. Zoals gezegd is er geen reden tot klagen, op één opvallende uitzondering na. Ik doel op het wijdverbreide gebruik om bij historische tabellen — met name mutatielogging, ofwel transaction timetabellen — vast te leggen welke gebruiker de mutatie heeft doorgevoerd. Daarbij wordt dan soms ook nog vastgelegd met welk programma die mutatie is doorgevoerd. Mijn ervaring is dat velen dit beschouwen als een wezenlijk aspect van het vastleggen van historische gegevens. Het ontbreken van enige referentie in deze serie naar deze functionaliteit is daarom nogal verbazingwekkend, maar kan even goed worden uitgelegd als bewijs van het hoge theoretische niveau van de lezers. De mogelijkheid om vast te leggen welke gebruiker met welk programma een mutatie heeft doorgevoerd heeft immers niets te maken met de historieproblematiek. Ook voor tabellen zonder historie is het namelijk denkbaar om het 'wie' en 'waarmee' te registreren.

Dat kan eenvoudigweg gebeuren door twee attributen toe te voegen aan een historieloze tabel. Natuurlijk registreren we dan alleen maar de laatste mutatie. Vaak zal dat niet genoeg zijn en is het gewenst om alle mutaties te registreren. In die gevallen zal het ook de bedoeling zijn om te registreren wat de gebruiker heeft uitgevoerd en dat vereist natuurlijk de vastlegging van het historische verloop. Kortom, de behoefte aan vastlegging van gebruikers- en programma-identificaties bij mutaties heeft niets te maken met de historieproblematiek, maar komt wel vaak voor in combinatie met die historieproblematiek. Dit betekent dat er behoefte is aan een generiek mechanisme om aan een tabel, al dan niet gegenereerd door het historiemechanisme, een gebruikers- en/of een programma-identificatie toe te voegen. Figuur 18 laat zien hoe dat uitpakt op een tabel Medewerker zonder historie en op een tabel waarvan het salaris historisch is gemaakt.

We zien dat in de tabel zonder historie kan worden vastgelegd door wie en met welk programma de laatste mutatie is doorgevoerd.

Voorbij, tijd en werkelijkheid

Geen historie

Medewerker(Med#, Naam, Sekse, ... Salaris, Afd#, User_id, Prog_id)

Salaris en sekse historisch

Medewerker(Med#, Naam, Sekse, ...Afd#, User_id)

Moneyline(Med#, His_Dat_In, His_Ts_In, ..., Salaris, User_id)

SekseHis(Med#, His_Ts_In, ..., Sekse, User_id, Prog_id)

Figuur 18. Gebruikers- en programmalogging onafhankelijk van historie.

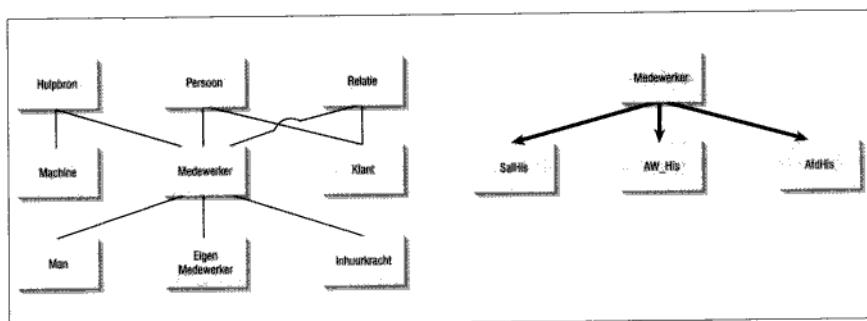
Na historisering is blijkbaar besloten dat het vastleggen van de gebruikers-identificatie afdoende is voor de niet gehistoriseerde gegevens, dat we voor de terugwerkend historisch vastgelegde rubriek Salaris per mutatie hetzelfde willen weten en dat we voor de separaat gelogde rubriek SekseHis zowel de gebruikers- als de programmareferentie willen vastleggen.

De historie voorbij: versioning en generalisatie

'Het wordt tijd om daadwerkelijk af te ronden. Dat kan echter pas met een bevredigd gevoel gebeuren nadat er enige aandacht is besteed aan de visie die al in het eerste hoofdstuk werd ontvouwd. Die visie houdt in dat het historieprobleem een beperkte uiting (een speciaal geval) is van een algemeen versieprobleem. Een andere uiting van dit versieprobleem is het eveneens besproken generalisatieprobleem. Bij het historieprobleem is er tegelijk sprake van objecten (bijvoorbeeld medewerker X) en van tijdgebonden versies van objecten (bijvoorbeeld de medewerker X van gisteren). Bij het generalisatieprobleem is er tegelijk sprake van generieke objecten (bijvoorbeeld persoon X) en subklassen van objecten (bijvoorbeeld medewerker X, huurder X, klant X, debiteur X) waarvan de eigenschappen deels overlappen en deels verschillen.

Het idee is dat generalisatie en historie twee symptomen zijn van eenzelfde achterliggende kwaal. Die kwaal is het versieprobleem, ofwel de onmogelijkheid om elk object van een verzameling objecten onder te verdelen in een eindige set niet-overlappende objectklassen. Figuur 19 toont twee overzichtelijke uitingen van de fuzzy wereld waarin we leven en werken. De modellen op basis waarvan we informatiesystemen ontwerpen en bouwen voorzien niet in ondersteuning voor deze veelvuldig voorkomende uitingen van het versieprobleem. Het resultaat is kostbare systeemontwikkeling en complex en traag onderhoud.

Als deze visie op de verwantschap tussen generalisatie en historie juist is, dan ligt het voor de hand dat het generalisatieprobleem op een gelijksoortige wijze moet worden opgelost als het historieprobleem. Ik raak er meer en meer van overtuigd dat dit inderdaad het geval is. Klopt dit intuïtieve gevoel



Figuur 19. Generalisatie en Historie, two of a kind?

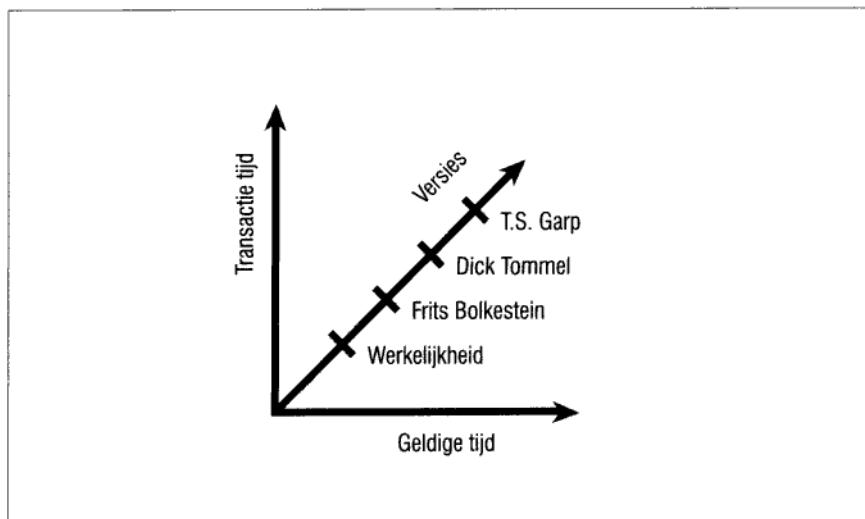
(meer is het niet) dan moet ik bekennen dat de ideeën over generalisatie die ik heb uitgewerkt in Tien Geboden voor goed database-ontwerp [15] (de implosie-benadering) misschien heel aardig en praktisch zijn, maar uiteindelijk geen uitzicht bieden op een werkelijk generieke oplossing voor het generalisatieprobleem. Voor wie zich het betreffende artikel herinnert of de daar uitgewerkte ideeën zelfs in de praktijk toepast wil ik dat toch even gemeld hebben. Breekt het inzicht door dan hoop ik nog eens op de generalisatieproblematiek terug te komen.

De historie voorbij: versioning in onversneden vorm

Naast het generalisatieprobleem lijkt de hier besproken aanpak van het historieprobleem ook perspectief te bieden voor het omgaan met het algemene versieprobleem als zodanig. De basis daarvoor is gelegd bij de bespreking van het fenomeen van terugwerkende kracht-historie in hoofdstuk 1 (zie figuur 1, rechtsonder). Een systeem dat de vragen kan beantwoorden als "Wat verdiende medewerker 'M1' op 1 januari 1998 volgens de informatie die het systeem bezat op 1 maart 1998 ?" kunnen hele interessante dingen worden gedaan in de what-if-sfeer. Stel dat we willen weten wat het effect is van een salarisverhoging met 10 procent op het netto-inkomen, dan is het mogelijk om die verhoging door te voeren, het bruto/netto-programma te draaien en vervolgens een tegengestelde mutatie door te voeren voor hetzelfde moment in de geldige tijd waarvoor de salarisverhoging werd doorgevoerd. Een systeem dat tweedimensionaal historisch is, is in potentie een fantastisch hulpmiddel voor het doorrekenen van scenario's, gebruiker-afhankelijke visies, offertes enzovoorts. Natuurlijk is het gebruik van een productiesysteem op zo'n wijze doorgaans onacceptabel: niets vermoedende gebruikers kunnen worden geconfronteerd met tijdelijk foutieve gegevens. En zou tijdens de what-if-analyse de betaalrun lopen dan zou medewerker 'M1' een aangenaam hoog bedrag krijgen overgemaakt, om natuurlijk één maand later met een inhouding te worden geconfronteerd. En dus blijven we van onze productiesystemen af en geven we waar nodig geld uit aan aparte sys-

Voorbij tijd en werkelijkheid

temen voor analyse-doeleinden, scenario's en offertes. Stel je voor dat het mogelijk zou zijn om een systeem – elk systeem! – met een druk op de knop om te zetten in what-if-modus en de door argeloze programmeurs ontwikkelde programma's – alle programma's! – te laten werken op een combinatie van werkelijke gegevens en door de gebruiker zelf geversioneerde gegevens. Wat we nodig hebben is een derde versie-dimensie naast – en onafhankelijk van – de twee historiedimensies. Figuur 20 toont zo'n situatie.



Figuur 20. Twee tijdsdimensies en een versiedimensie.

Figuur 20 toont een systeem waarin aan gegevenselementen niet alleen kan worden toegevoegd op welke tijdstippen ze betrekking hebben, maar eveneens volgens wiens visie ze geldig zijn. In een dergelijk systeem kunnen de gebruikers Frits, Dick en Garp mutaties doorvoeren voor een nul-versie: de werkelijkheid.

Zolang dat gebeurt is er sprake van een gewoon, historisch bewust systeem. Frits, Dick en Garp kunnen echter ook op de versie-button drukken en kunnen dan, onzichtbaar voor de anderen, hun eigen mutaties doorvoeren. Voor zover ze dat niet doen gaat het systeem uit van de werkelijkheid. Als Frits aan medewerker 'M1' in what-if-modus een salarisverhoging toekent dan gaat het systeem er van uit dat alle andere gegevens die noodzakelijk zijn voor een bruto/netto-berekening conform de werkelijkheid zijn. Het gevolg is een systeem dat zonder extra kosten van systeembouw en dataconversie tegelijk als productie en als scenario-systeem kan fungeren. Dergelijke systemen bestaan nog niet en dus zien we voortdurend dat kennis die is opgeslagen in data niet wordt benut.

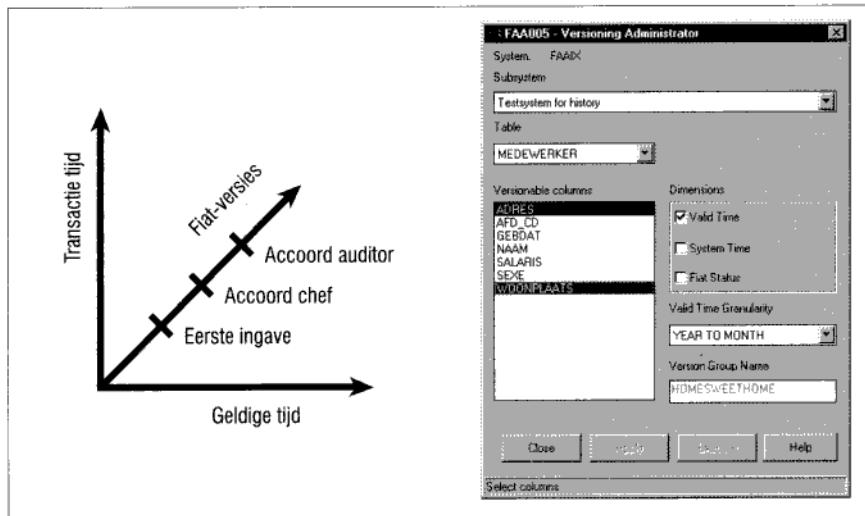
Uitgaande van de vorige twee hoofdstukken kan men zich gemakkelijk voorstellen hoe een versiedimensie zou kunnen worden geïmplementeerd: een-



voudigweg door het genereren van een versie-identificatie — in het geval van figuur 20 een gebruikerscode — bij de te versioneren rubrieken (zie hoofdstuk 4) en het opnemen van een versieselectie-button in alle schermen (zie hoofdstuk 5). Frits zou dan achter het scherm kunnen werken met de visie op de werkelijkheid van Dick, eventueel Dick's visie op het verleden. De drie dimensies van figuur 5 zijn dus weer vrij combineerbaar (orthogonaal). De meta-DBA (hoofdstuk 4) zou zelfs kunnen besluiten om voor bepaalde gegevens alleen een versiedimensie toe te voegen en de historiedimensies te laten zitten. Misschien zou de echte Dick dat maar wat graag willen. Het historiemechanisme van hoofdstuk 4 bood hem vier mogelijkheden; nu heeft hij de keuze uit acht mogelijkheden. Figuur 20 toont slechts de meest extreme situatie tot nu toe.

