

Dynamische Windows processing in RDF Mapping engines voor data streams

Sitt Min Oo

Promotoren: Prof. Dr. Ruben Verborgh and Dr. Anastasia Dimou
Begeleider: Gerald Haesendonck

Abstract—De huidige state-of-the-art benaderingen voor het mappen van niet-RDF naar RDF data in een streaming omgeving richten zich meer op de efficiëntie van het mapping proces met minimale ondersteuning voor multi-stream verwerking. De bestaande benaderingen voor de ondersteuning van eenvoudige multi-stream processing operatoren in mapping engines zijn zeer beperkt of passen een vaste venster grootte toe.

Daarom hebben we in RMLStreamer een dynamisch venstermechanisme geïmplementeerd, dat de grootte aanpast aan de veranderende kenmerken van de stroom met verwaarloosbare geheugenoverhead, lage latentie en hoge doorvoer. We evalueerden het dynamische venster onder verschillende werkbelastingen met variërende streamsnelheid. De resultaten tonen aan dat het een latentie bereikt in het milliseconden bereik, met een hogere doorvoer dan vensters met een vaste grootte in alle werklast situaties.

Index Terms—RDF, RMLStreamer, RML, Adaptieve vensters, Dynamische vensters, Stream joins, Multi-stream verwerking.

I. INTRODUCTIE

Diverse gegevensformaten, zoals CSV of HTML, zijn niet ontworpen om de semantiek van hun gegevens te bevatten waardoor machines deze gegevens niet automatisch kunnen interpreteren. Om ervoor te zorgen dat deze heterogene gegevensformaten interpreteerbaar en verwerkbaar zijn door machines, zijn gegevensformaten gebaseerd op W3C-normen, zoals Resource Description Framework (RDF) triples [1], worden ontwikkeld. Er bestaan state-of-the-art benaderingen om heterogene data te consolideren en deze te transformeren naar een RDF serialisatie in een streaming omgeving.

Deze benaderingen ondersteunen traditionele stream operatoren zoals joins en aggregaties. Echter, ze houden geen rekening met de karakteristieken van streaming data bronnen zoals snelheid en tijd-correlaties tussen de verschillende invoerstromen, hetzij als gevolg van de vaste grootte van vensters of aangepaste oplossingen die zij toepassen.

Daarom stellen wij een dynamisch venster voor als oplossing voor de variërende kenmerken van streaming data. Wij evalueerden onze benadering met de join operator toegepast binnen het venster. Het dynamische venster verbetert de prestaties van de join operator met hogere doorvoer, lagere latency en vergelijkbaar geheugengebruik vergeleken met een venster met vaste grootte.

De broncode en de code voor de evaluatieopstelling zijn te vinden op https://github.com/RMLio/RMLStreamer/tree/feature/window_joins, en respectievelijk <https://github.com/Kortika/Thesis-test-scripts.git>.

II. VERWANTE WERKEN

Er bestaan state-of-the-arts benaderingen om niet-RDF data te mappen naar RDF data in een streaming omgeving. Deze benaderingen implementeren verschillende strategieën als het gaat om het toepassen van multi-stream verwerking. Verder schetsen we ook de bestaande werken over windowing om veranderende karakteristieken van de input datastroom in de tijd.

A. Mapping implementaties

1) *SPARQL-Generate*: SPARQL-Generate [2] is gebaseerd op een uitbreiding van de SPARQL 1.1 query taal, om gebruik te maken expressiviteit en uitbreidbaarheid te benutten. Het kan worden geïmplementeerd bovenop van elke bestaande SPARQL query-engine. M. Lefrançois [2] verduidelijkte dat bij het samenvoegen van records van twee verschillende streams samenvoegt, SPARQL-Generate de records van de "ouder" stroom en deze eerst intern in het geheugen indexeren, voordat iteratief de "kind" stroom te consumeren voor het samenvoegen.

2) *TripleWave*: TripleWave [3] is gebaseerd op een uitbreiding van R2RML om heterogene gegevens te consumeren en als RDF-gegevens op het web te publiceren. Daarom richt het zich alleen op de mapping van een niet-RDF datastroom naar een RDF datastroom. Hoewel het een eenvoudige *join* operator ondersteunt, heeft het geen dynamisch venster om veranderende karakteristieken van een datastroom te behandelen.

3) *RDFGen*: RDF-Gen [4] is gebaseerd op zijn eigen aangepaste syntaxis die een combinatie is van SPARQL en Turtle syntaxis. Het verwerkt de invoerstromen individueel en maakt daarom gebruik van windowing om multistream operatoren uit te voeren. Echter, omdat de implementatie gesloten broncode is, konden we niet bevestigen of het venster een vaste grootte heeft of dynamisch is.

