UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

VINÍCIUS PITTIGLIANI PEREGO

Processing Theta-Joins in Streaming Environments under the Micro-batch Model

Work presented in partial fulfillment
of the requirements for the degree of
Bachelor in Computer Science

Advisor: Prof. Dr. Cláudio Fernando Resin Geyer
Coadvisor: Manuel Dossinger

Porto Alegre
July 2019

ABSTRACT

Join computations in stream requires support for state management since tuple pairs that would generate a result might arrive in distinct moments in the application. The solution offered by Stream Processing Systems (SPS) like Spark and Storm for state management are windows limited by time or size constraint. Published papers (LIN et al., 2015; ELSEIDY et al., 2014) offer support for storing tuples without time restriction in the record-at-a-time model. In this work, we propose a solution for computing joins in a stream environment under the micro-batch model with support for state management to theta-joins. The approach stores tuples and uses a broadcast shuffle to run the broadcast join algorithm, enumerating the cartesian product between streams and thus allowing arbitrary predicates. The model is implemented in Spark Streaming and uses RDDs as storages in main memory and Apache Kafka as message-queue for data input, besides using HDFS to store results. The methodology focuses on the scalability of the solution, using the synthetic benchmark TPC-H and the queries in a left-deep-tree model. The experiments investigate the execution time and resources like network and memory for a different number of nodes. The evaluation was executed in a cluster of virtual machines orchestrated by Kubernetes in Microsoft Azure. The results show a performance gain of 40% when we double the resources and high network usage as a consequence of Broadcast.

Keywords: Stream Processing. Micro-batch. Theta-join. Apache Spark.

Processando Theta-Joins em Ambientes de Streaming sobre o Modelo Micro-batch

RESUMO

A computação de *joins* em streams requer o suporte de gerenciamento de estados, pois os pares de tuplas que geram um resultado podem chegar em momentos distintos na aplicação. A solução oferecida por *Stream Service Providers* como *Spark* e *Storm* para o gerenciamento de estados são janelas limitas por um intervalo de tempo ou tamanho. Trabalhos publicados (LIN et al., 2015; ELSEIDY et al., 2014) oferecem suporte a um armazenamento sem restrição de tempo no modelo de streams *record-at-a-time*. Nesse trabalho propõe-se uma solução para computar joins em stream sobre o modelo *micro-batch* com suporte a um gerenciamento de estados em memória para *theta-joins*. O método armazena tuplas e realiza um *broadcast Shuffle* para então utilizar o algoritmo *broadcast join*, enumerando o produto cartesiano entre streams e permitindo predicados arbitrários. O modelo foi prototipado sobre a API Spark Streaming e utiliza RDDs como armazenamento em memória principal e o Apache Kafka como message-queue (MQ) para entrada de dados, além do HDFS para armazenamento de resultados. A metodologia foca na escalabilidade da solução, utilizando o *benchmark* sintétipo TPC-H e queries no modelo *left-deep-tree*. Os experimentos investigam o tempo de execução e o uso de recursos como rede e memória para diferentes números de nós . A avaliação foi executada em um cluster de máquinas virtuais orquestrado pelo Kubernetes na Microsoft Azure. Os resultados mostram um ganho de desempenho de 40% ao dobrar recursos e um alto uso de rede como consequência do *broadcast*.

Palavras-chave: Stream Processing, Micro-batch, Theta-join, Apache Spark.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BD	Big Data
DAGs	Directed Acyclic Graphs
DF	DataFrame
DS	Dataset
FW	Framework
HDFS	Hadoop Distributed File System
HMR	Hadoop MapReduce
MJ	Multiway Join
MQ	Message Queue
MR	MapReduce
RDBMS	Relational Database Management System
RDD	Resilient Distributed Dataset
TJ	Tributary Join
SPS	Stream Processing Systems

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Join is an operation traditionally used in Relational Database Management System (RDBMS), but also explored in parallel context since the 90's (WILSCHUT; APERS, 1993). Moreover, some works also used other architectures such as FPGA (TEUBNER; MUELLER, 2011) for computing joins. However, the focus of recent research is the implementation of joins in frameworks for batch processing like MapReduce (AFRATI; ULLMAN, 2010) and real-time stream applications(ELSEIDY et al., 2014; CHU; BALAZINSKA; SUCIU, 2015).

The popularity of the MapReduce paradigm (DEAN; GHEMAWAT, 2008), initially proposed by Google in 2004, comes mainly from its open-source implementation - Hadoop MapReduce (HMR) - the precursor of tools like Storm, Spark, and Flink that consolidates the current scenario of Big Data (MATTEUSSI et al., 2018). Some criticize MR due to its lack of schema and declarative query language (BLANAS et al., 2010), but the HMR Ecosystem offers support for SQL queries through extensions like Hive[1].

In this scenario, companies use joins - an operation for merging different sources based in predicate - for processing large volumes of data from continuous incoming sources. For instance, the Google's Advertisement System uses an application to join web search queries with user clicks (ANANTHANARAYANAN et al., 2013). This application produces messages containing the terms in the user search, and also the advertisement clicked and can be used to bill advertisers but also extract what customers of a particular business are searching. Other interesting join applications in stream processing include Internet Protocol (IP) network management and telephone fraud detection (DAS; GEHRKE; RIEDEWALD, 2003).

These examples represent the simplest case of join where only two relations are considered. However, there are joins with multiple relations, defined multiway joins (MJ). MJ introduce some complexities, such as the order of joining relations to mitigate response time. Although not very common, MJs also appears in real world: at LinkedIn[2], multiples stream sources are joined to produce an aggregated view on the dashboard (ZHUANG et al., 2016)

This work presents a solution for computing joins with arbitrary predicates in a streaming environment. The proposed model discuss different alternatives, such as storing intermediate and optimizations for equijoins. The implemented prototype is analyzed by

[1] https://hive.apache.org/
[2] https://www.linkedin.com

measuring the execution time and the tradeoff between network and memory when using left-deep trees. Results show the stability of the application and the advantages of using intermediate results, but also that it is not a suitable solution when the join selectivity is close to the cartesian product.

1.1 Motivation

Stream applications deal with continuous and unbounded sources of data in the form of tuples or events (BORDIN, 2017). These applications consist of multiple operators, which can be either stateless or stateful. Stream Processing Systems (SPS) offers limited support for stateful operations since this introduces additional complexities such as fault-tolerance and in some cases persistence (FERNANDEZ et al., 2013). Joins in streaming environments requires stateful management (FERNANDEZ et al., 2013), not only processing tuples but maintaining them for a possible result with late arriving tuples.

Most proposed solutions for joins are window-based, where the window is either time or size constraint (GULISANO et al., 2017). The solutions in the literature offer state management for joins (ELSEIDY et al., 2014; LIN et al., 2015) in record-at-a-time SPS, where a detailed routing is required. This motivates research towards a new alternative by using the micro-batch processing model, that mitigates difficulties such as tuple routing.

1.2 Goals

The focus of this work is to propose a solution for computig joins involving multiple relations with arbitrary predicates in streaming environment, ensuring correctness of the result with an exactly-once semantic (i.e., no duplicate results). To achieve this, we define the following specific goals:

- Promote a study of the state-of-the-art for computing joins in distributed environments to understand the challenges in the area and implement a solution capable of addressing such problems;
- Present a model for processing theta-joins in streaming micro-batch model using a stateful operator, implementing a prototype for this model in Spark Streaming;
- Present the scalability of the model, showing a consistent performance when more resources are added;

1.3 Organization

This text is organized as follows. Chapter 2 presents the background and is divided into two parts. The first part focuses on stream processing, with a review of some concepts and the mechanisms of frameworks. The second part of the chapter reviews join as in relational databases and also introduces some algorithms in online join processing. The chapter is concluded with some remarks of the limitations in the standard implementations of joins, which motivates the research of join processing. Chapter 3 explores recent works of join processing in both batch and streaming distributed environments. Chapter 4 presents the system design and also the implementation in Spark, along with the methodology and evaluation. At last, Chapter 5 concludes the research with a few observations and suggestions for future works.

2 BACKGROUND

This chapter shows a brief introduction to Big Data, with a focus in real-time processing by exploring SPSs used in this research and also recent work. We exploit the features and limitations of these tools (described in section 2.3). The second part recapitulates joins as defined in RDBMS, and also introduces some algorithms for processing join in different scenarios.