Een stapje terug: flattering

In de inleiding van dit hoofdstuk ben ik terughoudend geweest over de status van datgene wat ik te melden heb over de wereld voorbij historie. De opmerkingen die hiervoor bij het generalisatieprobleem werden geplaatst onderbouwen dat. Ook bij versie-ondersteuning conform figuur 20 past die bescheidenheid. De werkelijkheid is veel ingewikkelder dan figuur 20 suggerereert. Er is namelijk één wezenlijk verschil tussen de tijdsdimensies en de versiedimensies.



Figuur 21. Flattering als zwakkere vorm van versioning.

De historiedimensies bezitten een duidelijke ordening (van verleden naar toekomst), terwijl de versiedimensie die niet heeft. De ordening van de ver-

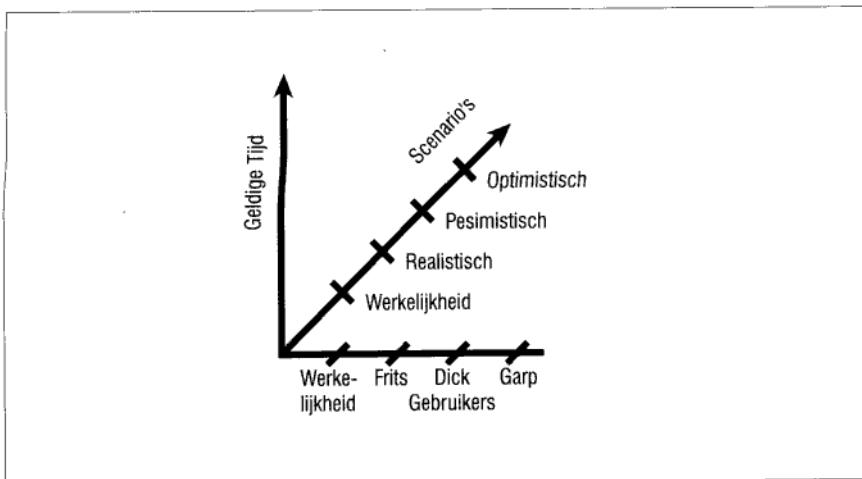
Voorbij tijd en werkelijkheid

sie-as in figuur 20 zegt niets over mijn politieke of literaire voorkeuren. Dit gebrek aan volgordelijkheid levert bij implementatie allerlei problemen op, voornamelijk op performance-gebied. Om die reden beperken wij ons vooralsnog tot de ondersteuning van een zwakkere variant waarbij er wel sprake is van een volgorde van versies. Figuur 21 toont zo'n situatie en herhaalt nog even kort de Versioning Administrator van hoofdstuk 4. Hopelijk wordt de betekenis van de Fiat Status-checkbox nu duidelijk.

Een sprong vooruit: n-dimensionale versie-ondersteuning

Er is nog een ander, meer fundamenteel probleem met de versioning-oplossing van figuur 19. Dat probleem is dat er potentieel niet sprake is van één versiedimensie, maar van meer dimensies naast elkaar. Figuur 22 toont een voorbeeld van zo'n fantastisch systeem waarin gebruikers vrijelijk scenario's kunnen maken op basis van historisch vastgelegde gegevens. Bij gebrek aan mogelijkheden om meer dimensies af te beelden is de transactietijddimensie weggelaten, maar het zal duidelijk zijn dat het aantal dimensies in beginsel onbeperkt is en dat alle dimensies naar believen met elkaar kunnen worden gecombineerd.

Het meest vergaande perspectief voorbij historie is wellicht niet voor iedereen aanlokkelijk. Ik moet zelf denken aan een reeds aangehaalde SF-boek van Isaac Asimov waar hij in relatie tot de consequenties van tijdreizen spreekt over "a ghostly kind of never-never land where the might-have-beens play with the ifs". Als dat het logische eindpunt is van de gedachtelin die in deze serie is uitgewerkt, dan moet dat maar zo zijn. Wij mensen kunnen alleen onze modellen van de werkelijkheid veranderen, niet die werkelijkheid zelf.



Figuur 22. De ultieme versie-database?

Implementatie van een generieke oplossing

Frido van Orden

In de eerste zes hoofdstukken van dit boek wordt toegewerkt naar de climax in de vorm van een generieke oplossing voor de historie-problematiek. Deze besprekking heeft doelbewust het karakter van een blauwdruk. In dit hoofdstuk zal een laag dieper op de implementatie-aspecten worden ingegaan.

Wie een poging waagt om een generiek historiemechanisme te realiseren moet beschikken over een flinke dosis inventiviteit en incasseringvermogen. Dat geldt natuurlijk voor wel meer ambitieuze automatiseringsprojecten, maar hier ligt de zaak op een aantal punten toch anders. Zo is een belangrijke factor de afhankelijkheid van de gebruikte tools. Dit hoeft natuurlijk geen verbazing te wekken: dbms'en en frontend tools zijn gemaakt om applicaties mee te bouwen, niet om de meest duivelse datomanipulatiekunsten mee uit te halen. Tot dat laatste zijn we echter wel gedoemd: de HTC die als basis voor de generieke oplossing dient, dwingt ons alles en iedereen voor de gek te houden en onder de motorkap iets heel anders te doen dan de argeloze gebruiker of tool denkt. Sommige tools lenen zich hiertoe meer of minder. In ieder geval onstaat gedurende de ontwikkeling een gestaag groeiend lijstje van wensen die men soms vervuld ziet bij een concurrerend product, maar waarvan we vaker nog slechts kunnen dromen.

Indien we in dit hoofdstuk op alle aspecten van de implementatie in zouden gaan zou deze bundel vele malen zo dik zijn geworden. Daarom concentreren we ons op drie daarvan:

- het herschrijven van SQL-commando's
- het toevoegen van tijdparameters
- het verwerken van DML.

Met een besprekking van deze drie zaken hebt u weliswaar geen volledige *Do it yourself*-handleiding in handen, maar krijgt u wel een indruk van de omvang en complexiteit van de technische aspecten.

SQL: gezichtsbedrog

Laten we eens met iets eenvoudigs beginnen: selecties op de gegenereerde historische tabelstructuur. Als voorbeeld zullen we weer de afdeling/medewerker database gebruiken die als case is gehanteerd in hoofdstuk 4 en 5. Het gegenereerde historische databaseschema ziet er uit als in figuur 6 rechtsonder (zie pagina 57).

We zullen een kleine maar belangrijke toevoeging op het schema maken: de gegenereerde historie-tabellen worden uitgebreid met (redundante) attributen die het einde van een tijdsinterval aanduiden: His_Dat_End voor het einde van de geldige tijd en His_Ts_End voor het einde van de transactietijd (uiteraard alleen voor zover van toepassing). Het voordeel hiervan is dat de selecties om het record te vinden dat de gegevens over een bepaald punt op een tijdas weergeeft (bijvoorbeeld geldige tijd '1 Januari 1998') een stuk eenvoudiger worden.

Wie nu een eenvoudige vraag als "Geef de naam en salaris van medewerker M1 op 1 januari 2000 met de kennis van het systeem van nu" wil beantwoorden zal aan het dbms de volgende SQL moeten voorleggen:

```
SELECT Medewerker.Med#, Nomen_Est_Omen.Naam, Moneyline.Salaris
FROM Medewerker, Nomen_Est_Omen, Moneyline
WHERE Medewerker.Med# = 'M1'
AND Medewerker.Med# = Nomen_Est_Omen.Med#
AND Medewerker.Med# = Moneyline.Med#
AND Nomen_Est_Omen.His_Dat_In <= '1-JAN-2000'
AND Nomen_Est_Omen.his_Dat_End > '1-JAN-2000'
AND Moneyline.His_Dat_In <= '1-JAN-2000'
AND Moneyline.his_Dat_End > '1-JAN-2000'
AND Moneyline.His_Ts_In <= NOW()
AND Moneyline.His_Ts_End > NOW()
```

Wie de theorie er nog eens op naleest moet concluderen dat de een na laatste conditie van dit SQL statement (His_Ts_In <= NOW()) overbodig is en de laatste (His_Ts_End > NOW()) altijd FALSE oplevert. De reden dat we het statement toch zo formuleren wordt duidelijk als we de vraag niet voor transactietijdstip nu maar voor gisteren middernacht stellen. Afgezien van de parameterwaarden hoeven we niets aan het statement te veranderen. De redenering dat de restrictie op His_Ts_End foutief is, is een drogredeneering: dit attribuut geeft aan wanneer de 'volgende' mutatie op deze tijdlijn' plaatsvindt. Voor de laatste mutatie moeten we hier dus eigenlijk een NULL-waarde invullen. In plaats daarvan kan ook voor de grootst mogelijke door het dbms toegestane waarde worden gekozen en de reden waarom dit onze voorkeur verdient moge na het bovenstaande duidelijk zijn.

Implementatie van een generieke oplossing

Merk op dat vragen over transactietijden in de toekomst automatisch leiden tot hetzelfde antwoord als bij transactietijd nu.

De HTC heeft tot gevolg dat nergens geprogrammeerd mag worden op de historische tabellen. Programmeurs, data-access-objecten en end-user query-gebruikers moeten dus gewoon het volgende statement kunnen uitvoeren:

```
SELECT Med#, Naam, Salaris  
FROM Medewerker  
WHERE Med# = 'M1'
```

Elke vorm van tijdsinformatie ontbreekt in deze query. Dit zullen we verderop rechtdoen, maar eerst concentreren we ons op herschrijven van de query.

Allereerst moet de SQL die wordt verstuurd worden afgevangen alvorens die door het dbms wordt verwerkt. Hiervoor zijn de volgende oplossingen mogelijk:

- Zorg ervoor dat elk SQL-statement, voor het naar de database wordt gestuurd, wordt geanalyseerd en eventueel herschreven. Dit noemen we de parserbenadering.
- Genereer views die een medewerker-timeslice selecteren en herschrijf het FROM-deel van elk SELECT-statement zodat van deze views gebruik wordt gemaakt. Dit noemen we de viewbenadering.
- Zorg ervoor dat het database-object Medewerker betrekking heeft op *getimeslicede* medewerkers. Dit noemen we de synoniembenadering.

De parser- en viewbenadering vereisen dat alle SQL-calls naar de database worden onderschept. Dit is in de meeste moderne objectgeoriënteerde ontwikkelomgevingen eenvoudig te realiseren omdat gebruik wordt gemaakt van statement-objecten. Lastiger wordt het al met 3GL's en embedded-SQL, maar het schrijven van een preprocessor die de embedded SQL detecteert en herschrijft is meestal nog wel mogelijk. Nog vervelender wordt het in omgevingen als Oracle Developer/2000, waar men wordt gedwongen tot middleware-achtige oplossingen en een grote hoeveelheid low-level programmeerwerk in het verschiet ligt.

De synoniembenadering heeft als charme dat de problematiek verplaatst wordt naar het dbms. Nadeel is echter dat het voor de DBA al heel gauw een enorme klus wordt om nog een goed overzicht te houden. Bovendien vereist deze oplossing de aanwezigheid van een groot aantal dbms-faciliteiten als het hernoemen van tabellen, inserts op multi-table views een dergelijke.