4) *RMLStreamer*: RMLStreamer [5] werd ontwikkeld om het invoer- en mapping proces van RDF data generatie pijplijn te paralleliseren. Het is gebaseerd op het werk van RMLMapper [6], een RDF mapping engine die begrensd

data consumeert en mapt naar RDF data met gebruik van RML. Daarom kan RMLStreamer ook heterogene gegevens verwerken en RDF-gegevens genereren. Het ondersteunt echter geen multi stream operators op dit moment, maar het gemak van schaalbaarheid dat RMLStreamer biedt is wenselijk als een stream mapping engine.

B. Windows

Er zijn verschillende studies uitgevoerd om join-algoritmen in vensters [7–11] te verbeteren. De benaderingen in [9–11] zijn gebaseerd op op het laten vallen van sommige van de te verbinden records door middel van *load shedding* indien de records niet aan een drempel voldoen. Deze drempelberekeningen vergen enige geheugenoverhead voor het opslaan van een statistisch model per venster of veronderstellen dat de stroom in orde is om te kunnen werken [9–11].

Anderzijds houdt VC-TWJoin [7] alleen rekening met de snelheid van de stroom om de venstergrootte aan te passen. Deze aanpak is echter onstabiel wanneer de gegevensstroom periodieke gegevensuitbarstingen vertoont. De grootte van het venster wordt vergroot of verkleind wanneer de metriek de drempelwaarden van het algoritme overschreden. In het slechtste geval kan dit kan het leiden tot cyclische toename en verkleinen van de venstergrootte als de metriek bij elke update van de metriek.

III. DYNAMISCH VENSTER

Een dynamisch venster is een type gepartitioneerd venster [12]. Wij definiëren het als een groep subvensters, die hun grootte dynamisch aanpassen volgens de kenmerken van de gegevensstroom. Het groepeerde de binnenkomende stromen eerst in verschillende partities, volgens de *key* attribuutwaarde van de records. Vervolgens worden deze gegroepeerde records toegewezen aan individuele subvensters. Deze subvensters passen hun eigen grootte onafhankelijk van elkaar aan bij elke update-cyclus. Bovendien zullen de subvensters onafhankelijk van elkaar zijn, zodat dat de stroomsnelheid lokaal is voor de attribuutwaarde waarvoor elk subvenster verantwoordelijk is.

A. Dynamische venster join

Het gebruikte join-algoritme is een variant van Symmetric Hash Join [13]. De subvensters zelf werken als een hashtable, die records bevat met een specifieke *sleutel* — aangezien ze *gepartitioneerd* zijn. De records worden samengevoegd door de relevante stroom af te tasten en het resultaat te genereren.

B. Dynamisch venster-algoritme

Voor elk subvenster worden de volgende configuratieparameters voorzien:

- Δn : het **initiële** tijdsinterval voor de volgende ontruimingstrigger.
- ϵ_l : de bovengrensdrempel voor de totale kosten metriek.

- ϵ_l : de ondergrensdrempel voor de totale kosten metriek.
- U : de bovengrens voor de venstergrootte.
- L : de ondergrens voor de venstergrootte.

Aangezien wij de join operator implementeren, wordt de trigger-event afgevuurd wanneer de huidige record r_c binnenkomt, en de symmetrische hash join wordt uitgevoerd. We duiden het huidige venster aan als W met de grootte als $|W|$. De stromen worden aangeduid als S_P en S_C met de overeenkomstige toestanden $List_P$ en $List_C$, voor respectievelijk de ouder- en de kindstroom. De toestanden bevatten de records van hun respectievelijke streams in de subvensters.

De ontruimingstrigger wordt afgevuurd telkens wanneer het huidige watermerk $w \geq |W| + \Delta n$. Bij elke ontruimingstrigger berekenen we de kosten voor elke staat van $List_P$ en $List_C$, die de records van S_P en S_C bevatten respectievelijk. De kosten voor $cost(List_P) = |S_P|/Size(List_P)$, idem voor $cost(List_C)$. De totale kosten bedragen $m = cost(List_P) + cost(List_C)$ en worden getoetst aan de drempels ϵ_l en ϵ_u . Indien $\epsilon_l \leq m \leq \epsilon_u$, Δn blijft het gelijk, anders wordt het dienovereenkomstig aangepast. Dit zorgt voor enige stabiliteit in de venstergrootte door dezelfde grootte te behouden als de kosten binnen de drempels liggen. Hoe hoger m , hoe hoger de stroomsnelheid. Het subvenster moet dus vaak worden worden uitgezet en Δn worden verlaagd om het geheugengebruik te verminderen. Er is ook een limiet op de minimum en maximum venstergrootte, respectievelijk L en U , om ervoor te zorgen dat de venstergrootte grootte Δn niet oneindig in grootte blijft groeien of krimpen in een worst-casescenario. De grootte van de lijsttoestanden worden ook geactualiseerd overeenkomstig $Size(List_P) = Size(List_P) * cost(List_P) + 0.5$. Hetzelfde geldt voor $Size(List_C)$. Het dynamische venster handhaaft dus een ideale grootte door Δn , $Size(List_P)$ en $Size(List_C)$ aan te passen aan de stroomsnelheid. De pseudocode voor het uitzettingsalgoritme wordt gepresenteerd in Algoritme 1.