2.1 Real-time Analysis

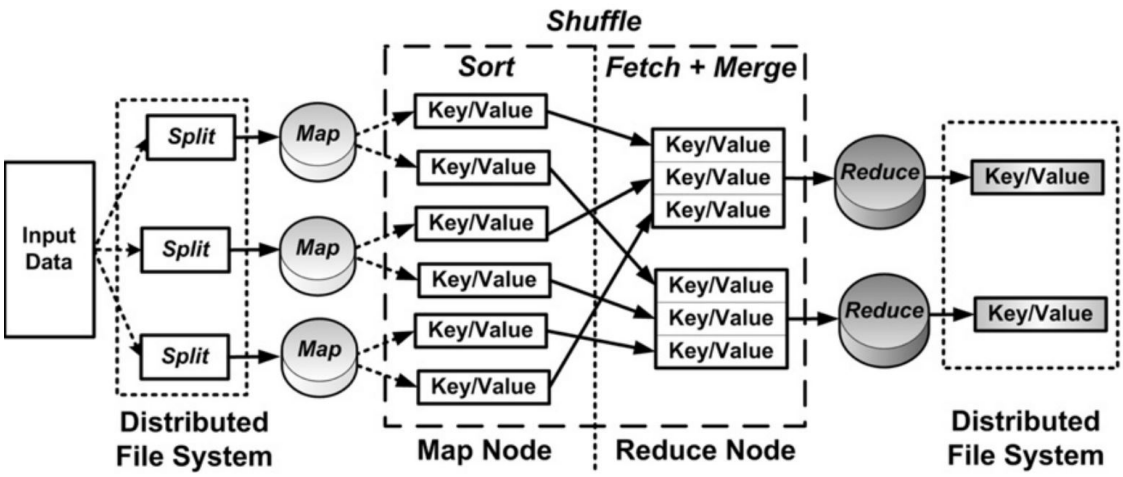
With the capacity offered by modern hardware and the demand for applications with answers in the order of seconds, arises the need for tools processing data in real-time or near real-time (BORDIN, 2017). These tools take away responsibilities from the programmer, such as task schedule and fault tolerance and uses the concept of streams: continuous and simultaneous data sources with unpredictable flows (JUNIOR, 2018).

SPS uses a Directed Acyclic Graph (DAG) to design applications, where each node is an operation. An operator can be *stateless* (e.g.,filter and map), processing tuples without storing them or *stateful* (e.g., joins and aggregations). An example of stateful application is the word-count that requires to store the number of words for updating.

A stream can be modeled in a record-a-time approach, where operators process tuples individually, or micro-batch, where a discrete period of time is defined to track a set of incoming tuples instead of processing them individually.

2.1.1 Hadoop MapReduce

Figure 2.1: MapReduce flow



(a) Image extracted from (KOLBERG et al., 2013)

MR (DEAN; GHEMAWAT, 2008) is a programming abstraction for parallel and distributed programming based in map and reduce functions from functional programming. A MR application has two three main phases: map, shuffle and reduce (KOLBERG et al., 2013). First, the map function takes a single instance of data as input and produces a set of intermediate key-value pairs. Secondly, the intermediate data sets are automatically grouped over the keys. Finally, the reduce function takes as input a single key and a list of all values generated by the map function for that key(MATTEUSSI et al., 2018). MapReduce uses a Master-Slave architecture, where the Master node controls task distribution and scheduling and workers execute map and reduce functions (KOLBERG et al., 2013).

Hadoop MapReduce is a popular open-source implementation of MR which uses the Hadoop Distributed File System (HDFS) for storing large blocks of data and provides high throughput for sequential read and writes operations (ANJOS et al., 2018). The MR paradigm motivated two classes of data processing in Big Data that are not only based in MR: Stream Processing and Batch Processing (SOUZA et al., 2018)

2.1.2 Apache Storm

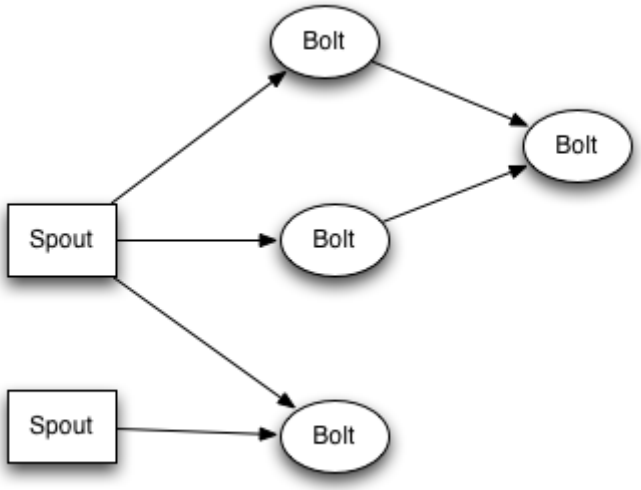
Storm is a framework proposed by Twitter for stream processing in the record-at-a-time model. The streaming operations are defined in Storm topologies, as a graph of

operations composed of two kinds of vertices: spouts for stream sources and bolts for general operations. The output of a bolt can be used as the input for a next bolt or just the output, and the Storm topology can have cycles (TOSHNIWAL et al., 2014). Figure 2.2a shows a topology example.

Storm permits a fine-grained solution for routing tuples between the vertices in the topology. Given one producer vertice and a set of possible consumers, tuples can be either randomly assigned (shuffle) to a consumer, based on a field (hash) or sent to all consumers.

The Storm topologies are submitted to the Storm cluster, composed of a master (called Nimbus) that is simmilar to the "JobTracker" in Hadoop, which coordinates the execution of the cluster and worker nodes; The worker nodes run one or more worker process, and each process can be assigned for a single topology. Processes manage the executors, who at last run tasks (bolts and spouts). In a high level, the worker processes are containers, and a single node can run multiple topologies by hosting multiple processes.

Figure 2.2: Storm topology example



(a) Image extracted from (APACHE. . . , 2019)

2.1.3 Apache Spark

Spark is an engine for processing massive data volumes in main memory, with support for batch or stream processing. Programmers write batch applications in Spark using the main RDD API or Spark SQL. Streaming applications can use either DStream or Structured Streaming, which extends RDD and Spark SQL, respectively. The Spark

Cluster uses the concept of Master and Workers. However, to allow distinction between different applications running in the cluster, Spark uses the name **Driver** for an application Master and **Executor** for the workers. Both Driver and Executor are instances running in Workers, and Spark Master orchestrates applications.

The core abstraction is the RDD (ZAHARIA et al., 2012), immutable collections of data partitioned among worker nodes that can be interacted through transformations and actions. Transformations are operations (e.g., map, filter, reduce) that triggers new tasks in each partition of the RDD and generates a new one as a result. Actions are output operations, such as counting the number of elements in the RDD or writing data to persistent storage (MATTEUSSI et al., 2019).

Spark tracks the set of transformations through a DAG, where each node is a transformation except for the last one that must be an action. This property is a consequence of the lazy evaluation, where no transformation executes if there is no output (i.e., an action) for data. The lazy evaluation also guarantees that intermediate nodes in the DAG are not kept in memory, thus saving memory resources. In case an operation executes often, users can configure Spark to keep that intermediate results stored in memory.

In case of task failure, Spark uses the DAG to recompute the previous transformations and finally execute the failed task for one of the partitions. This approach has a lower cost than using replication for the entire RDD.

The Spark Streaming API (ZAHARIA et al., 2013) extends the original RDD abstraction to enable stream processing. It implements the micro-batch model, where incoming data from a specific time interval is stored into multiple RDD's and processes as a deterministic batch computation (ZAHARIA et al., 2013). Operations in streaming data can be either stateless like map or stateful through sliding windows. However, continuous storage through the entire application is not offered.

Spark SQL (ARMBRUST et al., 2015) offers a relational approach to use the RDD model, using the query optimizer Catalyst to improve performance operations. For instance, Catalyst can optimize an arbitrary Join that would be computed with the cartesian product to use a broadcast join.

2.1.4 Communication and Storage

Besides the frameworks, it is also worth to mention the following tools used in this work for the purpouse of storage (HDFS) and communication (Kafka).

- **HDFS:** The distributed filesystem offers convenience for dealing with persistence in a cluster, in comparison with local filesystem;
- **Apache Kafka:** Distributed message system with high throughput, providing a partitioned and replicated commit log service named topics for processing data in real time (JUNIOR, 2018);

2.2 Join Processing

This section starts with a review of definitions from join processing as defined in classic database courses, based in (ELMASRI; NAVATHE, 2010). Later, we provide some solutions for processing joins in batch and stream.

2.2.1 RDBMS and Batch

In database systems, join ($\bowtie$) is an operation between two relations that produces tuples containing attributes from both relations. A join operation evaluates a predicate between attributes of each relation. The most common case of join is an equijoin, where an equality predicate is used. A theta-join ($\bowtie_\theta$) between relations R and S produces all combinations of tuples that satisfy an arbitrary binary predicate θ. The case of joining tuples without a predicate produces a cartesian product ($\times$).

The sequential solution for theta-join in RDBMS is a naïve nested-loop that tests all pairs of tuples. Implementations for parallel join use the principle of partitioning relations to run individual joins in each partition. The general case for a join $R \bowtie_\theta S$ takes r and s partitions and $r * s$ processes to compute the join separately. In case one of the relations - say S- is small enough, it can be replicated instead of partitioned, using then r processes.