Van de drie besproken varianten had in eerste instantie de parserbenadering onze voorkeur. Een SQL-parser is voor meerdere zaken dan alleen historieoplossingen een handig instrument. Bovendien hoeven qua performance

nauwelijks concessies te worden gedaan. Het herschrijven van een historisch SQL-statement is een eenvoudiger taak dan het opstellen van een query executieplan en dat laatste doet elk beetje rdbms fluitend. Wie zijn statements zoveel mogelijk hergebruikt en dus maar één keer *prepared* (sowieso een goed programmeerprincipe) verzacht de pijn nog meer. Het belangrijkste prestatievoordeel van de parserbenadering is echter dat een statement op maat kan worden herschreven. Zo komt in de herschreven query de tabel HomeSweetHome niet voor, wat de executie-overhead tot een minimum beperkt.

De viewbenadering ruilt ten opzichte van de parserbenadering prestaties in voor geld. Hoewel technisch niet de meest optimale heeft deze oplossing het voordeel dat het schrijven van een dure SQL-parser niet meer nodig is en daar een aantal wezenlijk eenvoudiger zaken voor terugkeren. Uiteindelijk is voor deze oplossing gekozen in de implementatie van de generieke historie-oplossing in de FAA-toolomgeving.

Het idee is om een view te maken die, in ons geval, medewerker-timeslices representeert en er als volgt uitzet:

```
CREATE VIEW Medewerker_TimeSlice AS
SELECT Medewerker.Med#, Nomen_Est_Omen.Naam, HomeSweetHome.Adres,
HomeSweetHome.Woonplaats, Medewerker.Seks, Medewerker.GebDat,
NineToFive.Afd#, Moneyline.Salaris, Session_Dim_Valx
FROM Medewerker, Nomen_Est_Omen, HomeSweetHome, NineToFive, Moneyline
WHERE Medewerker.Med# = Nomen_Est_Omen.Med#
AND Medewerker.Med# = HomeSweetHome.Med#
AND Medewerker.Med# = NineToFive.Med#
AND Medewerker.Med# = Moneyline.Med#
AND Nomen_Est_Omen.His_Dat_In <= Session_Dim_Valx.His_Dat_In
AND Nomen_Est_Omen.His_Dat_End > Session_Dim_Valx.His_Dat_In
AND HomeSweetHome.His_Dat_In <= Session_Dim_Valx.His_Dat_In
AND HomeSweetHome.His_Dat_End > Session_Dim_Valx.His_Dat_In
AND NineToFive.His_Dat_In <= Session_Dim_Valx.His_Dat_In
AND NineToFive.His_Dat_End > Session_Dim_Valx.His_Dat_In
AND Moneyline.His_Dat_In <= Session_Dim_Valx.His_Dat_In
AND Moneyline.His_Dat_End > Session_Dim_Valx.His_Dat_In
AND Moneyline.His_Ts_In <= Session_Dim_Valx.His_Ts_In
AND Moneyline.His_Ts_End > Session_Dim_Valx.His_Ts_In
AND Session_Dim_Valx.Session_ID = SESSIONID()
```

De diepere betekenis van de tabel Session_Dim_Valx wordt verderop duidelijk. Voor dit moment is het voldoende om te weten dat deze tabel voor elke actieve database-sessie een record bevat en dat op het moment van executie van de query de rubriek His_Dat_In in het record behorende bij de huidige sessie het geldige-tijd-tijdstip bevat waarin we geïnteresseerd zijn en de rubriek His_Ts_In het transactietijdstip waarin we geïnteresseerd zijn. De

Implementatie van een generieke oplossing

functie SESSIONID() in de laatste regel van de view is een dbms-functie die ons de huidige sessie-ID oplevert.

De voorbeeldquery kunnen we nu als volgt herschrijven:

```
SELECT Med#, Naam, Salaris  
FROM Medewerker_TimeTypeSlice Medewerker  
WHERE Med# = 'M1'
```

Voorwaar een heel wat eenvoudiger klus dan het volledig overhoop gooien van het statement zoals dat in de parserbenadering gebeurde. Merk op dat we door slim gebruik te maken van aliases in het FROM-deel van het statement nooit het SELECT-, WHERE-, ORDER- of wat voor ander deel van het statement dan ook hoeven te herschrijven.

De performance van de viewbenadering is wat minder dan die van de parserbenadering omdat bij statements die niet alle historische groepen raken een aantal loze joins moet worden uitgevoerd. Deze joins zijn niet weg te optimaliseren door de dbms-optimizer, omdat het al dan niet aanwezig zijn van records in de extra meegejoinde historische groepstabellen van invloed kan zijn op het joinresultaat. De regels die relationele concepten vertalen naar historische varianten (zie hoofdstuk 3) zorgen er weliswaar voor dat dit in de praktijk nooit het geval is, maar van deze regels heeft de argeloze optimizer geen weet.

De viewbenadering is alleen toepasbaar indien alle query's die we formuleren op één punt in het tijdvak betrekking hebben (zie hoofdstuk 6). Willen we vragen beantwoorden als "geef het huidige salaris van alle medewerkers die gisteren op afdeling A1 werkten", dan is de parserbenadering de enig mogelijk oplossing.

De tijd neerzetten

De HTC leidt ertoe dat verwijzingen naar tijddattributen uit onze SQL verbannen wordt. We zullen dus een andere manier moeten bedenken om tijdinformatie aan SQL-opdrachten mee te geven. Ook hier zijn weer meerdere oplossingen mogelijk:

- via uitbreidingen op de SQL-syntaxis
- via extra parameters die meegegeven worden aan de SQL-statement call.

Het moge duidelijk zijn dat uitbreidingen op de SQL-syntaxis alleen mogelijk zijn in combinatie met de parserbenadering. Dit duo levert zonder meer de best leesbare programmatuur en localiseert bovendien de historie-aspec-

ten op een punt: de SQL-parser. Stephen Cannan wijdt in dit boek twee hoofdstukken aan een mogelijk historisch bewuste SQL-syntaxis.

De tweede benadering zou er in een objectgeoriënteerde taal als volgt uit kunnen zien:

```
stmt = new sqlStmt();
stmt.prepare("SELECT Med#, Naam, Salaris FROM Medewerker WHERE
    Med# = 'M1'");
stmt.setValidTime('1-JAN-2000');
stmt.setTransactionTime(NOW());
stmt.execute();
row = stmt.fetch();
```

De acties die de aanroepen van de functies setValidTime() en setTransactionTime() tot gevolg hebben zouden kunnen leiden tot een herschrijving van het statement, maar er is ook een andere oplossing mogelijk die met name in combinatie met de viewbenadering uitstekend functioneert. Het idee is om een stuurtabel met tijdsinstellingen in de database op te nemen. Als we ervan uitgaan dat per database-connectie slechts één statement tegelijk kan worden uitgevoerd (een programma kan wel meerdere connecties openen) dan is de sleutel van deze tabel de sessie-ID, die in de meeste dbms'en met een functie opvraagbaar is. Tevens nemen we attributen His_Dat_In en His_Ts_In op om selectie-tijdstippen te specificeren, alsmede His_Dat_End die we straks bij DML-acties nodig zullen hebben om een update-interval te specificeren. We kunnen de tijdparameters nu naar de database overbrengen door vlak voor uitvoering van een SQL-statement de juiste tijdparameterwaarden in de tabel in te vullen. Dit kost uiteraard snelheid, maar het effect zal beperkt zijn omdat de stuurtabel nooit erg groot zal worden (maximaal het aantal gelijktijdig openstaande connecties). Bovendien is het eenvoudig om te controleren of de tijdparameters sinds de laatste executie op de betreffende connectie ongewijzigd zijn gebleven en in dat geval de update over te slaan.

Het programmavoorbeeld hierboven voert dan de volgende SQL uit op de database:

```
UPDATE Session_Dim_Valx
SET His_Dat_In = '1-JAN-2000', His_Dat_End = NOW()
WHERE Session_ID = SESSIONID();

SELECT Med#, Naam, Salaris
FROM Medewerker_TimeSlice_Medewerker
WHERE Med# = 'M1'
```

Implementatie van een generieke oplossing

Het tandem stuurtafel-timeslice-view werkt in de praktijk uitstekend en is een goed alternatief voor wie terugschrikt voor het moeten schrijven van een SQL-parser.

The times they are a-changing

Is de zaak tot zover nog te overzien, als we ook gegevens willen gaan toevoegen, verwijderen en muteren wordt de zaak complexer. We zullen u hier niet vermoeien met eindeloze SQL-scripts om bijvoorbeeld een terugwerkende-kracht-update over een beperkt interval uit te voeren. Interessanter is het om aandacht te besteden aan de implementatieaspecten daaromheen.

De parserbenadering is uiteraard ook bij DML weer toepasbaar, al is het performancevoordeel hier niet evident. Enerzijds is het perfect mogelijk om bijvoorbeeld bulk-updates hun bulkkarakter te laten behouden en niet uiteen te rafelen tot record voor record updates. Anderzijds leidt het updaten van één record in een historische database al snel tot een vier- of vijftal insert-, update- en delete-acties op de historische tabellen. Om performance-redenen is het aan te bevelen deze niet stuk voor stuk vanaf de client af te vuren maar binnen de server te laten geschieden, het liefst in gecompileerde vorm in stored procedures. Om het beste van beide te behouden dient de SQL-parser idealiter dus geïmplementeerd te worden in het dbms. Met de komst van de object-relationele database-servers met hun UDF's en UDT's (user defined functions en user defined data types) hoeft dit ideaal echter geen utopie meer te zijn.

In de door ons geïmplementeerde oplossing is gekozen om de processen die een historische (her)insert, update of delete van een record uitvoeren te implementeren in stored procedures. Om toch bulkacties op meer dan één record uit te voeren (vooral van belang in batchomgevingen) kan men meestal niet terugvallen op dbms-triggermechanismen, aangezien deze meestal niet de mogelijkheid bieden om een trigger uit te voeren *in plaats van* de triggerende actie. Dit is in historische databases echter wel van belang. Wie een historische delete uitvoert wil het record in de basistabel (bijvoorbeeld Medewerker) behouden. De truc om aan het einde van de trigger het originele record weer te inserten leidt tot recursieve problemen die wel oplosbaar zijn maar helaas door een aantal dbms'en, waaronder het door ons gebruikte Informix, worden verboden.

Een uitweg die wat krakkemikkig oogt maar in de praktijk wonder wel werkt wordt hieronder beschreven.

Voor elke historische tabel maken we een schaduwtafel die we verder trigger tabel zullen noemen. Deze tabel bevat alle rubrieken van de originele tabel, plus schaduwrubrieken voor elke rubriek.

In het voorbeeld ziet de schaduwtafel voor medewerker er als volgt uit:

```
MedewerkerTrigger(Med#, Med#_Sch, Naam, Naam_Sch, Adres,  
Adres_Sch, Woonplaats, Woonplaats_Sch, Sekse, Sekse_Sch, GebDat,  
GebDat_Sch, Afd#, Afd#_Sch, Salaris, Salaris_Sch)
```

Vervolgens dienen we er voor te zorgen dat alle DML-operaties worden herleid naar deze triggertabel. Hiervoor geldt dat voor elke rubriek de rubriek *zelf de nieuwe waarde bevat en zijn schaduwrubriek de oude waarde*.

```
INSERT INTO medewerker  
SELECT ...
```

wordt dus

```
INSERT INTO MedewerkerTrigger(Med#, Naam, Adres, Woonplaats,  
Sekse, GebDat, Afd#, Salaris)  
SELECT ...
```

Voor updates wordt

```
UPDATE Medewerker  
SET Salaris = Salaris * 1.1  
WHERE Salaris > 10000
```

herschreven tot

```
INSERT INTO MedewerkerTrigger(Med#, Med#_Sch, Naam, Naam_Sch,  
Adres, Adres_Sch, Woonplaats, Woonplaats_Sch, Sekse, Sekse_Sch,  
GebDat, GebDat_Sch, Afd#, Afd#_Sch, Salaris, Salaris_Sch)  
SELECT Med#, Med#, Naam, Naam, Adres, Adres, Woonplaats,  
Woonplaats, Sekse, Sekse, GebDat, GebDat, Afd#, Afd#,  
Salaris * 1.1, Salaris  
FROM Medewerker  
WHERE Salaris > 10000
```

en

Implementatie van een generieke oplossing

```
DELETE FROM Medewerker  
WHERE Salaris < 100;
```

tenslotte wordt

```
INSERT INTO MedewerkerTrigger(Med#_Sch)  
SELECT Med#  
FROM Medewerker  
WHERE Salaris < 100;
```

Het is nu niet moeilijk meer om geschikte triggers te definiëren die detecteren of inserts in de triggertabel een historische insert, update of delete representeren en voor elk record de geschikte stored procedure aan te roepen.

Om vollopen van de triggertabel te voorkomen wordt deze regelmatig geleegd, bijvoorbeeld na iedere transactie of ieder uur.

