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 window 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

Heterogene gegevensformaten, zoals CSV of HTML, zijn niet ontworpen om de gegevens die ze weergeven semantisch te verrijken en machines hebben moeite om deze gegevens automatisch te interpreteren. 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 windows 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 syntax. 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 parallelliseren. Het is gebaseerd op het werk van RMLMapper [6], een RDF mapping engine die begrensde 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 windows [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 drempels de drempelwaarden van het algoritme. 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 groepeert 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 vensterverbinding

Het gebruikte join-algoritme is een variant van Symmetric Hash Join [13]. De subvensters zelf werken als een hashtabel, 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_u: de bovengrensdrempel voor de totale kosten metriek.
- ϵ_l: de ondergrens voor de totale-kostenmethode.
- 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 €71$List_P$ en $List_C$, die de records van S_P en S_C bevatten respectievelijk. De kosten voor €17 €, idem voor €18 €. 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 €30. Het dynamische venster handhaaft dus een ideale grootte door $\Delta n, Size(List_P)$ 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-TWindow [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. IMPLEMENTATION

The dynamic window is implemented for RMLStreamer to extend its capabilities to do simple stream processing by joining multiple streams. Furthermore, we extended RML with new ontologies to allow the configuration of the window.

Algorithm 1: Dynamisch raam $onEviction$ routine

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

$cost(List_P) = |S_P| / Size(List_P)$

$cost(List_C) = |S_C| / Size(List_C)$

totale kosten $m = cost(List_P) + cost(List_C)$

if $m > \epsilon_u$ **then**

    $\Delta n = \Delta n / 2$

    $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$

    $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$

else if $m < \epsilon_l$ **then**

    $\Delta n = \Delta n * 2$

    $Size(List_P) = Size(List_P) * (cost(List_P) + 0.5)$

    $Size(List_C) = Size(List_C) * (cost(List_C) + 0.5)$

beide schoonmaken $List_C$ and $List_P$

A. RML ontology extension

We extended RML to support configuration of window when joining the *reference object map* with its *parent triples map*. Currently, it only supports the rudimentary configuration of the type of join to be applied.

B. Dynamic window

The dynamic window is implemented using Apache Flink's *KeyedCoProcessFunction* API with the state management done using 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 *compleetheid*.

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 Nederland [1]. 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 *ndwflow* en *ndwspeed*, voor respectievelijk het aantal auto's en de gemiddelde snelheid.

B. Evaluatie setup

Docker[2] 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].

[1]NDW-datasite: http://opendata.ndw.nu/

[2]Docker: `https://www.docker.com`

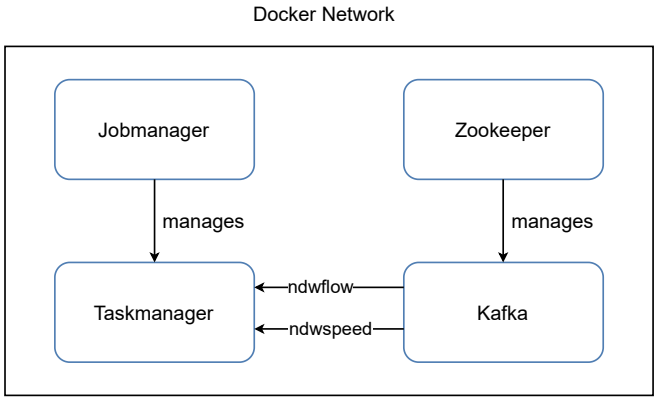


Fig. 1: Setup van applicatiecontainers binnen hetzelfde docker netwerk. De kafka topics *ndwflow* en *ndwspeed* worden verbruikt door de RMLStreamer job binnen de Taskmanager.

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 geparallelliseerde window operators. 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 operators 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-metriek te berekenen om de gelijkenis tussen de twee outputs te meten.

D. Werklastscenario's

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

1) Werkbelasting 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 werkbelasting 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 belasting van de Kafka-brokers. We duiden dit in de paper aan als *periodic* stream rate.

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

VI. RESULTS AND DISCUSSION

The workload evaluations are run multiple times for consistency of results. For each workload we give a brief overview of the performance gain achieved by the Dyanmic window.