Joins between multiple relations - named multiway joins (MJs) - require choosing an order of joining relations to reduce intermediate results. The result size of a join is a consequence of the *join selectivity(J_s)*, a factor expressed in the range [0,1], where 0 means an empty result and 1 is the cartesian product, obtained as in formula 2.1.

$$J_s = \frac{|R \bowtie S|}{|R \times S|} \tag{2.1}$$

Figure 2.3: Join order for 3 relations

(a) Left-deep tree (b) Multiway join

Figure 2.4 shows a theta-join example $Marteking \bowtie_\theta Production$, where $\theta = Marketing.Salary < Production.Salary$. In the result (Figure 2.4c), the attributes (i.e., columns) with M identifies $Marketing$ tuples and P $Production$. The join selectivity of this example is 0.5.

Figure 2.4: Theta-join example

Name	**Salary**
John Doe	2500
Richard Miles	4000

(a) Marketing relation

Name	**Salary**
Jane Doe	3000
John Smith	3500

(b) Production relation

M.Name	**M.Salary**	**P.Name**	**P.Salary**
John Doe	2500	Jane Doe	3000
John Doe	2500	John Smith	3500

(c) Result

MJs can be executed as a cascade of binary joins, as in the left-deep tree (Figure 2.3a), where relations are joined from left to right. The other is joining all relations at once (Figure 2.3b)

The MR paradigm does not support directly join application since it receives a single input file (OKCAN; RIEDEWALD, 2011). However, it is easy for a programmer to write an equijoin. Algorithm 1 (described in (OKCAN; RIEDEWALD, 2011)) shows map and reduce tasks for an equijoin $R.A = S.A$ between R and S. The map task append a new field to the tuple to identify the relation it belongs to and emits the join attribute as key and the tuple as value. The reduce tasks must produce the cartesian product of all tuples for a given key.

This solution does not work well for datasets with high skewness, where many tuples have the same key, leading to a heavy load for some reduce tasks. Also, joining

more than two relations requires additional jobs. (AFRATI; ULLMAN, 2010) studies the case of joining multiple relations in a single job instead of binary cascade joins, implementing the solution in MR but not only limiting to the paradigm. The authors explore the communication cost as an optimization problem solved through "Lagrangean Multipliers"(AFRATI; ULLMAN, 2010) to find the ideal number K of reduce tasks.

Algorithm 1 Equijoin map and reduce tasks

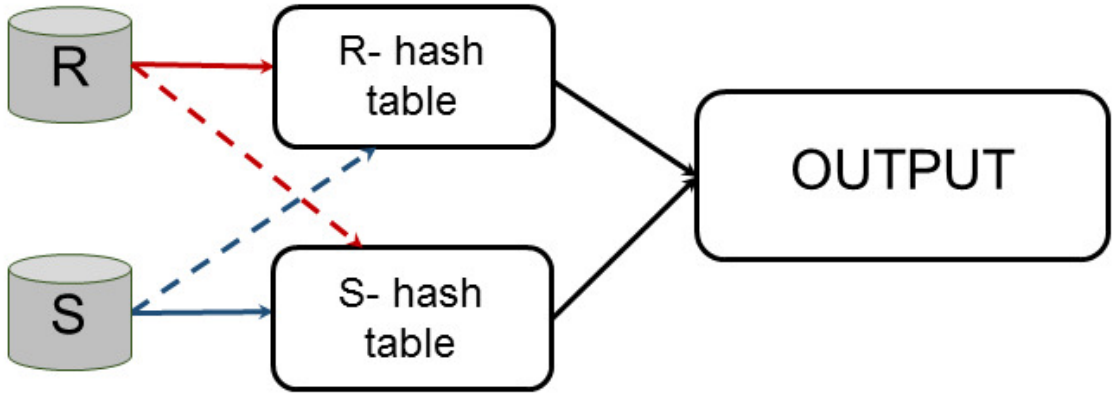
```
1:  procedure MAP(relation, document)
2:      for all t ∈ document do
3:          t.Origin ← relation
4:          EMIT(t.A,t)
5:      end for
6:  end procedure
7:  procedure REDUCE(key, tuples[t_1, t_2, ...])
8:      R ← tuples.filter(t.Origin = R)
9:      S ← tuples.filter(t.Origin = S)
10:     for all r ∈ R do
11:         for all s ∈ S do
12:             EMIT(r,s)
13:         end for
14:     end for
15: end procedure
```

2.2.2 Online Join Processing

The Symmetric Hash-Join (SHJ) (WILSCHUT; APERS, 1993) is a classic algorithm for processing equijoins in streaming. The algorithm creates a hash-table for the relations and each tuple is firstly inserted and later sent to probe in the other table, sending matches to the output (Figure 2.5). The join occurs in a single phase and enables a non-blocking streaming operator since the processing occurs symmetrically .

XJoin (URHAN; FRANKLIN, 2000) extends SHJ to enable joins between large relations by storing tuples in secondary storage. Tuples are stored in partitions that have a share both in memory and disk. If an incoming tuple arrives in a memory overflow scenario, a partition is chosen as a victim and sent to disk. The algorithm runs in three stages: it starts by joining tuples in main memory similar to SHJ, and it stops after a timeout or if sources become idle. The second phase takes tuples from disk and probe against data residing in memory from the other relation. The last phase "clean-up" relations by joining missing results from tuples that were kept in disk. To avoid duplicate in the second and

Figure 2.5: Symmetric Hash Join



the third phase, the algorithm uses a timestamp to check if the time that tuples resided in memory overlap.

(VIGLAS; NAUGHTON; BURGER, 2003) adapts the XJoin for MJs. The authors address some of the challenges in joining multiple relations, such as the join order to reduce intermediate results, which is solved by ordering according to the selectivity of the predicate.

2.3 Preliminary Remarks

Join is an expensive CPU task, and running in stream environments addresses challenges such as avoiding duplicate results and state management, which is challenging since SPS does not support state for stream operators and lend the responsibility for developers of stream applications (FERNANDEZ et al., 2013).

Both Storm [1] and Spark officially supports only joins between streams with equality predicate and through the usage of window. Recently, the new Spark API for streaming (Structured Streaming[2]) offers a full-history join option by implementing the SHJ.

The lack of support for joins with arbitrary predicates in the frameworks motivates research to find ways to compute these applications without limiting the constraint of a window based on time or size. In Chapter 3, we introduce recent work for performing this stream operation, but also some cases in batch processing.

[1] https://storm.apache.org/releases/2.0.0/Joins.html
[2] https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

3 RELATED WORK

This chapter presents recent works in join processing, addressing both streaming and batch solutions. Section 3.2 concludes the chapter with an overview of the features from mentioned work and this research.

3.1 Analisys

Since modern frameworks process data in main memory without using the disk as in the MapReduce approach, authors have addressed the network as the bottleneck of join processing, thus focusing in optimizing the shuffle phase of the computation (ZHU et al., 2017).

Some works (LIN et al., 2015; ELSEIDY et al., 2014) in Streaming Context describes a solution for Theta-Joins mainly for binary joins. Modern works focus on full-history join instead of the window-joins, where tuples are only joined if they arrive in the same period of time. Full-history joins require a stateful streaming operator, as the tuples must be kept in memory during the whole computation.

Matrix-model (ELSEIDY et al., 2014; FANG et al., 2016) implementations start with an initial partitioning and monitor the cluster to process new matrix schema and migrate task-loading. The join-matrix uses a matrix M with I rows and J columns to compute $R \bowtie_\theta S$, dividing relations R and S into I and J partitions, respectively. Each matrix cell corresponds to a task. To produce a correct result, each partition of R is replicated $J - 1$ times (analogously, the partitions of S have $I - 1$ replicas).

In (ELSEIDY et al., 2014), the authors focus on a non-blocking join operator with the Matrix model and also proposes an adaptative solution to reduce data replication by monitoring the storage. The author's implementation of the Matrix model assumes a cluster of N machines with $N = I * J$, where each node holds a pair of partitions. In the evaluation, a variation of TPC-H (THE..., 2019) with skew is used to test the

Figure 3.1: Assignments of partitions in Join-Matrix and Join-Biclique.

(a) Join-Matrix assignment

(b) Join-Biclique assignment

adaptive approach with two static partitions, one using optimal partitioning and another that assumes equal-sized streams to partition their data, in addition to performing joins with SHJ. The results show that for high skew indices, the equal-size static model and the SHJ overflow and spit to the disk fast, but the static model still has a shorter execution time than the hash partitioning algorithm since it assigns the entire join to only a few machines. The adaptative partitioning has a performance similar to the optimal solution.

In (FANG et al., 2016), the authors criticize the fixed partitioning of the traditional matrix model since it is not optimized for processing joins in cloud platforms, where idle resources can be returned. To have a better performance in the cloud, they propose an adaptatively to adjust workload and minimize resource such as network to reduce cloud costs. The architecture monitors the workload and generates a viable migration plan. Implementing the proposal in Storm, they use a variation in TPC-H with skewness and a Chinese social dataset to compare their approach against (ELSEIDY et al., 2014) and (LIN et al., 2015), to measure metrics like execution time, throughput. Results show a stable performance of both matrix solutions in skew datasets when comparing against bipartite-graph, as task assignments are random. Indeed, BiStream uses a hash partitioning for equi-joins.