U ziet dat de SQL herschrijvingen hier minder eenvoudig zijn dan de selectie-herschrijvingen van de viewbenadering. Een volledige SQL-parser is echter niet nodig om de klus te klaren.

En nu aan de slag

Historische databases zijn en blijven complexe materie, en veel valkuilen en aandachtspunten die in de theorie ter sprake komen waren ons nog niet bekend toen we in 1996 aan de implementatie van een generieke historieoplossing begonnen. Wie echter de theorie goed in de vingers heeft, creatief is en de beschikking heeft over ruimdenkende tools, komt een heel eind. Mocht het toch niet lukken, bedenk dan dat er een bedrijf is dat een generieke oplossing productierijp heeft gekregen.

Count the clock

Stephen J. Cannan

Tijd en ruimte. De twee laatste grote bastions tegen de tientellers, de drommen decimaal georiënteerde database-beheerders, dominees, directeuren, dagloners, druivenplukkers, dozenschuivers, draglinemachinisten en andere wereldburgers die onze aarde bevolken. We hebben het over de mensen die alleen kunnen tellen als ze de vingers en duimen van allebei hun handen mogen gebruiken. Op het eerste gezicht lijkt het ietwat vreemd, maar merkwaardigerwijs ook bemoedigend, dat in een wereld die vaarwel heeft gezegd — of zegt — tegen de pond, het ons, de mijl en de kan, om nog maar te zwijgen van de el, de landmetersketting en de korenmaat, twee belangrijke meetsystemen met succes weerstand hebben weten te bieden tegen de oprukkende decimalisering.

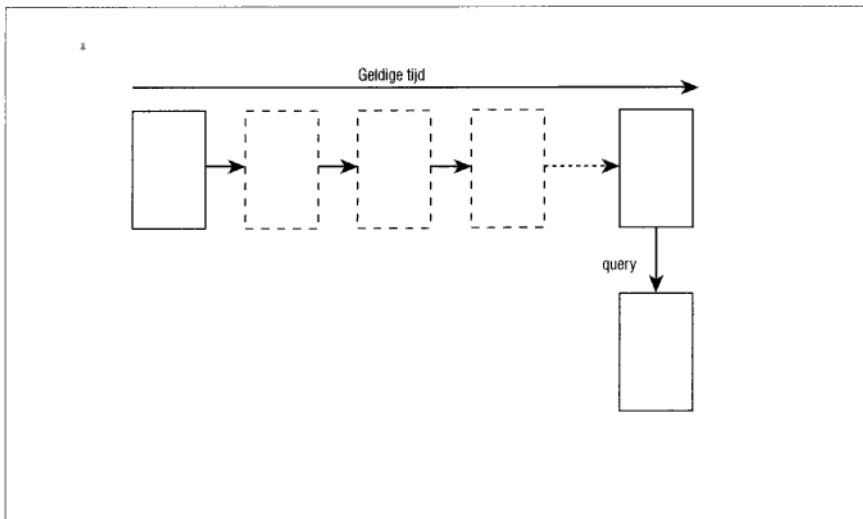
We hebben nog altijd te maken met graden en seconden. Bedenk daarbij dat een seconde is gedefinieerd als 9.192.631.770 perioden van de straling die wordt uitgezonden door een bepaalde overgang van het Cesium-133-atoom. Dus geen 10 met 10 nullen, maar exact 9.192.631.770. Hoewel ik er niet aan twijfel dat de rationaliseerders uiteindelijk de cirkel van 100 graden en de 10-uurs dag er door zullen weten te drukken, zal ik dat, godzijdank, in mijn huidige gedaante niet meer meemaken. Als ik een schatting moet doen, zal dat ongeveer samenvallen met het moment waarop per decreet wordt vastgesteld dat Pi gelijk is aan 3.

In dit en het volgende hoofdstuk zal ik het tijdsprobleem aan een hernieuwde beschouwing onderwerpen. In Database Magazine zijn al verschillende artikelen [zie 1, 2, 3 en 4] verschenen over verschillende aspecten van het begrip tijd en de manier waarop tijd in databases wordt weergegeven. Het gaat hier echter om een zo belangrijk en tegelijkertijd zo onderschat of zelfs genegeerd toepassingsgebied, dat het geen kwaad kan om ons andermaal op dit onderwerp te storten. Zeker niet op een moment waarop de International Organization for Standardization (ISO) bezig is om alternatieve voorstellen [zie 5, 6 en 7] te bekijken om de SQL-standaard uit te breiden met syntacti-

2.3. De snapshot-database

sche en semantische voorzieningen die het definiëren en toepassen van temporale gegevens moeten vergemakkelijken. Merk op dat de tijdsproblemen niets te maken hebben met het feit dat we tijd in een sexagesimaal of zestig-tallig stelsel weergeven, en dat decimalisatie hier dus geen oplossing biedt.

De meeste databases bevatten in de tijd variërende informatie (of zouden dat althans moeten doen). Het is moeilijk een voorstelling te maken van reëel bestaande entiteiten die geen verband houden met tijd, aangezien alle fysieke objecten — en zeker alle levende wezens — een begin en een eind hebben. Zoals vele succesvolle applicaties hebben laten zien, is SQL geschikt als mogelijk vehikel voor het implementeren van databases met tijd-variante gegevens. Het feit dat in een groot aantal databases ondersteuning voor zulke gegevens beperkt of ontoereikend is of zelfs ontbreekt, doet echter vermoeden dat SQL in zijn huidige gedaante op dit terrein weinig ondersteuning biedt. Deze bewering kan worden gestaafd door een aantal eenvoudige statements te bekijken en te proberen om binnen SQL een passende oplossing te vinden.



Figuur 23. De snapshot-database.

Stel dat we informatie willen vastleggen over werknemers en hun managers. Conventionele databases werken doorgaans volgens het principe van wat we een bewegende momentopname zouden kunnen noemen, een 'moving snapshot' zoals dat in figuur 23 wordt geïllustreerd (we zullen het verder over 'snapshot' hebben). In deze figuur wordt een conventionele tabel weergegeven door een rechthoek. De huidige toestand van de tabel is de rechthoek in de rechter bovenhoek van figuur 1. Telkens wanneer er een wijziging wordt aangebracht in deze tabel, wordt de vorige toestand weggegooid, zodat op

Count the clock

willekeurig welk moment alleen de actuele toestand beschikbaar is. De weggegooid, voorafgaande toestanden worden weergegeven door gestippelde rechthoeken; de naar rechts wijzende pijlen vertegenwoordigen de wijzigingsactie die de tabel van de ene toestand in de andere heeft gebracht. Dat wil zeggen: als de toestand van de database verandert, blijft alleen de laatste instantie bewaard, zodat alle queries op de tabel alleen op die meest recente instantie inwerken.

In een dergelijke snapshot-database kan men de volgende tabel aantreffen:

```
CREATE TABLE Werknemer (
    WnrNr      NUMERIC(5),
    Naam       CHARACTER(40),
    PositieCd  NUMERIC(3),
    Afd        CHARACTER(40),
    Manager    NUMERIC(5),
    PRIMARY KEY (WnrNr),
    FOREIGN KEY (Manager) REFERENCES Werknemer
);
```

Hoe we dit moeten converteren om ook de historie van een werknemer in de database te kunnen opnemen, is niet onmiddellijk duidelijk. Men kan overwegen een extra kolom op te nemen die de periode aangeeft gedurende welke de informatie geldig was, bijvoorbeeld:

```
Geldig PERIOD(DATE),
```

Deze kolom zou echter in de primaire sleutel moeten worden opgenomen. Om te voorkomen dat twee rijen voor overlappende perioden informatie over dezelfde werknemer bevatten en om er voor te zorgen dat de perioden voor een werknemer een aaneengesloten geheel vormen, zouden we een aantal behoorlijk gecompliceerde beperkingen moeten formuleren. Voor voorbeelden van dit soort beperkingen, zie hoofdstuk 23 van Handboek SQL [8].

Als de primaire sleutel op deze manier is veranderd zou de referentiële beperking op de Manager-kolom niet langer zijn toegestaan. We hebben een bevestigings- of een check-clausule nodig om het probleem te lijf te gaan, bijvoorbeeld:

```
CHECK (Manager IN
        (SELECT WnrNr FROM Werknemer),
```

Dit gaat echter voorbij aan het in de tijd variërende karakter van de inhoud van de werknemer-tabel. Op deze manier zouden we in de database bijvoorbeeld een manager kunnen hebben die het bedrijf al had verlaten voordat de onder hem vallende werknemer in dienst trad, of omgekeerd een werknemer kunnen hebben die valt onder een manager die nog niet in dienst is. Dit soort relaties tussen door ruimte en tijd gescheiden mensen doet denken aan de avonturen van Captain Picard in de laatste aflevering van Star Trek - The Next Generation. Voor dergelijke avonturen is een relationeel dbms, althans in mijn ogen, niet het aangewezen decor.

Het is mogelijk een beperking te bedenken die er voor zorgt dat werknemers alleen kunnen worden geleid door managers die in diezelfde periode bij dezelfde organisatie werkzaam zijn. Toch zou het me verbazen als iemand er in zou slagen in minder dan vijf minuten een dergelijke beperking te schrijven en zeker te weten dat de beperking ook inderdaad juist is. Hetzelfde probleem doet zich voor bij niet naar zichzelf verwijzende tabelbeperkingen. Als we uitgaan van de volgende salaristabel:

```
CREATE TABLE Salaris (
    WnrNr      NUMERIC(5),
    Salaris     NUMERIC(6),
    Geldig     PERIOD(DATE),
    PRIMARY KEY (WnrNr),
    FOREIGN KEY (WnrNr) REFERENCES werknemer
);
```

hebben we opnieuw het probleem dat we niet in staat zijn een geldige referentiële beperking te definiëren.

Als we naar de query's kijken in plaats van naar de structuur van de database, komen we soortgelijke problemen tegen. Neem de query "Geef me alle werknemers die geen manager zijn". In de snapshot-database is deze query eenvoudig:

```
SELECT WnrNr
  FROM Werknemer
 WHERE WnrNr NOT IN
       (SELECT Manager FROM Werknemer);
```

Om deze query zodanig te converteren dat hij in de database met de Geldig-kolom werkt, is iets ingewikkelder.

Count the clock

Hij zou er als volgt uit kunnen zien:

```
SELECT WnrNr
      FROM Werknemer
     WHERE Geldig CONTAINS CURRENT_DATE
       AND
          WnrNr NOT IN
            (SELECT Manager
              FROM Werknemer
             WHERE Geldig CONTAINS CURRENT_DATE);
```

hetgeen wil zeggen dat we er voor moeten zorgen dat we alleen de huidige werknemers en de huidige managementverhoudingen selecteren. Nu we eenmaal de moeite hebben genomen om de historische informatie vast te leggen, willen we natuurlijk informatie hebben over die werknemers die geen manager zijn of waren, gekoppeld aan informatie over de perioden waarin dit het geval was. Dit is echter zo gecompliceerd dat ik het bij wijze van vingeroefening graag aan de lezer overlaat.

Terwijl uit bovenstaand voorbeeld kan worden afgeleid dat het mogelijk is een database te definiëren en te manipuleren die met tijdafhankelijke gegevens kan omgaan, is zoiets duidelijk geen recht-toe-recht-aan klusje. Omdat het dus duidelijk niet van een een-twee-drie-klaar-is-Kees eenvoud is, zien we in veel gevallen dat het probleem ofwel niet op de juiste manier wordt aangepakt, of resulteert in een applicatie die even kostbaar als complex is en regelmatig struikelt. Waar we duidelijk behoeft te hebben, is een meer rechtstreekse ondersteuning voor het concept tijd in de SQL-taal (en dus, mogen we hopen, uiteindelijk in de implementaties van leveranciers). Dit zou ontwerpers en programmeurs helpen zich te concentreren op de logica van de applicatie die zij onderhanden hebben, in plaats van zich het hoofd te moeten breken over de technische problemen van het met de beschikbare hulpmiddelen implementeren van het concept tijd.

