POLITECNICO MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE E DELL'INFORMAZIONE

Executive Summary of the Thesis

DOZER: A Scalable and Fault-Tolerant Streaming Engine for Seraph Queries Evaluation

Laurea Magistrale in Computer Science and Engineering - Ingegneria Informatica

Author: Antonio Urbano

Advisor: Prof. Emanuele Della Valle

Co-advisor: Emanuele Falzone

Academic year: 2020-2021

1. Introduction

We are living in a connected world where data continuously flows and consumers are interested in continuously querying data in order to respond in real-time. With graph databases (e.g. Neo4j), it is possible to efficiently process and query (e.g. via Cypher) highly linked data regardless of the size of the dataset. On the other hand, Velocity combined with Volume introduced by the *Big Data* era limit the capabilities of graph DB's technologies. However, no significant work has been done by the scientific community to extend streams features to property graphs. Seraph, an extension of the syntax and semantics of Cypher, needs a streaming application that creates and maintains time-varying graphs that capture the dynamic evolution of the data flowing inside the query engine. To this purpose, we propose a reference architecture for implementing streaming applications for the Seraph queries evaluations and its implementation Dozer. The latter offers crucial industrial features such as scalability, fault tolerance, high throughput, and low latency, which have been experimentally evaluated. We tested the performance impact of Dozer against the performance of a canonical way of running Cypher queries in Neo4j over temporal marked streaming graphs, as well as Dozer's fault-tolerance. Finally, we discuss some limitations and some suggestions to improve the engine.

2. Problem Statement

Presenting Seraph [3], Falzone et al. assume that an underlying streaming application is required to generate and maintain time-varying graphs that capture the dynamic evolution of the data. The authors proposed a prototype that introduces streaming features in the context of property graph query languages, but with some limits making it not suitable for industrial development. With this as its starting point, we start working on a reference architecture for the implementation of streaming applications for Seraph queries evaluation, which offers crucial industrial features.

3. Requirements

Stream processing systems have emerged in recent years to provide high efficiency combined with high throughput at low latency. We start designing a reference architecture based on a Stream Processing Engine (SPE) that ingests stream (graph) data, performs some maintenance operations to capture the dynamic evolution of the data over time, and communicates with a Neo4j instance for the query execution before reporting the results. According to an architectural design choice, we could guarantee important requirements [4] a real-time processing system should met: 1) Keep the data mooving; 2) Query on Streams; 3) Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data); 4) Generate Predictable Outcomes; 5) Integrate Stored and Streaming Data; 6) Guarantee Data Safety and Availability; 7) Partition and Scale Applications Automatically; 8) Process and Respond Instantaneously.

4. Dozer

The use of a Streaming Processing Engine (e.g. Spark, Kafka Streams, Apache Flink) at the basis of our architecture allows us to exploit built-in mechanisms to satisfy the requirements. Dozer arises as the first implementation of the proposed architecture, built on Kafka and Kafka Streams. Seraph directly supports consumption from Kafka, as well as outputs the resulting stream into Kafka topics. Moreover, the Kafka distributed and parallel processing model allows us to design a microservice architecture and to build a distributed continuous query processing that provides dynamic scalability. Last but not least, Apache Kafka provides Connect API, a free, open-source component for scalably and reliably streaming data between Apache Kafka and other data systems. In particular, pre-built Neo4j Kafka Connector[1], able to ingest and sink graph data with the CDC design pattern, has played a key role in such a decision. Firstly, it allows us to focus on the implementation of the core of the architecture without worrying about the communication between our engine and Neo4j. Moreover, CDC [1] enables the monitoring and collection of data changes, as well as updating a target system with only the data that has changed from the source system. It enables to stream every single event occurring on a database into Kafka at very low latency and low impact. For this reason, it has emerged as an ideal solution to design event-driven architectures that provide real-time of data by moving and processing data continuously as new database events occur. It is ideal for high-velocity data environments where time-sensitive decisions must be taken, since it enables low-latency, reliable, and scalable data replication. Using the *Kafka Sink Connector*, combined with the CDC module, allows us to overcome the limits imposed by the batch nature of Neo4j, which is opposed to the strict latency requirements of stream processing applications.

[1]https://neo4j.com/labs/kafka/

4.1. Dozer Architecture

Figure 1 illustrates an overview of the Dozer architecture. The JSON-PG data format allows us to ingest graph-native data, possibly coming from the major graph databases. However, on the basis of the foregoing considerations, our streaming application needs to work with CDC event streams, and therefore we need a component in charge of converting JSON-PG data format in CDC events, which will become the new input stream of our pipeline. Moreover, the captured data changes denoting the dynamic evolution of the data according to the window operator will be propagated on a Neo4j instance using the Kafka Connector Sink. The Neo4j instance is an external system with which Dozer communicates. Finally, our streaming application runs the Cypher sub-query over the portion of the graphs extracted by the window operator. The result is then published in a dedicated Kafka topic using the JSON-PG format.

4.2. Dozer Topology

Figure 2 better depicts the internal structure of Dozer, consisting of the following modules.

4.2.1 JSON-PG to CDC converter

It consumes from the input stream source topic defined in the Seraph query and converts the data in CDC format. The *Converter Processor* processes each record and converts it from JSON-PG format to CDC format. Finally, a Kafka Producer sends the “create” event record into the appropriate topic that will be the source of the next module.

4.2.2 Delete CDC records producer

The goal of this module is to produce CDC “delete” events. It consumes the CDC “create” records produced by the converter module and

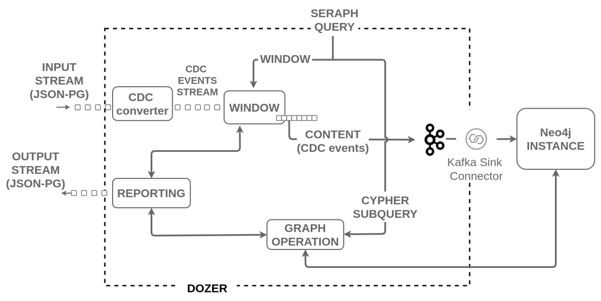


Figure 1: Dozer architecture

produces "delete events" with a proper custom timestamp. To generate the timestamp associated to the "delete" events, we can have two policies which depends on the window type.

Window Time Range. The window time range defines slices of times on which extract portions of the property graph. Let's assume a 2 hours wide time-based sliding window and that a CDC "create" event arrives at 7:52 AM; we need to create a CDC "delete" events at 9:52 AM. Kafka is **pull-based**, i.e. different consumers can consume the messages at different paces. On the other hand, producers cannot decide at which time to send the messages. To capture the dynamic evolution of the time-varying graphs data we need a way to postpone the CDC "delete" events. We decouple the process in two phases. As soon as a "create" events arrives, we produce the corresponding "delete" events with an associated timestamp in the future. Later, a dedicated component will be in charge of consuming this record at the proper timestamp and reproducing it on the Kafka Topic.

Window Event Range. In the case of an *event-based* window, the window range defines the last N events of the streams forming the portion of the property graph to consider. Let's assume an event range of 5 events, this means that we must maintain a sub-graph of the last 5 events. So, once the threshold is exceeded (in our example 5 events), every time a new event arrives we need to remove the oldest of the five and add the most recent one.

4.2.3 Dozer Pipeline

This is the core module of the Dozer engine, according to which a component generates the evaluation time instants, some components are in charge of the maintenance process for capturing the dynamic evolution of the data, and a

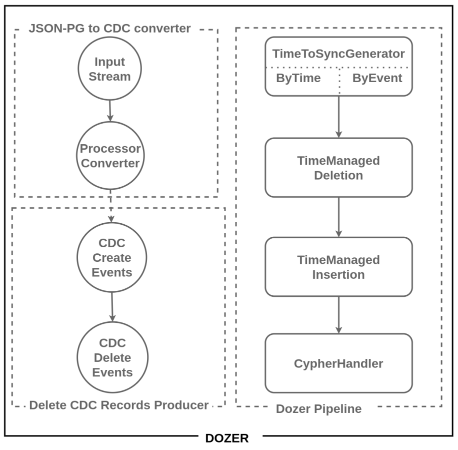


Figure 2: Dozer's modules

component run Cypher subquery on the portion of generated sub-graphs:

TimeToSyncGenerator. Once a Seraph query is registered, the application extracts the *EMIT Range*, to define a *time-to-sync*, which will be the input of the next phases. The *EVERY* operator specifies the frequency of the evaluation process and the *time-to-sync* corresponds to the several exact evaluation time instants. So, assuming that the first record arrives at 8:17 AM and the EVERY operator equals PT2M, the first time to sync is at 8:17 AM, then the second evaluation time will be at 8:19 AM, and so on. From a logical level perspective, a streaming model uses the concept of a *Tick* to drive the system in taking actions over input streams. Botan et al. in [2] define a Tick in three ways:

- *Tuple-driven (DD)*, where each tuple arrival causes a system to react.
- *Time-driven (TD)*, where the progress of the real time causes a system to react.
- *Batch-driven (BD)*, where either a new batch arrival or the progress of the time causes the system to react.

Dozer uses a *tuple-driven* approach. DD models typically provide lower latency than the TD and BD models for the query computation. On the other hand, the record-at-a-time model requires state maintenance for all operators with record-level granularity. This behavior obstructs system throughput and brings much higher latencies when recovering after a system failure. However, Kafka Streams handles out-of-order data, with low latency and high throughput *record-at-a-time processing*. This key feature, combined with the high fault tolerance of Apache Kafka during recovering, made us choose a tuple-driven model, which allows us to reduce the latency impact due to the transactional behavior associated with Neo4j.

TimeManagedComponents. During this phase, the streaming application creates and maintains the time-varying graphs to capture the dynamic evolution of the data flowing inside the query engine up to the *time-to-sync* generated by the previous component. Two decoupled components are in charge of consuming, at the proper timestamp, the "delete" and "create" CDC events produced by the first two modules, allowing to extract the portion of the subgraphs data, which is recreated on the Neo4j by sinking with the connector the CDC events.

CypherHandler. Finally, the engine communicates with the Neo4j instance to run the Cypher subquery on the portion of data and produce the result on the output Kafka topic.

5. Evaluation

All the experiments, which we present, are aimed at showing the industrial features for which this system was designed, such as high performance and fault-tolerance.

5.1. Experimental Setup

The experiments can be divided into three groups, different for their purpose and the information they produce. All the tests have been executed on an Amazon *EC2 t3.2xlarge*[2] instance, and the depicted results correspond to the average results of different independent executions. We repeated the tests for different window sizes (PT1S, PT5S, PT10S, PT15S, PT30S, PT45S, PT1M, PT5M, PT10M) over a time horizon of at most seven days long (from '2021-01-01T00:00:00Z' to '2021-01-08T00:00:00Z'). The following plots are just a portion of the executed tests and aim to describe the system behavior by analyzing the corner case scenarios.

5.2. Dozer vs Cypher Complexity

First of all, we analyze the complexity of running the same queries over the two systems, namely Dozer and querying timestamped Property Graphs with Cypher (hereafter referred to as "Cypher"). For both scenarios and for each selected window, we ran five experiments over a linear dataset simulating a linked list of nodes growing over time. The first node was created with a timestamp corresponding to the date '2021-01-01T00:00:00Z'. Then, every 500ms two new nodes, linked to the existing ones, enter the dataset. In Dozer, we have the system handling the deletion and insertion of relationships at each evaluation instant, selecting the portion of the Property Graph to query according to the window definition. On the other hand, with Cypher, we simulate a tumbling window by running a Cypher *MATCH* query at each evaluation step, filtering over the timestamp property of the relationships. In the latter scenario, the relationships continue to grow over time.

The system's complexity was determined by measuring the effort required by the Neo4j server to execute the *MATCH* query at each evaluation step. For this purpose, we used the Neo4j *ResultSummary* interface to collect the time it took for the server to obtain the query results.

The linear dataset grows linearly over time in the number of nodes and relationships. Assuming that the complexity for running the *MATCH* query depends on the number of nodes and relationships present at each evaluation instant, we expected that Cypher's complexity grows linearly as well. On the other hand, running the same *MATCH* query with Dozer requires a constant time complexity proportional to the number of nodes and relationships shrunk by the windows. However, with Dozer, we handle only the insertion and deletion of the relationships because we are mainly interested in capturing the dynamic evolution of graphs over the relationships which are the key elements of graph DBs. With this assumption, we reduce the maintenance costs at expenses of a slight increase of running Cypher queries. Because of the foregoing consideration, the time complexity analysis led us to the following evaluation. If we consider the time complexity only as function of the number of relationships $\mathbf{n}$, we expect:

- A constant complexity $\mathbf{O(1)}$ for Dozer; and
- A linear time complexity $\mathbf{O(n)}$ for Cypher.

Of course, as time passes, the dependence on the number of nodes cannot be overlooked. The plots in Figure 3 show the results of our experiments, which are coherent with our analysis. Moreover, we can see how the inefficiency

[2] https://aws.amazon.com/it/ec2/instance-types/t3/

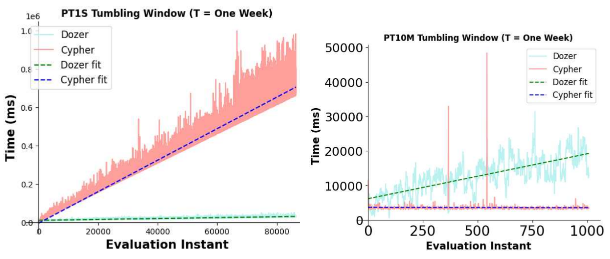


Figure 3: Dozer vs Cypher complexity

of Cypher reduces as window sizes increase until it starts outperforming Dozer (this behaviour starts from 10-minutes-wide window). We are not interested in understanding Neo4j's core execution strategy; however, it is optimized for bulk load ingestion. For larger windows, Cypher creates a large number of nodes and relationships in bulk, after which filtering on them becomes more efficient. On the other hand, with Dozer, we need to handle the maintenance phase. The deletion of some relationships and the accumulation of node instances introduce some inefficiency on the Neo4j server.

5.3. Execution Time Comparison

We also focused our interest on measuring the time impact experienced by the end-user in the two different approaches by comparing the total execution time needed by Dozer and Cypher to run the same query. We can notice that, in both scenarios, the total execution time decreases as window ranges increase, following the trend reported in the previous assessments. Moreover, Dozer outperforms Cypher, regardless of the window size. Of course, continuing to increase it, the result would change, in line with what has already been discussed.

5.4. Comparison among different datasets

In the light of the outcome discussed previously, we repeated some tests on a larger star-shaped dataset to understand if the results depend on whether or not the linear shape. We define a cost function as the cumulative sum of the time it took the Neo4j server to run the query at each evaluation step. Figure 4 refers to the results of the two corner cases window width.
In the star-shaped case, we have greater production frequency (each second, we create ten times the number of nodes and relationships as the

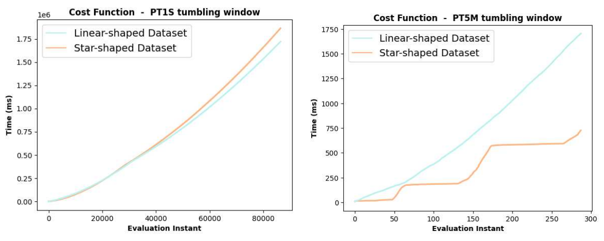


Figure 4: Dozer's cost function at different dataset

linear one) and a more complex structure. Consequently, the star-shaped cost function width will be greater, but it will have the same trend as the linear one. For middle windows, the difference in width is more evident. Indeed, with smaller windows, the behavior is almost similar because, at each evaluation step, Dozer handles a small number of relationships in both scenarios; while continuing to increase the window range, the result would change. Moreover, the plots highlight how the cost function has a **sub-quadratic complexity** due to the node accumulation discussed in the previous phases. Finally, with the star-shaped dataset, we handle more triples. The last plot, referred to as a five-minutes wide tumbling window, shows how Neo4j becomes more efficient with a bulk load (see discussion in Section 5.2).

5.5. Components Overhead Analysis

During all of the previous tests, we measured the portion of time spent by each component. Figure 6 show, for each of them, the overhead percentage w.r.t. the total execution time. The bar-plots highlight the following patterns.
SyncGenerator. its objective is to update the *time-to-sync*. Therefore, its timing is relatively smaller than the other components. Moreover, with window size increasing, the number of evaluation instants decreases, as well as the time spent by the *SyncGenerator*.
TimeManaged. Considering a specific window width, the percentage of the *TimeManagedDeletion* and the *TimeManagedInsertion* components is more or less the same.
CypherHandler. As window sizes increase, its timing decreases at the expense of increasing the time spent in the maintenance phases (both deletion and insertion). The reason is twofold: 1) Increasing the window size, the *TimeManaged* components handle, at each evaluation instant,

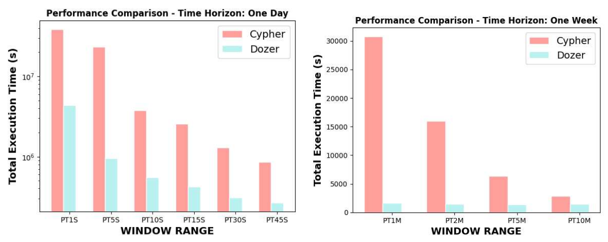


Figure 5: Total execution time comparison

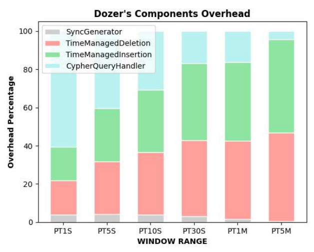


Figure 6: Dozer components' overhead at different window sizes

a larger number of nodes and relationships, requiring higher maintenance costs. 2) Neo4j efficiently manages bulk loads. Consistently with the analysis addressed in the previous sections, the *CypherHandler*'s timing reduces as window width increases.

5.6. Fault-tolerance Tests

We performed a set of tests aimed at evaluating the system failover. We ran, for each window, ten executions with and without failures, and both over a two-hour-long time horizon. For the latter, we simulate some anomalies by forcing the system to restart at regular intervals of ten minutes, having thus 11 failures within each execution. Under the assumption that every single recovery within an execution is independent of and does not influence the timing of the other recoveries, we measure the Dozer's *MTTR*. It ranges from 10 to 12 seconds, and increases as window width decreases. In addition to measuring the time Dozer took for a single recovery, we studied how failures affect end-user usability. The barplot in Figure 7 depicts the time increment due to the 11 recoveries. In contrast to the single recovery analysis, the total execution time increases as window size increases. This is because the engine takes some time before running at full capacity. Wider the windows, closer the evaluation instants at which the system fails and has to recover.

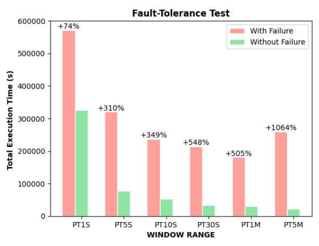


Figure 7: Fault-tolerance tests

6. Conclusions

This thesis aimed at providing a reference architecture for the implementation of streaming applications for the Seraph queries evaluation. The suggested architecture keeps the data moving and achieves low latency by incorporating built-in event/data-driven processing capabilities. Moreover, the use of a Streaming Processing Engine at the basis of the architecture allows us to exploit built-in mechanisms to deal with streams' challenges, such as delayed, missing, and out-of-order data, and to guarantee high availability and fault tolerance. In particular, we presented Dozer, a first implementation of the architecture, built on Kafka and Kafka Streams. With Dozer, we focused on modeling the dynamic evolution of graphs over the relationships, which are the key elements of graph DBs. This assumption allows us to cut maintenance costs by working on the insertion and deletion of the relationships over time without worrying about nodes. The different tests highlighted the performance of Dozer, outperforming the traditional way of querying timestamped Property Graphs with Cypher. However, the nodes accumulation affects the performance, which reduces as window width increases. In addition, dedicated fault-tolerance tests have been carried out to study how Dozer reacts to a failure and how the latter affects the end-user usability.
In conclusion, Dozer satisfies almost all the architectural requirements, being able to provide near-real-time predictable outcomes for high-volume with low latency and high throughput. It has been designed following the microservice architectural pattern to achieve high scalability, efficiency, and speed. However, this paradigm typically requires container management systems (e.g. Kubernetes, Docker Swarm), which, at the moment, have not been integrated.

References

[1] Itamar Ankorion. Change Data Capture efficient ETL for real-time BI. *Information Management*, 15(1):36, 2005.

[2] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[3] Emanuele Falzone, Riccardo Tommasini, Emanuele Della Valle, Petra Selmer, Stefan Plantikow, Hannes Voigt, Keith Hare, Ljubica Lazarevic, and Tobias Lindaaker. Semantic Foundations of Seraph Continuous Graph Query Language. `https://arxiv.org/abs/2111.09228`, 2021.

[4] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

POLITECNICO DI MILANO

Scuola di Ingegneria Industriale e dell'Informazione

Master of Science in Computer Science and Engineering

DOZER: A Scalable and Fault-Tolerant Streaming Engine for Seraph Queries Evaluation

Supervisor: Prof. Emanuele Della Valle
Co-supervisor: Emanuele Falzone

Master Graduation Thesis
Antonio Urbano
920760

Academic Year 2020/2021

“La Terra su cui viviamo
non l’abbiamo ereditata dai nostri padri,
l’abbiamo presa in prestito
dai nostri figli”

A mio nipote Lucas

Ti auguro un futuro di successo,
e che il mio contributo a tale risultato
sia il mio principio ispiratore.

Abstract

We are living in a connected world where data continuously flows and consumers are interested in continuously querying data in order to respond in real-time. With graph databases, it is possible to efficiently process and query highly linked data regardless of the size of the dataset.

Furthermore, in recent years, an increasing number of websites, applications and IoT sensors have generated data streams, which are potentially unbounded sequences of data having a timestamp and arriving in sequential order, one at a time. On the other hand, Velocity combined with Volume introduced by the *Big Data* era limit the capabilities of graph DB's technologies.

However, no significant work has been done by the scientific community to extend streams features to property graphs. At the moment, there exist only few PoC implementations extending CQL to property graphs. Seraph, an extension of the syntax and semantics of Cypher, needs a streaming application that creates and maintains time-varying graphs that capture the dynamic evolution of the data flowing inside the query engine.

Our effort proposes a reference architecture for implementing streaming applications for the Seraph queries evaluations, which offers crucial industrial features such as scalability, high availability, fault tolerance, high throughput, and low latency. Dozer arises as its first implementation to be a valid alternative to a prototype not suitable for industrial development. These features have been experimentally evaluated over the proposed system. We tested the performance impact of Dozer against the performance of a canonical way of running queries over temporal marked streaming graphs. We also performed fault-tolerance testing to evaluate how the system responds to failures. Finally, we discuss some system limitations and possible changes to improve its performance.

Sommario

Viviamo in un mondo connesso caratterizzato da un continuo flusso di dati ai quali gli utenti finali sono interessati per rispondere a query continue in real-time. I graph databases consentono di processare efficientemente dati altamente interconnessi tra loro, indipendentemente dalla dimensione del dataset.

Inoltre, di recente un numero crescente di websites, applications e sensori IoT hanno generato stream di dati, ovvero sequenze, potenzialmente infinite, di dati marcati temporalmente che arrivano in ordine sequenziale, uno alla volta. Tuttavia, volume e velocità introdotte dall'era dei *Big Data*, limitano le capacità dei DB a grafo.

Nonostante ciò, esistono soltanto pochi PoC e nessun lavoro degno di nota è stato sostenuto per estendere le proprietà degli stream ai property graph. Il nostro impegno propone Dozer, una streaming engine scalabile e fault-tolerant, in grado di effettuare valutazioni di query Seraph, un'estensione della sintassi e della semantica di Cypher. Seraph, necessita di un sistema in grado di creare e mantenere grafi al variare del tempo, catturando l'evoluzione dinamica del flusso di dati.

Vogliamo proporre un'architettura di riferimento per l'implementazione di applicazioni streaming per la valutazione di query Seraph, che offra funzionalità industriali cruciali come scalabilità, alta disponibilità, tolleranza ai guasti, throughput elevato e bassa latenza. Dozer, una prima implementazione di quest'architettura, nasce come una valida alternativa ad un prototipo non adatto allo sviluppo industriale. Queste features sono state valuatate sperimentalmente sul sistema proposto. Abbiamo testato l'impatto sulle prestazioni di Dozer rispetto le performance ottenute valutando le queries su grafi marcati temporalmente. Abbiamo testato anche la fault-tolerance, per valutare come il sistema risponde ai guasti. Infine, abbiamo discusso su possibili modifiche da apportare al sistema in modo da migliorarne ulteriormente le performance.

Ringraziamenti

Sembra come se fosse ieri, dopo un viaggio di 12 ore e più di mille chilometri, là ad affrontare quel test di ammissione che rappresentava l'inizio del mio percorso di studi da ingegnere. Ero accompagnato da tante paure e dubbi, non avendo minimamente un'idea chiara di cosa mi sarebbe aspettato.

Queste saranno probabilmente le mie ultime parole scritte da studente del politecnico. Spero, con questa piccola riflessione, di riuscire a riassumere degnamente un percorso iniziato più di cinque anni fa. Un percorso a tratti difficile, caratterizzato da notti in bianco, stress, lacrime, nervosismo e momenti di rabbia. Ma anche gioia, felicità, amicizia, crescita personale ed esperienze uniche e irripetibili.

"*Per Angusta Ad Augusta*" è una frase che ho sempre amato e trovo adeguata a riassumere un percorso universitario e tutto ciò che ne riguarda. Più grande è l'obiettivo, sicuramente più difficile sarà raggiungerlo e più grande è il prezzo da pagare. Ma al traguardo, guardando la strada percorsa, la soddisfazione sarà sicuramente maggiore.

Premesso che non sono mai stato bravo in questo genere di cose, le prossime parole vorrei dedicarle a quelle persone che mi hanno concesso di raggiungere tali risultati, che mi hanno sostenuto nei momenti più ma anche a coloro con cui ho condiviso momenti di gioia. Spero di non dimenticare nessuno e di cercare di essere il più esaustivo possibile, e di sfruttare al meglio questa sezione per esprimere la mia riconoscenza e gratitudine verso tutte quelle persone che mi hanno affiancato durante questi anni.

Inizierei col ringraziare il Prof. Emanuele Della Valle; innanziututto perchè mi ha concesso di lavorare a questo progetto e sopratutto per la sua disponibilità e supervisione durante tutte le sue fasi principali. La sua incredibile dedizione e la sua passione per il suo lavoro sono state per me fonte di ispirazione e motivazione. Voglio rivolgere anche un ringraziamento particolare ad Emanuele Falzone. Ti ringrazio per aver messo a disposizione il tuo progetto di ricerca e per la tua disponibilità, soprattuto nelle fasi finali del lavoro. Ci tengo a sottolineare come ogni chiamata abbia rappresentato per me un'occasione per avere un interessante

scambio di idee e mi sia servita ad acquisire nuove informazioni e a risolvere i vari problemi che ho dovuto affrontare.

Grazie poi a tutti coloro che mi hanno permesso di raggiungere questo obiettivo, standomi affianco durante questi anni. Tra tutti, la mia più profonda gratitudine la voglio rivolgere a tutta la mia famiglia, ma in particolare modo ai miei genitori. Può risultare sicuramente banale, ma innanzitutto devo ringraziarvi per il vostro sostegno economico, senza il quale non sarebbe stato possibile tutto ciò. Ma, seppur di fondamentale importanza, non può essere sicuramente messo a paragone col vostro supporto da un punto di vista personale. Grazie per aver sempre creduto in me, per essere riusciti a motivarmi e a tirarmi sul il morale nei momenti più difficili. Posso assicurarvi che il desiderio e la voglia di rendervi felici ed orgogliosi per ogni vostro sacrificio, sono stati per me la principipale fonte di ispirazione che mi ha spinto a fare sempre meglio e probabilmente il motivo principale che non mi ha mai fatto prendere in considerazione l'idea di arrendermi davanti ad un problema. Spero di essere riuscito nell'intento, e mi aguro che anche in futuro riesca a raggiungere dei risultati per cui possiate essere fieri di me, certo del vostro appoggio in qualsiasi scelta io faccia. Potrei andare avanti giorni interi, forse mesi, a citare ogni vostro singolo sacrificio. Siccome rischierei di dilungarmi troppo, vorrei ringraziarvi per tutto raccontando due semplici aneddoti che penso esprimano nel migliore dei modi tutti i sacrifici che avete affrontato. Grazie mamma per quella volta durante il primo anno che, a fronte di una semplice e banale influenza, hai preso il primo treno disponibile. Ti conosco molto bene e so cosa significava per te salire da sola su quel treno, quindi grazie! Tra tutte le cose, a te papà vorrei ringraziarti particolarmente per quella volta che siamo saliti con il furgone per fare il trasloco e dopo due giorni di intenso lavoro in cui non ti sei mai fermato, hai dovuto affrontare da solo 1200 chilometri ed un alternatore rotto a metà strada. Grazie!

Ringraziando la mia famiglia, un grazie speciale ai miei nonni e a mio Zio Salvatore, che con le sue chiamate quotidiane, seppur lontani mi ha sempre fatto sentire la sua vicinanza. Non posso che concludere col ringraziare mia cognata Simona e mio fratello Marco. Nonostante alcuni litigi tipici tra fratelli, voglio dirti che ti voglio bene e ringraziarvi entrambi per aver messo al mondo Lucas, chiedendovi scusa se magari non riesco a dimostrare tutto il bene che gli voglio. Ah dimenticavo, grazie per le partite a COD che abbiamo fatto, sono state un toccasana e probabilmente una salvezza durante i periodi di quarantena in cui ero da solo a Milano.

Grazie alla mia seconda famiglia, a Giacomo, Matteo, Salvatore, Gabriele, Marco, Jacopo e Francesco. Probabilmente, il sacrifico più grande è stato non

poter condivididere con voi tanti momenti, a causa della distanza. Potrei riempire pagine intere raccontando mille aneddoti (nella stragrande maggioranza, uno più stupido dell'altro) per spiegare il legame che ci unisce. Ci conosciamo da tantissimo tempo ormai (con alcuni di voi forse troppo?) e ne abbiamo condivise di ogni. Grazie per tutte le esperienze fatte insieme e per tutto ciò che abbiamo condiviso e per quello che spero continueremo a condividire. Sarà sicuramente un grande dispiacere non avervi qui al mio fianco in questo giorno particolare, ma oltre alla distanza ci sarà un motivo ben più imporante. Colgo quindi l'occasione per dire grazie anche a Zaira ormai parte integrante di questa famiglia, e sopratutto alla piccola Ginevra. Non so se al momento della presentazione di questa tesi sarai già nata, in tal caso tanti auguri, dal tuo "zio", nonché tuo non padrino, Antonio. Tra questi voglio rivolgere un ringraziamento speciale a Salvatore e Jacopo, che seppur per motivi differenti, sono stati entrambi imporanti durante questo percorso.

A Salvatore un particolare ringraziamento voglio farlo per il mio primo anno di università, in cui è stato fondamentale. Trasferirmi da un paesino di meno di dieci mila abitanti a Milano è risultato molto più semplice grazie a te. Mi hai aiutato ad ambientarmi velocemente, sopratutto nei periodi iniziali in cui, per ovvie ragioni, non conoscevo ancora nessuno. Ricordo ancora la prima frittata di patate di Guglielmo, e soprattutto non potrò mai dimenticare i messaggi che ci inviavamo alla fine di ogni giornata, con le ore di studio affrontate, cercando (a volte invano) di stimolarci a vicenda. Durante il mio quarto e quinto anno di superiori, ero un po' indeciso se intraprendere una carriera universitaria. Devo ammettere che te hai svolto un ruolo fondamentale, "convincendomi" sulla scelta dell'università.

Un ruolo fondamentale sotto questo aspetto è stato svolto anche da Jacopo. Probabilmente sei stato uno dei primi (forse il primo in assoluto) ad appoggiare questa scelta, affermando in diverse circostanze la possibilità di una prospettiva futura migliore, spronandomi e ribadendo più volte che sarebbe stato sprecato non andare all'università e come non ci avresti pensato due volte se fossi stato in me. Ti dico grazie dal profondo del cuore, sei per me come un fratello! E siccome mentre scrivo questo, sta per scendere una lacrima, devo sdramatizzare un po'. Quindi grazie per le risate che mi fai fare con i direct che ci scambiamo giornalmente (si sembra strano anche a me mentre lo scrivo), ma soprattutto voglio farti notare che non sto diventando "Il campione mondiale del Mondo del Massachusetts", ma mi sto semplicemente riconfermando come "Umpa Lumpa della scienza". Voglio chiudere dicendoti che comunque mi sarebbe piaciuto festeggiare con te la mia laurea qui a Milano, meglio nota come "Casarano senza la campana". Già mi immagino una serie di citazioni di Aldo, Giovanni e Giacomo del tipo:

"- E cosi domani ti laurei. - Si ma niente di serio.", o ancora
"Anto, a te te lo devono dare la lode, e se non te lo danno è perchè è tutto un magna magna".
Il tutto intervallato da una sfilza di "Lenta, Lentissima e Ciiippp". Chiudo col ricordarti che l'anno prossimo Vienna ci aspetta, e spero non succederà come con la caramella e non mi costringerai a dirti *"M'eri promesse addhe cose".*

Poi una piccola parentesi vorrei dedicarla a tutti quelli che, amici e non, hanno messo in dubbio l'utilità dell'università. Spero un giorno di dimostrare che non ci sia nulla di più lontano dalla realtà che frasi del tipo "È solo una perdita di tempo. Sprecare cinque anni per poi andare a lavorare in un fast food", e magari riuscire a convincerli del contrario. Certo non sarà la mia priorità far cambiare loro idea, ma voglio comunque cogliere l'occasione per ringraziarli. Tutto ciò mi è comunque servito a dimostrare a me stesso le mie potenzialità, cercando di rendere sempre al massimo.