Another paradigm, named the Join-Biclique (LIN et al., 2015), divides the cluster in a bipartited graph where each side represents a relation. Incoming tuples are stored in one of the nodes belonging to the storage of its relation and also probed against all nodes of the other relation in search of join partners. An advantage of this model in comparison to the join-matrix is that no replication is needed.

BiStream (LIN et al., 2015) is implemented in Storm and evaluates its performance in comparison with (ELSEIDY et al., 2014) by measuring latency, throughput, and memory consumption. Results prove that BiStream is more memory-efficient since matrix model requires storage. Another consequence of the replication is the difficulty to scale the amount of processing tuples when adding more resources, while BiStream has almost a linear scalability.

(HOFFMANN; MICHEL, 2017) proposes an alternative for MJs based in the Join-Biclique and implementing the solution using Storm. The paper evaluates a single query in different query-plans, discussing the advantages of storing intermediate results and reducing the costs with network.

Hypercube (BEAME; KOUTRIS; SUCIU, 2014) (HC) describes an approach with a single communication round (i.e., shuffle) for processing multi-way equi-joins. After

the shuffle, each server contains a part of the relations and the join is then processed in a binary way.

(CHU; BALAZINSKA; SUCIU, 2015) study the advantages of the single communication round by implementing the Hypercube shuffle, but also define a multi-way join algorithm called Tributary Join (TJ) based in the sort-merge join. The authors evaluate the performance of combining the proposed join algorithm ("Tributary Join") with Hypercube shuffle by comparing with another join (traditional SHJ) and also two shuffle approaches (broadcast and hash partitioning), measuring wall-clock time, CPU time and network I/O in a Twitter dataset. The broadcast requires more shuffles and CPU time with TJ since it is necessary to sort data, but the query runtime is smaller than regular shuffle since the skewness disturbs the load balance in the regular shuffle. The authors conclude that queries with large intermediate results and small final results perform better with the combination HC and TJ.

Considering queries with opposite behavior (i.e., small intermediate results), (ZHU et al., 2017) studies in which scenarios a multi-round approach can be used, but to choose between shuffle strategies, it would be necessary to know the intermediate result in advance. The authors propose a sampling algorithm to estimates the intermediate result size in the multi-round and also calculates the size in one-round and choosing the join strategy that implies in less shuffle. For the single communication, the HC is used again with TJ with optimization and a heuristic to select the join order.

AutoMJ (ZHU et al., 2017) is implemented in Spark, who uses by default a multi-round solution for joining multiple relations. This implementation is compared with the proposed one-round solution by measuring execution time, and the number of tuples shuffled for Twitter and Wikipedia datasets.

3.2 Discussion

Table 3.1 presents a comparison of the mentioned works according to the processing model and join characteristics. The proposed join solutions use workload characteristics to reduce communication cost for MJs, focusing in the impact of skewness in the shuffle performance for equijoins that partition tuples according to attributes to reduce CPU time in the join algorithm.

On the other hand, using workload statistics in the streaming scenario would lead to a high impact in the latency, and it is also not easy to make use of this information

since most SPS does not support dynamic graph to change applications without restarting (BORDIN, 2017). The analyzed works in streaming address theta-joins, giving a solution to this kind of problem that is not addressed by SPS. These methods exploit different partitioning methods such as matrix and the join-biclique, which shows resilience against skew workloads. (HOFFMANN; MICHEL, 2017) is the only stream solution to discuss MJs, introducing the problem of choosing the join order and the possibility to store intermediate results.

All the mentioned solutions for stream use record-at-a-time model and only (HOFFMANN; MICHEL, 2017) exploits MJs. Join applications could benefit from the micro-batch model, as it offers high throughput and mitigates difficulties such as tuple routing (BORDIN, 2017). Also, the broadcast shuffle is not mentioned in the streaming scenario, which could be exploit together with small micro-batches to compute theta-joins without shuffling a high network impact.

Table 3.1: Related Work

Reference	**Model**	**Join-Type**
(HOFFMANN; MICHEL, 2017)	Stream, record-at-a-time	Theta-Join, MJ
(ZHU et al., 2017)	Batch	Equi-Join, MJ
(FANG et al., 2016)	Stream, record-at-a-time	Theta-Join, Binary Join
(LIN et al., 2015)	Stream, record-at-a-time	Theta-Join, Binary Join
(CHU; BALAZINSKA; SUCIU, 2015)	Batch	Equi-Join, MJ
(ELSEIDY et al., 2014)	Stream, record-at-a-time	Theta-Join, Binary Join
(BEAME; KOUTRIS; SUCIU, 2014)	Theoretical	Equi-join, MJ
Proposal	***Stream, micro-batch***	***Theta-Join, MJ***

4 SYSTEM DESIGN

In this chapter, we present the model (section 4.1) and the prototype (section 4.2) developed in Spark Streaming. Section 4.3 defines the methodology used to evaluate the prototype, with specifications of workload and the metrics used. In 4.4 we present the results obtained.

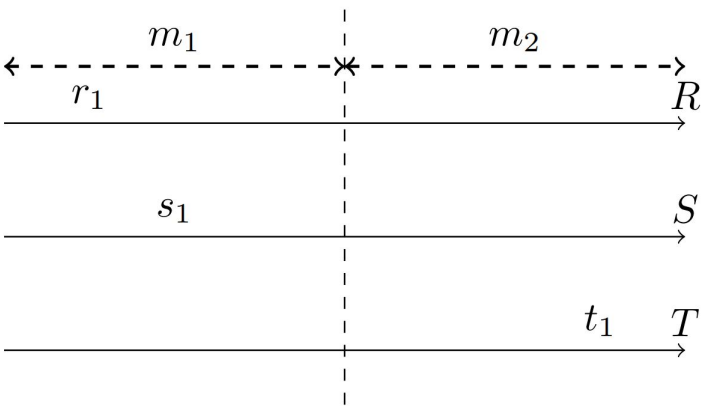
4.1 Model

Given stream sources (e.g., a folder in a filesystem or a MQ) $S_1, S_2, ..., S_{n-1}, S_n$ with a MJ (as in equation 4.1) involving all streams, we want to compute all joins in a distributed environment using main memory. In a stream application with the micro-batch model, the tuples that would generate a possible result - called "join partners"- may arrive in different batches and consequently not producing the result.

$$S_1 \bowtie_\theta S_2 \bowtie_\theta S_3, ..., S_{n-1} \bowtie_\theta S_n \tag{4.1}$$

This problem of tuples in different micro-batches is described in Figure 4.1, where r_1,s_1 e t_1 are join partners, however, r_1 and s_1 arrive in the first micro-batch (m_1), while t_1 belongs to m_2. To ensure that tuples from different micro-batches are evaluated (i.e., tested against the join predicates), it is required to maintain them in storage for each relation that guarantees the availability of tuples throughout micro-batches.. Supporting state in stream processing is commonly referred as state management, and in join literature is defined as full-history join (ELSEIDY et al., 2014; LIN et al., 2015; FANG et al., 2016).

Figure 4.1: Tuples arriving in different micro-batches



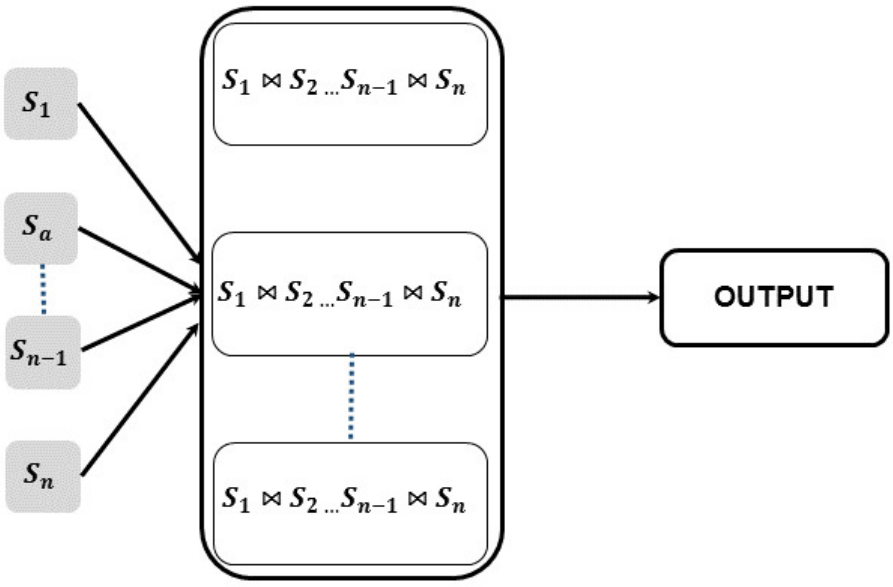
The stateful operator -referred here as storage - can be provided by any data struc-

ture that keeps tuples only in main memory and partitioned among nodes, without replicas. It is not in the scope of this project to support persistence in second storage and fault tolerance for retrieving stored tuples in case of application failure. Consequently, if the distributed memory is not enough for storage and task computation, errors may happen, requiring the application to be restarted with more resources.