Academisch onderzoek

Het gebied van temporele databases is al sinds jaar en dag het onderwerp van academisch onderzoek. In juni 1993 namen vijfenveertig experts deel aan de ARPA/NSF International Workshop on an Infrastructure of Temporal Databases. Vanuit deze workshop werd het initiatief genomen tot wat het TSQL2 Language Design Committee wordt genoemd. Deze groep onder leiding van Richard Snodgrass telt leden vanuit de universiteiten van Arizona, Athene, Bologna, Californië, New York, Zuid-Australië, Tel Aviv, Texas en Aalborg en een aantal andere instituten en bedrijven. De groep produceerde een specificatie die drie soorten ondersteuning voor het concept tijd toe-

voegde aan de 1992-versie van de SQL-standaard: geldige tijd, transactietijd en door de gebruiker gedefinieerde tijd. Deze specificatie werd in september 1994 aan de ISO aangeboden. Voorzover ik weet was dit de eerste keer dat de standaardisatie-organisatie die zich met SQL bezighoudt van een academische groep een (bijna) concreet voorstel tot wijziging van de standaard ontving. De in TSQL2 vervatte toevoegingen waren voor de ISO-commissie niet direct acceptabel. In de eerste plaats was ze zelf inmiddels al verder gevorderd dan de SQL:1992-standaard en was men druk bezig met het toevoegen van abstracte gegevenstypen, stored procedures en een groot aantal andere zaken. Het waren stuk voor stuk toevoegingen die men onder de loep zou moeten nemen als er in wat voor vorm dan ook een complete verzameling temporele uitbreidingen op SQL zou komen. Het tweede bezwaar van de commissie was dat de ingediende specificaties minder rigoureus waren geformuleerd dan voor een internationale standaard is vereist. Een groep leden van de TSQL2-commissie begon daarom, met hulp van een aantal ervaren leden van de ISO-commissie, met het verbeteren en het met het oog op de nieuwe situatie actualiseren van de oorspronkelijke TSQL2-specificatie.

Laten we echter, voordat we naar de voorgestelde uitbreidingen op SQL gaan kijken, eens zien wat voor mogelijkheden we, ongeacht welke temporele uitbreiding op SQL er precies komt, daar in terug willen zien.

Opwaartse compatibiliteit is misschien wel het belangrijkste aspect van het streven om de SQL-taal uit te breiden met directe ondersteuning voor temporele aspecten. Hiermee bedoelen we dat we er voor moeten zorgen dat alle bestaande applicatiecode met het nieuwe systeem (SQL/Temporal) exact dezelfde functionaliteit heeft als met het oude systeem en dus, nog belangrijker, exact dezelfde resultaten oplevert.

Als we nogmaals figuur 23 bekijken, zien we dat als een query wordt toegepast op een snapshot-database, alleen de huidige toestand van de tabel wordt bekeken. Dit komt doordat alleen de huidige toestand van de tabel beschikbaar is. Terwijl dit in figuur 23 alleen betrekking heeft op query's op één tabel, ligt uitbreiding naar query's die meerdere tabellen tegelijk bestrijken voor de hand. Welke SQL/Temporal-implementatie we ook hebben, we zullen een manier moeten hebben om deze semantiek te kunnen onderhouden. Een opwaarts compatible uitbreiding moet er voor zorgen dat:

- alle instanties van tabellen in SQL ook in SQL/Temporal geldige instanties van tabellen zijn
- alle bewerkingen op niet-temporele tabellen onder SQL/Temporal identiek zijn aan dezelfde bewerkingen onder SQL
- alle niet-temporele query's op temporele tabellen werken alsof ze een niet-temporele tabel ondervragen met dezelfde toestand als de huidige toestand van de temporele tabel
- alle niet-temporele SQL-wijzigingen op temporele tabellen resulteren in dezelfde huidige toestand van de temporele tabellen als in het geval van

Count the clock

de equivalentie niet-temporele tabel waarop dezelfde wijziging is toegepast.

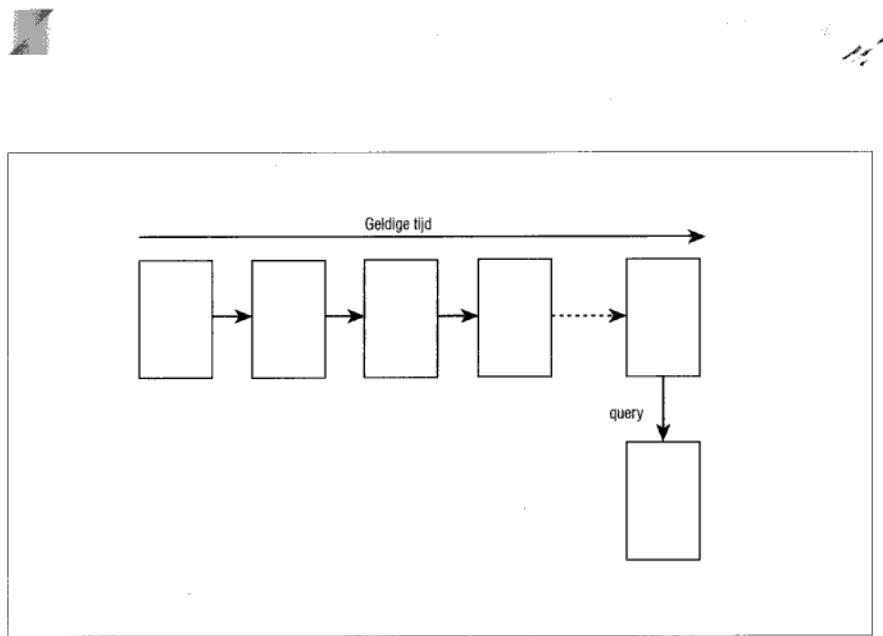
Naast opwaartse compatibiliteit in SQL/Temporal is ook ondersteuning voor de volgende zaken gewenst:

- minimaal één geldige tijdlijn; een geldige tijdlijn is een tijdlijn die veranderingen in de buitenwereld weerspiegelt die in de database worden gerepresenteerd
- de transactietijdlijn
- selecteerbare temporele ondersteuning voor elke tijdlijn op een tabel-voor-tabel basis
- afleiden van temporele en niet-temporele tabellen vanuit onderliggende temporele en niet-temporele tabellen
- temporele tegenhangers van de 'aggregate'- en 'set'-functies
- structurele beperkingen die automatisch rekening houden met de additionele temporele aspecten van de tabellen waarnaar zij verwijzen
- een eenvoudige en intuïtieve uitbreidingen op de syntaxis die het gebruik en de manipulatie van temporele gegevens mogelijk maken
- een eenvoudige syntaxis om temporele tabellen te definiëren of bestaande snapshot-tabellen in temporele tabellen om te zetten
- 'stofzuigen', dit is de mogelijkheid om temporele gegevens van voor een bepaald punt fysiek te verwijderen. Een van de nadelen van temporele databases is dat normaliter niets wordt weggegooid. Zonder een of ander concept zoals dit stofzuigen, zouden temporele databases voortdurend uitdijken.

In figuur 23 bekijken we een snapshot-database in de loop van de tijd, in figuur 24 is te zien hoe een temporele database daarvan verschilt. In dit geval zijn de gestippelde rechthoeken vervangen door doorlopende lijnen aangezien in een temporele database de informatie niet wordt weggegooid. De rechthoek rechts blijft de huidige toestand en wordt gebruikt als de bron voor alle niet-temporele query's.

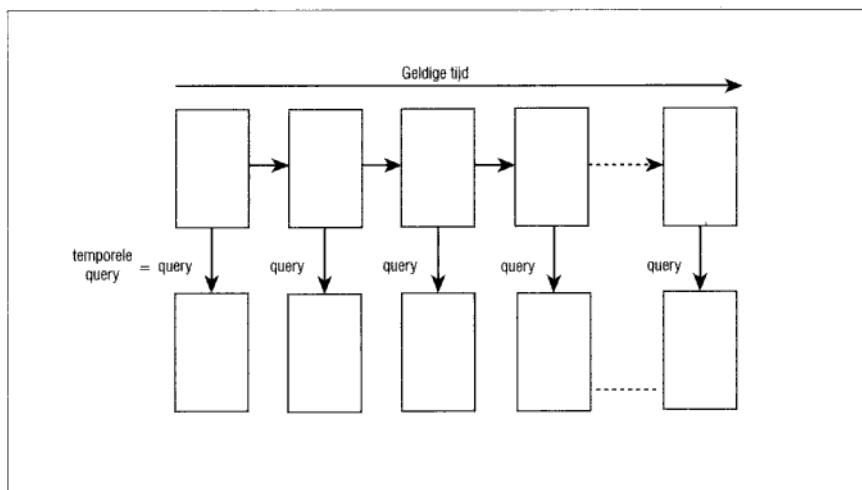
Laten we een aantal temporele query's eens nader bekijken. Twee verschillende typen temporele query's kunnen worden onderscheiden: sequentiële query's en niet-sequentiële query's.

Een sequentiële query is een soort 'natuurlijke' uitbreiding op de snapshot- of niet-temporele query. Wat hij in feite doet is een conventionele (niet-temporele) query op alle toestanden van de database tegelijk los laten. Dit wordt grafisch weergegeven in figuur 25. Het resultaat van de temporele query is dat de combinatie van de resultaten van de niet-temporele query wordt toegepast op iedere (snapshot-) toestand van de tabel die geldige tijd ondersteunt. Het resultaat is daarom ook een tabel met ondersteuning voor geldige tijd, waarvan de inhoud bestaat uit de verzameling snapshot-toestanden die van elk van de snapshot-query's is afgeleid. Elke snapshot-toestand is geldig voor dezelfde periode als de toestand waarvan hij is afgeleid.



Figuur 24. Geldige tijdlijn.

Een voorbeeld van een sequentiële query is: "Wat is het salarisverleden van elke werknemer?" De equivalentie snapshot-query is: "Wat is het salaris van elke werknemer?" Het verschil is dat we in de eerste query vragen om de geldige tijdlijn informatie en in het tweede geval alleen om de snapshot-waarde. Het sleutelkenmerk van een sequentiële query is dus dat hij een eenvoudige snapshot-tegenhanger heeft en zelf niet expliciet verwijst naar het tijdselement. In het volgende hoofdstuk zullen we niet-sequentiële query's onder de loep nemen.



Figuur 25. Een sequentiële query.

Count the clock

De voorgestelde oplossing

Nu we een aantal belangrijke aspecten van temporele databases de revue hebben laten passeren, kunnen we gaan kijken naar de voorgestelde oplossing. De voorbeelden zijn gebaseerd op het bij ISO ingediende voorstel van het TSQL2 Language Committee. Aan het eind van het volgende hoofdstuk zal ik een tegenvoorstel schetsen dat wordt gesteund door het Verenigd Koninkrijk en Griekenland.

De TSQL2-voorvechters geloven dat het, door de eis dat SQL/Temporal een strikte superset is (en dus alleen constructies en semantiek toevoegt), relatief eenvoudig is om er voor te zorgen dat SQL/Temporal opwaarts compatibel is met SQL3. Om de oorspronkelijke tabeldefinities in SQL/Temporal-vorm te gieten, zou men dus het volgende moeten schrijven:

```
CREATE TABLE TWerknemer (
    WnrNr      NUMERIC(5),
    Naam       CHARACTER(40),
    PositieCd  NUMERIC(3),
    Afd        CHARACTER(40),
    Manager    NUMERIC(5),
    VALIDTIME PRIMARY KEY (WnrNr),
    VALIDTIME FOREIGN KEY (Manager)
        REFERENCES TWerknemer
) AS VALIDTIME (DAY);

CREATE TABLE TSalaris (
    WnrNr      NUMERIC(5),
    Salaris    NUMERIC(6),
    VALIDTIME PRIMARY KEY (WnrNr),
    VALIDTIME FOREIGN KEY (WnrNr)
        REFERENCES TWerknemer
) AS VALIDTIME (DAY);
```

Merk op dat we als enige veranderingen aan het eind van de tabeldefinitie "VALIDTIME", en voorafgaand aan elke beperking de tekst "AS VALIDTIME (DAY)", hoeven toe te voegen. (NB: Ik heb de tabellen een andere naam gegeven, zodat ik later onderscheid kan maken tussen de temporele en de niet-temporele versies). "(DAY)" dient hier alleen om het oplossend vermogen van de tijdlijn voor deze tabel te definiëren. (In dit geval houden we de geschiedenis dus slechts bij in dagen. In een echt dynamische organisatie waarin de manager en/of het salaris van iemand meer dan eens per dag zou kunnen veranderen, zouden we met dit systeem dus informatie kwijtraken.)