Un ringraziamente finale voglio dedicarlo a tutte le persone conosciute grazie al politecnico. Grazie a tutti i miei compagni di corso, senza i quali questi anni non sarebbero stati gli stessi. Le partite a carte durante le pause pranzo, la birra dopo le intense e a volte deprimenti giornate studio, tutte le serate, le vacanze fatte insieme ed ogni singola esperianza condivisa con voi, mi hanno fatto vivere ogni giornata con più leggerezza e sono sicuro che mi porterò sempre con me il ricordo di questi momenti passati insieme. Grazie a Manuel, la prima persona che ho conosciuto qui al poli. Grazie al sig. Ciccio Sacchi, a tutte le giornate studio passate insieme durante la laurea triennale e a tutti i dubbi spesso con lui condivisi e prontamente sciolti da Robba. Grazie al Vieni e al sommo Raffa, e a tutti i dibattiti e scambi culturali e socio-politici con loro affrontati, paragonabili ai più importanti salotti letterari Ottocenteschi. Grazie ad Ale Stolfo e alle mitiche partite di calcio piene di colpi alla Holly e Benji. Grazie a Vincenzo, per avermi dato la possibilità di conoscere un molisano e spero che, qualora dovesse andare male la mia carriera da ingegnere, venga assunto ufficialmente come suo interprete personale. Grazie ad André, con cui siamo rimasti grandissimi amici nonostante il cambio di corso e con cui ho condiviso le ultime ore di studio in università dedicate alla stesura di questa tesi. Un grazie speciale va poi a Giacomo, con cui ho condiviso tantissimi momenti, e soprattutto senza il quale la mia laurea triennale sarebbe stata in forte dubbio. Grazie per avermi aiutato a superare un esame a me ostico. Per tutto il periodo di preparazione dell'esame sei stato disponibile a risolvere ogni minimo dubbio ti ponessi. Te ne sarò per sempre grato. Infine, tra i ringraziamenti riservati ai compagni di università, rivolgo una menzione particolare ad Enrico. Definirti semplicemente come compagno universitario sarebbe troppo riduttivo. Abbiamo condiviso tanto, tutti i progetti universitari (per cui dovresti essere te a ringraziarmi, gli unici 30 che hai preso durante la tua carriera) ma anche tante esperienze extra-universitarie, sei per me come un fratello.

Un grazie generale a tutti i compagni di corso conosciuti durante i vari progetti e i diversi gruppi studio improvvisati dal nulla in aula vuota, in cui ti sentivi meno solo quando qualcuno esprimeva i tuoi stessi dubbi, le tue stesse perplessità e paure.

Concludo col ringraziare i miei conquilini passati Sophien, Francesca, la Lulla, e il mio coinquilino attuale Giammy ed in generale tutte le persone che ho conosciuto in questi anni, anche al di fuori del contesto universitario, in quanto ogni singola esperienza è stata comunque possibile grazie al Poli. E tra le varie esperienze non posso non citare l'erasmus fatto a Berlino, che rimarrà una delle esperienze più belle della mia vita. Mi ha fatto crescere sotto diversi aspetti, e soprattutto mi ha arricchitto facendomi conoscere gente proveniente da ogni angolo del mondo.

Spero di aver nominato tutti e di non aver dimenticato nessuno. Chiudo ribadendo per un'ultima volta, un grazie generale a tutti voi per avermi affiancato in questo percorso unico e per avermi reso la persona che sono!

Contents

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Nowadays we live in a digital era where a high amount of data, coming from many different sources, like online websites, social media platforms, IoT sensors, is produced at a significant rate. The exponential growth of data produced by non-traditional sources has meant that streaming data becomes a core component of data architectures.

Moreover, we live in an interconnected world. This means that there are no isolated pieces of information, but rich, connected domains all around us. Graph databases, regardless of dataset size, excel at managing linked data and complex queries. Indeed, relational databases require expensive JOIN operations to compute relationships at query time. Only a database that natively supports relationships can efficiently store, process, and query connections. In a native graph database, accessing nodes and relationships is an efficient, constant-time operation that allows exploring millions of connections in a matter of seconds. We can distinguish two popular models of graph databases:

- *RDF[1] graphs:* conform to a set of W3C [2] standards designed to describe statements and are ideally suited to capturing complex metadata and master data. They are commonly used in the context of linked data, data integration, and knowledge graphs. They can represent complex concepts in a domain, or provide rich semantics and inference on data.
- *Property graphs:* used to represent relationships among data and allow query and data analytics based on them. A property graph is made up of vertices that provide comprehensive information about a subject and edges that rep-

[1]https://www.w3.org/RDF/
[2]Worldwide Web Consortium

resent the link between the vertices. Some attributes, called properties, can be associated with nodes and edges.

Neo4J[1] can be considered the most well-known and widely used graph database in the world. It uses the labeled-property graph model, which requires efficient storage, fast traversal, and querying connected data. For this, Neo4j is suitable for many common use cases, such as Social Network Analytics, Fraud Detection, and Analytics Network Monitoring.

Property Graphs have gotten a lot of interest from the scientific community in recent years. However, no significant work has been done to extend CQL to property graphs at this time. Cypher, in particular, is a declarative graph query language that allows expressive and efficient data querying in Neo4j. As against all these advantages, Cypher lacks the features for dealing with streams of (graph) data and continuous query evaluation. Volume combined with Velocity, two of the seven V's characterizing Big Data according to [52], limit the capabilities of property graph databases such as Neo4j.

Starting from the Big Data general question: *"How to tame 5'Vs?"* and focusing on *Continuous Query Language* world and *Stream Processing* technologies, Falzone proposes Seraph [22] as an extension of the syntax and semantics of Cypher with the goal of dealing with streams of (graph) data and continuous query evaluation. This new proposed query language requires the use of a stream application capable of creating and maintaining time-varying graphs by capturing the dynamic evolution of the data flowing inside the query engine. Presenting Seraph, the authors proposed a working prototype for evaluating Seraph queries, which is not suitable for industrial scenarios.

In this document, we present a reference architecture for Seraph queries evaluation and its first implementation, Dozer[2]. The proposed architecture introduces crucial industrial features such as scalability, high availability, fault-tolerance, high throughput, and low latency. The contributions of my thesis can be summarized as follows:

- We present Dozer and its design, a valid alternative to the first prototype aimed at overcoming the limitations of the latter.
- We provide deep performance insights by experimentally evaluating the Dozer's components overhead, as well as the performance impact of the proposed solutions against a state-of-art algorithm.
- We experimentally test the fault-tolerance of our system

[1]https://neo4j.com/
[2]https://github.com/openseraph/SeraphEngine

The document is structured as follows:

- **Chapter 2 - State of the Art:** It presents background knowledge needed for a complete understanding of the problem and its solution. It also contains an overview of research areas related to this thesis. Finally, it provides backgrounds about some state-of-the-art tools on top of which Dozer is designed.
- **Chapter 3 - Problem Statement:** describes the problem in depth, outlining an existing solution and its limitations.
- **Chapter 4 - Proposed Approach:** we present Dozer and its architecture, describing the path that led us to the Dozer implementation.
- **Chapter 5 - Implementation experience:** provides a technical overview of the proposed system with some code snapshots to highlight core components implementation.
- **Chapter 6 - Experimental Results:** goes through the experimental evaluation of our work to assess Dozer performances.
- **Chapter 7 - Conclusions and Future Work:** summarizes our work by retracing the main stages and discussing the conclusions. Finally, we propose some directions for future improvements.

Chapter 2

State of the Art

In this chapter, we provide an overview of research areas related to this thesis. Section 2.1 describes the reason why, in recent years, we move from SQL to NoSQL as well as introducing the main types of NoSQL solutions. In 2.2 we focus on graph databases definition and in Section 2.3 we present a list of notable graph databases with an overview of their main characteristics. Section 2.4 provides a brief description of the most used graph query languages, focusing in Section 2.4.1 on Cypher query language and its extensions. Section 2.5 introduces CQL, a SQL-based declarative language for registering continuous queries. In Sections 2.5.1 and 2.5.2, we analyze continuous query languages in the context of Graph DBs. Finally, Section 2.6 presents backgrounds about some powerful, state-of-the-art engines and tools on top of which Dozer is designed and built.

2.1 Moving from SQL to NoSQL

The relational databases or RDBMSs have been the dominant model for database management since they were developed. Nevertheless, in recent years with the advent of the digital era, the NoSQL movement has exponentially grown. The term NoSQL was used for the first time by Carlo Strozzi in 1998 to name his lightweight open-source relational database [47] that did not expose the standard SQL interface. The term was reintroduced in 2009 by Eric Evans [21], an employee of Rackspace, at a conference organized by Johan Oskarsson, a developer at Last.fm, to discuss "open-source distributed, non-relational databases". This acronym, which stands for "Not Only SQL", was not used in terms of opposition to the relational databases, but to describe a movement as the whole point of seeking alternatives is that you need to solve a problem that relational databases are a bad fit for. NoSQL is widely spread as a solution to tame the "3Vs" introduced by Douglas Laney [33] in his first Big Data model:

- Volume: the size of the generated data continues to grow - ranging from Tera to Zettabytes - but not as much as our tools' ability to process it.
- Variety: Big Data involves storing structured, semi-structured, and unstructured multimedia data (text, graphics, images, audio, and video). Extract information is not more straightforward as in traditional databases
- Velocity: Data moves fast and we must be able to process and analyze data streams in real-time as the data is gathered.

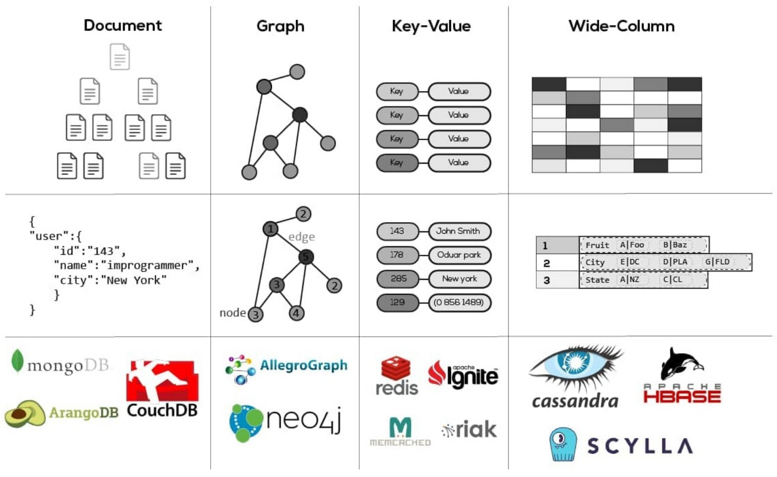
According to [9] the reasons why NoSQL allows us to deal with the problems Big Data has brought with it can be summarized as follows:

- Volume: Query execution times increase as the size of tables and the number of JOINs grow and so it can be difficult to deal with large datasets stored in relational databases. With NoSQL, we can avoid the so-called 'JOIN pain' at the expense of less expressivity. Moreover, NoSQL databases are horizontally scalable w.r.t. to vertical scaling in SQL in which database processes are comparatively more time-consuming and expensive, as the data size increases.
- Variety: NoSQL systems have a schema-less data model which allows handling large volumes of structured, semi-structured and unstructured data with high flexibility by modeling the data according to application requirements.
- Velocity: We can define two problems associated to it:
  - The rate at which the data structure changes, both in terms of rapid change of specific datapoints and change of the data model itself.
  - Variations in data velocity coupled with high volume require a database able to handle write loads.

  NoSQL databases address both data velocity challenges by optimizing for high write loads and by having more flexible data models.

2.2 Graph Databases

According to [36] Graph databases model is one of the core NoSQL technologies along with Key-value stores [41], Column-family stores [23], Document stores [40].

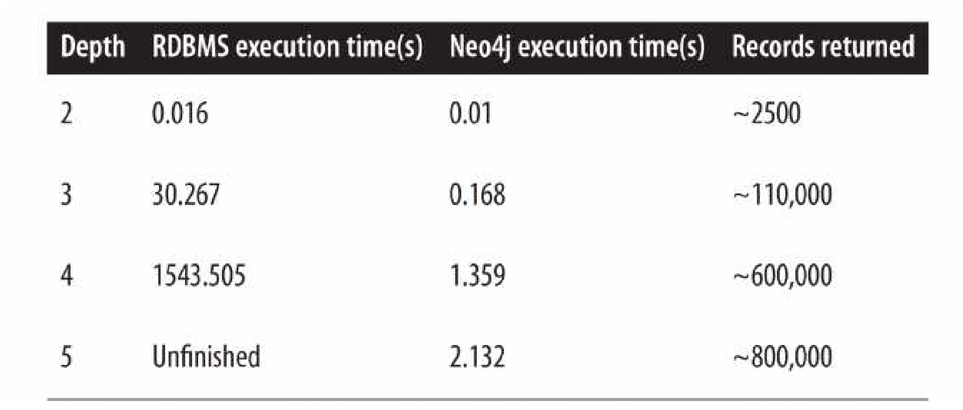


```
{
"user":{
    "id":"143",
    "name":"improgrammer",
    "city":"New York"
    }
}
```

Figure 2.1: Main NoSQL database models

A GDBMS - Graph Database Management System, often known simply as a Graph Database, is an online database management system that exposes a graph data model using CRUD (Create, Read, Update, and Delete) operations [15]. A Graph Data Model is a model in which data structures are represented as graphs, data manipulation is described using graph-oriented operations (i.e., a graph query language), and proper integrity constraints are defined over the graph structure. Formally, a graph consists of a pair $G = (V, E)$, where V is a set whose elements are called vertices, and E is a set of paired vertices, whose elements are called edges. Graph databases are especially suitable in all those scenarios where the information is natively in the form of a graph. Social networks, computer networks and data center management, recommendation systems, and geospatial applications are just some examples of real-world application fields. Relationships are first-class citizens of the graph data model. Graph databases store data-relationships as relationships and so are able to store, process, and query connections efficiently. Accessing nodes and relationships in a native graph database is an efficient, constant-time operation that allows exploring millions of connections in a matter of seconds.

Relational databases, ironically, perform badly with relationships. For years, developers have been attempting to handle linked, semi-structured data using relational databases. In traditional relational databases, as well as the other NoSQL (Not Only SQL) stores, we have to infer connections between entities using foreign keys or out-of-band processing such as map-reduce. As outlier data multiplies and the overall structure of the dataset becomes more complex and less uniform, the relational model becomes afflicted with large join tables, sparsely populated rows, and a lot of null-checking logic. In their book [55], Partner and Vukotic performed



Depth	RDBMS execution time(s)	Neo4j execution time(s)	Records returned
2	0.016	0.01	~2500
3	30.267	0.168	~110,000
4	1543.505	1.359	~600,000
5	Unfinished	2.132	~800,000

Figure 2.2: Performance comparison between relational databases (RDBMS) and Neo4j over data relationships (source: [55])

an experiment between a relational database and a graph database (Neo4j). In Figure 2.2 the results of their experiment in which they used a social network to find friends-of-friends connections to a depth of five degrees using a dataset of 1,000,000 people each with approximately 50 friends.

According to the applications domain, we can decide to adopt one of the two popular models of graph databases: property graphs and RDF graphs. The first one focuses on analytics and querying, while the second one emphasizes data integration.

2.2.1 RDF Graph DB

RDF stands for Resource Description Framework[1], which is a W3C[2] standard for data exchange on the Web. It is a graph data model for publishing semantically enriched information on the Web. They are ideal to model complex data and metadata. They are commonly utilized in the context of linked data, data integration, and knowledge graphs. They can represent complex concepts in a domain or provide rich semantics and inference on data. RDF can also be used with the Web Ontology Language (OWL)[3] to implement reasoning on this data.

RDF, originally designed as a data model for metadata, became very famous in the early 2000s with the publication of the article [49], in which the authors presented their vision of the Internet, in which people would publish data in a structured format with well-defined semantics in a fashion that agents could consume

[1]https://www.w3.org/RDF/
[2]https://www.w3.org/
[3]https://www.w3.org/OWL/

and share. It was designed to be an exchange model that prioritized interoperability between the semantic web and agents. RDF models facilitate information interchange by providing a way to publish data in a standardized format with well-defined semantics. RDF graphs use all of this information to generate a metadata layer that helps define whether different names refer to the same object, if the objects are correlated, and even if different items can be used interchangeably due to their similarities. For this reason, RDF graphs have been widely adopted by pharmaceutical companies, government statistics agencies, and healthcare institutions. A RDF store, also called triplestore, is optimized for the storage and retrieval of *triples*. At the core of RDF is this notion of a triple, which is a statement consisting of three elements representing two vertices connected by an edge. It's typically known as *subject-predicate-object*:

- The *subject* corresponds to a resource or to a node in the graph;
- the *predicate* represents an edge, or a relationship; and
- the *object* can be either another node or a literal value.

In RDF models, resources (vertices/nodes) and relationships (edges) are identified by a URI, or Unique Resource Identifier, which is a unique identifier. This means that neither nodes nor edges have an internal structure; they are simply labels. This is one of the main distinctions between RDF and labeled property graphs. Another reason why RDF are widely used on the web is the possibility to define the triples by using standardized XML syntax[1]. It is important to remark that XML is a possible syntax for RDF, not a component of RDF, i.e. RDF data model is an abstract, conceptual layer independent of XML.

2.2.2 Property Graph database

A property graph is made up of vertices that provide comprehensive information about a subject and edges that represent the link between the vertices. Most of the current graph database systems have been designed to support the property graph model. A property graph is a directed labeled multigraph with the following characteristics:

- It contains nodes, often used to represent entities, or to represent other domain components, depending on the use case
- Nodes can be labeled with zero, one or more labels. A label is a named graph construct that is used to group nodes into sets. All nodes labeled with

[1]https://www.w3.org/TR/rdf-syntax-grammar/

the same label belong to the same set. This allows writing easier and more efficient queries which work on a small portion of the whole graph.

- It contains named and directed edges that connect two nodes. They always have a start and end node (which cannot be an empty node). From a data modeling point of view, edge represents a relationship between entities
- Nodes and Relationships can contain properties, i.e. name-value pairs of data used to store relevant information about the entity/relationship they are associated with. Properties can support most standard data types

A formal definition of the Property Graph Model:

> ***Definition.*** Let $\mathcal{O}$ be a set of *objects*, $\mathcal{L}$ be a finite set of *labels*, $\mathcal{K}$ be a set of property *keys*, and $\mathcal{N}$ be a set of *values*. We assume these sets to be pairwise disjoint. A *property graph* is a structure $(V, E, \eta, \lambda, \nu)$ where:
>
> - $V \subseteq \mathcal{O}$ is a finite set of objects, called *vertices*;
> - $E \subseteq \mathcal{O}$ is a finite set of objects, called *edges*;
> - $\eta : E \rightarrow V \times V$ is a function assigning to each edge an ordered pair of vertices;
> - $\lambda : V \cup E \rightarrow \mathcal{P}(\mathcal{L})$ is a function assigning to each object a finite set of labels (i.e., $\mathcal{P}(\mathcal{S})$ denotes the set of finite subsets of set $\mathcal{S}$); and
> - $\nu : (V \cup E) \times \mathcal{K} \rightarrow \mathcal{N}$ is a partial function assigning values for properties to objects, such that the object sets V and E are disjoint (i.e. $V \cap E = \emptyset$) and the set of domain values where ν is defined is finite.

Property graph models are used in several real scenarios in which the main aspects are analytics and querying. For example in Fraud Detection and Analytics Network Monitoring, where we can use a graph to model transactions between consumers as well as information they share (e.g. the email addresses, passwords, addresses) or use a graph to model how nodes are connected over an IT network. Or even in Social Network Analytics scenarios, where graph databases can be used to simulate the social networks storing users' information and their relationships.

The main differences between the two presented models can therefore be summarized as follows:

1. **RDF does not uniquely identify instances of relationships of the same type**

In RDF it's not possible to uniquely identify instances of a relationship, i.e. it's not possible to have connections of the same type between the same pair of nodes because that would represent exactly the same triple.

2. **In RDF instances of relationships cannot be qualified**
   Since in RDF it's not possible to identify unique instances of relationships, it is also impossible to qualify them or to assign attributes to them.

3. **RDF can have multi-valued properties, while the Labeled Property Graph can have arrays**
   In RDF you can have multi-value properties, i.e. triples where the subject and predicate are the same but the object is different. In the labeled property graph, instead, you have to use arrays.

4. **Different application domains**
   RDF was intended as an exchange model that put interoperability between semantic web and agents in the first place. On the other hand, Label Property Graph model is mainly used to represent data as a graph with specific focus on efficiency, fast query end traversal.

2.3 Notable Graph Databases

In the following we provide a list of notable graph databases with an overview of their main characteristics and the used query language [56]. We mainly focus on presenting Neo4j, but for the sake of completeness, we also list some remarkable graph databases. Query languages will better described in Section 2.4

2.3.1 Neo4j

Neo4j[1] is an open-source, NoSQL, native graph database that provides an ACID-compliant transactional backend with native graph storage and processing. It is implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint, or through the binary "bolt" protocol. Neo4j is referred to as a native graph database because it efficiently implements the property graph model down to the storage level. This means that in Neo4j, everything is stored in the form of an edge, node, or attribute. Each node and edge can have any number of attributes. Both nodes and edges can be labelled. The database uses pointers to navigate and traverse the graph. In contrast to graph processing or in-memory libraries, Neo4j also provides full

[1]https://neo4j.com/

database characteristics, including ACID transaction compliance, cluster support, and runtime failover. It is considered the most popular and widely used property graph database in almost all industries, including financial services, government, energy, technology, retail, and manufacturing. Its main features are:

- *Native Graph Processing:* it is a schema-free database specifically designed to store and manage graph data;
- *High speed:* it provides constant time traversal, regardless the data size, for both depth and breadth due to efficient representation of nodes and relationships. This enables scale-up to billions of nodes on moderate hardware;
- *Fully transactional:* it is ACID(Atomic, Consistent, Isolated and Durable)-compliant to ensure data integrity. It is OLTP-oriented and so not so efficient for whole-graph analysis;
- *Flexibility:* flexible property graph schema that can adapt over time, making it possible to materialize and add new relationships later to shortcut and speed up the domain data when the business needs change;
- *Scalability:* it provides highly performant read and write scalability without specifying data integrity;
- It uses *drivers* for popular programming languages, including Java, JavaScript, .NET, Python, and many more.

2.3.2 Other graph databases

In this section, we present a list of well-known and widely used graph DBs, as well as some of their key features.

- **RedisGraph**
  It is a Redis module developed by Redis Labs[1] to add graph database functionality to the Redis database. It is an in-memory, queryable Property Graph database which uses sparse matrices to represent the adjacency matrix in graphs. This guarantees a fast and efficient way to store, manage and process graphs, making it significantly faster than comparable graph databases.
- **Eclipse RDF4J**[2] Its is an open source modular Java framework for storing, querying, and analysing RDF data. It offers a set of easy-to-use APIs

[1]https://redis.com/
[2]https://rdf4j.org/

that can be connected to all the main RDF storage solutions for highly scalable storage, reasoning, and retrieval of RDF and OWL. The main database solutions that implement the RDF4J APIs are:

  - *RDF4J Memory Store:* a transactional RDF database using main memory with optional persistent sync to disk. It is fast with excellent performance for small datasets.
  - *RDF4J Native Store:* a transactional RDF database using direct disk IO for persistence. It is a more scalable solution than the memory store, with a smaller memory footprint, and also offers better consistency and durability.
  - *RDF4J ElasticsearchStore:* an experimental RDF database that uses Elasticsearch[1] for storage.
  - *Third party database solutions:* RDF4J-compatible databases are developed by several third parties, both open-source/free and commercial, and they often offer better scalability or other extended features. RDF4J fully supports the SPARQL query (see Section 2.4.2) and update language for expressive querying and offers transparent access to remote RDF repositories using the exact same API as for local access.

- **SAP HANA**[2]
  It is a column-oriented in-memory database that acts as a single system, storing and retrieving data as requested by applications. This allows to process massive amounts of data with near zero latency, query data in an instant. SAP HANA is an OLTAP system, since it combines OLAP and OLTP operations into a single system. It supports graph database capabilities.

- **Amazon Neptune**[3]
  A fast, reliable, fully managed graph database build for cloud. It is used as a web service and is part of Amazon Web Services (AWS)[4].The core of Amazon Neptune is a purpose-built, high-performance graph database engine optimized for storing billions of relationships and querying the graph with milliseconds latency. It is ACID (Atomicity, Consistency, Isolation, Durability) compliant and it provides highly availability. Amazon Neptune supports both Property Graph and W3C's RDF models, and their respective query languages Apache TinkerPop Gremlin and SPARQL.

[1]https://www.elastic.co/
[2]https://www.sap.com/products/hana.html
[3]https://aws.amazon.com/it/neptune/
[4]https://aws.amazon.com/

- **CosmosDB**[1]
  It is Microsoft's proprietary globally distributed, multi-model, schema-free, and horizontally scalable NoSQL database system. It guarantees high availability, high throughput, low latency, and tunable consistency.

- OntotextGraphDB[2]: a highly efficient and robust graph database with RDF and SPARQL support, also available as a high-availability cluster.

2.4 Graph Query Languages

For many years a lot of effort has been put into enhancing graph query languages [57]. In this section, we present a list of the most used ones. For the sake of the thesis, we recommend focusing on the parts explaining Cypher in Section 2.4.1, which lays the foundations for the problem setting that will be exposed later in Section 3

2.4.1 Cypher

Cypher [24] is a declarative query language for querying graph data expressively and efficiently. It was initially designed and implemented as part of the Neo4j graph database, with the aim of defining a query language that is simple to learn, understand, and use for everyone, having the power and functionality of other standard data access languages. The language was developed taking inspiration from the strength of SQL, but with the ability to meet the requirements of a database based on graph theory principles. Currently, it used by several commercial database products and researchers, and several evolutions and extensions of versions have been proposed by the openCypher project (see Section 2.4.1.1).

During the years property graphs have been shown to be suited for shaping data in many research areas and industries, such as recommendation engines, fraud detection, IT operations and network management, social networks, software system analysis, and many more. Cypher is based on the Labeled Property Graph Mode, which is, previously described in Section 2.2.2, composed of:

- *Nodes*, representing entities (e.g. people, books, animals);
- *Relationships*, representing the directed, named connections between two nodes. A relationship always has a direction, a start and an end node, and exactly one relationship type;

[1]https://azure.microsoft.com/en-us/services/cosmos-db/#features
[2]https://www.ontotext.com/products/graphdb

- *Properties*, possible attributes in the form of key-value pairs, associated with nodes and relationships;
- *Labels*: nodes can be tagged with zero or more labels, representing their different roles in a domain.

Figure 2.3 shows an example of a Neo4j graph created using Cypher.

However, nodes and relationships are the simple components that build the most valuable and powerful piece of the property graph model - the *pattern*. Patterns consist of node and relationship elements and can express simple or complex traversals and paths. Cypher's syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. It is a declarative, SQL-inspired language for describing visual patterns in graphs using *ASCII-Art* syntax, a text-based visual art for computers. It allows us to state what we want to select, insert, update, or delete from our graph data without a description of exactly how to do it. This makes the language very visual and easy to read because it both visually and structurally represents the data specified in the query. For instance, nodes are represented with parentheses around the attributes and information regarding the entity. Relationships are depicted with an arrow (either directed or undirected) with the relationship type in brackets.

Figure 2.4 and Figure 2.5 show respectively an example of Cypher query (*"Find the friends of someone who works for Neo4j"*) defined over the graph in Figure 2.3, and its result. Similar to other query languages, Cypher contains a variety of keywords for specifying patterns, filtering patterns, and returning results. Among those [1], most common are:

- **MATCH**: used to describe the search pattern for finding nodes, relationships, or combinations of nodes and relationships together;
- **WHERE**: used to add additional constraints to patterns and filter out any unwanted patterns;
- **RETURN**: formats and organizes how the results should be outputted. It is possible to return the results with specific properties, lists, ordering, and more.

Through Cypher, users can construct expressive and efficient queries to handle needed *CRUD* (Create, Read, Update, and Delete) operations. Cypher's keywords that allows us to specify clauses for writing, updating, and deleting data are:

[1]https://neo4j.com/docs/cypher-manual/current/

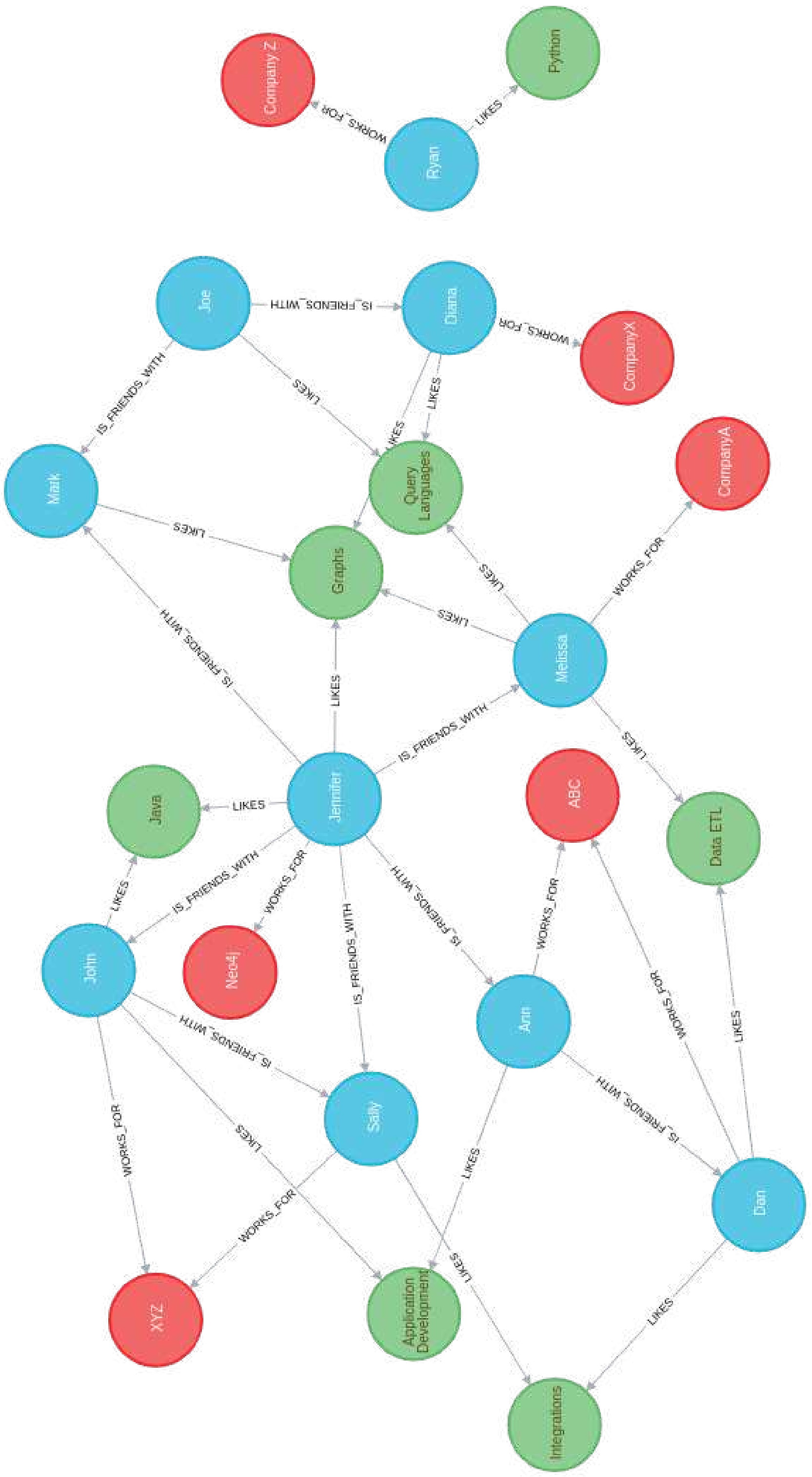
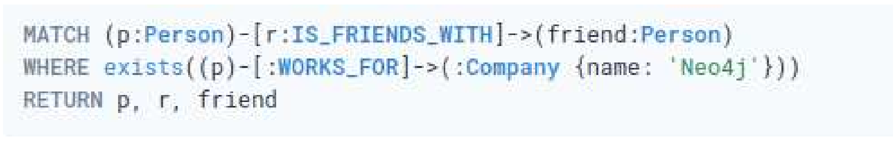


Figure 2.3: Example of a Neo4j graph created using Cypher



```
MATCH (p:Person)-[r:IS_FRIENDS_WITH]->(friend:Person)
WHERE exists((p)-[:WORKS_FOR]->(:Company {name: 'Neo4j'}))
RETURN p, r, friend
```

Figure 2.4: Example of a Cypher query[1]

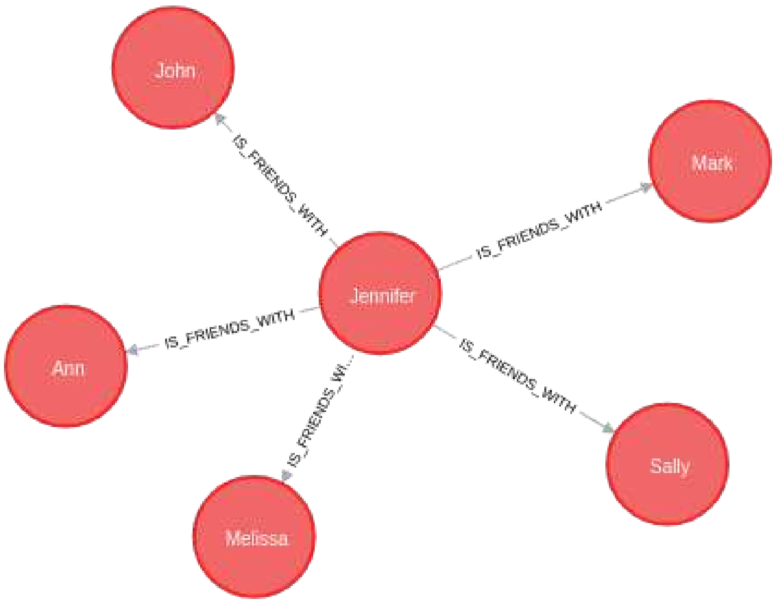


Figure 2.5: Result of the Cypher query in Figure 2.4[1]

- *CREATE* to create nodes and relationships, or *MERGE* if we want to create nodes uniquely without duplicates;
- *DELETE* to create and delete nodes and relationships. Nodes can only be deleted when they have no other relationships still existing;
- *SET* and *REMOVE* for updating purposes. They are used to set values to properties and to add/remove labels on nodes.