The micro-batch model requires the definition of time T_m that express the size expected to perform all the operations. If this time is exceeded, the application blocks since new tuples will not be ingested until the end of the micro-batch processing. It is necessary to define a rate in the sources or the stream application to reduce incoming tuples. The cost for processing stateful theta-joins is not trivial since the storages increase at each micro-batch and the join algorithm needs to enumerate all pair of tuples. Suppose a MJ between N streams and all streams have incoming rate R. This way, the worst-case (i.e., a full cartesian product) in the micro-batch m, the cost would be $O(m * n^r)$.

Figure 4.2 presents the overall architecture of system. For a single micro-batch, the application has two main phases: store (distribute tuples from $S_1, S_2, ...S_n$ among nodes) and join (run tasks in each node).

Figure 4.2: System Architecture



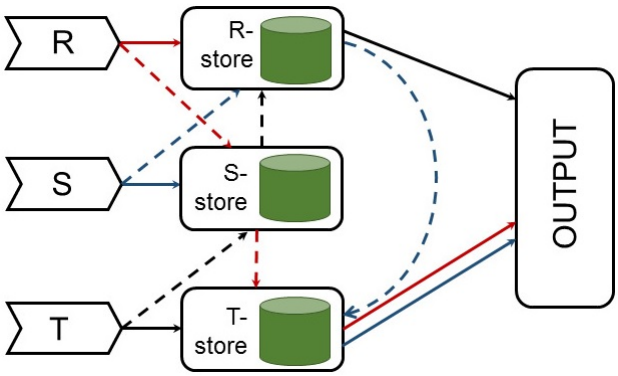
In the first part, each stream sends tuples to its storage according to a partitioning scheme. For equip-joins, we provide a hash-partitioning, using the attributes from the join predicate as the hash key. This way, in a join $R \bowtie S$ with predicate $R.A = S.B$ the tuples from R will be partitioned according to the value of A (analogously, S according to B). This solution reduces network communication in the shuffle phase since join partners

are in the same node.. Theta-joins use a random shuffle, which gives load balance and resilience against skewness at the cost of high network usage for shuffling tuples.

In the join phase, one task is created for each partition in the storage. The tasks run a join algorithm that enumerates the cartesian product and filters tuples that do not fill the requisites from predicates. In the case of equijoins, the join partners (i.e., tuples with the same key) are already in the same task, so the join algorithm used is the same as the reduce task defined in 1. For theta-joins, the Broadcast Join (Algorithm 2) is used.

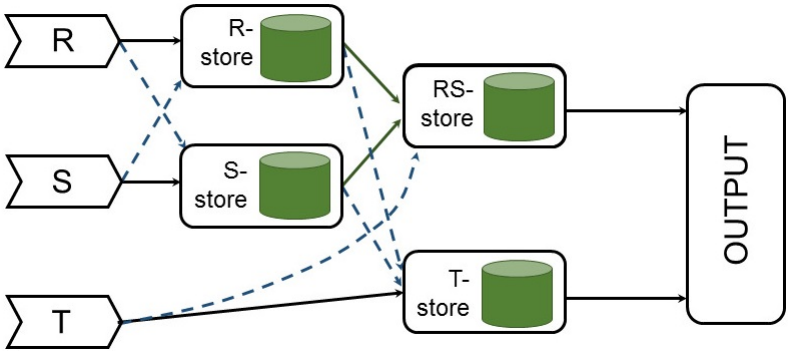
The join algorithms take advantage of small stream ingested in a single micro-batch, and thus **broadcasting** it for all nodes storing the relation that must be joined. In MJs, this approach can broadcast streams and join with one of the storages, or use a left-deep tree to consume less memory with broadcast and joining individually.

Figure 4.3: A Join without intermediate result



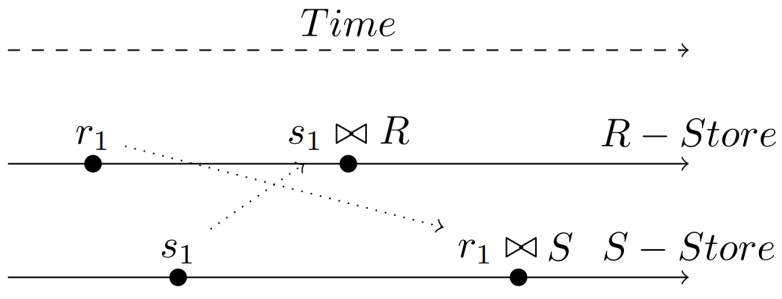
For joins of 3 or more relations, we use additional storage for intermediate results. Storing the intermediate data increases memory usage, but reduces data sent through the network when the *join selectivity* is low. Consider the example in Figure 4.3, where no intermediate result is stored. Tuples from T are first stored (dashed black line) and then must be probed twice (dotted black lines): first, against S store and later to R. Figure 4.4 describes the same join but with the intermediate result between R and S stored in *RS-Store* (this is similar to a left-deep tree). Here tuples from T are probed only against *RS-Store*. Notice that the particular case of joining two relations is similar to the BiStream model.

Figure 4.4: Intermediate result storage



To guarantee the correct result the join runs in the storage of both relations, as shown in the pictures above where $R \bowtie S$ runs in both $R - Store$ and $S - Store$. However, this also introduces duplicate results since the tuple might be generated in both storages. This problem is solved by using a timestamp that is appended to tuples when they are stored. The timestamp is used as an additional condition in the filter together with the join predicates. For a pair to be produced, it is required that the timestamp from the storage tuple is smaller than the probed tuple. Consider Figure 4.5, where tuples r_1 and s_1 must join. After being stored and sent to probe against the other relation (e.g., $s_1 \bowtie R$ in $R - Store$), the tuple (r_1, s_1) would be emitted twice in the output. With the timestamp, the result is generated only in $R - Store$ since the stored tuple (r_1) has a smaller timestamp than the probed one (s_1).

Figure 4.5: Timestamp example



4.2 Prototype

Algorithm 2 Broadcast Join Algorithm

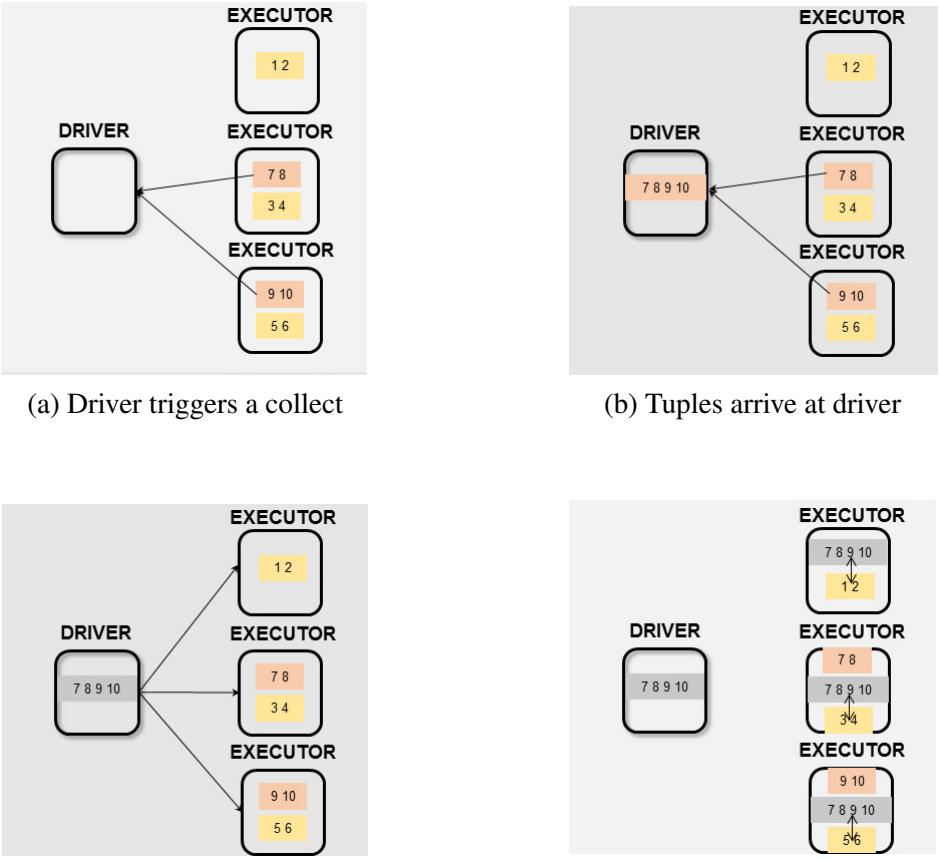
```
1: function BROADCASTJOIN(storageRDD,broadcastRDD)
2:     storageRDD.mapPartitions{  storageTuple =>
3:         broadcastRDD.map{  broadcastTuple =>
4:             (storageTuple, broadcastTuple)
5:         }
6:     }.filter{outputRow => joinCondition(outputRow)}
7: end function
```

