MEMO

AAN
Voor wie het aanbelangt
VAN
Rik Opgenoort
DATUM
20 december 2019
ONDERWERP
Versiebeheer LinkedData
TAG
Versiebeheer; LinkedData; RDF(S), OWL, SPARQL, W3C, Semantic web; VCS; Ontologie; Triplestore.

Doel

Het doel van dit document is een situatieschets geven omtrent de uitdagingen m.b.t. versiebeheer binnen LinkedData (LD). Ook worden de verschillende mogelijkheden beschreven, op basis waarvan per situatie een inschatting gemaakt kan worden voor de best mogelijke optie. Dit document is een samenvoeging van vakliteratuur en eigen kennis, en geeft geen waardeoordeel m.b.t. de genoemde onderdelen.

Aanleiding

De complexiteit van onze wereld groeit en mede daardoor worden we geconfronteerd met een totaal andere manier van het creëren, verspreiden en consumeren van grote hoeveelheden informatie. De recente trend om ons te committeren aan de gestandaardiseerde LD-technieken (RDF en SPARQL) gaat leiden tot gegevens uit verschillende domeinen die online worden gepubliceerd en beschikbaar komen voor een breed spectrum van gebruikers. Het dynamische karakter is een onmiskenbaar onderdeel van LD. De gegevens én het schema van de datasets zijn voortdurend in ontwikkeling. Dat geldt met name in de huidige tijd van adaptatie en transitie, vaak wordt de meeste aandacht gericht op de laatste versie. Maar toch is er nuttige informatie aanwezig in, tussen of over eerdere versies [1]. Dit verhoogt de noodzaak om verschillende versies van datasets bij te houden. En dat op zijn beurt brengt nieuwe uitdagingen met zich mee m.b.t. het waarborgen van de kwaliteit en de traceerbaarheid van de gegevens door de tijd [2].

Versiebeheer voor RDF-vocabulaires, en ontologieën, is een voortdurend onderzoeksprobleem in de internationale gemeenschap. Op de W3C site hierover wordt als eerste zin gezegd: "*Version management for RDF vocabularies, and ontology evolution in particular is an ongoing research problem*"[1]. Ondanks dit onderzoeksprobleen, wordt er wel gedeeld dat het beheren van meerdere versies van evoluerende vocabulaires essentieel is, en dat versiebeheer dat gebruikers in staat stelt om meerdere versies te onderscheiden en te herkennen van elkaar essentieel is. Als belangrijkste is men het er ook over eens dat niet één enkele aanpak die in alle situaties geschikt is.. Dit gezegd hebbende, kan eenvoudige versie-identificatie voldoen aan de behoeften van veel gebruikers, en is het van grote waarde gebleken voor ontwikkelaars van vocabulaires. Voor veel gebruikers kan "menselijk leesbare" versie-informatie minstens zo belangrijk zijn als "machine leesbare" versie-informatie. Vanuit de W3C wordt aangedrongen om de menselijke bruikbaarheid te beschouwen als een kritische factor bij het kiezen van een versiebeheer strategie [3].Tevens is voor veel toepassingen het toegang hebben tot de laatste versie van een dataset genoeg. Maar dit geldt niet in alle gevallen. Toepassingen kunnen bijvoorbeeld toegang nodig hebben tot zowel de oude en de nieuwe versie(s) om synchronisatie en/of integratie mogelijk te maken van autonoom ontwikkelde (maar onderling verbonden) datasets. Versiebeheer moet dus niet alleen de archivering van en de toegang tot verschillende versies behelzen, maar zou ook in staat moeten zijn om verschillende soorten zoekopdrachten te ondersteunen over de

1 https://www.w3.org/2006/07/SWD/Vocab/principles#Versions

Horaplantsoen 18, 6717 LT Ede
Postbus 37, 6710 BA Ede
T 0318 69 53 00
E info@crow.nl
www.crow.nl
IBAN NL63INGB0667211446
BIC INGBNL2A
BTW NL823697733B04
KVK 41150929

gegevens heen. Met inbegrip van zoekvragen (query's) die toegang hebben tot meerdere versies (cross-versie query's), query's die toegang hebben tot de evolutiegeschiedenis (delta) zelf, evenals combinaties van het bovenstaande.