2.4.1.1 Cypher-based Query Languages

As described in previous sections, in recent years, the interest around property graph query languages massively increases [56], and several vendors start providing new languages or improving already existing ones [53, 57, 25, 24, 42, 2, 19].

[1]source: https://neo4j.com/developer/cypher/filtering-query-results/

Among these, Cypher [24] has grown to be the most popular and widely used query language for property graphs. Initially developed as a Neo4j product, nowadays Cypher has been implemented commercially in several open-source projects such as SAP HANA [37], Redis Graph [11], AgensGraph[1] (over PostgreSQL) and Memgraph[2] via the openCypher[3] project [26].

The openCypher project offers an open language definition, a technical compatibility kit, and a reference implementation of the Cypher parser, planner, and runtime. It is supported by several database vendors and allows database implementors and customers to freely gain from, use, and contribute to the development of the openCypher language. Below, a more detailed overview of the most popular open-source projects and research projects, defined by the openCypher community.

- **Morpheus**
  Morpheus [20] is a Cypher extension for the Apache Spark[4] system. It allows for the integration of many data sources and supports multiple graph querying. Analytical graph queries can be processed over a Spark cluster. The result of a query can also have a graph format, allowing the creation of complex processing pipelines orchestrated by a powerful and expressive high-level language. Moreover, it is frequently used by data scientists that exploit its tools for the integration of disparate data sources into a single graph. From this graph, queries can extract subgraphs of interest into new result graphs, which can be conveniently exported for further processing. It builds on the Spark SQL DataFrame API, offering integration with standard Spark SQL processing and also allows integration with GraphX[5]. It is built on top of the Spark DataFrame API and uses features such as the Catalyst optimizer. The Spark representations are accessible and can be converted to representations that integrate with other Spark libraries. Morpheus supports only a subset of Cypher and is the first implementation of multiple graphs and graph query compositionality, which is one of the main features that Cypher 10 (next Cypher release) will bring in as a core component. An integrated data source API allows developers to plug in custom data importers for external graphs. Morpheus currently supports importing graphs from Hive[6], Neo4j[7], relational database systems via JDBC and from files stored either locally, in HDFS or S3.

[1]https://bitnine.net/agensgraph/

[2]https://memgraph.com/

[3]https://www.opencypher.org

[4]https://spark.apache.org/

[5]https://spark.apache.org/docs/latest/graphx-programming-guide.html

[6]https://github.com/hivedb/hive

[7]https://neo4j.com/

- **Cypher in Gradoop**
  Junghanns et al. in [30, 31] recognized that query languages are currently only supported by graph databases but not by distributed graph processing systems. Starting from that, they start a research project aiming to extend distributed graph processing systems by query capabilities that show the same expressiveness as those of graph database systems. The authors describe a large scale distributed graph query processing system based on Flink operations and built on top of Gradoop, a distributed open-source framework for graph analytics and processing, based on the dataflow framework Apache Flink [13]. Gradoop framework already provides operators for working with graphs such as subgraph extraction, graph transformation, graph grouping, as well as property-based aggregation and selection. So, the main idea of their work was to add the pattern matching core of Cypher to Gradoop, which scales out computation across multiple machines. Since the first experimental results, good scalability emerged for increasing computing resources and near-perfect scalability for increasing data set sizes. However, query performance depends heavily on data and query graph characteristics as well as the query execution strategy.

- **Cytosm**
  Steer et al. in [44] propose Cytosm (stands for Cypher to sql mapping)[1], a middleware application which enables the execution of property graph queries, on non-graph databases, without data migration. They start from the consideration that, in recent years, property graph models are becoming widely used, but despite this, a lot of companies store data in non-graph-specific databases. Data must be loaded into a specialized graph database in order to take advantage of the high expressiveness associated with declarative graph query languages. Additionally, property graphs are often schema-free, complicating efficient query execution. The goal of Cytosm is to efficiently execute OpenCypher queries on non-graph databases. To do this, Cytosm relies on gTop, a schema containing an abstract property graph topology, and its mapping to specific database backends. Their experiments show that openCypher queries translated via Cytosm have a similar execution time to manually tailored SQL queries, and also times comparable to the same queries executing on leading dedicated graph databases.

A final import remark is about *GQL* project proposal. The GQL project [38], led by Stefan Plantikow (the first lead engineer of Neo4j's Cypher for Apache Spark project) and Stephen Cannan (Technical Corrigenda editor of SQL), aims

[1]https://github.com/cytosm/cytosm

to complement the work of creating a standardized query language for property graphs. The GQL project proposal states:

> " There are two graph models in current use: the Resource Description Framework (RDF) model and the Property Graph model. The RDF model has been standardized by W3C in a number of specifications. The Property Graph model, on the other hand, has a multitude of implementations in graph databases, graph algorithms, and graph processing facilities. However, a common, standardized query language for property graphs (like SQL for relational database systems) is missing. GQL is proposed to fill this void."

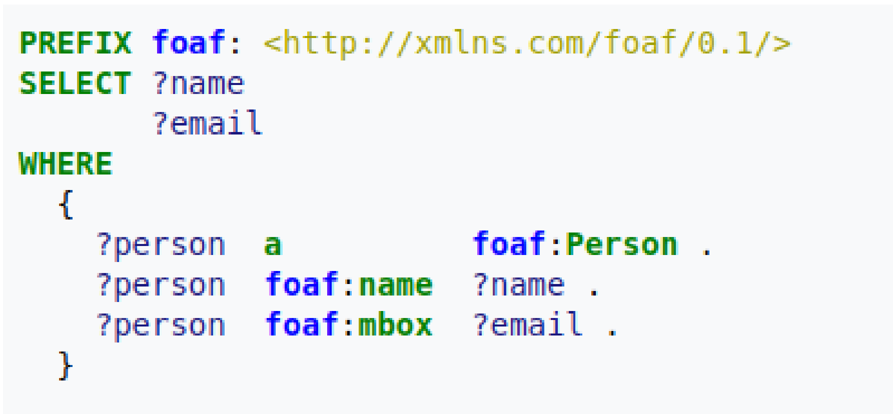
2.4.2 SPARQL

Since the introduction of RDF as a World Wide Web Consortium (W3C) Recommendation in 1998, various designs and implementations of RDF query languages have been suggested [27].

The RDF Data Access Working Group, which is part of the W3C Semantic Web Activity, published the first public working draft of SPARQL (recursive acronym for SPARQL Protocol and RDF Query Language), a query language for RDF, in 2004. Since then, SPARQL has been rapidly adopted as the standard for querying semantic Web data, and on 15 January 2008, W3C recognized SPARQL 1.0 as an official recommendation [29], followed by SPARQL 1.1 in March 2013 [28]. Nowadays, several SPARQL implementations for multiple programming languages exist[1]. The definition of a formal semantics for SPARQL has played a key role in the standardization process of this query language. SPARQL can be used to define queries across several data sources, regardless of whether the data is stored natively as RDF or seen as RDF via middleware. Figure 2.6 shows an example of a SPARQL query, which returns names and emails of every person in the dataset. A query defined in SPARQL is syntactically represented by three part:

- **Query form:** a block defined either by the keyword *SELECT*, *CONSTRUCT*, *ASK* or *DESCRIBE*;
- Zero or more **dataset clauses** defined through the keyword *FROM* or *FROM NAMED*;
- **WHERE** clause: it provides a graph pattern to match against the RDF dataset constructed from the dataset clauses;
- Possibly **solution modifiers**, e.g. *DISTINCT.*

[1]https://www.w3.org/wiki/SparqlImplementations



```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?name
       ?email
WHERE
  {
    ?person  a          foaf:Person .
    ?person  foaf:name  ?name .
    ?person  foaf:mbox  ?email .
  }
```

Figure 2.6: Example of a SPARQL query

2.4.3 Other Graph Query Languages

The following is a list of notable graph query languages, with a brief description of their uses and their features.

2.4.3.1 Gremlin

Gremlin[1] is the graph traversal language of Apache TinkerPop. It is is a functional, data-flow language that allows users to describe, in a concise manner, complex traversals or complex queries over a property graph. Every Gremlin traversal is made up of a series of (possibly nested) steps. A *step* is an atomic operation, i.e. a single unit of work on the data stream. Every step can be either:

- *Map-step:* it transforms all the items in the stream; or
- *Filter-step:* it removes objects from the stream; or
- *Side-Effect-step:* used for computing statistics about the stream.

Gremlin was designed according to the philosophy *"Write Once, Run Anywhere"*, i.e. every Gremlin traversal can be evaluated as either a real-time query (OLTP execution) or as a batch analytics query (OLAP execution). This universality is made possible by Gremlin traversal machine, a distributed, graph-based virtual machine which understands how to coordinate the execution of a multi-machine graph traversal. Gremlin naturally supports both imperative and declarative querying. The former tells the traversers how to proceed at each step in

[1]https://tinkerpop.apache.org/gremlin.html

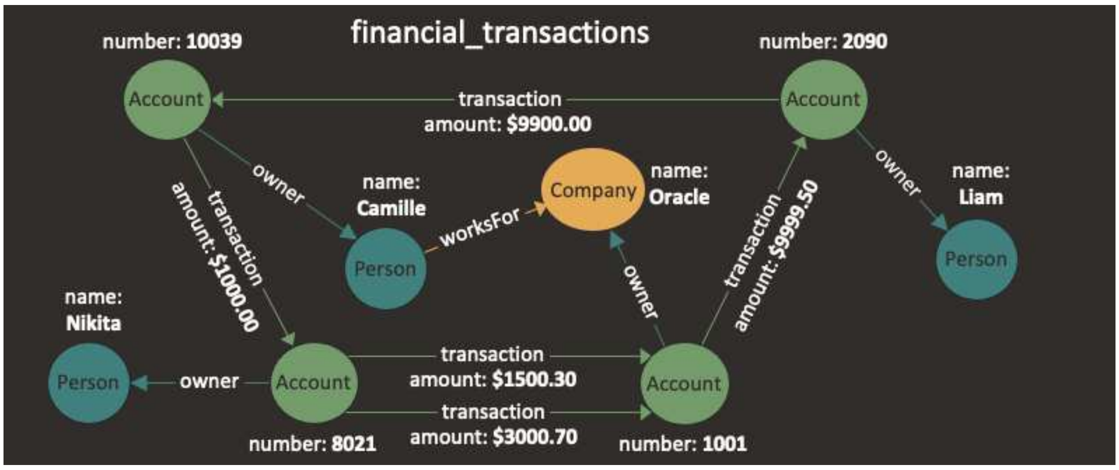


Figure 2.7: Simple Property Graph created with PGQL[1]

the traversal, the latter does not tell the traversers the order in which to execute their walk, but instead, allows each traverser to select a pattern to execute from a collection of (potentially nested) patterns.

2.4.3.2 PGQL - Property Graph Query Language

PGQL [53] is a SQL-like query language for property graph data structures designed and implemented by Oracle Inc., but made available as an open source specification. The language combines familiar SQL-expressions with a graph pattern matching language. Figure 2.8 shows a set of statements that allows creating the graph in Figure 2.7. In addition, Figure 2.9 shows an example of the following SELECT query over the defined graph: *"Produce an overview of account holders that have transacted with a person named Nikita"*.

2.4.3.3 GraphQL

GraphQL[2] is an open-source data query and manipulation language for APIs, and a runtime for fulfilling queries with existing data. It was developed internally by Facebook in 2012 before being publicly released in 2015. Despite its name, it does not provide the rich variety of graph operations present in a typical property graph database (like Neo4j). After a GraphQL service is running (typically at a URL on a web service), it can receive GraphQL queries to validate and execute. The service first checks a query to ensure it only refers to the types and fields defined, and then

[1]source: https://pgql-lang.org/
[2]https://graphql.org/

```
CREATE PROPERTY GRAPH financial_transactions
VERTEX TABLES (
  Persons LABEL Person PROPERTIES (name),
  Companies LABEL Companie PROPERTIES (name),
  Accounts LABEL Account PROPERTIES (number)
)

EDGE TABLES (
  Transactions
    SOURCE KEY (from_account) REFERENCES Accounts
    DESTINATION KEY (to_account) REFERENCES Accounts
    LABEL transaction PROPERTIES (amount),
  Accounts AS PersonOwner
    SOURCE KEY (id) REFERENCES Accounts
    DESTINATION Persons
    LABEL owner NO PROPERTIES,
  Accounts AS CompanyOwner
    SOURCE KEY (id) REFERENCES Accounts
    DESTINATION Companies
    LABEL owner NO PROPERTIES,
  Peson AS worskFor
    SOURCE KEY (id) REFERENCES Persons
    DESTINATION Companies
    NO PROPERTIES
)
```

Figure 2.8: PGQL statements for the creation of the graph in Figure 2.7

runs the provided functions to produce a result. GraphQL servers are available for multiple languages, including Java, JavaScript, Python, C++, Haskell, Perl, Ruby, Scala, PHP, and many more.

2.4.3.4 G-CORE

G-CORE [2] is a research graph query language for property graph databases. It was designed by the LDBC Graph Query Language Task Force, consisting of members from industry and academia. It is path-oriented meaning that paths are first-class citizens, i.e. core operations and queries are defined in terms of paths and nodes and edges are retrieved as being pieces of a specific path. In G-CORE, we can perform all the main core operations like graph patterns, path patterns, aggregation, subqueries, graph and path construction. An important feature is the composability: the input and the output of all the operations are graphs, i.e. graph inputs are processed to create a graph output, using graph projections and graph

```
SELECT owner.name AS account_holder, SUM(t.amount) AS tot_transact_Nikita
FROM MATCH (p:Person)<-[:owner]-(account1:Account),
     MATCH (account1))-[t:transaction]-(account2)
     MATCH (account2:Account)-[:owner]-(o:Person|Company)
WHERE p.name = 'Nikita'
GROUP BY o
```

Figure 2.9: PGQL SELECT query over the graph in Figure 2.7

set operations to construct the new graph. G-CORE queries are pure functions over graphs, having no side effects, which mean that the language does not define operations which mutate (update or delete) stored data.

2.5 CQL - Continous Query Languages

Traditional DBMSs are best suited for running one-time queries over finite stored data sets. However, nowadays, a lot of applications, such as network monitoring and sensor networks, require continuous queries over unbounded streams of data.

The STREAM [5] project at Stanford proposes a general-purpose prototype Data Stream Management System (DSMS), also called STREAM, that supports a large class of declarative continuous queries over continuous streams and traditional stored data sets. In [6] Arasu et al. presented, as part of the STREAM project, CQL (*Continuous Query Language*), an expressive SQL-based declarative language for registering general-purpose continuous queries against streams and updatable relations. In this paper, the authors define a precise abstract semantics for continuous queries, based on two data types, *streams* and *relations*, which are defined using a discrete, ordered *time domain* Γ:

> ***Definition (Stream).*** A *stream* S is a (possibly infinite) bag (multiset) of *elements* $\langle s, \tau \rangle$ where s is a tuple belonging to the schema of S and $\tau \in \Gamma$ is the *timestamp* of the element.

> ***Definition (Relation).*** A *relation* R is a mapping from Γ to a finite but unbounded bag of tuples belonging to the schema of R.

The abstract semantics uses three classes of operators over streams and relations:

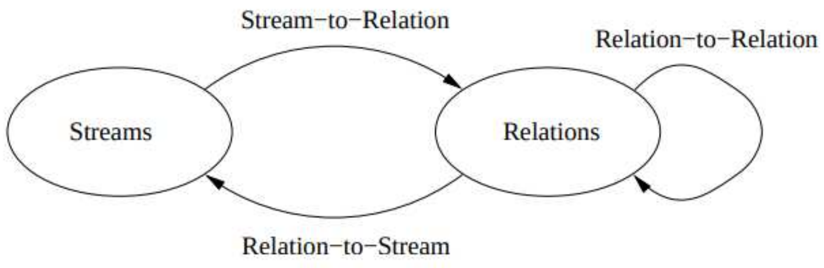


Figure 2.10: Operator classes and mappings used in abstract semantics [6]

- *stream-to-relation* operator takes a stream as input and produces a relation as output. The *S2R* operators in CQL are based on the concept of a *sliding window* [1] over a stream;
- *relation-to-relation* operator takes one or more relations as input and produces a relation as output. CQL uses SQL constructs to express its *R2R* operators, and much of the data manipulation in a typical CQL query is performed using these constructs, exploiting the rich expressive power of SQL; and
- *relation-to-stream* operator takes a relation as input and produces a stream as output. CQL has three *R2S* operators:
  - *Istream* streaming out all new entries of an instantaneous relation w.r.t. the previous one
  - *Dstream* streaming out all deleted entries of an instantaneous relation w.r.t. the previous one
  - *Rstream* streaming out all entries of an instantaneous relation at a certain instant

  where an instantaneous relation is a relation (bag of tuples) at a given instant.

2.5.1 RDF and CQL

In the early 2000s', the scientific community proposed continuous query languages to support continuous query evaluation on stream of linked data.

Stream Reasoning (SR) [18] is the research area that combines Stream Processing and Semantic Web technologies to make sense, in real-time, of vast, heterogeneous and noisy data streams. The SR community's contributions include data models, query languages, and algorithms, and benchmarks for *RDF Stream Processing (RSP)*. In the last decade, the RSP community has proposed several models and languages for continuously querying and reasoning over *RDF streams*[1]:

- Dell'Aglio et al. in [17] proposed the RSP-QL model, a unifying semantic, similar to the one used in DSMSs, for RSP engines and continuous SPARQL extensions.
- Barbieri et al. in [8] presented C-SPARQL (*Continuous SPARQL*), a continuous extension of SPARQL allowing to register continuous queries over RDF Streams.
- Le-Phuoc et al. [34] propose CQELS (*Continuous Query Evaluation over Linked Streams*), a native and adaptive query processor for unified query processing over Linked Stream Data and Linked Data.
- *Sparkwave* is a solution for continuous schema-enhanced pattern matching over RDF data streams, presented by Komazec et al. in [32]. The aim of Sparkwave is to achieve and retain high-throughput RDF graph pattern matching while providing a number of stream reasoning features such as support for fairly expressive pattern definitions, time-based sliding windows and schema-entailed knowledge.
- Anicic et al. in [3] propose *Event Processing SPARQL* (EP-SPARQL), a language extending SPARQL with its event processing [39] and stream reasoning capabilities.
- Calbimonte et al. in [12] presented $SPARQL_{stream}$, a SPARQL streaming extension to query virtual RDF streams composed of timestamped RDF triples.
- In [50] Tommasini et al. present RSP4J, a flexible API for the development of RSP engines and applications under RSPQL semantics.

2.5.2 Property Graph and CQL

Although in recent years the interest in Property Graphs has exponentially grown, at the moment only a few PoC implementations extending CQL to property graphs have been proposed, highlighting poor work from the scientific community in this field.

Marton et al. in [35] propose inGraph[2] a PoC system to perform live queries on graph data in scenarios in which real-time results play a key role (e.g. sensor

[1]https://www.w3.org/community/rsp/wiki/RDF_Stream_Models
[2]https://github.com/FTSRG/ingraph

systems for failure or danger detection, systems for critical monitoring tasks like railway monitoring). Traditional systems typically work with batch queries, i.e. they cannot obtain query results on demand, but they have to wait for databases response before the computations. On the contrary live queries are computed continuously. Initially, a client registers the queries. Then, every time the graph data changes, the results are maintained, recalculating the results of the queries. The propsed inGraph implementation follows all the standard steps for building a query engine:

1. Parse Cypher queries, producing syntax trees;
2. build a relational graph algebra expressions using a compiler;
3. use a transformer and optimizer (VIATRA[1]) to properly transform the graph into an incrementally maintainable graph that embeds relational algebra operation.

InGraph can therefore be considered an example of a distributed continuous graph query engine based on live queries and incremental query evaluation. Szárnyas et al. in [48] shows that incremental evaluation model have already good scaling capabilities both increasing computing resources and increasing data set sizes. Despite inGraph is an important step for the extension of CQL to property graphs, the engine still lacks many important features of traditional DSMSs, such as windowing and integration with stream processing systems.

Starting from the consideration that no notable mature work has been done by the scientifc community to extend CQL to property graphs, Falzone et al. propose Seraph [22], an extension of the syntax and semantics of Cypher able to cope with streaming (graph) data and continuous query evaluation. Seraph introduces streaming features in the context of property graph query languages, with *windows* (both *time-based* and *event-based*), *streaming operators* ispired by CQL model (*Dstream*, *RStream*, and *IStream* operators), as well as a new data model that extends the property Graph data model adding time dimension to it. In Chapter 3, we briefly introduce the Seraph language and its syntax and semantics.

2.6 Design Backgrounds

For the sake of completeness, this section provides some background about powerful, state-of-the-art components used during the design of the Dozer architecture, which will be better described in the next chapter.

[1]https://www.eclipse.org/viatra/

2.6.1 Apache Kafka

Apache Kafka[1] is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications. Kafka combines three key capabilities to implement event streaming solutions:

- To **publish** (write) and **subscribe to** (read) streams of events, including continuous import/export of your data from other systems.
- To **store** streams of events durably and reliably.
- To **process** streams of events as they occur or retrospectively.

Kafka is a distributed system consisting of servers and clients that communicate via a high-performance TCP network protocol. It is run as a highly scalable and fault-tolerant cluster of one or more servers that can span multiple datacenters or cloud regions. Some servers, called **brokers**, are used to store streams of records. Other servers run *Kafka Connect* (see 2.6.2.4) to continuously import and export data as event streams. It is a *publish-subscribe* based messaging system:

- **Producers**: client applications that publish (write) events to Kafka.
- **Consumers**: client applications that subscribe to (read and process) these events.

Messages are read from and published to Kafka in the form of *events*. A Kafka event has a key, value, timestamp, and optional metadata headers and the fact that "something happened" in the world. Events are organized and durably stored in *topics*.

Kafka topics are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Topics are partitioned over different Kafka brokers, allowing client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is appended to one of the topic's partitions. Events with the same event key are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition's events in exactly the same order as they were written. Every topic can be replicated. The replication factor defines the number of copies of a topic in a Kafka cluster and allows to customize the level of fault-tolerance and availability.

[1]https://kafka.apache.org/

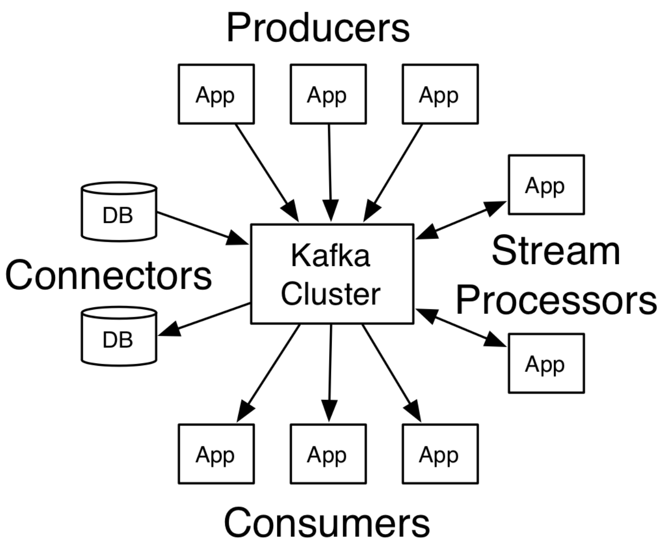


Figure 2.11: Client-Server communication via Kafka APIs[1]

2.6.2 Kafka APIs

Apache Kafka provides five core APIs:

- The **Producer API** allows applications to publish a stream of records to one or more Kafka topics in the Kafka cluster;
- The **Consumer API** allows applications to subscribe to one or more topics in the Kafka cluster;
- the **Streams API** allows application to transform streams of data from input topics to output topics. It provides higher-level functions to implement stream processing applications and microservices, such as transformations, aggregations and joins, windowing, and more;
- the **Connect API** allows to build and run reusable data source/sink connectors that continually consume or produce streams of events from and to external systems and applications;
- the **Admin API** allows managing and inspecting topics, brokers, and other Kafka objects.

[1]source: https://kafka.apache.org/20/documentation.html

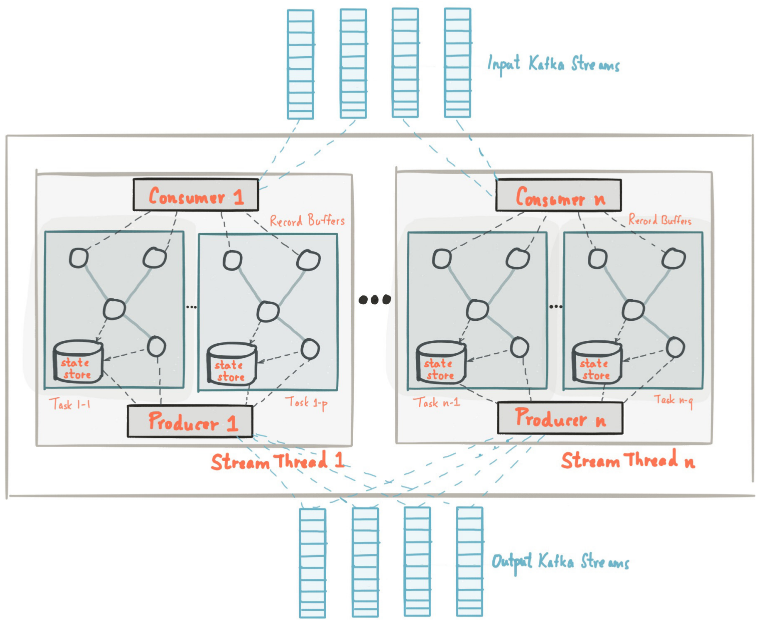


Figure 2.12: Logical view of a Kafka Streams application[1]

2.6.2.1 Kafka Streams

Kafka Streams is a client library for developing highly scalable, elastic, fault-tolerant distributed applications and microservices that store input and output data in an Apache Kafka cluster. It combines the benefits of Kafka's server-side cluster technology with the ease of developing and deploying ordinary Java and Scala apps on the client-side.

A stream processing application is a program that uses the Kafka Streams library. Kafka Streams is built on the Apache Kafka producer and consumer APIs (see 2.6.2.3) and it exploits the native features of Kafka to offer data parallelism, distributed coordination, fault tolerance, and operational simplicity. Figure 2.12 depicts a logical perspective of a Kafka Streams application with several stream threads, each of which contains multiple stream tasks.

When implementing stream processing applications, you typically need both databases and streams, to store respectively historical and live streaming data. Kafka's Streams API provides first-class support for streams and tables. The most significant abstraction offered by Kafka Streams is the *stream*, which represents an unbounded, constantly updating data collection and consists of one or more stream partitions.

[1]source: https://docs.confluent.io/platform/current/streams/architecture.html

A stream partition is an, ordered, replayable, and fault-tolerant sequence of immutable data records, where a data record is defined as a key-value pair. Partitioning enables data locality, elasticity, scalability, high performance, and fault tolerance. Kafka Streams uses the concepts of stream partitions and stream tasks as logical units of its *parallelism model*:

- Each stream partition is a totally ordered sequence of data records and maps to a Kafka topic partition.
- A data record in the stream maps to a Kafka message from that topic.
- The keys of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

In kafka, a stream can be viewed as a table, and a table can be viewed as a stream, where a table is a collection of key-value pairs. This relationship between streams and tables is called *stream-table duality*:

- *Stream as Table*: a stream can be considered a changelog of a table, where each data record in the stream captures a state change of the table
- *Table as Stream*: a table can be considered a snapshot, at a point in time, of the latest value for each key in a stream.

Kafka Streams models the duality explicitly via the *KStream* and *KTable* abstractions:

- *KStream*: an abstraction of a record stream, where each data record represents a self-contained datum in the unbounded data set;
- *KTable*: an abstraction of a changelog stream, where each data record represents an update.

Other key features that Kafka Streams includes are:

- It handles *out-of-order data*, with low latency and high throughput *record-at-a-time processing* (no micro-batching).
- Every stream task in a Kafka Streams application may embed one or more fault-tolerant local state stores, which enables very fast and efficient stateful operations, like windowed joins and aggregations. Local state stores can be accessed via APIs to store and query data required for processing. State stores are robust to failures. For each of them, Kafka Streams maintains a replicated changelog Kafka topic in which it tracks any state updates.

- According to the *exactly-once* processing semantics, each record is processed once and only once, even if there is a failure on either clients or Kafka brokers.

A stream processing application may define one or more *processor topologies*, which represent a logical abstraction of the stream processing code.

2.6.2.2 Processor Topology

High-level Kafka Stream client applications are based on the idea of building a *processor topology* (or simply *topology*). A topology consists of a graph of *stream processors* (nodes) that are connected by *streams* (edges) or *shared state stores*. It defines the stream processing computational logic of a streaming application, i.e. how input-data is transformed into output-data.

A *stream processor* is a node in the processor topology, as shown in Figure 2.13; it denotes a processing step that transforms data in streams by receiving one input record at a time from its upstream processors, applying its operation to it (such as map or filter, joins, and aggregations), and then producing one or more output records to its downstream processors. In every topology we have two special processors:

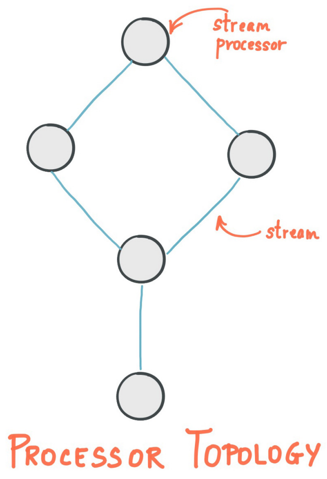


Figure 2.13: Processor topology[1]

- **Source Processor:** a special type of stream processor that does not have any upstream processors. It consumes records from one or multiple Kafka topics, which represent the input stream to defined topology, and forward such records to its down-stream processors.

- **Sink Processor:** a special type of stream processor that does not have down-stream processors. It sends any received records from its up-stream processors to a specified Kafka topic.

Kafka Streams offers two ways to define the stream processing topology:

- The imperative, lower-level *Processor API*[1] that provides allows developers to define and connect custom processors as well as to interact with state stores.
- The declarative, functional *Kafka Streams DSL*[2] that provides the most common built-in data-transformation operations such as map, filter, join and aggregations.

The low-level Processor API are more flexible than the Streams DSL, which builds on top of the former, but it require more manual coding work.

2.6.2.3 Consumer & Producer API

Kafka Producer and *Kafka Consumer* APIs allow client applications respectively to send and to read streams of data to/from the Kafka cluster.