We hebben geen van de beperkingen hoeven te herschrijven. De beperkingen, met de toevoeging van het nieuwe sleutelwoord, werken op elke individuele toestand van de tijdlijn. Het extra sleutelwoord is nodig om de geldige tijdlijnbeperking te kunnen onderscheiden van een beperking die alleen op de huidige toestand van de database hoeft te worden toegepast. We zouden bijvoorbeeld de volgende beperking kunnen toevoegen:

```
ALTER TABLE TSalaris ADD VALIDTIME  
    CHECK (Salaris > 2000 AND Salaris < 45000);
```

Dit zou de controle voor elke tijdsperiode geldig maken, terwijl

```
ALTER TABLE TSalaris ADD  
    CHECK (Salaris > 2000 AND Salaris < 45000);
```

alleen op de huidige toestand van toepassing zou zijn. De eerste beperking is te vergelijken met een sequentiële query, terwijl de tweede duidelijk een snapshot-query is. Als we verschillende beperkingen willen hebben voor verschillende tijdsperioden, kan dat met behulp van een niet-sequentiële check-clausule worden bereikt. Het bovenstaande maakt duidelijk dat, althans voorzover het om de Data Definition Language (DDL) gaat, SQL/Temporal met niet meer dan een reeks eenvoudige en intuïtieve uitbreidingen op de syntaxis een behoorlijk krachtige additionele semantiek kan krijgen. Dit zal natuurlijk van weinig nut zijn als de syntaxis van de query's zelf niet eveneens eenvoudig en intuïtief is. Gelukkig is dat naar mijn mening ook het geval. Neem de eerste hierboven geformuleerde query:

```
SELECT WnrNr  
    FROM Werknemer  
   WHERE Wnr      NOT IN  
          (SELECT Manager FROM Werknemer);
```

Dit werkt ongewijzigd op zowel Werknemer als TWerknemer (gesteld dat de tabelnaam is veranderd) en geeft in beide gevallen hetzelfde resultaat terug. Voor deze query wordt dus voldaan aan de eis van opwaartse compatibiliteit.

De tweede query die we stelden, maar niet oplosten, was "Geef de werknemers die geen manager zijn of waren, samen met informatie over de perioden waarin dit het geval was". Deze query is nu uiterst eenvoudig. De snapshot-query blijft hetzelfde als in de eerste query. Het enige dat we hoeven te doen is hem om te zetten in een sequentiële query. Zoals we bij de beperkingen hierboven hebben gezien, kan dit door eenvoudigweg een sleutelwoord toe te voegen. De sequentiële query wordt dan:

Count the clock

```
VALIDTIME SELECT Wnr
  FROM TWerknemer
 WHERE Wnr Nr NOT IN
      (SELECT Manager FROM TWerknemer);
```

In plaats van een snapshot-tabel terug te geven, geeft dit een geldige tijd-tabel terug die precies de gezochte informatie bevat. De voorgestelde syntaxis voor SQL/Temporal zou het ook mogelijk maken een geldige tijdsperiode te specificeren waarvoor de query zou moeten worden geëvalueerd. Als we dus alleen de informatie van de laatste drie jaar zouden willen hebben, zou dit als volgt kunnen worden gespecificeerd:

```
VALIDTIME
  PERIOD '[DATE'1993-09-01' - DATE'1996-09-01']'
  SELECT WnrNr
    FROM TWerknemer
   WHERE WnrNr NOT IN
      (SELECT Manager FROM TWerknemer);
```

We moeten natuurlijk ook de mogelijkheid hebben om op onze geldige tijdtabellen wijzigingen uit te voeren, in plaats van ze alleen te kunnen ondervragen. Een normale delete-opdracht op een temporele geldige-tijd-tabel mag als enige effect hebben dat de huidige (en toekomstige) toestand wordt gewijzigd.

```
DELETE TWerknemer
  WHERE WnrNr = 12036;
```

Dit wil zeggen dat de delete-actie de huidige tijd-periode van het record in kwestie afsluit, zodat het van nu af aan niet meer bestaat, maar dat zijn bestaan en zijn toestand voor alle voorafgaande tijdsperioden ongewijzigd blijft. Uiteraard zou het kunnen gebeuren dat men informatie in een niet-huidige periode moet corrigeren of een record wil verwijderen vanaf een bepaalde datum in het verleden. Niet alle informatie met betrekking tot de buitenwereld bereikt de database ook inderdaad op het moment dat de verandering zich feitelijk voordoet. Soms bereikt de informatie over veranderingen de betrokken mensen, laat staan de database, pas een behoorlijke tijd nadat de verandering zich voordeed.

Als we het bereik van veranderingen op de geldige tijd-tabel willen beïnvloeden, kunnen we daarvoor dezelfde periode-bepalende termen toepassen

die in de query's werden gebruikt. Als ik bijvoorbeeld de eerdere delete-actie wil antedateren, zou ik kunnen schrijven:

```
VALIDTIME
PERIOD '[DATE'1996-07-01' - CURRENT_DATE]'
DELETE TWerknemer
WHERE WnrNr = 12036;
```

Dit verwijdert de Werknemer per 1 juli 1996. Merk op dat de specificatie van de periode hier essentieel is. Eenvoudigweg:

```
VALIDTIME
DELETE TWerknemer
WHERE WnrNr = 12036;
```

schrijven, zou het effect hebben dat werknemer 12036 voor altijd wordt verwijderd. Dit zou rampzalig zijn, tenzij dat nu precies hetgeen is wat we willen natuurlijk.

Wijzigingen kunnen op een soortgelijke manier worden gedefinieerd. Zo stelt

```
VALIDTIME
PERIOD '[DATE'1994-03-01' - DATE'1994-03-31']'
UPDATE Werknemer
SET Manager = 99073
WHERE Afd = 'Systeemontwikkeling';
```

ons in staat het Manager-veld van alle werknemers die bij de afdeling 'Systeemontwikkeling' hebben gewerkt, te wijzigen. In dit geval om aan te geven dat er in 1994 gedurende de maand maart een interimmanager was aangesteld als tijdelijk vervanger van de oorspronkelijke manager (de ongelukkige die de kostbare ontwikkeling had goedgekeurd van een applicatie waarin onvoldoende rekening was gehouden met de temporele aspecten).

Het zal u zijn opgevallen hoe eenvoudig dit alles is. We kunnen het aan het database-managementsysteem overlaten om na te denken over de noodzaak om bestaande records op te splitsen, records te verwijderen of wat dan ook te doen om de correcte geldige tijd-volgorde te genereren. We hoeven alleen maar te specificeren wat we willen en het systeem doet de rest. De kans dat een programmeur deze syntactische specificatie correct uitvoert is aanzienlijk hoger dan als hij alle mogelijke situaties had moeten uitpluizen en elke oplossing met de hand had moeten programmeren.

Count the clock

In het volgende hoofdstuk zullen we niet-sequentiële queries, transactietijd en een alternatieve syntax bekijken.

De titel van dit hoofdstuk is gebaseerd op een uitspraak in William Shakespeare's Julius Caesar, Tweede Akte, Scene 1.



the first time in the history of the world, the people of the United States have
been called upon to decide whether they will submit to the law of force, or the law of the
globe.

—

In time we hate that which we often fear

Stephen J. Cannan

Wanneer we het over tijd hebben, is het belangrijk om duidelijk te zijn. Er is een groot verschil tussen "Wanneer dachten we dat x gebeurde?" en "Wanneer gebeurde x naar ons beste weten?". Vele query's zijn complex en temporele query's zijn nog complexer. Het is gemakkelijk om soortgelijke temporele query's te verwarringen maar dit kan tot zeer verschillende resultaten leiden. Er bestaat een duidelijke vaagheid bij onze benadering ten opzichte van de tijd hetgeen suggereert, zoals Shakespeare stelt in de titel van dit hoofdstuk, dat we niet echt goed kunnen omgaan met onze eigen sterfelijkheid. In ons dagelijks leven willen we, indien mogelijk, niets weten van Tijd, de grote opruimer, maar in onze databases doen we dat op eigen risico!

In het vorige hoofdstuk keken we naar de behoefte en de eisen met betrekking tot het omgaan met temporele gegevens in SQL en naar hoe een geldige tijdondersteuning kan worden gespecificeerd en hoe sequentiële query's kunnen worden uitgedrukt. In dit hoofdstuk zullen we kijken naar de meer gecompliceerde niet-sequentiële query's, de andere tijdlijn en een alternatieve syntaxis.

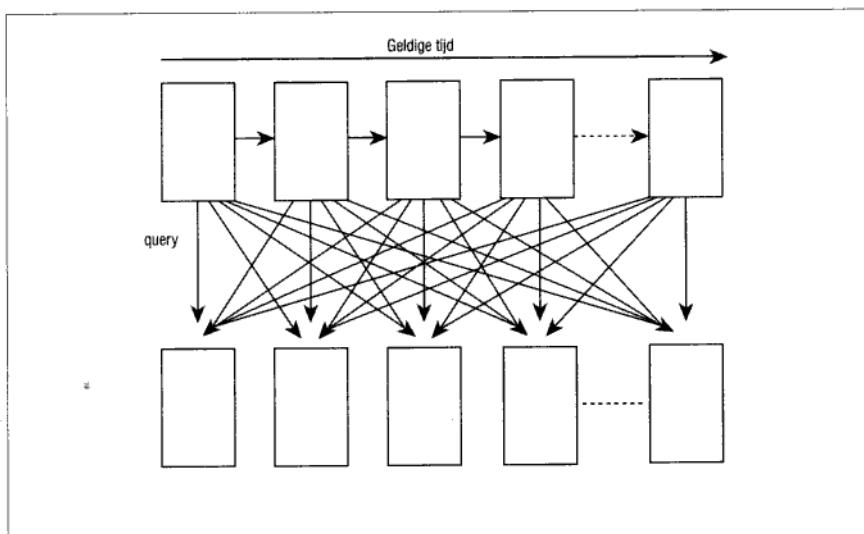
Zoals we zagen werkt een sequentiële query op elke geldige tijdstatus van de database en refereert zelf niet aan het tijdelement. Een niet-sequentiële query refereert echter expliciet aan het tijdelement en kan hetzij een tijdelijke, hetzij een niet-tijdelijke tabel opleveren.

In figuur 26 hebben we getracht de omvang van de niet-sequentiële query te illustreren. Opgemerkt zij dat we, wanneer we elke gegeven toestand van de database beschouwen, ook andere geldige toestanden dienen te overwegen. Dit wordt weergegeven in het grote aantal diagonale lijnen in het diagram.

Denk eens aan de volgende query: wie hebben opslag gekregen? Deze query maakt het noodzakelijk dat we de geldige tijdlijn inspecteren op salarisveranderingen. In dit geval hoeven we alleen maar de voorafgaande toestand te

2.3.2 Niet-sequentiële query's

bekijken, maar het principe is hetzelfde. Deze query maakt het niet noodzakelijk dat de lijst van de medewerkers in de tijd varieert; daarom is de resulterende tabel niet tijd-afhankelijk. Dit betekent dat alleen de rechthoek rechts onderaan betrekking heeft op het resultaat.



Figuur 26. De omvang van de niet-sequentiële query.

Een andere query is: van wie was het salaris meer dan tien procent meer dan het salaris één jaar daarvòòr en wanneer? In dit geval dient deze query het tijdelement van het salaris te inspecteren en ook de geldige tijdperioden terug te sturen waarbij aan deze conditie wordt voldaan. Voor elke toestand van de database, afhankelijk van hoe vaak het salaris is veranderd, maakt dit de potentiële inspectie van vele andere toestanden van de database noodzakelijk. De query is derhalve niet-sequentieel en levert een temporele tabel op.

Nadat we hebben gekeken naar sequentiële query's en modificaties, besteden we nu aandacht aan de meer gecompliceerde niet-sequentiële query's. Al eerder vroeg ik "Wie hebben opslag gekregen?". We kunnen nu vrij eenvoudig die vraag beantwoorden met de query:

In time we hate that which we often fear

```
NONSEQUENCED VALIDTIME
    SELECT WnrNr, Naam
        FROM Werknemer A E,
             Salaris as S1,
             Salaris AS S2
       WHERE E.WnrNr = S1.WnrNr
         AND E.WnrNr = S2.WnrNr
         AND S1.Salaris < S2.Salaris
         AND VALIDTIME(S1) MEETS VALIDTIME(S2);
```

Hier moeten we twee opeenvolgende toestanden van de salaristabel in de database beschouwen voor elke werknemer en kijken of het bedrag van het salaris is toegenomen. We hebben het NONSEQUENCED-sleutelwoord nodig om het mogelijk te maken de twee toestanden van de database direct te bekijken. Zonder dit zou de query geen resultaat opleveren omdat per definitie slechts één salaris actueel kan zijn op één moment en derhalve niet hoger kan zijn dan zichzelf. Wanneer we meer dan één toestand van de database beschouwen, moeten we er zeker van zijn dat we alleen records beschouwen die elkaar in een geldige tijdvolgorde volgen. Dit is wat het "VALIDTIME(S1) MEETS VALIDTIME(S2)"-gedeelte van de WHERE-clausule doet. De functie VALIDTIME extraheert de geldige tijdsperiode voor een bepaalde toestand van de salaristabel. Deze functie, indien toegepast in een selecte lijst, biedt ons ook een methode om uiterlijke vorm te geven aan de geldige tijdsperioden die impliciet zijn aan een geldige tijdstabel. De perioden van beide toestanden worden geëxtraheerd met behulp van deze functie en deze worden vergeleken met gebruikmaking van het MEETS-predikaat. MEETS is alleen waar indien de S1 periode onmiddellijk wordt gevuld, zonder hiaten of overlappingen, door de S2 periode. MEETS is een zeer belangrijk predikaat bij SQL/Temporal. Allen [8] definieert dertien tijdelijke predikaten:

EQUAL	CONTAINS
BEFORE	FINISHES
MEETS	FINISHED-BY
OVERLAPS	OVERLAPPED-BY
STARTS	MET-BY
STARTED-BY	AFTER
DURING	

SQL/Temporal stelt alleen voor er negen te bieden, namelijk:

OVERLAPS	(OVERLAPS, OVERLAPPED-BY)
MEETS	(MEETS, MET-BY)
PRECEDES	(BEFORE)
SUCCEEDS	(AFTER)
CONTAINS	(CONTAINS, DURING)
=	(EQUALS)

OVERLAPS is reeds beschikbaar in SQL:1992. Gelukkig kunnen de andere, STARTS, STARTED-BY, FINISHES en FINISHED-BY, alle worden uitgedrukt in de termen van de andere. In feite kan MEETS alleen al worden gebruikt om alle andere condities uit te drukken.