Versiebeheer

De eenvoudigste vorm van versiebeheer is de gegevens opslaan als verschillende snapshots (versies) van een dataset (lees: volledige materialisatie, Engels: *full materialization*). Dit is een 'document gebaseerde' manier van versiemanagement welke in de LD wereld pas waardevol wordt als deze een URI toegewezen krijgen [3]. Alternatieve manieren omvatten een deltabenadering (Engels: *delta-based approaches*), het gebruik van geannoteerde triples (tijdsannotaties, Engels: *temporal annotations*) en hybride benaderingen (Engels: *hybrid approaches*) die de bovenstaande technieken combineren. Al deze opties kunnen de behoeften m.b.t. versiebeheer ondersteunen, maar verschillende benaderingen blinken uit in verschillende aspecten of behoeften. Zo kunnen deltabenaderingen bijvoorbeeld snel antwoord geven op vragen over de evolutiegeschiedenis van de gegevens, maar mogelijk niet efficiënt zijn bij vragen cross-versie [2]. De verschillende strategieën worden hieronder toegelicht.

Drie+ verschillende strategieën

In de internationale literatuur en semantic web-gemeenschap worden drie strategieën voor versiebeheer onderscheiden, plus een aantal hybride oplossingen.

- *Full Materialization / Independent Copies*
  Full Materialization was de eerste en is de meest gebruikte aanpak voor de opslag van verschillende versies van datasets. Met behulp van deze strategie worden alle verschillende versies van een dataset expliciet opgeslagen. Het belangrijkste nadeel is schaalbaarheid met betrekking tot opslagruimte (en daarmee het aantal triples): aangezien elke versie in zijn geheel wordt opgeslagen, is de informatie die ongewijzigd gebleven is gedupliceerd (eventueel meerdere malen). Hier tegenover staat dat het bevragen over versies heen meestal efficiënt is, omdat alle versies al in het 'archief' zijn opgenomen [2].
  Deze techniek wordt ook wel de '*Independent Copies* (IC)-aanpak genoemd, omdat het aparte instanties creëert van datasets voor elke wijziging of set van wijzigingen [1].
- *Deltabenadering / Change-Based*
  Deltabenadering is een benadering waarbij voor elke nieuwe versie alleen de set van wijzigingen met ten opzichte van de vorige versie (lees: de delta) moet worden gehandhaafd. Deze behoeft minder ruimte, omdat delta's (typisch) veel kleiner zijn dan de dataset zelf. De op delta's gebaseerde strategie heeft als nadeel dat het meer rekenkracht kost, bijvoorbeeld voor het on-the-fly reconstrueren van de gehele set [2]. In bijna alle gevallen wordt de benadering geoptimaliseerd door minimaal één volledige versie van de dataset op te slaan en de delta's daarop vast te leggen (waarmee) het eigenlijk al een hybride strategie is.
  Deze techniek wordt ook wel de Change-Based (CB) benadering genoemd, omdat hiermee de gegevens over veranderingen tussen de versies opslaat [1].
- *Geannoteerde triples / Timestamp-Based*
  De geannoteerde triples-benadering gaat uit van het annoteren van een gehele triple met zijn tijdelijke geldigheid. Dit bestaat dan meestal uit twee tijdstempels die bepalen wanneer de triple werd gecreëerd en verwijderd. Hiermee kan de versie van de dataset op elk moment gereconstrueerd worden door alle triples te bevragen die geldig waren op een bepaald tijdstip. Een alternatief hiervan is i.p.v. datums een versienummer gebruiken, waarbij de versie de geldigheidsgegevens bevat [2].
  Deze techniek wordt ook wel de Timestamp-Based (TB)-benadering genoemd, omdat deze de tijdelijke geldigheid van triples opslaat [1].
- Hybride strategieën combineren de bovenstaande strategieën om de voordelen te benutten en de nadelen te elimineren. Meest gebruikt is combinatie van de Full Materialization- en de Deltabenadering (zoals al vermeld hierboven). Hierbij worden verschillende (maar niet alle) versies expliciet gematerialiseerd, terwijl de rest slechts impliciet wordt opgeslagen door de overeenkomstige delta's. Een andere combinatie is het gebruik van de Deltabenadering en geannoteerde triples. Bepaalde systemen slaan opeenvolgende delta's op, waarin elke triple wordt geannoteerd met een versie [2].

Aan versies zitten uiteraard informatiebehoeften van de gebruikers. Deze zijn hetzelfde voor al deze strategieën. Binnen de W3C gemeenschap lijkt er namelijk een vrij brede overeenstemming te bestaan over de noodzaak van bepaalde basiskenmerken van metagegevens m.b.t. versies. Dit zijn aanmaakdatum, auteur, geldige tijd (d.w.z. gegarandeerde up-tijd of automatische vervaltijd), herkomst (voornamelijk in de vorm van eenvoudige IRI's), en de mogelijkheid om deze vier uit te breiden door middel van aanvullende, arbitraire RDF-gecodeerde metadata. Ook wordt er gesuggereerd dat basisrelaties tussen ontologieversies, inclusief opvolger- en voorgangersrelaties, ook belangrijk zijn [2].