- **Kafka Consumer**

  A *Kafka Consumer* is a client that consumes records from a Kafka cluster. It interacts with the broker to allow groups of consumers to load balance consumption using consumer groups. A *consumer group* is a collection of consumers who work together to consume data from a specific topic. The topics' partitions are distributed among the consumers in the same group. Each consumer in a group can dynamically set the list of topics it wants to subscribe to through one of the subscribe APIs. Kafka will deliver each message in the subscribed topics to one process in each consumer group. This is achieved by balancing the partitions between all members in the consumer group so that each partition is assigned to exactly one consumer in the group. Moreover, every time new members join the group and old members leave, the group is *re-balanced*. Rebalancing the group means that the partitions are re-assigned so that each member receives a proportional share of the partitions. As a consumer in the group reads messages from the partitions assigned by the *coordinator*, it must commit the offsets corresponding to the messages it has read. A group's coordinator is responsible for managing the

[1]https://kafka.apache.org/10/documentation/streams/developer-guide/processor-api.html
[2]https://kafka.apache.org/20/documentation/streams/developer-guide/dsl-api.html

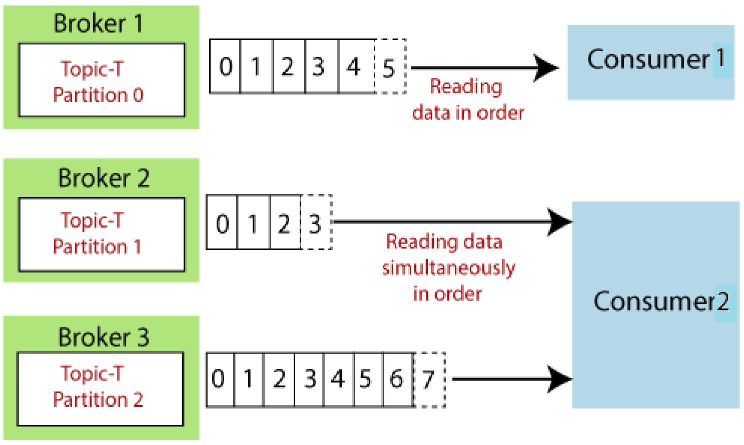


Figure 2.14: Kafka Consumer[1]

members of the group as well as their partition assignments. Kafka maintains a numerical *offset* for each record in a partition. This offset acts as a unique identifier of a record within that partition, and also denotes the position of the consumer in the partition. There are actually two notions of position relevant to the user of the consumer:

- The *position* of the consumer gives the offset of the next record that will be given out. It will be one larger than the highest offset the consumer has seen in that partition. It automatically advances every time the consumer receives messages in "*poll*" call.
- The *committed position* is the last offset that has been stored securely. Should the process fail and restart, this is the offset that the consumer will recover to.

By default, the consumer is configured to *auto-commit* offsets. Using auto-commit gives you *"at least once"* delivery, i.e. Kafka guarantees that no messages will be missed, but duplicates are possible. The consumer also supports a *commit API* which gives developers full control over offsets.Most of the time, the consumer just consumes records from beginning to end, periodically committing its position (either automatically or manually). Kafka, on the other hand, lets the consumer manage its position manually, moving forward or backward in a partition at will.

[1]source: https://www.javatpoint.com/apache-kafka-consumer-and-consumer-groups

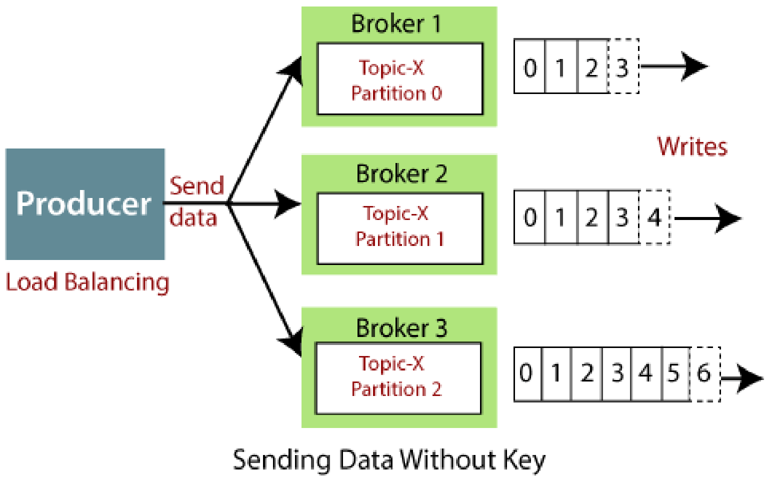


Figure 2.15: Kafka Producer[1]

- **Kafka Producer**

  A *Kafka Producer* is a client that publishes records to the Kafka cluster. It has no need for group coordination, and so it is conceptually much simpler than the consumer.

  A *producer partitioner* maps each message to a topic partition, and the producer sends a produce request to the leader of that partition. Partitioners guarantee that all messages with the same non-empty key are sent to the same partition. Each partition in the Kafka cluster has a leader and a set of replicas among the brokers. All writes to the partition must go through the partition leader. The replicas are kept in sync by fetching from the leader. Upon leader failure, a new leader is chosen from among the in-sync replicas. Producers can customize between three acks-level, to control message durability at some cost to overall throughput:

  – Acks=1, requires an explicit acknowledgement from the partition leader that the write succeeded;

  – Acks=all, the strongest guarantee that Kafka provides. The write is replicated to all of the in-sync replicas;

  – Acks=0, maximize the throughput at cost of no guarantee that the message was successfully written to the broker's log since the broker does not even send a response.

[1]source: https://www.javatpoint.com/apache-kafka-producer

Regardless of the producer's acknowledgement settings, messages written to the partition leader are not immediately readable by consumers. When all in-sync replicas have acknowledged the write, then the message is considered committed, which makes it available for reading. This ensures that messages cannot be lost by a broker failure after they have already been read.

2.6.2.4 Kafka Connect

Kafka Connect is a free, open-source component of Apache Kafka that works as a centralized data hub for simple data integration between databases, key-value stores, search indexes, and file systems. It allows to scalably and reliably streaming data between Apache Kafka and other data systems. It runs in its own process, separate from the Kafka brokers and it is distributed, scalable, and fault tolerant, just like Kafka itself. With Kafka Connect, developers can write a new connector plugin from scratch or can use built-in connectors. We distinguish two types of connectors:

- *Source connector* allows low-latency ingestion of data into Kafka topics for streaming processing;
- *Sink connector* to deliver data from Kafka topics into secondary indexes, or batch systems for offline analysis.

When combined with Kafka and a stream processing framework, Kafka Connector is an integral component of an ETL pipeline (Figure 2.16). Kafka Connect is based on some key concepts:

- **Connectors**

  A connector defines where data should be copied to and from. "Connectors" can refer to both connector instances and connector plugins. A *connector instance* is a logical job responsible for managing the copying of data between Kafka and another system. A *connector plugin* defines all the classes that implement or are used by a connector.

- **Tasks**

  Tasks play the key-role in the Kafka Connect data model. Each connector instance coordinates a set of tasks that actually copy the data. Kafka Connect provides built-in support for parallelism and scalable data copying with very little configuration. These tasks have no state stored within them. Task state is stored in Kafka in special topics and managed by the associated connector.

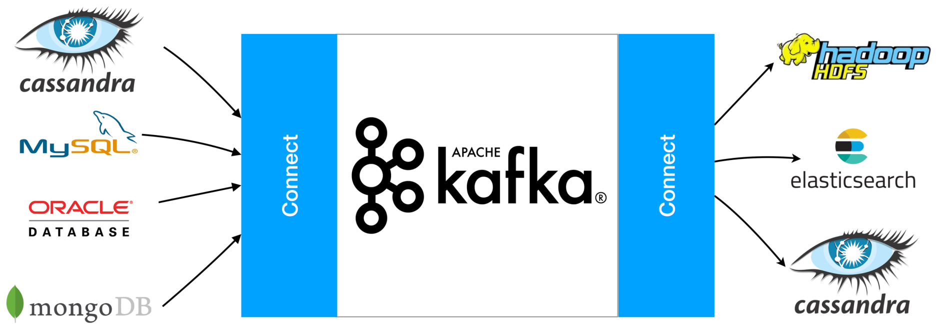


Figure 2.16: ETL pipeline with Kafka

- **Workers**

  Connectors and tasks are logical units of work and must be scheduled to execute in a *worker*. Workers call rebalancing procedure when connectors increase or decrease the number of tasks they require, or when a connector's configuration is changed, so that each worker has approximately the same amount of work. When a worker fails, tasks are rebalanced across the active workers. When a task fails, no rebalance is triggered as a task failure is considered an exceptional case. Kafka Connect has two types of workers:

  - *Standalone mode*: it requires minimal configuration and is the simplest mode, where a single process is responsible for executing all connectors and tasks;
  - *Distributed mode*: it provides scalability and automatic fault tolerance. In distributed mode, developers can start many worker processes (with the same group.id), which automatically coordinate to schedule execution of connectors and tasks across all available workers. Workers use consumer groups to coordinate and rebalance.

- **Converters**

  Converters (such as *AvroConverter*, *JsonConverter*, *StringConverter*) are necessary to have a Kafka Connect deployment support a particular data format when writing to or reading from Kafka. Converters are decoupled from connectors themselves to allow for reuse of converters between connectors naturally.

- **Transforms function**

  A simple function that take one record as an input and returns a modified record as output. Connectors can be configured with transformations to make simple and lightweight modifications to individual messages. This can be convenient for minor data adjustments and event routing.

2.6.3 Microservice Architecture

In recent years, application architectures are shifting from monolithic enterprise systems to flexible, scalable, event-driven approaches. Microservice architecture has become the de facto standard for modern Web application development. It is an architectural style that structures a monolithic application system as a collection of services that are:

- Loosely coupled with other services: each service can be developed independently without being impacted by, and affecting other services;
- independently deployable;
- highly maintainable and testable: enables rapid and frequent development and deployment;
- capable of being developed by a small team.

This approach allows each microservice to be developed, deployed and operated in parallel by different teams, favoring the rapid, frequent and reliable delivery of a large and complex application, as well as its deployment, testing, and maintenance. The problem with microservice architectures is the need for increased communication between distributed instances, and the need for microservices orchestration, new failover requirements, and resilient design patterns.

Apache Kafka plays a key role in microservices orchestration and provides important features that microservices aim to achieve, such as scalability, efficiency, and speed. It also facilitates inter-service communication while preserving ultra-low latency and fault tolerance.

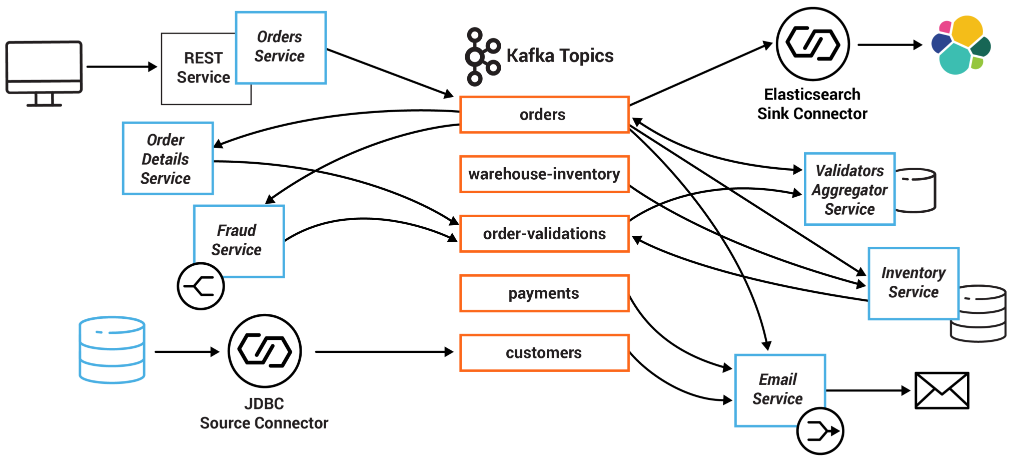


Figure 2.17: Microservice architecture using Kafka

Chapter 3

Problem Statement

In this chapter, we discuss the problem that this thesis is trying to address, detailing the characteristics the system should meet. Firstly, in Section 3.1, we provide a brief knowledge about Seraph, giving a formal specification of its semantics together with its syntax. Section 3.2 describes the problem in-depth, outlining an existing solution and its limitations. Finally, in Section 3.3, we present relevant knowledge about the crucial requirements to take into account when building an engine for continuous query processing.

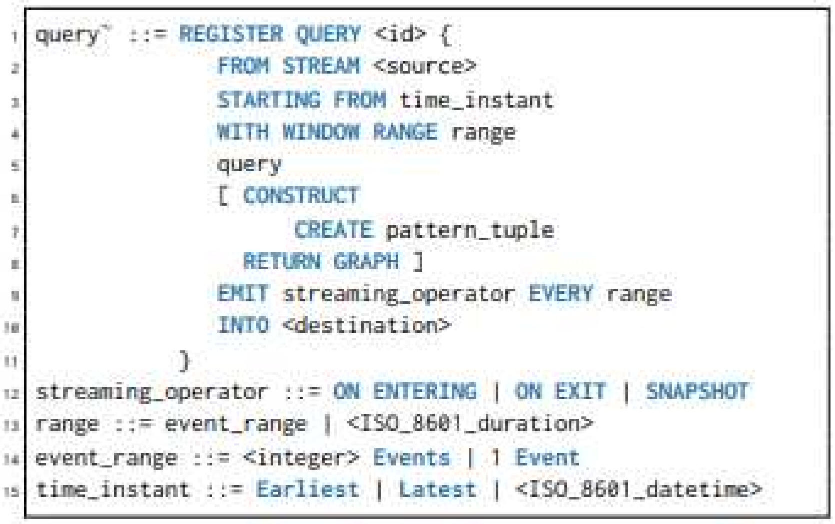
3.1 Seraph Language

Seraph [22] is a declarative language for the continuous query evaluation over streams of property graphs and arises as an extension of Cypher, intending to introduce streaming features in the context of property graph query languages. Its definition is based on the continuous-evaluation paradigm [6].

Figure 3.1 shows the syntax of a Seraph query. The **REGISTER QUERY** clause allows for registering a new query into the Seraph system. With **FROM STREAM** we specify the input stream of the Property Graph. With the **STARTING FROM** clause, we define the first evaluation time instant, expressed either as:

- An *ISO 8601 datetime*; or
- The datetime associated with the *first* (*last*) event, respectively with the keyword **Earliest** (**Latest**).

The **EVERY** clause, together with the **STARTING FROM** clause, can determine the sequence of evaluation time instance. In particular, the **EVERY**



```
query* ::= REGISTER QUERY <id> {
              FROM STREAM <source>
              STARTING FROM time_instant
              WITH WINDOW RANGE range
              query
              [ CONSTRUCT
                    CREATE pattern_tuple
                RETURN GRAPH ]
              EMIT streaming_operator EVERY range
              INTO <destination>
           }
streaming_operator ::= ON ENTERING | ON EXIT | SNAPSHOT
range ::= event_range | <ISO_8601_duration>
event_range ::= <integer> Events | 1 Event
time_instant ::= Earliest | Latest | <ISO_8601_datetime>
```

Figure 3.1: Seraph's syntax

clause defines the frequency of the evaluation, specified either with an *ISO 8601 duration* or in terms of *number of events*.

The **EVERY** clause is always preceded by the **EMIT** clause and determines which streaming operator to use. The streaming operators are time-aware, i.e. they require a time instant as input to produce their outputs.

Seraph admits three different streaming operators, relying on their first formal definition in [6]:

- **RStream operator:** it takes as input a time-varying table and annotates the instantaneous table with the evaluation time τ. This operator allows streaming out the whole answer produced at each evaluation iteration.

- **IStream operator:** it streams out only the difference between the answer of the current evaluation and the one of the previous iteration.

- **DStream operator:** it outputs only the part of the answer at the previous iteration that is not in the current one.

In Seraph, we can express to use the *RStream* with the **SNAPSHOT** clause, while with the **ON EXIT** and with the **ON ENTERING** clauses we select the *DStream* and the *IStream* respectively.

With the **WINDOW RANGE** clause, we can customize the width parameter of sliding windows. Seraph introduced both the concept of *time-based* and *event-*

based windows to create a Property Graph by extracting relevant portions of the Property Graph Stream:

- **Time-based window:** it is defined through two time instants, respectively named opening and closing time instants.
- **Event-based window:** it is defined through one time instant t, and a number N representing the number of Property Graphs to be extracted from the Property Graph Stream. An event-based window defines its output by extracting the last N events of the stream with the largest timestamps $\leq t$

Inspired by *Morpheus* (see Section 2.4.1.1), we can add an optional **CONSTRUCT CREATE** *pattern_tuple* **RETURN GRAPH** clause to create a Property Graph Stream. Finally, the **INTO** clause allows for specifying the output stream, corresponding the destination of the result stream.

Just for the sake of clarity, Figure 3.2 shows a specif example of a Seraph query in which we registered for a query that produces a stream of events whenever two or more people are in the same room with a time-based sliding window with a length of two minutes and a sliding interval of one minute.

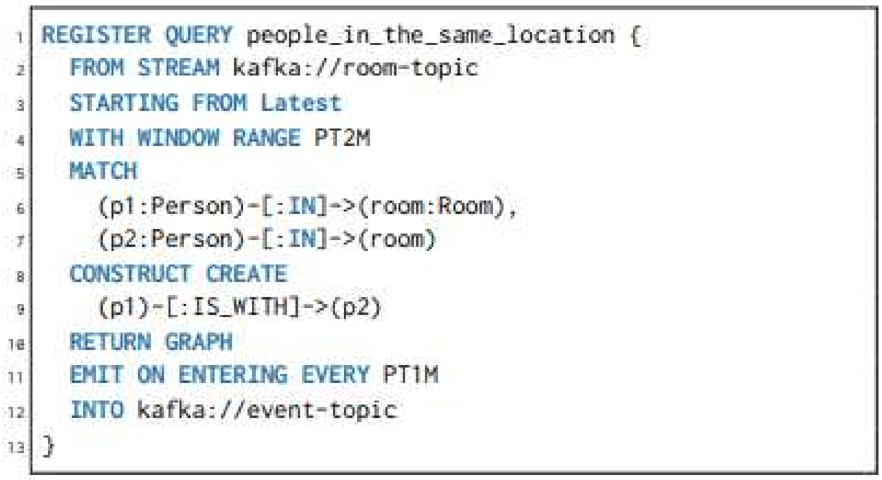
3.2 Problem Setting

Presenting Seraph (see Section 3.1), Falzone et al. assume that an underlying streaming application is required to generate and maintain time-varying graphs that capture the dynamic evolution of the data flowing inside the query engine. The only notable implemented engine about all the suggested proof-of-concept systems is GSP4J[1], an extension of RSP4J, briefly introduced in Section 2.5.1.

In [50] Tommasini et al. presented RSP4J library, a flexible API for the development of RSP engines and applications under RSPQL semantics [17]. The engine interface adopted to control the RSP4J's capabilities is based on the VoCaLS service *feature* idea [51]. The GSP4J solution adds a set of APIs to the RSP4J, for the development of an engine under the Seraph semantics. This can be considered an important step in the Property graphs area, since, for the first time, it introduces streaming features in the context of property graph query languages.

However, there are some challenges with this prototype that make it not suitable for industrial development. This system is fully in-memory, single-query, and only supports real-time analysis, ensuring high performances at the expense of lower scalability and fault tolerance.

[1]https://github.com/riccardotommasini/rsp4j/tree/gsp4j



```
REGISTER QUERY people_in_the_same_location {
  FROM STREAM kafka://room-topic
  STARTING FROM Latest
  WITH WINDOW RANGE PT2M
  MATCH
    (p1:Person)-[:IN]->(room:Room),
    (p2:Person)-[:IN]->(room)
  CONSTRUCT CREATE
    (p1)-[:IS_WITH]->(p2)
  RETURN GRAPH
  EMIT ON ENTERING EVERY PT1M
  INTO kafka://event-topic
}
```

Figure 3.2: Example of a Seraph query

Moroever, RSP4J is designed for RDF Stream Processing and does not directly support Property Graph operations. Figure 3.3 depicts the architecture presented in the article. Their solution makes use of RSP4J window operators and a reporting policy adapted to Seraph's EMIT clause, which controls how the output is emitted. They chose JSON-PG [14] as a data format in order to be graph-native at the ingestion level and to be able to represent property graph data used in current graph databases like as Neo4j, Oracle Labs PGX, and Amazon Neptune. In the paper, the authors highlight some challenges; first and foremost, RSP4J is designed for RDF Stream Processing and does not directly support Property Graph operations. Furthermore, the presented system is fully in-memory, single-query, and only supports real-time analysis. Although it guarantees high performance, it lacks some important industrial features such as scalability and fault tolerance.

With this as its starting point, we start working on a system aimed at industrial development. In Chapter 4, we present our solution aimed at overcoming such limitations.

3.3 Requirements

Dozer's objective is to provide the first implementation of the Seraph query language capable of ensuring crucial industrial features such as scalability, fault-tolerance, high throughput, and low latency. A continuous query system can easily run out of resources in case of a large amount of input stream data. Distributed continuous query processing is a scalable method to solve this problem. Distributed

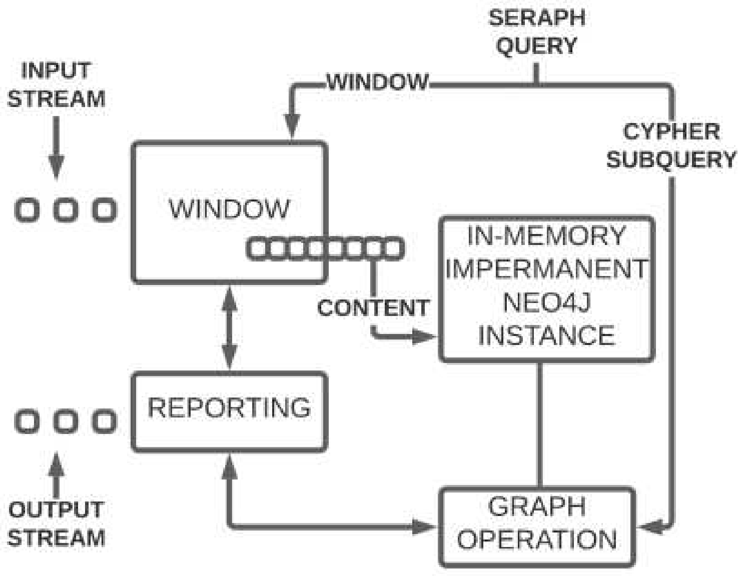


Figure 3.3: First prototype architecture presented by Falzone et al.

stream processing systems have emerged in recent years to provide high efficiency combined with high throughput at low latency. Architectural design choices play a central role when complex distributed systems must be deployed at scale.

In [46], Stonebraker et al. provide high-level guidance for developing a *real-time SPE (Stream Processing Engine)*, outlining eight requirements that the system should meet:

1. **Keep the data mooving**
   To achieve low latency, a system must be able to perform message processing without having a costly storage operation in the critical processing path. An additional latency problem exists with systems that are passive, while active systems avoid this overhead by incorporating builtin event/data-driven processing capabilities.

2. **Query using SQL on Streams (StreamSQL)**
   In streaming applications, some querying mechanism must be used to find output events of interest or compute real-time analytics. The system should support a high-level "StreamSQL" language with built-in extensible stream-oriented primitives and operators.

3. **Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)**

Real-time systems should have built-in mechanisms to deal with streams challenges, such as data that is late or delayed, missing or out-of-sequence.

4. **Generate Predictable Outcomes**
   An SPE must guarantee predictable and repeatable outcomes. This is also important from fault-tolerance and recovery perspective.

5. **Integrate Stored and Streaming Data**
   The engine should efficiently store, access, modify historical information, and combine it with live streaming data.
   For seamless integration, the system should use the same StreamSQL language when dealing with either type of data.

6. **Guarantee Data Safety and Availability**
   The applications should be always up and available, and the data integrity maintained anyway, despite failures.

7. **Partition and Scale Applications Automatically**
   The system should provide incremental scalability, by distributing the computation across multiple processors and machines. Ideally, the distribution should be automatic and transparent.

8. **Process and Respond Instantaneously**
   A stream processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

Chapter 4

Proposed Approach

This chapter presents the proposed approach to the problem detailed in Chapter 3. With the purpose of describing the path that led us to the Dozer[1] implementation, we introduce, in Section 4.1, an overview of Dozer's architecture based on some tools and concepts introduced in Section 2.6. We present a system architecture that aims at overcoming the limitations of the prototype described in Section 3.2, highlighting some architectural choices that allowed us to reach the requirements detailed in Section 3.3. Finally, Section 4.2 illustrates an overview of the internal system design laying the groundwork for the Dozer engine's implementation.

4.1 Dozer architecture

As described earlier, Falzone et al. design a system prototype based on RSP4J [50] which guarantees high performances at the expense of crucial industrial features such as scalability and fault tolerance. The first idea was to design a Stream Processing Engine (SPE) able to overcome such limitations and let the Neo4j query engine communicate with it, as shown in Figure 4.1.

SPEs can handle massive amounts of data with high throughput and low latency, addressing both data velocity and volume. We start taking into consideration several existing SPEs (e.g. Flink[2], Spark Streaming[3], Storm[4]) as the basis for our streaming application.

Among them, Kafka and Kafka Streams meet all the requirements presented in Section 3.3. In addition, Seraph directly supports consumption from Kafka, as well

[1]https://github.com/openseraph/SeraphEngine
[2]https://flink.apache.org/
[3]https://spark.apache.org/streaming/
[4]https://storm.apache.org/

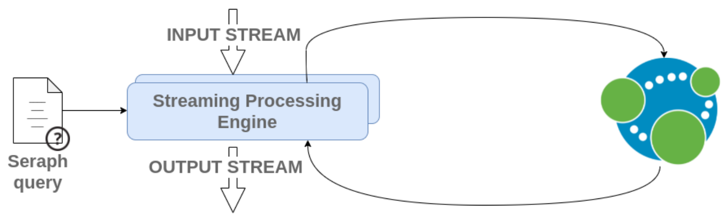


Figure 4.1: First design of our streaming processing engine

as outputs the resulting stream into Kafka topics. Moreover, the Kafka distributed and parallel processing model allows us to design a microservice architecture and to build a distributed continuous query processing that provides dynamic scalability. Last but not least, Apache Kafka provides Connect API, a free, open-source component for scalably and reliably streaming data between Apache Kafka and other data systems.

All this encouraged us to develop an underlying streaming application based on Kafka and Kafka Streams. Figure 4.2 depicts such a principle; we exploit the Kafka Connect component to ingest property graph data (possibly coming from several data sources) into Kafka topics. Then our underlying streaming application consumes event streams, representing graph data, to generate and maintain time-varying graphs that capture the dynamic evolution of the data. Finally, we use sink connector API to export the data to the Neo4j instance.

The Neo4j Labs Team has worked on *neo4j-streams*[1], a project aimed at integrating Neo4j with streaming data solutions. Neo4j Streams can be used to ingest data from external sources into the graph or to send update events to the event log for later consumption. It can run in two modes:

- as a **Neo4j plugin**
  It acts as a Neo4j Server extension and provides both sink and source functionalities. It consists of a *Neo4j Streams Source*, a transaction event handler that sends data to a Kafka topic, and a *Neo4j Streams Sink*, a Neo4j application that ingest data from Kafka topics into Neo4j;

- as a **Kafka-Connect Plugin**
  A plugin for the Confluent Platform that allows to ingest data into Neo4j, from Kafka topics (actually it offers only the Sink functionality).

[1]https://github.com/neo4j-contrib/neo4j-streams

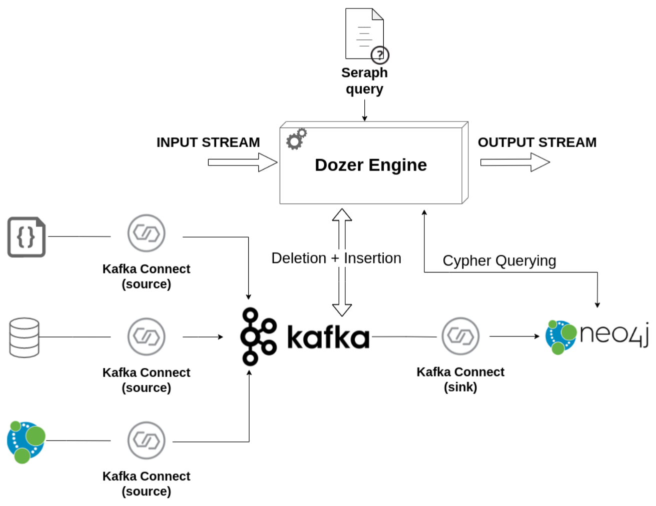


Figure 4.2: The main principle of the Dozer design

Under the assumption that our system could ingest graph data coming from several data sources in JSON-PG format [14], we focused on the *Kafka Sink Connector*. It can work in several ways:

- By providing **Cypher template**
  It works with template Cypher queries stored into properties with a specific format. Each Cypher template must refer to an event object that will be injected by the Sink;
- By ingesting the events emitted from another Neo4j instance via the **Change Data Capture (CDC)** module
- By providing a **pattern extraction to a JSON or AVRO file**
- Managing a **CUD file format**
  The CUD file format is JSON file that represents Graph Entities (Nodes/Relationships) and how to manage them in term of Create/Update/Delete operations.

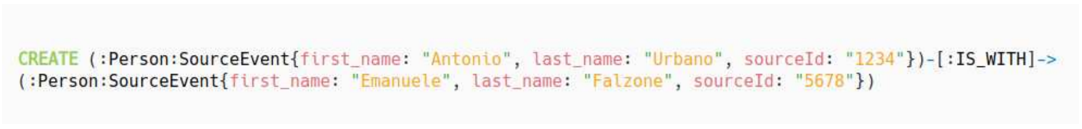
A typical design pattern is to make information in databases available in Kafka using *Change Data Capture (CDC)*, in conjunction with Kafka's Connect API to extract data from the database. CDC [43, 4, 58] enables the monitoring and collection of data changes, as well as updating a target system with only the data that has changed from the source system. A well-known CDC system is *Debezium*[1],

[1]https://debezium.io/

an open-source distributed platform for change data capture. Debezium is a set of distributed services to capture changes in a database and then uses Kafka and Kafka Connect to make the change data available scalably and reliably to multiple downstream systems.

CDC enables to stream every single event occurring on a database into Kafka at very low latency and low impact. For this reason, it has emerged as an ideal solution to design event-driven architectures that provide real-time or near-real-time movement of data by moving and processing data continuously as new database events occur. It is ideal for high-velocity data environments where time-sensitive decisions must be taken, since it enables low-latency, reliable, and scalable data replication.

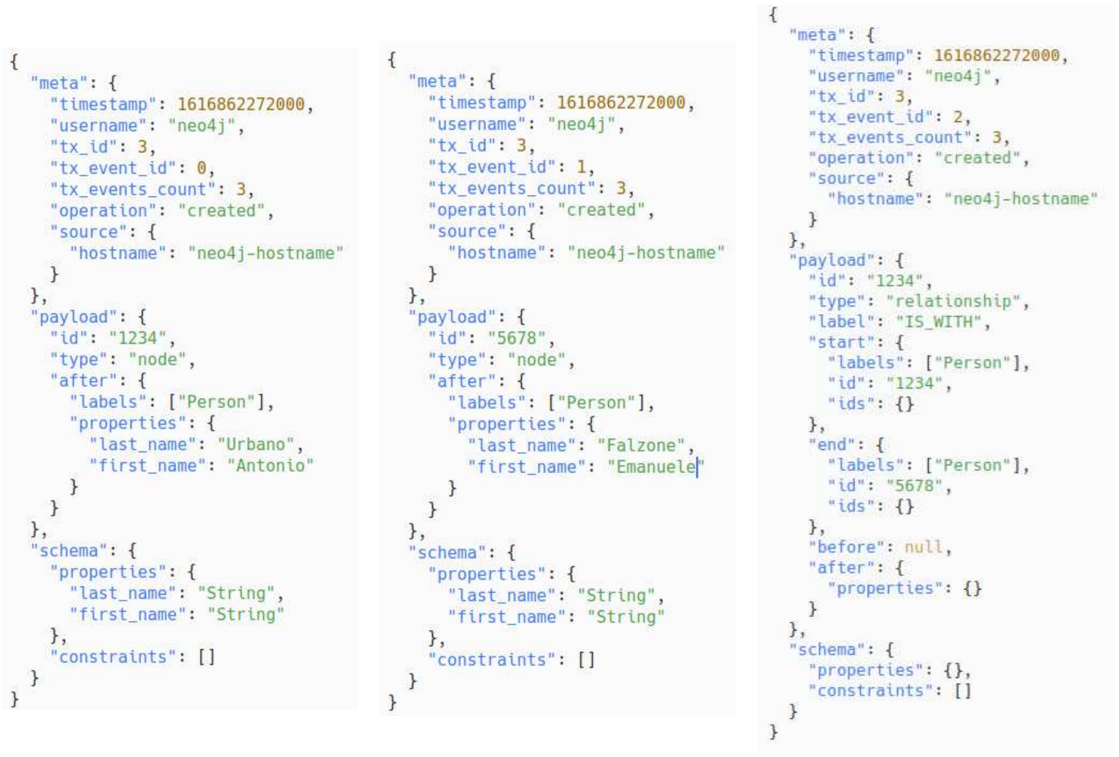
Using the *Kafka Sink Connector*, combined with the CDC module, allows us to overcome the limits imposed by the transactional nature of Neo4j, which is opposed to the strict latency requirements of stream processing applications. Moreover, with the plugin and the CDC module, we can automatically ingest events emitted from other Neo4j instances. Figure 4.3 shows an example of streams event that will be projected into the related graph entity as the following Cypher "CREATE" statement:



```
CREATE (:Person:SourceEvent{first_name: "Antonio", last_name: "Urbano", sourceId: "1234"})-[:IS_WITH]->
(:Person:SourceEvent{first_name: "Emanuele", last_name: "Falzone", sourceId: "5678"})
```

Owing to such considerations, we have re-designed the GSP4J-based architecture in Figure 3.3 to fit our design choices. Figure 4.4 illustrates an overview of the Dozer architecture and highlights the main differences between the two architectures. In line with the strategy adopted by Falzone et al. for the first prototype, we used the JSON-PG data format, which allows us to ingest graph-native data, possibly coming from the major graph databases. However, on the basis of the foregoing considerations, our streaming application needs to work with CDC event streams, and therefore we need a component in charge of converting JSON-PG data format in CDC events, which will become the new input stream of our pipeline.