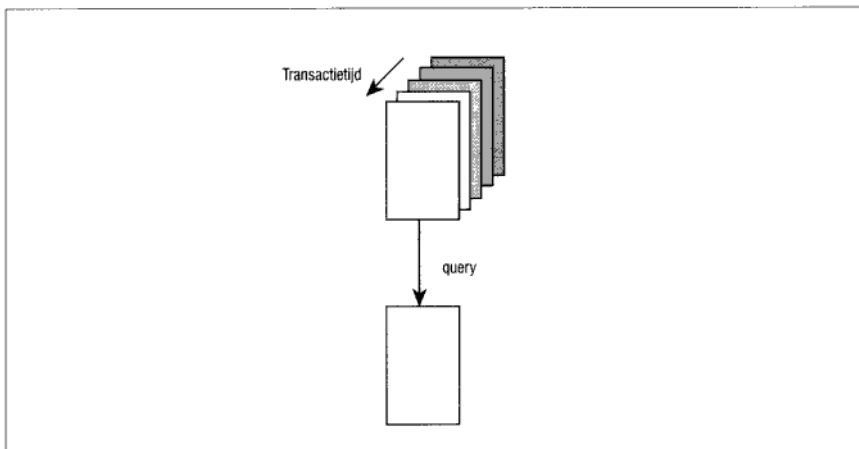
We moeten niet alleen niet-sequentiële query's kunnen schrijven, we moeten ook niet-sequentiële modificaties kunnen schrijven. Zodoende moeten we in staat zijn "de laatste salarisverhoging te schrappen" of, in het meer werknemer-vriendelijke bedrijf, "een salarisverhoging te geven van tien procent aan alle medewerkers die nog nooit een salarisverhoging hebben gehad". Dit laatstgenoemde zouden we kunnen implementeren met de statement:

```
UPDATE Salaris
    SET Salaris = Salaris * 1.1
    WHERE NOT EXISTS (
        NONSEQUENCED VALIDTIME
        SELECT WnrNr, Naam
        FROM Werknemer A E,
             Salaris as S1,
             Salaris AS S2
        WHERE E.WnrNr = S1.WnrNr
          AND E.WnrNr = S2.WnrNr
          AND S1.Salaris < S2.Salaris
          AND VALIDTIME(S1)MEETS VALIDTIME(S2));
```

Tot nog toe hebben we alleen de geldige tijdlijn bekeken. Zoals we hebben gezien heeft deze tijdlijn betrekking op de verslaglegging wanneer iets in werkelijkheid is gebeurd of veranderd (of althans in het model van de applicatie). Er is, natuurlijk, een andere tijdlijn die niet betrekking heeft op wanneer iets werkelijk gebeurde maar op wanneer die gebeurtenis werd vastgelegd. Dit is de transactietijdlijn. Deze legt vast wanneer er veranderingen werden aangebracht in de database en kan worden gebruikt om veranderingen aan tabellen vast te leggen ongeacht het feit of die tabellen geldige tijdtemporele tabellen of goede ouderwetse snapshot-tabellen zijn. De transactietijdlijn wordt in figuur 27 grafisch getoond. Zoals deze figuur laat zien is

In time we hate that which we often fear

de transactietijdlijn orthogonaal aan de geldige tijdlijn. De transactietijdlijn verschilt van de geldige tijdlijn in een aantal opzichten. Zo wordt de geldige tijd alleen begrensd (tenminste in theorie) door het begin en het einde van het universum; dit wil zeggen ruimte en tijd zelf. De transactietijd is nauwer begrensd, door het vervaardigen van de database en nu.



Figuur 27. De transactietijdlijn.

Op deze wijze kan de transactietijd, in tegenstelling tot de geldige tijd, nooit toekomstige gebeurtenissen vastleggen. Een ander belangrijk verschil is dat de waarden van de tijden die zijn vastgelegd in de transactietijdperioden, niet zijn en niet kunnen worden gespecificeerd of veranderd door een gebruiker, hoe dan ook. De waarden worden altijd betrokken van de systeemklok op het moment dat de database wordt veranderd. Niet alleen kunnen de tijdsperioden niet worden veranderd, geen enkel gegeven in een enkel transactietijd-record dat niet het huidige (nu geldige) record is, kan worden bijgewerkt. We zijn niet van plan om Captain Picard of Captain Kirk terug te laten gaan en hun verleden te laten veranderen. Het kan goede TV opleveren maar het resulteert in een slecht database-beheer.

Alhoewel het waar is dat de transactietijdlijn zeldzamer is dan de allesheersende geldige tijdlijn, heeft deze wel degelijk zijn nut. Neem bijvoorbeeld een tabel waarin een userid is opgeslagen. De transactietijd temporale tabelomschrijving zou er als volgt uit kunnen zien:

```
CREATE TABLE User_Code (
    Userid      CHARACTER(8),
    NONSEQUENCED TRANSACTIONTIME
        PRIMARY KEY (Code)
) AS TRANSACTIONTIME;
```

In deze omschrijving wordt de primaire sleutel omschreven met een reikwijdte over alle transactietijd-records; hierdoor is verzekerd dat een userid altijd uniek is. Het schrappen van een userid zal hem ongeldig maken maar omdat het bijbehorende record nog steeds in de transactietijd-tabel aanwezig zal zijn kan de beperking van de primaire sleutel deze nog steeds in aanmerking nemen. Een userid kan zodoende maar één keer worden verstrekt zelfs nadat deze wordt ingetrokken. De transactietijd kan niet alleen worden gebruikt voor beperkte query's. Op een geschikte tabel zouden we kunnen vragen hoe vaak de naam van een werknemer is gecorrigeerd. Bijvoorbeeld:

```
TRANSACTIONTIME
    SELECT WnrNr, COUNT(*)-1 AS Verandering;
        FROM Werknemer
            GROUP BY WnrNr;
```

Natuurlijk willen we een beperking aanbrengen aan de transactietijd-periode die de query in beslag neemt en net als bij de geldige tijd kunnen we een periode specificeren:

```
VALIDTIME
    PERIOD '[DATE'1993-09-01' -DATE'1996-09-01']'
        SELECT WnrNr, COUNT(*)-1 AS Veranderingen
            FROM Werknemer
                GROUP BY WnrNr;
```

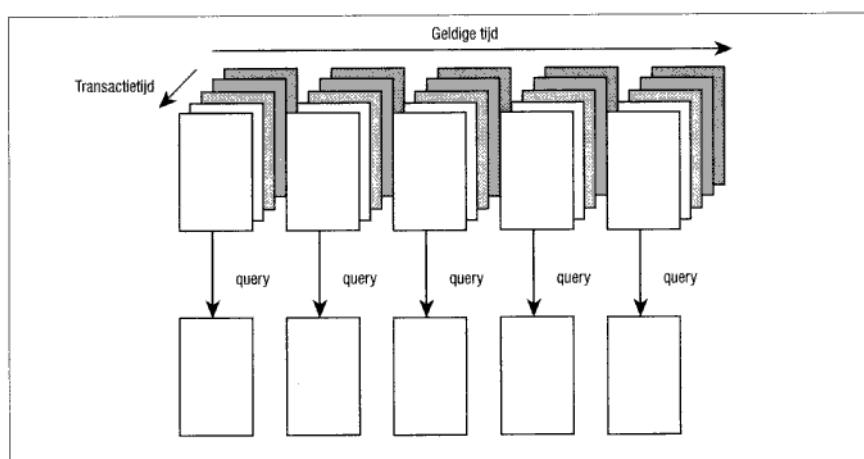
In principe kunnen transactietijd regel-query's een tijdelijk opwaarts compatibele, sequentiële en niet-sequentiële vorm aannemen. Terwijl we in geldige tijd de betekenis van een tijdelijk opwaarts compatibele query ruwweg kunnen vertalen als "vanaf nu", in transactietijd wordt het "als beste bekend". Voor sequentiële geldige tijden "op hetzelfde tijdstip" wordt het transactietijd-equivalent "wat wij op hetzelfde tijdstip dachten" en niet-sequentiell van "op elk tijdstip" tot "op elk vastgelegd moment".

Het bedenken van goede voorbeelden is moeilijk omdat in de praktijk de transactietijdlijn meestal wordt gebruikt in samenwerking met de geldige tijdlijn. Deze schrikaanjagende configuratie, waarbij beide tijdlijken zijn gecombineerd, wordt weergegeven in figuur 28.

We kunnen nu vragen stellen als "Wanneer toonde de database dat een werknemer dezelfde manager had gedurende meer dan negen maanden?". Dit vertaalt zich in:

In time we hate that which we often fear

```
VALIDTIME AND TRANSACTIONTIME
SELECT WnrNr
FROM TWerknemer
WHERE INTERVAL(VALIDTIME(Werknemer) MONTH)
> INTERVAL '6' MONTH;
```



Figuur 28. Combinatie van de geldige tijdlijn en de transactietijdlijn.

Anderzijds zouden we kunnen vragen: "Wanneer werd de vastgelegde positie van 1995 van een werknemer gecorrigeerd?". Dit omvat niet-sequentiële transactiequery-semantiek en sequentiële geldige tijd query-semantiek met een temporele omvang. Is dit ingewikkeld genoeg voor iedereen?

```
NONSEQUENCED TRANSACTIONTIME
AND VALIDTIME PERIOD '[DATE'1995-01-01'-'DATE'1995-12-31']'
SELECT WnrNr,
       e1.PositieCd ASOld_pos,
       e2.PositieCd ASNew_pos,
       BEGIN(TRANSACTIONTIME(e2))
              ASTrans_time
FROM TWerknemer AS e1
      TWerknemer AS e2
WHERE e1.WnrNr = e2.WnrNr
  AND TRANSACTIONTIME(e1) MEETS TRANSACTIONTIME(e2)
  AND e1.PositieCd #\$ e2.PositieCd;
```

Tot nu toe zijn alle voorbeelden betrokken van de voorstellen [3,4] van TSQL2 aan de ISO SQL-commissie. Er is een alternatief voorstel aan de commissie uit het Verenigd Koninkrijk [5]. Dit is gebaseerd op IXSQL, het werk van Nikos Lorenzos van de landbouwuniversiteit van Athene. IXSQL staat voor Interval eXtended SQL.

Het alternatieve voorstel heeft het voordeel dat het meer dan één geldige tijdlijn kan ondersteunen, alsmede de transactietijdlijn. Dit kan omdat het in tegenstelling met TSQL2 geen impliciete periodekolommen heeft ter ondersteuning van de tijdlijnen. Zodoende is een database-ontwerper vrij om net zo veel periodekolommen toe te voegen als hij wil ter ondersteuning van meervoudige tijdlijnen.