Toepassing en bijbehorende query's

De keuze voor een strategie voor versiebeheer hangt uiteraard af van de toepassing (lees: use case). Versiebeheer dient een doel en dit doel kan afgeleid worden naar een functionaliteit waarin de techniek moet voorzien. Deze functionaliteit kan in eerste instantie platgeslagen worden naar vragen (lees: zoekopdrachten of query's) aan de dataset. Een gebruiker heeft namelijk een vraag die hij beantwoord wil zien, door data uit de dataset. Deze vragen bestaan er in verschillende vormen, zoals o.a. uiteengezet in verschillende vakliteratuur [1] [2].

Query's worden o.a. onderscheiden op hun **doel** en **focus**:

- Vragen die *op* versies gesteld worden, ofwel informatie *uit* een versie willen.
  De focus is hierbij de *data* zelf: "Wat staat er in deze versie?";
- Vragen die *tussen* versies gesteld worden, ofwel informatie wat er veranderd is (evolutiegegevens zijn hier voor nodig).
  De focus hierbij is de *delta*: "Wat is het verschil tussen deze versies?";
- Vragen die *naar* versies vragen, ofwel informatie over in welke versie iets staat.
  De focus hierbij is de informatie over de versie (*versie informatie*): "In welke versie komt dit voor?".

Naast dit doel/focusgebied worden query's onderscheiden op hun '**timeframe**':

- Vragen die gesteld worden aan de *laatste versie*;
- Vragen die gesteld worden aan *eerdere* versie(s);
- Vragen die gesteld worden aan *alle* versies (zowel de laatste als eerdere).

Verder kunnen query's nog ingedeeld worden in hun '**type**':

- *Materialisatie* query's vragen om alle data: "Geef me alle data, uit die versie of die delta";
- *Single* versie query's vragen om deel van de data uit één versie (doormiddel van beperkingen of filters): "Geef me dat specifieke deel van de data, uit die versie of die delta";
- *Cross* versie query's vragen om een deel de data uit alle versie die beschikbaar zijn: "Geef me dat specifieke deel van de data, uit alle versies of delta's";

Een laatste categorisering kunnen query's bij een bepaald 'query **atoom**' geschaard worden. Deze zijn in de vakliteratuur geïntroduceerd als fundamentele (niet nader af te pellen) query patronen [1]:

- Versie materialisatie (VM): verkrijgt een antwoord in data door een query te stellen aan één versie. *"Welke boeken waren er beschikbaar in de bibliotheek gisteren?"*;
- Delta materialisatie (DM): verkrijgt een antwoord als deltagegevens door een query te stellen aan de delta data tussen twee versies. *"Welke boeken zijn teruggebracht of uitgeleend tussen gisteren en nu?"*;
- Versie query (VQ): verkrijgt een antwoord door een vraag te stellen aan het hele archief (alle versies) met als resultaat in welke versies het antwoord klopt. *"Wanneer was boek X aanwezig in de bibliotheek?"*;
- Cross-versie join (CV): voegt de resultaten van twee data query's op twee versies samen. "*Welke boeken waren aanwezig in de bibliotheek gisteren en vandaag?*";
- Change materialisatie (CM): geeft als antwoord een lijst van versies waarin de gestelde vraag leidt tot opeenvolgende verschillende resultaten. *"Op welke dagen is boek X uitgeleend of teruggebracht"*.

Bovenstaande beschrijvingen zijn niet uitputtend. Combinaties kunnen gemaakt worden, bv. een query die toegang vraagt tot een delta, evenals meerdere versies.