Moreover, the captured data changes denoting the dynamic evolution of the data according to the window operator will be propagated on a Neo4j instance using the Kafka Connector Sink. Unlike the previous model, the Neo4j instance is no longer in-memory as part of the application but rather is an external system with which Dozer communicates. Finally, as it was before, our streaming application runs the Cypher sub-query over the portion of the graphs extracted by the window operator. The result is then published in a dedicated Kafka topic using



```
{
  "meta": {
    "timestamp": 1616862272000,
    "username": "neo4j",
    "tx_id": 3,
    "tx_event_id": 0,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname": "neo4j-hostname"
    }
  },
  "payload": {
    "id": "1234",
    "type": "node",
    "after": {
      "labels": ["Person"],
      "properties": {
        "last_name": "Urbano",
        "first_name": "Antonio"
      }
    }
  },
  "schema": {
    "properties": {
      "last_name": "String",
      "first_name": "String"
    },
    "constraints": []
  }
}
```

```
{
  "meta": {
    "timestamp": 1616862272000,
    "username": "neo4j",
    "tx_id": 3,
    "tx_event_id": 1,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname": "neo4j-hostname"
    }
  },
  "payload": {
    "id": "5678",
    "type": "node",
    "after": {
      "labels": ["Person"],
      "properties": {
        "last_name": "Falzone",
        "first_name": "Emanuele"
      }
    }
  },
  "schema": {
    "properties": {
      "last_name": "String",
      "first_name": "String"
    },
    "constraints": []
  }
}
```

```
{
  "meta": {
    "timestamp": 1616862272000,
    "username": "neo4j",
    "tx_id": 3,
    "tx_event_id": 2,
    "tx_events_count": 3,
    "operation": "created",
    "source": {
      "hostname": "neo4j-hostname"
    }
  },
  "payload": {
    "id": "1234",
    "type": "relationship",
    "label": "IS_WITH",
    "start": {
      "labels": ["Person"],
      "id": "1234",
      "ids": {}
    },
    "end": {
      "labels": ["Person"],
      "id": "5678",
      "ids": {}
    },
    "before": null,
    "after": {
      "properties": {}
    }
  },
  "schema": {
    "properties": {},
    "constraints": []
  }
}
```

Figure 4.3: Example of CDC "Create" events

the JSON-PG format. Figure 4.5 better depicts the internal structure of the Dozer application and its workflow, which consists of three main phases:

1. Once a Seraph query is registered, the application extracts the *EMIT Range*, to define a *Timestamp To Sync*, which will be the input of the next phases. The *EVERY* operator specifies the frequency of the evaluation process and the timestamp to sync corresponds to the several exact evaluation time instants. So, assuming that the first record arrives at 8:17 AM and the EVERY operator equals PT2M, the first time to sync is at 8:17 AM, then the second evaluation time will be at 8:19 AM, and so on. From a logical level perspective, a streaming model uses the concept of a *Tick* to drive the system in taking actions over input streams. Botan et al. in [10] define a Tick in three ways:

   - **Tuple-driven (DD)**, where each tuple arrival causes a system to react.
   - **Time-driven (TD)**, where the progress of the real time causes a system to react.
   - **Batch-driven (BD)**, where either a new batch arrival or the progress

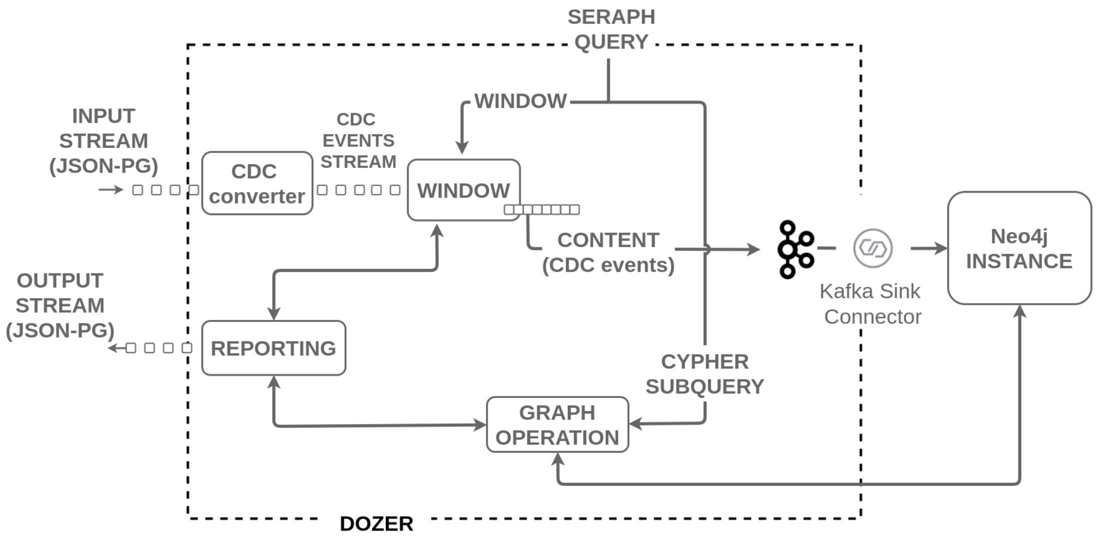


Figure 4.4: Dozer architecture

of the time causes the system to react.

Dozer uses a *tuple-driven* approach. DD models typically provide lower latency than the TD and BD models for the query computation. On the other hand, the record-at-a-time model requires state maintenance for all operators with record-level granularity. This behavior obstructs system throughput and brings much higher latencies when recovering after a system failure [54]. However, as better explained in Section 2.6.2.1, Kafka Streams handles out-of-order data, with low latency and high throughput *record-at-a-time processing*. This key feature, combined with the high fault tolerance of Apache Kafka during recovering, made us choose a tuple-driven model, which allows us to reduce the latency impact due to the transactional behavior associated with Neo4j.

2. The second step involves the core function of the pipeline. During this phase, the streaming application creates and maintains the time-varying graphs to capture the dynamic evolution of the data flowing inside the query engine. It extracts the portion of the subgraphs data, which is recreated on the Neo4j instance via a set of CDC event streams.

3. Finally, the engine communicates with the Neo4j instance to run the Cypher subquery on the portion of data and produce the result on the output Kafka topic.

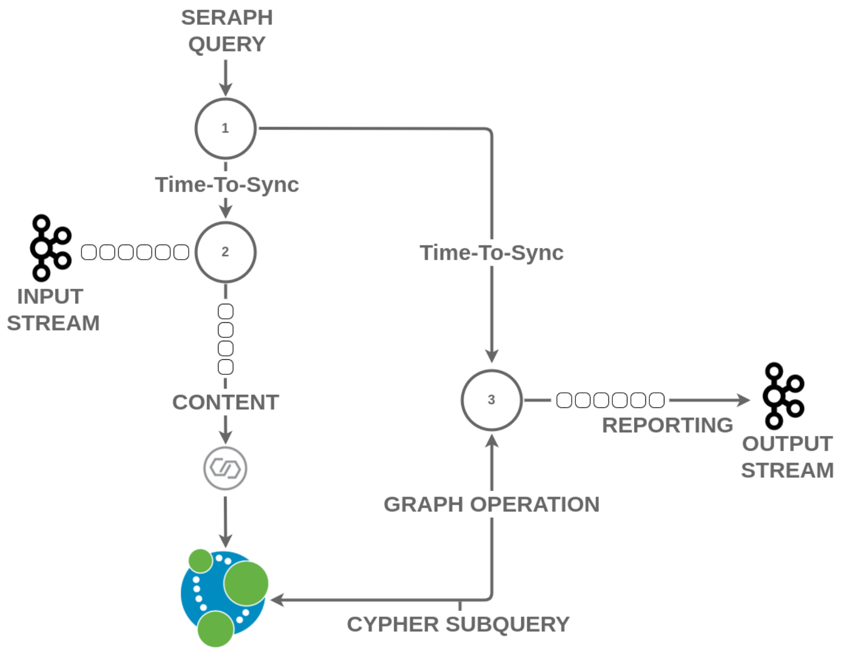


Figure 4.5: Dozer internal workflow

Each described phase is carried out by one or more components. A more detailed description of the internal design will be presented in the next Section, and the actual implementation will be discussed in detail in Chapter 5.

4.2 Internal System Design

The objective of this Section is to provide a high-level overview of the Dozer internal structure, outlining its main components and how they work together. Section 4.2.1 describes the main modules characterizing the workflow of our engine. Then, Section 4.2.2 details the behavior of some of the main components whose implementation will be detailed in Chapter 5.

4.2.1 Dozer Topology

Dozer consists of three main decoupled modules which, working together, are in charge of simulating the workflow presented in Figure 4.5. Figure 4.6 depicts an overview of the Dozer's modules. We have:

- **JSON-PG to CDC converter**
  It consumes from the input stream source topic defined in the Seraph query

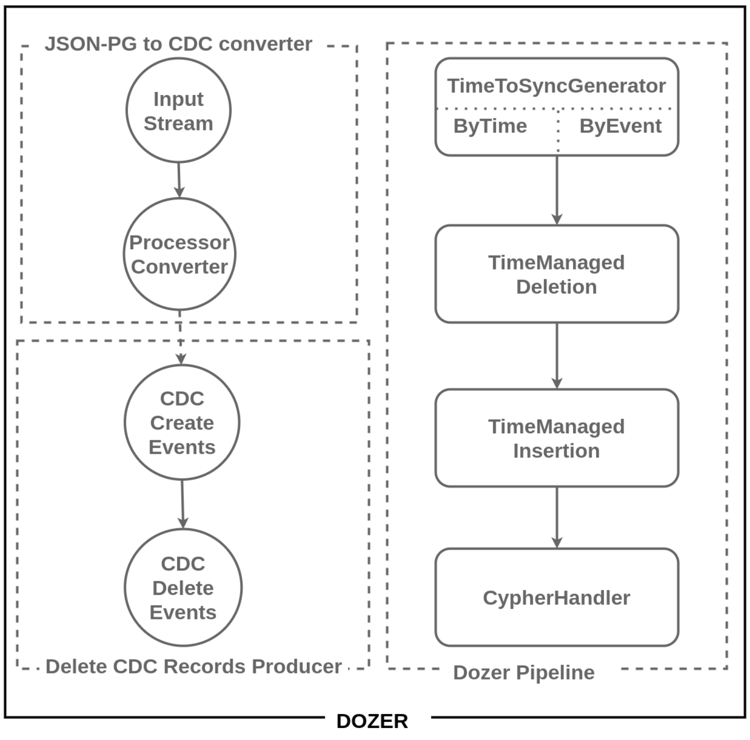


Figure 4.6: Three main Dozer's modules

and converts the data in CDC format. The *Converter Processor* processes each record and converts it from JSON-PG format to CDC format. Finally, a Kafka Producer sends the "create" event record into the appropriate topic that will be the source of the next module.

- **Delete CDC records producer**
  The goal of this module is to produce CDC "delete" events. It consumes the CDC "create" records produced by the converter module and produces "delete events" with a proper custom timestamp. To generate the timestamp associated to the "delete" events, we can have two policies which depends on the window type:

  - **Window Time Range**
    The window time range defines slices of times on which extract portions of the property graph. Let's assume a 2 hours wide time-based sliding window and that a CDC "create" event arrives at 7:52 AM; we need to create a CDC "delete" events at 9:52 AM. Kafka is **pull-based**, i.e. different consumers can consume the messages at different paces. On the

other hand, producers cannot decide at which time to send the messages. To capture the dynamic evolution of the time-varying graphs data we need a way to postpone the CDC “delete” events.

  - **Window Event Range**
    In the case of an *event-based* window, the window range defines the last N events of the streams forming the portion of the property graph to consider. Let’s assume an event range of 5 events, this means that we must maintain a subgraph of the last 5 events. So, once the threshold is exceeded (in our example 5 events), every time a new event arrives we need to remove the oldest of the five and add the most recent one.

- **Dozer Pipeline**
  This is the key module of the Dozer engine. It is the actual implementation of the workflow depicted in Figure 4.5, in which a component generates the evaluation time instants, some components are in charge of the maintenance process and capturing the dynamic evolution of the data, and a component run Cypher subquery on the portion of generated sub-graphs.

The activity diagram in Figure 4.7 describe the complete workflow of each single modules. The first module consumes the graph data in JSON-PG format from the input stream and converts it into CDC “create” events. The CDC “create” stream will be the input of the second module, in charge of traducing the “create” records into CDC “delete” events. The generation of the “delete” events depends on the window type that can be either *time-based* or *event-based*. With a time-based window range, we produce “delete” records with a timestamp in the future; with an event-based window, we count the number of processed “create” events and we generate the associated “delete” records only once the queue is full. Finally, the last module is in charge of the graph maintenance and running the Cypher sub-query on the portion of the subgraph defined by the window. Once the Seraph query has been parsed, a component will generate the *time-to-sync* corresponding to the several exact evaluation time instants, defined by the *EVERY* operator. Then, using the output of the first two modules, the engine will delete and insert records up to the time-to-sync. This operation generates the portion of the graph up to the evaluation time instant defined by the time-to-sync. Finally, the last component runs the Cypher query on the portion of the subgraph. This entire process continues until a defined *end-instant*.

4.2.2 Dozer Components

In this section, we present how some of the main components previously introduced cooperate. Subsequently, in Chapter 5, we better detail the internal functioning

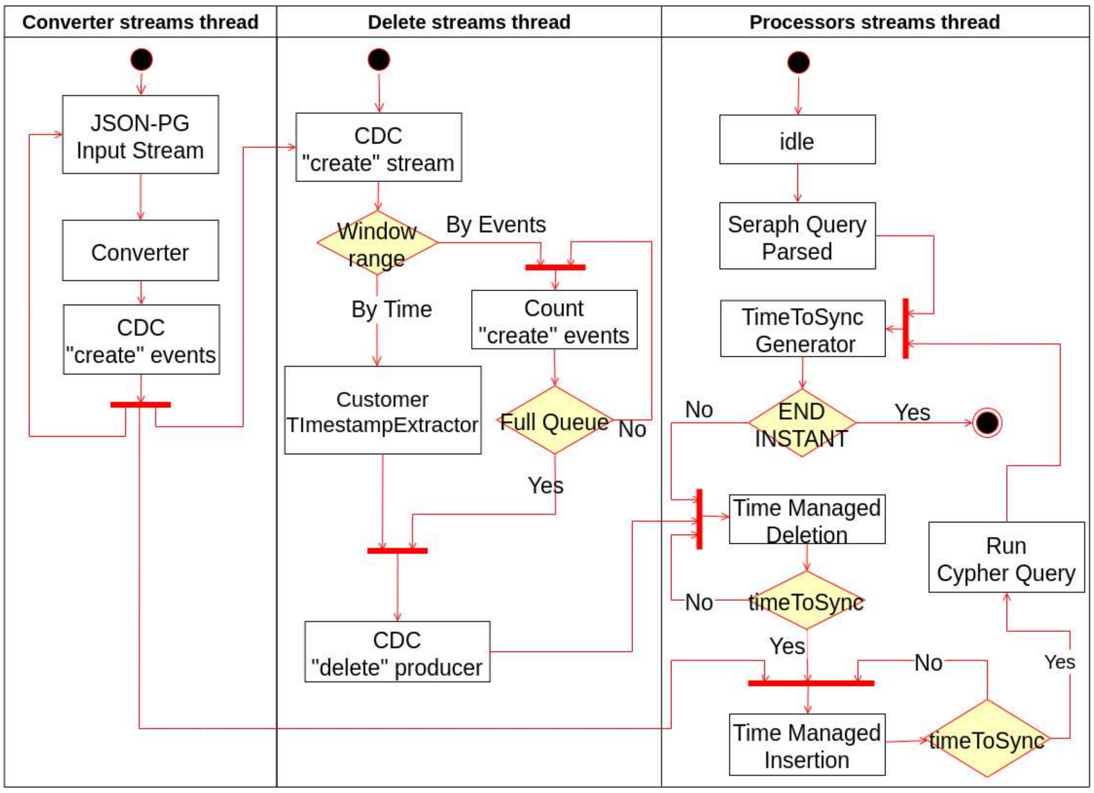


Figure 4.7: Activity diagram of the Dozer workflow

of some of these components. The component diagram in Figure 4.8 depicts the internal structure of our system, presenting the key components and how they are wired together. The first two blocks, namely *"JSON-PG To CDC Converter"* and *"CDC Delete Production"*, have been widely analyzed previously. In the following, we will focus on the description of the components of the other two blocks, namely *"Dozer's processors"* and *"Dozer's engine"*, which corresponds to the *"Dozer Pipeline"* module presented in Figure 4.6. The several processors exchange the *CurrentAgent* item, a sort of token of the current running processor. Every time a processor ends its operations, sends the token to the next processor designed to start. According to the workflow defined in Figure 4.6 and to the activity diagram in Figure 4.7, the first component designed to start is the *Time-To-Sync Generator*, in charge of generating the first evaluation time instant according to the *EVERY* operator. Once it finishes, it sends the *currentAgent* token to the *TimeManagedDeletion Processor*, in charge of performing deletion up to the defined time-to-sync. The following component is the *TimeManagedInsertion Processor* that insert "create" records up to the defined time-to-sync. Finally, the *CypherHandlerProcessor* is triggered to run the Cypher subquery on the subgraph. Once it gets the Cypher result, it sends the *currentAgent* token to the *Time-To-Sync Generator* which updates the *time-to-sync* and the working cycle starts again. In the following, we explain how the *timestamp-to-sync* is generated and how the components capturing the graph data evolution work.

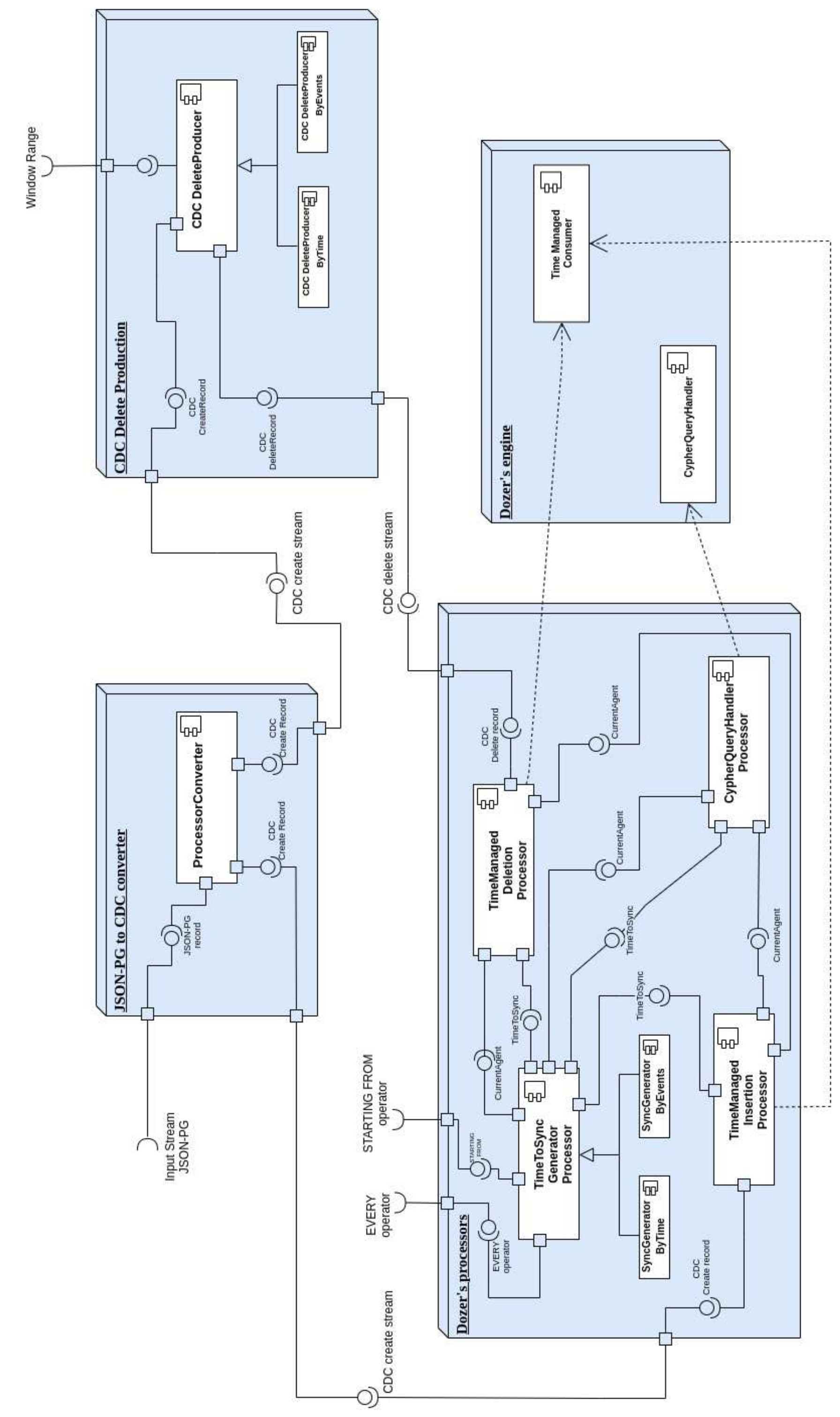


Figure 4.8: Dozer's component diagram

4.2.2.1 Timestamp To Sync Generator Processor

All the Dozer components work by synchronizing themselves to a timestamp, called *time-to-sync*. This parameter corresponds to the evaluation time instant and is updated at a frequency determined by the *EVERY* operator. Seraph language admits the definition of the latter either in terms of time or number of events.

- **Sync Generator By Time**
  In the case the *EVERY* operator is defined in terms of time interval, the processor will add such an interval to the last recent timestamp. The first time, it is triggered after the Seraph query has been parsed, and it initializes the *time-to-sync* according to the *STARTING FROM* clause in the Seraph query. The latter can be either:

  - *Earliest*, and the timestampToSync must be initialized to the timestamp of the first record in the input stream;
  - *Latest*, and the *time-to-sync* must be initialized to the timestamp of the last record in the input stream; or
  - A specific time interval, which corresponds to the *time-to-sync* initialization.

  Subsequently, the *SyncGenerator* will be triggered after the *CypherHandler* completion, and it will update the *time-to-sync* by adding the defined time interval to the last valid timestamp.

- **Sync Generator By Time**
  The *EVERY* operator can be also defined in terms of number of events after which update the evaluation time instant. It basically works according to the same principle of the *SyncGenerator By Time*. The first time, it is triggered after the Seraph query has been parsed, and it initializes the *time-to-sync* according to the *STARTING FROM* clause defined in the Seraph query. Subsequently, it will be triggered after the *CypherHandler* completion, but with a different update's policy. In this scenario, we need to count the number of input events we process and, once the counter is equal to the *EVERY* value, we update the *time-to-sync*, by assigning the timestamp of the last read record.

4.2.2.2 TimeManaged Processor

This component is in charge of maintaining the time-varying graphs, capturing the dynamic evolution of the data. We can decouple this process into two phases, carried out by two independent components:

1. **Deletion**
   After the definition of the *time-to-sync*, we need to extract the portion of the graph defined by the window. In Section 4.2.1, we explained that CDC "delete" events are produced with a custom timestamp, according to the *WINDOW RANGE* operator. This component consumes from the CDC "delete" events topic up to the *time-to-sync* timestamp in order to delete all the events which do not fall in the window scope.

2. **Creation**
   After the deletion, we need to capture the events in the window, up to the *time-to-sync*, and reproduce the CDC "create events" with the proper customized timestamp.

With this engine, we are mainly interested in capturing the dynamic evolution of graphs over the relationships which are the key elements of graph DBs. With this assumption, we handle the insertion and deletion of the relationships; while we keep all the nodes we ingest from the input stream. If we wanted to delete also the nodes, at every evaluation time instant we would have to check that a specific node didn't have any relationships with other nodes within the window scope. This computation would be expensive. According to this model, we improve the performance, by significantly reducing the maintenance costs at expenses of a slight increase of running Cypher queries. Further consideration of the performances will be covered in Chapter 6.

Chapter 5

Implementation Experience

This chapter aims to provide a technical overview of the architecture we proposed in this thesis work. We describe, helping with some code and pseudo-code snapshots, how the components presented in Chapter 4 have been implemented.

5.1 CDC “Delete” Records Producer

In Section 4.2.1, we have described the goal of this module. It consumes the CDC “create” records produced by the converter module and produces “delete” events with a proper custom timestamp, which depends on the window type.

5.1.1 Time Range Window

Kafka is **pull-based**, i.e. different consumers can consume the messages at different paces. On the other hand, producers cannot decide at which time to send the messages. To capture the dynamic evolution of the time-varying graphs data we need a way to postpone the CDC “delete” events. One possibility could have been to create an internal queue, wait with an internal countdown and then publish the deletion record at the proper time. Such an in-memory solution would cause scalability problems, contradicting our requirements.

The solution we adopt consists of producing CDC “delete” events, as soon as a “create” events arrives, with an associated timestamp in the future. Listing 5.1 shows the implementation of the Customer Extractor in which we change the record timestamp by adding a delay equal to the window size. Then, a dedicated component will be in charge of consuming this record at the proper timestamp and reproducing it on the Kafka Topic from which the Kafka Connector will sink in Neo4j.

Listing 5.1: Customer Extractor

```java
public class CustomerExtractor implements TimestampExtractor {
    long windowTimeRange;

    public CustomerExtractor(long windowTimeRange) {
        this.windowTimeRange = windowTimeRange;
    }

    @Override
    public long extract(ConsumerRecord<Object, Object> consumerRecord, long l) {
        return consumerRecord.timestamp() + this.windowTimeRange;
    }
}
```

5.1.2 Event Range Window

In this case, the window range defines the last N events of the streams to consider. We need an internal queue to count the number of events we have processed. Pseudo-code 1 outline the key phases. For fault-tolerance purpose, we store the event queue in the Kafka processor local state store.

Algorithm 1 CDC Delete Producer By Event Pseudo-code

```
1: $c \leftarrow$ CDC “create” record
2: $\delta \leftarrow produceDeleteCDC(c)$
3: $q \leftarrow localStore.getQueue()$
4: $q.add(\delta)$
5: if $q.size() > WINDOW\ RANGE$ then
6:     $h \leftarrow removeHead(q)$
7:     Publish $h$ on Kafka Topic
8: end if
9: $localStore.update(q)$
```

5.2 Dozer Processors

In Section 4.2.2, we described how the several processors exchange a sort of token of the current running processor. Figure 5.1 illustrates the actual implementation of this module. To simulate such behaviour we implemented a *Kafka Topology* in which each processor consumes and produces from the same topic, let's call it *Worflow Topic*. Moreover, Kafka Topology allows us to use local state stores associated with the processors. In this way, we could implement a *stateful* engine, able to recover its state in case of failure.

Listing 5.2 shows the definition of *CurrentAgent* class, in which we store the name of the component currently performing operations, the status of the components (started or completed) and the timestamp to sync.

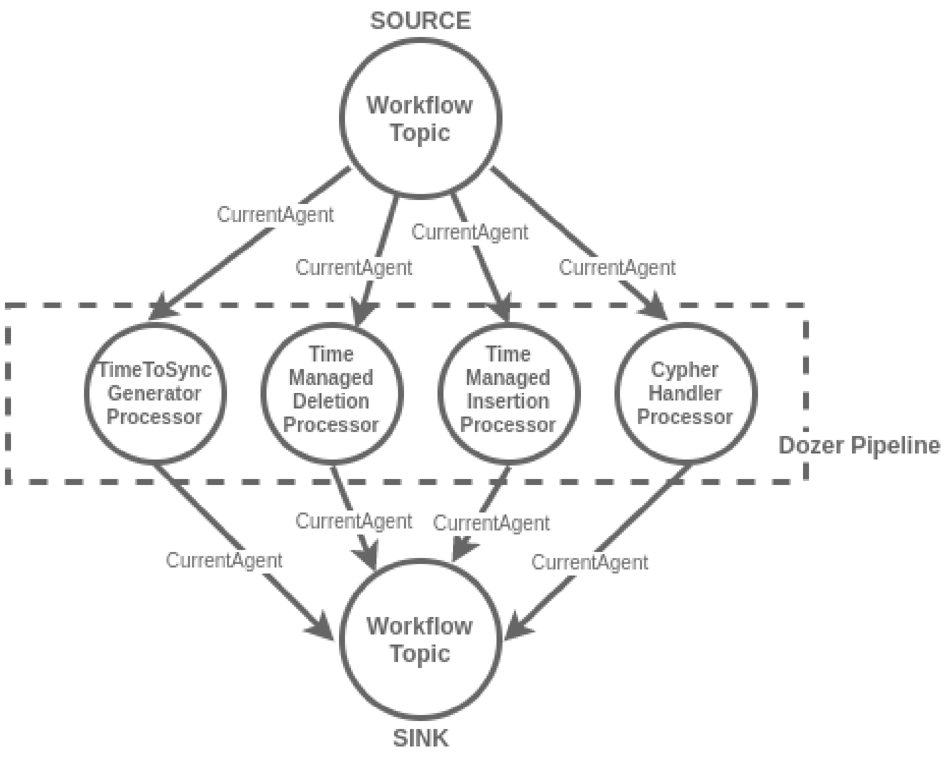


Figure 5.1: CurrentAgent token and Processors Topology Implementation

Listing 5.2: Current Agent class definition

```java
public class CurrentAgent implements Serializable {

    private String agentName;
    private String status;
    private Long timestampToSync;
    /**
     * @param agent      the class name of the component currently performing operations
     * @param status     the status of the current components - may be "started" or "completed"
     * @param timestampToSync
     */
    @JsonCreator
    public CurrentAgent(@JsonProperty("agentName") String agent,
                        @JsonProperty("status") String status,
                        @JsonProperty("timestampToSync") Long timestampToSync){
        this.agentName = agent;
        this.status = status;
        this.timestampToSync = timestampToSync;
    }

    public String getAgentName() { return agentName; }
    public String getStatus() { return status;  }
    public Long getTimestampToSync() { return timestampToSync; }
}
```

Every time a processor ends to perform its operations, it updates the local store and publishes on the *Workflow Topic* a *CurrentAgent* record indicating its finish. Each processor will also receive such a record, because the Workflow Topic corresponds also to the source topic from which the processors consume. At the *CurrentAgent* record arrival, only the processor designed to perform its operation is triggered, and once it finishes, it updates the state store and publishes its completion.

5.2.1 Time-To-Sync Generator

As already described during the design phase, all the components in Dozer work by synchronizing themselves to *time-to-sync* timestamp. This parameter corresponds to the evaluation time instant, and it is updated with a frequency defined by the *EVERY* operator. Pseudo-codes 2 and 3, outline the behavior of the *SyncGenerator* components, when the *EVERY* operator is defined respectively by events or by time.