We implement our model in the Spark Streaming API, using Kafka's MQ functionality as the stream sources, where each topic is a relation. The integration between Kafka and Spark helps to define the degree of parallelism since the partitions in a topic translate to the partitions for the RDD. Consequently, the number of join tasks can be easily defined when creating a topic. Also, the partitioning scheme used by Kafka is random, which is kept in Spark application to avoid an additional shuffle. Applications that use other stream sources (e.g., a folder in a filesystem) can be used but requires a shuffle operation in Spark (which requires the number of partitions as an input) in order to guarantee the desired level of parallelism

The storages are RDDs that updates in each micro-batch through a union transformation followed by a cache, which tells Spark to keep the RDD for the next micro-batches. After storing, the DStream containing only the new tuples are probed against the other storage (i.e., another RDD) and joined. We use the Broadcast Join (BLANAS et al., 2010), which enumerates the cartesian product of the joining relations, but it does not keep this data in memory. Algorithm 2 describes the used join. Suppose a cluster with N nodes; the *storageRDD* Storage (an RDD containing all tuples from a relation, such as $R - Store$ and $S - Store$ from Figure 4.4) has a number P of partitions (defined initially in the Kafka topic), where each node stores one partition (it may contain more if $P > N$), and all nodes have a copy of *broadcastRDD* (i.e., the tuples sent to probe against the other storage). This *broadcastRDD* contains only tuples from the current micro-batch instead of the whole relation. Line 4 of algorithm 2 enumerates tha pairs of tuples and the *filter* operation in line 6 takes a function (*joinCondition*) with the join predicates and the timestamp verification in the result of the *mapPartitions* operation.

Broadcast is generated through an operation offered by Spark with the same name that only shares data contained in the Driver (recall from ection 2.1.3 that the *Driver* is the Master node in the context of a Spark application). Thus, the stream needs to be first collected to the driver to be able to broadcast it later. To reduce network usage, only the DStream (i.e., the new tuples in a micro-batch) is used as broadcast instead of the whole relation.

Figure 4.6: Join execution



(a) Driver triggers a collect

(b) Tuples arrive at driver

(c) Broadcast is sent to executors

(d) Broadcast is performed

Figure 4.6 illustrates a broadcast join example, $R \triangleright\!\!\lhd_\theta S$. Recall from section 2.1.3 that in a Spark application the Driver serves as the master and Executors are the workers. In this example, the colored boxes are partitions, where the orange ones belong to relation R and the yellow from S. In 4.6a R partitions are collect to the driver, and once they arrive 4.6b Spark creates a broadcast variable (the gray box in Figure 4.6c) that is sent to all nodes. At last, in Figure 4.6d the broadcast join executes, sending join partners to the output. After the join finishes, the broadcast variable is removed from all workers, and the next micro-batch repeats the process.

4.3 Methodology

The evaluation of this work focus on the scalability of the proposed join by checking execution time, memory usage, and network (as defined in 4.3.1). To achieve this, we use TPC-H (THE. . . , 2019) benchmark with queries Q2, Q3, and Q5 (specified in 4.3.2).

We measure the execution time of Q3 and Q5 to check the performance gain when we double the resources. Therefore we run both queries with 1, 2, 4, and 8 nodes and a fixed workload with SF 0.1 (100 MB). In the second set of experiments, query Q2 is executed with two different values of SF: 0.1 (100 MB) and 0.2 (200 MB). We vary the number of workers with as 2, 4, 6, and 8. In all cases, the queries run in a left-deep tree, using intermediate results for each pair of joins.

For all experiments, the topics in Kafka are already filled with tuples before Spark starts. Therefore Spark reads all tuples and joins in a single micro-batch. In this case, we fix the micro-batch size (i.e., the time) with 12 seconds, as this has no impact on the performance. The queries run in a left-dep tree, using intermediate results for each pair of join. Also, the numbers obtained are an average of six runs.

4.3.1 Metrics

The execution time (ΔT) is measured as $\Delta T = T_f - T_0$, where T_0 and T_f are the timestamp for the first and the last pair of joined tuples, respectively. To obtain an approximation of the join time, we ingest the whole dataset in a single burst into Kafka, allowing Spark to process data without having to wait for incoming tuples.

The overlall memory cost (M_c) for **storage** in a join between relations $S_1, S_2, ..., S_n$ with joins $S_1 \bowtie_\theta S_2, S_2 \bowtie_\theta S_3, ...S_{n-1} \bowtie_\theta S_n$ can be estimated by the size of relations and intermediate results. In 4.2 formula below, $|S_i|$ denotates the size of a relation, $|S_i \bowtie_\theta S_{i+1}|$ is the size of the intermediate result of a join between two relations and Bt_{Si} is relation Si in the broadcast variable at a given microbach t. B can be equal to a fraction of Si or the whole relation (i.e., $|Bt_{Si}| = |S_i|$). We use the default fractions of memory defined by Spark[1]: 40% reserved by Spark and the other 60% is split in half between

[1]https://spark.apache.org/docs/latest/tuning.htmlmemory-management-overview

storage and execution (i.e., memory used to compute tasks).

$$M_c = \sum_{i=1}^{n} |S_i| + \sum_{i=1}^{n-2} |S_i \bowtie_\theta S_{i+1}| + B_{Ts} \tag{4.2}$$

The network usage (N_c) is measured as the number of shuffled tuples. Remember from the example in figure 4.6 that for each micro-batch, a broadcast operation needs to send tuples first to the driver and later to all the executors. Consider a join running in N workers, broadcasting T tuples at each micro-batch and requiring M micro-batch to be complete(the limit is theoretically infinite since a streaming job is endless. However, we suppose an end for the computation). Thus, the total network usage (i.e., shuffled tuples) can be expressed as in 4.3

$$N_c = \sum_{i=1}^{M} T * (N + 1) \tag{4.3}$$

4.3.2 Workload

TPC-H is a decision support benchmark, consisting of ad-hoc queries in a database populated by synthetic data generated through the *dbgen* tool, that comes shipped with the benchmark. We use queries Q2, Q3, and Q5 from the specification 2.18 of the TPC-H benchmark . According to the document, the queries have the following description:

- Minimum Cost Supplier Query (**Q2**): This query finds which supplier should be selected to place an order for a given part in a given region;
- Shipping Priority Query (**Q3**):This query retrieves the 10 unshipped orders with the highest value;
- Local Supplier Volume Query (**Q5**): This query lists the revenue volume done through local suppliers;

Although the original queries contemplate other operation (e.g., filters and aggregations), we omit these operators from the computations since we are only interested in the performance of the join. Listings 4.1, 4.2, and 4.3 shows the executed queries for Q2, Q3 and Q5, respectively, without the other operations.

Listing 4.1: TPC-H Q2

```
SELECT C.customerId , L.orderKey
```

```
  FROM part P, partSupp PS, supplier S, nation N, region R
 WHERE P.partkey = PS.partkey
   AND PS.suppKey = S.suppKey
   AND S.nationKey = N.nationKey
   AND N.nationKey = R.retionKey
```

Listing 4.2: TPC-H Q3

```
SELECT C.customerId, L.orderKey
  FROM customer C, order O, lineitem L
 WHERE C.custKey = O.custKey
   AND O.OrderKey = L.OrderKey
```

Listing 4.3: TPC-H Q5

```
SELECT C.customerId, L.orderKey
  FROM customer C, order O, lineitem L, supplier S
 WHERE C.custKey = O.custKey
   AND O.OrderKey = L.OrderKey
   AND L.suppKey = S.suppKey
```

The benchmark uses the concept of scale-factor (SF) to define the size of input relations. For instance, an SF of 1 stands for 1GB and 10 for 10 GB. Table 4.1 resumes the size of each relation. The "Factor" column multiplied by SF determines the size. The two missing values in the table (nation and region) have a static size that does not change with SF. The executions times shown in the results were obtained by running each experiment 6 times and taking the mean value.

Table 4.1: Relation size

Relation	**Factor**
Customer	150 000
Order	1 500 000
LineItem	6 000 000
Supplier	10000
PartSupp	800 000
Part	200 000

4.3.3 Execution Environment

This sections describes the hardware running the VMs in Microsoft Azure and software stack within the VMs.

4.3.3.1 Hardware

Table 4.2 shows the hardware resource available for each VM in the cluster, wich are instances of the A4 v2 from the Microsoft Azure catalogue [2]. In total, 11 VMs were allocated with the same specification.

Table 4.2: Node description

Component	**Specification**
Processor	Intel® Xeon® E5-2673 v3 (Haswell) 4 Core (2.4-3.1 GHz)
Memory	8 GiB RAM
Storage	40 GiB

4.3.3.2 Software

The cluster uses Ubuntu 18.04.2 LTS as the operating system and Kubernetes 1.13.4 to orchestrate nodes, facilitating the communication and deployment of the software stack through the usage of containers. Since Kubernetes runs its services on the nodes, it was necessary to reduce resources for containers. Therefore, each VM can offer the 4 cores but only 7 GB of RAM for containers.