Query's kunnen zodoende gecategoriseerd worden op verschillende 'assen'. De meest gebruikte query's m.b.t. versies worden hieronder opgesomd en daarna ingedeeld in de verschillende categorieën:

- "Moderne versie-materialisatiequery's" (MVMQ) vragen om een volledige actuele versie van de huidige versie. Voorbeeld: Geef me alle informatie over alle boeken die op dit moment in de bibliotheek staan.
- "Moderne single-versiequery's" (MSVQ) worden aan de huidige versie gesteld. Voorbeeld: Hoeveel boeken staan er op dit moment in de bibliotheek.
- "Historische versiematerialisatie query" (HVMQ) betreft het vragen om een volledig versie uit het verleden. Voorbeeld: Geef me alle informatie over alle boeken die gisteren in de bibliotheek stonden.
- "Historische single-versiequery's" (HSVQ) bevragen binnen één versie uit het verleden. Voorbeeld: Hoeveel boeken stonden er gisteren in de bibliotheek.
- "Delta-materialisatiequery's" (DMQ) vragen om een volledige delta op te halen. Voorbeeld: Welke boeken zijn teruggebracht of uitgeleend tussen 18 januari en 26 februari?;
- "Single-deltaquery's" (SDQ) zijn query's die in twee opeenvolgende versies worden uitgevoerd. Voorbeeld: Welke boeken zijn teruggebracht of uitgeleend tussen gisteren en nu?
- "Cross-deltaquery's" (CDQ) gaan over vragen binnen verschillende wijzingen van de dataset. Voorbeeld: Hoe vaak is boek 'x' uitgeleend, dan wel teruggebracht?
- "Cross-versiequery's" (CVQ) gaan over vragen op verschillende versies van de dataset, waardoor informatie in meerdere versies moet worden opgevraagd. Voorbeeld: Welke boeken waren aanwezig in de bibliotheek eergisteren, gisteren en vandaag?;
- "Versie info query's" (VIQ) gaan over vragen naar versies in het gehele archief. Voorbeeld: Wanneer was boek 'x' aanwezig in de bibliotheek?;
- "Versie info change query's" (VICQ) gaan over vragen van verandering tussen twee verschillende opeenvolgende versies. Voorbeeld: Op welke dagen is boek 'x' uitgeleend of teruggebracht?

In de onderstaande tabel staan de bovengenoemde query's ingedeeld in de eerder genoemde categorieën. Er is te concluderen dat er enorm veel verschillende combinaties te maken zijn en dat de use case(s) dus erg bepalend zijn voor de strategie die er gekozen wordt. De gekozen strategie bepaald uiteindelijk welke publicatiewijze en techniek er het best gekozen kan worden.

Query	Doel	Focus	Timeframe	Type	Atoom
MVMQ	Op versie	Data	Laatste versie	Materialisatie	VM
MSVQ	Op versie	Data	Laatste versie	Materialisatie	VM
HVMQ	Op versie	Data	Eerdere versies	Materialisatie	VM
HSVQ	Op versie	Data	Eerdere versies	Single	VM
DMQ	Tussen versies	Delta	Alle versies	Materialisatie	DM
SDQ	Tussen versies	Delta	Alle versies	Single	DM
CDQ	Tussen versies	Delta	Alle versies	Cross	DM
CVQ	Op versie	Data	Alle versies	Cross	CV
VIQ	Naar versies	Versie info	Alle versies	Cross	VQ
VICQ	Naar versies	Versie info	Alle versies	Cross	CM

Publicatiewijze

Naast de strategie en de functionaliteit is er nog een belangrijke factor te onderscheiden, namelijk de publicatiewijze. Hiervoor zijn grofweg 5 manieren [4].

1. RDF-bestanden gehost door een webserver;
2. RDF ingebed in andere documentformaten op het web;
3. LD-views bovenop databases;
4. LD-interfaces naar RDF-datasets die zijn opgeslagen in triple stores;

5. LD-wrappers[2] om bestaande applicaties of web API's heen.

1. RDF-bestanden gehost door een webserver
Bij het werken met een relatief klein aantal triples is de eerste wijze een eenvoudige, bestandsgeoriënteerde aanpak met lage overhead. RDF-bestanden kunnen handmatig worden bewerkt met een teksteditor of worden geëxporteerd vanuit een softwaretool. De bestanden worden vervolgens gepubliceerd door ze te uploaden naar een webserver die ze onder een bepaalde URI en met de juiste content type (lees: MIME-type). Deze aanpak wordt vaak gebruikt om persoonlijke FOAF-profielen of RDF-vocabulaires te publiceren [5].

2. RDF ingebed in andere documentformaten op het web
Wijze nummer twee laat RDF-beschrijvingen ook in andere documentformaten op het web worden ingebed. Op deze manier wordt vermeden dat afzonderlijke bronnen moeten worden onderhouden voor een computer-interpreteerbare RDF-beschrijving naast een grafische weergave die bedoeld is voor menselijke gebruikers. Ingebedde RDFa[3] kan zich bijvoorbeeld bevinden in statische HTML-bestanden of dynamische pagina's die door een contentmanagementsysteem worden gegenereerd.