Algorithm 2 Sync Generator By Events Pseudo-code

```
1: read currentAgent from Workflow Topic
2: if currentAgent = PARSED QUERY then
3:    initialize time-to-sync
4:    update currentAgent
5:    publish currentAgent to Workflow Topic
6: else if currentAgent = CypherHandler then
7:    $\Delta \leftarrow localStore.getLastOffset()$
8:    counter= 0
9:    while counter < EVERY do
10:       read CDC “create” record from $\Delta$
11:       update $\Delta$
12:       counter++
13:   end while
14:   $\tau \leftarrow$ timestamp of last read record
15:   $localStore.update(\Delta)$
16:   time-to-sync = $\tau$
17:   update currentAgent
18:   publish currentAgent to Workflow Topic
19: end if
```

Algorithm 3 Sync Generator By Time Pseudo-code

```
1: read currentAgent from Workflow Topic
2: if currentAgent = PARSED QUERY then
3:   initialize time-to-sync
4:   update currentAgent
5:   publish currentAgent to Workflow Topic
6: else if currentAgent = CypherHandler then
7:   time-to-sync += EVERY operator
8:   update currentAgent
9:   publish currentAgent to Workflow Topic
10: end if
```

In the first pseudo-code, when the *EVERY* operator is defined in terms of events, we use a local state store associated with the processor to store the last *offsetToRead* to be resilient in case of failure.

5.2.2 Time Managed Consumer

This module is in charge of maintaining the time-varying graphs, capturing the dynamic evolution of the data. As described in the design phase, we can decouple this process into two components, one managing the deletion and the following managing the insertion of records up to the *time-to-sync*. Both components have the same behavior. The only difference is the type of records they consume and reproduce, respectively CDC "delete" and "create". The following pseudo-code refers to the deletion component:

Algorithm 4 Time Managed Deletion Pseudo-code

```
1: read currentAgent from Workflow Topic
2: if currentAgent = SyncGenerator then
3:   $\Delta \leftarrow localStore.getLastOffset()$
4:   while record.timestamp() < time-to-sync do
5:     read CDC "delete" record from $\Delta$
6:     produce on "Neo4j Topic"
7:     update $\Delta$
8:   end while
9:   $localStore.update(\Delta)$
10:  update currentAgent
11:  publish currentAgent to Workflow Topic
12: end if
```

It starts reading the records with the customized timestamp in order to reproduce them on the topic connected with Neo4j at the proper time. Also with this processor, we use a local store to recover, in case of failure, the offset where starts to read.

5.2.3 Cypher Handler

This is the last component of the workflow before the *time-to-sync* is updated, and it is triggered after the *TimeManaged Insertion* completion. It uses the *Neo4j Java Driver API*[1] to connect to the Neo4j instance and run the Cypher subquery.

Finally, the result of the Cypher query is converted in JSON-PG format and sent on the output stream topic via a Kafka Producer.

[1]https://neo4j.com/developer/java/

Chapter 6

Evaluation

This chapter contains an explanation of all the experiments done to evaluate the proposed system. Section 6.1 describes the purpose of the experiments, illustrating them, the used datasets, and the system that hosted them. Section 6.2 highlights the results of a set of experiments carried out to analyze the Dozer performances. Finally, Section 6.3 covers an analysis of the fault-tolerance of our engine.

6.1 Experimental Setup

This section provides an overview of the experiments we carried out to analyze the performance of our work. In Section 6.1.1, we introduce the test and describe the purpose of each of them. Then, in Section 6.1.2, we describe the query used for our experiments, as well as the used datasets. Finally, in Section 6.1.3, we give a brief description of the execution environment, presenting some of the used tools and the system that hosted our experiments.

6.1.1 Test Cases

All the experiments we are going to present are aimed at showing the industrial features for which this system was designed, such as high performance and fault-tolerance. The experiments can be divided into three groups, different for their purpose and the information they produce.

Firstly, Dozer's performances have been tested against a standard way of querying timestamped Property Graphs with Cypher. As a preliminary step, we create a dataset, better described in the next section, in JSON-PG format. Then, we use the same query to analyze the performance of the two ways of running queries over streaming graphs w.r.t. the window size. In this scenario, we tested both the total execution time over a predefined time horizon, as well as the time needed to the Neo4j server to have the result available and to consume it at each evaluation step. For the latter case, we used the Neo4j *ResultSummary* interface[1] to investigate the time needed by the Neo4j server to retrieve the result when the window size changes. For both the tests, we used time-based tumbling windows with different sizes and different time horizons. We will better define both parameters in the next section, in which we discuss the dataset and the executed query.

Then, we test the performance of the Dozer itself at different window sizes. To this purpose, we analyze the overhead of each component of the system w.r.t. window range changes. Moreover, we test how the system performs with different shaped datasets.

Finally, we evaluate the fault-tolerance of our system, studying its behavior in case of failure and the cost needed in the recovering phase.

In the performance analysis tests, we ran, for each window, five executions with Dozer and other five executions by querying timestamped Property Graphs with Cypher. The five executions over Dozer have been used also for evaluating the overhead of each component. For the fault-tolerance testing, we repeated ten runs with and without failures across a smaller time horizon. More information on each specific setting, however, will be presented in the dedicated section.

6.1.2 Datasets

In the following subsections, we present two different datasets we used for our experiments.

6.1.2.1 Linear-Shaped Dataset

In the first scenario, we create a dataset in JSON-PG format, simulating a linked list of nodes growing over time. The first node was created with a timestamp corresponding to the date '2021-01-01T00:00:00Z'. Then, every 500ms two new nodes, linked to the existing ones, enter the dataset. Considering a five-seconds wide tumbling window, Figure 6.1 and Figure 6.2 show respectively the dataset at the first and the second evaluation step. Each relationship in the dataset will have an increasing timestamp (e.g. the relationship between the first two nodes will have a timestamp corresponding to the date '2021-01-01T00:00:00.500Z', the relationship between the second and the third node will have a timestamp corresponding to the date '2021-01-01T00:00:01.0Z', and so on).

Figure 6.3 reports the query used for our experiments. We registered the query into the Dozer system, consuming data from the Kafka topic 'input-dataset-

[1]https://neo4j.com/docs/api/java-driver/current/org/neo4j/driver/summary/ResultSummary.html

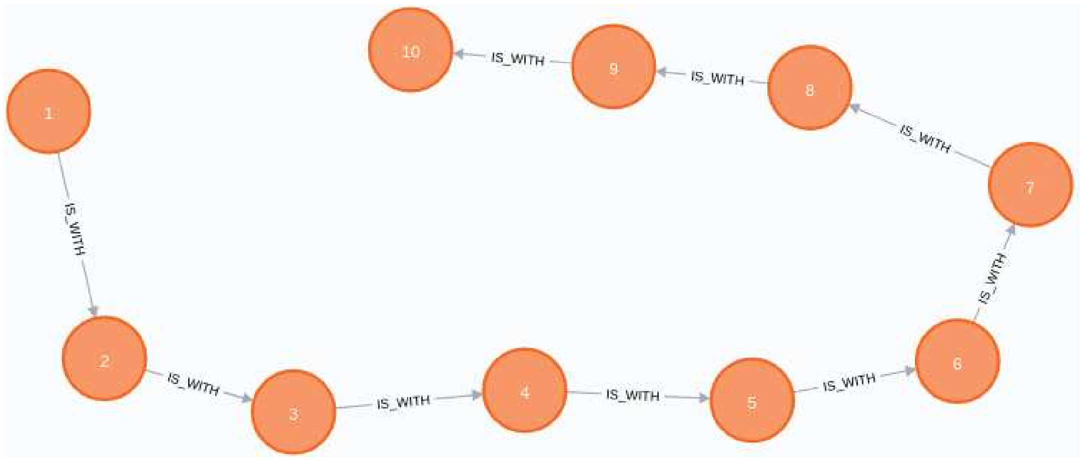


Figure 6.1: Linear dataset used for the experiments at first evaluation step

stream', containing the JSON-PG records described previously. We are interested in searching, at each evaluation step, all the people linked to another person. With **STARTING FROM Earliest**, we specify that the first evaluation time instant is related to the oldest event in the source stream, in our case corresponding to the record with date '2021-01-01T00:00:00.0Z'. The **WINDOW RANGE** PT5S, combined with the **EVERY** PT5S, defines a five-seconds wide tumbling window. The **EMIT SNAPSHOT** clause selects the *RStream* operator that directly emits the results without performing any additional operation. Finally, the **INTO** clause specifies to publish the result into the 'output-result-stream' Kafka topic.

The same query has been repeated for different window sizes (PT1S, PT5S, PT10S, PT15S, PT30S, PT45S, PT1M, PT5M, PT10M) over a time horizon of at most seven days long (from '2021-01-01T00:00:00Z' to '2021-01-08T00:00:00Z'), according to the considered experiment. However, further information about settings will be described later, in each dedicated section.

The same dataset depicted in Figure 6.1 has been created in Neo4j with Cypher, by temporally marking each relationship. Then, we simulate a tumbling window by running a Cypher query at each evaluation step. So, considering the example of a five-second wide tumbling window, Figure 6.4 shows the query at the first evaluation step. After that, new nodes enter the dataset up to '2021-01-01T00:00:10Z'. Then a new Cypher query requires the match for the nodes and relationships with timestamps between '2021-01-01T00:00:05Z' and '2021-01-01T00:00:10Z'.

With this configuration, Dozer maintains, at each evaluation step, only the portion of the graph defined by the window, eliminating all the relationships that do not fall in the window scope. On the other hand, by querying timestamped

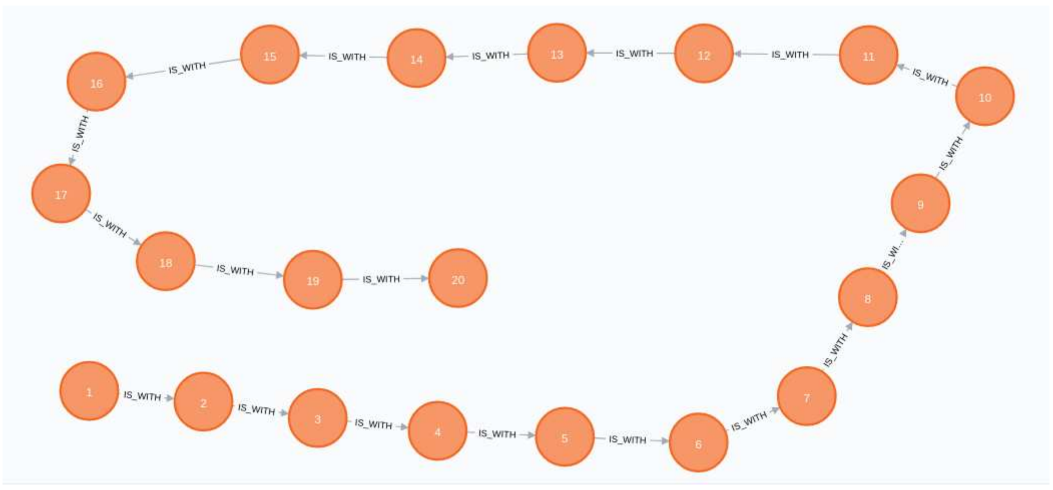


Figure 6.2: Linear dataset used for the experiments at second evaluation step

```
REGISTER QUERY <dozer_evaluation_query> {
FROM STREAM <input-dataset-stream>
STARTING FROM Earliest
WITH WINDOW RANGE PT5S
   MATCH (p:Person)-[r:IS_WITH]->(p2:Person)
   RETURN p,r,p2;
EMIT SNAPSHOT EVERY PT5S
INTO <output-result-stream>
}
```

Figure 6.3: Example of Seraph query used for the tests

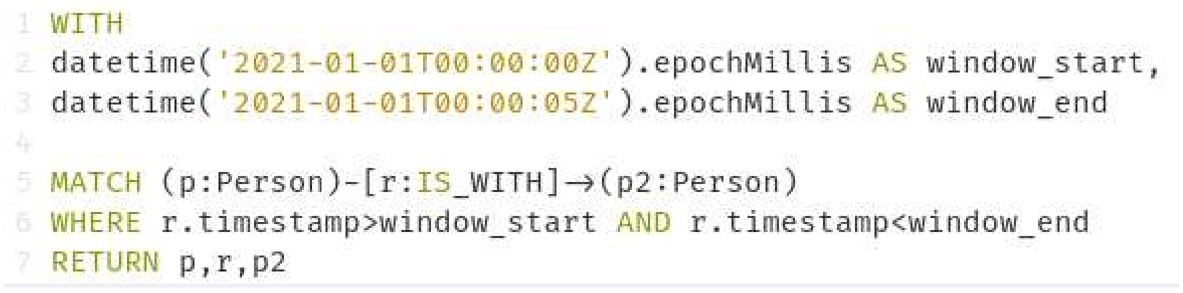
Property Graphs with Cypher, we work on a dataset that linearly grows as time passes. More consideration about this will be addressed in Section 6.2, in which we discuss the performances of the two approaches at different window sizes.

6.1.2.2 Star-Shaped Dataset

In the second scenario, we create a dataset in JSON-PG format, with a different shape. This dataset was used to inspect if the obtained performances that will be described later were dependent on the specific linear shape of the previous dataset.

We create a star-shaped dataset, in which every 250ms five nodes linked to an existing one enter the dataset. Figure 6.5 shows the dataset after one second.

In Section 6.2.2.2, we will describe how Dozer's performance changes w.r.t. the



```
WITH
datetime('2021-01-01T00:00:00Z').epochMillis AS window_start,
datetime('2021-01-01T00:00:05Z').epochMillis AS window_end

MATCH (p:Person)-[r:IS_WITH]→(p2:Person)
WHERE r.timestamp>window_start AND r.timestamp<window_end
RETURN p,r,p2
```

Figure 6.4: Cypher query simulating a five-seconds tumbling window

used dataset.

6.1.3 Execution Environment

All the introduced tests have been executed on an Amazon *EC2 t3.2xlarge*[1] instance. In particular, the Dozer system was running in a Docker container without any resources limitation: CPUs (8) and RAM (32 GBytes). During each test, Dozer consumes JSON-PG events from Kafka topics as fast as possible while executing the Cypher query according to the specifications. We use a Kafka cluster, composed of a single broker running on the Amazon instance. The Kafka broker hosted the input and the output stream topics, as well as the internal topics used by Dozer for the maintenance phases. Finally, the system communicates using the *Neo4j Kafka Connect plugin*[2] to sink data to a dockerized Neo4j instance[3].

For the timestamped Property Graph scenario, on the other hand, we constructed nodes and relationships directly on the same Neo4j instance[3] at each evaluation time indicated by the window definition, as described in the previous section.

6.2 Performance Evaluation

In this section, we present a set of experiments aimed at evaluating the performance of our system. In Section 6.2.1, we present the analysis we made to evaluate the complexity of Dozer against querying timestamped Property Graphs with Cypher. In Section 6.2.2, we address a performance analysis of the two systems, as well as a study about the performance of Dozer at different datasets. Finally, in Section

[1]https://aws.amazon.com/it/ec2/instance-types/t3/
[2]https://neo4j.com/labs/kafka/4.0/kafka-connect/
[3]neo4j:4.0.3

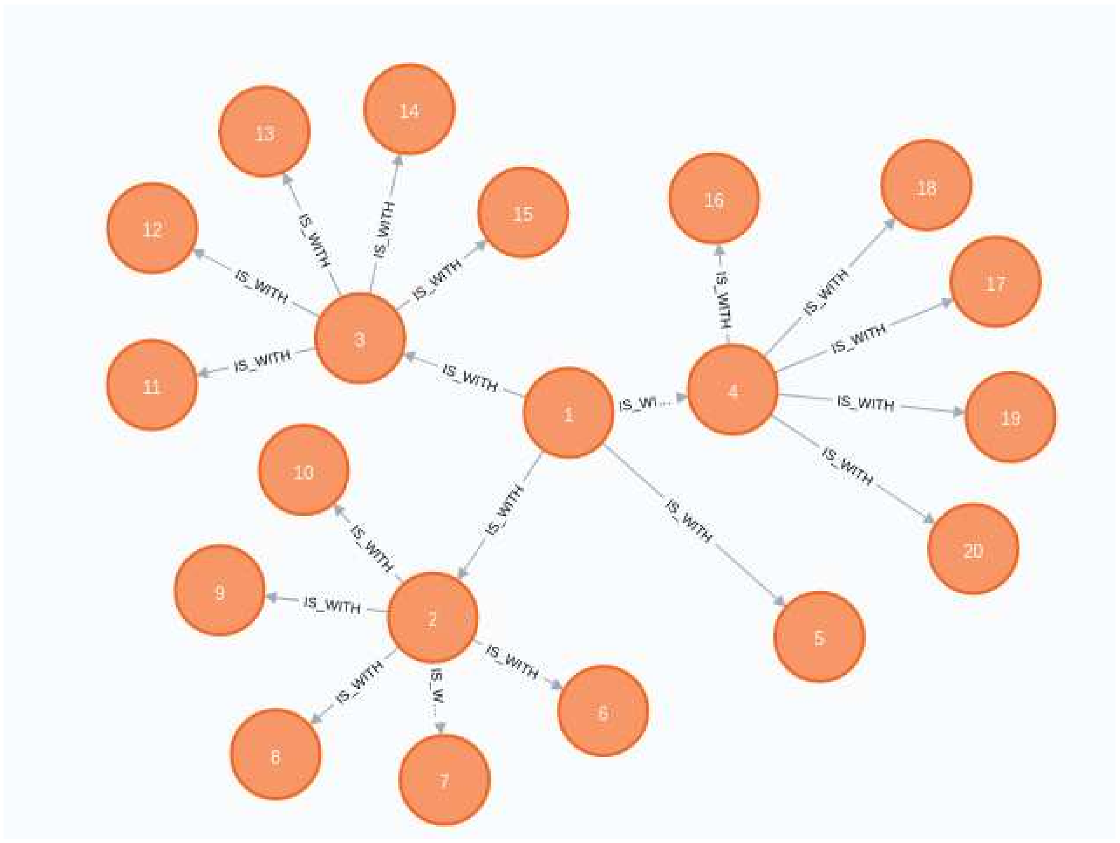


Figure 6.5: Star-shaped dataset

6.2.3, we analyze the overhead of each component of the system w.r.t. window range changes.

Each experiment will be explained in detail with some useful plots, helping us to highlight the evaluation outcomes.

6.2.1 Big-O Cost Analysis

First of all, we analyze the complexity of running the same queries over the two systems, namely Dozer and querying timestamped Property Graphs with Cypher (hereafter referred to as "Cypher"). For both scenarios and for each selected window, we ran five experiments over the linear dataset (see section 6.1.2.1). For smaller windows (i.e. PT1S, PT5S, PT10S, PT15S, PT30S, and PT45S), we run five experiments simulating a one-day long time horizon (i.e. from '2021-01-01T00:00:00Z' to '2021-01-02T00:00:00Z'). While for wider window (i.e. PT1M, PT2M, PT5M, and PT10M), the experiments were carried out over a seven-day long time horizon (from '2021-01-01T00:00:00Z' to '2021-01-08T00:00:00Z').

The system's complexity was determined by measuring the effort required by the Neo4j server to execute the *MATCH* query at each evaluation step. For this purpose, we used the Neo4j *ResultSummary* interface[1] to collect the time it took for the server to obtain the query results.

As introduced in section 6.1.2.1, Dozer manages streaming graphs natively, and therefore at each evaluation step, we have only the sub-graph corresponding to the portion selected by the window. On the other hand, with Cypher, we do not perform any maintenance, and therefore the property graphs continuously grow over time. In the latter case, at each evaluation step, we shrink the portion of the graph of our interest by filtering with the ***WHERE*** clause, as shown in Figure 6.4.

The linear dataset grows linearly over time in the number of nodes and relationships. Assuming that the complexity for running the *MATCH* query depends on the number of nodes and relationships present at each evaluation instant, we expected that Cypher's complexity grows linearly as well.

On the other hand, running the same *MATCH* query with Dozer requires a constant time complexity proportional to the number of nodes and relationships shrunk by the windows. However, in Chapter 4 we explain that with Dozer, we keep all the nodes we ingest from the input stream and we handle only the insertion and deletion of the relationships because we are mainly interested in capturing the dynamic evolution of graphs over the relationships which are the key elements of graph DBs. With this assumption, we reduce the maintenance costs at expenses of a slight increase of running Cypher queries.

Because of the foregoing consideration, the time complexity analysis led us to the following evaluation:

> ***Time Complexity Analysis.*** Let Γ be a finite, discrete, ordered sequence of $k \in \mathbb{N}$ time instants $(t_1, t_2, ..., t_k)$, where $t_i \in \mathbb{N}$; and let ρ be the production frequency (e.g. in our scenario two nodes and relationships per second). Additionally, let:
>
> - $G_{\mathbb{W}}$ be the portion of Property Graph defined by a time-based sliding window of widith α, so that at each time instant t, $G_{\mathbb{W}}(t)$ contains the content of the time-based window $W = (t - \alpha, t]$;
> - $r_i = \rho * i * \alpha$, be the number of relationships in the Property Graph at the instant t_i;
> - $n_i = \rho * i * \alpha$, be the number of nodes in the Property Graph at the instant t_i;

[1] https://neo4j.com/docs/api/java-driver/current/org/neo4j/driver/summary/ResultSummary.html

- $r_w = \rho * \alpha$, be the number of relationships in the portion $G_{\mathbb{W}}$, which is constant at each evaluation instant t_i;
- $R = r_k$ and $N = n_k$ respectively be the number of relationships and nodes at the final time instant.

We define the time complexity of executing the *MATCH* query at time instant t_i in Cypher and Dozer respectively as:

- $\delta_{Cypher}(t_i) \sim (r_i + n_i)$
- $\delta_{Dozer}(t_i) \sim (r_w + n_i)$

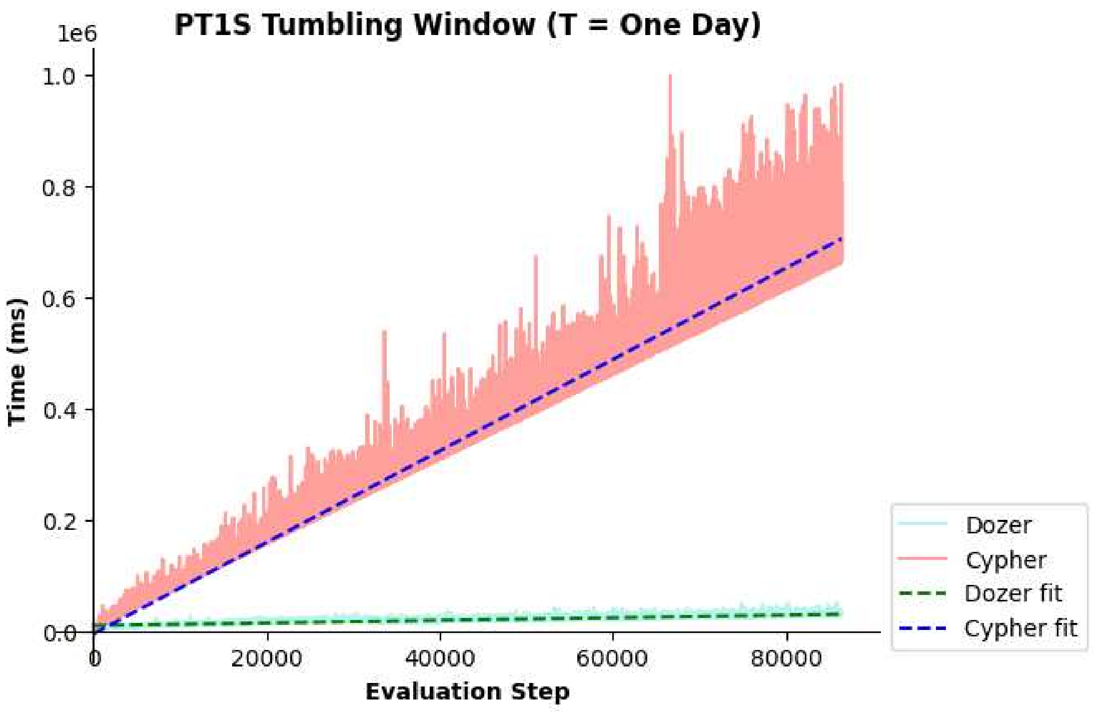
Then, extending the analysis to an infinite, discrete, ordered time domain $\mathcal{T} = (t_1, t_2, ...)$, where $t_i \in \mathbb{N}$, in the worst case we have a time complexity of:

- $O(N)$ for Dozer; while
- $O(N + R)$ for Cypher.

Finally, if we consider the time complexity only as function of the number of relationships **n**, we expect:

- A constant time complexity $\mathbf{O(1)}$ for Dozer; and
- A linear time complexity $\mathbf{O(n)}$ for Cypher.

Of course, as time passes, the dependence on the number of nodes cannot be overlooked. The plots in Figures 6.6, 6.7, and 6.8 depict the actual value and the result of the time complexity analysis for different window sizes over a one-day long time horizon; while in Figures 6.9 and 6.10, the plots associated of a seven-day long time horizon. The charts show the results of our experiments, which are coherent with our analysis. Moreover, we can see how the inefficiency of Cypher reduces as window sizes increase until it starts outperforming Dozer (see Figure 6.12). The frequency of evaluation instants decreases as the window size rises, while the amount of processed triples at each instant grows. We are not interested in understanding Neo4j's core execution strategy; however, it is optimized for bulk load ingestion. For larger windows, Cypher creates a large number of nodes and relationships in bulk, after which filtering on them becomes more efficient. On the other hand, with Dozer, we need to handle the maintenance phase. The deletion of some relationships and the accumulation of node instances introduce some inefficiency on the Neo4j server.



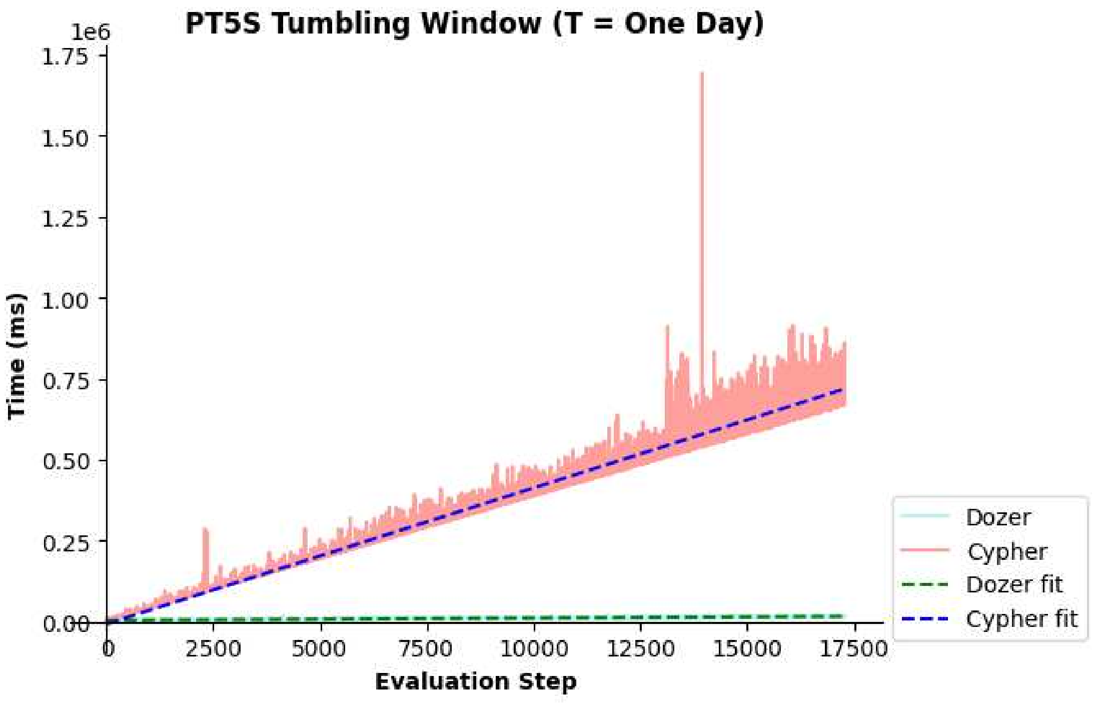


Figure 6.6: Big-O Analysis of Dozer vs Cypher Time Complexity at different window sizes (1-second, 5-seconds) over a one-day long time horizon

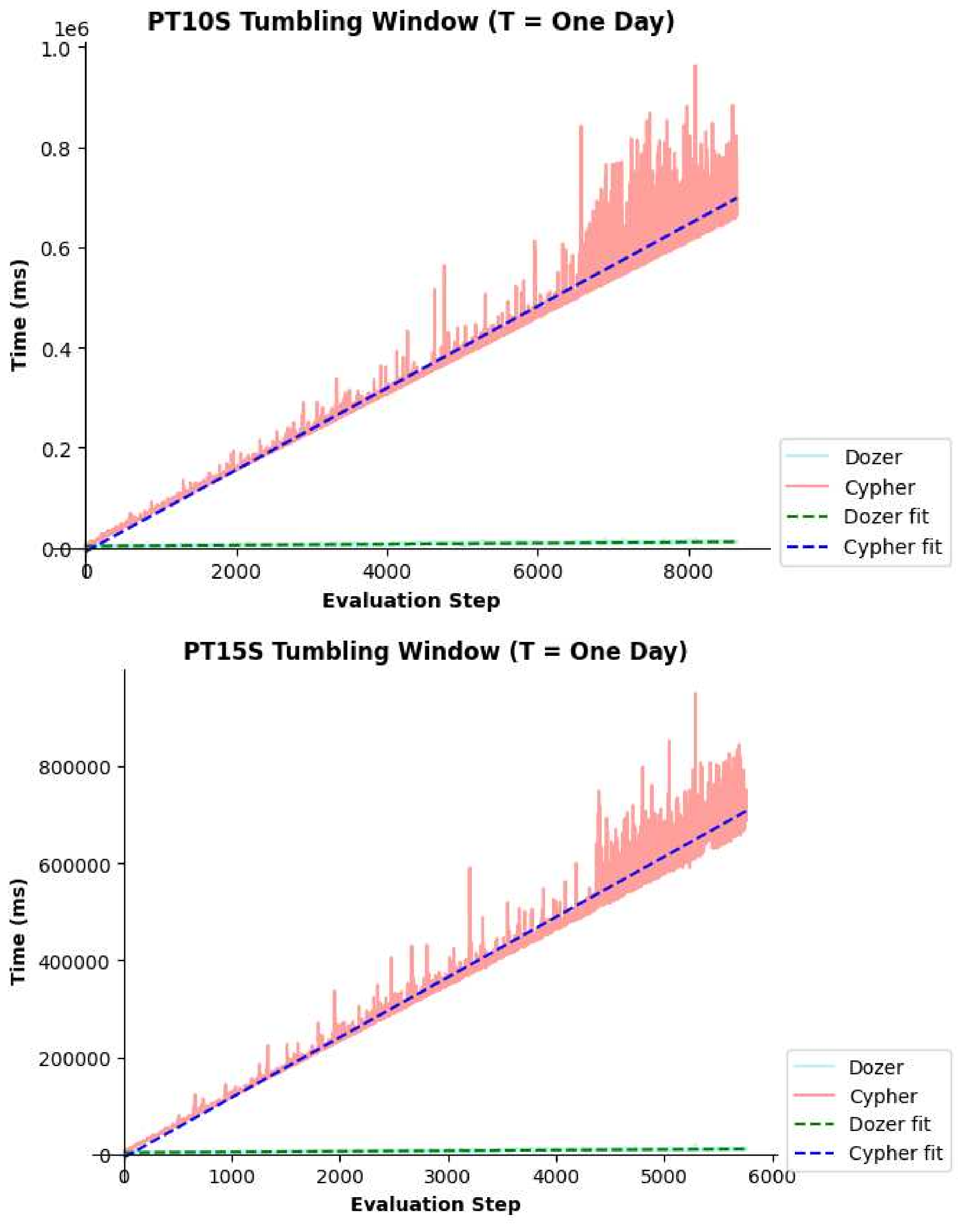
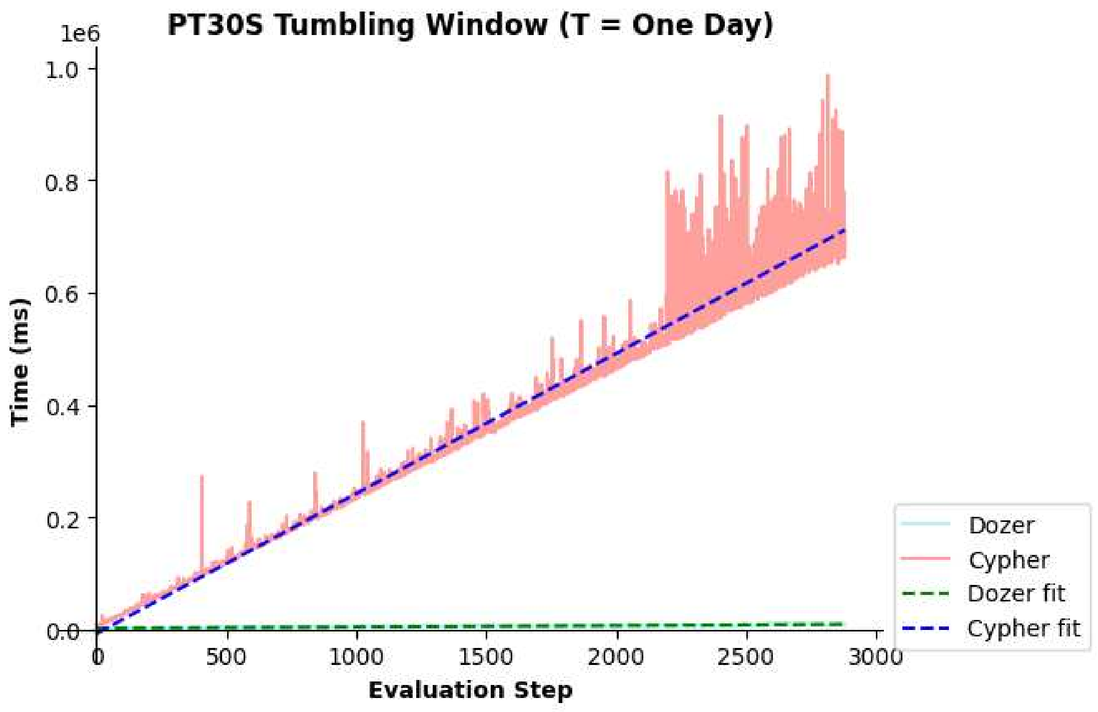