Dit algoritme voor het dynamische venster is een aanpassing van VC-TWWindow [7] met verbeteringen voor stabiliteit in de venstergrootte, duidelijkheid in de voor updates gebruikte metriek, en lokalisatie van de bijwerking van de venstergrootte voor elk subvenster.

IV. IMPLEMENTATIE

Het dynamische venster is geïmplementeerd voor RML-Streamer om de mogelijkheden ervan uit te breiden voor eenvoudige streamverwerking door meerdere streams samen te voegen. Bovendien hebben we RML uitgebreid uitgebreid met nieuwe ontologieën om de configuratie van het venster mogelijk te maken.

A. RML ontologie-uitbreiding

We hebben RML uitgebreid met ondersteuning voor de configuratie van het venster wanneer het samenvoegen van de *reference object map* met zijn *parent triples*

Algorithm 1: Dynamisch venster *onEviction* routine

Data: $\Delta n, \epsilon_u, \epsilon_l, U, L, List_P, List_C, S_P, S_C$

```

1:  $cost(List_P) = |S_P| / Size(List_P)$ 
2:  $cost(List_C) = |S_C| / Size(List_C)$ 
3: totale kosten  $m = cost(List_P) + cost(List_C)$ 
4: if  $m > \epsilon_u$  then
5:    $\Delta n = \Delta n / 2$ 
6:    $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$ 
7:    $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$ 
8: else if  $m < \epsilon_l$  then
9:    $\Delta n = \Delta n * 2$ 
10:   $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$ 
11:   $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$ 
12: beide schoonmaken  $List_C$  en  $List_P$ 
  
```

map. Momenteel ondersteunt het alleen de rudimentaire configuratie van het type verbinding dat moet worden toegepast.

B. Dynamisch venster

Het dynamische venster is geïmplementeerd met behulp van Apache Flink's *KeyedCoProcessFunction* API met het statusbeheer gedaan met behulp van Flink's *ListStates*.

V. EVALUATIE

Om de doeltreffendheid te meten van dynamische windowing voor multi-stream operatoren tijdens de mapping van niet-RDF heterogene gegevens moeten we de volgende metrieken van ons stream processing framework: *CPU gebruik*, *latency*, *doorvoer*, *geheugengebruik*, en *completeheid*.

A. Gegevens

We gebruiken dezelfde gegevens als de benchmark in het artikel van Van Dongen en Van den Poel [14]. De gegevens zijn afkomstig van de NDW (Nationale Databank Wegverkeersgegevens) uit Nederland¹. Het bestaat uit metingen van het aantal auto's en hun gemiddelde snelheid over de verschillende rijstroken op een snelweg. De sensorgegevens werden door een Kafka-publisher naar twee topics doorgespeeld, namelijk *ndwflow* en *ndwspeed*, voor respectievelijk het aantal auto's en de gemiddelde snelheid.

B. Evaluatie setup

Docker² is opgezet met de containers zoals geïllustreerd in figuur 1. De docker-containers draaien op een standalone machine om de invloed van netwerkcommunicatie zo veel mogelijk te beperken. Apache Kafka wordt gebruikt als de berichtenmakelaar voor onze gegevens. De setup voor de Kafka broker is hetzelfde als beschreven in [14].

¹NDW-datasite: <http://opendata.ndw.nu/>

²Docker: <https://www.docker.com>

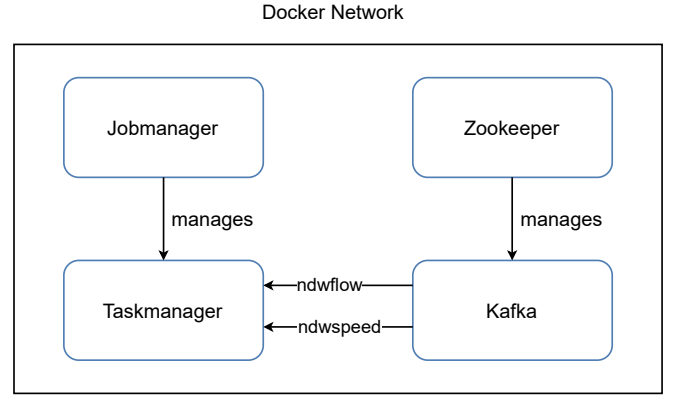


Fig. 1: Setup van applicatiecontainers binnen hetzelfde docker netwerk. De kafka topics *ndwflow* en *ndwspeed* worden verbruikt door de RMLStreamer job binnen de Taskmanager.