3. LD-views bovenop databases
Nummer drie gaat over het presenteren van reguliere databases met een LD-perspectief. Hiervoor zijn een aantal talen en hulpmiddelen beschikbaar [6]. Bepaalde software zorgt er bijvoorbeeld voor dat de inhoud van een relationele databases als LD geraadpleegd kan worden [5].

4. LD-interfaces naar RDF-datasets die zijn opgeslagen in triple stores
Publicatiewijze vier wordt gezien als de meeste cleane manier en maakt gebruikt van pure LD. RDF-datasets die in triplestores worden onderhouden, kunnen vaak automatisch worden aangeboden als LD-interfaces. Triplestores zijn speciaal gebouwde RDF-databases die beschikken over hun eigen querytaal genaamd SPARQL. Gebruikers kunnen dan via een zgn. SPARQL-endpoint zoekvragen stellen en deze worden vervolgens beantwoord in een gestructureerde vorm [5].

5. LD-wrappers om bestaande applicaties of web API's heen
Ten slotte kunnen wrappers rond bestaande toepassingen of web-API's gegevens omzetten in LD. Meestal is dit 'middleware' dat LD uit verschillende, niet LD, bronnen kan genereren. Veel van deze oplossing zijn opensource omdat LD een open format is, dat dus niet aan software van bepaalde partijen gebonden is [5].

Al deze publicatiewijzen zijn correcte manieren om LD te ontsluiten. Ze hebben allemaal voor- en nadelen. Deze worden hier niet verder beschreven. Er wordt wel verder ingegaan op de implicaties voor versiebeheer met zo verschillende publicatiewijzen. Hoewel het bij LD niet verplicht is om versies te maken, erkennen datasetbeheerders de waarde van regelmatige snapshots, bijvoorbeeld voor hersteldoeleinden en retrospectie.

Implementatie in techniek

Er bestaan verschillende technieken om de verschillende versiebeheerstrategieën te ondersteunen. Waarschijnlijk de meest bekende zijn [5]:

1. Gebruik maken van een versiebeheersysteem ('version control system' VCS)
2. Periodieke dumps delen van dataset of database
3. Het beschrijven van de delta (in een RDF-ontologie) en meepubliceren
4. Versiebeheer binnen triplestores
5. Gebruik maken van bestaande versiemanagementservices

[2] In de informatica is een wrapper een entiteit die een ander item inkapselt (omwikkelt). Wrappers worden gebruikt voor twee primaire doeleinden: om gegevens om te zetten naar een compatibel formaat of om de complexiteit van de onderliggende entiteit te verbergen door middel van abstractie [9].
[3] https://www.w3.org/TR/rdfa-core/

1. Gebruik maken van een versiebeheersysteem
Meestal worden bij RDF-bestanden met versiebeheer wijzigingen bijgehouden met behulp van een VCS zoals CVS, Subversion, of Git. Dit zijn beproefde tools in de software-engineering, voornamelijk voor het versioneren van tekstbestanden (in het bijzonder broncode). Wanneer RDF in een geschikt formaat geserialiseerd wordt, kan zo'n VCS goed worden gebruikt voor het versioneren van RDF. Dit werkt het best als er gekozen wordt voor RDF syntax die werkt met een vaste regelindeling en een stabiele ordening. Dit is namelijk hoe VCS bestanden verwerken. Echter, vanwege de verschillen tussen RDF-semantiek en sequentiële aard van tekstbestanden, kan het versiebestand snel groter worden dan de dataset zelf. Dit is niet efficiënt qua opslag en het bevragen hiervan. Dit is dus vooral haalbaar voor kleine datasets. Daarnaast wordt het probleem van het ontdekken dat er een nieuwe versie is en toegang daartoe niet persé opgelost. Sommige versiebeheersystemen hebben een webinterface om verschillen tussen bestanden in te zien (zoals Github), maar wat de relatie is tussen de huidige staat en een eerdere versie wordt niet uitgedrukt [5].