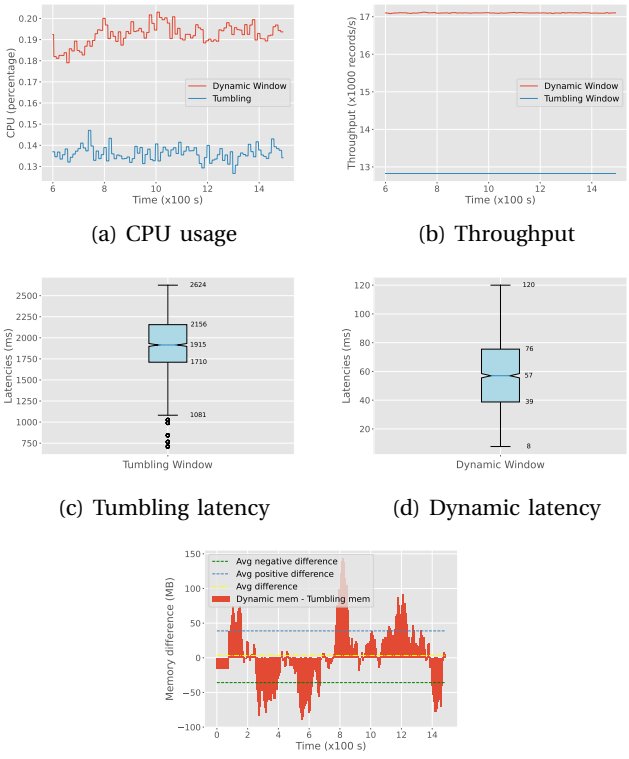
A. Workload for latency measurement

For latency, Tumbling window has a median value of 1915ms, with latency ranging from 1081ms to 2624ms (Figure 2c) more than 10× the lantency of Dynamic window, which has a median of 57 and ranging from 39 to 120ms (Figure 2d). Our improvement to fire the *trigger* event whenever a new record arrives inside the subwindow, allows Dynamic window to achieve sub second latency.

Dynamic window has a steady throughput of around 17200 records per second whereas Tumbling window fluctuates between 12500 and 12800 records per second before stabilizing at 12750 records per second (Figure 2b). This is due to the adjustment of window sizes at the subwindow level, allowing Dynamic window to wait for more records with infrequent *key* attribute in the stream, before evicting the subwindow. In contrast, Tumbling window always evicts the content of the window after 2 seconds without waiting for more records.

Relative memory usage of Dynamic window compared to Tumbling window is similar over the lifetime of the evaluation run (Figure 2e). Dynamic window causes infrequent *spikes* in memory usage of over 100 MB more than Tumbling window at a certain point in the lifetime of evaluation. This can be attributed to the subwindows of Dynamic window growing larger due to low stream rate. However, Dynamic window stabilizes to a more optimal window size, where it uses less memory than Tumbling windows. At worst case, it uses as much memory as Tumbling window does over the course of the evaluation.

CPU usage is higher by around 7% for Dynamic window since it requires extra processing of the calculation of metrics and it increases the throughput causing RMLStreamer to process more joined records. (Figure 2a).



(a) CPU usage

(b) Throughput

(c) Tumbling latency

(d) Dynamic latency

(e) Relative difference in memory usage from the perspective of dynamic window

Fig. 2: Metrics measurements for latency workload. Dynamic window performs better in all measurements with the exception of CPU usage due to overhead in dynamic adjustment of subwindow sizes.

B. Workload for periodic burst

Dynamic window still handles the periodic burst of data with lower latency than Tumbling window with latency in the range from 8ms to 1669ms (Figure 4d) compared

to Tumbling window's range from 891ms to 3904ms (Figure 4c) which is double the latency measured for Dynamic window. However, there is a temporary increase in latency at the beginning (Figure 3b), when the burst of data arrives at every 10th second. This is due to the initial size of 2s subwindows for the initial low stream rate of 400 records per second. The subwindow sizes start to grow **larger** than 2s because of the low stream rate. The increase in the subwindow size results in the window having more records to join; causing a back pressure to form and latency to increase. This results in a positive skew in the latency distribution for Dynamic window (Figure 4d). However, Dynamic window eventually manages to shorten the subwindow sizes as an adaptation to the periodic burst of data until it achieves sub second latency again.

The throughput of both windows increased as expected. Moreover, we observe a bigger difference in the throughput between the two windows of about 6000 records per second (Figure 4b). However, the constant and flat throughput measurement does not agree with the results of [14] where there are clear "spikes" in the throughput measurement which correspond to the periodic burst of records processed by the windows. We ran our evaluation as part of the whole RMLStreamer pipeline, causing a slight back pressure leading to a high and flat throughput measurement.

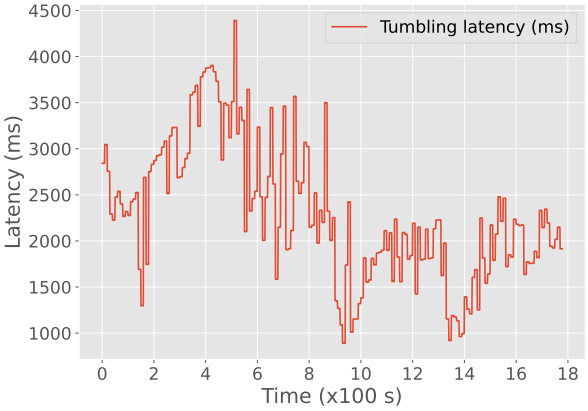
CPU usage difference of the windows, is similar to the workload for latency measurement with relative increase for burst data procesing (Figure 4a).