C. Metingen

CPU-gebruik, doorvoer, geheugen, en latency blootgesteld door Flink's Rest API, worden constant elke 100ms opgevraagd door een Python script scraper. CPU gebruik, doorvoer en geheugengebruik worden intern gemeten door Flink, terwijl latency meting handmatig wordt geïmplementeerd met behulp van Flink's Metrics API om nauwkeuriger de hoeveelheid tijd te meten tijd die een record in het venster doorbrengt voordat het verwerkt wordt.

De bovengenoemde metrieken worden gemiddeld over de geparalleliseerde window operatoren. Intersection over union (IOU) wordt gebruikt als de metriek om de volledigheid te meten van het samengevoegde resultaat dat door de vensters wordt gegenereerd.

1) *CPU-gebruik*: Voor CPU-gebruik meten we het CPU-gebruik van Taskmanager, aangezien die verantwoordelijk is voor het uitvoeren van de RMLStreamer-code.

2) *Doorvoer*: De doorvoermeting wordt gedefinieerd als het aantal records dat per seconde door de vensteroperator per seconde. Deze wordt gemeten aan de uitvoer van de window operator, aangezien we de uitvoerprestaties van de windowoperators willen meten.

3) *Geheugengebruik*: Vanwege de beperking in granulariteit van metingen in Flink, JVM heap geheugen van de job gebruikt om het geheugengebruik van de window operator. We verwachten dat het geheugengebruik door andere operatoren in de job consistent en laag is over de verschillende evaluatieruns, omdat het stateless operators zijn.

4) *Latency meting*: We meten de latentie door een tijdstempel toe te voegen aan de records voordat ze het venster binnenkomen. Zodra de records verwerkt zijn en na de join worden uitgezonden, wordt het verschil tussen de huidige verwerkingstijd en de bijgevoegde oude verwerkingstijd genomen als de *latency* voor de records.

5) *Voltooidheidsmeting*: Om *begrensde* invoergegevens te genereren voor de static mapping engine, schrijven we

de opgeslagen gegevens van de topics in een bestand op schijf. Deze invoergegevens worden door RMLStreamer verwerkt in de modus voor verwerkingsmodus om de *complete* set triples te genereren.

De gegenereerde output triples van de evaluatie van de vensters en de verwerking van de begrensde gegevens worden gebruikt om de IOU-metrick te berekenen om de gelijkenis tussen de twee outputs te meten.

D. Werklastsценario's

We evalueren onze dynamische vensterimplementatie onder verschillende werklastsценario's. Deze werklasten zijn vergelijkbaar met die welke in [14] zijn gebruikt, voor zover relevant.

1) *Werklast voor latentiemeting*: Meting van de *latency* die alleen wordt veroorzaakt door de vensterimplementaties, vereist dat het stroomverwerkingsraamwerk niet wordt door aanzienlijke *doorvoer* wordt belast. Dus, de Kafka makelaar de records op een zeer lage constante snelheid van ongeveer 400 berichten per seconde voor deze werklast. We noemen dit in de paper *constant stream rate*.

2) *Werklast met periodieke burst*: We evalueren onze implementatie op zijn vermogen om te gaan met onstabiele streaming gegevensbronnen met variërende snelheid. De werklast heeft een constante lage streamsnelheid met af en toe een uitbarsting van gegevens. Daarom worden elke 10 seconden 38 000 records gepubliceerd, wat ongeveer 170 tot 180 ms duurt, aangezien de tijd die nodig is om te publiceren kan afwijken op basis van de last van de Kafka-brokers. We duiden dit in de paper aan als *periodic stream rate*.

3) *Werklast voor de volledigheid*: We zullen twee stream rates gebruiken, constant en periodiek, om de *completeheid* te meten van de resultaten gegenereerd door de verschillende vensters.

VI. RESULTATEN EN DISCUSSIE

De evaluaties van de werklasten zijn meerdere keren uitgevoerd om de resultaten consistent te houden. Voor elke werklast geven we een kort overzicht van de prestatiewinst die door het Dynamische window wordt bereikt.