Het belangrijkste nadeel is dat de query's veel ingewikkelder zijn om te schrijven. In het vorige hoofdstuk vroegen we om een query die een lijst moest opleveren van "die werknemers die hetzij geen managers zijn of waren, in combinatie met informatie over de perioden gedurende welke dit van kracht was". Uit hoofde van het TSQL-voorstel voor SQL/Temporal hadden we de volgende query nodig:

```
VALIDTIME SELECT WnrNr
    FROM TWerknemer
   WHERE WnrNr NOT IN
        (SELECT Manager FROM TWerknemer);
```

Om een gelijksoortig resultaat te verkrijgen onder het voorstel uit het Verenigd Koninkrijk zouden we een query moeten schrijven als:

```
WITH e1 AS
    (SELECT WnrNr, EXPAND(When) ASew
     FROM Werknemer)
SELECT WnrNr, PERIOD [Valid, Valid] AS Valid
    FROM e1, TABLE(e1.eW) AS e2(Valid
   WHERE WnrNr NOT IN
        (SELECT Manager
         FROM Werknemer AS e3
        WHERE e3.Valid = e2.Valid)
  NORMALIZE ON Valid;
```

Weer een ander ontbrekend onderwerp uit het huidige voorstel uit het Verenigd Koninkrijk wordt gevormd door temporeel gevoelige primaire en vreemde sleutels. Alle beperkingen zouden moeten worden geschreven als vrij ingewikkelde multi-tabel check-clausules die het uitdrukkingsvermogen van de huidige beperkingen zouden missen. Erger nog, zij zouden door een

In time we hate that which we often fear

optimaliseerder moeilijk zijn te herkennen tussen de ontelbare mogelijke check-clauses waardoor het prestatievermogen (of de integriteit) een probleem wordt.

De periodekolom die wordt gebruikt voor de transactietijdlijn is een algemeen beschikbare kolom en kan als zodanig worden bijgewerkt. Startrek, we komen er aan!

Voor dat de SQL/Temporal-standaard gereedkomt kunnen we alleen maar hopen dat de extra flexibiliteit van het voorstel uit het Verenigd Koninkrijk kan worden gecombineerd met de uitdrukkingskracht en eenvoud van de TSQL2-voorstellen. Indien dit niet het geval is, weet ik welk alternatief ik steun. Het alternatief dat mij een goede kans biedt een query te schrijven die het antwoord oplevert dat ik verwacht. Hoe denkt u daarover?

De titel van dit hoofdstuk is gebaseerd op een uitspraak in William Shakespeare's Anthony and Cleopatra, Eerste Akte, Scene 3.



Literatuur

1. S.J. Cannan, *Tijd voor SQL? (deel 1)*, Database Magazine 1990/4.
2. S.J. Cannan, *Tijd voor SQL? (deel 2)*, Database Magazine 1990/5.
3. R. Veldwijk, *Tussen eeuwig heden en complex verleden*, Database Magazine 1995/3.
4. Jan van Rooijen, *Tijdafhankelijkheid in het relationele model*, Database Magazine 1993/2.
5. Richard T. Snodgrass, Michael H. Böhlen, Christian S. Jensen en Andreas Steiner, *Adding Valid Time to SQL/Temporal*, ISO/IEC JTC1/SC21/WG3 DBL MCI-142, 10 mei 1996.
6. Hugh Darwen, *Expanded Table Operations*, ISO/IEC JTC1/SC21/WG3 DBL MCI-67, 11 april 1996.
7. S.J. Cannan en G.A.M. Otten, *Handboek SQL*, Alphen aan den Rijn, 1992.
8. J. Allen, *Maintaining knowledge about temporal intervals*, Communications of ACM. 26, 123-154, 1983.
9. Hugh Darwen, *Expanded Table Operations*, ISO/IEC JTC1/SC21/WG3 DBL MCI-67, 11 April 1996.
10. Jensen, C.S. et al. (eds.), *A Glossary of Temporal Database Concepts*, ACM SIGMOD Record 23(1):52-64, maart 1994.
11. Ozsoyoglu, G. and R.T. Snodgrass, *Temporal and Real-Time Databases: A Survey*, IEEE Transactions on Knowledge and Data Engineering 7 (4):513-532, August 1995.
12. Tansel, A. et al., *Temporal Databases: Theory, Design, and Implementation*, Benjamin/Cummings, 1993.
13. Tsotras V.J. and A. Kumar, *Temporal Database Bibliography Update*, ACM SIGMOD Record 25(1):41-51, maart 1996.
14. Zaniolo, C. et al., *Advanced Database Systems*, Morgan Kaufmann, 1997.
15. Dr. R.J. Veldwijk, *Tien Geboden voor goed database-ontwerp*, Array Publications b.v., 1996.

FAA Groep

Architecten en ontwikkelaars van flexibele informatiesystemen

Informatietechnologie wordt vaak aangeprezen als middel om de flexibiliteit en daarmee het concurrend vermogen van organisaties te verhogen. De praktijk wijst echter uit dat het resultaat van gangbare wijzen van systeemontwikkeling eerder een vertragende werking heeft op de 'time-to-market'. Daarbij komt nog eens dat de met IT gepaard gaande kosten in hoog tempo toenemen.

De roep om meer flexibiliteit heeft wel geleid tot een keur aan systeemontwikkelingsmethoden, -filosofieën, -technieken en -hulpmiddelen. De nadruk wordt echter steeds gelegd op de technologie. Technologie die - op de keper beschouwd — voornamelijk gericht is op het verbeteren van de productiviteit van systeemontwikkeling, zowel bij nieuwbouw als bij onderhoud. Productiviteit concentreert zich op de eenvoud en op de snelheid van wijzigen van code.

Voor de FAA Groep is productiviteit maar een deel van het verhaal. Het streven naar flexibiliteit zou allereerst gericht moeten zijn op het vermijden van de noodzaak van veranderingen in programmatuur en/of database structuur. De FAA Groep definieert flexibiliteit als "de mate waarin en het gemak waarmee de gebruiker, of de applicatiebeheerder namens de gebruiker, zelf uitbreidingen of variaties op de applicatie kan aanbrengen, zonder dat de programmatuur wordt aangepast."

Voor onderhoud is de gebruiker dan niet afhankelijk van de ontwikkel- en onderhoudsorganisatie. Voor de FAA Groep begint productiviteit waar flexibiliteit ophoudt.

Eén samenhangend raamwerk: FAA

FAA staat voor Flexibele Applicatie Architectuur. FAA is een IT-concept, een architectuurmodel en generieke software waarmee informatiesystemen kunnen worden ingericht die vergaand voldoen aan bovenstaande definitie.

De FAA concepten, modellen en software richten zich op generieke, domein-onafhankelijke aspecten van applicaties waarin ook moderne IT-omgevingen niet voorzien. Domeinspecifieke applicatie aspecten worden hierop gebaseerd.

FAA is meer dan een set slimme generieke features. Het FAA concept eist dat elke generieke component zinvol combineerbaar is met elke andere generieke component. Daarmee wordt het aantal benodigde generieke componenten geminimaliseerd. Deze eis van combineerbaarheid wordt aangeduid met de term Orthogonaliteit.

Rond het begrip flexibiliteit speelt een fundamentele paradox: flexibiliteit kan alleen worden gerealiseerd binnen de kaders van een vast raamwerk. Onbegrenste flexibiliteit kan niet bestaan. Zowel "alles kan" als "niets kan" komen uiteindelijk neer op "business as usual". Architectuur betekent binnen FAA een doordachte, optimale set van onveranderlijke kaders. Deze moet samenvallen met het constante in de veranderende bedrijfsprocessen.

Bij FAA draait alles om de drieënheid flexibiliteit, orthogonaliteit en architectuur. De FAA Groep past deze principes toe op het niveau van producten en processen, applicaties en IT-infrastructuren. Daarbij ligt de focus primair op het niveau van de applicatie-architectuur. De FAA Groep beheert het vertalen van bedrijfsproces architecturen in applicatie architecturen en Informatiesystemen. Omgekeerd leidt het toepassen van FAA principes ook tot het vooraf kunnen identificeren van beperkingen in de mogelijkheid om een bedrijfsstrategie te implementeren. In de complexe bedrijfsmogelijkheden van vandaag is het perspectief dan een optimum tussen utopia en status quo.

De FAA Groep: producten en diensten

⇒ **FAA architectuurmodellen**

Gedetailleerde en branche-onafhankelijke specificaties voor onderhoudsarme informatiesystemen.

⇒ **FAA Toolset**

Implementatie van de FAA modellen.

⇒ **FAA Scan**

Kort onderzoek gericht op het in kaart brengen van de mogelijkheden van de FAA benadering voor uw organisatie.

⇒ **Workshop Geavanceerd Gegevensmodelleren**

Twee- of driedaagse workshop gericht op theorie en praktijk van het modelleren van flexibele informatiesystemen voor administratieve omgevingen

⇒ **Workshop Platform Selectie**

Eendaagse, interactieve workshop gericht op het kwalificeren en selecteren van het optimale ontwikkelplatform voor uw organisatie

⇒ **RVIS aanpak voor IT-kostenreductie**

Rationeel Vervangen van Informatie Systemen: bedrijfseconomisch gedreven vervanging van (te) kostbare en starre informatiesystemen

⇒ **Advies en ontwerp van flexibele applicaties in diverse moderne ontwikkelomgevingen**

Vertaling van informatiebehoeften en flexibiliteitseisen in eenduidige specificaties

⇒ **(Fixed Price) systeemrealisatie**

Beheerste en risicotvrije realisatie van op de FAA-principes en -modellen ontworpen informatiesystemen

⇒ **Toegankelijke know how en know why op flexibiliteitsgebied**

Een veelheid van publicaties over flexibiliteit: proefschriften, white papers, vakpublicaties en brochures

De FAA Groep

De FAA Groep, bestaande uit FAA Partners en FAA Software Engineering, richt zich op het adviseren over en realiseren van flexibele systemen op basis van de FAA benadering. Zij geeft vorm aan een systematische IT-architectuur en ontwikkelt het applicatie-raamwerk voor uw bedrijf op basis van de algemene FAA-modellen, haar bedrijfseconomische en IT-expertise en uw kennis van de bedrijfsspecifieke eisen ten aanzien van functionaliteit en flexibiliteit.

Organisatie-advies, IT-consultancy en systeemontwikkeling worden door de FAA Groep tot één samenhangend geheel vanuit één visie. Deze visie is het best te verwoorden in de kernactiviteit: het realiseren van flexibele ofwel onderhoudsgevoelige informatiesystemen.



FAA Groep

Tel. 0346 - 587076, fax 0346 - 587086, Email: faapartners@faapartners.com

Actuele informatie over de bestelwijze van in dit boek genoemde uitgaven en toekomstige uitgaven van Array Publications is te vinden op internet:

www.array.nl



Dr René Jan Veldwijk studeerde economie aan de VU Amsterdam met als specialisaties administratieve organisatie en informatica. Na zijn afstuderen in 1986 trad hij in dienst bij

het softwarehuis van Raet alwaar hij ervaring opdeed met alle facetten van systeemontwikkeling en zich met name op de gebieden van gegevensmodellering en systeemflexibiliteit ontwikkelde tot expert. In 1993 rondde hij een parttime promotieonderzoek op het gebied van flexibele applicatie-architecturen af met een proefschrift. Na een periode waarin hij afwisselend werkzaam was als manager Research & Business Development bij Raet IT-Services en als zelfstandig adviseur, startte hij in 1996 met enkele gelijkegezinden het bedrijf FAA Partners. Dit bedrijf richt zich op het ontwikkelen en projectmatig implementeren van flexibele applicatie-architecturen gericht op het minimaliseren van applicatie-onderhoud.

Sinds 1993 is René Veldwijk als vaste auteur aan Database Magazine verbonden en heeft hij met name de rubriek database-ontwerp onder zijn hoede. Hij gebruikte dit medium om zijn verbazing uit te spreken over de alledaagse praktijk van modellering waarin middelmaat en dilettantisme soms de norm lijkt te zijn. Dit inspireerde hem tot het formuleren van de Tien Geboden voor Goed Database Ontwerp, een reeks die zoveel bijval kende dat deze als bundel is uitgebracht.

Zijn bijdragen kenmerken zich door een voor het informaticavak zeldzame combinatie van speelsheid, humor, diepgang en eruditie. Als lezer raakt men snel geboeid en wordt men als vanzelf meegevoerd in zijn enthousiasme voor de zeker niet eenvoudige en abstracte materie.

Als een van de eersten in het vakgebied onderkende hij het groeiende belang van tijdafhankelijkheid in gegevensbestanden. Hij achte vorig jaar de tijd rijp om er een zestal afleveringen van de rubriek aan te wijden. Dit boek, de weerslag daarvan, is een bundeling en bloemlezing tegelijk.

Van de hand van Stephen Cannan, de vaste auteur van de SQL-rubriek in Database Magazine, zijn twee artikelen opgenomen over voorstellen tot ondersteuning van tijd door de vraagtaal SQL.

Een bijdrage van Frido van Orden compleert dit boek. Zijn bijdrage behandelt implementatieaspecten van tijdafhankelijkheid in zowel relationele als ook de in opkomst zijnde objectgeoriënteerde databasesystemen.

Deze uitgave werd mede mogelijk gemaakt door:



FAA Groep



9 789074 56204 1

ISBN 90-74562-04-3
NUGI 851

Array PUBLICATIONS