2. Periodieke dumps delen van dataset of database
Daar staat tegenover dat data in een database of triplestore meestal direct gemodificeerd wordt en overschreven of verwijderde data niet bewaard wordt. Datamanipulatie door bijvoorbeeld SQL of SPARQL Update, wijzigen data en terugdraaien kan meestal niet. Waarschijnlijk is de meest gebruikelijke strategie in zo'n geval het doen van een periodieke dump gepubliceerd volgens een naamgevingsschema. Meestal met een versie-ID of datum gecodeerd in het bestandspad en/of de bestandsnaam. Dit kan online zijn (dan noemen we het URL's) of offline. Deze dumps zijn doorgaans monolithische exporten van de volledige dataset, wat resulteert in grote downloads, zelfs als men alleen geïnteresseerd is in individuele dingen. Vooral bij het werken met meerdere datasets zorgt dit voor een veel werk bij het ophalen van versies, terwijl het hele punt van LD het verbinden van verschillende LD-sets met elkaar is. Ook hier geldt dat er geen enkele manier is om te vertellen hoe modificaties worden geïdentificeerd en hoe ze hebben met elkaar te maken hebben, zonder de gebruikte naamgevingsconventies te kennen. Deze conventies zijn ook niet gestandaardiseerd. Hiervoor worden pogingen ondernomen middels het bedenken van 'URI-strategieën' in een afspraken stelsel binnen een bepaald domein, bijvoorbeeld in de Nederlandse NTA8035[4] of de W3C[5]. Maar dit biedt geen oplossing voor alle use cases. Soortgelijke overwegingen gelden voor LD-wrappers bij web-API's. Ze genereren gewoonlijk LD op basis van de huidige status van de dataset in de onderliggende gegevensbron en zijn meestal niet versiebewust. Als er snapshots worden genomen van wrapper-gegenereerde bronnen, worden ze meestal ook gepubliceerd in de vorm van dumps, bijv. webcrawlers en monitoringtools [5].

3. Het beschrijven van de delta en meepubliceren
Het beschrijven en meepubliceren van wijzigingen bij RDF-datasets is een andere manier om de historie bij het houden. Hiervoor zijn twee verschillenden manieren. Enerzijds het gebruiken van RDF-vocabulaires en ontologieën gemaakt voor het beschrijven van wijzigingen. Anderzijds het gebruik maken van een 'patch'-bestand. Het gebruik van vocabulaires en ontologieën gaat uit van het beschrijven van de wijzigingen in een RDF-bestand, met de RDF-semantiek (het is gebaseerd op Full materialization). Voorbeelden zijn Changeset[6], SemVersion [7] en PROV-O[7]. In het algemeen wordt er gebruik gemaakt van meerdere triples om een wijziging te beschrijven, inclusief de metadata. Een probleem van het gebruik van RDF-vocabulaires en ontologieën voor versiebeheer is dat de hoeveelheid triples die ze opleveren over het algemeen erg omvangrijk zijn. Voor elke kleine verandering wordt een veelheid aan nieuwe RDF-statements gecreëerd. Geen van deze vocabulaires heeft zich ontwikkeld tot een geaccepteerde standaard en wijzigingen beschrijven in RDF is (nog) niet gebruikelijk geadopteerd. Patchbestanden zijn in principe de syntactische tegenhangers van voorgaande, maar zijn wel gebaseerd op de deltabenadering. Ze beschrijven namelijk de verschillen tussen RDF-sets. Bijvoorbeeld RDF Delta[8] stelt een speciaal bestandsformaat voor om de verschillen

[4] https://www.nen.nl/NEN-Shop/ICTnieuwsberichten/Ontwikkeling-NTA-8035-Semantisch-modelleren-in-de-gebouwde-omgeving.htm
[5] https://www.w3.org/2006/07/SWD/wiki/BestPracticeRecipesIssues/ServingSnapshots
[6] https://vocab.org/changeset/schema.html
[7] https://www.w3.org/TR/prov-o/
[8] https://afs.github.io/rdf-delta/delta.html

weer te geven tussen RDF-datasets. Patchbestanden zijn niet zo semantisch rijk als RDF, maar zijn over het algemeen compacter dan voorgaande oplossingen [5].

4. Versiebeheer binnen triplestores
Grote RDF-datasets worden meestal onderhouden in een triplestore. Naast dat de voorgenoemde technieken dan ook mogelijk zijn, zijn er implementaties van triplestores die zelf aan versiebeheer doen. Hiervoor zijn verschillende manieren mogelijk, maar de algemene noemer is dat aan iedere triple een identificatie wordt gehangen. Er is dan sprake van een 'quad'. Zodoende wordt het mogelijk om naar de graaf te refereren. Dit is dus een implementatie van de geannoteerde triple benadering. Bij triples is de vorm namelijk: <*subject*> <*predicaat*> <*object*>. Terwijl er bij een quad sprake is van: <*subject*> <*predicaat*> <*object*> <*context*>. Sommige stores hangen daar versieinformatie aan die niet in RDF beschreven wordt, anderen juist weer wel, Daarnaast zit er verschil in dat sommigen alleen de delta's geïdentificeerd worden, terwijl bij anderen alle triples een identificatie krijgen; in de praktijk resulteert dit vaak in een quad die wordt aangehaald in minimaal vier andere quad/triples, namelijk voor het subject, het predicaat, het object en het type van de wijziging (add/delete/modify). RDF heeft hier een specifieke implementatie van, te weten de named graph. Deze maakt eigenlijk gebruik van de 'context' om er een naam aan te geven (vaak een URI): <*subject*> <*predicaat*> <*object*> <*graaf naam*>. De named graph hoeft echter niet te gelden voor één triple, maar kan gelden voor een set aan triples. Ook deze techniek kan worden gebruikt om versieinformatie te expliciteren.