A. Werklast voor latentiemeting

Voor latency heeft Tumbling venster een mediane waarde van 1915ms, met latentie variërend van 1081ms tot 2624ms (Figuur 2c) meer dan 10× de latentie van Dynamisch venster, dat een mediaan heeft van 57 en variërend van 39 tot 120ms (Figuur 2d). Onze verbetering om de *trigger* gebeurtenis af te vuren telkens wanneer een nieuw record arriveert in het subvenster, maakt het mogelijk Dynamisch venster een latentie van minder dan een seconde bereiken.

Dynamisch venster heeft een gestage doorvoer van ongeveer 17200 records per seconde terwijl Tumbling venster schommelt tussen 12500 en 12800 records per

seconde alvorens zich te stabiliseren op 12750 records per seconde (figuur 2b). Dit is het gevolg van de aanpassing van de venstergrootte op subvensterniveau, waardoor Dynamisch venster op meer records kan wachten met onregelmatige *key* attributen in de stroom, alvorens het subvenster uit te zetten. Tumbling venster daarentegen verwijdert altijd de inhoud van het venster na 2 seconden zonder te wachten op meer records.

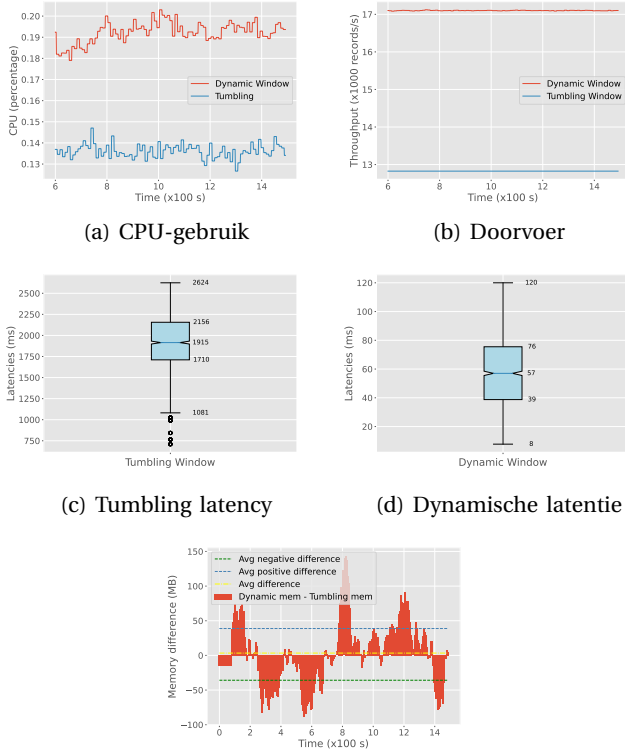
Het relatieve geheugengebruik van Dynamisch venster vergeleken met Tuimelend venster is vergelijkbaar gedurende de levensduur van de evaluatierun (figuur 2e). Dynamisch venster veroorzaakt onregelmatige *pieken* in geheugengebruik van meer dan 100 MB meer dan Tumbling venster op een bepaald punt in de evaluatieduur. Dit kan worden toegeschreven aan de subvensters van Dynamisch venster die groter worden als gevolg van de lage stream snelheid. Echter, Dynamisch venster stabiliseert zich op een meer optimale venstergrootte, waar het minder geheugen gebruikt dan Tumbling vensters. In het slechtste geval gebruikt het net zoveel geheugen als Tumbling venster gedurende de evaluatie.

Het CPU-gebruik is ongeveer 7% hoger voor Dynamisch venster, omdat extra verwerking van de berekening van de metrick nodig is en de doorvoer stijgt tegelijkertijd, waardoor RMLStreamer meer samengevoegde records moet verwerken. (Figuur 2a).

B. Werklast voor periodieke burst

Dynamisch venster verwerkt nog steeds de periodieke burst van gegevens met lagere latency dan Tumbling venster met latency in het bereik van 8ms tot 1669ms (Figuur 4d) vergeleken met Tumbling venster's bereik van 891 ms tot 3904 ms (Figuur 4c) die het dubbele is van de latentie gemeten voor Dynamisch venster. Er is echter een tijdelijke toename in latentie aan het begin (Figuur 3b), wanneer de gegevensstoot elke 10e seconde arriveert. Dit is te wijten aan de aanvankelijke grootte van 2s subvensters voor de aanvankelijk lage streamsnelheid van 400 records per seconde. De grootte van de subvensters begint groter te worden dan 2s vanwege de lage stroomsnelheid. De toename in de subvenster grootte resulteert in het venster meer records te verwerken krijgt, waardoor tegendruk ontstaat en de latentie toeneemt. Dit resulteert in een positieve scheefheid in de latentedistributie voor Dynamisch venster (figuur 4d). Dynamisch venster slaagt er echter uiteindelijk in de subwindow-groottes te verkorten als een aan te passen aan de periodieke uitbarsting van gegevens tot het weer een latentietijd van minder dan een seconde bereikt.