The application uses Spark 2.3.3, Kafka 2.2.2, and Hadoop 2.7.7 to store data in HDFS. All software runs through Docker images, using as a base image the official OpenJDK [3] 8 image.

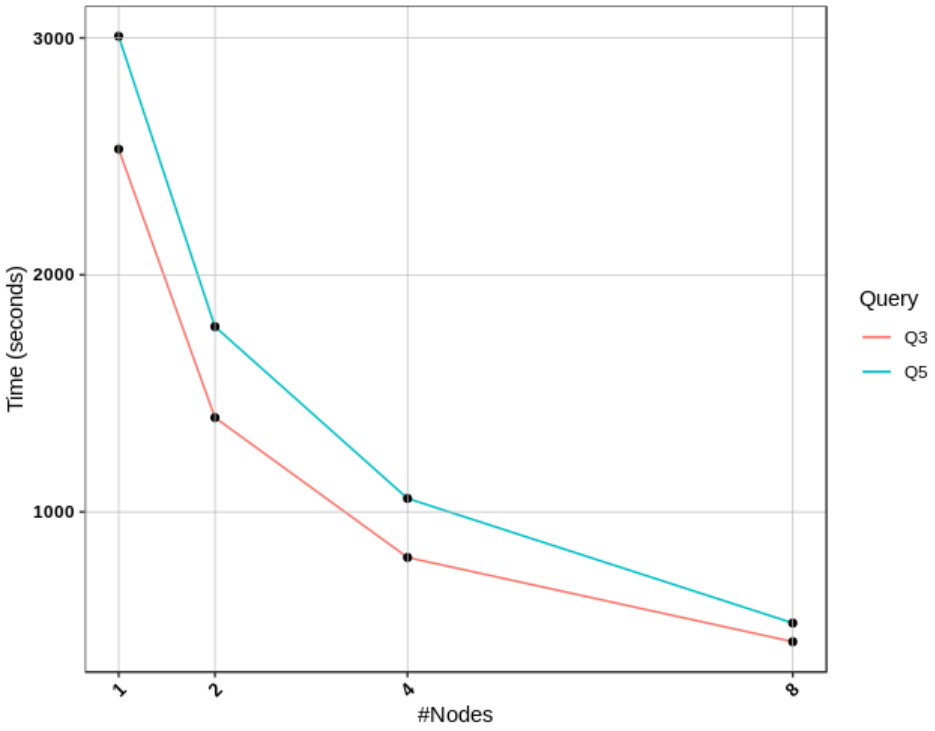
From the 11 VMs instances, one runs only the Kubernetes master service, creating and deploying containers. Kafka and the HDFS share a single VM also, and the Spark Driver also uses all resources from a single machine, leaving the other 8 VMs for workers process.

[2] https://azure.microsoft.com/en-us/pricing/details/virtual-machines/windows/
[3] https://hub.docker.com/$_{/}openjdk$

4.4 Experimental Evaluation

In Figure 4.7 we see the performance speed up for queries Q3 and Q5. As the number of workers doubles, the execution time reduces in an average of 44% for Q3 and 40% for Q5. The additional Supplier relation in Q5 implies in an increase of 23% of time in comparison with Q3. The difference is noticeable when running standalone since the enumeration of the Cartesian product by each task is higher. The increase in partitioning with 8 nodes turns the relation Suppliers (originally, 1000 tuples under SF 0.1) small for 32 tasks (on average, 32 tuples per task). However, there is still a difference in the order of seconds between the two queries, which is a consequence of the shuffle in Q5, as observed in figure 4.8.

Figure 4.7: Execution Time - Q3 and Q5



Based in Equation 4.3, figure 4.8 shows the number of shuffled tuples for Q3 and Q5. In a left-deep tree query, the performance is a consequence of the join selectivity, since the intermediate results will be shuffled. Both queries have a low join selectivity, but the LineItem relation size is big and still generates a significant intermediate result. The impact of this difference is observable in the difference between Q3 and Q5.

Figure 4.8: Network results - Q3 and Q5

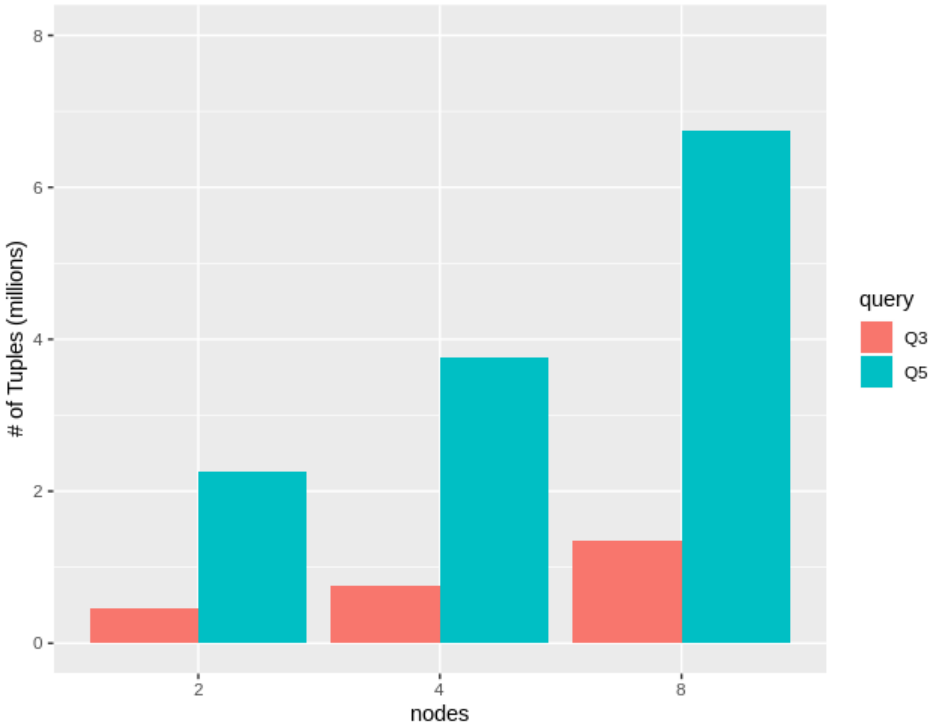


Table 4.3 presents the total number of tuples stored for relations and intermediate results. Besides these values, the total memory used for storage (equation 4.2) is composed by the broadcast variable, that variates during the micro-batch according to the join being computed. For Q3, the variable does not present a high cost, since it stores only the first relation *Customer* (the smallest relation from Q3 and Q5) and then the intermediate result. In Q5, the size of the storage is similar to Q3, but the intermediate results here are almost equal to the number of tuples from relations. The broadcast variable in Q5 has a considerable impact since the last intermediate result has almost the same size as the LineItem relations.

Table 4.3: Memory usage

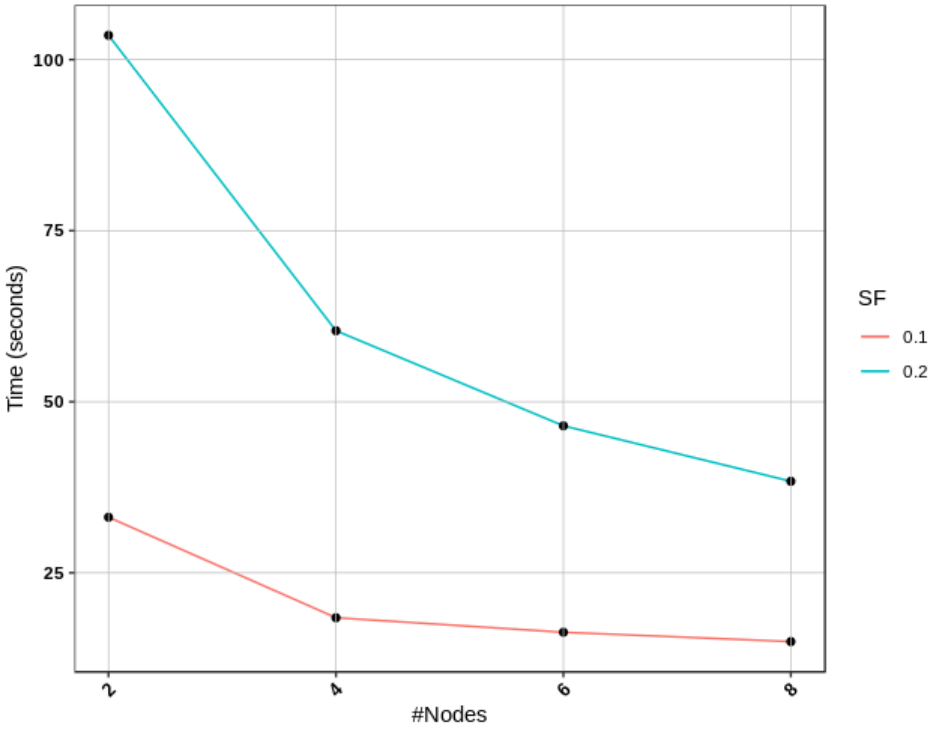
Application	**Relations**	**Intermediate Results**
Q3	765 572	150 000
Q5	766 572	750 572

The numbers obtained in the memory and networks results do not impact the computation for the small scale used in Q3 and Q5, besides the join selectivity for this query is also low. However, it is important to notice that using a left-deep tree when increasing the workload and also computing a join with *selectivity* close to the cartesian product the intermediate result would impact the network I/O and in the worst case may not even fit

in memory. Such a scenario may be computed by using a MJ tree (as shown in Figure 2.3b)

Figure 4.9 shows the performance with different workloads in the query Q2. The execution time is shorter than Q3 and Q5 even with the SF 0.2 since this query addresses the smallest relations from the benchmark. In the worst case, the difference in execution time is 70% for running with two machines. The smallest difference is with 8 machines, with 23%. For both workloads, the average speedup is 25%. However, the execution with SF 0.1 gains almost no speedup after 6 machines, since the improvement when adding 2 nodes is only 8%. This shows that for this workload, no more performance is gained with an increase of parallelism.