Figure 6.7: Big-O Analysis of Dozer vs Cypher Time Complexity at different window sizes (10-seconds, 15-seconds) over a one-day long time horizon



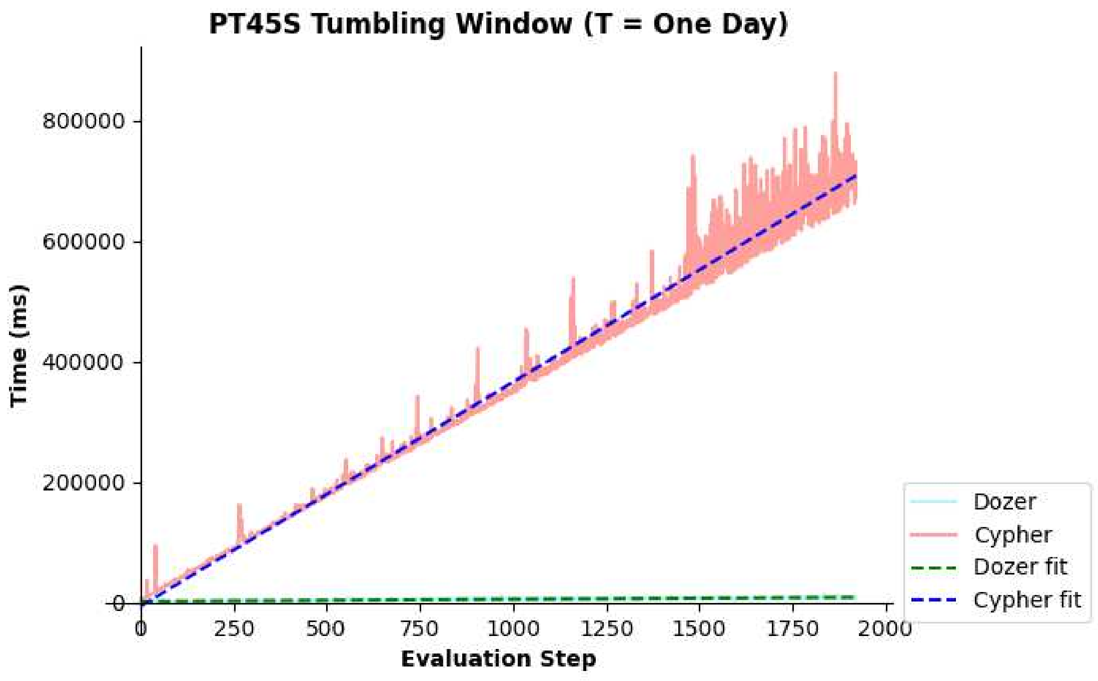


Figure 6.8: Big-O Analysis of Dozer vs Cypher Time Complexity at different window sizes (30-seconds, 45-seconds) over a one-day long time horizon

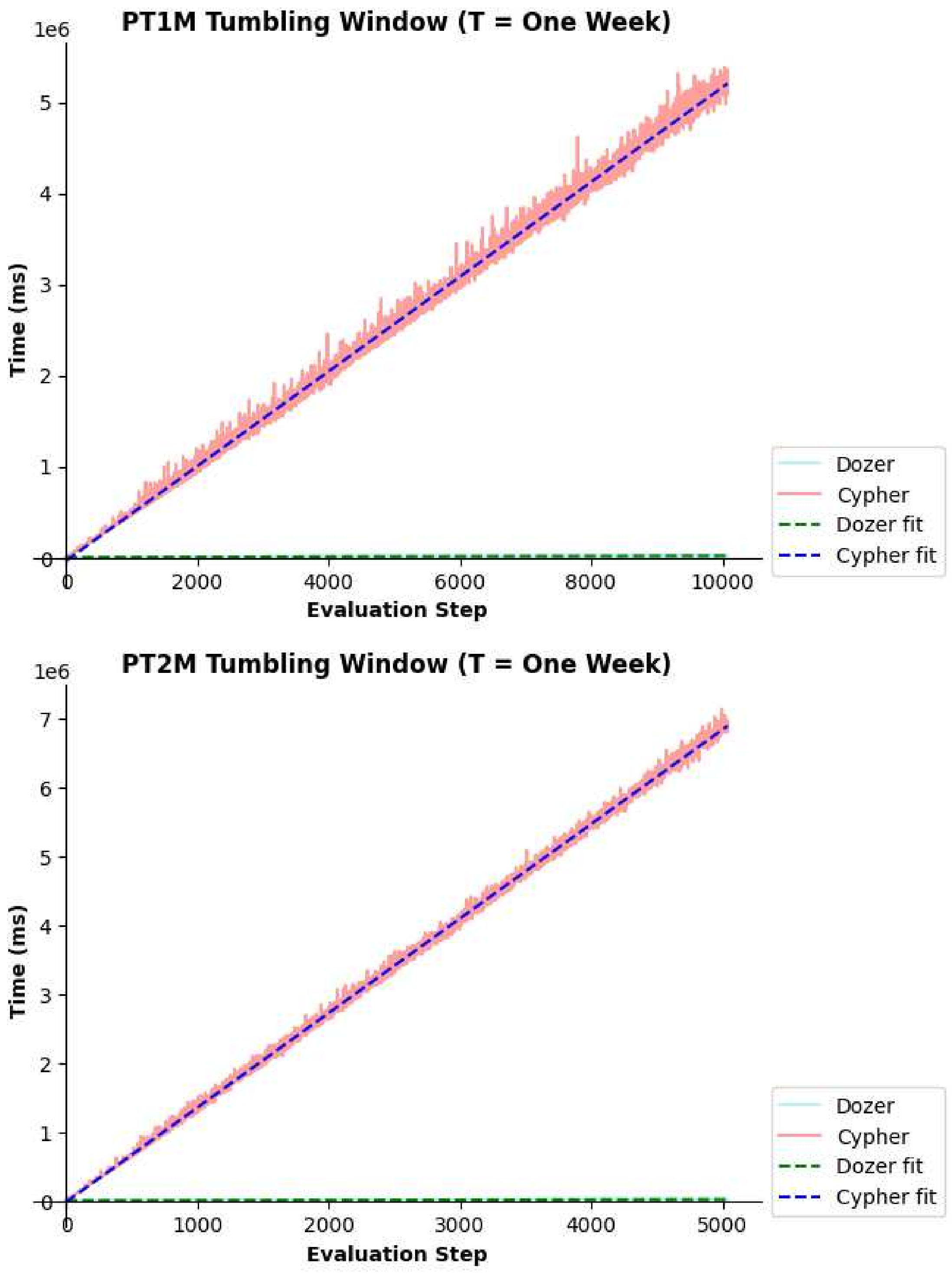
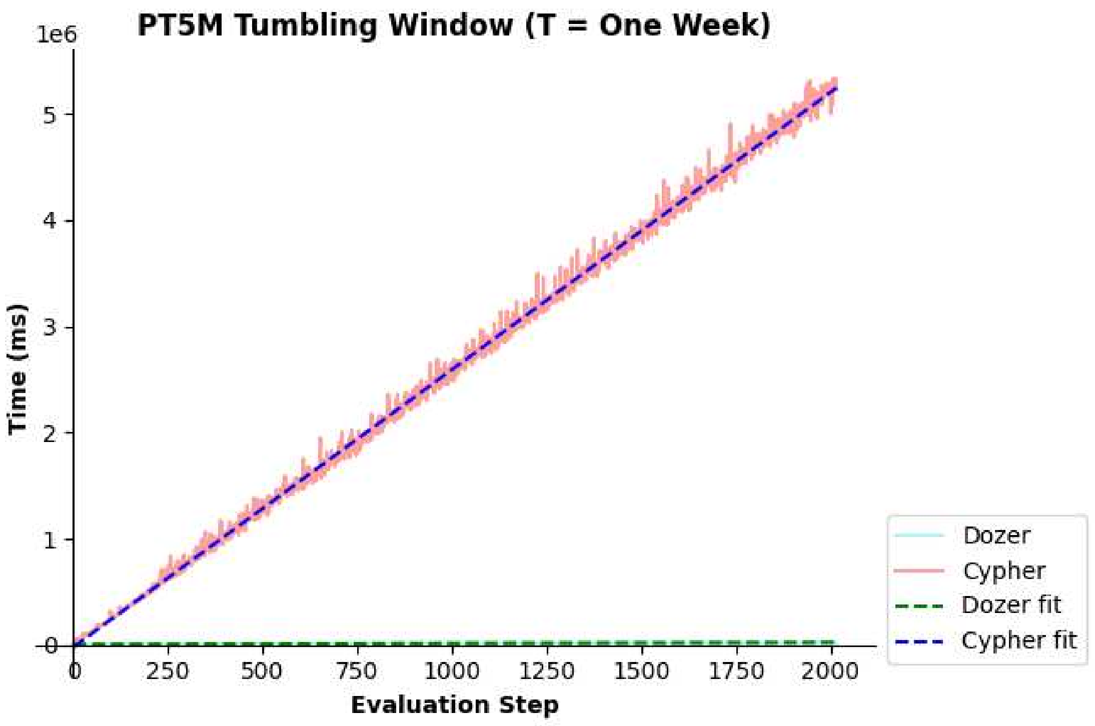


Figure 6.9: Big-O Analysis of Dozer vs Cypher Time Complexity at different window sizes (1-minute, 2-minutes) over a seven-day long time horizon



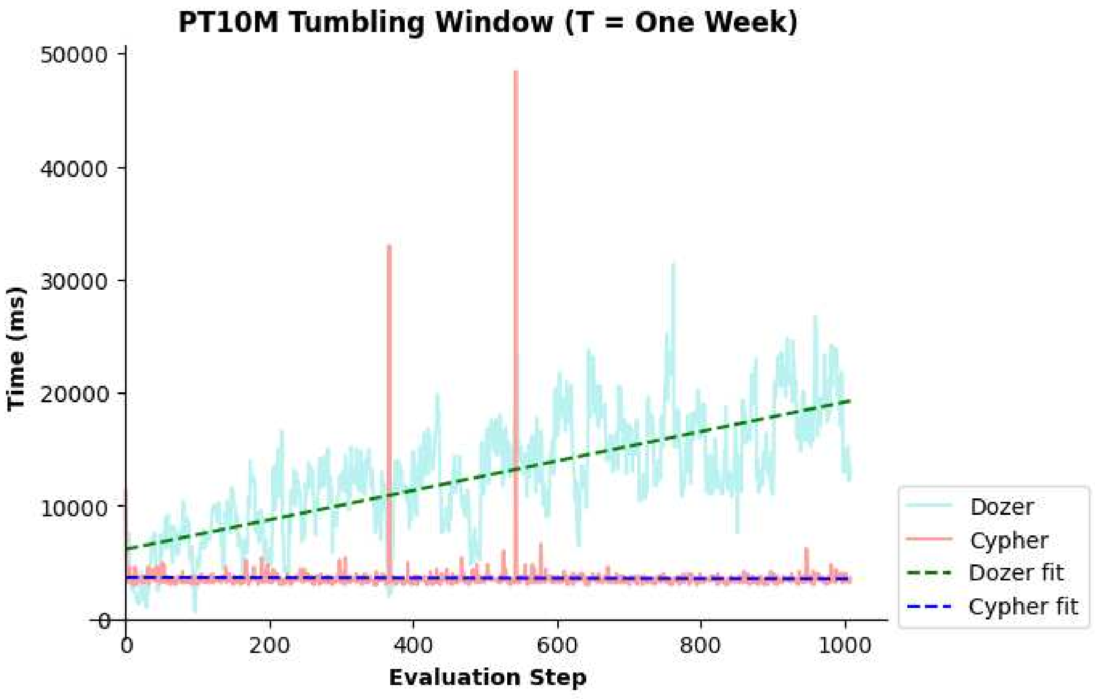


Figure 6.10: Big-O Analysis of Dozer vs Cypher Time Complexity at different window sizes (5-minutes, 10-minutes) over a seven-day long time horizon

6.2.2 Perfomance Comparison

In this section, we will further investigate the Dozer performance. Firstly, going in-depth in the difference between Dozer and querying timestamped Property Graphs with Cypher. Starting from the analysis in the previous section, we will define a cost function that allows us to compare the two approaches. Then, we will analyze, for each window, the total execution time needed by the two systems to perform the same query over a predefined time horizon. Finally, using the same metrics used for comparing Dozer with Cypher, we will discuss how Dozer reacts to a different dataset.

6.2.2.1 Dozer vs Querying Timestamped Graphs with Cypher

Starting from the time complexity analysis described in Section 6.2.1, and using the data collected for the latter, we define a cost function as a metric to evaluate and compare the two approaches, namely Dozer and Cypher.

Cost function definition. Let consider k different executions. Let Γ be a finite, discrete, ordered sequence of $N \in \mathbb{N}$ time instants $(t_1, t_2, ..., t_N)$, with $t_i \in \mathbb{N}$; let α be the width of a time-based tumbling window; and let $\delta_j = [\delta_{j1}, \delta_{j2}, ..., \delta_{jN}]$ be the vector of the $j-th$ execution (with $j \in 1, 2, ...k$), where each element δ_{ji} (for $i \in 1, 2, ..., N$) corresponds to the time it took for the Neo4j server to obtain the *MATCH* query results at the $i - th$ execution step. We define the cost function of the $j - th$ execution:

$$c_j(\alpha) = [\delta_{j1}, \delta_{j1} + \delta_{j2}, ..., \delta_{j1} + \delta_{j2} + ... + \delta_{jN}]$$

as the *cumulative sum* of the vector δ_j for an α-width time-based tumbling window.

Then, we define the *TOTAL COST FUNCTION* for an α-width time-based tumbling window:

$$\overline{C}(\alpha) = \frac{1}{k} \sum_{j=1}^{k} c_j(\alpha)$$

as the mean of the k costs $c_1, c_2, ..., c_k$.

According to the analysis made in Section 6.2.1, if we consider, for a given α, the complexity of the $\overline{C}(\alpha)$ as a function of the number of relationships $\mathbf{n}$, we expect:

- A linear cost function $\overline{\mathbf{C}}(\alpha) \sim \mathbf{O}(\mathbf{n})$ for Dozer; and
- A quadratic cost function $\overline{\mathbf{C}}(\alpha) \sim \mathbf{O}(\mathbf{n^2})$ for Cypher.

The plots in Figures 6.11 and 6.12 depict the cost function of Dozer versus Cypher calculated for different window sizes over a seven-day long time horizon; while in Figures 6.13 and 6.14 the cost function for other window sizes over a a one-day long time horizon. Each cost function has been averaged over five independent executions, as indicated in Section 6.1. The cost function depends on the timing collected for the big-o analysis. As for the previous section, it increases for wider windows, while decreasing in Cypher.

Furthermore, we focused our interest on measuring the time impact experienced by the end-user in the two different approaches. For this purpose, we compare the total execution time needed by Dozer and Cypher to run the same query. As for the previous settings, the bar-plots in Figure 6.15 and 6.16 have been computed by averaging the total execution time of five independent execution over a one-day and seven-day long time horizon, respectively. We can notice that, in both scenarios, the total execution time decreases as window ranges increase, following the trend reported in the previous assessments. Moreover, Dozer outperforms Cypher, regardless of the window size. Of course, continuing to increase it, the result would change, in line with what has already been discussed.

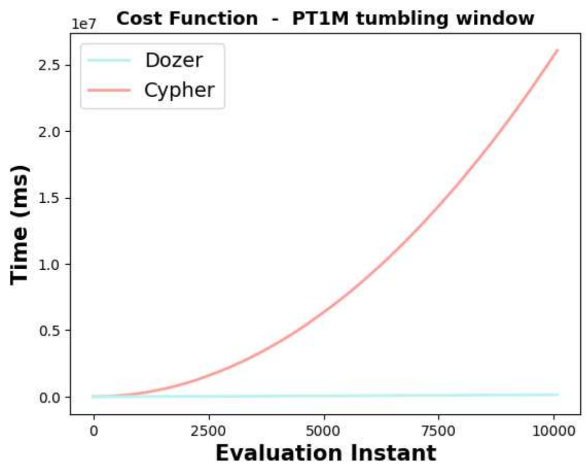


Figure 6.11: Cypher's vs Dozer's Cost function over a seven-day long time horizon for a 1-minute tumbling window

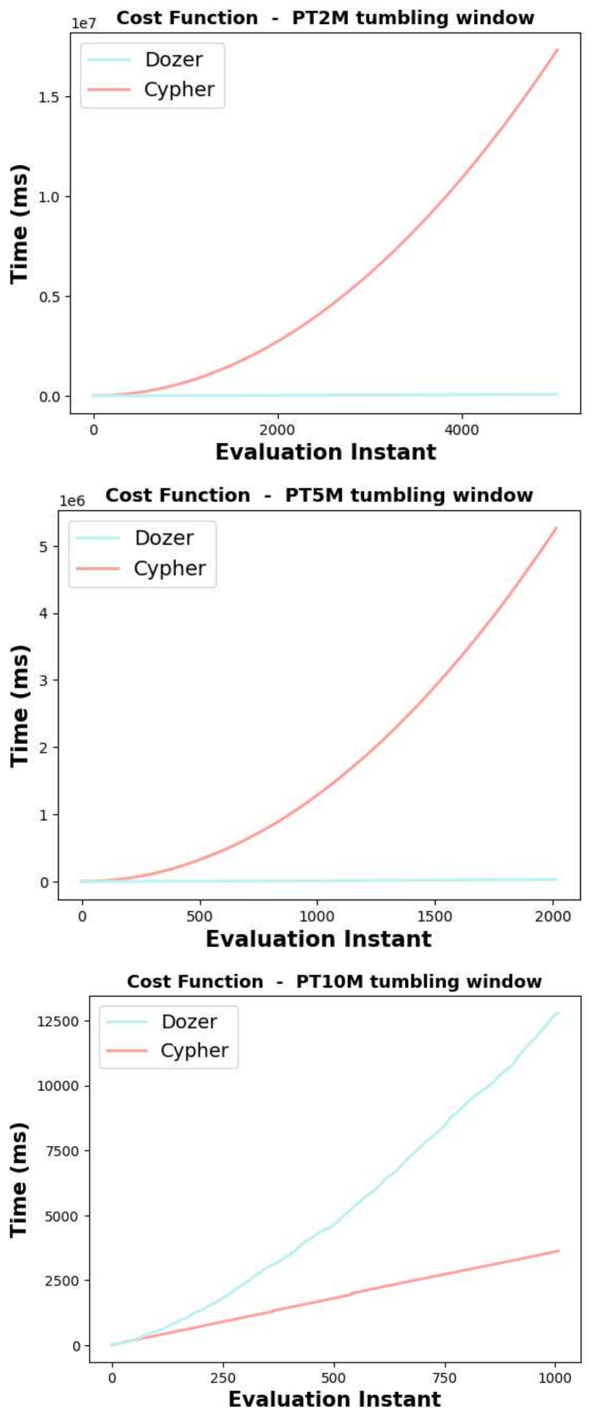


Figure 6.12: Cypher's vs Dozer's Cost function at different window sizes (2-minutes, 5-minutes, 10-minutes) over a seven-day long time horizon

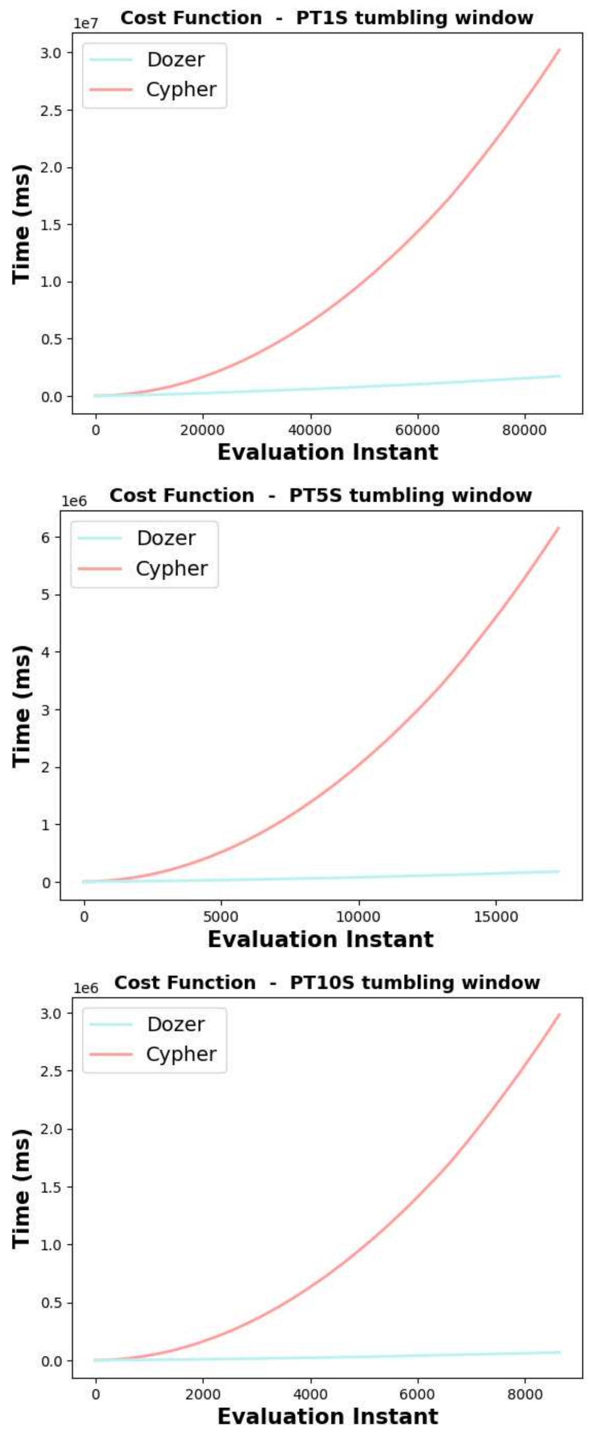


Figure 6.13: Cypher's vs Dozer's Cost function at different window sizes (1-second, 5-seconds, 10-seconds) over a one-day long time horizon

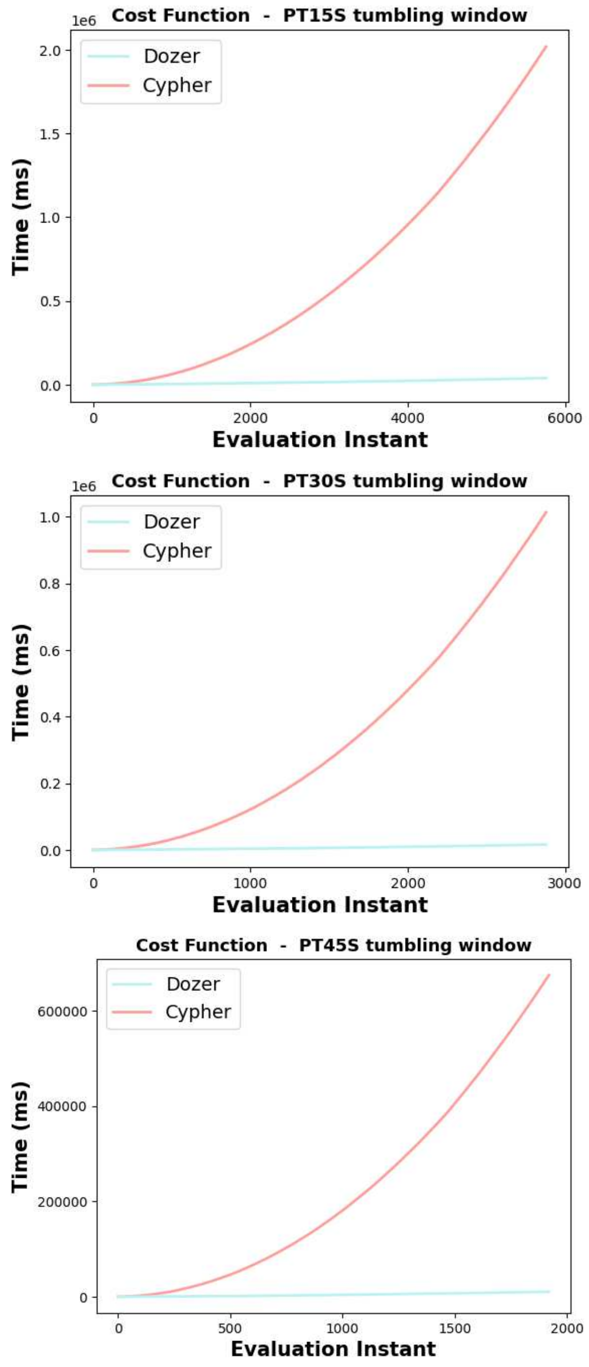


Figure 6.14: Cypher's vs Dozer's Cost function at different window sizes (15-second, 30-seconds, 45-seconds) over a one-day long time horizon

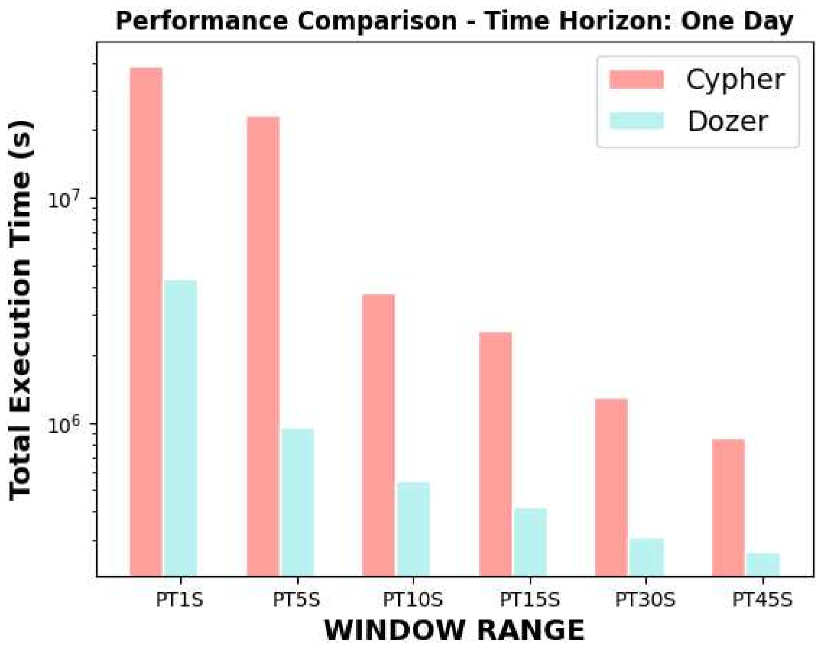


Figure 6.15: Total execution time comparison over a one-day long time horizon

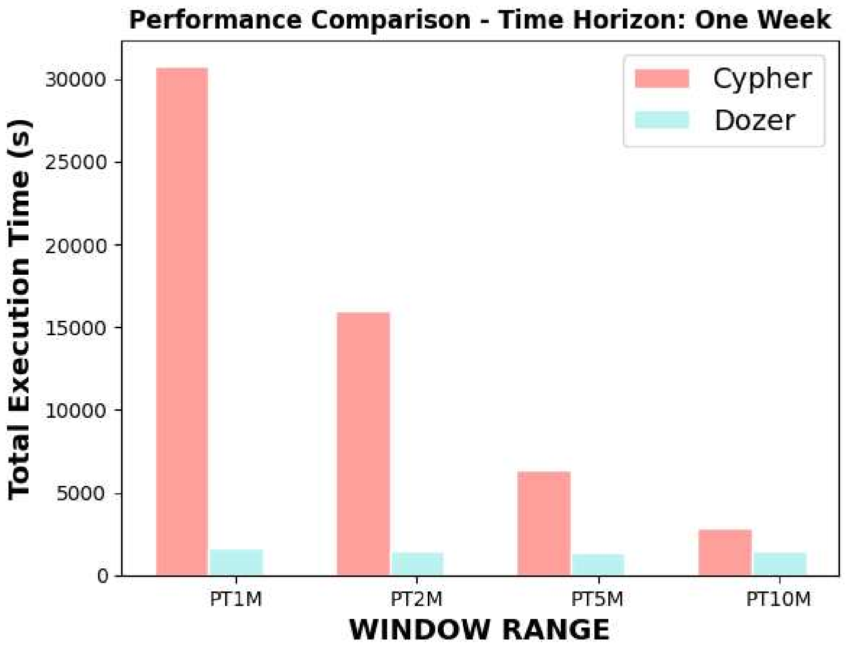


Figure 6.16: Total execution time comparison over a seven-day long time horizon

6.2.2.2 Dozer Performance At Different Datasets

In the light of the outcome discussed in the previous sections, we repeated some tests on a different dataset (see Section 6.1.2.2) to understand if the results depend on whether or not the linear shape of the dataset used to carry out the experiments. Firstly, we ran five experiments with the query in Figure 6.3 on the two different datasets over a one-day long time horizon. We collect the time it took for the Neo4j server for the results at different window sizes, and we plot the cost functions in the two scenarios. Figure 6.17 show the result related to the executions with a one-second, five-second, one-minute and five-minute tumbling window, on which some important considerations:

1. In the star-shaped case, we have greater production frequency (each second, we create ten times the number of nodes and relationships as the linear one) and a more complex structure. Consequently, the star-shaped cost function width will be greater, but it will have the same trend as the linear one. For middle windows, the difference in width is more evident. Indeed, with smaller windows, the behavior is almost similar because, at each evaluation step, Dozer handles a small number of relationships in both scenarios; while continuing to increase the window range, the result would change (see item 3);

2. Analyzing Dozer with different datasets, we can figure out how the cost function has a **sub-quadratic complexity** due to the node accumulation discussed in the previous phases;

3. As discussed in item 1, with the star-shaped dataset, we handle more triples. The last plot, referred to as a five-minutes wide tumbling window, shows how Neo4j becomes more efficient with a bulk load (see discussion in Section 6.2.1).

Furthermore, also for these experiments, we measured the time impact experienced by the end-user, by computing the total execution time to run the same query in the two different datasets. As for the previous settings, the bar-plot in Figure 6.18 has been computed by averaging the total execution time of five independent executions over a one-day long time horizon. We can notice that:

- In both scenarios, the total execution time decreases as window ranges increase, following the trend reported in the previous assessments; and

- We have greater execution times in the star-shaped dataset because it is larger. However, the two cases have almost the same order of magnitude.