De doorvoer van beide vensters steeg zoals verwacht. Bovendien zien we een groter verschil in de doorvoer tussen de twee vensters van ongeveer 6000 records per seconde (figuur 4b). Echter, de constante en vlakke doorvoermeting stemt echter niet overeen met de resultaten van [14] waar er duidelijke "pieken" zijn in de doorvoermeting die overeenkomen met de periodieke uitbarsting van records die door de vensters worden verwerkt. We



(e) Relatief verschil in geheugengebruik vanuit het oogpunt van een dynamisch venster

Fig. 2: Metingen voor latentie werklast. Dynamisch venster presteert beter in alle metingen met uitzondering van CPU-gebruik als gevolg van overhead in dynamische aanpassing van subvenster maten.

voerden onze evaluatie uit als onderdeel van de gehele RMLStreamer pijplijn, waardoor een lichte tegendruk wat leidt tot een hoge en vlakke doorvoermeting.

CPU gebruik verschil van de vensters, is vergelijkbaar met de werklast voor latentiemeting met relatieve toename voor burst data verwerking (Figuur 4a).

Verrassend, gebruikt het dynamische venster gemiddeld ongeveer 10 MB minder geheugen dan het Tumbling-venster gedurende de evaluatieperiode (Figuur 4e). Het aanvankelijke geheugengebruik voor Dynamisch venster is ongeveer 100 MB hoger dan voor Tumbling venster als gevolg van de lange groei van het venster veroorzaakt door de lage streamsnelheid. Echter, zodra de dynamische aanpassing van subwindow in werking treedt, wordt het geheugengebruik verminderd.

C. Werklast voor volledigheidsmeting

Uit de resultaten in tabel I, en Tabel II, concluderen we dat Dynamisch venster beter presteert dan Tumbling venster wat betreft het genereren van een meer *complete* output. Dynamisch venster heeft een IOU score van 1 voor constante lage stream rate doordat de subwindow groot genoeg wordt groot genoeg worden om alle vereiste records te bevatten, om de *complete* output set te gener-

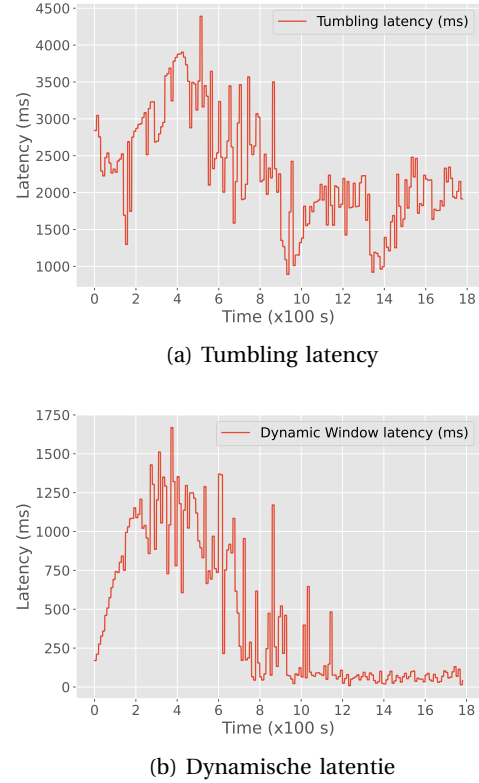


Fig. 3: Latency meting van periodieke werklast gedurende de evaluatieperiode

eren. Daarentegen scoort het Tumbling venster slechts **0.749**, wat leidt tot de conclusie dat een venstergrootte van 2s niet genoeg is om de lage stroomsnelheid van de evaluatiegegevens.

Ook voor periodieke burst-invoer presteert Dynamic window beter dan Tumbling venster met een score van **0.982**, terwijl Tumbling venster slechts **0.780** scoort. De hoge IOU score van Dynamisch venster is te danken aan zijn vermogen om zich aan te passen aan de veranderende stroomsnelheid om genoeg records vast te houden voor maximale gezamenlijke output generatie.

Stream rate	IOU score
Constance snelheid	1
Periodieke uitbarsting	0.982

TABLE I: Dynamisch venster meting van volledigheid.

Stream rate	IOU score
Constance koers	0.749
Periodieke uitbarsting	0.780

TABLE II: Tumbling venster meting van volledigheid.