5. Gebruik maken van bestaande versiemanagementservices
Tot slot, bestaan er een aantal versiebeheerdiensten op het web, zowel voor gewone webpagina's als voor LD. De Wayback Machine[9] archiveert een groot deel van het web, met inbegrip van veel gekoppelde gegevensbronnen. De gebruikers van de site hebben echter alleen maar beperkte controle over wanneer snapshots worden gemaakt. Linked Open Vocabularies[10] en LOD Laundromat[11] werken ongeveer hetzelfde maar dan voor LinkedData. Ze houden ook enkele versies bij van gevonden sets. Deze technieken zijn nog vrijblijvend, maar een dergelijke way-of-working kan ook geïmplementeerd worden voor gepubliceerde datasets.

Combinaties
Een recente, veel belovende, ontwikkeling die eigenlijk twee van de bovenstaande technieken combineert is de RDF* (lees: RDFstar) ontwikkeling [8]. Hierin wordt een alternatieve aanpak voorgesteld die gebaseerd is op het nesten van RDF-triples en van querypatronen. Deze aanpak maakt een compactere weergave van data en query's mogelijk, en is te combineren met de bestaande benaderingen. De basis is om RDF uit te breiden met een mogelijkheid tot genestelde triples. Het wordt mogelijk om triples te maken die metadata over een andere triple representeren door deze andere triple direct als subject of object te gebruiken. Uiteraard hoort hier ook een uitbreiding op SPARQL (SPARQL*) en een syntax (bijvoorbeeld Turtle*) bij. Doordat het dus geannoteerde triples zijn, kan het gebruikt worden om versie-informatie uit te drukken. Hoewel deze oplossing zeer dicht tegen het bestaande LD aanschurkt en weinig aanpassingen vraagt, moet er in de verwerkende LD-software wel aanpassingen worden gemaakt. Nog niet veel triplestores en aanverwante pakketten hebben deze aanpassingen gemaakt, maar het lijkt anno 2019 een vlucht te nemen.

Combinatie mogelijkheden
Een belangrijke afweging is uiteraard het maken van combinaties tussen de strategie, de gevraagde functionaliteit, de wijze van publiceren en de implementatie in de techniek. Uit onderzoek blijkt dat het meeste wel te combineren valt, maar dat er met name in efficiëntie en effectiviteit van opslag en opvragen wat te winnen is. Omdat dit niet persé het onderwerp van dit document is wordt er volstaan met het geven van een kwantitatieve conclusie uit de literatuur [1]: Er is met name correlatie tussen de query atomen en de strategie. VM-query's zijn namelijk efficiënt in opslagoplossingen die gebaseerd zijn op IC, omdat er een indexering op versie plaatsvindt. Aan de andere kant kunnen IC-gebaseerde oplossingen een grote hoeveelheid 'kosten' met zich meebrengen in termen van opslagruimte, omdat

[9] http://waybackmachine.org/
[10] https://lov.linkeddata.es/dataset/lov
[11] http://lodlaundromat.org/

elke versie afzonderlijk wordt opgeslagen. Bovendien zijn DM- en VQ-query's minder efficiënt voor IC-oplossingen. Dat komt omdat voor DM-query's twee volledig gematerialiseerde versies ter plekke moeten worden vergeleken, en VQ vereist dat *alle* versies op hetzelfde moment moeten worden opgevraagd. DM-query's kunnen efficiënt zijn in CB-oplossingen als de query-versie ranges overeenkomen met de opgeslagen deltaranges. In alle andere gevallen, evenals voor VM- en VQ-query's, moeten de gewenste versies on-the-fly gematerialiseerd worden, dat zal uiteraard meer tijd vergen voor grotere delta's. CB-oplossingen vereisen echter meestal minder opslagruimte dan VM indien er voldoende overlap is tussen elke opeenvolgende versie. Tot slot, VQ query's presteren goed voor TB-oplossingen omdat de annotatie van de tijdstempel rechtstreeks komt overeen met het resultaatformaat van VQ. VM- en DM-query's zijn in dit geval minder efficiënt voor IC-benaderingen, vanwege de ontbrekende versie-index. Bovendien, TB-oplossingen hebben minder opslagruimte nodig in vergelijking met VM's als de veranderingen tussen de dataset niet te groot zijn. Samengevat kunnen IC-, CB- en TB-benaderingen goed presteren voor bepaalde query's, maar ze kunnen traag zijn voor anderen. Maar deze efficiëntie gaat meestal ten koste van een grote opslagruimte, zoals het geval is bij een op IC gebaseerde aanpak [1]. Veel gebruikte combinaties zijn degenen in onderstaande tabel.