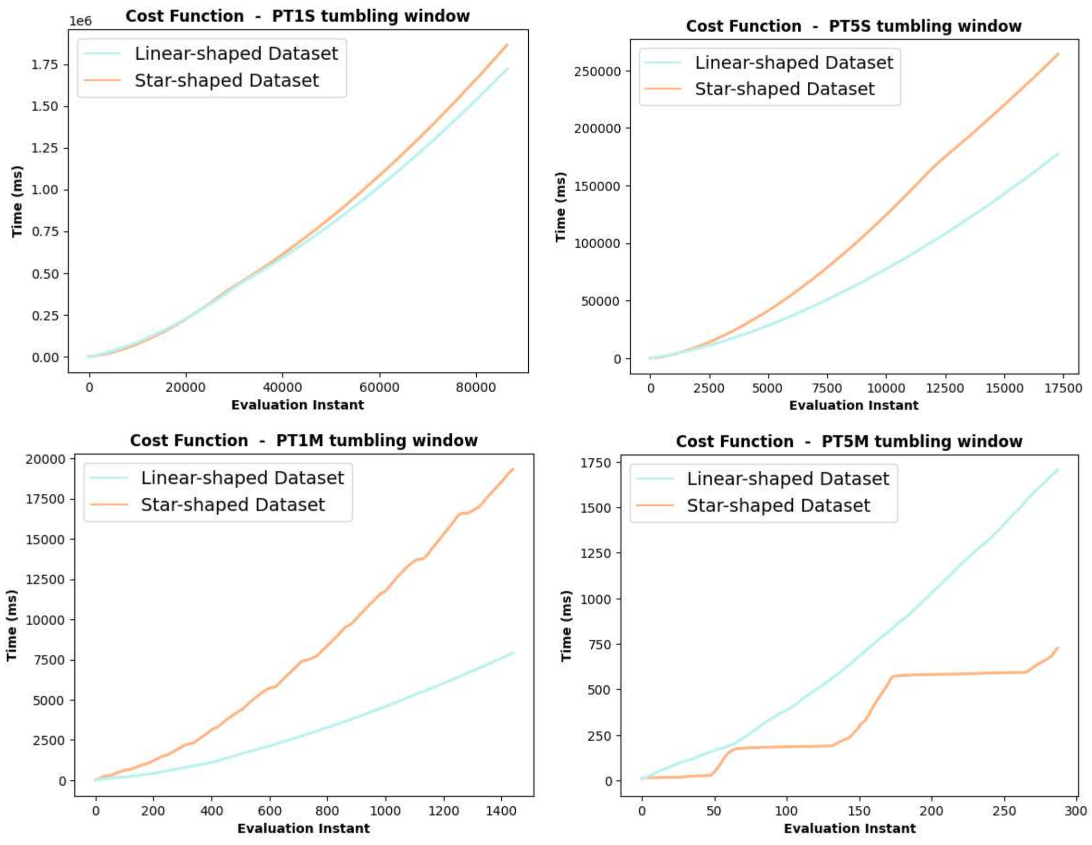


Figure 6.17: Dozer's cost function with a linear-shape dataset against a star-shape dataset at different window sizes (1-second, 5-seconds, 1-minute, 5-minutes) over a one-day long time horizon

6.2.3 Components Overhead Analysis

This section is devoted to the overhead analysis of each component in Dozer w.r.t. window changes. During all of the previous tests, we measured the portion of time spent by each component. Figures 6.19 and 6.20 show, for each of them, the overhead percentage w.r.t. the total execution time, for a one-day and a seven-day long time horizon, respectively. As usual, all computations have been averaged across five independent executions. The bar-plots highlight the following patterns:

1. The **SyncGenerator**'s objective is to update the *time-to-sync*. Therefore, its timing is relatively smaller than the other components. Moreover, with window size increasing, the number of evaluation instants decreases, as well as the time spent by the *SyncGenerator*.

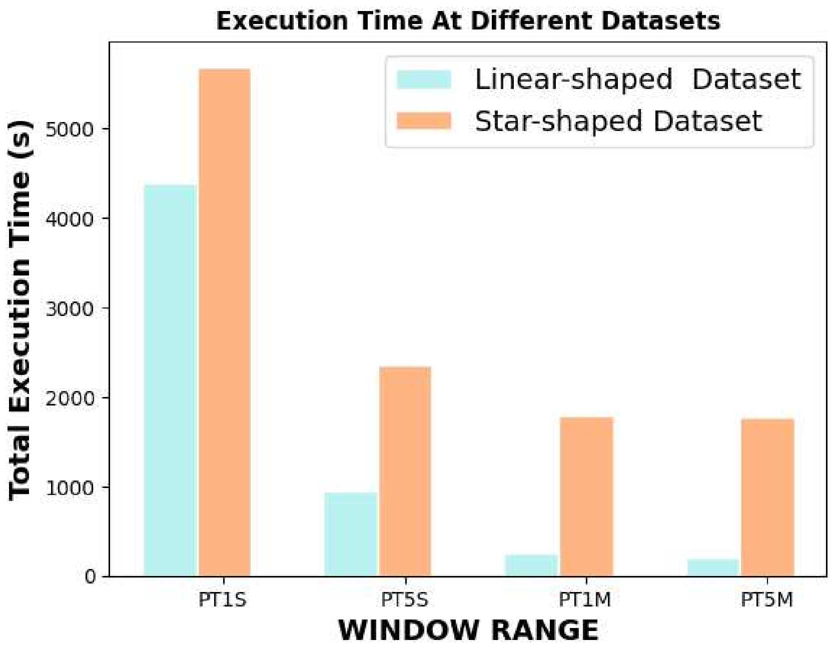


Figure 6.18: Dozer's total execution time at different dataset over a one-day long time horizon

2. Considering a specific window width, the percentage of the **TimeManaged-Deletion** and the **TimeManagedInsertion** components is more or less the same.

3. As window sizes increase, the **CypherHandler**'s timing decreases at the expense of increasing the time spent in the maintenance phases (both deletion and insertion). The reason is twofold:

   (a) Increasing the window size, the *TimeManagedDeletion* and the *TimeManagedInsertion* components handle, at each evaluation instant, a larger number of nodes and relationships, requiring higher maintenance costs.

   (b) Neo4j efficiently manages bulk loads. Consistently with the analysis addressed in the previous sections, the *CypherHandler*'s timing reduces as window width increases.

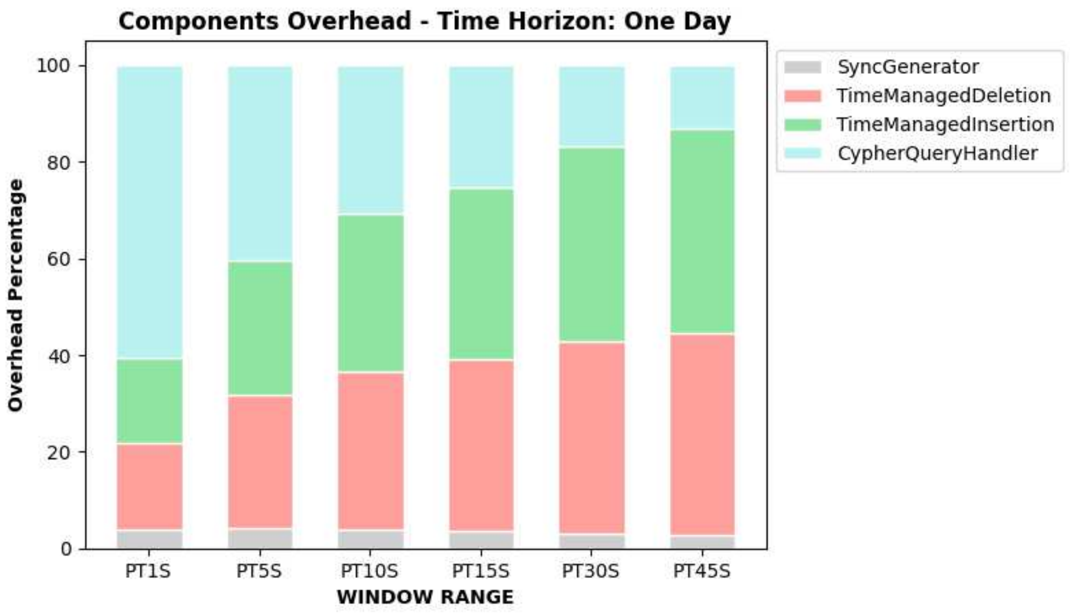


Figure 6.19: Dozer components' overhead at different window sizes (1-second, 5-seconds, 10-seconds, 15-seconds, 30-seconds, 45-seconds) over a one-day long time horizon

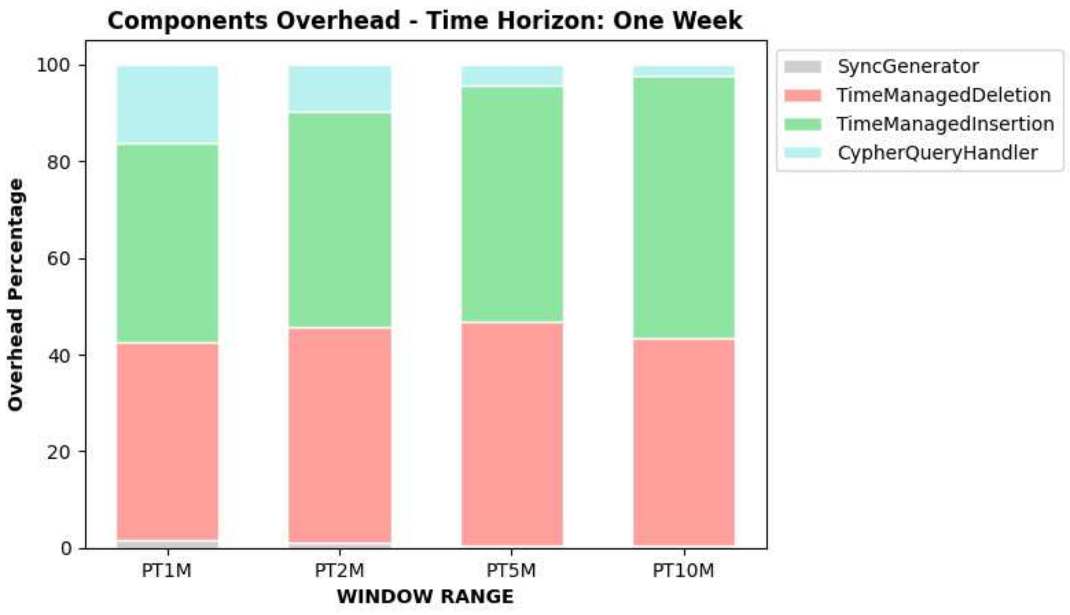


Figure 6.20: Dozer components' overhead at different window sizes (1-minute, 2-minutes, 5-minutes, 10-minutes) over a seven-day long time horizon

6.3 Fault-tolerance Tests

All the tests presented in the previous sections highlighted the capacity of Dozer to continuously work without any failure. Therefore, we performed a set of tests aimed at evaluating the system failover by inducing some anomalies. We ran, for each window, ten executions with and without failures, and both over a two-hour-long time horizon. For the latter, we simulate some anomalies by forcing the system to restart at regular intervals of ten minutes, having thus 11 failures within each execution.

Under the assumption that every single recovery within an execution is independent of and does not influence the timing of the other recoveries, we wanted to evaluate how the window width affects the recovering time. Let $\mathcal{T}$ be a finite, discrete, ordered time domain; let $\mathcal{N}$ be the number of recoveries occurring in the time horizon $\mathcal{T}$; and let r_{ij} be the collected time referring to the j-th recovery in the i-th execution. We define the *Mean Time To Recover* for the i-th execution:

$$MTTR_i = \frac{1}{N}\sum_{j=1}^{N} r_{ij}$$

as the average time that Dozer took to recover from any failure in the i-th execution. Then, we analyze the recovery distribution w.r.t. the window width by plotting for each window the ten computed MTTRs. The boxplots in Figure 6.21 show the result of our tests:

- Dozer's MTTR ranges from 10 to 12 seconds; and
- The MTTR increases as window width decreases.

In addition to measuring the time Dozer took for a single recovery, we studied how failures affect end-user usability. The barplot in Figure 6.22 depicts the total execution time increment due to the 11 recoveries. Differently from what Figure 6.21 highlights, the total execution time increases as window size increases.

The boxplots depict the average time required by Dozer to recover and resume running. However, the engine takes some time before running at full capacity. Wider the windows, closer the evaluation instants at which the system fails and has to recover. Let's consider the two corner cases of the Figure 6.21. With a 1-second-wide tumbling window, we have a recovery every 600 evaluation steps, while with a 5-minutes-wide tumbling window, we need to recover every two evaluation instants. Therefore, increasing the window size, a failure happens before the system start working at full capacity, and then the total execution time increases.

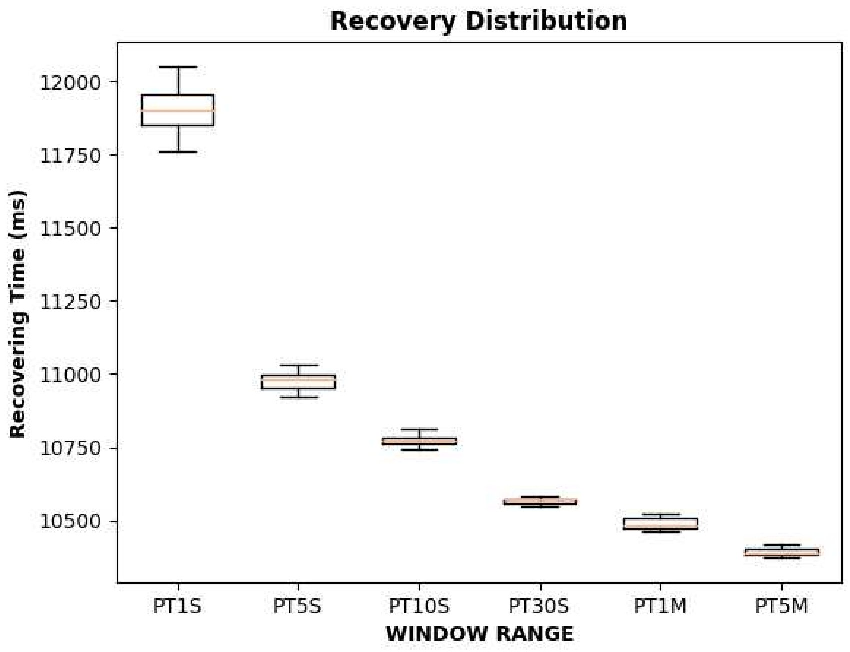


Figure 6.21: Recovery distribution for different window ranges

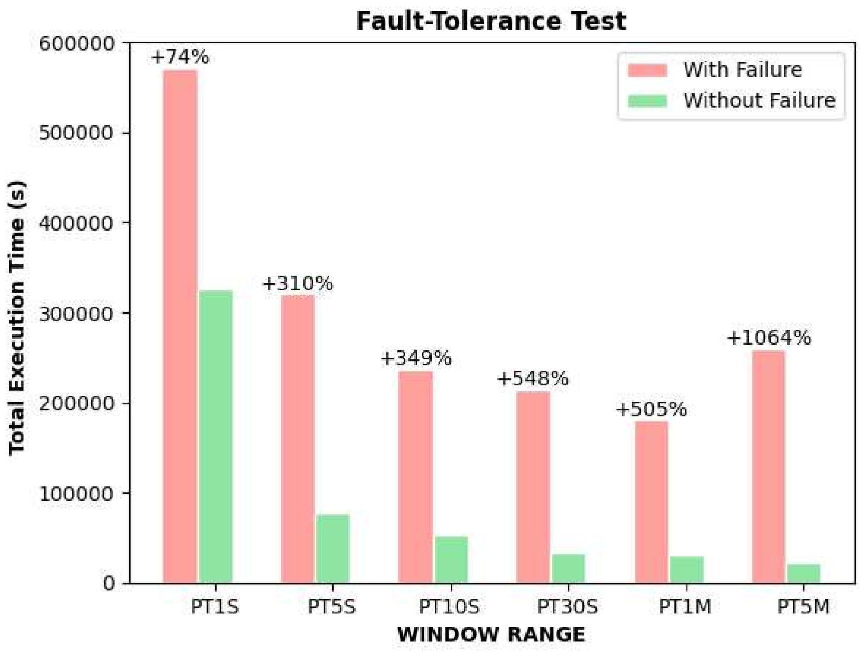


Figure 6.22: Failures' impact on the total execution time

Chapter 7

Conclusions and Future Work

This thesis aimed at providing a reference architecture for the implementation of streaming applications for the Seraph queries evaluation. The suggested architecture keeps the data moving and achieves low latency by incorporating built-in event/data-driven processing capabilities. Moreover, the use of a Streaming Processing Engine at the basis of the architecture allows us to exploit built-in mechanisms to deal with streams' challenges, such as delayed, missing, and out-of-order data, and to guarantee high availability and fault tolerance.

For the definition of the requirements the architecture should meet, we based on the high-level guidance for developing a real-time SPE (Stream Processing Engine), outlining eight requirements that the system should meet [46].

The use of a Streaming Processing Engine (e.g. Spark, Kafka Streams, Apache Flink) at the basis of our architecture allows us to exploit built-in mechanisms to satisfy the requirements. In particular, we presented Dozer[1], the first implementation of the proposed architecture, to be a valid alternative to a prototype not suitable for industrial development. Of course, the requirement of *"Query using SQL on Streams (StreamSQL)"* has been the first one we considered. In general, the streaming applications should provide some querying mechanism to find output events of interest or compute real-time analytics. In our case, we define a reference architecture for the implementation of streaming applications for the Seraph queries evaluation, which is an extension of Cypher, intending to introduce streaming features in the context of property graph query languages. With Dozer, we focused on the basic features while ignoring certain more complex operations, such as the management of the optional clause inspired by *Morpheus*[2] for the creation of a Property Graph Stream. Moreover, during the design phase, we adopted some simplifications such as managing only the basic types and imple-

[1]https://github.com/openseraph/SeraphEngine
[2]https://github.com/opencypher/morpheus

menting only the *RStream* operator. The use of the other two operators (*IStream* and *DStream*), as well as the use of more complex data structures allowed in Cypher (*Map*, *List*, and *Path*) and the use of more complex queries (e.g. enabling *UNION* or *JOIN* between streams), do not impact the Dozer's design but requires coding implementation and the corresponding tests.

The fact the Seraph directly supports consumption (output) from (to) Kafka, combined with the availability of the pre-built Neo4j Kafka Connector[1], able to ingest and sink graph data into Kafka, has played a key role in choosing Kafka and Kafka Streams as the core of the Dozer architecture. In particular, during Dozer's design, scalability and fault-tolerance were the major driving factors. Among the eight requirements, Stonebraker et al. [46] require the system to guarantee the following:

- *"Partition and Scale Applications Automatically".* The system should provide incremental scalability, by distributing the computation across multiple processors and machines. Ideally, the distribution should be automatic and transparent.
- *"Guarantee Data Safety and Availability".* The applications should be always up and available, and the data integrity maintained anyway, despite failures.
- *"Generate Predictable Outcomes".* An SPE must guarantee predictable and repeatable outcomes. This is also important from fault-tolerance and recovery perspective.

The Kafka distributed and parallel processing model allows us to design a microservice architecture and to build a distributed continuous query processing that provides dynamic scalability. However, this paradigm typically requires container management systems (e.g. Kubernetes, Docker Swarm), which, at the moment, have not been integrated. An orchestrator may facilitate the load distribution between and within Kafka clusters, improving the first requirement.

Moreover, with Kafka, data can be fault-tolerant and highly available, by replicating every topic. Dedicated fault-tolerance tests have been executed to study how Dozer reacts to a failure and how the latter affects the end-user usability, while no tests about availability and data integrity have been carried out. For the latter, Kafka enables the customization of the number of replicas according to the design needs, as well as the configuration of the acknowledgment level (*acks=0*, *acks=1*, *acks=all*). This allows to trade-off between durability guarantees and performance. Assuming to have no frequent failures with Dozer, we focused on improving performances without worrying about availability and replicas.

[1]https://neo4j.com/labs/kafka/

Finally, the several tests, for both performance and fault-tolerance purposes, highlighted the coherence in the results and Dozer's ability to generate repeatable outcomes. On the other hand, correctness verification requires a more complex analysis with the design and implementation of an *oracle*, helping to automatically check the correctness [16].

However, the use of a *pull-based* architecture like Kafka allowed us to gain in scalability and fault tolerance, at the expense of higher latency. In addition, the design of a *tuple-driven* architecture introduced further latency. Even though we were aware of these limits, the *Neo4j Kafka Connector* has played a key role. Firstly, it allows us to focus on the implementation of the core of the architecture without worrying about the communication between our engine and Neo4j. Moreover, it can ingest and sink graph data with the *CDC design pattern*, enabling to update of the Neo4j with only the data that has changed from the source system. The CDC pattern has emerged as an ideal solution to design event-driven architectures that provide real-time data by moving and processing data continuously as new database events occur because it allows streaming every single event occurring on a database into Kafka at very low latency and low impact. Using the Kafka Sink Connector, combined with the CDC module, allows us to overcome the limits imposed by the batch nature of Neo4j, as well as to improve the latencies due to a pull-based system.

Moreover, with Dozer, we focused on modeling the dynamic evolution of graphs over the relationships, which are the key elements of graph DBs. This assumption allows us to cut maintenance costs by working on the insertion and deletion of the relationships over time without worrying about nodes. The different tests highlighted the performance of Dozer, outperforming the traditional way of querying timestamped Property Graphs with Cypher. However, the nodes accumulation affects the performance, which reduces as window width increases. Furthermore, in some scenarios, the variation of nodes over time is crucial and must be correctly managed.

On the basis of the foregoing considerations, other two key requirements have been satisfied by Dozer and experimentally evaluated.

- *"Keep the data moving"*. To achieve low latency a system must be able to perform message processing without having a costly storage operation in the critical processing path.
- *"Process and Respond Instantaneously"*. A stream processing system must have a highly-optimized, minimal-overhead execution engine to deliver real-time response for high-volume applications.

Of course, further improvement for better performance, both in terms of latency reduction and throughput increment, could be achieved. Here is a list of suggestions sorted by their impact w.r.t the architecture re-design:

1. Maintain Kafka and run performance tuning, to optimize the system latency and throughput.

2. Define a *Time-Driven Tick* [10], reducing the latency for the query computation. On the other hand, the *record-at-a-time* model requires state maintenance for all operators with record-level granularity. This behavior obstructs system throughput and brings much higher latency when recovering after a system failure.

3. Change the core of the architecture, choosing a *push-based* model (e.g. with *Apache Flink*[1]) to reduce the latency, or a *micro-batch* model (e.g. *Apache Spark*[2]) to improve throughput even with wider windows.

Another critical aspect in stream processing is the notion of time, and how it is modeled and integrated. Among the requirements, the architecture must *"Handle Stream Imperfections (Delayed, Missing and Out-of-Order Data)"*. In Dozer, we worked with Kafka *ingestion-time*, and for this reason, we do not have stream imperfections by design. In the case of moving towards an *event-time-based* model, the stream imperfections should be handled. Maintaining Kafka, we could achieve the requirement by using the *ProcesorAPI* from Kafka Streams, by implementing some kind of scheduler with the *Punctuator*[3]. Alternatively, in the case we change the core of the architecture, we could adopt Apache Spark with *watermarking* [7]. Or Apache Flink that enables the Out-of-Order events processing accurately.

Finally, according to Stonebraker et al. [46], the system must *"Integrate Stored and Streaming Data"*. At the moment, this is part of the future work of the Seraph language. The latter, and consequently the designed architecture, can be extended to accommodate more use-cases. Once Seraph is extended with *Static graph* support, the engine should efficiently store, access, modify historical information, and combine it with live streaming data. Moreover, with *Multi-Stream* and *Multi-window* supports, developers should be able to perform queries across streams and with multiple sliding windows, respectively.

Another interesting future work could be an implementation of a totally distributed version of Dozer, in which every single node deals only with a single evaluation instant, leaving to another node the management of the following evaluation instant. For this purpose, we suggest a Chord-based [45] architecture.

[1]https://flink.apache.org/
[2]https://spark.apache.org/
[3]https://kafka.apache.org/10/documentation/streams/developer-guide/processor-api.html

Bibliography

[1] Models and issues in data stream systems, author=Babcock, Brian and Babu, Shivnath and Datar, Mayur and Motwani, Rajeev and Widom, Jennifer. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, 2002.

[2] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1421–1432, 2018.

[3] Darko Anicic, Paul Fodor, Sebastian Rudolph, and Nenad Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644, 2011.

[4] Itamar Ankorion. Change Data Capture efficient ETL for real-time BI. *Information Management*, 15(1):36, 2005.

[5] Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Utkarsh Srivastava, and Jennifer Widom. Stream: The stanford data stream management system. In *Data Stream Management*, pages 317–336. Springer, 2016.

[6] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal*, 15(2):121–142, 2006.

[7] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for Real-Time applications in Apache Spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613, 2018.

[8] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL: SPARQL for continuous querying. In *Proceedings of the 18th international conference on World wide web*, pages 1061–1062, 2009.

[9] Jagdev Bhogal and Imran Choksi. Handling Big Data Using NoSQL. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications Workshops*, pages 393–398, 2015.

[10] Irina Botan, Roozbeh Derakhshan, Nihal Dindar, Laura Haas, Renée J Miller, and Nesime Tatbul. SECRET: a model for analysis of the execution semantics of stream processing systems. *Proceedings of the VLDB Endowment*, 3(1-2):232–243, 2010.

[11] Pieter Cailliau, Tim Davis, Vijay Gadepally, Jeremy Kepner, Roi Lipman, Jeffrey Lovitz, and Keren Ouaknine. Redisgraph graphblas enabled graph database. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 285–286. IEEE, 2019.

[12] Jean-Paul Calbimonte, Hoyoung Jeung, Oscar Corcho, and Karl Aberer. Enabling query technologies for the semantic sensor web. *International Journal On Semantic Web and Information Systems (IJSWIS)*, 8(1):43–63, 2012.

[13] Paris Carbone, Asterios Katsifodimos, Stephan Ewen, Volker Markl, Seif Haridi, and Kostas Tzoumas. Apache Flink: Stream and batch processing in a single engine. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[14] Hirokazu Chiba, Ryota Yamanaka, and Shota Matsumoto. Property Graph Exchange Format. *ArXiv*, abs/1907.03936, 2019.

[15] codecademy. What is CRUD? Create, Read, Update, and Delete (CRUD) are the four basic functions that models should be able to do, at most. `https://www.codecademy.com/articles/what-is-crud`.

[16] Daniele Dell'Aglio, Jean-Paul Calbimonte, Marco Balduini, Oscar Corcho, and Emanuele Della Valle. On correctness in rdf stream processor benchmarking. In Harith Alani, Lalana Kagal, Achille Fokoue, Paul Groth, Chris Biemann, Josiane Xavier Parreira, Lora Aroyo, Natasha Noy, Chris Welty, and Krzysztof Janowicz, editors, *The Semantic Web – ISWC 2013*, pages 326–342, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[17] Daniele Dell'Aglio, Emanuele Della Valle, Jean-Paul Calbimonte, and Oscar Corcho. RSP-QL semantics: A unifying query model to explain heterogeneity

of RDF stream processing systems. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 10(4):17–44, 2014.

[18] Daniele Dell'Aglio, Emanuele Della Valle, Frank van Harmelen, and Abraham Bernstein. Stream reasoning: A survey and outlook. *Data Science*, 1(1-2):59–83, 2017.

[19] A. Deutsch. Querying Graph Databases with the GSQL Query Language. In *SBBD*, 2018.

[20] Rydberg Mats et al. Morpheus. 2016. `https://github.com/opencypher/morpheus`.

[21] Eric Evans. NOSQL 2009. May 2009. – Blog post of 2009-05-12. `https://web.archive.org/web/20110716174012/http://blog.sym-link.com/2009/05/12/nosql_2009.html`.

[22] Emanuele Falzone, Riccardo Tommasini, Emanuele Della Valle, Petra Selmer, Stefan Plantikow, Hannes Voigt, Keith Hare, Ljubica Lazarevic, and Tobias Lindaaker. Semantic Foundations of Seraph Continuous Graph Query Language. `https://arxiv.org/abs/2111.09228`, 2021.

[23] Avrilia Floratou. *Columnar Storage Formats*, pages 464–469. Springer International Publishing, Cham, 2019.

[24] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 1433–1445, New York, NY, USA, 2018. Association for Computing Machinery.

[25] R. Giugno and D. Shasha. GraphGrep: A fast and universal method for querying graphs. In *2002 International Conference on Pattern Recognition*, volume 2, pages 112–115 vol.2, 2002.

[26] Alastair Green, Martin Junghanns, Max Kießling, Tobias Lindaaker, Stefan Plantikow, and Petra Selmer. openCypher: New Directions in Property Graph Querying. In *EDBT*, pages 520–523, 2018.

[27] Peter Haase, Jeen Broekstra, Andreas Eberhart, and Raphael Volz. A Comparison of RDF Query Languages. In *SEMWEB*, 2004.

[28] I Herman. Eleven SPARQL 1.1 specifications are W3C recommendations. *w3.org*, 2013.

[29] Ivan Herman. SPARQL is a Recommendation. *W3C Semantic Web Activity News*, 2008.

[30] Martin Junghanns, Max Kießling, Alex Averbuch, André Petermann, and Erhard Rahm. Cypher-based graph pattern matching in Gradoop. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–8, 2017.

[31] Martin Junghanns, André Petermann, Niklas Teichmann, Kevin Gómez, and E. Rahm. Analyzing extended property graphs with Apache Flink. *Proceedings of the 1st ACM SIGMOD Workshop on Network Data Analytics*, 2016.

[32] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. In *Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68, 2012.

[33] Doug Laney et al. 3D data management: Controlling data volume, velocity and variety. *META group research note*, 6(70):1, 2001.

[34] Danh Le-Phuoc, Minh Dao-Tran, Josiane Xavier Parreira, and Manfred Hauswirth. A native and adaptive approach for unified processing of linked streams and linked data. In *International Semantic Web Conference*, pages 370–388. Springer, 2011.

[35] József Marton, Gábor Szárnyas, and Márton Búr. Model-driven engineering of an opencypher engine: Using graph queries to compile graph queries. In *International SDL Forum*, pages 80–98. Springer, 2017.

[36] Andreas Meier and Michael Kaufmann. *SQL & NoSQL Databases: Models, Languages, Consistency Options and Architectures for Big Data Management.* 01 2019.

[37] M. Paradies. Graph pattern matching in SAP HANA. First openCypherImplementers Meeting, Feb. 2017. `https://tinyurl.com/ycxu54prl`.

[38] Stefan Plantikow. Towards an International Standard for the GQL Graph Query Language. *W3C workshop in Berlin on graph data management standards*, 2019.

[39] D Robins. Complex Event Processing. In *Second International Workshop on Education Technology and Computer Science. Wuhan*, pages 1–10. Citeseer, 2010.

[40] Sherif Sakr and Albert Y. Zomaya, editors. *Document Store*, pages 691–691. Springer International Publishing, Cham, 2019.

[41] Sherif Sakr and Albert Y. Zomaya, editors. *Key-Value Store*, pages 1067–1067. Springer International Publishing, Cham, 2019.

[42] Jiwon Seo, Stephen Guo, and Monica S. Lam. SociaLite: An Efficient Graph Query Language Based on Datalog. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1824–1837, 2015.

[43] JinGang Shi, YuBin Bao, FangLing Leng, and Ge Yu. Study on log-based change data capture and handling mechanism in real-time data warehouse. In *2008 international conference on computer science and software engineering*, volume 4, pages 478–481. IEEE, 2008.

[44] Benjamin A Steer, Alhamza Alnaimi, Marco ABFG Lotz, Felix Cuadrado, Luis M Vaquero, and Joan Varvenne. Cytosm: Declarative property graph queries without data migration. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*, pages 1–6, 2017.

[45] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on networking*, 11(1):17–32, 2003.

[46] Michael Stonebraker, Uğur Çetintemel, and Stan Zdonik. The 8 requirements of real-time stream processing. *ACM Sigmod Record*, 34(4):42–47, 2005.

[47] Carlo Strozzi. NoSQL: A relational database management system. *Lainattu*, 5:2014, 1998.

[48] Gábor Szárnyas, Benedek Izsó, István Ráth, Dénes Harmath, Gábor Bergmann, and Dániel Varró. IncQuery-D: A distributed incremental model query framework in the cloud. In *International Conference on Model Driven Engineering Languages and Systems*, pages 653–669. Springer, 2014.

[49] Ora Lassila Tim Berners-Lee, Jim Hendler. *The Semantic Web.* Scientific American, 2001.

[50] Riccardo Tommasini, Pieter Bonte, Femke Ongenae, and Emanuele Della Valle. Rsp4j: An api for RDF stream processing. In *European Semantic Web Conference*, pages 565–581. Springer, 2021.

[51] Riccardo Tommasini, Yehia Abo Sedira, Daniele Dell'Aglio, Marco Balduini, Muhammad Intizar Ali, Danh Le Phuoc, Emanuele Della Valle, and Jean-Paul Calbimonte. Vocals: Vocabulary and catalog of linked streams. In *International Semantic Web Conference*, pages 256–272. Springer, 2018.

[52] Muhammad Fahim Uddin, Navarun Gupta, et al. Seven V's of Big Data understanding Big Data to extract value. In *Proceedings of the 2014 zone 1 conference of the American Society for Engineering Education*, pages 1–5. IEEE, 2014.

[53] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. PGQL: A Property Graph Query Language. GRADES '16, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Shivaram Venkataraman, Aurojit Panda, Kay Ousterhout, Michael Armbrust, Ali Ghodsi, Michael J Franklin, Benjamin Recht, and Ion Stoica. Drizzle: Fast and adaptable stream processing at scale. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 374–389, 2017.

[55] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in Action*. Manning Publications Co., USA, 1st edition, 2014.

[56] Ran Wang, Zhengyi Yang, Wenjie Zhang, and Xuemin Lin. An empirical study on recent graph database systems. In *International Conference on Knowledge Science, Engineering and Management*, pages 328–340. Springer, 2020.

[57] Peter T. Wood. Query Languages for Graph Databases. *SIGMOD Rec.*, 41(1):50–60, April 2012.

[58] Lin Ziyu, Yang Dongqing, and Song Guojie. Study on change data capture in Real-time data warehouse. *Journal of Computer Research and Development*, 44:447–451, 2007.