D. Samenvatting

Samenvattend, tonen deze resultaten aan dat onze implementatie van Dynamisch venster lagere latency, hogere doorvoer, en een meer complete uitvoer dan Tumbling

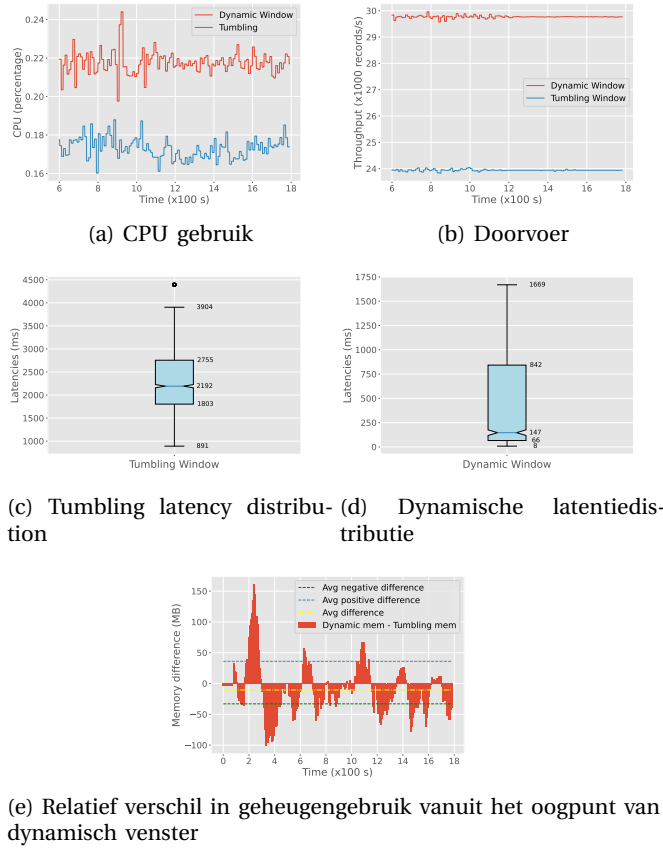


Fig. 4: Metrics metingen voor periodieke werklust. Dynamisch venster heeft gemiddeld lager geheugengebruik dan Tumbling venster voor periodieke werklust.

venster voor zowel werklusten van constante stream rate, en onstabiele periodieke burst stream rate.

Hoewel het geheugengebruik daalde in de werklust met periodieke burst snelheid, konden we niet met zekerheid concluderen dat het Dynamisch venster effectief minder geheugen gebruikte dan het Tumbling venster. De meting was gebaseerd op het heap geheugen van de hele evaluatie job, niet alleen van de window operator. Daarom is er behoefte aan een nauwkeurigere meting van het geheugengebruik.

De resultaten voor doorvoer wijken ook af van die van Van Dongen en Van den Poel(2020) [14]. Dit is te wijten aan het feit dat we de evaluatie hebben uitgevoerd met de volledige pijplijn van RMLStreamer. Hierdoor ontstaat enige tegendruk van de mapping stage, waardoor de doorvoer van de join-fase constant en vlak blijft.

Over het geheel genomen kunnen we concluderen dat Dynamic windowing een goed alternatief is voor de vaste vensters. Vooral in gebruik gevallen, waar vensters niet van vaste grootte hoeven te zijn met een variabele stroomsnelheid.

VII. CONCLUSIE EN TOEKOMSTIG WERK

In deze paper hebben we een aanpak gepresenteerd voor Dynamisch venster die de venstergrootte aan-

past aan de stroomsnelheid van de invoergegevensbronnen. We introduceerden een eenvoudige heuristiek om window-grootte dynamisch aan te passen zonder enorme geheugen- of rekenoverhead.

We hebben ons dynamische venster geïmplementeerd bovenop de bestaande RMLStreamer, om de prestaties ervan in een realistische verwerkingsomgeving te evalueren. We hebben het benchmarkkader zoals beschreven in [14] aangepast om nauwkeurig de prestaties van onze implementatie te evalueren ten opzichte van het standaard Tumbling-venster van vaste grootte.

De resultaten tonen aan dat onze implementatie van Dynamisch venster beter presteert dan Tuimelend venster in termen van latentie, doorvoer, en volledigheid met slechts een kleine toename in CPU gebruik. Hoewel we niet met zekerheid kunnen concluderen dat geheugengebruik lager is in Dynamisch venster, geven onze voorlopige resultaten aan dat het hetzelfde presteert als Tumbling window in het slechtste scenario.