Query	Atoom	Storage strategie
MVMQ	VM	IC
MSVQ	VM	IC
HVMQ	VM	IC
HSVQ	VM	IC
DMQ	DM	CB
SDQ	DM	CB
CDQ	DM	CB
CVQ	CV	IC
VIQ	VQ	TB
VICQ	CM	TB

Conclusie

De algemene conclusie is dat de keuze voor een versiebeheer strategie altijd maatwerk is, er is geen one-size-fits-all oplossing. Om tot een gedegen discussie te kunnen komen is het belangrijk om overzicht en een gemeenschappelijk begrip te hebben. Het doel van dit document is het uiteenzetten van de mogelijkheden en laten zien wat het speelveld is. Alle oplossingen hebben voor- en nadelen in verschillende situaties. Er is geen oplossing die voor alle gebruiksdoeleinden geschikt is, brede consensus geniet, én voor alle LD-verwerkende systemen geschikt is. Versiestabiliteitsvereisten zorgen mede voor de lage mate van (her)bruikbaarheid van al hetgeen er aan LD is [5]. De genoemde onderdelen moeten voor elke gebruikstoepassing opnieuw worden afgewogen, om tot een versiebeheerstrategie te komen voor een bepaalde LD-dataset. De eerste stap is het bepalen van de use case: wat voor doel dient het versie beheer. Dit kan uitgewerkt worden in query vragen. Deze kunnen vervolgens onder één van de categorieën geschaard worden en met de eerste tabel kan gecheckt worden of de categorisering klopt. Vervolgens kan een bijpassende publicatiewijze en technische implementatie afgewogen worden. Hier is geen vaste keus voor. Zelfs de literatuur geeft verschillende adviezen. Het belangrijkste hier is een onderbouwde afweging maken en deze vastleggen, zodat hier later overwogen op teruggekomen kan worden.

Voornaamste bronnen

[1] R. Taelman, M. V. Sande, J. V. Herwegen, E. Mannens en R. Verborgh, „Triple storage for random-access versioned querying of RDF archives," *Web Semantics: Science, Services and Agents,* vol. 2019, nr. 54, p. 28, 2018.

[2] V. Papakonstantinou, G. Flouris, I. Fundulaki, K. Stefanidis and G. Roussakis, "Versioning for Linked Data: Archiving Systems," in *H2020 project*, EU, 2016.

[3] T. Baker en A. Miles, „Principles of Good Practice for Managing RDF Vocabularies and OWL Ontologies," 16 3 2008. [Online]. Available: https://www.w3.org/2006/07/SWD/Vocab/principles#Versions. [Geopend 6 12 2019].

[4] T. Heath and C. Bizer, Linked Data: Evolving the Web into a Global Data Space, 1e ed., Morgan & Claypool, 2011.

[5] P. Meinhardt, „Versioning Linked Datasets: Towards Preserving History on the Semantic Web," Universität Potsdam, Potsdam, 2015.

[6] F. Michel, J. Montagnat en C. Faron-Zucker, „A survey of RDB to RDF translation approaches and tools," HAL, I3S, 2014.

[7] M. Völkel, W. Winkler, Y. Sure, S. R. Kruk en M. Synak, „SemVersion: A Versioning System for RDF and Ontologies," in *Proceedings of the 2nd European Semantic Web Conference (ESWC-05)*, Heraklion, 2005.

[8] O. Hartig, „RDF* and SPARQL*: An Alternative Approach toAnnotate Statements in RDF," Dept. of Computer and Information Science (IDA, Linköping University, 2017.

[9] P. Christensson, „The Tech Terms Computer Dictionary," TechTerms, 6 Augustus 2019. [Online]. Available: https://techterms.com/definition/wrapper. [Geopend 3 December 2019].