Surprisingly, Dynamic window uses about 10 MB less memory on average than Tumbling window during the lifetime of the evaluation (Figure 4e). The initial memory usage for Dynamic window is about 100 MB higher than for Tumbling window due to the long window growth caused by the low stream rate. However, once the dynamic adjustment of subwindow kicks in, reducing the memory usage.

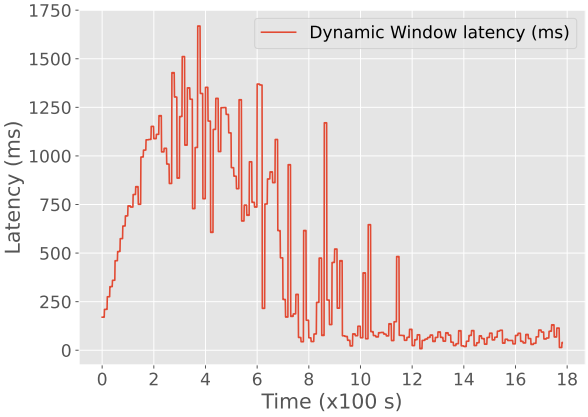
C. Workload for completeness measure

From the results in Table I, and Table II, we conclude that Dynamic window outperforms Tumbling window in terms of generating a more *complete* output. Dynamic window has an IOU score of **1** for constant low stream rate due to the subwindow sizes growing large enough to accommodate all the required records, to generate the *complete* set of output. In contrast, Tumbling window scores only **0.749**, leading to a conclusion that a window size of 2s is not enough to process the low stream rate of the evaluation data.

Similarly for periodic burst input, Dynamic window outperforms Tumbling window with a score of **0.982** whereas Tumbling window only scores **0.780**. The high IOU score of Dynamic window is due to its ability to adapt to the changing stream rate to hold enough records for maximal joined output generation.



(a) Tumbling latency



(b) Dynamic latency

Fig. 3: Latency measurement of periodic workload over the lifetime of evaluation

Stream rate	**IOU score**
Constant rate	**1**
Periodic burst	**0.982**

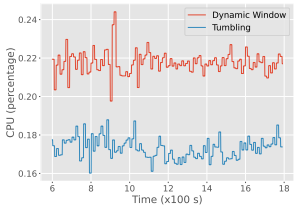
TABLE I: Dynamic window's completeness measurement.

D. Summary

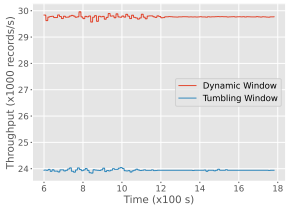
In summary, these results show that our implementation of Dynamic window provides lower latency, higher throughput, and a more complete output than Tumbling window for both workloads of constant stream rate, and unstable periodic burst stream rate.

Although memory usage dropped in the workload with periodic burst rate, we could not confidently conclude that the Dynamic window effectively used less memory on average than Tumbling window. The measurement was based on the heap memory of the whole evaluation job, not just the window operator. Therefore, there is a need for a more precise measurement of memory usage.

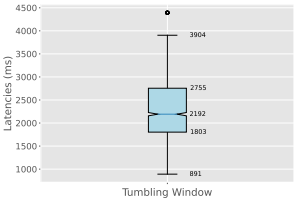
The results for throughput also deviate from those reported by Van Dongen and Van den Poel(2020) [14]. This



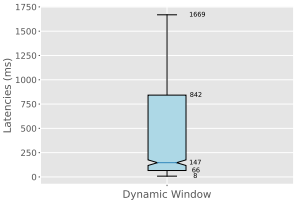
(a) CPU usage



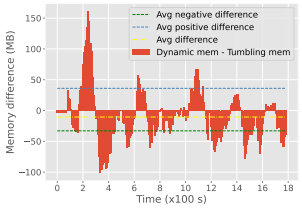
(b) Throughput



(c) Tumbling latency distribution



(d) Dynamic latency distribution



(e) Relative difference in memory usage from the perspective of dynamic window

Fig. 4: Metrics measurements for periodic workload. Dynamic window has lower memory usage on average than Tumbling window for periodic workload.

Stream rate	**IOU score**
Constant rate	**0.749**
Periodic burst	**0.780**

TABLE II: Tumbling window's completeness measurement.

is due to the fact that we ran the evaluation using the whole pipeline of RMLStreamer. This incurs some back pressure from the mapping stage, causing the throughput of the join stage to stay constant and flat.

Overall, we could conclude that Dynamic windowing is viable to replace the fixed size windows. Especially in use cases, where windows are not required to be of fixed size with variable stream rate.

VII. Conclusie en toekomstig werk

In deze paper hebben we een aanpak gepresenteerd voor Dynamisch venster die de venstergrootte aanpast 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.

[4] G. Santipantakis, K. Kotis, G. Vouros and C. Doulkeridis, "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 heterogeneous 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 international 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.