Er zijn dus nog gebieden die voor verbetering vatbaar zijn. Aan de evaluatiekant kunnen we de nauwkeurigheid van onze geheugenmeting kunnen verhogen door alleen het aantal records te tellen in de vensters te tellen in plaats van de hele JVM-heap van de RMLStreamer-taak. Bovendien kan de evaluatie worden uitgevoerd in dezelfde benchmarkpijplijnen als in [14] om de prestaties van het dynamische venster verder te evalueren in een algemeen stroomverwerkingsgeval. Verbeteringen van onze dynamische aanpak kunnen worden bereikt door gebruikers in staat te stellen andere statistische benaderingen te definiëren om de drempel voor het aanpassen van de venstergrootte beter te berekenen.

REFERENCES

- [1] E. Miller, "An introduction to the resource description framework," *Bulletin of the American Society for Information Science and Technology*, vol. 25, no. 1, pp. 15–19, 1998. DOI: <https://doi.org/10.1002/bult.105>. eprint: <https://asistdl.onlinelibrary.wiley.com/doi/pdf/10.1002/bult.105>. [Online]. Available: <https://asistdl.onlinelibrary.wiley.com/doi/abs/10.1002/bult.105>.
- [2] M. Lefrançois, A. Zimmermann and N. Bakerally, "A sparql extension for generating rdf from heterogeneous formats," in *The Semantic Web*, E. Blomqvist, D. Maynard, A. Gangemi, R. Hoekstra, P. Hitzler and O. Hartig, Eds., Cham: Springer International Publishing, 2017, pp. 35–50, ISBN: 978-3-319-58068-5.
- [3] A. Mauri, J.-P. Calbimonte, D. Dell'Aglio, M. Balduini, M. Brambilla, E. Della Valle and K. Aberer, "Triplewave: Spreading rdf streams on the web," Oct. 2016, pp. 140–149, ISBN: 978-3-319-46546-3. DOI: [10.1007/978-3-319-46547-0_15](https://doi.org/10.1007/978-3-319-46547-0_15).

- [4] G. Santipantakis, K. Kotis, G. Vouros and C. Douk-
eridis, “Rdf-gen: Generating rdf from streaming and
archival data,” Jun. 2018, pp. 1–10, ISBN: 978-1-4503-
5489-9. DOI: 10.1145/3227609.3227658.
- [5] G. Haesendonck, W. Maroy, P. Heyvaert, R. Verborgh
and A. Dimou, “Parallel rdf generation from hetero-
geneous big data,” Jul. 2019, pp. 1–6. DOI: 10.1145/
3323878.3325802.
- [6] A. Dimou, M. V. Sande, P. Colpaert, R. Verborgh, E.
Mannens and R. Walle, “Rml: A generic language for
integrated rdf mappings of heterogeneous data,” in
LDOW, 2014.
- [7] Y. Ji, S. Liu, L. Lu, X. Lang, H. Yao and R. Wang, “Vc-
twjoin: A stream join algorithm based on variable
update cycle time window,” in *2018 IEEE 22nd
International Conference on Computer Supported
Cooperative Work in Design ((CSCWD))*, IEEE, 2018,
pp. 178–183.
- [8] M. A. Hammad, W. G. Aref and A. K. Elmagarmid,
“Stream window join: Tracking moving objects in
sensor-network databases,” in *15th International
Conference on Scientific and Statistical Database
Management, 2003.*, IEEE, 2003, pp. 75–84.
- [9] B. Gedik, K.-L. Wu, S. Y. Philip and L. Liu, “Grubjoin:
An adaptive, multi-way, windowed stream join with
time correlation-aware cpu load shedding,” *IEEE
Transactions on Knowledge and Data Engineering*,
vol. 19, no. 10, pp. 1363–1380, 2007. DOI: 10.1109/
TKDE.2007.190630.
- [10] A. Ojewole, Q. Zhu and W.-C. Hou, “Window join
approximation over data streams with importance
semantics,” in *Proceedings of the 15th ACM inter-
national conference on Information and knowledge
management*, 2006, pp. 112–121.
- [11] A. Das, J. Gehrke and M. Riedewald, “Approximate
join processing over data streams,” in *Proceedings
of the 2003 ACM SIGMOD international conference
on Management of data*, 2003, pp. 40–51.
- [12] B. Gedik, “Generic windowing support for extensible
stream processing systems,” *Software: Practice and
Experience*, vol. 44, Sep. 2014. DOI: 10.1002/spe.
2194.
- [13] A. Wilschut and P. Apers, “Dataflow query execution
in a parallel main-memory environment,” Jan. 1992,
pp. 68–77, ISBN: 0-8186-2295-4. DOI: 10.1109/
PDIS.1991.183069.
- [14] G. van Dongen and D. Van den Poel, “Evaluation of
stream processing frameworks,” *IEEE Transactions
on Parallel and Distributed Systems*, vol. 31, no. 8,
pp. 1845–1858, 2020. DOI: 10.1109/TPDS.2020.
2978480.