Figure 4.9: Execution Time - Q2



4.5 Observations

This chapter presented the model and the prototype in Spark Streaming, and also a methodology with strategies to validate the scalability of the proposed solutions by using queries in the left-deep tree model. The next chapter concludes this research with future works that were not addressed in this thesis.

5 CONCLUSION

This research presents a model for computing theta-joins in streaming with support for stateful operators in main memory. The proposed solution is implemented in Spark Streaming, using a broadcast join to produce results. We use a synthetic benchmark to evaluate the system performance in left-deep trees, storing intermediate results. The evaluation focuses on the scalability of the application and uses metrics based on cost models for resources (memory and network) and execution time. Results show an average performance gain of 40% when the resources are doubled. Although other join orders were not executed to compare with the left-deep tree, the discussion with obtained results show empirically that a left-deep tree is not always the best solution when the join produces significant intermediate results.

We leave a few suggestions for future work. A critical evaluation not addressed in this research is the comparison with state-of-the-art solutions in the record-at-a-time model. This evaluation requires a complex methodology as it implies the usage of different frameworks, thus requiring a methodical definition of how to obtain metrics and making a comparison. Another aspect necessary to test is the resilience against skewed workloads in queries with equality predicate. This research can be carried by comparing with Spark implementation of SHJ in the Structured Streaming API.

REFERENCES

AFRATI, F. N.; ULLMAN, J. D. Optimizing joins in a map-reduce environment. In: ACM. **Proceedings of the 13th International Conference on Extending Database Technology**. [S.l.], 2010. p. 99–110.

ANANTHANARAYANAN, R. et al. Photon: Fault-tolerant and scalable joining of continuous data streams. In: ACM. **Proceedings of the 2013 ACM SIGMOD international conference on management of data**. [S.l.], 2013. p. 577–588.

ANJOS, J. C. S. dos et al. Enabling Strategies for Big Data Analytics in Hybrid Infrastructures. In: . [S.l.]: IEEE Computer Society, 2018. (HPCS - International Conference on High Performance Computing and Simulation), p. 869–876. ISBN 978-1-5386-7878-7.

APACHE Storm. 2019. <http://storm.apache.org/>. Accessed in: 2019-07-04.

ARMBRUST, M. et al. Spark sql: Relational data processing in spark. In: ACM. **Proceedings of the 2015 ACM SIGMOD international conference on management of data**. [S.l.], 2015. p. 1383–1394.

BEAME, P.; KOUTRIS, P.; SUCIU, D. Skew in parallel query processing. In: ACM. **Proceedings of the 33rd ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems**. [S.l.], 2014. p. 212–223.

BLANAS, S. et al. A comparison of join algorithms for log processing in mapreduce. In: ACM. **Proceedings of the 2010 ACM SIGMOD International Conference on Management of data**. [S.l.], 2010. p. 975–986.

BORDIN, M. V. A benchmark suite for distributed stream processing systems. 2017.

CHU, S.; BALAZINSKA, M.; SUCIU, D. From theory to practice: Efficient join query evaluation in a parallel database system. In: ACM. **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. [S.l.], 2015. p. 63–78.

DAS, A.; GEHRKE, J.; RIEDEWALD, M. Approximate join processing over data streams. In: ACM. **Proceedings of the 2003 ACM SIGMOD international conference on Management of data**. [S.l.], 2003. p. 40–51.

DEAN, J.; GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. **Communications of the ACM**, ACM, v. 51, n. 1, p. 107–113, 2008.

ELMASRI, R.; NAVATHE, S. **Fundamentals of database systems**. [S.l.]: Addison-Wesley Publishing Company, 2010.

ELSEIDY, M. et al. Scalable and adaptive online joins. **Proceedings of the VLDB Endowment**, VLDB Endowment, v. 7, n. 6, p. 441–452, 2014.

FANG, J. et al. Cost-effective stream join algorithm on cloud system. In: ACM. **Proceedings of the 25th ACM International on Conference on Information and Knowledge Management**. [S.l.], 2016. p. 1773–1782.

FERNANDEZ, R. C. et al. Integrating scale out and fault tolerance in stream processing using operator state management. In: ACM. **Proceedings of the 2013 ACM SIGMOD international conference on Management of data**. [S.l.], 2013. p. 725–736.

GULISANO, V. et al. Performance modeling of stream joins. In: ACM. **Proceedings of the 11th ACM International Conference on Distributed and Event-based Systems**. [S.l.], 2017. p. 191–202.

HOFFMANN, M.; MICHEL, S. Scaling out continuous multi-way theta-joins. In: ACM. **Proceedings of the 4th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond**. [S.l.], 2017. p. 8.

JUNIOR, P. R. R. d. S. A data driven dispatcher for big data applications in heterogeneous systems. 2018.

KOLBERG, W. et al. Mrsg–a mapreduce simulator over simgrid. **Parallel Computing**, Elsevier, v. 39, n. 4-5, p. 233–244, 2013.

LIN, Q. et al. Scalable distributed stream join processing. In: ACM. **Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data**. [S.l.], 2015. p. 811–825.

MATTEUSSI, K. J. et al. Understanding and minimizing disk contention effects for data-intensive processing in virtualized systems. In: IEEE. **2018 International Conference on High Performance Computing & Simulation (HPCS)**. [S.l.], 2018. p. 901–908.

MATTEUSSI, K. J. et al. Analysis and performance evaluation of deep learning on big data. In: **2019 IEEE Symposium on Computers and Communications (ISCC) (IEEE ISCC 2019)**. Barcelona, Spain: [s.n.], 2019.

OKCAN, A.; RIEDEWALD, M. Processing theta-joins using mapreduce. In: ACM. **Proceedings of the 2011 ACM SIGMOD International Conference on Management of data**. [S.l.], 2011. p. 949–960.

SOUZA, P. R. de et al. Aten: A dispatcher for big data applications in heterogeneous systems. In: IEEE. **2018 International Conference on High Performance Computing & Simulation (HPCS)**. [S.l.], 2018. p. 585–592.

TEUBNER, J.; MUELLER, R. How soccer players would do stream joins. In: ACM. **Proceedings of the 2011 ACM SIGMOD International Conference on Management of data**. [S.l.], 2011. p. 625–636.

THE TPC-H Benchmark. 2019. <http://www.tpc.org/tpch/>. Accessed in: 2019-07-04.

TOSHNIWAL, A. et al. Storm@ twitter. In: ACM. **Proceedings of the 2014 ACM SIGMOD international conference on Management of data**. [S.l.], 2014. p. 147–156.

URHAN, T.; FRANKLIN, M. J. Xjoin: A reactively-scheduled pipelined join operatorỳ. **Bulletin of the Technical Committee on**, p. 27, 2000.

VIGLAS, S. D.; NAUGHTON, J. F.; BURGER, J. Maximizing the output rate of multi-way join queries over streaming information sources. In: VLDB ENDOWMENT. **Proceedings of the 29th international conference on Very large data bases-Volume 29**. [S.l.], 2003. p. 285–296.

WILSCHUT, A. N.; APERS, P. M. Dataflow query execution in a parallel main-memory environment. **Distributed and Parallel Databases**, Springer, v. 1, n. 1, p. 103–128, 1993.

ZAHARIA, M. et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: USENIX ASSOCIATION. **Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation**. [S.l.], 2012. p. 2–2.

ZAHARIA, M. et al. Discretized streams: Fault-tolerant streaming computation at scale. In: ACM. **Proceedings of the twenty-fourth ACM symposium on operating systems principles**. [S.l.], 2013. p. 423–438.

ZHU, G. et al. Automj: Towards efficient multi-way join query on distributed data-parallel platform. In: IEEE. **2017 IEEE 23rd International Conference on Parallel and Distributed Systems (ICPADS)**. [S.l.], 2017. p. 161–169.

ZHUANG, Z. et al. Effective multi-stream joining in apache samza framework. In: IEEE. **2016 IEEE International Congress on Big Data (BigData Congress)**. [S.l.], 2016. p. 267–274.